Leibniz Universität IT Services

Leibniz Universität Hannover

C++ - Einführung in die Programmiersprache

Nutzung von Modulen

Modularisierung

- Unterteilung von sehr großen Programmen in viele kleine Einheiten.
- Pro Datei wird ein Arbeitsprozess beschrieben.
- Trennung von Deklaration und Definition von Subroutinen und globalen Variablen.
- Wiederverwendung von einzelnen Subroutinen.

Module in C++

- Jedes Modul besteht aus einer Header-Datei (*.h) und einer Quelltextdatei (*.cpp).
- Die Deklaration in der Header-Datei einer Subroutine wird von der Definition in der Quelltextdatei getrennt.

Quelldatei taschenrechner.cpp

```
#include "Taschenrechner.h"

int addition(int paramL, int paramR){
    return(paramL + paramR);
}


int subtraktion(int paramL, int paramR){
    return(paramL + paramR);
}

int multiplikation(int paramL, int paramR){
    return(paramL * paramR);
}
```

Quelldatei: Start eines C++Programms (main.cpp)

`#include …;`	Präprozessor-Anweisungen
`using namespace …;`	Namensraum-Deklaration

`int main(int argc, char** argv)`
`{`
`  …;`
`}`
	Funktionsdefinition

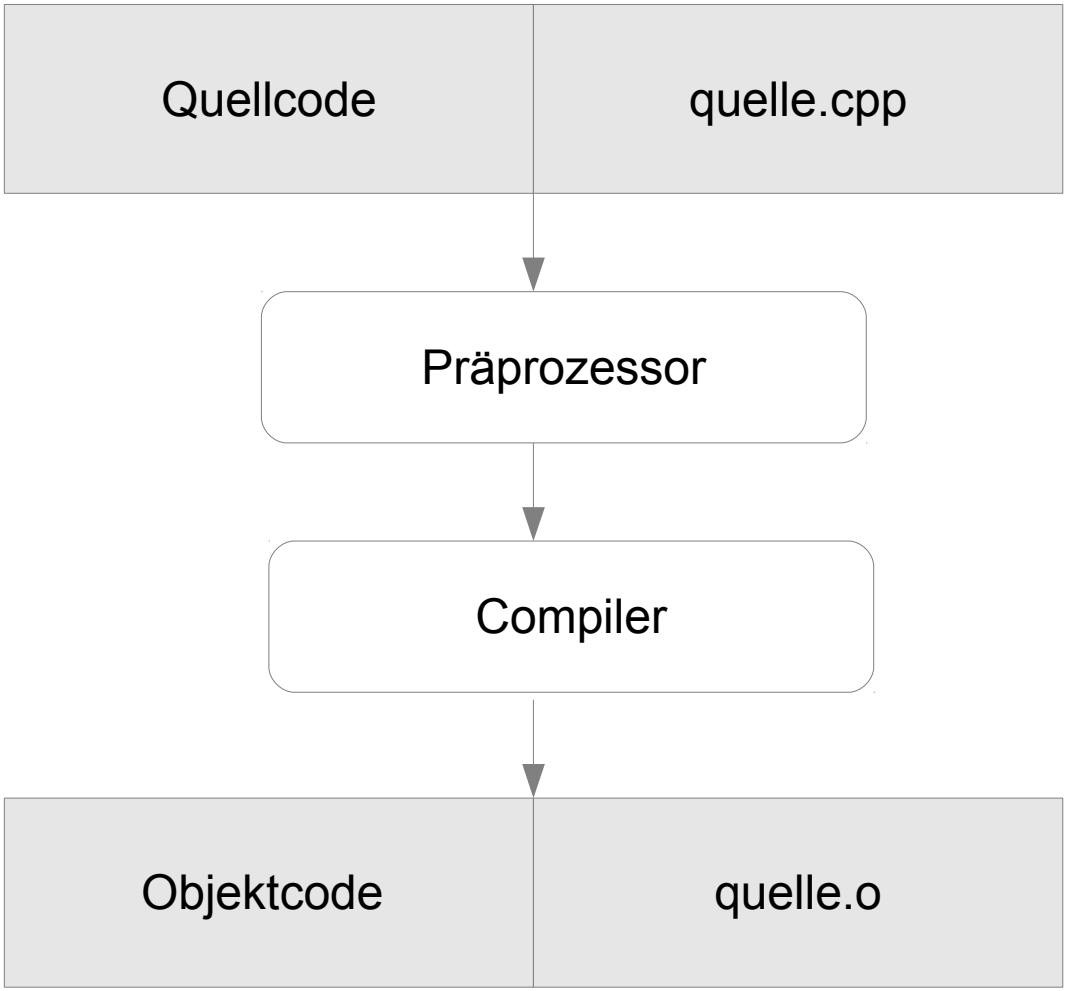
Weitere Quelldateien (*.cpp)

`#include …;`	Präprozessor-Anweisungen
`using namespace …;`	Namensraum-Deklaration

`typ fktName(typ para01, typ para02)`
`{`
`  …;`
`}`
	Funktionsdefinition

`void przName(typ para01, typ para02)`
`{`
`  …;`
`}`
	Prozedurdefinition

Kompilierung von Quelldateien



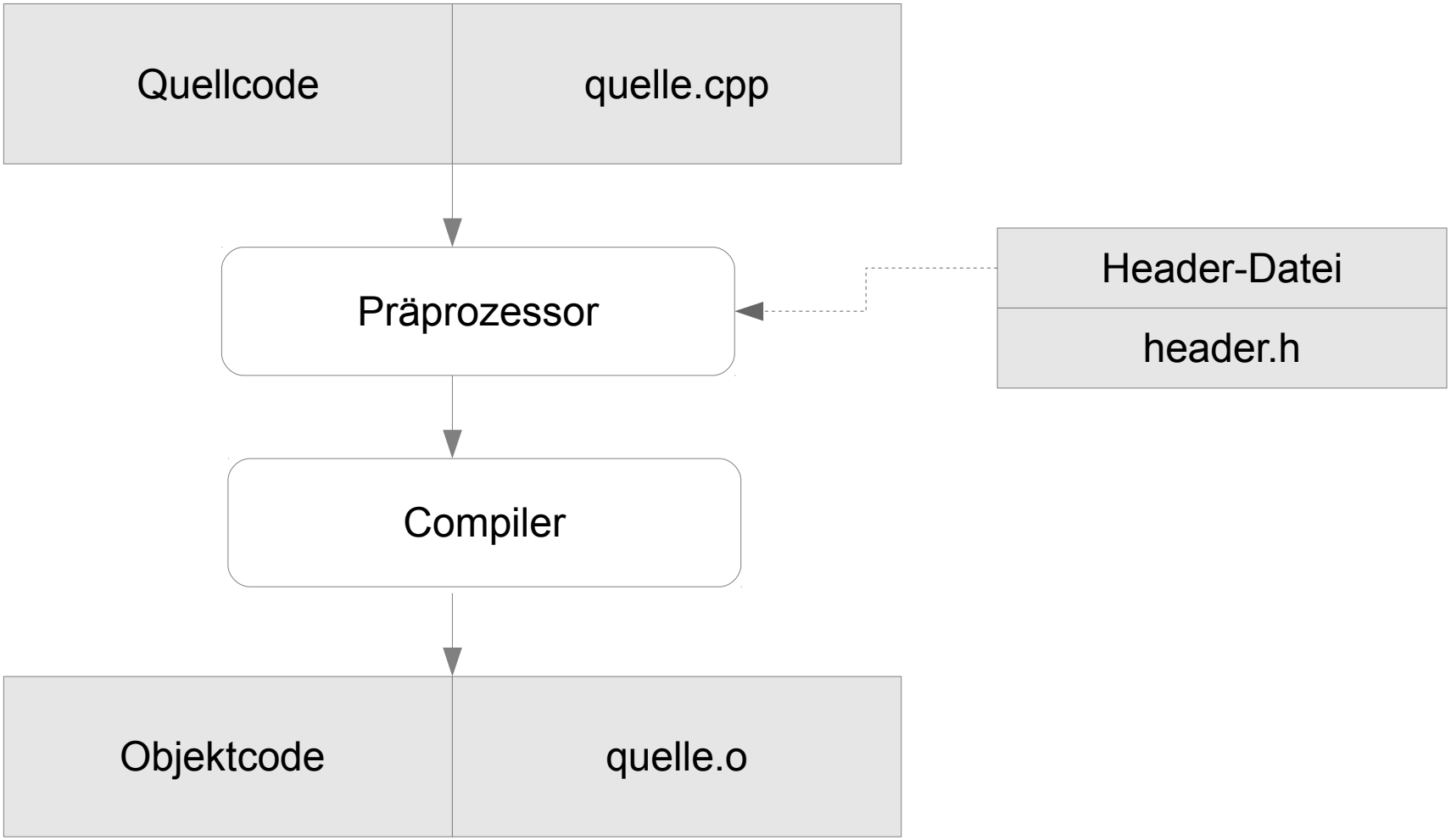
Header-Datei taschenrechner.h

```
#ifndef TASCHENRECHNER_H
#define TASCHENRECHNER_H


int addition(int, int);

int subtraktion(int, int);

int multiplikation(int, int);


#endif
```

Aufgaben

- Vorspanndatei zu einer Quelltextdatei.
- Instruktionen für den Compiler.
- Deklaration von Subroutinen und globalen Konstanten.
- Schnittstelle nach außen.

Kompilierung des Programms



Präprozessor-Anweisungen

- Beginn mit einem Hash (#).
- Ein Semikolon am Ende einer Präprozessor-Anweisung erzeugt eine Warnung.
- Pro Zeile ist eine Präprozessor-Anweisung erlaubt.
- Anweisungen für den Präprozessor stehen immer am Anfang einer Datei.

Einbindung von Header-Dateien

```
#include <iostream>
#include "Taschenrechner.h"
```

- Das Schlüsselwort `#include` bindet an dieser Position die gewünschte Header-Datei ein.
- An dieser Position wird die Anweisung durch den Inhalt der Header-Datei ersetzt.

… aus der Standard-Bibliothek

```
#include <iostream>
```

- Der Name der Header-Datei wird durch die spitzen Klammern begrenzt.
- Die Dateiendung muss nicht angegeben werden.
- Alle Header-Dateien, die in der Standardbibliothek definiert sind. Die Dateien sind im Ordner *include* des Compilers abgelegt.
- Siehe http://en.cppreference.com/w/cpp/header

„include“-Ordner in NetBeans

- *Tools – Options*.
- Aktivierung der Kategorie *C/C++.*
- Registerkarte *Code Assistance.* Registerkarte *C++ Compiler.*
- Hinweis: Die Ordner werden in der angegebenen Reihenfolge nach einer passenden Header-Datei durchsucht.

Eigene Header-Dateien einbinden

```
#include <iostream>
#include "Taschenrechner.h"
```

- Der Name plus die Dateiendung werden in Anführungszeichen gesetzt.
- Die dazugehörige Quelldatei hat den gleiche Namen wie die Header-Datei.
- Standardmäßig wird die Header-Datei zuerst im Projektverzeichnis gesucht.

„include“-Ordner in NetBeans

- Rechter Mausklick auf den Projektnamen.
- Im Kontextmenü wird der Eintrag *Properties* ausgewählt.
- Die Kategorie *Build – C++Compiler* wird aktiviert.
- Im Abschnitt *General* kann mit Hilfe der drei Punkte zu dem Element *Include Directories* die benötigten Pfade eingebunden werden.

Ist die Datei eingebunden?

```
#ifndef TASCHENRECHNER_H
#define TASCHENRECHNER_H

#endif
```

Symbolische Konstante

```
#define TASCHENRECHNER_H
```

- Die Präprozessor-Anweisung `#define` definiert eine symbolische Konstante oder Makro.
- In diesem Beispiel wird eine symbolische Konstante `TASCHENRECHNER_H` definiert.
- Symbolische Konstanten können mit der Anweisung `#undef` entfernt werden.

Existiert die Konstante?

```
#ifndef TASCHENRECHNER_H

#endif
```

- `#ifndef` entspricht „Wenn die Konstante nicht definiert ist“.
- Die if-Bedingung endet mit der Anweisung `#endif`.

Funktionsprototypen in einer Header-Datei

- Der Funktionskopf wird als Prototyp für eine Funktion genutzt.
- Die Deklaration und Definition von Funktionen werden getrennt.
- Definition einer Schnittstelle nach außen.
- Irgendwo in diesem Programm existiert eine Funktion ...

… für den Compiler

- Korrekter Umgang mit dem Rückgabewert einer Funktion.
- Überprüfung, ob die korrekte Anzahl von Parametern übergeben wird.
- Überprüfung, ob die Parameter mit den richtigen Datentypen übergeben werden.

Beispiele

```
int addition(int, int);

int subtraktion(int, int);

int multiplikation(int, int);
```

Prototypen

datentyp	fktName	(			)	;
void	fktName	(			)	;

datentyp	fktName	(	typ	,	typ	)	;
void	fktName	(	typ	,	typ	)	;

Hinweise

- Funktionsprototypen werden in einer Header-Datei oder am Anfang einer Quelldatei geschrieben.
- Funktionsprototypen enden mit einem Semikolon.
- Die Parameterliste enthält beliebig viele Parameter. Für jeden Parameter wird der Datentyp im Prototypen angegeben, aber nicht der Name.

Optionale Parameter im Prototypen

```
#ifndef TASCHENRECHNER_H
#define TASCHENRECHNER_H

int multiplikation(int, int = 1);
```

- Dem Datentyp wird mit Hilfe eines Gleichheitszeichens ein Wert zugewiesen.
- Optionale Parameter stehen am Ende der Parameterliste. Optionale Parameter folgen nur weitere optionale Parameter.

Definition der Funktion

```
int multiplikation(int paramL, int paramR){
    return(paramL * paramR);
}
```

- Der Funktionskopf in der Quelldatei enthält keine Hinweise auf einen optionalen Parameter.

Aufruf

```
result = multiplikation(argL, argR);

result = multiplikation(argL);
```

- Falls kein Argument an den Parameter übergeben wird, wird der optionale Wert genutzt.

Überladung von Funktionen

- Funktionen, die den gleichen Namen haben, werden anhand der Parameterliste überladen.
- Funktionen werden mit Hilfe ihres Namens und der Anzahl der Parameters aufgerufen.
- Der Datentyp des Rückgabewertes kann bei Funktionen gleichen Namens identisch sein, muss aber nicht.

Prototypen

```
int subtraktion(int, int);
int subtraktion(int, int, int);
double subtraktion(double, double);
```

- Alle drei Prototypen haben den gleichen Namen. Hinweis: Die Groß- und Kleinschreibung wird beachtet.
- Aber die Parameterliste unterscheidet sich in der Anzahl der Parameter und / oder dem Datentyp der Parameter.
- Der Datentyp der Funktion kann unterschiedlich sein, muss aber nicht.

Implementierung

```
int subtraktion(int paramL, int paramR){
    return(paramL - paramR);
}

int subtraktion(int paramL, int paramM, int paramR){
    return(paramL - paramM - paramR);
}

double subtraktion(double paramL, double paramR){
    return(paramL - paramR);
}
```

Aufruf

```
result = subtraktion(5, 3);
result = subtraktion(20, 3, 4);
ergebnis = subtraktion(3.21, 1.4);
```

- In Abhängigkeit der Parameterliste werden die verschiedenen Funktionen aufgerufen.

Namensraum

- In welchen Bereich sind die Funktionen sichtbar?
- Vermeidung von Namenskonflikten.
- Programme werden zu „Paketen“ zusammengefasst.

Standard-Namensraum

- Alle Funktionen etc. der Programmiersprache C++ können in dem Namensraum `std` liegen.
- `std` ist der Standard-Namensraum für Objekte, die in der Standardbibliotheken definiert sind.

Globale Definition

```
using namespace std;
```

- Am Anfang des Programms wird der Namensraum festgelegt.
- Welcher Namensraum wird genutzt?
- Wenn möglich, sollte die globale Definition von Namensräumen in Header-Dateien vermieden werden.

Lokale Definition

```
std::cout << "Addition: " << result << '\n';
```

- Der Namensraum und darin definierte Elemente werden mit Hilfe des Bereichs- oder Gültigkeitsoperators verbunden.
- In diesem Beispiel wird das Objekt `cout` aus dem Standard-Namensraum genutzt.