Mapra: C++ Teil 4

Felix Gruber, Sven Groß

IGPM, RWTH Aachen

9. Mai 2017

Was bisher geschah

- Debuggen mit `gdb`, `ddd` und `kdbg`
- Zusammengesetzte Datentypen mit `struct`
- `std::vector`

Wie es heute weiter geht

1. Makefiles
2. Funktionen-Templates
3. Ein-/Ausgabe über Dateien

Bedingtes Kompilieren mit Makefiles

- Programm mit vielen cpp/h-Dateien mit Quellcode, einige davon seit letztem Kompilieren geändert
- welche Objektdateien müssen neu erzeugt werden?
- Hilfe: sogenanntes **`Makefile`**, besteht aus **Regeln** der Form

  `Ziel: Abhaengigkeiten`
  `<Tab> Befehl`
- konkretes Beispiel:

  `sort.o: sort.cpp student.h`
  `<Tab> g++ -c sort.cpp`
- `make` <Ziel> führt entprechende Regel und alle davon abhängigen aus
- `make` ohne Argument: nimmt erste Regel im Makefile

student.h

student.cpp

```
#include<student.h>
```

sort.cpp

```
#include<student.h>

int main()
{

}
```

Makefiles (1)

```
# Inhalt der Datei "Makefile"

sort: sort.o unit.o student.o
    g++ sort.o unit.o student.o -o sort

sort.o: sort.cpp student.h
    g++ -c sort.cpp

student.o: student.cpp student.h
    g++ -c student.cpp

clean:
    rm -f *.o sort
```

Makefiles (2)

```
CXXFLAGS = -O0 -g              # Makefile-Variable

sort: sort.o unit.o student.o
    $(CXX) $(CXXFLAGS) sort.o unit.o student.o -o sort

sort.o: sort.cpp student.h
    $(CXX) $(CXXFLAGS) -c sort.cpp

student.o: student.cpp student.h
    $(CXX) $(CXXFLAGS) -c student.cpp

.PHONY: clean
clean:
    rm -f *.o sort
```

- Makefile-Variablen benutzen, z. B. für Compiler-Optionen
- vordefinierte Variablen, z. B. `CXX = g++` (Standard-Compiler)
- Kommentarzeilen beginnen mit `#`
- `.PHONY`-Regel für Regeln, die keine Dateien erzeugen

Makefiles (3)

```
CXXFLAGS = -O0 -g              # Makefile-Variable

sort: sort.o unit.o student.o
    $(CXX) $(CXXFLAGS) $^ -o $@

sort.o: sort.cpp student.h
student.o: student.cpp student.h

%.o: %.cpp
    $(CXX) $(CXXFLAGS) -c $<

.PHONY: clean
clean:
    rm -f *.o sort
```

- Pattern-Rules mit `%`
- Pattern-Variablen: `$@`, `$<`, `$^`
- *Bem.:* Für große Projekte bieten sich Build-Systeme wie die GNU autotools oder CMake an, um Makefiles zu erzeugen.

Zum Weiterlesen

- kurze Übersicht über make
  `https://en.wikibooks.org/wiki/Make`
- ausführliche make-Dokumentation
  `http://www.gnu.org/software/make/manual/`

Funktionen-Templates

```
int Produkt(int a, int b)
{
  int Ergebnis = a * b;
  return Ergebnis;
}
```

...für jeden Datentyp eine extra `Produkt`-Funktion schreiben?
Nicht nötig dank Templates! ☺

```
template<typename ValueT>
ValueT Produkt(ValueT a, ValueT b)
{
  ValueT Ergebnis = a * b;
  return Ergebnis;
}
```

- **Template-Parameter** `ValueT` stellvertretend für beliebigen Datentyp
- konkrete Festlegung des Typs zur Compile-Zeit

Funktionen-Templates

```
int i = 3, j = 5;
double x = 1.4, y = 2.1;

Produkt(i, j);       // ruft Produkt<int> auf
Produkt(x, y);       // ruft Produkt<double> auf

Produkt(x, i);       // Typ nicht eindeutig: Compiler-Fehler
Produkt<int>(x, i); // x wird auf int gecastet
```

- Compiler erzeugt automatisch Code-Variante für jeden Template-Aufruf

$\rightsquigarrow$ Unterschied zu gew. Funktionen: Deklaration + Definition in selber Datei!

- auch mehrere Template-Parameter möglich:

```
template<typename MatT, typename VecT>
void Solve(const MatT& A, VecT& x, const VecT& b);
```

- es gibt außerdem noch Klassen-Templates (z. B. `std::vector`), dazu später mehr

File Streams: Ein-/ Ausgabe über Dateien

- zuerst header-Datei für *file streams* einbinden

```
#include <fstream>
```

- Öffnen des *file streams* zur Ein- bzw. Ausgabe

```
std::ifstream ifs;
ifs.open("mydata.txt");

std::ofstream ofs("results.dat");
```

- gewohnte Verwendung von Ein- und Ausgabeoperator
(wie mit `std::cin` und `std::cout`)

```
ifs >> a;
ofs << "Das␣Ergebnis␣lautet␣" << f(x) << ".\n";
```

- Schließen des *file streams* nach Verwendung

```
ifs.close();    ofs.close();
```

(passiert automatisch, wenn `ifs` und `ofs` out of scope gehen)

Einlesen

- Eingabeoperator `>>` erwartet, dass Daten durch Whitespace getrennt sind (Leerzeichen, Tabulator, Zeilenumbruch)
- Abfrage des Stream-Status:
  - `ifs.good()` : Stream bereit zum Lesen
  - `ifs.eof()` : Ende der Datei (*end of file*) erreicht
  - `ifs.fail()` : letzte Leseoperation nicht erfolgreich
  - `ifs.clear()`: setzt Status zurück, ermöglicht weiteres Lesen

⚠ Leseoperation `>>` wird nur bei `good`-Status durchgeführt

Beispiel:

```
std::ifstream ifs("input.txt");
double d; int i; std::string s;

ifs >> d >> i;    // good
ifs >> i;         // fail
ifs.clear();      // good
ifs >> s;         // good
ifs >> s;         // fail eof
```

Datei `input.txt`:

```
17.5
4711 Hallo
```