Einführung in das wissenschaftliche Arbeiten

Einheit VII:
Programmiersprachen - Teil 1: C++

Dr. Björn Eichmann (eiche@tp4.rub.de)

Bochum, 01. April 2019

Übersicht an Programmiersprachen

- >> 100 Sprachen (https://de.wikipedia.org/wiki/Liste_von_Programmiersprachen)
- Maschinensprache (Assemblersprache)

  Verwenden den Befehlsvorat eines bestimmten Computertyp (verschiedene prozessorabhängige *Assembler* wie z.B Microsoft Macro Assembler (MASM), Borland Turbo Assembler (TASM), Netwide Assembler (NASM),…)
- Höhere Progammiersprachen

  Verwenden einen *Compiler* der die abstrakteren (komplexeren, nicht auf den Prozessor-Befehlssatz begrenzte) Befehle in den Maschinencode der gegebenen Zielarchitektur übersetzt (z.B. Fortran, C, C++,…)
- Skriptsprachen

  Ermöglichen unmittelbare Ausführung über einen *Interpreter*, mit dem Verzicht auf gewisse Sprachelemente wie dem Deklarationszwang von Variablen (z.B. Python, Perl,…)

Objektorientierte Programmiersprachen

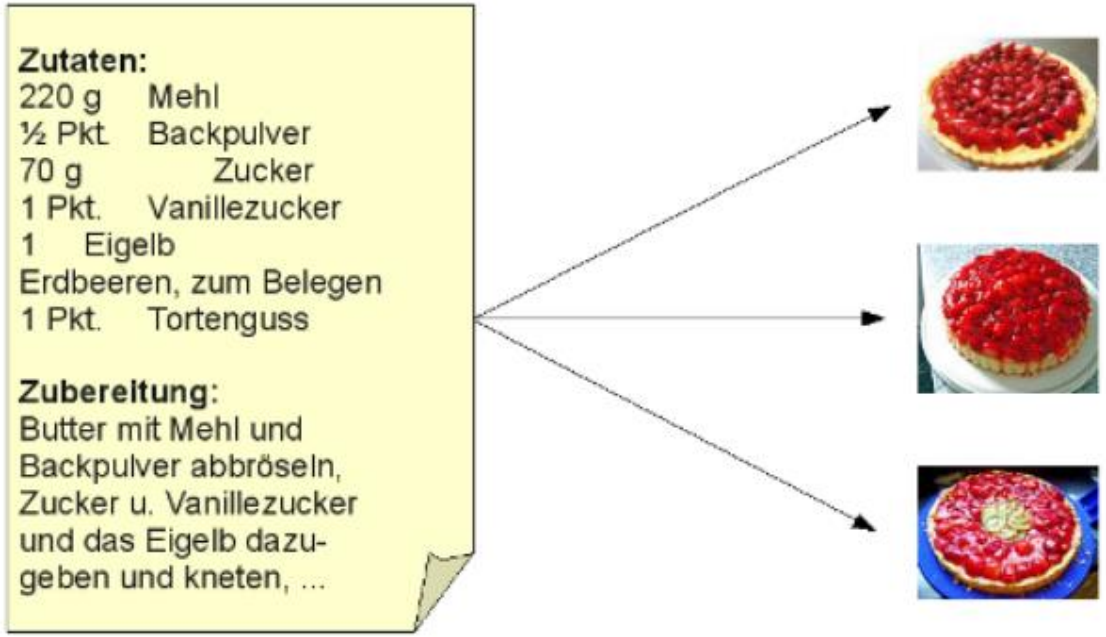
- Objektorientiertes programmieren (OOP), das bedeutet
  - „eine Technik oder Programmiersprache, welche **Objekte, Klassen und Vererbung unterstützt**.“ – ISO/IEC 2382-15
  - „die Architektur einer Software wird an den Grundstrukturen desjenigen Bereichs der Wirklichkeit ausgerichtet, der die gegebene Anwendung betrifft“ – WIKIPEDIA
- Einführung von neuen Begriffen/Bausteinen/*Objekten*
  - *Objekt*: Ein Element, welches *Funktionen*, *Methoden*, *Prozeduren*, einen inneren Zustand, oder mehrere dieser Dinge besitzt.
  - *Funktion:* Ordnet einer gegebenen Eingabe einen bestimmten Rückgabewert zu (verändert aber nicht den Zustand eines Objekts).
  - *Methode:* Verändert den Zustand eines Objekts; liefert einen Rückgabewert.
  - *Prozedur:* Verändert den Zustand eines Objektes; liefert keinen Rückgabewert
  - ***Klasse**:* Zur besseren Verwaltung gleichartiger Objekte. Die Datenstruktur eines Objekts wird durch die *Attribute* (Eigenschaften) seiner Klassendefinition festgelegt. Das Verhalten wird von den Methoden der Klasse bestimmt. Klassen können von anderen Klassen *abgeleitet* werden (Vererbung).

Objektorientierte Programmiersprachen

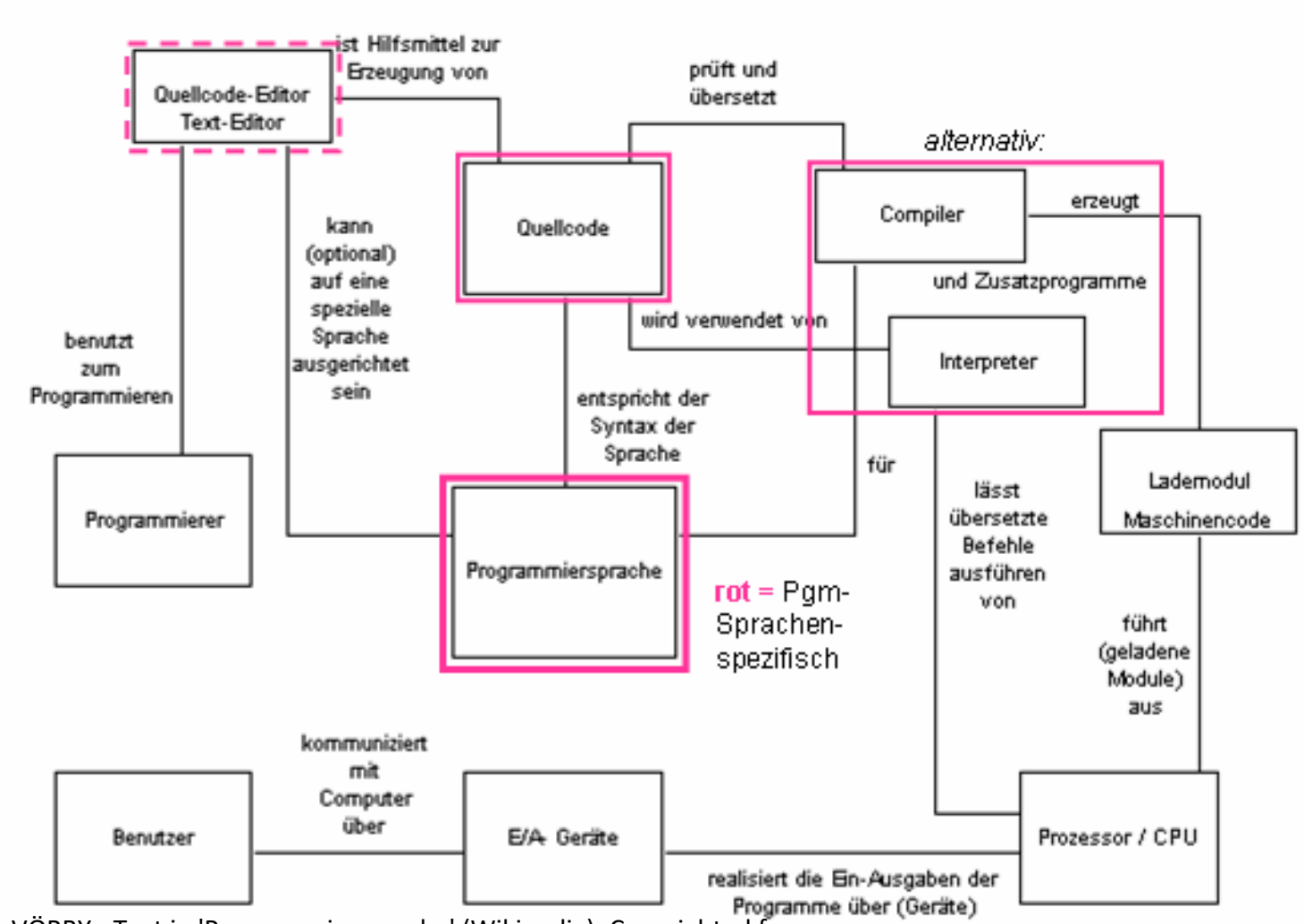
Ein Grundkonzept der objektorientierten Programmierung (OOP) besteht darin, **Daten und deren Funktionen (Methoden) in einem Objekt zusammenzufassen** und nach außen zu kapseln.

- Solche Objekte werden über ***Klassen*** definiert.
  - *Klassen* sind Vorlagen („Baupläne“) nach denen Objekte (*Instanz* einer Klasse) erzeugt werden

- Klasse als Backrezept:
  - Das Rezept (die *Klasse*) bestimmt wie der Kuchen (eine *Instanz* der Klasse) beschaffen sein muss.
  - Verschieden *Methoden* (wie z.B.:„Teig anrühren“)
  - Ein (allgm.) Kuchen *vererbt* seine Eigenschaften an die Unterklasse Erdbeerkuchen



Vom Quellcode zur Ausführung



VÖRBY - Text in 'Programmiersprache' (Wikipedia): Copyrighted free use
Quelle: https://de.wikipedia.org/wiki/Programmiersprache#/media/Datei:Programmiersprache_Umfeld.png

Eine (ganz kurze) Einführung in C++

Hilfreiche Literatur

- Bjarne Stroustrup: Programming. Principles and Practice Using C++. Addision Wesley.
- Ulla Kirch & Peter Prinz: C++ - Lernen und professionell anwenden. mitp.
- Ulla Kirch & Peter Prinz: C++. Das Übungsbuch. mitp.
- Ulrich Breymann: Der C++-Programmierer: C++ lernen – professionell anwenden – Lösungen nutzen. Carl Hanser Verlag.
- Torsten T. Will: C++. Das umfassende Handbuch. Rheinwerk.
- Jürgen Wolf: Grundkurs C++. Galileo Press.
- TU Braunschweig: https://www.tu-braunschweig.de/Medien-DB/statik/software/einfuehrung_in_cpp.pdf
- Universität Graz: https://imsc.uni-graz.at/haasegu/Lectures/Kurs-C/Script/html/script_programmieren.pdf
- DHBW Stuttgart: http://wwwlehre.dhbw-stuttgart.de/~kfg/cpp/cpp.pdf

Web Tutorials

- http://www.stroustrup.com/C++.html
- http://www.cplusplus.com/doc/tutorial/
- http://www.cprogramming.com/tutorial.html
- http://www.learncpp.com/
- http://www.etp.physik.uni-muenchen.de/kurs/Computing/ckurs/cxx.html
- https://de.wikibooks.org/wiki/C%2B%2B-Programmierung

- …auf spielerische Art: https://www.codingame.com/start

Zur Geschichte

https://en.wikipedia.org/wiki/C%2B%2B

- Ab 1979 von Bjarne Stroustrup (DNK) entwickelt, zunächst unter dem Namen „C mit Klassen“.
- 1983: Namensänderung in C++.
- 1985: Erste Veröffentlichung.
- 1989: Erweiterung um Mehrfachvererbung, abstrakte Klassen, statische Elementfunktionen, konstante Elementfunktionen und die Erweiterung des Zugriffsmodells (C++2.0)
- 1998: endgültige Fassung der Sprache (C++98)
- 2011: viele weitreichende Neuerungen (C++11)
- 2014: kleinere Ergänzungen/ Änderungen (C++14)

Compiler

- C++ ist eine **Compilersprache**, keine Interpretersprache
- Die GNU-Compiler für C++ sind insbesondere für Linux, Windows, Mac-OS, kostenlos verfügbar. https://gcc.gnu.org/
- Alternativ: *Integrated Development Environment (**IDE**),* z.B.:
  - *eclipse* (*https://www.eclipse.org/cdt/*)

    Für Windows, Linux, Mac OS X; open-source, kostenlos, sehr popular, einsteigerfreundlich
  - *Code::Blocks* (http://www.codeblocks.org/)

    Für Windows, Linux, Mac OSX; kostenlos, sehr umfangreich
  - *KDevelop* (https://www.kdevelop.org/)

    Für Windows, Linux, Mac OSX; open-source, kostenlos, unterstützt auch Python, QML/JavaScript und PHP
  - *Geany* (https://www.geany.org/)

    Für Windows, Linux, Mac OSX; kostenlos, übersichtlich (umfangarm), unterstützt auch Java, PHP, HTML, Python, Perl, Pascal

Das „Hello World“ - Programm

- Quelltext editieren (beliebige Texteditor oder IDE):

```
#include <iostream>                    // deklariert cout, endl
using namespace std;                   // erlaubt Nutzung des Namensraumes std
                                       //         nutze cout statt std::cout
int main()                             // Beginn Hauptprogramm
{                                      // Beginn Scope
 std::cout << "Hello World" << std::endl;
 return 0;                             // Beende das Programm
}                                      // Ende Scope, Ende Hauptprogramm
```

- Quelltext *compilieren*: in IDE oder **$ g++ HelloWorld.cpp**
- Programm *ausführen*: in IDE oder **$ ./a.out**

ODER:

- Quelltext *compilieren*: **$ g++ HelloWorld.cpp -o hello.out**
- Programm *ausführen*: **$ ./hello.out**

Das „Hello World“ - Programm

- Quelltext editieren (beliebige Texteditor oder IDE):

```
#include <iostream>                    // deklariert cout, endl
using namespace std;                   // erlaubt Nutzung des Namensraumes std
                                       //         nutze cout statt std::cout
int main()                             // Beginn Hauptprogramm
{                                      // Beginn Scope
 std::cout << "Hello World" << std::endl;
 return 0;                             // Beende das Programm
}                                      // Ende Scope, Ende Hauptprogramm
```

- **#include:** Befehl des Präprozessors (Teil des Compilers), den Bibliothekquellcode der dahinter angegebene Datei **iostream** einzufügen

Das „Hello World“ - Programm

- Quelltext editieren (beliebige Texteditor oder IDE):

```
#include <iostream>                    // deklariert cout, endl
using namespace std;                   // erlaubt Nutzung des Namensraumes std
                                       //         nutze cout statt std::cout
int main()                             // Beginn Hauptprogramm
{                                      // Beginn Scope
 std::cout << "Hello World" << std::endl;
 return 0;                             // Beende das Programm
}                                      // Ende Scope, Ende Hauptprogramm
```

- **#include:** Befehl des Präprozessors (Teil des Compilers), den Bibliothekquellcode der dahinter angegebene Datei **iostream** einzufügen
- **using namespace std:** Den Namensraum **(=namespace) std** verwenden (so dass std::cout = cout und std::endl=endl)

Das „Hello World“ - Programm

- Quelltext editieren (beliebige Texteditor oder IDE):

```
#include <iostream>                    // deklariert cout, endl
using namespace std;                   // erlaubt Nutzung des Namensraumes std
                                       //         nutze cout statt std::cout
int main()                             // Beginn Hauptprogramm
{                                      // Beginn Scope
 std::cout << "Hello World" << std::endl;
 return 0;                             // Beende das Programm
}                                      // Ende Scope, Ende Hauptprogramm
```

- **#include:** Befehl des Präprozessors (Teil des Compilers), den Bibliothekquellcode der dahinter angegebene Datei **iostream** einzufügen
- **using namespace std:** Den Namensraum **(=namespace) std** verwenden (so dass std::cout = cout und std::endl=endl)
- **main():** Hauptfunktion eines jeden Programms (durch geschweifte Klammern zusammengefasst). Gibt mit **return** den Rückgabewert 0 (vom Dateityp **int**) zurück.

Das „Hello World“ - Programm

- Quelltext editieren (beliebige Texteditor oder IDE):

```
#include <iostream>                    // deklariert cout, endl
using namespace std;                   // erlaubt Nutzung des Namensraumes std
                                       //         nutze cout statt std::cout
int main()                             // Beginn Hauptprogramm
{                                      // Beginn Scope
 std::cout << "Hello World" << std::endl;
 return 0;                             // Beende das Programm
}                                      // Ende Scope, Ende Hauptprogramm
```

- **#include:** Befehl des Präprozessors (Teil des Compilers), den Bibliothekquellcode der dahinter angegebene Datei **iostream** einzufügen
- **using namespace std:** Den Namensraum **(=namespace) std** verwenden (so dass std::cout = cout und std::endl=endl)
- **main():** Hauptfunktion eines jeden Programms (durch geschweifte Klammern zusammengefasst). Gibt mit **return** den Rückgabewert 0 (vom Dateityp **int**) zurück.
- Anweisungsblock: **cout** = Ausgabe, **endl** = Zeile beenden

Das „Hello World“ - Programm

- Quelltext editieren (beliebige Texteditor oder IDE):

```
#include <iostream>
using namespace std;

int main()
{
 std::cout << "Hello World" << std::endl;
 return 0;
}
```

Anweisungen ohne einen Anweisungsblock (geschweifte Klammern) werden mit einem Semikolon abgeschlossen!

- **#include:** Befehl des Präprozessors (Teil des Compilers), den Bibliothekquellcode der dahinter angegebene Datei **iostream** einzufügen
- **using namespace std:** Den Namensraum **(=namespace) std** verwenden (so dass std::cout = cout und std::endl=endl)
- **main():** Hauptfunktion eines jeden Programms (durch geschweifte Klammern zusammengefasst). Gibt mit **return** den Rückgabewert 0 (vom Dateityp **int**) zurück.
- Anweisungsblock: **cout** = Ausgabe, **endl** = Zeile beenden

Klassischer Programmaufbau

Compileranweisung (z.B. #include)

Deklaration globaler Objekte und Funktionen

Hauptprogramm

```
int main()
{
    Anweisungen;
    return 0;
}
```

Funktionsdefinitionen

Variablen und Dateityp

- Daten basieren auf dem *binäre Zahlensystem* (G.W. Leibniz), so gilt z.B. für die ganze Zahl 117:

$$117_{(10)} = 01110101_{(2)} = 0 \cdot 2^7 + 1 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$$

- Allgemeine Form der Variablenvereinbarung:

  typ bezeichner1 [, bezeichner2, …] ;

Typ	Speicherbedarf in Byte	Inhalt	mögliche Werte
`char`	1	Character-Zeichen	'H', 'e', '\n'
`bool`	1	Booleanvariable	`false`, `true` [nicht in C]
`int`	4		-32767, c
`short [int]`	2	Ganze Zahlen	-32767, $[-2^{15}, -2^{15}-1]$
`long [int]`	4	(Integer)	wie `int`
`signed char`	1		-117, 67, $[-128, 127]$
`float`	4	Gleitkommazahlen	1.1, -1.56e-32
`double`	8	(floating point numbers)	1.1, -1.56e-32, 5.68e+287
`unsigned [int]`	4	Natürliche Zahlen	32767, 32769, $[0, 2^{32}-1]$
`long long [int]`	8	Ganze Zahlen	$[-2^{63}, -2^{63}-1]$
`long double`	12	Gleitkommazahlen	5.68e+287, 5.68e+420

Variablen und Dateityp

- Daten basieren auf dem *binäre Zahlensystem* (G.W. Leibniz), so gilt z.B. für die ganze Zahl 117:

$$117_{(10)} = 01110101_{(2)} = 0 \cdot 2^7 + 1 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$$

- Allgemeine Form der Variablenvereinbarung:

  typ bezeichner1 [, bezeichner2, …] ;

- Variablenbezeichner müssen gewissen Regeln folgen, z.B.:

Gültig	Ungültig	Grund
`i`		
`j`		
`ijl`		
`i3`	`3i`	3 ist kein Buchstabe
`_3a`	`_3*a`	* ist Operatorzeichen
`Drei_mal_a`	`b-a`	- ist Operatorzeichen
`auto1`	`auto`	`auto` ist Schlüsselwort

Operatoren

- Ein-& Ausleseoperatoren: >> & <<
  - Unter Verwendung der Ein- & Ausgabefunktionen **cin** & **cout**:

    cin >> x // Einlesen einer Zahl und Belegung

    cout << y // Ausgabe des Wertes der Variablen y
- Logische Operatoren:

  && (UND), z.B. **if(x<3 && y>4)** = wenn x<3 UND y>4;

  || (ODER), z.B. **if(x<3 || y>4)** = wenn x<3 ODER y>4;

  ! (Negation), z.B. **! (3>4)** = es gilt nicht, dass 3>4 ist

Operatoren

- Vergleichsoperatoren:

  > (größer), z.B. **b>a** (b größer als a);

  >= (größer oder gleich), z.B. **b>=a** (b größer als oder gleich a);

  < (kleiner), z.B. **b<a** (b kleiner als a);

  <= (kleiner oder gleich), z.B. **b<=a** (b kleiner als oder gleich a);

  == (gleich), z.B. **b==a** (b gleich a);

  != (ungleich), z.B. **b!=a** (b ungleich a);

- Zuweisungsoperatoren:

  = , z.B. **x=y** (der Variablen x wird der die Variable y zugewiesen)

  Achtung: **x=y** ist ein Ausdruck, erst **x=y;** ist eine Anweisung!

Mathematische Operatoren

- Arithmetische Operatoren:
  - **+** (Addition), z.B. **b + a**;
  - **-** (Subtraktion), z.B. **b - a**;
  - ***** (Multiplikation), z.B. **b * a**;
  - ***/*** (Division), z.B. **b / a** (*Achtung* bei Integers: **8 / 3** liefert **2**);
  - **%** (Rest bei ganzzahliger Division), z.B. **b % a**;
  - **+=** (Aufsummieren), z.B. **b += a** (entspricht **b = b + a**);
  - **-=, *=, /=, %=** entsprechend, z.B. **b *= a** (entspricht **b = b * a**);
- Inkrement- und Dekrementoperatoren:
  - **++** (Inkrementoperator): Variable wird um 1 erhöht;
  - **- -** (Dekrementoperator): Variable wird um 1 veringert;

Mathematische Operatoren

- Im *Headerfile* **cmath** werden diverse mathematische Funktionen und Konstanten bereitgestellt, wie z.B.

Funktion/Konstante	Beschreibung
`sqrt(x)`	Quadratwurzel von x: $\sqrt[2]{x}$ $(x \geq 0)$
`exp(x)`	e^x
`log(x)`	natürlicher Logarithmus von x: $\log_e x$ $(x > 0)$
`pow(x,y)`	Potenzieren ($x > 0$ falls y nicht ganzzahlig)
`fabs(x)`	Absolutbetrag von x: $\|x\|$
`fmod(x,y)`	realzahliger Rest von x/y $(y \neq 0)$
`ceil(x)`	nächste ganze Zahl $\geq x$
`floor(x)`	nächste ganze Zahl $\leq x$
`sin(x)`, `cos(x)`, `tan(x)`	trigonometrische Funktionen
`asin(x)`, `acos(x)`	trig. Umkehrfunktionen $(x \in [-1, 1])$
`atan(x)`	trig. Umkehrfunktion
`M_E`	Eulersche Zahl e
`M_PI`	π

Kontrollstrukturen (Verzweigungen)

- **if** - Verzweigung:

```
if( Ausdruck ) {
    Anweisung;
}
```

- Alternative **else** - Verzweigung:

```
if( Ausdruck ) {
    Anweisung1;
}
else {
    Anweisung2;
}
```

- **else if** - Verzweigung:

```
if( Ausdruck1 ) {
    Anweisung1;
}
else if ( Ausdruck2 )  {
    Anweisung2;
}
// weitere else if-Anweisungen möglich
// zusätzliche esle-Anweisung auch möglich
```

Hinweis: Falls nur eine Anweisung im Anweisungsblock existiert, können die geschweiften Klammern auch weggelassen werden!

Kontrollstrukturen (Verzweigungen)

- Fallunterscheidung **switch**:

  Zur Auswertung von ganzzahligen Ausdrücken (**char**-, **int**- oder **long**-Werte), mit folgender Syntax:

```
switch( Ausdruck ) {
    case Ausdruck1: Anweisung1; break;
    case Ausdruck2: Anweisung2; break;
    ...
    case AusdruckN: AnweisungN; break;
    default: Anweisung;                    //optional
}
```

Kontrollstrukturen (Verzweigungen)

- Fallunterscheidung **switch**:

  Zur Auswertung von ganzzahligen Ausdrücken (**char**-, **int**- oder **long**-Werte), mit folgender Syntax:

```
switch( Ausdruck ) {
    case Ausdruck1: Anweisung1; break;
    case Ausdruck2: Anweisung2; break;
    ...
    case AusdruckN: AnweisungN; break;
    default: Anweisung;                    //optional
}
```

break - Anweisung führt zum Sprung aus dem **switch** - Anweisungsblock

Kontrollstrukturen (Schleifen)

- **while** - Anweisung:

```
while( Ausdruck ) {
    // Abarbeiten der Anweisungen
}
```

- **do-while** - Anweisung:

```
do{
// Abarbeiten der Anweisungen
} while( Ausdruck )
```

- **for** - Anweisung:

```
for( Initialisierung(en); Ausdruck; Reinitialisierung(en)){
// Anweisungen
}
```

Kontrollstrukturen (Sprunganweisungen)

- **break** – Anweisung:

  Sprung aus der direkt umfassenden Schleife/ dem Anweisungsblock (oder der **switch**-Anweisung)

- **continue** – Anweisung:

  Beendent die aktuelle Schleifenausführung, d.h. hinter **continue** befindliche Anweisungen werden ausgelassen und es wird zum nächsten Schleifendurchlauf gesprungen.

- Zudem gibt es die Sprunganweisungen **return** und **exit**, die allerdings keine schleifentypischen Anweisungen sind.

Kontrollstrukturen (Beispiele)

```cpp
#include <iostream>
using namespace std;

int main(){
    int cnt=0;
    while(cnt<6){
        ++cnt;
        cout << "aktueller Zählerstand: " << cnt << "\n";
    }
    return 0;
}
```

Output:

```
aktueller Zählerstand: 1
aktueller Zählerstand: 2
aktueller Zählerstand: 3
aktueller Zählerstand: 4
aktueller Zählerstand: 5
aktueller Zählerstand: 6
```

Kontrollstrukturen (Beispiele)

```cpp
#include <iostream>
using namespace std;

int main(){
    int cnt=0;
    while(cnt<6){
        ++cnt;
        cout << "aktueller Zählerstand: " << cnt << "\n";
    }
    return 0;
}
```

Output:

```
aktueller Zählerstand: 1
aktueller Zählerstand: 2
aktueller Zählerstand: 3
aktueller Zählerstand: 4
aktueller Zählerstand: 5
aktueller Zählerstand: 6
```

```cpp
#include <iostream>
using namespace std;

int main(){
    int cnt=0;
    do{
        cnt += 1;
        cout << "aktueller Zählerstand: " << cnt << "\n"
    } while(cnt<6);
    return 0;
}
```

?

Kontrollstrukturen (Beispiele)

```cpp
#include <iostream>
using namespace std;

int main(){
    int cnt=0;
    while(cnt<6){
        ++cnt;
        cout << "aktueller Zählerstand: " << cnt << "\n";
    }
    return 0;
}
```

Output:

```
aktueller Zählerstand: 1
aktueller Zählerstand: 2
aktueller Zählerstand: 3
aktueller Zählerstand: 4
aktueller Zählerstand: 5
aktueller Zählerstand: 6
```

```cpp
#include <iostream>
using namespace std;

int main(){
    int cnt=0;
    do{
        cnt += 1;
        cout << "aktueller Zählerstand: " << cnt << "\n"
    } while(cnt<6);
    return 0;
}
```

```
aktueller Zählerstand: 1
aktueller Zählerstand: 2
aktueller Zählerstand: 3
aktueller Zählerstand: 4
aktueller Zählerstand: 5
aktueller Zählerstand: 6
```

Kontrollstrukturen (Beispiele)

```cpp
#include <iostream>
using namespace std;

int main(){
    int cnt=0;
    while(cnt<6){
        ++cnt;
        cout << "aktueller Zählerstand: " << cnt << "\n";
    }
    return 0;
}
```

Output:

```
aktueller Zählerstand: 1
aktueller Zählerstand: 2
aktueller Zählerstand: 3
aktueller Zählerstand: 4
aktueller Zählerstand: 5
aktueller Zählerstand: 6
```

```cpp
#include <iostream>
using namespace std;

int main(){
    int cnt=0;
    do{
        cnt += 1;
        cout << "aktueller Zählerstand: " << cnt << "\n"
    } while(cnt<6);
    return 0;
}
```

```
aktueller Zählerstand: 1
aktueller Zählerstand: 2
aktueller Zählerstand: 3
aktueller Zählerstand: 4
aktueller Zählerstand: 5
aktueller Zählerstand: 6
```

```cpp
#include <iostream>
using namespace std;

int main(){
    for(int cnt=0; cnt<6; ++cnt){
        cout << "aktueller Zählerstand: " << cnt << "\n";
    }
    return 0;
}
```

?

Kontrollstrukturen (Beispiele)

```cpp
#include <iostream>
using namespace std;

int main(){
    int cnt=0;
    while(cnt<6){
        ++cnt;
        cout << "aktueller Zählerstand: " << cnt << "\n";
    }
    return 0;
}
```

Output:

```
aktueller Zählerstand: 1
aktueller Zählerstand: 2
aktueller Zählerstand: 3
aktueller Zählerstand: 4
aktueller Zählerstand: 5
aktueller Zählerstand: 6
```

```cpp
#include <iostream>
using namespace std;

int main(){
    int cnt=0;
    do{
        cnt += 1;
        cout << "aktueller Zählerstand: " << cnt << "\n"
    } while(cnt<6);
    return 0;
}
```

```
aktueller Zählerstand: 1
aktueller Zählerstand: 2
aktueller Zählerstand: 3
aktueller Zählerstand: 4
aktueller Zählerstand: 5
aktueller Zählerstand: 6
```

```cpp
#include <iostream>
using namespace std;

int main(){
    for(int cnt=0; cnt<6; ++cnt){
        cout << "aktueller Zählerstand: " << cnt << "\n";
    }
    return 0;
}
```

```
aktueller Zählerstand: 0
aktueller Zählerstand: 1
aktueller Zählerstand: 2
aktueller Zählerstand: 3
aktueller Zählerstand: 4
aktueller Zählerstand: 5
```

Kontrollstrukturen (Beispiele)

```cpp
#include <iostream>
using namespace std;

int main(){
    for(int cnt=0; cnt<6; ++cnt){
        if(cnt<=3){
            cout << "(if) Zählerstand: " << cnt << "\n";
        }
        else if(cnt==3){
            cout << "(else if) Zählerstand: " << cnt << "\n";
        }
        else{
            cout << "(else) Zählerstand: " << cnt << "\n";
        }
    }
    return 0;
}
```

Output:

?

Kontrollstrukturen (Beispiele)

```cpp
#include <iostream>
using namespace std;

int main(){
    for(int cnt=0; cnt<6; ++cnt){
        if(cnt<=3){
            cout << "(if) Zählerstand: " << cnt << "\n";
        }
        else if(cnt==3){
            cout << "(else if) Zählerstand: " << cnt << "\n";
        }
        else{
            cout << "(else) Zählerstand: " << cnt << "\n";
        }
    }
    return 0;
}
```

Output:

```
(if) Zählerstand: 0
(if) Zählerstand: 1
(if) Zählerstand: 2
(if) Zählerstand: 3
(else) Zählerstand: 4
(else) Zählerstand: 5
```

Kontrollstrukturen (Beispiele)

```
1   #include <iostream>
2   using namespace std;
3
4   int main(){
5       for(int cnt=0; cnt<6; ++cnt){
6           if(cnt%2){
7               cout << "(if) Zählerstand: " << cnt << "\n";
8               break;
9           }
10          else{
11              cout << "(else) Zählerstand: " << cnt << "\n";
12          }
13      }
14      return 0;
15  }
16
```

Output:

?

Kontrollstrukturen (Beispiele)

```cpp
#include <iostream>
using namespace std;

int main(){
    for(int cnt=0; cnt<6; ++cnt){
        if(cnt%2){
            cout << "(if) Zählerstand: " << cnt << "\n";
            break;
        }
        else{
            cout << "(else) Zählerstand: " << cnt << "\n";
        }
    }
    return 0;
}
```

Output:

```
(else) Zählerstand: 0
(if) Zählerstand: 1
```

```cpp
#include <iostream>
using namespace std;

int main(){
    for(int cnt=0; cnt<6; ++cnt){
        if(cnt%2){
            continue;
        }
        else{
            cout << "Zählerstand: " << cnt << "\n";
        }
    }
    return 0;
}
```

?

Kontrollstrukturen (Beispiele)

```cpp
#include <iostream>
using namespace std;

int main(){
    for(int cnt=0; cnt<6; ++cnt){
        if(cnt%2){
            cout << "(if) Zählerstand: " << cnt << "\n";
            break;
        }
        else{
            cout << "(else) Zählerstand: " << cnt << "\n";
        }
    }
    return 0;
}
```

Output:

```
(else) Zählerstand: 0
(if) Zählerstand: 1
```

```cpp
#include <iostream>
using namespace std;

int main(){
    for(int cnt=0; cnt<6; ++cnt){
        if(cnt%2){
            continue;
        }
        else{
            cout << "Zählerstand: " << cnt << "\n";
        }
    }
    return 0;
}
```

```
Zählerstand: 0
Zählerstand: 2
Zählerstand: 4
```

Funktionen

- *Funktionen* sind kleine Unterprogramme, mit denen Daten verarbeitet oder Teilprobleme gelöst werden können
- Eine erste Funktion wurde bereits verwendet: die **main** – Funktion:
  - **Zwingend notwendig**
  - **Erste Funktion** die beim Programmstart aufgerufen wird
- Grundlegende Syntax einer Funktion:

```
[Spezifizierer]  Datentyp  Funktionsname( [Parameter] ){
        // Anweisung(en)
}
```

Funktionen

- *Funktionen* sind kleine Unterprogramme, mit denen Daten verarbeitet oder Teilprobleme gelöst werden können
- Eine erste Funktion wurde bereits verwendet: die **main** – Funktion:
  - Zwingend notwendig
  - Erste Funktion die beim Programmstart aufgerufen wird
- Grundlegende Syntax einer Funktion:

```
[Spezifizierer] Datentyp  Funktionsname( [Parameter] ){
      // Anweisung(en)
}
```

optionales Speicherklassenattribut, z.B. **extern** (Speicherobjekt steht im gesamten Programm zur Verfügung) oder **static** (Funktion ist nur in der aktuellen Quelldatei gültig)

Funktionen

- *Funktionen* sind kleine Unterprogramme, mit denen Daten verarbeitet oder Teilprobleme gelöst werden können
- Eine erste Funktion wurde bereits verwendet: die **main** – Funktion:
  - Zwingend notwendig
  - Erste Funktion die beim Programmstart aufgerufen wird
- Grundlegende Syntax einer Funktion:

```
[Spezifizierer] Datentyp Funktionsname( [Parameter] ){
    // Anweisung(en)
}
```

Typ des Rückgabewertes. Falls sie keinen Rückgabewert ausgibt: **void** verwenden

Funktionen

- *Funktionen* sind kleine Unterprogramme, mit denen Daten verarbeitet oder Teilprobleme gelöst werden können
- Eine erste Funktion wurde bereits verwendet: die **main** – Funktion:
  - Zwingend notwendig
  - Erste Funktion die beim Programmstart aufgerufen wird
- Grundlegende Syntax einer Funktion:

```
[Spezifizierer]  Datentyp Funktionsname( [Parameter] ){
        // Anweisung(en)
}
```

Eindeutiger Name (es gelten diesselben Regeln wie bei Variablennamen), mit dem sich die Funktion von einer anderen Programmstelle aus aufrufen lässt.

Funktionen

- *Funktionen* sind kleine Unterprogramme, mit denen Daten verarbeitet oder Teilprobleme gelöst werden können
- Eine erste Funktion wurde bereits verwendet: die **main** – Funktion:
  - Zwingend notwendig
  - Erste Funktion die beim Programmstart aufgerufen wird
- Grundlegende Syntax einer Funktion:

```
[Spezifizierer]  Datentyp  Funktionsname( [Parameter] ){
        // Anweisung(en)
}
```

Optionale Parameter der Funkion; die Klammern **()** sind aber nicht optional

Funktionen

- *Funktionen* sind kleine Unterprogramme, mit denen Daten verarbeitet oder Teilprobleme gelöst werden können
- Eine erste Funktion wurde bereits verwendet: die **main** – Funktion:
  - Zwingend notwendig
  - Erste Funktion die beim Programmstart aufgerufen wird
- Grundlegende Syntax einer Funktion:

```
[Spezifizierer]  Datentyp  Funktionsname( [Parameter] ){
    // Anweisung(en)
}
```

Anweisungsblock mit Funktionsanweisungen und *lokalen* Deklarationen

Funktionen

- Funktionen aufrufen:
  - Vor dem ersten Aufruf (**main**) *definieren*; oder

```
1   #include <iostream>
2   using namespace std;
3
4   void myFunc(){
5       cout << "Ich bin myFunc() \n";
6   }
7
8   int main(){
9       cout << "Vor dem Funktionsaufruf \n";
10      myFunc();
11      cout << "Nach dem Funktionsaufruf \n";
12      return 0;
13  }
14
```

  - Vor dem ersten Aufruf (**main**) *deklarieren* (d.h. nur Funktionskopf ohne Anweisungsblock – mit Semikolon!), und an beliebiger Stelle *definieren*

```
1   #include <iostream>
2   using namespace std;
3
4   void myFunc();
5
6   int main(){
7       cout << "Vor dem Funktionsaufruf \n";
8       myFunc();
9       cout << "Nach dem Funktionsaufruf \n";
10      return 0;
11  }
12
13  void myFunc(){
14      cout << "Ich bin myFunc() \n";
15  }
16
```

Höhere Datentypen (Zeiger)

- *Zeiger (Pointer)* stellen lediglich die Adresse und den Datentyp auf ein Speicherobjekt dar.
  - *Deklaration* eines *Zeigers:*

    Datentyp *Name;
  - Mit dem *Adressoperator* **&** lässt sich dem *Zeiger* die Adresse einer Variablen zuweisen.
  - Mit dem *Indirektionsoperator* * lässt sich über den Zeiger der Wert auslesen oder ändern, auf den er verweist.
  - Der Null-Zeiger **0, 0L, nullptr** (letzteres ab C++11)
- *Zeiger* können den Programmablauf beschleunigen oder helfen Speicherplatz zu sparen

Höhere Datentypen (Zeiger)

…ein Beispiel:

```
1   #include <iostream>
2   using namespace std;
3
4   int main(){
5       int ival = 123;
6       int *iptr = 0L;
7       //int *iptr{nullptr};      // ab C++11 um Zeideutigkeit zu vermeiden
8       iptr = &ival;
9       cout << "Adresse von ival: " << &ival << " \n";
10      cout << "iptr verweist auf: = " << iptr << " \n";
11      cout << "Adresse von iptr: " << &iptr << " \n";
12      cout << "*iptr = " << *iptr << " \n";
13      cout << "ival = " << ival << " \n";
14      // ival indirekt einen neuen Wert zuweisen:
15      *iptr = 256;
16      cout << "*iptr = " << *iptr << " \n";
17      cout << "ival = " << ival << " \n";
18      return 0;
19  }
20
```

Output:

```
Adresse von ival: 0x7ffcb3cb9d3c
iptr verweist auf: = 0x7ffcb3cb9d3c
Adresse von iptr: 0x7ffcb3cb9d30
*iptr = 123
ival = 123
*iptr = 256
ival = 256
```

Höhere Datentypen (Referenzen)

- Eine *Referenz* ist ein anderer Name *(Alias)* für ein Speicherobjekt.
  - *Initialisierung* einer *Referenz* mit bestehendem Objekt:

    Datentyp &Name = Objekt;
  - Können keine Adressen von anderen Speicherobjekten aufnehmen (→← Zeiger).
- Anwendungsgebiet von *Referenzen* liegt vorwiegend in der Parameterübergabe und Wertrückgabe von Funktionen.

Höhere Datentypen (Referenzen Beispiele)

```cpp
#include <iostream>
using namespace std;

int main(){
    int var = 0;
    int &rvar = var;
    var = 123;
    cout << "rvar = " << rvar << " \n";
    rvar = 256;
    cout << "rvar = " << rvar << " \n";
    cout << "var = " << var << " \n";
    cout << "&rvar = " << &rvar << " \n";
    cout << "&var = " << &var << " \n";
    return 0;
}
```

Output:

?

Höhere Datentypen (Referenzen Beispiele)

```cpp
#include <iostream>
using namespace std;

int main(){
    int var = 0;
    int &rvar = var;
    var = 123;
    cout << "rvar = " << rvar << " \n";
    rvar = 256;
    cout << "rvar = " << rvar << " \n";
    cout << "var = " << var << " \n";
    cout << "&rvar = " << &rvar << " \n";
    cout << "&var = " << &var << " \n";
    return 0;
}
```

Output:

```
rvar = 123
rvar = 256
var = 256
&rvar = 0x7fff2bbb5b74
&var = 0x7fff2bbb5b74
```

```cpp
#include <iostream>
using namespace std;

int main(){
    int var1 = 1;
    int var2 = 2;
    int& rvar = var1;
    cout << "(1.) rvar = " << rvar << " \n";
    rvar = var2;
    cout << "(2.) rvar = " << rvar << " \n";
    cout << "var1 = " << var1 << " \n";
    var2 = 20;
    cout << "(3.) rvar = " << rvar << " \n";
    return 0;
}
```

?

Höhere Datentypen (Referenz Beispiele)

```cpp
#include <iostream>
using namespace std;

int main(){
    int var = 0;
    int &rvar = var;
    var = 123;
    cout << "rvar = " << rvar << " \n";
    rvar = 256;
    cout << "rvar = " << rvar << " \n";
    cout << "var = " << var << " \n";
    cout << "&rvar = " << &rvar << " \n";
    cout << "&var = " << &var << " \n";
    return 0;
}
```

Output:

```
rvar = 123
rvar = 256
var = 256
&rvar = 0x7fff2bbb5b74
&var = 0x7fff2bbb5b74
```

```cpp
#include <iostream>
using namespace std;

int main(){
    int var1 = 1;
    int var2 = 2;
    int& rvar = var1;
    cout << "(1.) rvar = " << rvar << " \n";
    rvar = var2;
    cout << "(2.) rvar = " << rvar << " \n";
    cout << "var1 = " << var1 << " \n";
    var2 = 20;
    cout << "(3.) rvar = " << rvar << " \n";
    return 0;
}
```

```
(1.) rvar = 1
(2.) rvar = 2
var1 = 2
(3.) rvar = 2
```

Höhere Datentypen (Array/ Vektor)

- Ein *Array* (oder *Vektor*) stellt eine Reihe von Elementen desselben Datentyps dar.
- Dabei gibt es zwei Möglichkeiten:
  - Die **vector** – Containerklasse (komfortabler, weniger fehleranfällig) werden definiert mit der Syntax:

    vector<Datentyp> Bezeichner (Anzahl_der_Elemente);
  - Die klassischen *C-Arrays* (eingeschränktere Anwendung) werden definiert mit der Syntax:

    Datentyp Bezeichner[Anzahl_der_Elemente];

➢ Wenn möglich besser die C++ - Containerklasse **vector** als die klassischen C-Arrays verwenden!

Höhere Datentypen (Array/ Vektor)

- Zugriff auf ein einzelnes Element des *Vektor* mit dem *Indizierungsoperator* **[ ]** oder die Elementfunktion (nur bei **vector** Objekten!) **at()** verwenden.
  - *Achtung:* erstes Element hat den Index 0.
- Weitere hilfreiche Methoden (nur für **vector** Objekte!):
  - Größe des Arrays ändern: **resize()**
  - Anzahl der Elemente ausgeben: **size()**
  - Neues Element ans Ende anhängen: **push_back()**
  - Erste bzw. letzte Element im Vektor: **front()** bzw. **back()**
  - Letzte Element löschen: **pop_back()**

Höhere Datentypen (Array)

…ein Beispiel:

```
1   #include <iostream>
2   #include <vector>
3   using namespace std;
4
5   int main(){
6       vector<int> ivec(10);
7       cout << "Anzahl der Elemente: " << ivec.size() << endl;
8       ivec[0] = 123;
9       ivec.at(1) = 234;
10      ivec.push_back(345);
11
12      // Array um 2 Elemente vergrößern:
13      ivec.resize(ivec.size()+2);
14
15      cout << "Wert für das 7. Element: ";
16      cin >> ivec.at(6);
17
18      // Alle Elemente ausgeben:
19      for(size_t i=0; i < ivec.size(); i++){
20          cout << i+1 <<". Element : " << ivec[i] << "\n";
21      }
22      return 0;
23  }
24
```

Output:

```
Anzahl der Elemente: 10
Wert für das 7. Element: 777
```

```
Anzahl der Elemente: 10
Wert für das 7. Element: 777
1. Element : 123
2. Element : 234
3. Element : 0
4. Element : 0
5. Element : 0
6. Element : 0
7. Element : 777
8. Element : 0
9. Element : 0
10. Element : 0
11. Element : 345
12. Element : 0
13. Element : 0
```

Höhere Datentypen (Array)

…ein Beispiel:

```
1   #include <iostream>
2   #include <vector>
3   using namespace std;
4
5   int main(){
6       vector<int> ivec(10);
7       cout << "Anzahl der Elemente: " << ivec.size() << endl;
8       ivec[0] = 123;
9       ivec.at(1) = 234;
10      ivec.push_back(345);
11
12      // Array um 2 Elemente vergrößern:
13      ivec.resize(ivec.size()+2);
14
15      cout << "Wert für das 7. Element: ";
16      cin >> ivec.at(6);
17
18      // Alle Elemente ausgeben:
19      for(size_t i=0; i < ivec.size(); i++){
20          cout << i+1 <<". Element : " << ivec[i] << "\n";
21      }
22      return 0;
23  }
24
```

size_t = Kurzform von **unsigned int**

Output:

```
Anzahl der Elemente: 10
Wert für das 7. Element: 777
```

```
Anzahl der Elemente: 10
Wert für das 7. Element: 777
1. Element : 123
2. Element : 234
3. Element : 0
4. Element : 0
5. Element : 0
6. Element : 0
7. Element : 777
8. Element : 0
9. Element : 0
10. Element : 0
11. Element : 345
12. Element : 0
13. Element : 0
```

Höhere Datentypen (String)

- Die **string** – Containerklasse ermöglicht die Darstellung von Zeichenketten, welche sich aus Zeichen des Typs **char** zusammensetzen.
  - *Initialisierung* eines *String*:

    string Bezeichner(“Zeichenkette“);
  - Ganze Zeile (d.h. bis zum nächsten Newline-Zeichen) mit **getline()** einlesen.
  - Viele weitere hilfreiche Methoden: **append(), assign(), replace(),…** (siehe: http://www.cplusplus.com/reference/string/string/)

Höhere Datentypen (String)

…ein Beispiel:

```
1   #include <iostream>
2   #include <string>
3   using namespace std;
4
5   int main(){
6       string str1("String Nr. 1");
7       string str2 = str1;
8       string str3;
9
10      cout << "String eingeben: ";
11      getline(cin, str3);
12
13      cout << "str1: " << str1 << endl;
14      cout << "str2: " << str2 << endl;
15      cout << "str3: " << str3 << endl;
16
17      str1.assign(str3); // str1 durch str3 ersetzen
18      str2.append(str1); // str1 an str2 anhängen
19      // Letzte Zeichen von str3 durch "1" ersetzen:
20      str3.replace(str3.size()-1,1, "1");
21
22      cout << "str1: " << str1 << endl;
23      cout << "str2: " << str2 << endl;
24      cout << "str3: " << str3 << endl;
25      // Anzahl der Zeichen im str3 ausgeben:
26      cout << "Zeichen in str3: " << str3.size() << endl;
27
28      return 0;
29  }
30
```

Output:

```
String eingeben: String Nr. 3
```

```
String eingeben: String Nr. 3
str1: String Nr. 1
str2: String Nr. 1
str3: String Nr. 3
str1: String Nr. 3
str2: String Nr. 1String Nr. 3
str3: String Nr. 1
Zeichen in str3: 12
```

Präprozessor-Direktiven

- *Bevor der Compiler das C++-Programm übersetzt*, wird der ***Präprozessor*** aktiv.
- Dieser führt **rein textuelle Manipulationen** am C++-Quelltext durch
- Spezielle an den Präprozessor gedachte Zeilen, sog. (*Präprozessor-) Direktiven,* beginnen immer mit einem Hash-Zeichen #
- *Achtung:* Direktiven enden nicht mit einem Semikolon.
- **Drei** unterschiedliche **Arten** von *Direktiven*:
  - Header- und Quelldateien in den Quelltext einbinden (**#include**),
  - symbolische Konstanten/ Makros einbinden (**#define**),
  - bedingte Kompilierung (**#ifdef, #elseif, …**).

Präprozessor-Direktiven (#include)

- Einfügen einer Dateien (z.B. iostream) aus dem *include*-Verzeichnis:
  #include <datei>
- Einfügen der lokalen *Headerdateien* datei.h:
  #include “datei.h“

- **Ein guter Stil** besagt, dass man
  - den Code in .cpp – Dateien schreibt, und
  - den Rest (Definitionen, Deklarationen, Makros, Konstante, u.ä.) in Headerdateien schreibt.

Präprozessor-Direktiven

- Mehrfaches Inkludieren vermeiden:
  - *Bedingte Kompilierung*, um zu überprüfen ob eine Headerdatei bereits inkludiert ist.
  - Beim Erstellen einer Headerdatei sollte man immer ergänzen:

```
#ifndef MYHEADER_H_
```

Makro (üblicherweise in Großbuchstaben)

(Falls MY_HEADER_H_ nicht definiert ist, wird der folgende Programmteil ausgeführt)

```
#define MYHEADER_H_
```

(Definiert den Makro MY_HEADER_H_)

```
// Quellcode für die Headerdatei myheader.h
#endif
```

(Zeigt das Ende der bedingten Kompilierung an)

Präprozessor-Direktiven

- Mehrfaches Inkludieren vermeiden:
  - *Bedingte Kompilierung*, um zu überprüfen ob eine Headerdatei bereits inkludiert ist.
  - Beim Erstellen einer Headerdatei sollte man immer ergänzen:

```
#ifndef MYHEADER_H_
#define MYHEADER_H_
        // Quellcode für die Headerdatei myheader.h
#endif
```

Beim **1.** Inkludieren: Das *Makro* MYHEADER_H_ ist noch nicht definiert, so dass der Inhalt zwischen #ifndef und #endif eingefügt wird (wobei auch das Makro MYHEADER_H_ erzeugt wird).

Präprozessor-Direktiven

- Mehrfaches Inkludieren vermeiden:
  - *Bedingte Kompilierung*, um zu überprüfen ob eine Headerdatei bereits inkludiert ist.
  - Beim Erstellen einer Headerdatei sollte man immer ergänzen:

```
#ifndef MYHEADER_H_
#define MYHEADER_H_
        // Quellcode für die Headerdatei myheader.h
#endif
```

Beim **2.** Inkludieren: Das *Makro* MYHEADER_H_ ist dem Präprozessor bekannt, so dass #ifndef falsch ist und alles zwischen #ifndef und #endif ignoriert wird.

Klassen

- In *Klassen* lassen sich **Elemente (*Members*) fast beliebigen Typs zusammenfassen**.
- Mitglieder einer *Klasse* sind die Daten (***Attribute***) und die ***Elementfunktionen(en).***
- Klassennamen müssen eindeutig sein, und üblicherweise verwendet man einen *großen Anfangsbuchstaben*.
- **Zugriffskontrolle** mit **public** und **privat**:
  - **privat** (Daten und Elementfunktionen können nur innerhalb der Klasse verwendet werden.)
  - **public** (Zugriff auf Daten und Elementfunktionen von außerhalb der Klasse ermöglichen.)
  - Ohne Angabe der Zugriffsart ist immer alles **privat**.
  - Üblicherweise sind die Daten einer Klasse **privat**.

Klassen

- Allgemeine Definition einer *Klasse:*

```
class Klassenname {
        privat:
                // private Daten und Elementfunktionen;
        public:
                // öffentlich Daten und Elementfunktionen;
};
```

Klassen

- Allgemeine Definition einer *Klasse:*

```
class Klassenname {
    privat:
        // private Daten und Elementfunktionen;
    public:
        // öffentliche Daten und Elementfunktionen;
};
```

Klassen mit *Semikolon* abschließen - sind schließlich Deklarationen eines (zusammengesetzten Datentyps)!

Klassen

- Objekte erzeugen und benutzen:
  - Die definierte Klasse ist zunächst nur „reine Vorstellung“, d.h. sie müssen noch ein Objekt (***Instanz***) der zuvor definierten Klasse erzeugen/ ***deklarieren***:

    Klassenname Objekt;
  - *Direkter Zugriff* (auf die public Elemente) mit dem *Punktoperator*:

    Objekt.Daten = Datenwert; (nicht zu empfehlen!)

    Objekt.Elementfunktion(Parameter);
    - Mit einer Elementfunktion init() die Daten einer Klasse initialisieren:

    Objekt.init(Datenwert1[, Datenwert2,…]);
  - Elementfunktionen definieren:

    Typ Klassenname::Funktionsname(Parameter){

    \\ Anweisungen der Funktion

    }

Klassen

- Objekte erzeugen und benutzen:
  - Die definierte Klasse ist zunächst nur „reine Vorstellung“, d.h. sie müssen noch ein Objekt (*Instanz*) der zuvor definierten Klasse erzeugen/ *deklarieren*:

    Klassenname Objekt;
  - ***Direkter Zugriff*** (auf die public Elemente) mit dem *Punktoperator*:

    Objekt.Daten = Datenwert; (nicht zu empfehlen!)

    Objekt.Elementfunktion(Parameter);
    - Mit einer Elementfunktion init() die Daten einer Klasse initialisieren:

    Objekt.init(Datenwert1[, Datenwert2,…]);
  - Elementfunktionen definieren:

    Typ Klassenname::Funktionsname(Parameter){

    \\ Anweisungen der Funktion

    }

Skope-Operator (Bereichsoperator) um dem Compiler mitzuteilen, dass die Funktion **Funktionsname** eine Elementfunktion der Klasse **Klassenname** ist.

Klassen

- Objekte erzeugen und benutzen:
  - ***Indirekter Zugriff*** mit dem *Objektzeiger* und dem *Pfeiloperator*:

    Klassenname Objekt;

    Klassenname *ObjektPtr;

    ObjektPtr = &Objekt; (gültige Adresse an ObjektPtr zuweisen)

    ObjektPtr -> Elementfunktion(Parameter); (ind. Zugriff auf Elementfunktion)

    ODER:

    Klassenname *ObjektPtr;

    ObjektPtr = new Klassenname; (Speicher zur Laufzeit anfordern)

    ObjektPtr -> Elementfunktion(Parameter); (ind. Zugriff auf Elementfunktion)

Klassen

- Objekte erzeugen und benutzen:
  - ***Indirekter Zugriff*** mit dem *Objektzeiger* und dem *Pfeiloperator*:

Achtung: Der mit dem **new** - Operator angeforderte dynamische Speicher muss anschließend wieder mit dem **delete** – Operator freigegeben werden.

Falls möglich die in C++11 eingeführten *Smart-Pointer*
unique_ptr<Klassenname> ObjektPtr(new Klassenname)
(sind in <memory> definiert, d.h. entsprechenden Header *includen*!)
verwenden, da diese sich selbst um die Freigabe des Speichers kümmern.

Klassenname *ObjektPtr;

ObjektPtr = new Klassenname; (Speicher zur Laufzeit anfordern)

ObjektPtr -> Elementfunktion(Parameter); (ind. Zugriff auf Elementfunktion)

Klassen (Beispiel)

Wir wollen nun die Klasse Automat schaffen, welche neben den Daten (geld, standort), Elementfunktionen besitzt, die diese modifizieren.

- Die Klassendefinition steht dabei in der Headerdatei automat.h
- Elementfunktionen werden außerhalb definiert in automat.cpp

automat.h

```
1  #ifndef _AUTOMAT_H_
2  #define _AUTOMAT_H_
3  #include <string>
4  using namespace std;
5
6  class Automat{
7      private:
8      //Daten:
9      unsigned long geld;
10     string standort;
11
12     public:
13     //Elementfunktionen:
14     unsigned long get_geld();
15     void set_geld(unsigned long g);
16     string get_standort();
17     void set_standort(string s);
18     void init(unsigned long g, string s);
19 };
20 #endif
21
```

Klassen (Beispiel)

Wir wollen nun die Klasse Automat schaffen, welche neben den Daten (geld, standort), Elementfunktionen besitzt, die diese modifizieren.

- Die Klassendefinition steht dabei in der Headerdatei automat.h
- Elementfunktionen werden außerhalb definiert in automat.cpp

automat.h

```
1  #ifndef _AUTOMAT_H_
2  #define _AUTOMAT_H_
3  #include <string>
4  using namespace std;
5
6  class Automat{
7      private:
8      //Daten:
9      unsigned long geld;
10     string standort;
11
12     public:
13     //Elementfunktionen:
14     unsigned long get_geld();
15     void set_geld(unsigned long g);
16     string get_standort();
17     void set_standort(string s);
18     void init(unsigned long g, string s);
19 };
20 #endif
21
```

automat.cpp

```
1  #include <string>
2  #include "automat.h"
3  using namespace std;
4
5  unsigned long Automat::get_geld(){
6      return geld;
7  }
8
9  void Automat::set_geld(unsigned long g){
10     geld = g;
11 }
12
13 string Automat::get_standort(){
14     return standort;
15 }
16
17 void Automat::set_standort(string s){
18     standort = s;
19 }
20
21 void Automat::init(unsigned long g=0, string s=""){
22     set_geld(g);
23     set_standort(s);
24 }
25
```

Klassen (Beispiel)

Verwendung der Klasse Automat durch **direkten Zugriff** (*1. Variante*):

```
1   #include <iostream>
2   #include <vector>
3   #include <string>
4   #include "automat.h"
5   #include "automat.cpp"
6   using namespace std;
7
8   int main(void){
9       Automat device;
10      device.init(10000, "Bochum, Universitaetsstrasse 70");
11
12      cout << "Standort: " << device.get_standort() << endl;
13      cout << "Aktueller Kontostand: " << device.get_geld() << endl;
14
15      unsigned long abhebung;
16      cout << "Wieviel Geld möchten Sie abheben: ";
17      cin >> abhebung;
18      device.set_geld(device.get_geld()-abhebung);
19      cout << "Aktueller Kontostand: " << device.get_geld() << endl;
20
21      return 0;
22  }
23
```

Output:

```
Standort: Bochum, Universitaetsstrasse 70
Aktueller Kontostand: 10000
Wieviel Geld möchten Sie abheben: 50
Aktueller Kontostand: 9950
```

Klassen (Beispiel)

Verwendung der Klasse Automat durch **indirekten Zugriff** (*2. Variante*):

```
1  #include <iostream>
2  #include <vector>
3  #include <string>
4  #include "automat.h"
5  #include "automat.cpp"
6  using namespace std;
7
8  int main(void){
9      Automat *device_ptr = new Automat;
10     device_ptr -> init(10000, "Bochum, Universitaetsstrasse 70");
11
12     cout << "Standort: " << device_ptr -> get_standort() << endl;
13     cout << "Aktueller Kontostand: " << device_ptr -> get_geld() << endl;
14
15     unsigned long abhebung;
16     cout << "Wieviel Geld möchten Sie abheben: ";
17     cin >> abhebung;
18     device_ptr -> set_geld(device_ptr->get_geld()-abhebung);
19     cout << "Aktueller Kontostand: " << device_ptr -> get_geld() << endl;
20
21     delete device_ptr;
22     return 0;
23 }
24
```

Output:

```
Standort: Bochum, Universitaetsstrasse 70
Aktueller Kontostand: 10000
Wieviel Geld möchten Sie abheben: 100
Aktueller Kontostand: 9900
```

Klassen (Beispiel)

Verwendung der Klasse Automat durch **indirekten Zugriff** (*2. Variante*):

```
1  #include <iostream>
2  #include <vector>
3  #include <string>
4  #include "automat.h"
5  #include "automat.cpp"
6  using namespace std;
7
8  int main(void){
9      Automat *device_ptr = new Automat;
10     device_ptr -> init(10000, "Bochum, Universitaetsstrasse 70");
11
12     cout << "Standort: " << device_ptr -> get_standort() << endl;
13     cout << "Aktueller Kontostand: " << device_ptr -> get_geld() << endl;
14
15     unsigned long abhebung;
16     cout << "Wieviel Geld möchten Sie abheben: ";
17     cin >> abhebung;
18     device_ptr -> set_geld(device_ptr->get_geld()-abhebung);
19     cout << "Aktueller Kontostand: " << device_ptr -> get_geld() << endl;
20
21     delete device_ptr;
22     return 0;
23 }
24
```

Output:

```
Standort: Bochum, Universitaetsstrasse 70
Aktueller Kontostand: 10000
Wieviel Geld möchten Sie abheben: 100
Aktueller Kontostand: 9900
```

Klassen (Beispiel)

Verwendung der Klasse Automat durch **indirekten Zugriff** (*3. Variante*):

```
1  #include <iostream>
2  #include <vector>
3  #include <string>
4  #include <memory>
5  #include "automat.h"
6  #include "automat.cpp"
7  using namespace std;
8
9  int main(void){
10     std::unique_ptr<Automat> device_ptr(new Automat); // ab C++11 möglich
11     device_ptr -> init(10000, "Bochum, Universitaetsstrasse 70");
12
13     cout << "Standort: " << device_ptr -> get_standort() << endl;
14     cout << "Aktueller Kontostand: " << device_ptr -> get_geld() << endl;
15
16     unsigned long abhebung;
17     cout << "Wieviel Geld möchten Sie abheben: ";
18     cin >> abhebung;
19     device_ptr -> set_geld(device_ptr->get_geld()-abhebung);
20     cout << "Aktueller Kontostand: " << device_ptr -> get_geld() << endl;
21
22     return 0;
23 }
```

Output:

```
Standort: Bochum, Universitaetsstrasse 70
Aktueller Kontostand: 10000
Wieviel Geld möchten Sie abheben: 200
Aktueller Kontostand: 9800
```

Klassen (Konstruktor)

- Bisher noch keine „sichere“ Initialisierung von Variablen!
  (Zugriff auf undefinierten Wert möglich)
- Dazu sind ***Konstruktoren,*** die bei der Definition des Objekts die Daten mit gültigen Werten belegen.
  - Der Name des Konstruktor ist derselbe wie der Name der Klasse
  - Der Konstruktor besitzt keinen Rückgabewert
- *Konstruktoren* **deklarieren**: Im public Bereich der Klasse
  (Funktionsüberladung möglich!)
- *Konstruktoren* **definieren**:

```
Klassenname::Klassenname( Parameter ){
    // Daten von Klassenname initialisieren
}
```

Klassen (Konstruktor - Beispiel)

Zurück zur Klasse Automat unter Verwendung von Konstruktoren:

automat2.h

```
1  #ifndef _AUTOMAT2_H_
2  #define _AUTOMAT2_H_
3  #include <string>
4  using namespace std;
5
6  class Automat{
7      private:
8      //Daten:
9      unsigned long geld;
10     string standort;
11
12     public:
13     //Deklaration der Konstruktoren:
14     Automat(unsigned long, string);
15     Automat(unsigned long);
16     Automat(string);
17     Automat();
18     //Elementfunktionen:
19     unsigned long get_geld();
20     void set_geld(unsigned long g);
21     string get_standort();
22     void set_standort(string s);
23 };
24 #endif
25
```

automat2.cpp

```
1  #include <string>
2  #include "automat2.h"
3  using namespace std;
4
5  // Definition der Konstruktoren:
6  Automat::Automat(unsigned long g, string s){
7      set_geld(g);
8      set_standort(s);
9  }
10 Automat::Automat(unsigned long g){
11     set_geld(g);
12     set_standort("");
13 }
14 Automat::Automat(string s){
15     set_geld(0);
16     set_standort(s);
17 }
18 Automat::Automat(){
19     set_geld(0);
20     set_standort("");
21 }
22
23 // Definition der Elementfunktionen:
24 unsigned long Automat::get_geld(){
25     return geld;
26 }
27 void Automat::set_geld(unsigned long g){
28     geld = g;
29 }
30 string Automat::get_standort(){
31     return standort;
32 }
33 void Automat::set_standort(string s){
34     standort = s;
35 }
```

Klassen (Konstruktor - Beispiel)

Zurück zur Klasse Automat unter Verwendung von Konstruktoren:

```
1   #include <iostream>
2   #include <vector>
3   #include <string>
4   #include "automat2.h"
5   #include "automat2.cpp"
6   using namespace std;
7
8   int main(void){
9       Automat device1(10000, "Bochum, Universitaetsstrasse 70");
10      Automat device2("Bochum, Huestrasse 13");
11      Automat device3{5000};          // ab C++11 möglich
12
13      cout << "(1.) Standort: " << device1.get_standort() << endl;
14      cout << "(1.) Aktueller Kontostand: " << device1.get_geld() << endl;
15
16      cout << "(2.) Standort: " << device2.get_standort() << endl;
17      cout << "(2.) Aktueller Kontostand: " << device2.get_geld() << endl;
18
19      cout << "(3.) Standort: " << device3.get_standort() << endl;
20      cout << "(3.) Aktueller Kontostand: " << device3.get_geld() << endl;
21
22      return 0;
23  }
24
```

Output:

```
(1.) Standort: Bochum, Universitaetsstrasse 70
(1.) Aktueller Kontostand: 10000
(2.) Standort: Bochum, Huestrasse 13
(2.) Aktueller Kontostand: 0
(3.) Standort:
(3.) Aktueller Kontostand: 5000
```

Klassen (Konstruktor)

- Als ***Standardkonstruktor*** *(Default-Konstruktor)* bezeichnet man einen *Konstruktor ohne Parameter*
- Als ***Kopierkonstruktor*** *(Copy-Konstruktor)* bezeichnet man einen speziellen Konstruktor, der die Aufgabe hat eine *Kopie des Objektes* zu erstellen
  - Wichtig bei dynamischen Klassenelementen
  - Ansonsten kann auch der Zuweisungsoperator (=) verwendet werden (z.B.: **Automat device2 = device1;**)
  - Deklaration: **Klassenname (const Klassenname &);**
  - Definition, z.B.:

```
Automat::Automat(const Automat &a){
        geld = a.geld;
        standort = a.standort;
}
```

Klassen (Destruktor)

- Der ***Destruktor,*** um alles wieder zu „zerstören“:
  - Freigabe von dynamischem Speicher, Aufheben von Sperren, Dateifreigaben, usw.
- *Destruktor* **deklarieren**: Ähnlich wie der Konstruktor, nur mit einem vorangestellten *Komplement-Zeichen* **~**:

```
~Klassenname();
```

- *Destruktor* **definieren**:

```
Klassenname::~Klassenname(){
        // Anweisungen
}
```

- *Destruktor* sollten insbesondere bei der Verwendung von *Kopierkonstruktoren* oder *Zuweisungsoperatoren* ebenfalls implementiert werden („The Big Three“)

Klassen (Destruktor - Beispiel)

Zurück zur Klasse Automat unter Verwendung von Destruktoren:

automat3.h

```
1  #ifndef _AUTOMAT3_H_
2  #define _AUTOMAT3_H_
3  #include <string>
4  using namespace std;
5
6  class Automat{
7      private:
8      //Daten:
9      unsigned long geld;
10     string standort;
11
12     public:
13     //Deklaration der Konstruktoren:
14     Automat(unsigned long, string);
15     Automat(unsigned long);
16     Automat(string);
17     Automat();
18     //Deklaration des Destruktors:
19     ~Automat();
20     //Elementfunktionen:
21     unsigned long get_geld();
22     void set_geld(unsigned long g);
23     string get_standort();
24     void set_standort(string s);
25 };
26 #endif
27
```

automat3.cpp

```
1  #include <string>
2  #include "automat3.h"
3  using namespace std;
4
5  // Definition der Konstruktoren:
6  Automat::Automat(unsigned long g, string s){
7      set_geld(g);
8      set_standort(s);
9  }
10 Automat::Automat(unsigned long g){
11     set_geld(g);
12     set_standort("");
13 }
14 Automat::Automat(string s){
15     set_geld(0);
16     set_standort(s);
17 }
18 Automat::Automat(){
19     set_geld(0);
20     set_standort("");
21 }
22 // Definition der Destruktors:
23 Automat::~Automat(){
24     cout << get_standort() << " entfernt" << endl;
25 }
26
27 // Definition der Elementfunktionen:
28 unsigned long Automat::get_geld(){
29     return geld;}
30 void Automat::set_geld(unsigned long g){
31     geld = g;}
32 string Automat::get_standort(){
33     return standort;}
34 void Automat::set_standort(string s){
35     standort = s;}
```

Klassen (Destruktor - Beispiel)

Zurück zur Klasse Automat unter Verwendung von Destruktoren:

automat3.h

```
1  #ifndef _AUTOMAT3_H_
2  #define _AUTOMAT3_H_
3  #include <string>
4  using namespace std;
5
6  class Automat{
7      private:
8      //Daten:
9      unsigned long geld;
10     string standort;
11
12     public:
13     //Deklaration der Konstruktoren:
14     Automat(unsigned long, string);
22     void set_geld(unsigned long g);
23     string get_standort();
24     void set_standort(string s);
25 };
26 #endif
27
```

Die Textausgabe dient in diesem Beispiel nur zur Demonstration wann der Destruktor aufgerufen wird.

automat3.cpp

```
1  #include <string>
2  #include "automat3.h"
3  using namespace std;
4
5  // Definition der Konstruktoren:
6  Automat::Automat(unsigned long g, string s){
7      set_geld(g);
8      set_standort(s);
9  }
10 Automat::Automat(unsigned long g){
11     set_geld(g);
12     set_standort("");
13 }
14 Automat::Automat(string s){
15     set_geld(0);
16     set_standort(s);
17 }
18 Automat::Automat(){
19     set_geld(0);
20     set_standort("");
21 }
22 // Definition der Destruktors:
23 Automat::~Automat(){
24     cout << get_standort() << " entfernt" << endl;
25 }
26
27 // Definition der Elementfunktionen:
28 unsigned long Automat::get_geld(){
29     return geld;}
30 void Automat::set_geld(unsigned long g){
31     geld = g;}
32 string Automat::get_standort(){
33     return standort;}
34 void Automat::set_standort(string s){
35     standort = s;}
```

Klassen (Destruktor - Beispiel)

Zurück zur Klasse Automat unter Verwendung von Destruktoren:

```
1   #include <iostream>
2   #include <vector>
3   #include <string>
4   #include <memory>
5   #include "automat3.h"
6   #include "automat3.cpp"
7   using namespace std;
8
9   Automat device1("global: Frankfurt");
10
11  int main(void){
12      Automat *device2 = new Automat(" main(new): Altenbochum");
13      static Automat device3("main(static): Langendreer");
14      Automat device4("main(): Wattenscheid");
15      unique_ptr<Automat> device5(new Automat("main(): Querenburg")); // ab C++11 möglich
16
17      delete device2;
18      return 0;
19  }
20
```

Output:

```
 main(new): Altenbochum entfernt
main(): Querenburg entfernt
main(): Wattenscheid entfernt
main(static): Langendreer entfernt
global: Frankfurt entfernt
```

Klassen (Destruktor - Beispiel)

Zurück zur Klasse Automat unter Verwendung von Destruktoren:

```
1   #include <iostream>
2   #include <vector>
3   #include <string>
4   #include <memory>
5   #include "automat3.h"
6   #include "automat3.cpp"
7   using namespace std;
8
9   Automat device1("global: Frankfurt");
10
11  int main(void){
12      Automat *device2 = new Automat(" main(new): Altenbochum");
13      static Automat device3("main(static): Langendreer");
14      Automat device4("main(): Wattenscheid");
15      unique_ptr<Automat> device5(new Automat("main(): Querenburg")); // ab C++11 möglich
16
17      delete device2;
18      return 0;
19  }
20
```

static (oder globale) Objekte werden am Ende des Programms gelöscht

Output:

```
 main(new): Altenbochum entfernt
main(): Querenburg entfernt
main(): Wattenscheid entfernt
main(static): Langendreer entfernt
global: Frankfurt entfernt
```

Klassen (Destruktor - Beispiel)

Zurück zur Klasse Automat unter Verwendung von Destruktoren:

```
1  #include <iostream>
2  #include <vector>
3  #include <string>
4  #include <memory>
5  #include "automat3.h"
6  #include "automat3.cpp"
7  using namespace std;
8
9  Automat device1("global: Frankfurt");
10
11 int main(void){
12     Automat *device2 = new Automat(" main(new): Altenbochum");
13     static Automat device3("main(static): Langendreer");
14     Automat device4("main(): Wattenscheid");
15     unique_ptr<Automat> device5(new Automat("main(): Querenburg")); // ab C++11 möglich
16
17     delete device2;
18     return 0;
19 }
20
```

lokal deklarierte (nicht static) Objekte werden am Ende des zugehörigen Anweisungsblocks gelöscht

Output:

```
 main(new): Altenbochum entfernt
main(): Querenburg entfernt
main(): Wattenscheid entfernt
main(static): Langendreer entfernt
global: Frankfurt entfernt
```

Klassen (Destruktor - Beispiel)

Zurück zur Klasse Automat unter Verwendung von Destruktoren:

```
1   #include <iostream>
2   #include <vector>
3   #include <string>
4   #include <memory>
5   #include "automat3.h"
6   #include "automat3.cpp"
7   using namespace std;
8
9   Automat device1("global: Frankfurt");
10
11  int main(void){
12      Automat *device2 = new Automat(" main(new): Altenbochum");
13      static Automat device3("main(static): Langendreer");
14      Automat device4("main(): Wattenscheid");
15      unique_ptr<Automat> device5(new Automat("main(): Querenburg")); // ab C++11 möglich
16
17      delete device2;
18      return 0;
19  }
20
```

dynamische Objekte werden an Ort und Stelle gelöscht

Output:

```
 main(new): Altenbochum entfernt
main(): Querenburg entfernt
main(): Wattenscheid entfernt
main(static): Langendreer entfernt
global: Frankfurt entfernt
```

Klassen (Vererbung)

- Mittels **Vererbung** (Ableiten von Klassen) läßt sich eine bereits existierende Klasse (*Basisklasse*) in einer neuen Klassen verwenden
  - Die abgeleitete Klasse erbt die **public** Daten und Elementfunktionen der Basisklasse
  - Syntax zur Definition einer abgeleiteten Klasse:

```
class Klassenname : Zugriffsrecht Basisklasse {
        // Daten und Elementfunktionen
}
```

    Damit legen Sie folgendes fest:
    - **Basisklasse** deren Daten und Elementfunktionen vererbt werden
    - **Zugriffsrechte** auf die Daten der Basisklasse
    - **Zusätzlichen Daten und Elementfunktionen**, um die erweitert wird

…und noch soviel mehr

- Es soll hiermit *lediglich ein Einstieg* in einige der grundlegenden Eigenschaften von C++ aufgezeigt werden
- Vieles ist dabei (zwangsläufig) unerwähnt geblieben:
  - Schlüsselwort **const** um Datentypen als Konstante zu definieren
  - Move – Konstruktoren (ab C++11)
  - **inline** – Elementfunktionen
  - **this** – Zeiger
  - Operatoren überladen
  - Funktions-Templates
  - Exception-Handling (Fehlerbehandlung)
  - …

Ausblick

Übungshinweis:

Südpol-Nutzung/ Betreuung heute nur von 15 bis 18 Uhr

(und morgen von 10 bis 13 Uhr)

Nächste Vorlesung

(Mittwoch, 03.04.19, 8:30Uhr, **HGB30**):

Einführung in Python