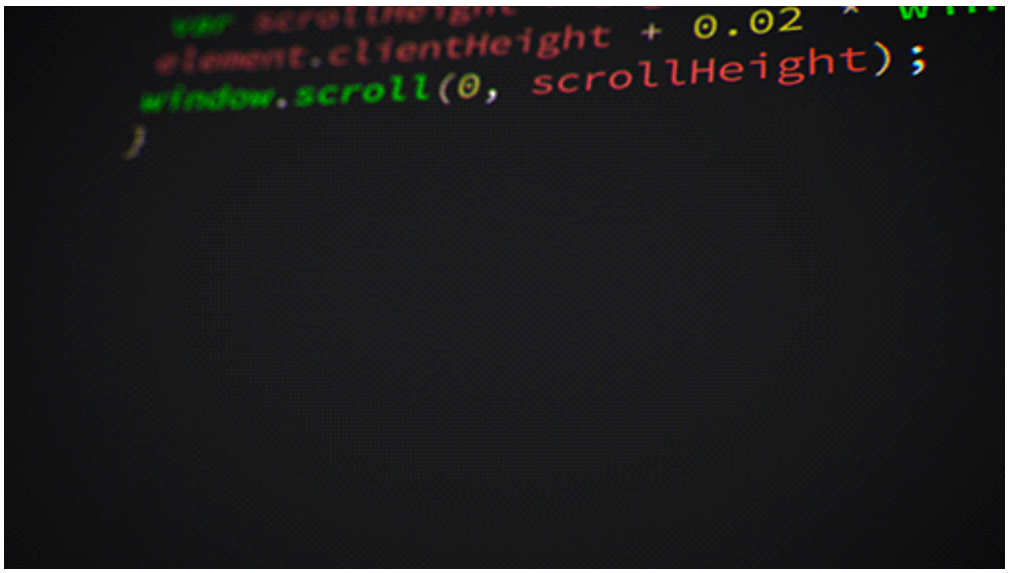
Grundlagen der Sprache C++

Parameter	Kursinformationen
Veranstaltung:	`Prozedurale Programmierung / Einführung in die Informatik / Erhebung, Analyse und Visualisierung digitaler Daten`
Semester	`Wintersemester 2025/26`
Hochschule:	`Technische Universität Freiberg`
Inhalte:	`Funktionen`
Link auf Repository:	https://github.com/TUBAF-IfI-LiaScript/VL_EAVD/blob/master/04_Funktionen.md
Autoren	Sebastian Zug & André Dietrich & Galina Rudolf & Copilot



Fragen an die heutige Veranstaltung ...

- Welche Komponenten beschreiben Definition einer Funktion?
- Wozu werden in der Funktion die Parameter gebraucht?
- Wann ist es sinnvoll Referenzen-Parameter zu verwenden?
- Warum ist es sinnvoll Funktionen in Look-Up-Tables abzubilden, letztendlich kostet das Ganze doch Speicherplatz?

Reflexion Ihrer Fragen / Rückmeldungen

> Zur Erinnerung ... Wettstreit zur partizipativen Materialentwicklung mit den Informatikern ...

Preis für das aktivste Auditorium

Format	Informatik Studierende
Verbesserungsvorschlag	2
Fragen	2
generelle Hinweise	0

Einlesen eines Arrays aus einer Datei

In der Praxis liegen Daten oft nicht direkt im Quellcode, sondern werden aus einer Datei eingelesen. Das ist zum Beispiel nützlich, wenn Messwerte, Konfigurationsdaten oder größere Datenmengen verarbeitet werden sollen.

Beispiel: Angenommen, in der Datei `daten.txt` stehen Zahlen, jeweils durch Leerzeichen getrennt:

```
3 7 2 9 4 1 5
```

Das folgende C++-Programm liest die Zahlen aus der Datei in ein Array ein:

// array_from_file.cpp

```cpp
#include <iostream>
#include <fstream>

int main() {
  std::ifstream file("daten.txt");
  const int MAX = 100;
  int arr[MAX];
  int n = 0;
  while (file >> arr[n] && n < MAX) {
    n++;
  }
  std::cout << "Gelesene Werte:";
  for (int i = 0; i < n; i++) {
    std::cout << " " << arr[i];
  }
  std::cout << std::endl;
  return 0;
}
```

Hinweis: - Die Datei `daten.txt` muss im gleichen Verzeichnis wie das Programm liegen. - Die Schleife liest so lange Zahlen ein, bis das Dateiende erreicht ist oder das Array voll ist.

Motivation

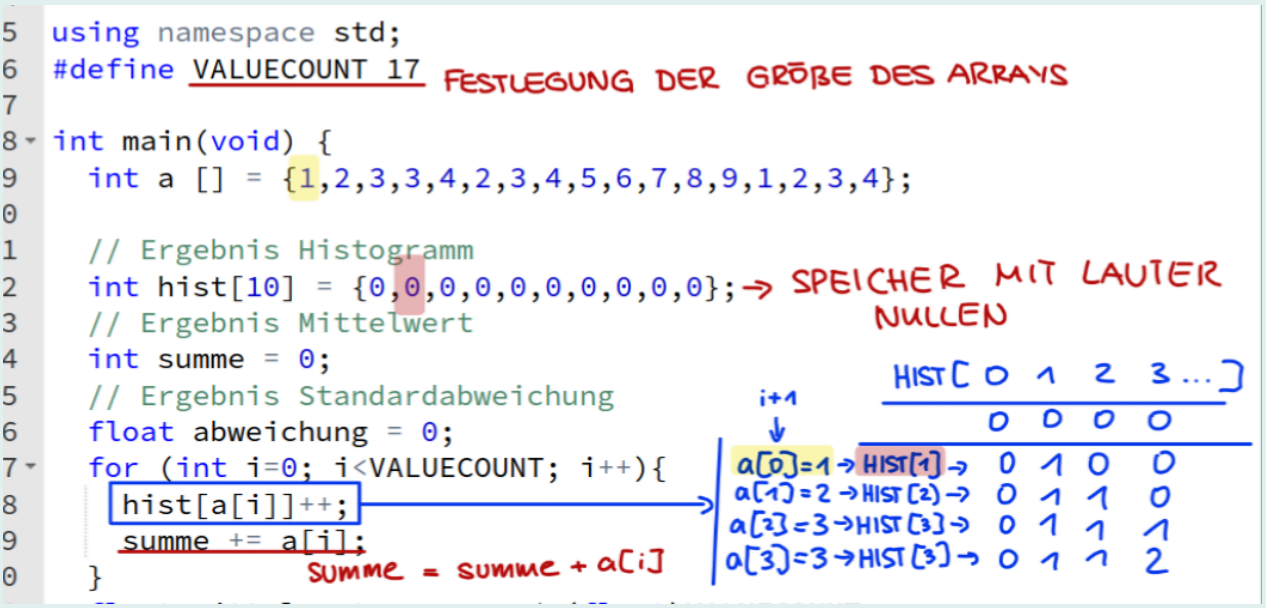
Erklären Sie die Idee hinter folgendem Code.

onBlock.cpp

```cpp
#include <iostream>
#include <cmath>
#include <iomanip>

#define VALUECOUNT 17

int main(void) {
  int a [] = {1,2,3,3,4,2,3,4,5,6,7,8,9,1,2,3,4};

  // Ergebnis Histogramm
  int hist[10] = {0,0,0,0,0,0,0,0,0,0};
  // Ergebnis Mittelwert
  int summe = 0;
  // Ergebnis Standardabweichung
  float abweichung = 0;
  for (int i=0; i<VALUECOUNT; i++) {
    hist[a[i]]++;
    summe += a[i];
  }
  float mittelwert = summe / (float)VALUECOUNT;
  for (int i=0; i<VALUECOUNT; i++) {
    abweichung += pow((a[i]-mittelwert),2.);
  }
  // Ausgabe
  for (int i=0; i<12; i++) {
      std::cout << std::setw(2) <<  i << " ";
  }
  std::cout << std::endl;
  for (int i=0; i<12; i++) {
      std::cout << std::setw(2) << a[i] << " ";
  }
  std::cout << std::endl;
  // Ausgabe Mittelwert
  std::cout << "Die Summe betraegt " << summe << ", der Mittelwert "
    mittelwert << "\n";
  // Ausgabe Standardabweichung
  float stdabw = sqrt(abweichung / VALUECOUNT);
  std::cout << "Die Standardabweichung der Grundgesamtheit betraegt "
    stdabw << "\n";
  return 0;
}
```

```
 0  1  2  3  4  5  6  7  8  9 10 11
 1  2  3  3  4  2  3  4  5  6  7  8
Die Summe betraegt 67, der Mittelwert 3.94118
Die Standardabweichung der Grundgesamtheit betraegt 2.28732
 0  1  2  3  4  5  6  7  8  9 10 11
 1  2  3  3  4  2  3  4  5  6  7  8
Die Summe betraegt 67, der Mittelwert 3.94118
Die Standardabweichung der Grundgesamtheit betraegt 2.28732
```

Für die Verbesserung des Verständnisses bei der Generierung des Histogramms hat einer Ihrer Kommilitonen eine sehr anschauliche grafische Darstellung vorbereitet. Vielen Dank dafür!



Sie wollen den Code in einem neuen Projekt wiederverwenden. Was sind die Herausforderungen dabei?

Stellen Sie das Programm so um, dass es aus einzelnen Bereichen besteht und überlegen Sie, welche Variablen wo gebraucht werden.

Prozedurale Programmierung Ideen und Konzepte

Bessere Lesbarkeit

Der Quellcode eines Programms kann schnell mehrere tausend Zeilen umfassen. Beim Linux Kernel sind es sogar über 15 Millionen Zeilen und Windows, das ebenfalls zum Großteil in C geschrieben wurde, umfasst schätzungsweise auch mehrere Millionen Zeilen. Um dennoch die Lesbarkeit des Programms zu gewährleisten, ist die Modularisierung unerlässlich.

Wiederverwendbarkeit

In fast jedem Programm tauchen die gleichen Problemstellungen mehrmals auf. Oft gilt dies auch für unterschiedliche Applikationen. Da nur Parameter und Rückgabetyp für die Benutzung einer Funktion bekannt sein müssen, erleichtert dies die Wiederverwendbarkeit. Um die Implementierungsdetails muss sich der Entwickler dann nicht mehr kümmern.

Wartbarkeit

Fehler lassen sich durch die Modularisierung leichter finden und beheben. Darüber hinaus ist es leichter, weitere Funktionalitäten hinzuzufügen oder zu ändern.

> Finden Sie Fehler im zuvor gezeigten Code?

In allen 3 Aspekten ist der Vorteil in der Kapselung der Funktionalität zu suchen.

Anwendung

Funktionen sind Unterprogramme, die ein Ausgangsproblem in kleine, möglicherweise wiederverwendbare Codeelemente zerlegen.

standardabweichung.cpp

```
#include <iostream>

// Funktion für den Mittelwert
// Mittelwert = f_Mittelwert(daten)

// Funktion für die Standardabweichung
// Standardabweichung =  f_Standardabweichung(daten)

// Funktion für die Histogrammgenerierung
// Histogramm = f_Histogramm(daten)

// Funktion für die Ausgabe
// f_Ausgabe(daten, {Mittelwert, Standardabweichung, Histogramm})

int main(void) {
  int a[] = {3,4,5,6,2,3,2,5,6,7,8,10};
  // b = f_Mittelwert(a) ...
  // c = f_Standardabweichung(a) ...
  // d = f_Histogramm(a) ...
  // f_Ausgabe(a, b, c, d) ...
  return 0;
}
```

Wie findet sich diese Idee in großen Projekten wieder?

Write Short Functions

Prefer small and focused functions.

We recognize that long functions are sometimes appropriate, so no hard limit is placed on functions length. If a function exceeds about 40 lines, think about whether it can be broken up without harming the structure of the program.

Even if your long function works perfectly now, someone modifying it in a few months may add new behavior. This could result in bugs that are hard to find. Keeping your functions short and simple makes it easier for other people to read and modify your code.

You could find long and complicated functions when working with some code. Do not be intimidated by modifying existing code: if working with such a function proves to be difficult, you find that errors are hard to debug, or you want to use a piece of it in several different contexts, consider breaking up the function into smaller and more manageable pieces.

Stackoverflow

Funktionen in C++

```
Rückgabedatentyp Funktionsname([Parameterliste]) {
  /* Anweisungsblock mit Anweisungen */
  [return Rückgabewert]
}
```

- **Rückgabedatentyp** - Welchen Datentyp hat der Rückgabewert?

  Eine Funktion ohne Rückgabewert wird vom Programmierer als `void` deklariert. Sollten Sie keinen Rückgabetyp angeben, so wird automatisch eine Funktion mit Rückgabewert vom Datentyp `int` erzeugt.

- **Funktionsname** - Dieser Bestandteil der Funktionsdefinition ist eine eindeutige Bezeichnung, die für den Aufruf der Funktion verwendet wird.

  Es gelten die gleichen Regeln für die Namensvergabe wie für Variablen.

- **Parameterliste** - Parameter sind Variablen (oder Pointer bzw. Referenzen darauf) die durch einen Datentyp und einen Namen spezifiziert werden. Mehrere Parameter werden durch Kommas getrennt.

  Die Parameterliste ist optional, die Klammern jedoch nicht. Alternative zur fehlenden Parameterliste ist die Liste aus einen Parameter vom Datentyp `void` ohne Angabe des Namen.

- **Anweisungsblock** - Der Anweisungsblock umfasst die im Rahmen der Funktion auszuführenden Anweisungen und Deklarationen. Er wird durch geschweifte Klammern gekapselt.

Für die Funktionen gelten die gleichen Gültigkeits- und Sichtbarkeitsregeln wie für die Variablen.

Beispiele für Funktionsdefinitionen

```
int main (void) {
  /* Anweisungsblock mit Anweisungen */
}
```

```
double pow (double base, double exponent) {
  /* Anweisungsblock mit Anweisungen */
}

// double y = pow(25.0,0.5));
```

```
void tauschen(int &var1,int &var2) {
  /* Anweisungsblock mit Anweisungen */
}
```

```
int mittelwert(int * array) {
  /* Anweisungsblock mit Anweisungen */
}
```

Aufruf der Funktion

Merke: Die Funktion (mit der Ausnahme der `main`-Funktion) wird erst ausgeführt, wenn sie aufgerufen wird. Vor dem Aufruf muss die Funktion definiert oder deklariert werden.

Der Funktionsaufruf einer Funktionen mit dem Rückgabewert kann Teil einer Anweisung, z.B. einer Zuweisung oder einer Ausgabeanweisung.

callAFunction.cpp

```cpp
#include <iostream>
#include <cmath>

void info() {
  std::cout << "Dieses Programm rundet Zahlenwerte.\n";
  std::cout << "-----------------------------------\n";
}

int runden(float a) {
  if (a < 0)
    return (int)(a - 0.5);
  else
    return (int)(a + 0.5);
}

int main(void) {
  info();
  float input = -8.4565;
  std::cout << "Eingabewert " << input << " - Ausgabewert " << runden
    ) << "\n";
  return 0;
}
```

```
Dieses Programm rundet Zahlenwerte.
-----------------------------------
Eingabewert -8.4565 - Ausgabewert -8
Dieses Programm rundet Zahlenwerte.
-----------------------------------
Eingabewert -8.4565 - Ausgabewert -8
```

Die Funktion `runden` nutzt die Funktionalität des Cast-Operators `int` aus.

- Wenn N eine positive Zahl ist, wird 0.5 addiert
  - $15.2 + 0.5 = 15.7$ `(int)(15.7) = 15`
  - $15.7 + 0.5 = 16.2$ `(int)(16.2) = 16`
- Wenn N eine negative Zahl ist, wird 0.5 subtrahiert
  - $-15.2 - 0.5 = -15.7$ `(int)(-15.7) = -15`
  - $-15.7 - 0.5 = -16.2$ `(int)(-16.2) = -16`

Hinweis: C++ unterstützt gleiche Codenahmen bei unterschiedlichen Parametern. Der Compiler "sucht sich" die passende Funktion aus. Der Mechanismus wird als *Funktionsüberladung* bezeichnet.

callAFunction.cpp

```cpp
#include <iostream>
#include <cmath>

void info() {
  std::cout << "Dieses Programm rundet Zahlenwerte.\n";
  std::cout << "----------------------------------\n";
}

int runden(float a) {
  if (a < 0)
    return (int)(a - 0.5);
  else
    return (int)(a + 0.5);
}

float rundenf(float a, int nachkomma) {
  float shifted = a * pow(10, nachkomma);
  int result = 0;
  if (shifted < 0)
    result = int(shifted - 0.5);
  else
    result = int(shifted + 0.5);
  return (float)result * pow(10, -nachkomma);
}

int main(void) {
  info();
  float input = -8.4565;
  std::cout << "Eingabewert " << input << " - Ausgabewert " << runden
    ) << "\n";
  std::cout << "Eingabewert " << input << " - Ausgabewert " << runden
    (input,1) << "\n";
  return 0;
}
```

```
Dieses Programm rundet Zahlenwerte.
----------------------------------
Eingabewert -8.4565 - Ausgabewert -8
Eingabewert -8.4565 - Ausgabewert -8.5
Dieses Programm rundet Zahlenwerte.
----------------------------------
Eingabewert -8.4565 - Ausgabewert -8
Eingabewert -8.4565 - Ausgabewert -8.5
```

Welche Verbesserungsmöglichkeit sehen Sie bei dem Programm? Tipp: Wie können wir den redundanten Code eliminieren?

Fehler

Rückgabewert ohne Rückgabedefintion

return.cpp

```
1  void foo() {
2    /* Code */
3    return 5; /* Fehler */
4  }
5
6  int main(void) {
7    foo();
8    return 0;
9  }
```

```
main.cpp: In function ‘void foo()’:
main.cpp:3:10: error: return-statement with a value, in function
returning ‘void’ [-fpermissive]
    3 |   return 5; /* Fehler */
      |          ^
main.cpp: In function ‘void foo()’:
main.cpp:3:10: error: return-statement with a value, in function
returning ‘void’ [-fpermissive]
    3 |   return 5; /* Fehler */
      |          ^
```

Erwartung eines Rückgabewertes

returnII.cpp

```cpp
#include <iostream>

void foo() {
  std::cout << "Ausgabe";
}

int main(void) {
  int i = foo();
  return 0;
}
```

```
main.cpp: In function ‘int main()’:
main.cpp:8:14: error: void value not ignored as it ought to be
    8 |   int i = foo();
      |           ~~~^~
main.cpp:8:7: warning: unused variable ‘i’ [-Wunused-variable]
    8 |   int i = foo();
      |       ^
main.cpp: In function ‘int main()’:
main.cpp:8:14: error: void value not ignored as it ought to be
    8 |   int i = foo();
      |           ~~~^~
main.cpp:8:7: warning: unused variable ‘i’ [-Wunused-variable]
    8 |   int i = foo();
      |       ^
```

Falscher Rückgabetyp

conversion.cpp

```cpp
#include <iostream>

float pi() {
  return 3.1415926f;
}

int main(void) {
  int i = pi();
  std::cout << i << "\n";
  return 0;
}
```

```
main.cpp: In function ‘int main()’:
main.cpp:8:13: warning: conversion from ‘float’ to ‘int’ may change
value [-Wfloat-conversion]
    8 |   int i = pi();
      |           ~~^~

main.cpp: In function ‘int main()’:
main.cpp:8:13: warning: conversion from ‘float’ to ‘int’ may change
value [-Wfloat-conversion]
    8 |   int i = pi();
      |           ~~^~


3
```

Parameterübergabe ohne entsprechende Spezifikation

paramters.cpp

```cpp
#include <iostream>

int foo(void) {       // <- Die Funktion erwartet explizit keine Para
  return 3;
}

int main(void) {
  int i = foo(5);
  return 0;
}
```

```
main.cpp: In function ‘int main()’:
main.cpp:8:14: error: too many arguments to function ‘int foo()’
    8 |   int i = foo(5);
      |           ~~~^~~
main.cpp:3:5: note: declared here
    3 | int foo(void) {       // <- Die Funktion erwartet explizit
keine Parameter
      |     ^~~
main.cpp:8:7: warning: unused variable ‘i’ [-Wunused-variable]
    8 |   int i = foo(5);
      |       ^
main.cpp: In function ‘int main()’:
main.cpp:8:14: error: too many arguments to function ‘int foo()’
    8 |   int i = foo(5);
      |           ~~~^~~
main.cpp:3:5: note: declared here
    3 | int foo(void) {       // <- Die Funktion erwartet explizit
keine Parameter
      |     ^~~
main.cpp:8:7: warning: unused variable ‘i’ [-Wunused-variable]
    8 |   int i = foo(5);
      |       ^
```

Anweisungen nach dem return-Schlüsselwort

codeOrder.cpp

```cpp
#include <iostream>

int foo() {
  return 5;
  std::cout << "foo!\n";   // Wird nie erreicht!
}

int main(void) {
  int i = foo();
  std::cout << i << "\n";
}
```

```
5
5
```

Falsche Reihenfolgen der Parameter

conversion.cpp

```cpp
#include <iostream>

void foo(int index, float wert) {
  std::cout << "Index   -  Wert\n";
  std::cout << index << "   - " << wert << "\n\n";
}

int main(void) {
  foo(4, 6.5);
  foo(6.5, 4);
  return 0;
}
```

```
main.cpp: In function ‘int main()’:
main.cpp:10:7: warning: conversion from ‘double’ to ‘int’ changes value
from ‘6.5e+0’ to ‘6’ [-Wfloat-conversion]
   10 |   foo(6.5, 4);
      |       ^~~

main.cpp: In function ‘int main()’:
main.cpp:10:7: warning: conversion from ‘double’ to ‘int’ changes value
from ‘6.5e+0’ to ‘6’ [-Wfloat-conversion]
   10 |   foo(6.5, 4);
      |       ^~~

Index   -  Wert
4   - 6.5

Index   -  Wert
6   - 4

Index   -  Wert
4   - 6.5

Index   -  Wert
6   - 4
```

Funktionsdeklaration

experiments.cpp

```cpp
#include <iostream>

// int foo(void); // Explizite Einführung der Funktion foo()

int main(void) {
  int i = foo();        // <- Aufruf der Funktion
  std::cout << "i=" << i << "\n";
  return 0;
}

int foo(void) {          // <- Definition der Funktion
  return 3;
}
```

```
main.cpp: In function ‘int main()’:
main.cpp:6:11: error: ‘foo’ was not declared in this scope
    6 |   int i = foo();       // <- Aufruf der Funktion
      |           ^~~
main.cpp: In function ‘int main()’:
main.cpp:6:11: error: ‘foo’ was not declared in this scope
    6 |   int i = foo();       // <- Aufruf der Funktion
      |           ^~~
```

Damit der Compiler überhaupt von einer Funktion Kenntnis nimmt, muss diese vor ihrem Aufruf bekannt gegeben werden. Im vorangegangenen Beispiel wird die die Funktion erst nach dem Aufruf definiert. Der Complier zeigt dies an.

Das Ganze wird dann relevant, wenn Funktionen aus anderen Quellcodedateien eingefügt werden sollen. Die Deklaration macht den Compiler mit dem Aussehen der Funktion bekannt. Diese werden mit dem Schlüsselwort `extern` markiert.

```cpp
extern float berechneFlaeche(float breite, float hoehe);
```

Parameterübergabe und Rückgabewerte

Bisher wurden Funktionen betrachtet, die skalare Werte als Parameter erhielten und ebenfalls einen skalaren Wert als einen Rückgabewert lieferten. Allerdings ist diese Möglichkeit sehr einschränkend.

Student.cpp

```cpp
#include <iostream>

#include <iostream>

//           Rückgabe der Ergebnisse
//  +-----------------------------------+
//  |                                   |
int add(int a, int b) { // <--+         |
  return a + b;         //    | Aufruf |
}                       //    | mit    |
                        //    | Para-  |
int main(void) {        //    | metern |
  int i = add(5, 6);    //----+         |
//   ^                                  |
//   +----------------------------------+
  std::cout << "i=" << i << "\n";
  return 0;
}
```

```
i=11
i=11
```

Es wird in vielen Programmiersprachen, darunter in C/C++, zwei generelle Konzepte der Parameterübergabe realisiert.

Call-by-Value

In allen Beispielen bis jetzt wurden Parameter an die Funktionen *call-by-value*, übergeben. Das bedeutet, dass innerhalb der aufgerufenen Funktion mit einer Kopie der Variable gearbeitet wird und die Änderungen sich nicht auf den ursprünglichen Wert auswirken.

Student.cpp

```cpp
#include <iostream>

void doSomething(int a) {
  // eine KOPIE von a wird um 1 erhöht
  std::cout << ++a << " a in der Funktion\n";
}

int main(void) {
  int a = 5;
  std::cout << a << " a in main\n";
  doSomething(a);
  std::cout << a << " a in main\n";
  return 0;
}
```

```
5 a in main
6 a in der Funktion
5 a in main
5 a in main
6 a in der Funktion
5 a in main
```

Merke: Die *Call-by-value*-Funktionen können den Wert der äußeren Variablen nicht verändern.

Call-by-Reference

Bei einer Übergabe als Referenz wirken sich Änderungen an den Parametern auf die ursprünglichen Werte aus, es werden keine Kopien von Parametern angelegt. *Call-by-reference* wird unbedingt notwendig, wenn eine Funktion mehrere Rückgabewerte hat.

In C++ kann die "call-by-reference"- Parameterübergabe mit Hilfe der Referenzen oder Pointern

realisiert werden.

Parameter_Reference_I.cpp

```cpp
#include <iostream>

void inkrementieren(int &variable) {
  variable++;
}

int main(void) {
  int a = 0;
  inkrementieren(a);
  std::cout << "a = " << a << "\n";
  inkrementieren(a);
  std::cout << "a = " << a << "\n";
  return 0;
}
```

```
a = 1
a = 2
a = 1
a = 2
```

Vgl. Illustration mit Python-Tutor Link

Der Vorteil der Verwendung der Referenzen als Parameter besteht darin, dass in der Funktion mehrere Variablen auf eine elegante Weise verändert werden können. Die Funktion hat somit quasi mehrere Ergebnisse.

Parameter_Reference_II.cpp

```cpp
#include <iostream>

void tauschen(char &anna, char &hanna) {
  char aux = anna;
  anna = hanna;
  hanna = aux;
}

int main(void) {
  char anna = 'A', hanna = 'H';
  std::cout << anna << " und " << hanna << "\n";
  tauschen(anna,hanna);
  std::cout << anna << " und " << hanna << "\n";
  return 0;
}
```

```
A und H
H und A
A und H
H und A
```

Es besteht ebenfalls die Möglichkeit, "call-by-reference"- Parameterübergabe mit Hilfe der Zeiger (Pointer) zu realisieren. Allerdings muss dann im Unterschied zur Referenz jeweils eine Dereferenzierung vorgenommen werden `*a = ...`.

Parameter_Pointer_I.cpp

```cpp
#include <iostream>
#include <cmath>

void runden(float* a) {
  if (*a < 0)
    *a = (int)(*a - 0.5);
  else
    *a = (int)(*a + 0.5);
}

int main(void) {
  float value = -8.4565;
  float *pointer = &value;
  std::cout << "Eingabewert " << value << " - Ausgabewert ";
  runden(pointer);
  std::cout << *pointer << "\n";
  return 0;
}
```

```
Eingabewert -8.4565 - Ausgabewert -8
Eingabewert -8.4565 - Ausgabewert -8
```

Ein realistisches Beispiel könnte die Verwendung eines Arrays sein.

> Zur Erinnerung Eine Variable, die ein Array representiert zeigt auf den ersten Eintrag.

Dabei können zwei Varianten genutzt werden - die explizte Pointerschreibweise und die Array Schreibweise - beide drücken die Übergabe eines Pointers im Sinne von *Call-by-Reference* aus.

Parameter_Pointer_II.cpp

```cpp
#include <iostream>
#include <cmath>

//                  Variante 1
double hypothenuse(double *lookup_sin, int winkel, double gegenkathet
  return gegenkathete / lookup_sin[winkel];
}
/*
                    Variante 2
double hypothenuse(double lookup_sin[], int winkel, double gegenkathe
  return gegenkathete / lookup_sin[winkel];
}
*/

int main(void) {
  double sin_values[360] = {0};
  for (int i=0; i<360; i++) {
    sin_values[i] = sin(i * M_PI / 180);
  }
  std::cout << "Größe des Arrays " << sizeof(sin_values) / sizeof(dou
    << "\n";
  std::cout << "Result =  " << hypothenuse(sin_values, 30, 20) << " \
  return 0;
}
```

```
Größe des Arrays 360
Result =  40
Größe des Arrays 360
Result =  40
```

Die Visualisierung des Zugriffs finden Sie in einem Python-Tutor Beispiel.

Zeiger und Referenzen als Rückgabewerte

Analog zur Bereitstellung von Parametern entsprechend dem "call-by-reference" Konzept können auch Rückgabewerte als Pointer oder Referenz vorgesehen sein. Allerdings sollen Sie dabei aufpassen ...

returnReferenz.cpp

```cpp
#include <iostream>

int& doCalc(int &wert) {
  int a = wert + 5;
  return a;
}

int main(void) {
  int b = 5;
  std::cout << "Irgendwas stimmt nicht " << doCalc(b) << "\n";
  return 0;
}
```

```
main.cpp: In function ‘int& doCalc(int&)’:
main.cpp:5:10: warning: reference to local variable ‘a’ returned [-
Wreturn-local-addr]
    5 |   return a;
      |          ^
main.cpp:4:7: note: declared here
    4 |   int a = wert + 5;
      |       ^

main.cpp: In function ‘int& doCalc(int&)’:
main.cpp:5:10: warning: reference to local variable ‘a’ returned [-
Wreturn-local-addr]
    5 |   return a;
      |          ^
main.cpp:4:7: note: declared here
    4 |   int a = wert + 5;
      |       ^
```

Mit dem Beenden der Funktion werden deren lokale Variablen vom Stack gelöscht. Um diese Situation zu handhaben können Sie mehrere Lösungsansätze realisieren.

Variante 1 Sie übergeben den Rückgabewert in der Parameterliste.

ReferenzAsParameter.cpp

```cpp
#include <iostream>
#include <cmath>

void kreisflaeche(double durchmesser, double &flaeche) {
  flaeche = M_PI * pow(durchmesser / 2, 2);
  // Hier steht kein return !
}

int main(void) {
  double wert = 5.0;
  double flaeche = 0;
  kreisflaeche(wert, flaeche);
  std::cout << "Die Kreisfläche beträgt für d=" << wert << "[m] " <<
    flaeche << "[m²] \n";
  return 0;
}
```

```
Die Kreisfläche beträgt für d=5[m] 19.635[m²]
Die Kreisfläche beträgt für d=5[m] 19.635[m²]
```

Variante 2 Für den Rückgabezeiger wird der Speicherplatz mit `new` dynamisch angelegt, aber Achtung: zu jedem new gehört ein `delete`.

PointerInsteadOfReturnII.cpp

```cpp
#include <iostream>
#include <cmath>

double* kreisflaeche(double durchmesser) {
  double *flaeche = new double;
  *flaeche = M_PI * pow(durchmesser / 2, 2);
  return flaeche;
}

int main(void) {
  double wert = 5.0;
  double *flaeche;
  flaeche=kreisflaeche(wert);
  std::cout << "Die Kreisfläche beträgt für d=" << wert << "[m] " <<
    *flaeche << "[m²] \n";
  delete flaeche;
  return 0;
}
```

```
Die Kreisfläche beträgt für d=5[m] 19.635[m²]
Die Kreisfläche beträgt für d=5[m] 19.635[m²]
```

Variante 3 Sie geben den referenzierten Parameter zurück.

ReferencedParameter.cpp

```cpp
#include <iostream>

struct Foo {
  int index;
  double value;
};
std::ostream &operator<<(std::ostream &stream, const Foo &foo) {
  stream << "Foo{index = " << foo.index << ", value = " << foo.value
    ;
  return stream;
}

int main(void) {
  Foo a{0, 3.1415926};
  std::cout << a << "\n";
}
```

```
Foo{index = 0, value = 3.14159}
Foo{index = 0, value = 3.14159}
```

Besonderheit Arrays

conversion.c

```
#include <iostream>
#include <cstdlib>

void printSizeOf(int intArray[]) {
  std::cout << "sizeof of array as parameter: " << sizeof(intArray) <
    ;
}

void printLength(int intArray[]) {
  std::cout << "Length of parameter: " << sizeof(intArray) / sizeof
    (intArray[0]) << "\n";
}

int main() {
  int array[] = { 0, 1, 2, 3, 4, 5, 6 };

  std::cout << "sizeof of array: " << sizeof(array) << "\n";
  printSizeOf(array);

  std::cout << "Length of array: " << sizeof(array) / sizeof(array[0]
    "\n";
  printLength(array);

  std::cout << "Size of int pointer: " << sizeof(int*) << "\n";
}
```

```
main.cpp: In function ‘void printSizeOf(int*)’:
main.cpp:5:59: warning: ‘sizeof’ on array function parameter ‘intArray’
will return size of ‘int*’ [-Wsizeof-array-argument]
    5 |   std::cout << "sizeof of array as parameter: " <<
sizeof(intArray) << "\n";
      |
~^~~~~~~~~
main.cpp:4:22: note: declared here
    4 | void printSizeOf(int intArray[]) {
      |                  ~~~~^~~~~~~~~~
main.cpp: In function ‘void printLength(int*)’:
main.cpp:9:50: warning: ‘sizeof’ on array function parameter ‘intArray’
will return size of ‘int*’ [-Wsizeof-array-argument]
    9 |   std::cout << "Length of parameter: " << sizeof(intArray) /
sizeof(intArray[0]) << "\n";
      |                                                 ~^~~~~~~~~
main.cpp:8:22: note: declared here
    8 | void printLength(int intArray[]) {
      |                  ~~~~^~~~~~~~~~

main.cpp: In function ‘void printSizeOf(int*)’:
main.cpp:5:59: warning: ‘sizeof’ on array function parameter ‘intArray’
will return size of ‘int*’ [-Wsizeof-array-argument]
    5 |   std::cout << "sizeof of array as parameter: " <<
sizeof(intArray) << "\n";
      |
~^~~~~~~~~
main.cpp:4:22: note: declared here
    4 | void printSizeOf(int intArray[]) {
      |                  ~~~~^~~~~~~~~~
main.cpp: In function ‘void printLength(int*)’:
main.cpp:9:50: warning: ‘sizeof’ on array function parameter ‘intArray’
will return size of ‘int*’ [-Wsizeof-array-argument]
    9 |   std::cout << "Length of parameter: " << sizeof(intArray) /
sizeof(intArray[0]) << "\n";
      |                                                 ~^~~~~~~~~
main.cpp:8:22: note: declared here
    8 | void printLength(int intArray[]) {
      |                  ~~~~^~~~~~~~~~

sizeof of array: 28
sizeof of array as parameter: 8
Length of array: 7
```

```
Length of parameter: 2
Size of int pointer: 8
sizeof of array: 28
sizeof of array as parameter: 8
Length of array: 7
Length of parameter: 2
Size of int pointer: 8
```

`main`-Funktion

In jedem Programm muss und darf nur eine `main`-Funktion geben. Diese Funktion wird beim Programmstart automatisch ausgeführt.

Definition der `main`-Funktion:

```
int main(void) {
  /*Anweisungen*/
}
```

```
int main(int argc, char *argv[]) {
  /*Anweisungen*/
}
```

Die Bezeichner `argc` und `argv` sind traditionell, können aber beliebig gewählt werden. `argc` ist die Anzahl der Argumente, die von den Benutzern des Programms in der Kommandozeile angegeben werden. Der argc-Parameter ist immer größer als oder gleich 1. `argv` ist ein Array von Befehlszeilenargumenten, wobei `argv[0]` das Programm selbst und `argv[argc]` immer NULL ist.

Im Beispiel wird die kompilierte Version von mainArgumente.cpp intern mit `./a.out 1 2 3 aus die Maus` aufgerufen.

mainArgumente.cpp

```
1  #include <iostream>
2
3  int main(int argc, char *argv[]) {
4    for (int i=0;i<argc;i++)
5      std::cout << argv[i] << " ";
6    return 0;
7  }
```

```
./a.out 1 2 3 aus die Maus ./a.out 1 2 3 aus die Maus
```

Der integrale Rückgabewert stellt traditionell einen Statuswert da, welcher ausdrückt, ob das Programm ohne Probleme lief (0) oder Fehler aufgetreten sind (1, 2, 3...). Die genaue Bedeutung letzterer Werte ist dabei dem Entwickler überlassen.

Zusammenfassung: Parameterübergabe in C++

Übergabeart	Syntax im Funktionskopf	Was wird übergeben?	Auswirkungen auf Originalwert?	Typische Anwendung
Call-by-Value	`int foo(int x)`	Kopie des Werts	Nein	Reine Berechnung, keine Änderung am Original
Call-by-Reference	`int foo(int &x)`	Alias/Referenz auf Original	Ja	Wenn Funktion Wert verändern soll, mehrere Rückgaben
Call-by-Pointer	`int foo(int *x)`	Adresse (Pointer)	Ja (über Dereferenzierung)	Dynamische Daten, optionale Übergabe, Arrays

Merke:

- Call-by-Value ist sicher, aber Änderungen wirken sich nicht auf die Ursprungsvariable aus.
- Call-by-Reference ist elegant, wenn der Wert verändert werden soll.
- Call-by-Pointer ist flexibel, aber fehleranfällig (z. B. bei nullptr).

Beispiel des Tages

+ Functions.cpp

```cpp
#include <iostream>
#include <cmath>
#include <iomanip>


double Mittelwert(int values[], int entries);
int Summe(int *values, int entries);
void Ausgabe(int *values, int entries);

double Mittelwert(int values[], int entries) {
  double summe = Summe(values, entries);
  return summe / entries;
}

int Summe(int values[], int entries) {
  int summe = 0;
  for (int i=0; i<entries; i++) {
    summe += values[i];
  }
  return summe;
}

void Ausgabe(int values[], int entries) {
  for (int i=0; i<entries; i++) {
    std::cout << std::setw(2) <<  i << " ";
  }
  std::cout << "\n";
  for (int i=0; i<entries; i++) {
    std::cout << std::setw(2) << values[i] << " ";
  }
  std::cout << "\n";
}

int main(void) {
  int a [] = {1, 2, 3, 3, 4, 2, 3, 4, 5, 6, 7, 8, 9, 1, 2, 3, 3};
  const int entries = sizeof(a)/sizeof(a[0]);
  Ausgabe(a, entries);

  int summe = Summe(a, entries);
  double mittelwert = Mittelwert(a, entries);
  std::cout << "Die Summe betraegt " << summe << ", der Mittelwert "
    mittelwert << "\n";
  return 0;
}
```

```
 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16
 1  2  3  3  4  2  3  4  5  6  7  8  9  1  2  3  3
Die Summe betraegt 66, der Mittelwert 3.88235
 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16
 1  2  3  3  4  2  3  4  5  6  7  8  9  1  2  3  3
Die Summe betraegt 66, der Mittelwert 3.88235
```

Quiz

Funktionen

Funktionsdefinitionen

Wechen Rückgabewert liefert eine als `void` deklarierte Funktion?

- ◯ Ganzzahlen
- ◯ Fließkommazahlen
- ◯ Zeichenketten
- ◯ Arrays
- ◯ Es wird kein Wert zurückgegeben.

Welcher Datentyp wird automatisch als Rückgabewert ausgewählt, wenn Sie keinen Rückgabetyp angeben?

- ◯ `void`
- ◯ `int`
- ◯ `float`
- ◯ `double`
- ◯ `char`
- ◯ `boolean`

Muss die Parameterliste einer Funktionen wenigstens einen Parameter enthalten?

- ◯ Ja
- ◯ Nein

Aufruf von Fuktionen

Wodurch muss `[_____]` ersetzt werden, damit die Funktion `divi` ermittelt ob `a` ein Teiler von `b` ist? Die Lösung ist ohne Leerzeichen einzugeben.

```
#include <iostream>

bool divi(int x, int y) {
  if(x%y == 0)
    return true;
  else
    return false;
}

int main() {
  int a = 11;
  int b = 1001;
  bool bdiv = [_____]
  if (bdiv == 1)
    std::cout <<  a << " ist ein Teiler von " << b << "." << "\n";
  else
    std::cout <<  a << " ist kein Teiler von " << b << "." << "\n";
}
```

Fehler

ist dieses Programm fehlerfrei?

```
#include <iostream>

int foo() {
  return 42;
}

int main(void) {
  int i = foo();
  return 0;
}
```

- ◯ Ja
- ◯ Nein

Ist dieses Programm fehlerfrei?

```
#include <iostream>

void foo() {
  return 42;
}

int main(void) {
  foo();
  return 0;
}
```

- ◯ Ja
- ◯ Nein

Welche Fehler liegen bei diesem Programm vor?

```
#include <iostream>

void foo(int index, float wert) {
```

```
  std::cout << "Index   -  Wert\n";
  std::cout << index << "   - " << wert << "\n\n";
  return index;
}

int main(void) {
  float f = foo(6.5, 4);
  return 0;
}
```

- [ ] Datentypen der Parameter beim Aufruf und der Defintion stimmen nicht überein
- [ ] Rückgabewert ohne Definition des Rückgabetyps
- [ ] Anweisung nach dem `return` Schlüsselwort

Funktionsdeklaration

Ersetzen Sie `[_____]` durch eine explizite Deklaration der Funktion `hw`.

```
#include <iostream>

[_____]

int main(void) {
  hw();
  return 0;
}

void hw(void) {
  std::cout << "Hello World!" << endl;
  return;
}
```

Mit welcher dieser Anweisungen kann eine Funktion aus einer anderen Quellcodedatei einzufügt werden?

- ◯ `extern int x(int y, bool z);`
- ◯ `import int x(int y, bool z);`
- ◯ `using int x(int y, bool z);`

Parameterübergabe und Rückgabewerte

Ordnen Sie die Eigenschaften den entsprechenden Arten der Parameterübergabe zu.

call-by-value	*call-by-reference*	
◯	◯	Ermöglicht mehrere Rückgabewerte
◯	◯	Arbeitet mit einer Kopie der Variablen
◯	◯	Beeinflusst nicht den tatsächlichen Wert von Variablen in der `main`
◯	◯	Beeinflusst den tatsächlichen Wert von Variablen in der `main`

Wie lautet die Ausgabe dieses Programms?

```
#include <iostream>

void f_a(int &variable) {
  variable++;
}

void f_b(int variable) {
  variable--;
}

void f_c(int &variable) {
  variable = 18;
}
```

```
int main(void) {
  int a = 0;
  f_a(a);
  f_c(a);
  f_b(a);
  f_b(a);
  f_a(a);
  std::cout << a;
  return 0;
}
```

Womit werden Array-Parameter übergeben?

- ◯ Referenz
- ◯ Zeiger

Zeiger und Referenzen als Rückgabewerte

Wo liegt der Fehler im folgenden Programm?

```
#include <iostream>

int& doCalc(int &wert) {
  int a = wert++;
  return a;
}

int main(void) {
  int b = 0;
  std::cout << doCalc(b);
  return 0;
}
```

- ◯ Die mit `return` übergebene Referenz zeigt außerhalb der Funktion `doCalc` auf einen nicht existierenden Speicherplatz
- ◯ Referenzen dürfen nicht für die Rückgabe mit `return` verwendet werden

Für die Variable `volumen` soll der Speicherplatz dynamisch zur Verfügung gestellt werden. Ersetzen Sie `[_____]` um die notwendige Ergängung der Anweisung.

```
#include <iostream>
#include <cmath>

double* kugelvolumen(double durchmesser) {
  double *volumen = [_____]
  *volumen = (4.0/3.0) * M_PI * pow(durchmesser / 2, 3);
  return volumen;
}

int main(void) {
  double wert = 5.0;
  double *volumen;
  volumen = kugelvolumen(wert);
  std::cout << "Das Kugelvolumen beträgt für d=" << wert << "[m] " << *volu
    "[m³] \n";
  delete volumen;
  return 0;
}
```

`main`-Funktion

Beurteilen Sie ob folgende Aussagen wahr oder falsch sind.

Wahr	Falsch	
◯	◯	In jedem Programm darf es nur eine (1) `main`-Fuktion geben.
◯	◯	Solange alle Funktionen `void` zurückgeben darf es auch mehrere `main`-Funktionen geben.
◯	◯	`argc` wird als erstes Argument in der Befehlszeile übergeben und kann alle ganzzahligen positiven Werte grösser 0 annehmen.
◯	◯	`argc` ist ein Array von Befehlszeilenargumenten.
◯	◯	`argv` ist ein Array von Befehlszeilenargumenten.
◯	◯	`argv[0]` ist das Programm selbst.

Was ist `argv[argc]`?

- ◯ NULL-Zeiger
- ◯ Das letzte Argument in der Befehlszeile

Wie lautet die Ausgabe dieses Programms wenn die kompilierte Version des Programms intern mit `./a.out 3 2 1 Maus im Haus` aufgerufen wird?

```
#include <iostream>

int main(int argc, char *argv[]) {
  std::cout << argv[4];
  return 0;
}
```