Leibniz Universität IT Services

Leibniz Universität Hannover

C++ - Einführung in die Programmiersprache

„Programmierparadigmen“

C++

- ISO genormte Programmiersprache.
- Objektorientierte und generische Programmierung.
- Komplexe, flexible Programmiersprache.
- Hardware-nahe Programmierung.

Standard

- C++98. ISO/IEC 14882:1998. Publiziert im Jahr 1998. Erste Standardisierung der Programmiersprache. Ergänzung der Norm im Jahr 2003.
- C++11. ISO/IEC 14882:2011. Publiziert im April 2011. Ersetzung der alten Versionen.
- C++14. ISO/IEC 14882:2014. Publiziert im Januar 2015. Erweiterungen und Ergänzungen zu C++11.
- C++17. ISO/IEC 14882:2017. Publiziert im März 2017. Die Änderung werden auf der Webseite https://isocpp.org/files/papers/p0636r0.html beschrieben.
- Aktueller Status: https://isocpp.org/std/status

Programmierparadigma

- C++ beschreibt, wie etwas zu tun ist.
- Die strukturierte und prozedurale Programmierung lehnt sich stark an die Programmiersprache C an.
- Die objektorientierte Programmierung beschreibt Objekte und deren Handlung
- Die generische Programmierung bietet Vorlagen für das Erstellen von Objekten an.

Strukturierte Programmierung

- Befehle werden sequentiell ausgeführt. Mit Hilfe von Ausdrucksanweisungen werden Berechnungen wie zum Beispiel die Addition ausgeführt.
- Befehle werden in Abhängigkeit von Bedingungen ausgeführt. Mit Hilfe von „Wenn-Dann“-Anweisungen wird der Status einer Variablen geprüft. Entsprechend des Ergebnisses der Prüfung werden Befehle ausgeführt oder nicht.
- Befehle können wiederholt werden. Mit Hilfe von Schleifen werden Anweisungen wiederholt.

Beispiel

```
int main()
{
  int ergebnis = 0;

  ergebnis = 4 + 5;

  return 0;
}
```

Anweisungen

- Die Anweisungen werden Zeile für Zeile abgearbeitet.
- Jede Anweisung in der Programmiersprache C++ endet mit einem Semikolon.
- Anweisungen werden mit Hilfe der geschweiften Klammern zusammengefasst. Jeder Block hat in irgendeiner Form einen Kopf.

… in diesem Beispiel

- Im ersten Schritt wird die Variable `ergebnis` deklariert. Diese Variable wird mit dem Wert 0 initialisiert.
- Im nächsten Schritt werden zwei Werte addiert. Das Ergebnis der Addition wird zur weiteren Verwendung in `ergebnis` gespeichert.
- Die Anweisungen werden in dem Block `main` zusammengefasst.

Prozedurale Programmierung

- Die Gesamtaufgabe wird in verschiedene Teilaufgaben aufgelöst.
- Aufgaben, die an verschiedenen Stellen benötigt werden, werden ausgelagert. Der Code kann wiederverwendet werden.
- Aufgaben zu einem Thema werden in Dateien zusammengefasst. Diese Module können für verschiedene Aufgaben genutzt werden.

Beispiel

```
int addieren(int paramL, int paramR){
    int ergebnis = 0;

    ergebnis = paramL + paramR;
    return ergebnis;
}

int main(){
  int argL = 4;
  int argR = 5;
  int ergebnis = 0;

  ergebnis = addieren(argL, argR);
  return 0;
}
```

… in diesem Beispiel

- Die Aufgabe wird in die Funktionen `main` und `addieren` aufgeteilt.
- Die Funktion `addieren` fasst alle Anweisungen zusammen, die die Aufgabe „Addition“ benötigt.
- Die Funktion `main` bildet eine Startzentrale für das ausführende Programm ab. Die Funktion verzweigt in die verschiedenen Funktionen.

Funktion „main“

```
int main(){
   int argL = 4;
   int argR = 5;
   int ergebnis = 0;

   ergebnis = addieren(argL, argR);
   return 0;
}
```

Aufgabe der Funktion main()

- Jedes Projekt besitzt eine Steuerungszentrale. Diese Zentrale ist durch den Namen `main` gekennzeichnet.
- Durch Starten des Programms wird die Funktion aufgerufen. Die Funktion wird Zeile für Zeile abgearbeitet.
- Eine Verzweigung in Funktionen, die Teile der Gesamtaufgabe beschreiben, ist möglich.

Anweisungen in der Funktion main()

- In den ersten drei Zeilen werden Variablen deklariert und initialisiert.
- In der vierten Zeile wird die Funktion `addieren` aufgerufen. Die Funktion gibt einen Wert zurück. Dieser Wert wird in der Variablen `ergebnis` gespeichert.
- Die Funktion selber gibt eine Ganzzahl an den Aufrufer zurück. Die Null kennzeichnet eine fehlerfreie Ausführung.

Deklaration und Initialisierung

- Variablen: Platzhalter für dynamische Werte in einem Programm.
- Deklaration: Bekanntmachung der Variablen. Eine Variable muss bekannt sein, um sie zu gebrauchen.
- Initialisierung: Zuweisung eines eindeutig definierten Wert als Anfangswertes. Der Wert kann durch Anweisungen überschrieben werden.

Funktion „addieren“

```
int addieren(int paramL, int paramR){
    int ergebnis = 0;

    ergebnis = paramL + paramR;
    return ergebnis;
}
```

Funktionskopf

```
int addieren(int paramL, int paramR)
```

- Identifizierung durch den Namen. In diesem Beispiel `addieren`.
- Parameterliste: Begrenzung durch die runden Klammern. Liste von Parametern. Parameter sind von einem bestimmten Typ. Trennung der Parameter durch das Komma.
- Funktionstyp: Ganzzahl (`int`). Das Ergebnis der Berechnung wird als Ganzzahl an den Aufrufer der Funktion zurückgegeben.

Anweisungen im Codeblock

- Der Codeblock beginnt und endet mit den geschweiften Klammern.
- In der ersten Zeile wird die Variable `ergebnis` deklariert und gleichzeitig mit dem Wert 0 initialisiert.
- In der zweiten Zeile werden die zwei übergebenen Parameter addiert. Das Ergebnis der Addition wird in der Variablen `ergebnis` gespeichert.
- Mit Hilfe der Anweisung `return` wird das Ergebnis der Addition an den Aufrufer der Funktion zurückgegeben.

Auslagerung von Funktionen

- Datei *main.cpp*: Häufig nur die Funktion `main()`. Alle anderen Funktionen werden in weitere Quelldateien ausgelagert.
- Die Deklarationen (den Kopf) von Funktion kann von der Definition getrennt werden. Der Kopf der Funktion wird in einer Header-Datei gespeichert. Die Funktion wird aber in einer Datei mit der Endung *.cpp definiert.

CodeLite : C++- Dateien einem Projekt hinzufügen

- Rechter Mausklick auf den Ordner *src* in der w*orkspace pane*.
- *Add a new File* im Kontextmenü.

Auswahl *C++ Source File (.cpp)*

- Eingabe eines Dateinamens.
- Auswahl des Speicherorts. Standardmäßig wird die Datei im Ordner *src* des Projekts gespeichert.
- Klick auf *OK*. Der Dateiname wird um die Dateiendung „.cpp“ ergänzt.

Auswahl Header File (.h)

- Eingabe eines Dateinamens. Die Header-Datei sollte den gleichen Namen wie die dazugehörige Quelldatei haben.
- Auswahl des Speicherorts. Standardmäßig wird die Datei im Ordner *src* des Projekts gespeichert. Die Header-Datei und die Quelldatei sollten den gleichen Speicherort haben.
- Klick auf *OK*. Der Dateiname wird um die Dateiendung „.h“ ergänzt.

Header-Datei „taschenrechner.h“

```
int addieren(int, int);
```

- Deklarationsdatei. Kopf einer Quelldatei.
- Funktionsdeklaration und so weiter.
- Definition von Schnittstellen nach außen.

Funktionsdeklaration

```
int addieren(int, int);
```

- Kopf einer Funktion.
- Funktionstyp (hier `int`, Ganzzahlen): Was wird zurückgegeben?
- Name der Funktion: Wie wird die Aktivität aufgerufen?
- Parameterliste bestehend aus den Typen der Parameter (hier zwei Parameter vom Typ Ganzzahl): Welche Werte werden zum Starten der Aktivität benötigt?

Quelldatei „taschenrechner.cpp“

```
int addieren(int paramL, int paramR){
    int ergebnis = 0;

    ergebnis = paramL + paramR;
    return ergebnis;
}
```

- Definition von Funktionen.
- Was macht eine Funktion?
- Beschreibung der Aktivität.

Definition der Funktion „addieren“

```
int addieren(int paramL, int paramR){
    int ergebnis = 0;

    ergebnis = paramL + paramR;
    return ergebnis;
}
```

- Die Definition besteht aus dem deklarierten Funktionskopf und -rumpf.
- Der Rumpf beginnt und endet mit den geschweiften Klammern. In diesem Codeblock werden die benötigen Anweisungen geschrieben.
- Der Funktionskopf beschreibt die zu übergebenen Parameter und den Rückgabewert der Funktion.

Quelldatei „main.cpp“

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

int main()
{
  int argL = 4;
  int argR = 5;
  int ergebnis = 0;

  ergebnis = addieren(argL, argR);
  std::cout << argL << " + " << argR
            << " = " << ergebnis << std::endl;
  return 0;
}
```

Erläuterung

- Steuerzentrale / Cockpit eines Programms.
- Am Anfang der Datei stehen Präprozessor-Anweisung.
- Häufig enthält die Datei nur die Funktion `main()`.

Präprozessor-Anweisungen

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

- Beginn mit dem Hash-Zeichen (#).
- Beendigung mit der Zeile und nicht mit einem Semikolon.
- Positionierung am Anfang einer Quelldatei (.cpp).
- Vor der Kompilierung des Programms werden diese Zeilen durch den entsprechenden Text ersetzt.

Einbindung von Header-Dateien

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

- Der Befehl `#include` bindet Header-Dateien in eine Quelldatei ein.
- Die, in der Header-Datei enthaltenen Deklarationen machen Funktionen in der Quelldatei bekannt. Deklarierte Funktionen können in der Quelldatei genutzt werden.

… aus der Standard-Bibliothek

```
#include <iostream>
```

- Die Standard-Bibliothek wird mit dem Compiler ausgeliefert.
- Die Bibliothek ist eine Sammlung von vordefinierten Funktionen.
- Mit Hilfe der spitzen Klammern wird der Name der Bibliothek begrenzt. Die Dateiendung wird nicht benötigt.
- In diesem Beispiel werden Funktionen für die Ein- und Ausgabe eingebunden.

… , die selbst definiert sind

```
#include "taschenrechner.h"
```

- Der Name der Datei ist ein String. Der String wird durch die Anführungszeichen begrenzt.
- Die Dateiendung muss angegeben werden.
- Zuerst wird die Datei im Verzeichnis der Quelldatei gesucht. Anschließend in den include-Verzeichnissen des Compilers.

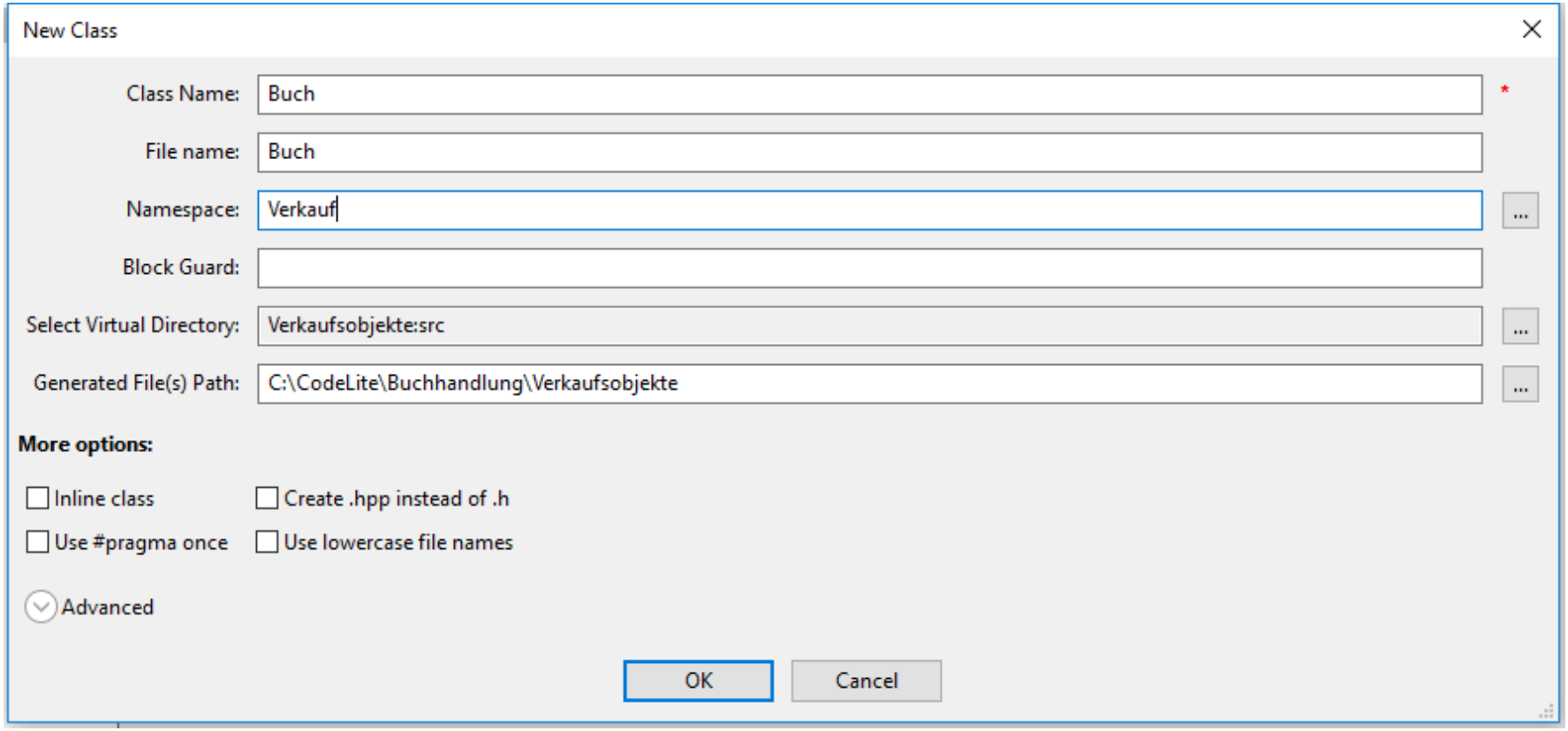
Objektorientierte Programmierung

- Allgemeine Beschreibung von Objekten.
- Objekte haben Attribute. Zum Beispiel das Objekt „Mensch“ hat eine bestimmte Größe, Augenfarbe, Haarfarbe etc.
- Objekte handeln. Zum Beispiel läuft oder schläft das Objekt „Mensch“.

CodeLite: Klassen einem Projekt hinzufügen

- Rechter Mausklick auf den Ordner *src* zu einem Projekt im *workspace pane*.
- *Add Class* im Kontextmenü.

Dialogfenster



CodeLite: Klassen einem Projekt hinzufügen

- Zeile *Class Name:* Name der Klasse. Name des zu beschreibenden Objekts aus der realen Welt.
- Zeile *File name*. Speicherung in der Datei. Klassenname und Dateiname sollte gleich sein. Pro Datei wird eine Klasse definiert.
- Zeile Namespace: Angabe eines Namensraumes. Der Name ist frei wählbar. Vermeidung von Namenskonflikten.

Header-Datei „taschenrechner.h“

```
#ifndef CLSTASCHENRECHNER_H
#define CLSTASCHENRECHNER_H

class clsTaschenrechner {
    public:
        clsTaschenrechner();
        clsTaschenrechner(int, int);

        int addieren();

    private:
        int ergebnis;
        int wertL;
        int wertR;

};

#endif /* CLSTASCHENRECHNER */
```

Erläuterung

- Jede Deklarationsdatei hat die Endung „.h“.
- In dieser Deklarationsdatei wird eine Klasse clsTaschenrechner definiert.
- Die Klasse beschreibt ein Objekt „Taschenrechner“.
- Das Objekt hat Attribute. Attribute beschreiben das Objekt. Die Beschreibung ist privat. Die Attributwerte können nicht von Außenstehenden verändert werden.
- Das Objekt hat Methoden, die Aktivitäten beschreiben. Es gibt in der Klasse Methoden zum Erzeugen von Objekten und andere Tätigkeiten. Diese Methoden sind öffentlich. Ein Außenstehender kann diese aufrufen.

Präprozessor-Anweisung

```
#ifndef CLSTASCHENRECHNER_H

   #define CLSTASCHENRECHNER_H

#endif
```

- Wenn die symbolische Konstante `CLSTASCHENRECHNER_H` nicht definiert ist, definiere diese.
- Durch diese Präprozessor-Anweisungen wird ein mehrmaliges Einbinden ein und derselben Header-Datei verhindert.

#pragma-Anweisungen

```
#pragma once
```

- Anweisungen an den Compiler.
- Wenn du die Anweisung verstehst, dann führe sie aus.
- In diesem Beispiel wird jede Header-Datei nur einmal eingebunden.

Quelldatei „taschenrechner.cpp“

```
#include "taschenrechner.h"

clsTaschenrechner::clsTaschenrechner() {
    this->wertL = 0;
    this->wertR = 0;
    this->ergebnis = 0;
}

clsTaschenrechner::clsTaschenrechner(int paramL, int paramR)
{
    this->wertL = paramL;
    this->wertR = paramR;
    this->ergebnis = 0;
}

int clsTaschenrechner::addieren(){
    this->ergebnis = this->wertL + this->wertR;
    return this->ergebnis;
}
```

Erläuterung

- Die Deklarationsdatei zur Klasse muss am Anfang der Datei eingebunden werden.
- Anschließend werden die verschiedenen Methoden definiert.
- Methoden, die den gleichen Namen wie die Klasse tragen, erzeugen ein Objekt von dieser Klasse.
- Methoden beschreiben Tätigkeiten eines Objektes. Methoden beschreiben Schnittstellen nach außen.

Quelldatei „main.cpp“

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

int main()
{
  int argL = 4;
  int argR = 5;
  int ergebnis = 0;

  class clsTaschenrechner rechnerLeer;
  std::cout << "Addition ohne Parameter: "
            << rechnerLeer.addieren() << std::endl;

  class clsTaschenrechner rechner(argL, argR);
  ergebnis = rechner.addieren();
  std::cout << argL << " + " << argR
            << " = " << ergebnis << std::endl;
  return 0;
}
```

Erzeugung eines Objektes

```
class clsTaschenrechner rechnerLeer;
class clsTaschenrechner rechner(argL, argR);
```

- Deklaration eines Objekts.
- `class clsTaschenrechner`. Herkunft des Objekts. Von welcher Klasse wird das Objekt abgeleitet?
- Name des Objekts (`rechnerLeer`, `rechner`).
- In den runden Klammern werden wichtige Parameter zur Erzeugung des Objekts übergeben oder nicht.

Aufruf der Methoden

```
rechnerLeer.addieren();
```

- Die Methode wird immer über den Namen des Objekts aufgerufen.
- Objekt und Methode werden mit dem Punkt verbunden.
- Die Methode wird entsprechend ihrer Deklaration aufgerufen.

Konzepte

- Datenkapselung. Die Attribute eines Objekts können nur durch Handlungen des Objektes verändert werden. Die Attribute sind im Objekt gekapselt und können nur über definierte Schnittstellen von außen verändert werden.
- Vererbung. Attribute und Methode können von Eltern-Objekten an deren Kind-Objekte vererbt werden. Die Kind-Objekte können diese wiederum verändern oder überschreiben.
- Polymorphie. Vielgestaltigkeit. Austauschbarkeit von Objekten.

Generische Programmierung

- Zusammenfassung von Objekten zu einer Gruppe.
- Alle Handlungen und Attribute dieser Gruppe werden allgemeingültig beschrieben.
- Jedes Objekt aus der Gruppe kann mit Hilfe einer Vorlage beschrieben werden.

Header-Datei „taschenrechner.h“

```
#ifndef CLSTASCHENRECHNER_H
#define CLSTASCHENRECHNER_H

template <typename T>
T addieren(T paramL, T paramR){
    T ergebnis = 0;
    ergebnis = paramL + paramR;

    return ergebnis;
}

#endif /* CLSTASCHENRECHNER */
```

Template

- Muster, um Funktionen oder Klassen zu erzeugen.
- In diesem Beispiel wird die Addition für einen beliebigen Datentyp angelegt.
- Der Buchstabe `T` ist ein Platzhalter für einen beliebigen Datentyp.

Quelldatei „main.cpp“

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


int main()
{
  decltype(addieren(4, 5)) ergebnis = addieren(4, 5);
  std::cout << " 4 + 5  = " <<   ergebnis << std::endl;

  auto result = addieren(3.2, 4.5);
  std::cout << " 3.2 + 4.5  = " << result << std::endl;
  return 0;
}
```

Aufruf der Funktion

- Die Deklaration des Templates muss am Anfang der Quelldatei eingebunden werden.
- Die Funktion wird entsprechend der Deklaration im Template aufgerufen.
- In diesem Beispiel kann der Funktion `addieren` Parameter vom beliebigen Datentyp übergeben werden. Entsprechend der gewählten Parameter wird ein Ergebnis zurückgegeben.

Speicherung des Rückgabewertes

- Durch das Schlüsselwort `auto` kann die Variable `result` jeden beliebigen Datentyp annehmen.
- Durch das Schlüsselwort `decltype` bekommt die Variable `ergebnis` den Datentyp des Rückgabewertes der Funktion zugewiesen.