Einführung in die Programmierung mit C++

Strukturierung des Quellcodes

Uwe Naumann

Informatik 12:
Software and Tools for Computational Engineering (STCE)

RWTH Aachen

Inhalt

Eine Quelldatei

- Deklaration vs. Definition von Funktionen
- Übersetzung mit `make`

Mehrere Quelldateien

- Quellcodestruktur und statische Bibliotheken
- Übersetzung mit `make`

Verzeichnisstruktur

Newton

- Mehrere Quelldateien
- Statische Bibliothek
- Verzeichnisstruktur

Inhalt

Deklaration von Funktionen

vs. deren Definition

- Deklaration der Schnittstelle is Mindestanforderung für Verwendung innerhalb nachfolgender Funktionen; sonst generiert Compiler einen Fehler
- Definition der gesamten Funktion inkl. Schnittstelle und Funktionskörper ist Vorraussetzung für die Generierung eines ausführbaren Programms; sonst generiert Binder (*Linker*) einen Fehler.

Beispiel

```
1 int f(); // (Vorwärts−)Deklaration / Signatur
2
3 int main() { cout << f() << endl; }
4
5 int f() { return 42; } // Definition / Implementierung
```

Deklaration vs. Definition

Fallstudie


```
1  #include<iostream>
2
3  void f(char* s) { std::cout << "Hi " << s << "!" << std::endl; }
4
5  int main(int c, char* v[]) {
6    if (c!=2) {
7      std::cerr << "1 Argument bitte!" << std::endl; return -1;
8    }
9    f(v[1]);
10   return 0;
11 }
```

OK!

Fallstudie

Fehlende Deklaration


```
1  #include<iostream>
2
3
4
5  int main(int c, char* v[]) {
6    if (c!=2) {
7      std::cerr << "1 Argument bitte!" << std::endl; return -1;
8    }
9    f(v[1]);
10   return 0;
11 }
12
13 void f(char* s) { std::cout << "Hi " << s << "!" << std::endl; }
```

Fehler: f nicht deklariert!

```
1  #include<iostream>
2
3  void f(char*);
4
5  int main(int c, char* v[]) {
6    if (c!=2) {
7      std::cerr << "1 Argument bitte!" << std::endl; return -1;
8    }
9    f(v[1]);
10   return 0;
11 }
12
13 void f(char* s) { std::cout << "Hi " << s << "!" << std::endl; }
```

OK!

Übersetzung mit make

Makefile v1


```
1 main.exe : main.cpp
2         g++ -Wall -Wextra -pedantic -O3 main.cpp -o main.exe
3
4 clean:
5         rm -f main.exe
6
7 .PHONY: clean
```

- Regel `main.exe : main.cpp` definiert "echtes" Ziel `main.exe` in Abhängigkeit von `main.cpp` und dessen Generierung mittels `g++` ...
- "unechtes" Ziel `clean` entfernt alle generierten Dateien (hier nur `main.exe`)
- `.PHONY` markiert "unechte" Ziele und garantiert Korrektheit der Abhängigkeiten auch bei Existenz von Dateien gleichen Namens

Übersetzung mit make

Makefile v2


```
1  COMPILER=g++
2  COMPILER_FLAGS=-Wall -Wextra -pedantic -O3
3
4  main.exe : main.cpp
5          $(COMPILER) $(COMPILER_FLAGS) $< -o$@
6
7  clean:
8          rm -f main.exe
9
10 .PHONY: clean
```

- Definition von Variablen COMPILER und COMPILER_FLAGS
- Zugriff auf Inhalt von Variable VAR mittels $VAR
- spezielle Variablen innerhalb von Regeln
  - $< ist erster Eintrag der rechten Seite einer Regel
  - $@ ist Eintrag auf der linken Seite einer Regel

Inhalt

- Definition von Funktionen, z.B. `f`, kann in separater Datei, z.B. `f.cpp`, geschehen. Deren Übersetzung mittels `g++ -c f.cpp` resultiert in einer Objektdatei `f.o`, die später durch den Binder mit anderen Objektdateien, die `f` verwenden, verbunden werden kann.
- Die (Vorwärts-)Deklaration von `f` in den sie verwendenden Dateien ist weiterhin notwendig.
- Alternativ kann die Schnittstelle in einer separaten Schnittstellendatei (auch *header*-Datei), z.B. `f.hpp` deklariert und mittels **#include** "f.hpp" durch den Präprozessor in `f` verwendende Dateien eingefügt werden. Die Hochkommenotation lässt den Präprozessor zuerst im aktuellen Verzeichnis nach `f.hpp` zu suchen.
- Mehrere Objektdateien lassen sich mithilfe des Archivierers `ar` in eine statische Bibliothek zusammenfassen, welche durch den Binder in Programme eingebunden werden können.
- Eine sinnvolle Verzeichnisstruktur verbessert Übersichtlichkeit des Quellcodes

- Deklaration in f.hpp

```
1 void f(char*);
```

- Definition in f.cpp

```
1 #include<iostream>
2
3 void f(char* s) { std::cout << "Hi " << s << "!" << std::endl; }
```

- Anwendung in main.cpp

```
1  #include "f.hpp"
2  #include<iostream>
3
4  int main(int c, char* v[]) {
5    if (c!=2) {
6      std::cerr << "1 Argument bitte!" << std::endl; return -1;
7    }
8    f(v[1]);
9    return 0;
10 }
```

Übersetzung mit make

Makefile v1

```
1  main.exe : main.cpp f.hpp f.o
2          g++ -Wall -Wextra -pedantic -O3 main.cpp f.o -o main.exe
3
4  f.o : f.cpp
5          g++ -c -Wall -Wextra -pedantic f.cpp
6
7  clean:
8          rm -f f.o main.exe
9
10 .PHONY: clean
```

- main.exe hängt auch von f.o und f.hpp ab
- f.o hängt von f.cpp ab
- Modifikation ausgewählter Dateien resultiert in spezifischen Übersetzungsprozessen

Übersetzung mit make

Makefile v2 (statische Bibliothek / Archiv)

Objektdateien (*.o) können in Bibliotheken zusammengefasst werden, z.B.

```
1  main.exe : main.cpp f.hpp libf.a
2          g++ -Wall -Wextra -pedantic -O3 main.cpp -o main.exe -L. -lf
3
4  libf.a : f.o
5          ar -r libf.a f.o
6
7  f.o : f.cpp
8          g++ -c -Wall -Wextra -pedantic f.cpp
9
10 clean:
11         rm -f libf.a f.o main.exe
12
13 .PHONY: clean
```

- Archivierer (ar -r für *replace*) generiert statische Bibliothek
- Binder (*linker*) koppelt Anwendung mit Bibliothek(en)
- Linuxbefehl (*nm libf.a*) listet den Inhalt der Bibliothek

Übersetzung mit make

Makefile v3 (statische Bibliothek / Archiv)

Variablen verbessern die Wart- und Erweiterbarkeit von Makefiles.

```
1  COMPILER=g++
2  ARCHIVER=ar
3  COMPILER_FLAGS=-Wall -Wextra -pedantic -O3
4
5  main.exe : main.o libf.a
6          $(COMPILER) $< -o$@ -L. -lf
7
8  libf.a : f.o
9          $(ARCHIVER) -r $@ $<
10
11 %.o : %.cpp
12         $(COMPILER) -c $(COMPILER_FLAGS) $< -o$@
13
14 main.o : f.hpp
15
16 clean:
17         rm -f *.o *.exe *.a
18
19 .PHONY: clean
```

Inhalt

Verzeichnisstruktur

Fallstudie


```
libf/
    inc/
        f.hpp
    src/
        f.cpp
        Makefile
    libf.a
    Makefile.inc
    Makefile
```

- Deklarationen in libf/inc/; Definitionen in libf/src/; libf.a in libf/src/;
- Makefile.inc definiert Variablen zur gemeinsamen Verwendung innerhalb verschiedender Makefiles
- Linuxbefehl tar −czvf libf.tgz libf erzeugt komprimiertes Archiv (z.B. zur Publikation als *download*)

- libf/Makefile.inc

```
1 COMPILER=g++
2 ARCHIVER=ar
3 COMPILER_FLAGS=-Wall -Wextra -pedantic -O3
```

- libf/Makefile

```
1 libf.a :
2         cd src && make
3         mv src/libf.a .
4
5 clean:
6         cd src && make clean
7         rm -f libf.a
8
9 .PHONY: clean
```

- libf/src/Makefile

```
1  include ../Makefile.inc
2
3  libf.a : f.o
4          $(ARCHIVER) -r $@ $<
5
6  %.o : %.cpp
7          $(COMPILER) -c $(COMPILER_FLAGS) $< -o$@
8
9  clean:
10         rm -f *.o *.a
11
12 .PHONY: clean
```

Verzeichnisstruktur

Verwendung statischer Bibliotheken


```
./
    libf/
        inc/
            f.hpp
        src/
            f.cpp
            Makefile
        libf.a
        Makefile
        Makefile.inc
    main/
        main.cpp
        Makefile
        Makefile.inc
```

- Bibliothek installiert (libf.a generiert)
- main/main.cpp verwendet Bibliotheksfunktion f

- main/Makefile.inc wie bei libf
- main/Makefile

```
1  include Makefile.inc
2
3  LIB_INC_PATH=../libf/inc
4  LIB_PATH=../libf
5
6  main.exe : main.cpp
7          $(COMPILER) -I$(LIB_INC_PATH) $< -o$@ -L$(LIB_PATH) -lf
8          ./$@ C++-Community
9
10
11 clean:
12         rm -f *.exe
13
14 .PHONY: clean
```

Inhalt

```
1  f.hpp // Zielfunktion (Deklaration)
2  f.cpp // Zielfunktion (Definition)
3
4  df1.hpp // Ableitung symbolisch (Deklaration)
5  df1.cpp // Ableitung symbolisch (Definition)
6
7  df2.hpp // Ableitung numerisch (Deklaration)
8  df2.cpp // Ableitung numerisch (Definition)
9
10 newton.hpp // Newton (Deklaration)
11 newton.cpp // Newton (Definition)
12
13 nle.cpp // Anwendung 1
14 nle_fd.cpp // Anwendung 2
15
16 Makefile // Übersetzungsprozess
```

Newton

Übersetzung mit `make`


```
1  COMPILER=g++ -c
2  COMPILER_FLAGS=-std=c++14 -Wall -Wextra -pedantic -O3
3  LINKER=g++
4  EXECUTABLES=nle.exe nle_fd.exe
5
6  all: $(EXECUTABLES)
7
8  nle.exe : nle.o newton.o f.o df1.o
9          $(LINKER) $^ -o$@
10
11 nle_fd.exe : nle_fd.o newton.o f.o df2.o
12         $(LINKER) $^ -o$@
13
14 %.o : %.cpp
15         $(COMPILER) $(COMPILER_FLAGS) $< -o$@
16
17 clean:
18         rm -f *.o *.exe
19
20 .PHONY: all clean
```

Newton

Übersetzung mit `make` (statische Bibliothek)


```
1  COMPILER=g++ -c
2  COMPILER_FLAGS=-std=c++14 -Wall -Wextra -pedantic -O3
3  LINKER=g++
4  EXECUTABLES=nle.exe nle_fd.exe
5  ARCHIVER=ar
6
7  all: $(EXECUTABLES)
8
9  libnewton.a : newton.o
10         $(ARCHIVER) -r $@ $^
11
12 nle.exe : libnewton.a nle.o f.o df1.o
13         $(LINKER) -L. $^ -o$@ -lnewton
14
15 nle_fd.exe : libnewton.a nle_fd.o f.o df2.o
16         $(LINKER) -L. $^ -o$@ -lnewton
17
18 %.o : %.cpp
19         $(COMPILER) $(COMPILER_FLAGS) $< -o$@
20 ...
```

Newton

Verzeichnisstruktur

```
libnewton // (statische) Bibliothek
    inc
        newton.hpp
    src
        Makefile
        newton.cpp
    Makefile
    Makefile.inc
main // Anwendung
    inc
        df1.hpp
        df2.hpp
        f.hpp
    src
        df1.cpp
        df2.cpp
        f.cpp
        Makefile
    nle.cpp
    nle_fd.cpp
    Makefile
    Makefile.inc
```

Verzeichnisstruktur

Übersetzung mit `make`

- libnewton/Makefile.inc

```
1  COMPILER=g++ -c
2  COMPILER_FLAGS=-std=c++14 -Wall -Wextra -pedantic -O3
3  LINKER=g++
4  ARCHIVER=ar
```

- libnewton/Makefile

```
1  include Makefile.inc
2
3  libnewton.a :
4          cd src && make
5          $(ARCHIVER) -r $@ src/*.o
6
7  clean:
8          cd src && make clean
9          rm -f *.o *.exe *.a
10
11 .PHONY: clean
```

- libnewton/src/Makefile

```
1  include ../Makefile.inc
2
3  INCLUDE_PATHS=-I../inc
4  TARGETS=newton.o
5
6  all: $(TARGETS)
7
8  %.o : %.cpp
9          $(COMPILER) $(COMPILER_FLAGS) $(INCLUDE_PATHS) $< -o$@
10
11 clean:
12         rm -f *.o
13
14 .PHONY: all clean
```

Verzeichnisstruktur

Übersetzung mit `make`

- main/Makefile.inc wie bei libnewton
- main/Makefile

```
1  include Makefile.inc
2
3  all:
4          @echo "USAGE: make <target>.exe"
5
6  %.exe : %.o
7          cd src && make
8          $(LINKER) -L../libnewton $^ src/*.o -o$@ -lnewton
9
10 %.o : %.cpp
11         $(COMPILER) $(COMPILER_FLAGS) -I./inc -I../libnewton/inc $< -o$@
12
13 clean:
14         cd src && make clean
15         rm -f *.o *.exe *.a
16
17 .PHONY: all clean
```

- main/src/Makefile

```
1  include ../Makefile.inc
2
3  INCLUDE_PATHS=-I../inc
4  TARGETS=$(addsuffix .o, $(basename $(wildcard *.cpp)))
5
6  all: $(TARGETS)
7
8  %.o : %.cpp ../inc/%.hpp
9          $(COMPILER) $(COMPILER_FLAGS) $(INCLUDE_PATHS) $< -o$@
10
11 clean:
12         rm -f *.o
13
14 .PHONY: all clean
```

Zusammenfassung

Eine Quelldatei

- Deklaration vs. Definition von Funktionen
- Übersetzung mit `make`

Mehrere Quelldateien

- Quellcodestruktur und statische Bibliotheken
- Übersetzung mit `make`

Verzeichnisstruktur

Newton

- Mehrere Quelldateien
- Statische Bibliothek
- Verzeichnisstruktur