Entwicklung von Bildverarbeitungssoftware mit ToPAs

— Toolbox for Processing and Analysing ImageS —

Institut für Informatik
Mustererkennung und Bioinformatik

Angewandte Bildverarbeitung, SS 2007

Überblick

Programmieren mit C++

- zum Erstellen eines C++-Programms sind grundsätzlich 3 Schritte notwendig:
  1. Quelldatei(en) anlegen (mit einem Editor, z.B. *emacs*)
  2. Quelldatei(en) **compilieren** ($\rightarrow$ Objektfiles *.o)
  3. Objektfiles zum ausführbaren Programm **linken**
- Compilieren und Linken werden manchmal auch in einem Schritt zusammengefasst
- C bzw. C++-Compiler/Linker unter Linux/Unix: **gcc** bzw. **g++**

1. Quelldateien

- enthalten die Sourcen des Programms:
  - Klassendeklarationen und -implementierungen
  - Funktionendeklarationen und -implemetierungen
  - Variablendeklarationen
  - die *main*-Funktion und diverses andere

```
#include <iostream>   // Einbindung externer Funktionen

class HelloWorld {    // Klassendeklaration
   public:
     void sayHello();
};

int main(int argc, char **argv) {
  HelloWorld helloWorldObject;
  helloWorldObject.sayHello();
};

void HelloWorld::sayHello() {
   std::cout << "Hello World!!!" << std::endl;
};
```

2. Compilieren

```
pioneer->bimoelle[pts/2]: g++ -c helloWorld.cc
```

- der Compiler ”übersetzt” die Quelldateien in Objektcode: jede Klasse / Funktion / Variable / . . . wird symbolisch codiert
- erkannte Fehler:
  - Syntaxfehler (Tippfehler, fehlendes Semikolon, ...)
  - nicht bzw. doppelt deklarierte Variablen / Funktionen / . . .
- Resultat: für jede Quelldatei wird ein Objektfile generiert

$$\textit{datei.cc} \quad \Rightarrow \quad \textit{datei.o}$$

- Optionen für den g++:

  -c : nur Compilieren, nicht Linken
  -I : Pfad zu Dateien, die inkludiert werden

3. Linken


```
pioneer->bimoelle[pts/2]: g++ helloWorld.o
```

- der Linker bindet die verschiedenen Objektdateien (und ggf. externe Libraries) zu einem Executable zusammen
- erkannte Fehler:
  - undefinierte Symbole, d.h. beispielsweise Funktionen, die deklariert, aber nicht implementiert sind
- Resultat: ausführbares Programm *"a.out"*
- Optionen für den g++:

  -o : Name des Programms / Ausgabedatei
  -L : Pfad für Libraries, die zugelinkt werden sollen
  -l : Name der entsprechenden Library
- Beispiel: Einbinden der Mathematik-Bibliothek *"libm.a"*

```
pioneer->bimoelle: g++ -L/usr/lib/ -lm helloWorld.o
```

Organisation des Source-Codes

- grundsätzlich könnte der gesamte Code in **eine** Datei geschrieben werden

  $\Rightarrow$ die vermutlich einfachste Art, Chaos zu erzeugen !!!

  $\Rightarrow$ Wiederverwendung / Modularisierung schwierig !!!
- Trennung von Deklarationen und Implementierungen

  $\Rightarrow$ erhöht die Übersichtlichkeit

  $\Rightarrow$ vereinfacht Anlegen und Verwendung von Libraries
- Deklarationen werden in *Header-Files* geschrieben (Endung .h)
- Implementierungen finden sich in den entsprechenden *Source-Files* (Endung .c oder .cc)

Organisation des Source-Codes II

- Header-Files werden #include eingebunden:

```
#include "helloWorldClass.h"

int main(int argc, char **argv)
{
  HelloWorld helloWorldObject;
  helloWorldObject.sayHello();
};
```

- zu unterscheiden:

  #include <{*file.h*}> :
  Header aus den Standard-Includeverzeichnissen

  #include "file.h" :
  ein Header, der sich **nicht** *dort befindet*

- zu jedem Header müssen die zugehörigen Objektfiles beim Linken eingebunden werden

Bibliotheken

- Library: Sammlung von Objektfiles
- Libraries stellen oft benötigte Funktionen / Klassen zur Verfügung:
  - **libm.a** - mathematische Funktionen und Konstanten
  - **libX11.a** - Unterstützung grafischer Ausgaben
  - **libimageBase.a** - (unser) Bilddatentypen, Ein-/Ausgabe
  - ...
- zu einer Library gehören
  - der oder die Header mit den Deklarationen
  - das eigentliche Library-File (Endung .a oder .so)

Was sind Templates und wozu sind sie da?

Wikipedia-Definition

Templates oder Schablonen sind ”Programmgerüste”, die eine Datentyp-unabhängige Programmierung ermöglichen. Templates unterstützen generische Programmierung.

- Generizität
  - Klassen und Funktionen, die von *Typen* abhängen
  - Funktionen, denen Funktionen übergeben werden, z.B. für Vergleiche
    $\Rightarrow$ *Übersetzungs- / Parameter-Polymorphismus*
  - im Gegensatz dazu:
    Laufzeit-Polymorphismus bei virtuellen Funktionen
- Geschwindigkeit
  - wird komplett während der Compilierung aufgelöst, so als ob man die Klassen von Hand geschrieben hätte

Wozu Templates?

- . . . beispielsweise Matrizen
- eine Matrix definiert sich. . .
  - über ihre Dimensionen
  - darüber, was man mit ihr anstellen kann
- die eigentlichen Einträge sind erstmal zweitrangig

```
template <class matrixelement> class matrix
{
   public:
     matrixelement getElem(int x, int y) {
        return m_rows[y][x]; }
     void setElem(int x, int y, matrixelement value) {
        m_rows[y][x] = value; }

   private:
     matrixelement **m_rows;
};
```

Wozu Templates?

- eine Template-Definition ergibt viele Klassen

```
matrix<int> mint;
matrix<double> mdouble;

class complex;
matrix<complex> mcomplex;

int value = 200;
mint.setElem(10,20, value);
```

- Aber Achtung!!
  Manchmal muss in der Implementierung auch unterschieden werden!!!

Eigenschaften

- Typäquivalenz: Benutzung wie normale Typen
- Typprüfung und Instanziierung:
  Bei der Template-Instanziierung werden (mindestens) alle benötigten Methoden der Template-Klasse instanziiert (*Spezialisierungen* des Templates)
- bei der Instanziierung werden auch Typprüfungen durchgeführt:

```
template<class matrixelement>
void matrix<matrixelement>::printElem(int x, int y)
{
  getElem(x,y).print();
}
```

⇒ Typprüfung ergibt, dass der Typ *matrixelement* eine methode *print()* haben muss!

Definition von Templates

- *template*-Schlüsselwort + Parameter:

```
template<class elem, int layer = 1>
class image
{
  ...
}
```

- Parameter können sein:
  - *class* (oder *typename*) - für allgemeine Typen
  - einfache Typen - wie int,float
  - Adressen/Zeiger von Objekten oder Funktionen
  - Funktionsnamen
- Default-Parameter sind auch möglich

Gebrauch von Templates

- impliziter Gebrauch:

  wenn klar ist, welches Template benutzt wird, braucht man das Template nicht kenntlich zu machen

```
template<class T> int g(matrix<T> &m) { ... }

matrix<int> m;
g(m);
```

- expliziter Gebrauch:

```
tempate<class T> T erzeugeCopie(T) { ... }

int zahl = erzeuge(20.0);
erzeuge<int> oder erzeuge<float>?

deshalb:
int zahl = erzeuge<int>(20.0);
```

Gebrauch von Templates

- Überladen: wie bei anderen Funktionen auch

  Aber: es können leicht Mehrdeutigkeiten auftreten
  $\Rightarrow$ durch expliziten Gebrauch lösen
- Typedefs und Typenamen:

  Typedefs sind Typ-Synonyme, damit leichter lesbar

```
template<class imageElement, int layers>
class image
{
...
};

typedef image<unsigned char,1> grayImage;
typedef image<unsigned char,3> colorImage;
```

Instanziierung und Organisation

- Wie stellt der Compiler sicher, dass die entsprechende Spezialisierug an der richtigen Stelle zur Verfügung steht, und nicht doppelt vorkommt?
- Borland Model:
  Compiler erzeugt Template Instanziierungen, wo immer sie benötigt werden. Der Linker erkennt dabei doppelte Instanziierungen und fasst sie zusammen.
  - Vorteile: Linker braucht nur .o files zu betrachten
  - Nachteil: hohe Compile-Time weil Template-Code mehrfach übersetzt wird
  - Folge: beim Compilieren muss Template-DEFINITION verfügbar sein

    $\Rightarrow$ üblicherweise wird die Template-Definition in den Header geschrieben

Instanziierung und Organisation

- Cfront Model (AT&T Compiler):
  - alle beim Übersetzen benötigten Template-Instanziierungen werden zentral gesammelt
  - ein Link-Wrapper übersetzt vorm Linken alle notwendigen Dateien dort
  - Vorteil: man braucht keinen speziellen Linker, optimale compile-time
  - Nachteil: stark erhöhte Komplexität bei der Codeverwaltung
  - Folge: Trennung von Template-Definition und Deklaration in separate Dateien
- g++ benutzt das Borland Model, kann aber auch mit zentralen Verzeichnissen arbeiten oder die Instanziierung dem Benutzer überlassen

Instanziierung und Organisation

- in ToPAs: *explizite* Template-Instanziierung
- g++-Option *-fno-implicit-templates*, damit kein Code für Templates automatisch erzeugt wird
- Templates werden explizit instanziiert in einer extra Datei:

```
template class matrix<int>;
template void f<int>(matrix<int> &);
....
```

Instanziierung und Organisation

- Organisation des Quellcodes:
  - Deklaration im Header (.h)
  - normale Methoden-Definition im .cc-File
  - Template-Definitionen im .tcc-file
  - used_templates.cc: zu instanziierende Templates
- Vorteile:
  - alle Instanziierungen explizit (d.h. wenig Code-Duplizierung)
  - einfache Generierung von STL Template-Instanziierungen

Warum noch eine Bibliothek für die Bildverarbeitung?

- ”damals” (2001) gab es noch nicht ”die” Referenzbibliothek
- ToPAs ist zugeschnitten auf unsere Forschung:
  - relativ wenig Low-Level-Algorithmik,
    viel Bewegungsanalyse / Bildregistrierung
  - Bildtransformationen
  - Umgang mit undefinierten Bildpixeln
  - hochgradig templatisiert (”float-Bilder mit 17 Layern. . . ”)
- ToPAs ist zugeschnitten auf unsere Hardware:
  - Unterstützung der AG-Grabber
  - direkte Ansteuerung der Kameras

Warum noch eine Bibliothek für die Bildverarbeitung? - II

- ToPAs konkurriert nicht mit OpenCV & Co, sondern ergänzt eher
- ToPAs...
  - hat keine integrierte Visualisierung
  - ist nicht auf Highspeed getrimmt
  - läuft nur unter Linux
- ToPAs ist klassisches Forschungssystem...
  - rudimentär dokumentiert
  - stabil, aber nicht fehlerfrei implementiert

$\Rightarrow$ ToPAs als Grundlage für Hardwarezugriffe, Algorithmik eher extern...

Einfaches Beispiel zur Motivation

- ein Grauwert-Bild einlesen und seine Dimensionen ausgeben:

```
#include <iostream>
#include "image/base/HBMImageDirect.h"

int main(int argc, char *argv[])
{
  HBMImageDirect<double, 1> *pgmBild;
  pgmBild= new HBMImageDirect<double, 1>;
  pgmBild->loadPGM(argv[1]);
  unsigned long breite= pgmBild->getWidth();
  unsigned long hoehe= pgmBild->getHeight();

  std::cout << "B x H = "
            << breite << " x " << hoehe << std::endl;
  return 0;
}
```

Was ist drin in ToPAs?

- die Klassenstruktur ist installiert unter:

```
/vol/agsoft/src/topas/.

        |-- CVS
        |-- bin
        |-- doc       Dokumentation
        |-- etc       globale Konfigurationsdateien
        |-- lib       dynamische Bibliotheken
        |-- share     sonstige Daten (hier unwichtig)
        |-- src       der Quellcode
```

- im Verzeichnis

```
/vol/export/angewandteBV_SS07/ToPAs/sample
```

befindet sich ein Beispiel als Startgerüst
$\Rightarrow$ kopieren, ausprobieren, ändern!!!

Was ist drin in ToPAs?


```
/vol/agsoft/src/topas/src/libs/.

  |-- activeVision      aktive Szenenexploration
  |-- features          Punktdetektion (Harris, SUSAN)
  |-- geometry          Punkte, Geraden, proj. Koord.
  |-- hardware          Kamera, Grabber, Stereokopf
  |-- image             Bildklassen
  |-- math              Matrizen, Graphen
  |-- misc              LEDA-Zusaetze
  |-- mosaic            Bildregistrierung, Mosaicing
  |-- motion            Bewegungsdetektion, Tracking
  |-- multiView         Multi-Mosaikbilder
  |-- registQuality     Registrierungsqualitaet
  |-- segmentation      Component Labeling, Watershed
  |-- templates         Template-Zusaetze
  |-- transformations   Bildtransformationen
  |-- utils             Zeitmessung, ...
  |-- viewer            Bildviewer
```

Für uns vorrangig wichtig...

- Basisbildklassen
- Hardwarezugriff
- ggf. Transformationen
- ggf. Punktdetektion

Bilder in ToPAs

- 3 wesentliche Basisklassen:
  - *HBMImageInterface*:
    grundlegende Funktionalität und Schnittstelle
  - ***HBMImageDirect***:
    Implementiert die Schnittstelle auf Basis der matrix-Klasse
    (je *layer* eine matrix)
  - *validPoints*:
    definiert eine Maske *valider* Bildpixel
    $\Rightarrow$ nur solche werden in einigen Funktionen berücksichtigt

Bilder in ToPAs

- Nutzung folgt allgemeiner Template-Verwendung:

```
#include "image/base/HBMImageDirect.h"

HBMImageDirect<double, 1> *pgmBild;
pgmBild= new HBMImageDirect<double, 1>;
pgmBild->loadPGM(argv[1]);

HBMImageDirect<unsigned char, 3> rgbBild;
```

Bilder in ToPAs

- Zugriff auf Elemente:

```
HBMImageDirect<double, 1> pgmBild;
unsigned long breite= pgmBild.getWidth();
unsigned long hoehe=  pgmBild.getHeight();

pixel<double,1> p= pgmBild.getPixel(x,y);
std::cout << p.layer[0] << std::endl;

bool b;
pixel<double,1> p= pgmBild.getPixelSavely(x,y,b);
if (b)
  std::cout << p.layer[0] << std::endl;

double d= pgmBild.getPixelInLayer(x,y,0);
```

- analog: set-Routinen

Bilder in ToPAs

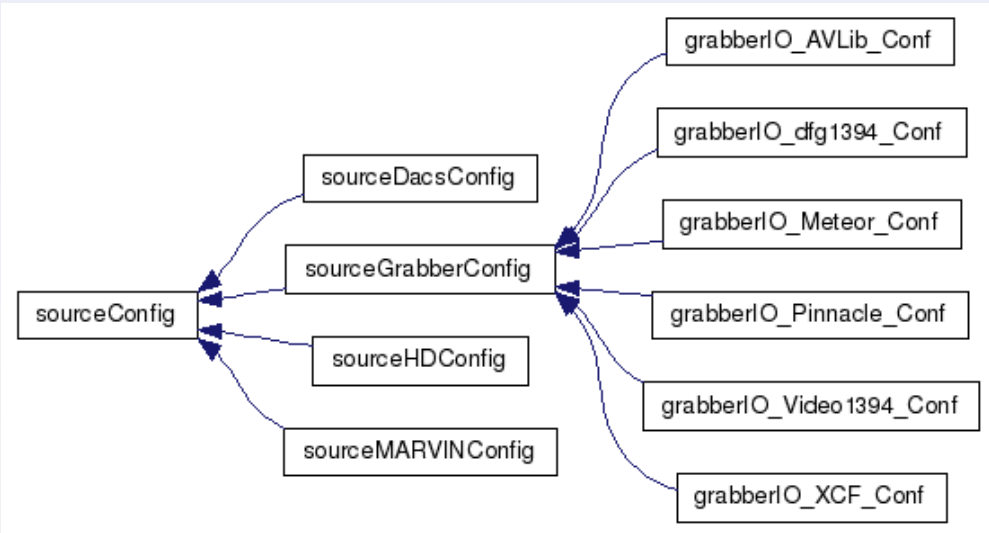
- wichtige Funktionen:

```
setSize(w,h);
setValid(x,y);  isValid(x,y);

downSample(&result, mask);

xDerivative(&result); yDerivative(&result);
calcGradMagnitude(&result);

savePGM(char *file);   loadPGM(char* file);
savePPM(char *file);   loadPPM(char* file);

convolve(&result, kernel);
sobel(&result);
```

- ausserdem: Addition/Subtraktion/Skalierung/...

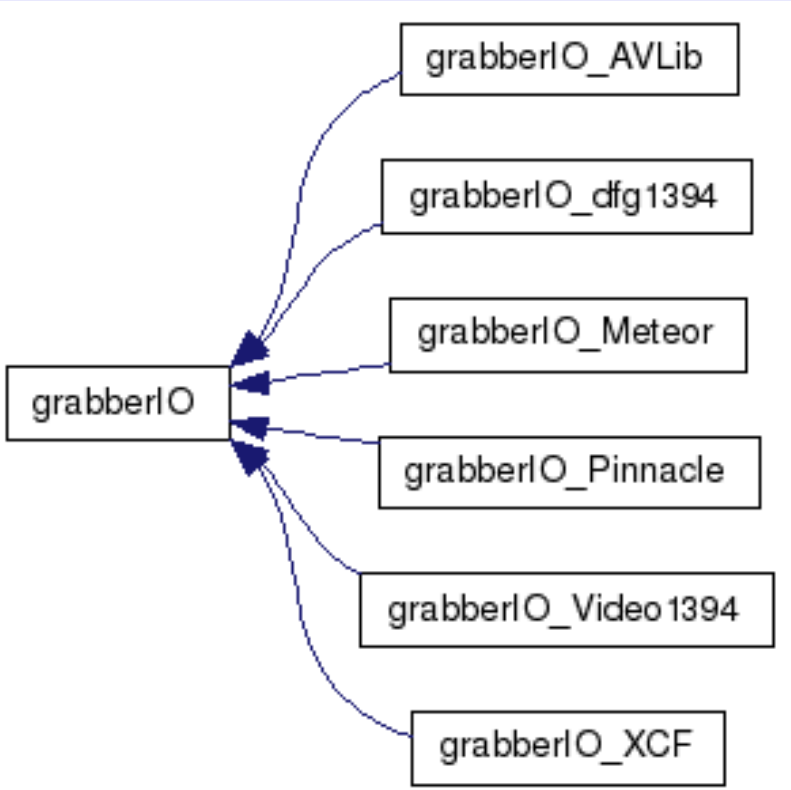
Bildaufnahme in ToPAs

- Bilder können...
  - von Platte gelesen werden
  - direkt vom Grabber geholt werden
- für jede Quelle gibt es ein
  - Kontroll-Objekt, das die Bilder liest
  - ein Konfigurationsobjekt, das Grösse und Format angibt



Bildaufnahme

- Datenquellen:



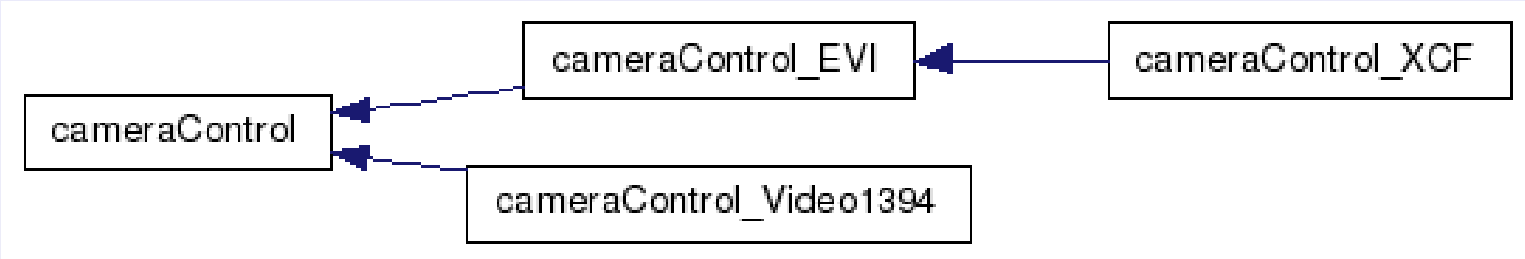
- jede Datenquelle muss initialisiert werden
- Wichtig: am Ende auch wieder freigeben!
  (→ eventuell Prozesse direkt abschiessen)

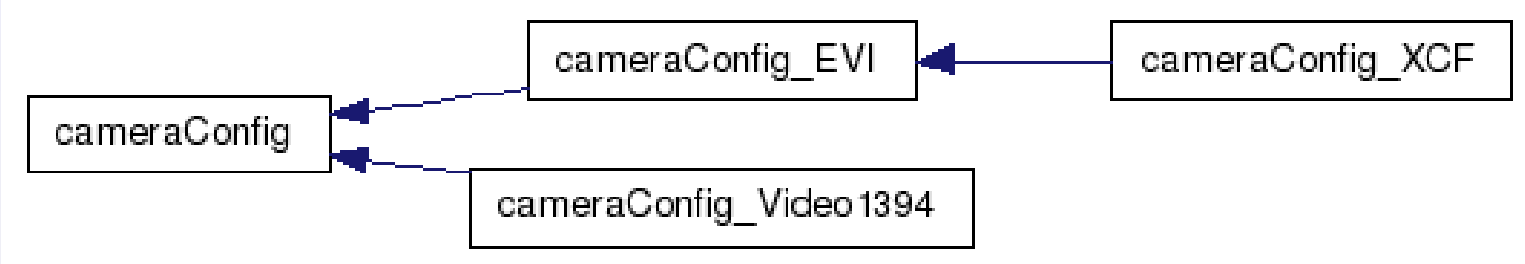
Bildaufnahme in ToPAs


```
// Grabber-Konfigurationsobjekt anlegen
grabberIO_dfg1394_Conf* conf=
                new grabberIO_dfg1394_Conf(0,1,_rgb);
grabberIO_dfg1394 grabDev(conf, true);

if (!grabDev.registerGrabber())
  std::cout << "Cannot register!" << std::endl;

HBMImageDirect<double, 4> testBild4;
if (!grabDev.getGrabberImage(testBild4))
  std::cout << "Cannot get image!" << std::endl;

conf->setInFormat(_gray); // YUV-Bilder lesen
HBMImageDirect<double, 1> testBild1;
grabDev.getGrabberImage(testBild1);

grabDev.unregisterGrabber();
```

Kameraansteuerung

- auch hier unterscheiden wir zwischen der Kamerakontrolle und ihrer Konfiguration:





- das Kontroll-Objekt steuert die Bewegungen
- das Konfigurations-Objekt spezifiziert welche

Kameraansteuerung II


```
// Kontroll- und Konfigurationsobjekte anlegen:
// 0 ist das Device, 1 die Bewegungsrichtung (rechts),
// 20 die Schrittweite und 3 die Default-Geschwindigkeit
cameraConfig_EVI eConf(0, 1, 20, 3);
cameraControl_EVI camCom(&eConf,true);

// immer zuerst: Device oeffnen
if ( !camCom->openCameraDevice () )
  std::cerr << "Kein Kamera-Zugriff!!!" << std::endl;

// eine Bewegung machen gemaess Default...
camCom->performNextStep();

// oder explizit
std::cout << "Move to the right..." << std::endl;
camCom->panRight(100,5);

// am Ende Device schliessen
camCom->closeCameraDevice();
```

Automatische Programmgenerierung: make

- Motivation:
  - Abhängigkeiten zwischen Source-/Headerfiles und Libraries
  - Änderungen in einzelnen Dateien bedingen neues Übersetzen / Linken auch in anderen Programmteilen
  - oft sind wiederholt dieselben g++-Aufrufe notwendig

    $\Rightarrow$ Automatisierung wünschenswert !!!
- Tool hierfür: *make*
  - Make dient der Vewaltung von Abhängigkeiten
  - ermöglicht automatische Neugenerierung aller von Änderungen betroffenen Programmteile
  - universell einsetzbar, z.B. auch für große Dokumente mit Latex

Automatische Programmgenerierung II

- *make* wird durch eine Datei konfiguriert → *Makefile*
- die Konfigurationsdatei enthält Regeln:

```
Ziel: Objekt1 Objekt2 ....
  auszufuehrende Kommandos
```

- *Ziel* kann eine Datei oder ein symbolischer Name sein
- die Objekte sind Dateien oder Ziele
- die Kommandos beinhalten die Compiler- / Linker-Aufrufe

Automatische Programmgenerierung - Beispiel-Makefile


```
all:    helloWorldProgram

clean:
        rm -f *.o helloWorldProgram

helloWorldProgram:      main.o helloWorldClass.o
        g++ -o $@ $^

%.o:    %.c
        g++ -c $^
```

- *all* bezeichnet Default-Ziel
- *clean* räumt auf
- make ermöglicht Verwendung von Makros:
  - `$@` = Ziel der aktuellen Regel
  - `$^` = Objekte der aktuellen Regel
  - % = Platzhalter für beliebige Namen (Suffixregeln)

Automatische Programmgenerierung

- beim Aufruf können Ziele als Parameter an *make* übergeben werden:

```
pioneer->bimoelle[pts/2]: make clean all

pioneer->bimoelle[pts/2]: make helloWorldClass.o
```

- Ausserdem:
  - Deklaration von Variablen und Konstanten
  - bedingte Anweisungen (if, else, . . .)
  - Einbettung von Shell-Kommandos
  - . . .

Code-Entwicklung mit ToPAs

- Einstieg: Beispiel unter

  `/vol/export/angewandteBV_SS07/ToPAs/sample`

- für eigenen Code am besten ”Makerules” erweitern:

```
##############################################################
# NAMES AND RELATED FILES AND PATHS ...
##############################################################

# Name des Zielprogramms
TARGET = beispielProgramm

# zugehoerige Source- und Object-Dateien
TARGET_SRC.linux-halle = beispielProgramm.cc newFile.cc
TARGET_OBJS.linux-halle= $(TARGET_SRC.linux-halle:.cc=.o)
```

Was sonst noch hilfreich sein kann. . .

- *evi_ctrl* –
  eine Software-Fernsteuerung für die Kamera
- *dfg1394_ctrl* –
  eine GUI zum Testen der Kameraeinstellungen
- *gscanbus* –
  ein Busmonitor für den Firewire-Bus
- *xv* –
  ein Bildbetrachter, der fast alle Formate kennt
- *convert* – Kommandozeilentool, das **alles** kann. . .
  - Bilder konvertieren
  - skalieren, croppen, rotieren
  - normalisieren, filtern
  - mosaicen
  - . . .