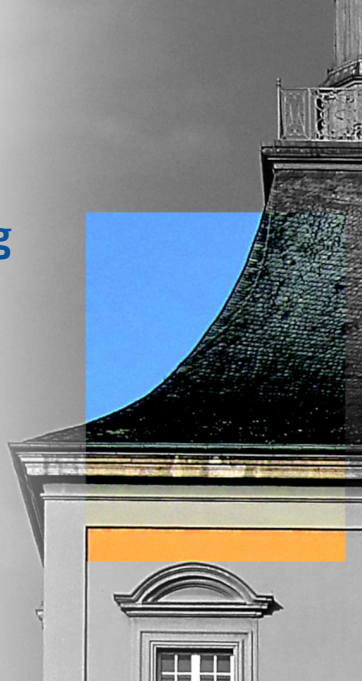
UNIVERSITÄT BONN

Algorithmen und Programmierung

Imperative Programmierung mit C++

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



KURZE WIEDERHOLUNG

Die Programmiersprache C++

Zum Computermodell

UNIVERSITÄT BONN

Computermodell

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.

Die Programmiersprache C++

Grundlegende Datentypen

UNIVERSITÄT BONN

Grundlegende Datentypen

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.

Die Programmiersprache C++

Variablen

UNIVERSITÄT BONN

Variablen

Variablen haben immer einen **Namen**, einen **Typ**, einen **Wert** und eine **Adresse**. Namen beginnen mit einem Buchstaben und bestehen aus Buchstaben und Zahlen.

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

// 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);
```

Sie haben gelernt dass das Scope die Verfügbarkeit einer Variable im Programmcode beschreibt.

Die Programmiersprache C++

Literale, Konstanten und Operatoren

UNIVERSITÄT BONN

Literale und Konstanten

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

Durch konstante Variablen werden „neue feste Werte“ definiert.

UNIVERSITÄT BONN

Operatoren

Operatoren kombinieren Expressions (zu neuen Expressions). Wir haben arithmetische, vergleichende, logische und bitweise Operatoren kennen gelernt.

Wir haben etwas über implizites Typecasting gelernt.

Wir haben gelernt dass das Klammern von Operatoren die Lesbarkeit erhöht.

Die Programmiersprache C++

Mehr Statements

UNIVERSITÄT BONN

Mehr Statements

Wir haben Compound Statements kennen gelernt und wir können uns vielfältiger durch Selection Statements und Iterations Statements ausdrücken.

Die Programmiersprache 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.

Die Programmiersprache C++

Wiederkehrenden Programmcode zusammenfassen in Funktionen

Übergeordnete Frage

Wie fasst man wiederkehrende Codeabschnitte verständlich und wiederverwendbar zusammen?

UNIVERSITÄT BONN

Beispiel: Sinus berechnen

Angenommen wir wollen mehrfach einen Sinus[1] berechnen. Dann ist eine Möglichkeit den Sinus bei jeder Verwendung durch eigenen Programmcode zu bestimmen.

```
drucke_text("Nenne mir eine Kommazahl, ich nenne dir den Sinus davon");
double x = lies_double();

double sinus = 0;     // Berechnet den Sinus als Taylorentwicklung
double summand = x;   // Aktueller Summand der Taylorentwicklung
double zaehler_helfer = 1; // Hilft beim Berechnen des Nenners
double klein = 1e-8;  // Eine kleine Zahl

// Solange |summand| > klein
while (summand > klein || -summand > klein) {
  sinus += summand;
  summand = -(summand*x*x) / ((zaehler_helfer+1) * (zaehler_helfer+2));
  zaehler_helfer += 2;
}

drucke_zahl(sinus);
```

[1] Z.B. durch Berechnen des Anfangs der Taylorentwicklung https://de.wikipedia.org/wiki/Taylorreihe#Trigonometrische_Funktionen

UNIVERSITÄT BONN

Beispiel: Sinusfunktion

Wenn wir das Teilprogramm zur Sinusberechnung bei jeder Verwendung neu hinschreiben müssen, verringert das die Lesbarkeit und Wartbarkeit enorm.

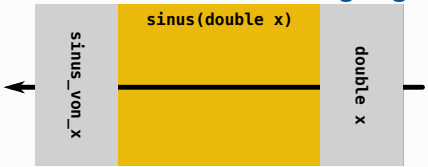
Bemerkung: Der Programmcode hängt variabel von einem `double x` ab und produziert dazu einen `double sin(x)`.

UNIVERSITÄT BONN

Ziel und Zweck von Funktionen bildlich

Um die Lesbarkeit und Wartbarkeit zu verbessern wird der Code ausgelagert.

```
...
double x = 1.2;
Berechne Sinus von x
double ergebnis = der Sinus von x;
...
double x = 9.2;
Berechne Sinus von x
double coole_zahl = der Sinus von x;
```




```
...
double ergebnis = sinus(1.2);
...
double ergebnis = sinus(2.9);
...
```

UNIVERSITÄT BONN

Funktionen

Nachdem wir verstanden haben, von welchen (variablen) Datentypen ein gegebener, wiederkehrender Programmabschnitt abhängt und welchen festen Datentyp der Programmabschnitt produziert, können wir den Programmcode zu einer **Funktion** zusammenfassen.

Jede Funktion hat eine(n) fest definierten

- **Namen**.
- **Folge von Parametervariablen**. Diese stellen „den Input“ der Funktion dar.
- **Rückgabetyp**. Dies stellt „den Output“ der Funktion dar.
- **Programmcode**.

Die **Signatur** einer Funktion besteht aus Rückgabetyp, Funktionsname, Parametertypen und -reihenfolge.

UNIVERSITÄT BONN

Funktionen definieren I

Funktionen werden in C++ wie folgt definiert.

```
RÜCKGABETYP FUNKTIONSNAME () // Keine Parameter
  COMPOUND_STATEMENT

RÜCKGABETYP FUNKTIONSNAME (TYP_1 NAME_1) // Ein Parameter vom Typ TYP_1
  COMPOUND_STATEMENT

RÜCKGABETYP FUNKTIONSNAME (TYP_1 NAME_1, TYP_2 NAME_2) // Zwei Parameter
  COMPOUND_STATEMENT

...
```

Die Parameter sind Variablen die innerhalb der Funktion zur Verfügung stehen.

UNIVERSITÄT BONN

Funktionen definieren II

Bei der Funktionsdefinition

```
RÜCKGABETYP FUNKTIONSNAME ( . . . )
  COMPOUND_STATEMENT
```

muss im Compound Statement ein **`return`**-Statement erreicht werden.

```
return EXPRESSION;
```

Die Expression muss denselben Typ haben wie der Rückgabetyp. Beim Funktionsaufruf gibt die Funktion den Wert der Expression über das **`return`**-Statement zurück.

UNIVERSITÄT BONN

Beispiel: Sinusfunktion

Die Sinusfunktion von oben wird dann wie folgt zu einer Funktion zusammengefasst.
Die Funktion hat einen Parameter (`double x`) und hat als Rückgabetyp einen `double`.

```
double sin(double x) {
  double sinus = 0;          // Berechnet den Sinus als Taylorentwicklung
  double summand = x;        // Aktueller Summand der Taylorentwicklung
  double zaehler_helfer = 1; // Hilft beim Berechnen des Nenners
  double klein = 1e-8;       // Eine kleine Zahl

  // Solange |summand| > klein
  while (summand > klein || -summand > klein) {
    sinus += summand;
    summand = -(summand*x*x) / ((zaehler_helfer+1) * (zaehler_helfer+2));
    zaehler_helfer += 2;
  }
  return sinus;
}

double ergebnis = sin(0.4);
```

UNIVERSITÄT BONN

Noch ein Beispiel

Das Compound Statement der Funktion muss in jedem Fall immer ein **`return`**-Statement erreichen.

```
// Die Funktion soll grade Zahlen halbieren und
// ungerade Zahlen um eins verringern
int komisch_funktion(int gegebene_zahl) {
  // Ist die Zahl gerade?
  if (gegebene_zahl % 2 == 0) {
    return gegebene_zahl / 2;
  } else {
    return gegebene_zahl - 1;
  }
}
```

Bemerken Sie, dass das Compound Statement immer ein **`return`**-Statement erreicht.

UNIVERSITÄT BONN

Funktionen aufrufen

Beim Nutzen einer Funktion müssen alle geforderten Parameter übergeben werden. Die Parametervariablen sind nur im Funktionskörper bekannt. Pro Parameter wird eine Expression ausgewertet. Der Wert jeder Expression wird dann in der zugehörigen Parametervariablen gespeichert.

Sobald im Funktionskörper ein **`return`**-Statement erreicht, wird die Berechnung der Funktion beendet. Die Funktion selbst ist eine Expression und wertet zu der Expression des **`return`**-Statements aus.

UNIVERSITÄT BONN

Noch ein Beispiel

```
// Die Funktion soll grade Zahlen halbieren und
// ungerade Zahlen um eins verringern
int komisch_funktion(int gegebene_zahl) {
  // Ist die Zahl gerade?
  if (gegebene_zahl % 2 == 0) {
    return gegebene_zahl / 2;
  } else {
    return gegebene_zahl - 1;
  }
}

// Hier wird die Expression komische_funktion(9)
// ausgewertet. So wird x der Wert 8 zugewiesen.
int x = komische_funktion(9);

// Hier wird die Expression komische_funktion(x)
// ausgewertet. Zuerst wird x ausgeweret. Das ist 8.
// Es wird dann also komisch_funktion(8) ausgewertet.
// Das ist Dann wird x der Wert 4 zugewiesen.
x = komisch_funktion(x);
```

UNIVERSITÄT BONN

Return und Ausführungsende

Wird ein **`return`**-Statment erreicht, so wird dort die Ausführung der aktuellen Funktion beendet.

```
// Wir zeigen Programmcode der nie erreicht wird.
int lotr() {
  int king = 3
  return king;
  // Der König ist zurückgekehrt. Das hier wird nicht ausgeführt.
  drucke_text("Du bist doof");
  return 1;
}
```

UNIVERSITÄT BONN

Der Datentyp void

Es gibt Expressions die „einen leeren Wert“ produzieren. Zum Beispiel sind das Funktionen die „nichts“ zurückgeben sollen.

Um festzulegen dass eine Funktion nichts zurückgibt, wird als Rückgabetyp der grundlegende Datentyp `void` verwendet. Der grundlegenden Datentyp `void` hat darüber hinaus noch weitere Anwedungszwecke. Dazu später mehr.

Der Datentyp `void` besitzt keinen zulässigen Wert. Es ist nicht spezifiziert wieviele Byte der Datentyp `void` hat[2].

[2] Wenn ihr Compiler sagt: „sizeof(void) ist 1“ dann ist das eine „Erweiterung der Programmiersprache“ durch Ihren Compiler.

UNIVERSITÄT BONN

Noch ein Beispiel

Hier ein einfaches Beispiel.

```
void drucke_sinus(double x) {
  drucke_kommazahl(sin(x));

  return;
}
```

UNIVERSITÄT BONN

Funktionen überladen

In C++ ist es möglich Funktionen zu **überladen**. Das heißt, es ist möglich Funktionen zu definieren die **denselben Funktionsnamen** und **unterschiedliche Parameteranzahl und/oder Parametertypen** haben.

Allerdings ist es unmöglich, zwei Funktionen zu definieren die deneslben Namen, dieselben Parameter aber einen unterschiedlichen Rückgabetyp haben.

Das Überladen von Funktionen wird eher selten genutzt. Funktionen sollen nur dann überladen werden wenn es den Programmcode nachvollziehbarer macht.

UNIVERSITÄT BONN

Beispiel

Wir betrachten ein sehr simples Beispiel.

```
#include <iostream>

void drucke(double zahl) {
  // Drucke die Zahl
}

void drucke(std::string erklaerung, double zahl) {
  // Drucke zuerst die Erklärung und dann die Zahl
}

int main() {
  drucke("Die Menge in Gramm beträgt: ", 0.9);
  drucke(3.9);
}
```

Haben Sie Fragen?

UNIVERSITÄT BONN

Gibt es Funktionen mit mehreren Rückgabetypen?

Mehrere Rückgabetypen in einer Funktion direkt zurückzugeben, können wir hier technisch noch nicht umsetzen. Im Sinne von C++ will man auch nicht mehrere Rückgabetypen haben sondern einen „neuen“ Rückgabetyp der aus mehreren grundlegenden Datentypen besteht. Dazu später mehr.

UNIVERSITÄT BONN

Funktionen verhalten sich wie konstante Variablen

Eine Funktion hat einen **Namen**, einen **Typ** (der durch die Folge der Typen der Parametervariablen und des Rückgabetyps festgelegt ist), einen konstanten **Wert** (der durch den Programmcode der Funktion festgelegt ist) und eine (Start-)**Adresse** im virtuellen Adressraum.

In C++ ist es hilfreich, Funktion als konstante Variablen zu verstehen. Es ist sogar praktisch möglich, Funktionen als Parameter an Funktionen zu übergeben. Dazu später mehr.

UNIVERSITÄT BONN

Variable Funktionsparameter

In C++ ist es möglich Funktionen zu definieren, die eine variable Anzahl und Typen von Funktionsparameter zulassen.

Wie man solche Funktionen selbst schreibt ist nicht Thema dieser Vorlesung. Wenn sie solche Funktionen schreiben wollen, warten Sie bis zum Ende der Vorlesungen mit dem Titel *Imperative Programmierung mit C++* ab und lesen Sie sich selbst durch wie solche Funktionen implementiert werden.

Haben Sie Fragen?

Zusammenfassung

Funktionen fassen wiederkehrende Codeabschnitte zusammen

Der „Input“ der Funktion wird durch die Funktionsparameter definiert

Der „Output“ der Funktion wird durch den Rückgabetyp und das `return`-Statement definiert

Die Aufteilung von Codestücken in nachvollziehbare Funktionen muss gelernt werden

Sie lernen nun erste Eigenschften von GIT kennen