Grundlagen der Sprache C++

Parameter	Kursinformationen
Veranstaltung:	`Prozedurale Programmierung / Einführung in die Informatik / Erhebung, Analyse und Visualisierung digitaler Daten`
Semester	`Wintersemester 2023/24`
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

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?

Einschub - Klausurhinweise

- Während der Klausur können Sie "alle Hilfsmittel aus Papier" verwenden!

- Im OPAL finden sich Klausurbeispiele.

Beispielhafte Klausuraufgabe aus dem vergangenen Jahr

Die Zustimmung (in Prozent) für die Verwendung der künstlichen Intelligenz im Pflegebereich unter der Bevölkerung von Mauritius und Réunion soll vergleichend betrachtet werden. Die Ergebnisse der Umfragen für die Jahre 2010 bis 2020 (je ein Wert pro Jahr) in zwei Arrays erfasst werden (je ein Array pro Insel) und in einem Programm ausgewertet werden.

- *Für beide Inseln soll aus den in Arrays erfassten Werten je ein Mittelwert berechnet werden. Schreiben Sie dazu eine Funktion, die ein Array übergeben bekommt und einen Mittelwert als ein Ergebnis an die main-Funktion zurück liefert. Rufen Sie die Funktion in der main-Funktion für jedes beider Arrays auf und geben Sie die Mittelwerte in der main-Funktion aus.*
- *Schreiben Sie eine weitere Funktion, die die korrespondierenden Werte beider Arrays miteinander vergleicht. Geben Sie für jedes Jahr aus, auf welcher Insel die Zustimmung größer war, bei den gleichen Werte ist eine entsprechende Meldung auszugeben. Rufen Sie die Funktion in der main-Funktion auf.*
- *In der main()-Funktion sind die Werte von der Console einzulesen und in die Arrays zu speichern.*

Motivation

Erklären Sie die Idee hinter folgendem Code.

onBlock.cpp

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

using namespace std;
#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<10; i++){
    printf("%d - %d\n", i, hist[i]);
  }
  // Ausgabe Mittelwert
  cout<<"Die Summe betraegt "<<summe<<", der Mittelwert "<<mittelwert
    ;
  // Ausgabe Standardabweichung
  float stdabw = sqrt(abweichung / VALUECOUNT);
  cout<<"Die Standardabweichung der Grundgesamtheit betraegt "<<stdab
    <<"\n";
  return 0;
}
```

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

Ihre Aufgabe besteht nun darin ein neues Programm zu schreiben, das Ihre Implementierung der Mittelwertbestimmung integriert. Wie gehen Sie vor? 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.

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>

using namespace std;
// 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.*
>
> Google Style Guide für C++ Projekte

C++ Funktionen

Funktionsdefinition

```
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.

  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 */
}

//int 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>
using namespace std;

void info(){
    cout<<"Dieses Programm rundet Zahlenwerte.\n";
    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;
    cout<<"Eingabewert "<<input<<" - Ausgabewert "<<runden(input)<<"\
    cout<<"Eingabewert "<<input<<" - Ausgabewert "<<rundenf(input,1)<
    return 0;
}
```

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

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`

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

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

Fehler

Rückgabewert ohne Rückgabedefintion

return.cpp

```
 1  void foo()
 2  {
 3    /* Code */
 4    return 5; /* Fehler */
 5  }
 6
 7  int main(void)
 8  {
 9    foo()
10    return 0;
11  }
```

```
main.cpp: In function ‘void foo()’:
main.cpp:4:16: error: return-statement with a value, in function
returning ‘void’ [-fpermissive]
    4 |         return 5; /* Fehler */
      |                ^
main.cpp: In function ‘int main()’:
main.cpp:9:8: error: expected ‘;’ before ‘return’
    9 |   foo()
      |        ^
      |        ;
   10 |   return 0;
      |   ~~~~~~
```

Erwartung eines Rückgabewertes

returnII.cpp

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

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

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

```
main.cpp: In function ‘int main()’:
main.cpp:9:14: error: void value not ignored as it ought to be
    9 |   int i = foo();
      |           ~~~^~
main.cpp:9:7: warning: unused variable ‘i’ [-Wunused-variable]
    9 |   int i = foo();
      |       ^
```

Falscher Rückgabetyp

conversion.cpp

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

float foo(){
  return 3.123f;
}

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

```
main.cpp: In function ‘int main()’:
main.cpp:9:14: warning: conversion from ‘float’ to ‘int’ may change
value [-Wfloat-conversion]
    9 |   int i = foo();
      |           ~~~^~

3
```

Parameterübergabe ohne entsprechende Spezifikation

paramters.cpp

```cpp
#include <iostream>

int foo(void){        // <- Die Funktion erwartet explizit keine Param
  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);
      |       ^
```

Anweisungen nach dem return-Schlüsselwort

codeOrder.cpp

```cpp
int foo()
{
  return 5;
  /* Code */   // Wird nie erreicht!
}
```

Falsche Reihenfolgen der Parameter

conversion.cpp

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

void foo(int index, float wert){
  cout<<"Index   -  Wert\n";
  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:11:7: warning: conversion from ‘double’ to ‘int’ changes value
from ‘6.5e+0’ to ‘6’ [-Wfloat-conversion]
   11 |   foo(6.5, 4);
      |       ^~~

Index   -  Wert
4   - 6.5

Index   -  Wert
6   - 4
```

Funktionsdeklaration

experiments.cpp

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

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

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

```
main.cpp: In function ‘int main()’:
main.cpp:5:11: error: ‘foo’ was not declared in this scope
    5 |   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.

Eine explizite Deklaration zeigt folgendes Beispiel:

explicite.cpp

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

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

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

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

```
i=3
```

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.

Es wird in vielen Programmiersprachen, darunter in C/C++, zwei 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>
using namespace std;

// Definitionsteil
void doSomething(int a) { cout << ++a << " a in der Schleife\n"; }

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

```
5 a in main
6 a in der Schleife
5 a in main
```

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 unbedigt notwendig, wenn eine Funktion mehrere Rückgabewerte hat.

In C++ wird die "call-by-reference"- Parameterübergabe mit Hilfe der Referenzen realisiert. In der Liste der formalen Parameter wird eine Referenz eines passenden Typs definiert. Beim Funktionsaufruf wird als Argument eine Variable des gleichen Datentyps übergeben.

ParameterI.cpp

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

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

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

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

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.

ParameterIII.cpp

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

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

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

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

Es besteht ebenfalls die Möglichkeit, die bereits in C zur Verfügung stand, "call-by-reference"-Parameterübergabe mit Hilfe der Zeiger (Pointer) zu realisieren. Im allgemeinen ist die Verwendung von Referenzen vorzuziehen, bei Übergabe der Array-Parameter wird jedoch der Zeiger verwendet.

ParameterII.cpp

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

double sinussatz(double *lookup_sin, int angle, double opositeSide){
  return opositeSide*lookup_sin[angle];
}
/*
double sinussatz(double lookup_sin[], int angle, double opositeSide){
  return opositeSide*lookup_sin[angle];
}
*/

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

```
Größe des Arrays 2880
Result =  10
```

Statt Zeiger kann die Notation als Array undefinierter (definierter) Größe verwendet werden. Unabhängig von der Notation entspricht die Größe des übergebenen Arrays der Definition in der aufrufenden Funktion (hier main-Funktion).

Auch bei der Verwendung von Zeigern und Referenzen werden keine Kopien von Paramern angelegt, sondern die Parameter selbst direkt verändert. Falls keine Veränderung angestrebt wird, aber das Anlegen von Kopien vermieden werden soll, können konstante Zeiger bzw. Referenzen verwendet werden.

ParameterIV.cpp

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

double sinussatz(const double *lookup_sin, int angle, double oposite
  return oppositeSide*lookup_sin[angle];
}

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

```
Größe des Arrays 2880
Result =  10
```

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>
using namespace std;

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

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

```
main.cpp: In function ‘int& doCalc(int&)’:
main.cpp:6:10: warning: reference to local variable ‘a’ returned [-
Wreturn-local-addr]
    6 |   return a;
      |          ^
main.cpp:5:7: note: declared here
    5 |   int a = wert + 5;
      |       ^
```

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

Variante 1 Sie übergeben den Rückgabewert in der Parameterliste.

ReferenzAsParameter.cpp

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

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);
  cout<<"Die Kreisfläche beträgt für d="<<wert<<"[m] "<<flaeche<<"[m²
  return 0;
}
```

```
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>
using namespace std;

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);
  cout<<"Die Kreisfläche beträgt für d="<<wert<<"[m] "<<*flaeche<<"[m
    ;
  delete flaeche;
  return 0;
}
```

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

Besonderheit Arrays

conversion.c

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

void printSizeOf(int intArray[])
{
    cout<<"sizeof of parameter: "<<sizeof(intArray)<<"\n";
}

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

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

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

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

```
main.cpp: In function ‘void printSizeOf(int*)’:
main.cpp:7:43: warning: ‘sizeof’ on array function parameter ‘intArray’
will return size of ‘int*’ [-Wsizeof-array-argument]
    7 |     cout<<"sizeof of parameter: "<<sizeof(intArray)<<"\n";
      |                                          ~^~~~~~~~~
main.cpp:5:22: note: declared here
    5 | void printSizeOf(int intArray[])
      |                  ~~~~^~~~~~~~~~
main.cpp: In function ‘void printLength(int*)’:
main.cpp:12:43: warning: ‘sizeof’ on array function parameter
‘intArray’ will return size of ‘int*’ [-Wsizeof-array-argument]
   12 |     cout<<"Length of parameter: "<<sizeof(intArray) /
sizeof(intArray[0])<<"\n";
      |                                          ~^~~~~~~~~
main.cpp:10:22: note: declared here
   10 | void printLength(int intArray[])
      |                  ~~~~^~~~~~~~~~

sizeof of array: 28
sizeof of parameter: 8
Length of array: 7
Length of parameter: 2
```

`main`-Funktion

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

Definition der `main`-Funktion:

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

```c
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

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

int main(int argc, char *argv[]) {
  for (int i=0;i<argc;i++)
    cout<<argv[i]<<" ";
  return 0;
}
```

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

Beispiel des Tages

Wie werden Funktionen im realen Programmierbetrieb angewendet? Werfen Sie einen Blick auf die Dokumentation unseres Mikrocontrollers.

https://microsoft.github.io/azure-iot-developer-kit/docs/apis/display/

Dort finden Sie einzelne Funktionen, die der Display Klasse (wird in der nächsten Woche behandelt) zugordnet sind.+

+ + +

ArduinoDisplay.cpp

```
#include <OledDisplay.h>

                                        // Aus Platzgründen entfernt
unsigned char BMP[] = {0,0,0,0,0,0,0,0,0,0, ..... 0,0,0,0,0,0,0,0};

void setup(){
    Screen.init();
}
void loop(){
    // print a string to the screen with wrapped = false
    Screen.print("This is OLEDDisplay Testing", false);
    delay(1000);
    // print a string to the screen with wrapped = true
    Screen.print("This is a very small display including only 4 lines", tru
    delay(1000);
    for(int i =0; i<=3; i++)
    {
        char buf[100];
        sprintf(buf, "This is row %d", i);
        Screen.print(i, buf);
    }
    delay(1000);
    // draw a bitmap to the screen
    Screen.draw(0, 0, 128, 8, BMP);
    delay(1000);
    // clean screen ready for next loop
    Screen.clean();
}
```

Aufgabe: Wie könnten wir den Code abwandeln, um eine Laufschrift umzusetzen?

BuggyCode.cpp

```cpp
void setup() {
  pinMode(LED_BUILTIN, OUTPUT);
  Serial.begin(9600);
  String text = "Das ist ein Test";
  Serial.println(text);
  String output = "";
  int signs_per_line = 5;
  for (int i=0; i<text.length(); i++){
    if (i <= text.length()-signs_per_line)
      output = text.substring(i, i+signs_per_line);
    else
      output = text.substring(i, text.length()) +
               text.substring(0, signs_per_line-(text.length()-i));
    Serial.println(output);
  }
}

void loop() {
}
```

```
Sketch uses 3596 bytes (11%) of program storage space. Maximum is 32256
bytes.
Global variables use 214 bytes (10%) of dynamic memory, leaving 1834
bytes for local variables. Maximum is 2048 bytes.


Das ist ein Test
Das i
as is
s ist
 ist
ist e
st ei
t ein
 ein
ein T
in Te
n Tes
 Test
TestD
estDa
stDas
tDas
```

Quiz

Funktionen

Funktionsdefinition

Wechen Rückgabewert liefert eine als `void` deklatierte Funktion?

- ◯ Es können alle Arten von Rückgabewerten zurückgegeben werden.
- ◯ 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>
using namespace std;

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)
    cout <<  a << " ist ein Teiler von " << b << "." << endl;
  else
    cout <<  a << " ist kein Teiler von " << b << "." << endl;
}
```

Fehler

ist dieses Programm fehlerfrei?

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

int foo(){
   return 42;
}

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

- ◯ Ja
- ◯ Nein

Ist dieses Programm fehlerfrei?

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

void foo(){
    return 42;
}

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

- ◯ Ja
- ◯ Nein

Welche Fehler liegen bei diesem Programm vor?

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

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

[_____]

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

void hw(void) {
  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>
using namespace std;

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);
  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>
using namespace std;

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

int main(void) {
  int b = 0;
  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>
using namespace std;

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);
  cout << "Das Kugelvolumen beträgt für d=" << wert << "[m] " << *volumen <
    "[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>
using namespace std;

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