Programmentwicklung mit C++ (unter Unix/Linux)

- Erste Schritte
- Der gcc - Compiler & Linker
- Organisation des Source-Codes & Libraries
- Automatische Programmgenerierung: Make

Birgit Möller & Denis Williams
AG Bioinformatik & Mustererkennung
Institut für Informatik
Martin-Luther-Universität Halle-Wittenberg

Erste Schritte

- 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++**

Erste Schritte

1. Quelldateien:

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

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

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

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

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

Compiler

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

  datei.cc $\Rightarrow$ *datei.o*
- Optionen für den g++:

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

Linker

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[pts/2]: 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 dann eher schwierig !!!

- Daher: Trennung von Deklarationen und Implementierungen

  $\Rightarrow$ erhöht die Übersichtlichkeit

  $\Rightarrow$ vereinfacht das Anlegen und die 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

- Header-Files können mit #include eingebunden werden:

```
#include "helloWorldClass.h"

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

- zu unterscheiden:

  #include <*file.h*> :
  ein Header, der sich in den Standard-Includeverzeichnissen befindet

  #include ”*file.h*” :
  ein Header, der sich **nicht** in den Standard-Includeverzeichnissen befindet

- zu jedem Header, der benutzt wird, müssen die zugehörigen Objektfiles beim Linken eingebunden werden

Libraries

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

Automatische Programmgenerierung

Motivation:

- Abhängigkeiten zwischen Source- und Headerfiles sowie zu Libraries
- Änderungen in einzelnen Dateien bedingen neues Übersetzen und Linken auch in anderen Programmteilen
- oft sind wieder und wieder 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

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

```
Ziel: Objekt1 Objekt2 ....
   auszuführende Kommandos
```

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

Automatische Programmgenerierung

Ein Makefile für unser ”HelloWorld”-Programm:

```
all:    helloWorldProgram

clean:
        rm -f *.o helloWorldProgram

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

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

- *all* bezeichnet Default-Ziel, das defaultmäßig angesprungen wird
- *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
  - . . .
- bei Interesse: Details im Manual (”/home/moeller/manuals/make.ps.bz2”)