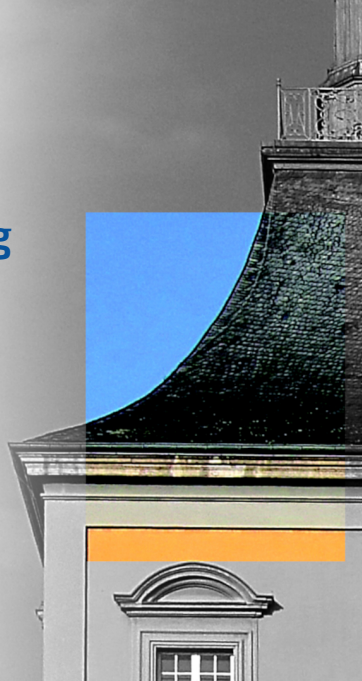
UNIVERSITÄT BONN

Algorithmen und Programmierung

Imperative Programmierung mit C++

Dr. Felix Jonathan Boes
boes@cs.uni-bonn.de
Institut für Informatik



Motivation

Wir wollen das praktische und theoretische Handwerkzeug erlernen um Sortieralgorithmen zu studieren

Wir erlernen zunächst die Grundlagen der imperativen Programmierung in C++

Anschließend erlernen wir erste theoretische Grundlagen um Algorithmen zu studieren

Imperative Programmierung mit C++

UNIVERSITÄT BONN

Literatur

Wir empfehlen Ihnen folgende Literatur. Schlagen Sie dort gern nach wenn Sie mehr wissen wollen oder einen Sachverhalt mit anderen Worten erklärt haben möchten.

- Buch zur Programmiersprache (vom Erfinder):
  Bjarne Stroustrup. *The C++ programming language*. Pearson Education, 2013.
- Hochwertige Onlinereferenz (technisch sehr genau und ausführlich):
  `https://cppreference.com`

Imperative Programmierung mit C++

Zum Computermodell

UNIVERSITÄT BONN

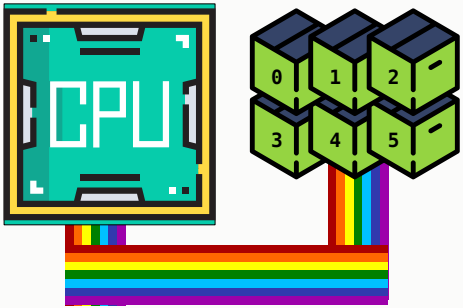
Einleitung

C++-Programme werden auf Computern ausgeführt. Bevor wir besser verstehen welche Datentypen uns zur Verfügung gestellt werden, schauen wir uns ein vereinfachtes Computermodell an auf dem C++-Programme ausgeführt werden.

UNIVERSITÄT BONN

Computermodell I

Wir konzentrieren uns hier auf eine **x86-64**-Architektur. Das ist ein Computermodell wie es in (fast) jedem Desktop-PC oder Laptop realisiert ist. Es gibt eine oder mehrere CPUs und eine Menge von Speicheradressen um Daten abzulegen.



In diesem Computermodell werden schlussendlich **Bytes** verarbeitet. Wir lernen zunächst was ein **Byte** ist.

UNIVERSITÄT BONN

Bits und Bytes

Menschen stellen Zahlen aus praktischen Gründen durch die zehn Ziffern 0 bis 9 dar. Computer benutzen nur die zwei Ziffern 0 und 1. Diese nennen wir **Bits**. Beispiel:

$$13_{10} = 1 \cdot 1000_2 + 1 \cdot 100_2 + 0 \cdot 10_2 + 1 \cdot 1_2 = 1101_2$$

Die kleinsten Speichereinheiten in modernen Computern sind 8-stellige Bitzahlen. Diese nennen wir **Bytes**. Beispiel:

$$10110011_2 = 128_{10} + 32_{10} + 16_{10} + 2_{10} + 1_{10} = 179_{10}$$

Um mit Bytes praktisch zu arbeiten, teilt man sie in zwei Päckchen ein $1011\ \ 0011_2$ und beschreibt jedes Päckchen durch eine **Hexadezimalziffer**. Beispiel:

$$10110011_2 = 1011\ \ 0011_2 = B3_{16} = \texttt{0xB3}$$

UNIVERSITÄT BONN

Computermodell II

Wir konzentrieren uns auf eine **x86-64**-Architektur auf der ein **GNU/Linux** Betriebssystem ausgeführt wird. Das Betriebssystem führt schlussendlich **Nutzerprozesse** aus. Das sind „normale“ Prozesse die keine besonderen Rechte haben.

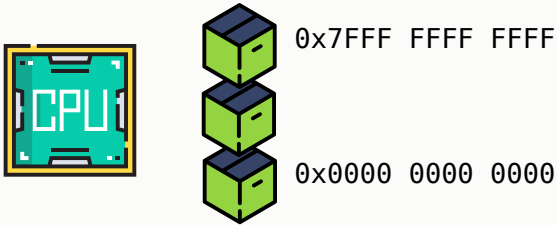
In dieser Vorlesung behandeln wir ausschließlich Nutzerprozesse. Wir schreiben in dieser Vorlesung ab jetzt nur noch **Prozess** und meinen damit **Nutzerprozess**.

UNIVERSITÄT BONN

Computermodell III

In unserem Computermodell verfügt jeder Prozess über folgende Ressourcen:

Mindestens **eine CPU** sowie den gesamten **virtuelle Adressraum 0x0000 0000 0000** bis **0x7FFF FFFF FFFF**. An jeder Adresse liegt genau ein Byte.



Alle Programmbefehle und alle Programmdaten liegen im virtuellen Adressraum (sowie den CPU-Registern) und werden von der CPU verarbeitet.

UNIVERSITÄT BONN

Computermodell IV

Aus der Sicht eines jeden C++-Programms gibt es also eine (oder mehrere) CPUs und einen durchnummerierte Folge von Bytes. Die Folge enthält `0x8000 0000 0000` $= 2^{47}$ Bytes[1].

Das durch C++ beschriebene Modell legt unter anderem fest,

- welche grundlegenden Datentypen es gibt,
- wie diese Datentypen durch Bytes dargestellt werden
- und wie die grundlegenden Datentypen durch Operatoren verändert werden.

Wir lernen die ersten grundlegenden Datentypen kennen und wie man diese durch Operatoren verändert[2].

[1] Wenn man die Adressen des Kernelspace dazu nimmt, sind es sogar 2^{48} Bytes. Aber diese Bytes sind nur für Systemprozesse zugänglich und wirs schauen uns hier nur Nutzerprozesse an.

[2] Genauer lernen wir wie die Werte von Variablen eines Datentyps durch Operatoren verändert werden.

Haben Sie Fragen?

Zusammenfassung

Jeder Prozess sieht eine CPU und einen eigenen virtuellen Adressraum

Imperative Programmierung mit C++

Grundlegende Datentypen

Offene Frage

Grundlegende Datentypen sind die Grundbausteine aus denen alle anderen Datentypen zusammengesetzt werden.

Welche Grundlegenden Datentypen gibt es?

UNIVERSITÄT BONN

Ganze Zahlen

C++ bietet uns verschiedene Möglichkeiten an um mit ganzen Zahlen zu arbeiten.

Mit dem Dateityp `int` wird ein zusammenhängender Block, bestehend aus 4 Byte, als ganze Zahl interpretiert.

Ein Byte kann $2^8 = 256$ verschiedene Zustände annehmen. Mit einem `int` stehen uns die ganzen Zahlen von ca. -2^{31} bis ca. 2^{31} zur Verfügung. Wie das im Detail funktioniert lernen Sie in der Vorlesung Techniche Informatik.

UNIVERSITÄT BONN

Natürliche Zahlen

C++ bietet uns verschiedene Möglichkeiten an um mit natürlichen Zahlen zu arbeiten.

Mit dem Dateityp `unsigned int` wird ein zusammenhängender Block, bestehend aus 4 Byte, als natürliche Zahl interpretiert.

Mit einem `unsigned int` stehen uns die ganzen Zahlen von 0 bis $2^{32} - 1$ zur Verfügung. Details: TI.

UNIVERSITÄT BONN

(Gleit)kommazahlen

C++ bietet uns verschiedene Möglichkeiten an, um mit „Gleitkommazahlen“[3] zu arbeiten.

Mit dem Dateityp `double` wird ein zusammenhängender Block, bestehend aus 8 Byte, als Kommazahl interpretiert. In C++ notiert man Kommazahlen mit einem Punkt (und keinem Komma). ¯_(ツ)_/¯

Umsetzung ist motiviert durch Erkenntnisse der Physik und des wissenschaftlichen Rechnens: Produkte sind wichtiger als Summen. Die Konsequenz: Erst Aufteilung in Bereiche der Größe $2^E, 2^{E-1}, 2^{E-2}, \ldots$. Unterteile diese Bereiche dann gleichmäßig. Details: TI.

[3] Die Gleitkommafreunde sind eine studentische „Hackergruppe“, gegründet hier, und mittlerweile Teil des RedRocket e.V.. Siehe `https://redrocket.club`

UNIVERSITÄT BONN

Zeichen

C++ bietet uns verschiedene Möglichkeiten an, um mit Zeichen zu arbeiten.

Mit dem Dateityp `char` wird ein zusammenhängender Block, bestehend aus 1 Byte, als Zeichen interpretiert.

Mit dem Dateityp `wchar` wird ein zusammenhängender Block, bestehend aus 1, 2 oder 4 Byte, als Zeichen interpretiert.

Text

C++ bietet uns verschiedene Möglichkeit an, um mit Texten zu arbeiten.

Wie das Arbeiten mit Texten funktioniert schauen wir uns später im Detail an. Insbesondere verstehen wir erst später welche Texttypen es gibt. Texte sind keine Grundlegenden Datentypen.

UNIVERSITÄT BONN

Wahrheitswerte

C++ bietet uns einen Datentyp an um mit Wahrheitswerten zu arbeiten.

In dem Datentyp `bool` können die Wahrheitswerte **`true`** und **`false`** gespeichert werden. Wie dieser Datentyp im Speicher umgesetzt ist, wird durch den C++-Standard nicht vollständig definiert und kann abhängig vom jeweiligen Compiler unterschiedlich gehandhabt werden.

UNIVERSITÄT BONN

Zusammenfassung

Wir lernen nach und nach die grundlegenden Datentypen von C++ kennen. Wir kennen bereits Folgende.

Datentyp	Beschreibung	Wichtige Eigenschaft
`int`	4-Byte Ganzzahl	$\approx$ -2,1 Milliarden bis $\approx$ 2,1 Milliarden
`unsigned int`	4-Byte Ganzzahl	0 bis $\approx$ 4.3 Milliarden
`double`	8-Byte Gleitkommazahl	$\approx -1.8 \cdot 10^{308}$ bis $\approx 1.8 \cdot 10^{308}$
`char`	1-Byte Zeichen	Kann 256 einfache Zeichen darstellen
`wchar`	mehr-Byte Zeichen	Kann „alle“ Zeichen darstellen
`bool`	Speichert Wahrheitswert	Größe ist implementierungsabh.

Es gibt noch weitere grundlegende Dateitypen. Diese sind bei der hardwarenahen Programmierung wichtig. Sie lernen diese unter anderem in der Vorlesung IT Sicherheit kennen.

UNIVERSITÄT BONN

Der sizeof-Operator

Um die Anzahl der verwendeten Bytes eines Datentyps zu bestimmen, können Sie den Operator **`sizeof`**`(type)` nutzen.

```
drucke_zahl(sizeof(int));
drucke_zahl(sizeof(double));
drucke_zahl(sizeof(char));
```

Haben Sie Fragen?

Zusammenfassung

Wir kennen nun wichtige grundlegende Datentypen

Imperative Programmierung mit C++

Variablen

Variablen geben einem Datentyp im Speicher einen Namen

Mit Variablen benennen wir also Werte im Speicher, die sich über die Zeit ändern können

UNIVERSITÄT BONN

Variablen

Variablen haben immer einen **Namen**, einen **Typ**, einen **Wert** und eine **Adresse**. Beispiel:

```
int x;              // Name: x
                    // Typ : int
                    // Wert: ??? Vielleicht 42?
                    // Adr : 0x7FFF FFFF FF50

double y = 33.3;    // Name: y
                    // Typ : double
                    // Wert: 33.3
                    // Adr : 0x7FFF FFFF FF38
```

Namen beginnen mit einem Buchstaben und bestehen aus Buchstaben und Zahlen.

UNIVERSITÄT BONN

Variablen deklarieren

Um Variablen im Programmcode nutzen zu können, müssen Variablen deklariert werden[4]. Es gibt verschiedene Möglichkeiten Variablen zu deklarieren.

```
// Deklaration einer Variable von gegebenem Typ.
TYP VARIABLENNAME;                 // Bsp: int x;

// Deklaration einer Variable von gegebenem Typ und Zuweisung eines
// (Start)werts.
TYP VARIABLENNAME = WERT;          // Bsp: int x = 9;

// Deklaration einer Variable von gegebenem Typ und Zuweisung eines
// (Start)werts der zunächst durch eine Expression "berechnet" wird.
TYP VARIABLENNAME = EXPRESSION;    // Bsp: int x = 7 * (2 + 4);
```

Die Frage „wo“ oder „wie lange“ Variablen verfügbar sind, klären wir später.

[4] Die Variablen können nur im zugehörigen Scope verwendet werden, dazu später mehr.

UNIVERSITÄT BONN

Expressions

Einfach gesagt ist eine **Expression** ein Stück Programmcode, dass den Wert eines definierten Typs berechnet. Expressions haben immer einen Typ und einen Wert.

Beispiele für Expressions sind:

- Zahlen wie die Ganzzahl `6`.
- Zeichen wie das `'m'`.
- Variablen. Hier wird der Wert der Variable zum Ausführungszeitpunkt der Expression verwendet.

```
int x = 52; // In x wird der Wert 52 gespeichert.
int y = x;  // Die Expression 'x' wertet als 52 aus.
            // In y wird dann der Wert 52 gespeichert.
```

- Durch Operatoren verknüpfte Expressions, wie `3+4` oder `x*3` (dazu gleich mehr).
- Rückgabewerte von Funktionen (das verstehen Sie später).

UNIVERSITÄT BONN

Zuweisungen

Variablen die in einem gegeben Programmabschnitt bekannt sind[5], können einen neuen Wert zugewiesen bekommen. Der zugewiese Wert ist im Allgemeinen eine Expression.

```
int x = 3; // Deklaration mit Startwert
x = 4 + 6; // Zuweisung des Werts 10
```

[5] Gedulden Sie sich noch. Wie gesagt lernen Sie später alles wichtige über Scopes.

Haben Sie Fragen?

Zusammenfassung

Variablen besitzen Namen, Typ, Wert und Adresse und geben so einem Datentyp im Speicher einen Namen

Expressions haben Typ und Wert

Variaben wird der Wert einer Expression zugewiesen

Imperative Programmierung mit C++

Literale, Konstanten und Operatoren

Wir beantworten folg. Fragen

Wie notieren wir feste Werte?

Wie konstruieren wir neue Werte?

Literale

(Literal)

Als **Literal** bezeichnen wir die textuelle Darstellung eines festen Werts (eines gegebenen Datentyps) im Programmcode.

Wir schauen uns zwei Beispiele für Literale an.

```
int x = 1236; // Der Programmtext bestehend aus der 1, 2, 3 und der 6
              // ist ein Literal für die Ganzzahl 1236.

char buchstabe = 'F'; // Der Programmtext 'F' ist ein Literal für den
                      // Buchstaben F.

drucke_text("Hallo zusammen"); // der Programmtext "Hallo zusammen" ist
                           // ein Literal für den Text "Hallo zusammen".
```

UNIVERSITÄT BONN

Konstanten

In manchen Situationen möchten wir konstante Werte einfach notieren können. Beispielsweise möchten wir die Kreiszahl π als konstanten Wert nutzen ohne uns daran zu erinnern dass $\pi \approx 3,14159265358979323846$ ist.

Zu diesem Zweck nutzt man **Konstanten**. Wir lernen hier zunächst eine Möglichkeit kennen um Konstanten zu definieren.

```
// So definiert man eine Variable die einen
// nicht änderbaren Wert enthält.
const TYP VARIABLENNAME = EXPRESSION;

// Beispiel
const double pi = 3.14159265358979323846;
// Das hier verbietet der Compiler:
pi = 3;
```

Haben Sie Fragen?

UNIVERSITÄT BONN

Operatoren

Mit **Operatoren** werden aus zwei Expressions eine neue Expression konstruiert.

```
EXPRESSION OP EXPRESSION
```

Welche Typen die beteiligten Expressions haben dürfen, ist durch den Standard definiert.

UNIVERSITÄT BONN

Arithmetische Operatoren für Ganzzahlen

Für ganze Zahlen stehen uns folgende Operatoren zur Verfügung.

```
a + b // Berechnet die Summe
a - b // Berechnet die Differenz
a * b // Berechnet das Produkt
a / b // Berechnet den Quotienten OHNE REST
a % b // Berechnet den Rest der Ganzzahldivision
```

Nebenbemerkung: Es kann geschehen, dass ein Operator eine Zahl produziert, die nicht in einem Ganzzahltyp gespeichert werden kann. Das ist zum Beispiel der Fall wenn das Ergebnis zu groß / klein ist. Ist das der Fall, dann produziert C++ (genauer die CPU) dennoch einen Wert, dieser hat aber einen für uns unerwarteten, falschen Wert.

UNIVERSITÄT BONN

Arithmetische Operatoren für Gleitkommazahlen

Für Gleitkommazahlen stehen uns folgende Operatoren zur Verfügung.

```
a + b // Berechnet die Summe
a - b // Berechnet die Differenz
a * b // Berechnet das Produkt
a / b // Berechnet den Quotienten
```

Nebenbemerkung: Es kann geschehen, dass ein Operator eine Zahl produziert, die nicht in einem Gleitkommazahltyp gespeichert werden kann. Ist das der Fall, dann produziert C++ (genauer die CPU) Werte wie `inf` (zu groß und positiv), `-inf` zu klein und negativ, oder `NaN` (kein Zahlenwert, z.B. bei der Division durch 0).

UNIVERSITÄT BONN

Operatoren und implizites Typecasting

Es ist praktischerweise möglich verschiedene Zahlentypen durch Operatoren zu verknüpften. Als Faustregel erhalten wir immer einen Datentyp „der das Ergebnis ohne Genauigkeitsverlust speichern kann und dabei am wenigsten Bytes verwendet“.

```
3 + 0.9 // Erstellt zunächst die Gleitkommazahl 3.0
        // und addiert darauf 0.9.

3 / 2   // Da die Expressions 3 und 2 beide Ganzzahlen sind,
        // wird die Ganzzahldivision ohne Rest verwendet.

3 / 2.0 // Erstellt zunächst die Gleitkommazahl 3.0
        // und berechne dann die Gleitkommazahl 3.0 / 2.0.
```

Man nennt diesen Vorgang **implizites Typecasting** weil der Typ des Ausdrucks ohne „explizit sichtbare Aufforderung“ in einen anderen Typ umgewandelt wird. Explizites Typecasting lernen wir auch noch kennen.

UNIVERSITÄT BONN

Operatorpräzedenz und Assoziativität

Bei einer Folge von Operatoren (wie zum Beispiel `1.0 + 2.0 + 3.0` oder `1.0 * 2 / 3`) legt der Standard fest, welcher Operator zuerst ausgewertet wird. Dabei legt die **Assoziativität** das Verhalten fest, wenn die Operatoren gleich sind. Die **Operatorpräzedenz** legt das Verhalten fest, wenn die Operatoren unterschiedlich sind.

Sie finden eine (lange und schlecht zu merkende) Liste hier:

`https://en.cppreference.com/w/c/language/operator_precedence`

UNIVERSITÄT BONN

Klammern

Da die Operatorpräzedenz und die Assoziativität für viele Programmierer:innen verwirrend sind, erhöht man die Lesbarkeit, indem man die Ausführungsreihenfolge durch Klammern erzwingt.

```
// weit weit weg definiert und vergessen was die Typen sind...
double eins = 1; // Impliziter Typecast von 1 zu 1.0
int zwei    = 2;
int drei    = 3;

// mein cleverer Code
double x;
x = zwei / drei * eins;
x = zwei / (drei * eins);
```

Nutzen Sie Klammern um Ihren Code lesbarer zu machen.

UNIVERSITÄT BONN

Logische Operatoren (Un)Gleichheit

Logische Operatoren sind Expressions. Der Standard legt Folgendes fest.

```
// Für beliebige Expressions a und b gilt Folgendes.
a == b // Wertet zu true  aus wenn a und b gleich sind.
       // Wertet zu false aus wenn a und b nicht gleich sind.
a != b // Wertet zu true  aus wenn a und b verschieden sind.
       // Wertet zu false aus wenn a und b nicht verschieden sind.
```

UNIVERSITÄT BONN

Logische Operatoren
Logisches NICHT

Für die `bool`-Expression a gilt:

```
// Das logische NICHT:
!a    // Wertet a aus und prüft ob diese Expression false ist.
not a // Falls ja dann wertet !a als true aus.
      // Falls a zu true auswertet dann wertet !a zu false aus.
```

UNIVERSITÄT BONN

Logische Operatoren
Logisches UND

Für `bool`-Expressions a und b gilt:

```
// Das logische UND:
a && b  // Wertet a aus und prüft ob diese Expression false ist.
a and b // Falls ja dann wird b nicht ausgeweret und a && b wertet
        // zu false aus.
        // Falls a nicht zu false auswertet wird b ausgewertet und
        // geprüft ob diese Expression false ist. Falls ja
        // wertet a && b zu false aus und sonst zu true.
```

UNIVERSITÄT BONN

Logische Operatoren
Logisches UND

Die Auswertung von && ist nicht kommutativ. Das heißt, es macht einen Unterschied ob `a && b` oder `b && a` ausgewertet wird. Manchmal ist das erwünscht (zur „unleserlichen hacky Optimierung“) oft ist das aber nicht erwünscht.

```
// Hacky: Falls Felix in der Nähe ist geht man hin und guckt
// ob er rosa Schnürsenkel hat. In dem Fall lobt man Felix.
// Das Nachsehen ist total teuer wenn man grad nicht in der Nähe ist...
if (in_der_naehe(felix) && hat_er_rosa_schnuersenkel(felix)) {
  lobe(felix);
}
```

UNIVERSITÄT BONN

Logische Operatoren
Logisches ODER

Für Expressions a und b die ganze oder natürliche Zahlen sind gilt:

```
// Das logische ODER:
a || b // Prüft ob a zu true auswertet. Falls ja
a or b // wird b nicht ausgeweret und a || b wertet zu true aus.
       // Falls a nicht zu false auswertet wird b ausgewertet
       // und geprüft ob diese Expression true ist.
       // Falls ja wertet a || b zu true aus und sonst zu false.
```

Die Auswertung von || ist ebenfalls nicht kommutativ.

UNIVERSITÄT BONN

Bitweise Operatoren

Schlussendlich ist jeder Dateityp eine Folge von Bytes im Speicher. Also ist jeder Datentyp eine Folge von Bits im Speicher. C++ stellt uns folgende Operatoren zur Verfügung, um mit der die Bitdarstellung von Zahlen Ausdrücke zu bilden.

```
a << b  // Schiebt die Bits von a um b stellen nach links
a >> b  // Schiebt die Bits von a um b stellen nach rechts
a &  b  // Berechnet das Bitweise UND von a und b
a |  b  // Berechnet das Bitweise ODER von a und b
a ^  b  // Berechnet das Bitweise XOR von a und b
~a      // Kehrt jedes Bit von a um
```

Achtung: `a & b` wertet `a` und `b` aus und bildet dann das bitweise UND. Insbesondere ist die Auswertung von `&` kommutativ. Wir sehen somit ein, dass die Operatoren `&` und `&&` unterschiedlich bei der Auswertung verhalten. Das gilt auch für `|` und `||`.

UNIVERSITÄT BONN

Operatoren - Abkürzungen

Folgene Abkürzungen stellt C++ zur Verfügung. Diese Abkürzungen *können* die Lesbarkeit erhöhen.

```
a OP= b;  // steht für a = a op b;
a += b;   // steht für a = a + b;
a -= b;   // steht für a = a - b;
a *= b;   // steht für a = a * b;
a /= b;   // steht für a = a / b;
a %= b;   // steht für a = a % b;
a &= b;   // steht für a = a & b;
a |= b;   // steht für a = a | b;
a ^= b;   // steht für a = a ^ b;
```

Haben Sie Fragen?

Zusammenfassung

Durch Literale und Konstanten werden feste Werte ausgedrückt

Die Operatoren zum Verändern der grundlegende Datentypen sind eigentlich wohlbekannt

Aber es gibt einige Nebeneffekte die man beim Schreiben von schlecht lesbarem Code beachten muss ¯_(ツ)_/¯

Imperative Programmierung mit C++

Mehr Statements

Ziel: vielfältiger ausdrücken

Sie lernen Compound Statements kennen

Sie lernen weitere Selection Statements und Iteration Statements

UNIVERSITÄT BONN

Compound Statements

Ein **Compound Statement** ist eine Folge von Statements, die ein einem {}-Block zusammgefasst sind.

```
// Dies ist ein Compound Statement:
{
  int x = 3;
  drucke_zahl(3);
}
```

UNIVERSITÄT BONN

Scope

Man bezeichnet die Bereiche in denen eine Variable bekannt ist als **Scope**.

Wird eine Variable in einem Compound Statement deklariert, dann ist die Variable innerhalb des Compound Statements bekannt und auch innerhalb aller in diesem Compound Statement vorhandenen Compound Statements.

```
{
  int x; // Hier wird x deklariert

  {
    x = 3; // Hier ist x bekannt
  }
}
x = 4; // Hier ist x unbekannt
```

UNIVERSITÄT BONN

Scope

Es ist erlaubt Variablen mit demselben Namen zu deklarieren falls diese in verschiedenen Compound Statements deklariert werden.

Soll eine Variable verwendet werden, dann wird diese innerhalb des aktuellen Compound Statements gesucht. Wird sie dort nicht gefunden, wird dass darum befindliche Compound Statement durchsucht. So wird entweder die erste vorkommende Deklaration der Variable gefunden und verwendet oder es ist nicht möglich das Programm zu kompilieren.

```
...
  int x = 4;
  {
    int x = 5;
    {
      x = 9; // ändert x = 5 zu x = 9
    }
  }
  drucke_zahl(x); // druckt 4
```

UNIVERSITÄT BONN

Selection Statements

Die Programmiersprache legt fest, dass folgende Selection Statements zulässig sind.

```
if (EXPRESSION)
  STATEMENT

if (EXPRESSION)
  STATEMENT
else
  STATEMENT
```

Hier sind bei STATEMENT alle zulässigen Statements nutzbar. Zum Beispiel ist unsere bereits bekannte Konstruktion zulässig.

```
if (EXPRESSION) {
  ...
} else {
  ...
}
```

Selection Statements

Aus der Definition der zulässigen Selection Statements **folgt** jetzt auch, dass folgende Konstruktion zulässig ist.

```
if (EXPRESSION) {
  ...
} else if (EXPRESSION) {
  ...
} else if (EXPRESSION) {
  ...
}
```

Dies ist ein das Werk unser ständigen Begleiterin der Abstraktion. Die Programmiersprache soll durch möglichst wenig und gleichzeitig einfach verständlicher Grundbausteine erschaffen werden.

UNIVERSITÄT BONN

Iteration Statements

Wir kennen schon das Iteration Statement **`while`**. Folgendes Statment ist zulässig:

```
while (EXPRESSION)
  STATEMENT
```

Wie oben sind bei `STATEMENT` alle zulässigen Statements nutzbar. Das führt zu der (im besten Fall) immer genutzten Konstruktion:

```
while (EXPRESSION) {
  ...
}
```

UNIVERSITÄT BONN

Iteration Statements

Wenn man ein (Compound) Statement für eine Folge von Zahlen ausführen möchte, nutzt man statt **`while`** lieber **`for`**.

```
for (INIT; BEDINGUNG; RUNDEN_ENDE)
  STATEMENT
```

Hierbei ist `INIT` meist ein Statement zur Initialisierung. Dieses Statement wird immer ausgeführt. Zu Beginn des Schleifendurchlaufs wird die Bedingung geprüft. Am Ende jedes Schleifendurchlaufs wird `RUNDEN_ENDE` ausgeführt.

```
// Drucke die 2-er Potenzen kleiner als 100
for (int i = 1; i < 100; i = i*2) {
  drucke_zahl(i);
}
```

Iteration Statements

Zwischen **for**-Schleife und **while**-Schleife besteht folgender Zusammenhang.

```
// Diese beiden Schleifen werden gleichwertig ausgeführt.
for (INIT; BEDINGUNG; RUNDEN_ENDE) {
  ...
}

INIT;
while (BEDINGUNG) {
  ...
  RUNDEN_ENDE;
}
```

UNIVERSITÄT BONN

Noch mehr Abkürzungen

Beim Angeben von RUNDEN_ENDE werden sehr oft diese Abkürzungen verwendet:

```
a++; // Wertet aus zu: a;   Setzt a auf a+1
++a; // Wertet aus zu: a+1; Setzt a auf a+1
a--; // Wertet aus zu: a;   Setzt a auf a-1
--a; // Wertet aus zu: a-1; Setzt a auf a-1

// Beispiel:
for (int i = 0; i < 10; i++) {
  drucke_zahl(i);
}
```

Haben Sie Fragen?

Zusammenfassung

Wir haben Compound Statements kennen gelernt

Wir wissen dass die Verfügbarkeit einer Variable durch das Scope angegeben wird

Wir können uns vielfältiger durch Selection Statements und Iteration Statements ausdrücken

Imperative Programmierung mit C++

Zur Nachvollziehbarkeit

UNIVERSITÄT BONN

Faustregel für Programmieranfänger:innen

Aussagekräftigen, gut lesbaren Code zu schreiben muss gelernt werden. Hier sind einige Faustregeln für Programmieranfänger:innen.

- Ziel und Zweck von Funktionen zu kommentieren ist immer wichtig.
- Ein Überblick über die Details des Funktionscode zu geben ist sehr oft wichtig.
- Iterations Statement zu kommentieren ist sehr oft wichtig.
- Selection Statements zu kommentieren ist oft wichtig.
- (Einfache) Expression Statements müssen nicht kommentiert werden.