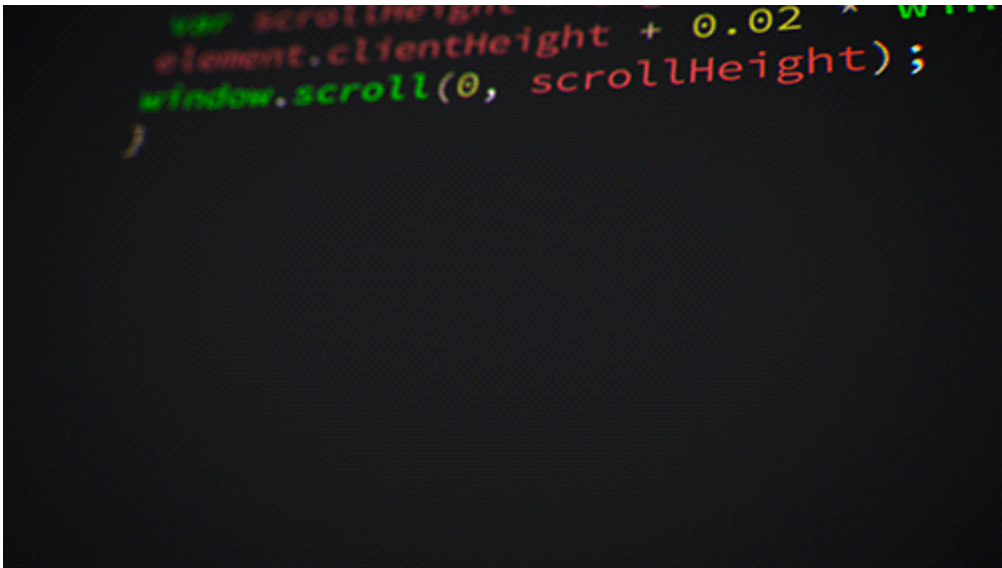


# Einführung

Parameter	Kursinformationen
Veranstaltung:	Prozedurale Programmierung / Einführung in die Informatik / Erhebung, Analyse und Visualisierung digitaler Daten
Semester	Wintersemester 2025/26
Hochschule:	Technische Universität Freiberg
Inhalte:	Vorstellung des Arbeitsprozesses
Link auf Repository:	<a href="https://github.com/TUBAF-lfl-LiaScript/VL_EAVD/blob/master/00_Einfuehrung.md">https://github.com/TUBAF-lfl-LiaScript/VL_EAVD/blob/master/00_Einfuehrung.md</a>
Autoren	Sebastian Zug & André Dietrich & Galina Rudolf & Copilot



Fragen an die heutige Veranstaltung ...

- Welche Aufgabe erfüllt eine Programmiersprache?
- Erklären Sie die Begriffe Compiler, Editor, Programm, Hochsprache!
- Was passiert beim Kompilieren eines Programmes?
- Warum sind Kommentare von zentraler Bedeutung?
- Worin unterscheiden sich ein konventionelles C++ Programm und eine Anwendung, die mit dem Arduino-Framework geschrieben wurde?

## Reflexion Ihrer Fragen / Rückmeldungen

Wettstreit zur partizipativen Materialentwicklung mit den Informatikern ...



Preis für das aktivste Auditorium

Format	Informatik Studierende
Verbesserungsvorschlag	0
Fragen	1
generelle Hinweise	0





Hat Sie die letztwöchige Vorstellung der Ziele der Lehrveranstaltung überzeugt?

- ☐ Ja, ich gehe davon aus, viel nützliches zu erfahren.
- ☐ Ich bin noch nicht sicher. Fragen Sie in einigen Wochen noch mal.
- ☐ Nein, ich bin nur hier, weil ich muss.

## Warum sollten Sie programmieren lernen?

Sie fragen sich vielleicht: "Ich studiere doch gar nicht Informatik - wozu brauche ich das?" Eine berechtigte Frage! Lassen Sie mich Ihnen zeigen, warum Programmierkenntnisse für IHRE Arbeit wertvoll sind.

Stellen Sie sich folgende Situationen vor:

-  Sie haben 500 Excel-Dateien mit Messdaten, die Sie alle auswerten müssen
-  Ihr Experiment muss 1000 mal wiederholt werden - immer mit leicht veränderten Parametern
-  Sie möchten Ihre Versuchsergebnisse professionell visualisieren
-  Das Experiment soll kontinuierlich Daten erfassen - auch nachts und am Wochenende

**Frage:** Wie würden Sie diese Aufgaben ohne Programmierung lösen? Wie lange würde das dauern?

**Was Programmierung für Sie bedeutet:**





Ohne Programmierung	Mit Programmierung
2 Wochen manuelles Kopieren	5 Minuten Skript schreiben, dann automatisch
Wiederholung = Langeweile + Fehler	Computer macht die Wiederholung fehlerfrei
Begrenzt auf Standard-Tools	Individuelle Lösungen für Ihre Probleme
Abhängig von anderen	Selbstständig arbeiten können

**Programmieren ist wie ein Schweizer Taschenmesser für wissenschaftliches Arbeiten!**

**Programmieren macht einmal arbeit, spart dann aber viel Zeit und Mühe!**

Viele Aufgaben in Wissenschaft und Technik lassen sich mit ein wenig Programmierkenntnis enorm vereinfachen. Lassen Sie sich nicht abschrecken – Sie müssen kein Profi werden, um große Effekte zu erzielen!

**Das Beste daran:**

-  Sie werden **unabhängiger** in Ihrer wissenschaftlichen Arbeit
-  Sie **sparen Zeit** bei repetitiven Aufgaben
-  Sie können **präziser** arbeiten (Computer machen keine Tippfehler)
-  Sie erschließen sich **neue Möglichkeiten** für Ihre Forschung

*"Ich kann jetzt in 10 Minuten auswerten, wofür ich früher einen Tag gebraucht habe!" - Studentin der Geowissenschaften nach diesem Kurs*

## Und wie "erkläre" ich dem Computer, was ich will?

Warum können wir Computer nicht einfach in unserer Alltagssprache programmieren? Warum brauchen wir überhaupt Programmiersprachen wie C++ oder Python? Die Antwort liegt in der fundamentalen Funktionsweise von Computern ... und der unterschiedlichen Natur von menschlicher und maschineller Kommunikation.

### Warum versteht der Computer keine natürliche Sprache?

*"Computer, erhöhe bitte die Temperatur etwas!"*

Was bedeutet "etwas"? 1 Grad? 5 Grad? 10 Grad? Und wie schnell? Sofort oder graduell?

#### Das Problem mit natürlicher Sprache:

- 🤔 **Mehrdeutigkeit:** "Erhöhe etwas" kann vieles bedeuten
- 💭 **Kontext:** "Es ist kalt" → Mensch versteht: "Heizung aufdrehen" | Computer: "???"
- 🗣️ **Interpretation:** "Das ist ja toll!" kann Lob oder Sarkasmus sein
- 🌐 **Implizites Wissen:** Wir setzen voraus, dass andere verstehen, was wir meinen.



#### Computer hingegen:

- ⚙️ Verarbeiten **nur exakte Anweisungen** (Nullen und Einsen)
- 🤖 Haben **kein Weltwissen** oder Erfahrung
- 📏 Brauchen **präzise Definitionen** für jeden Schritt
- ❌ Können **nicht raten oder interpretieren**

**Computer sind wie ein extrem pedantischer Bürokrat, der JEDE Anweisung wörtlich nimmt und nichts zwischen den Zeilen liest!**

Programme sind daher Anweisungslisten, die vom Menschen erdacht, auf einem Rechner zur Ausführung kommen. Eine zentrale Hürde ist dabei die Kluft zwischen menschlicher Vorstellungskraft, die **implizite Annahmen und Erfahrungen** einschließt, und der **"stupiden" Abarbeitung von Befehlsfolgen** in einem Rechner.

Programmiersprachen sind ein Kompromiss zwischen:

-  **Menschlicher Lesbarkeit** → Wir können sie (einigermaßen) verstehen
-  **Maschinellem Eindeutigkeit** → Computer können sie exakt ausführen

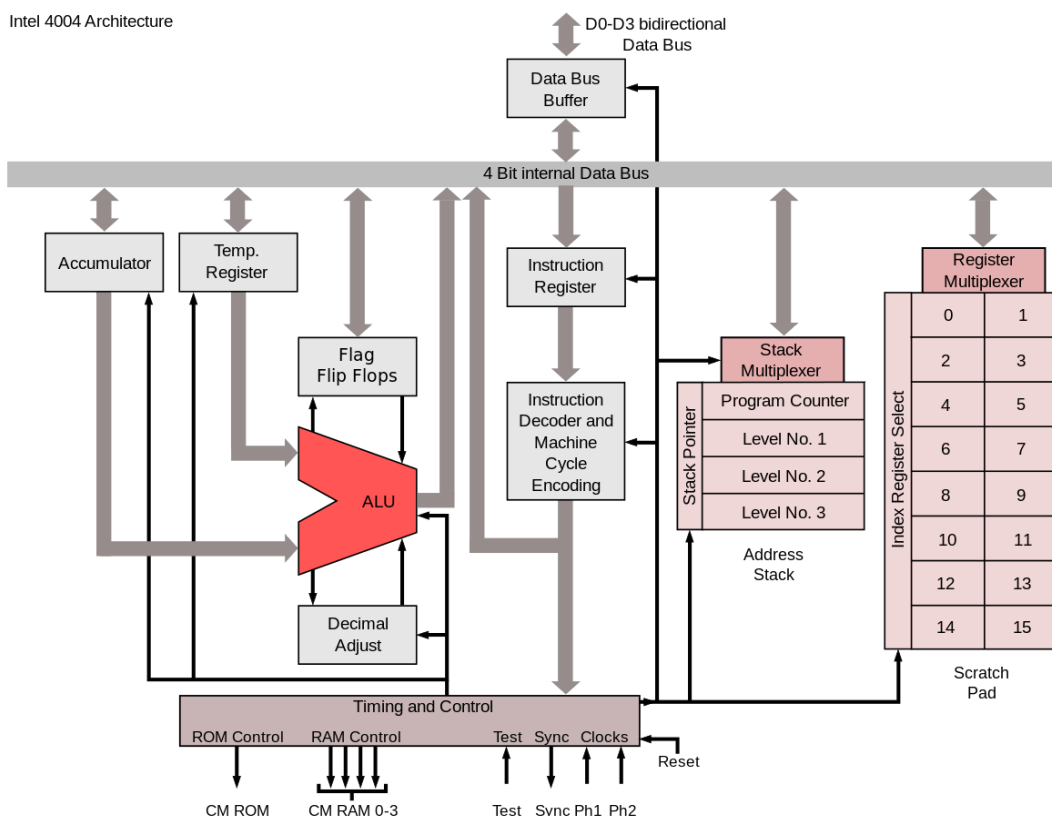
Beispiel:

Natürliche Sprache	Programmiersprache	Maschinensprache
"Erhöhe die Temperatur etwas"	<code>temperature = temperature + 2;</code>	<code>01011010 11001001 ...</code>
Mehrdeutig	Präzise	Für Menschen unlesbar
Flexibel	Strukturiert	Direkt ausführbar

## Wie arbeitet ein Rechner eigentlich?

Die Intel 4004 war der erste kommerziell erhältliche Mikroprozessor (1971). Er konnte einfache Rechenoperationen ausführen, hatte ein eigenes Steuerwerk, ein Rechenwerk (ALU), Register und einen Speicherzugriff – alles auf einem einzigen Chip. Das nachfolgende Bild zeigt die grundlegenden Komponenten und deren Zusammenspiel: Steuerwerk, Rechenwerk, Register und Busse. Er konnte übrigens nur 4-Bit Zahlen verarbeiten!

Beispiel: Intel 4004-Architektur (1971)



Intel 4004 Architekturdarstellung Autor Appaloosa, Intel 4004,

[https://upload.wikimedia.org/wikipedia/commons/thumb/8/87/4004\\_arch.svg/1190px-4004\\_arch.svg.png](https://upload.wikimedia.org/wikipedia/commons/thumb/8/87/4004_arch.svg/1190px-4004_arch.svg.png)

Jeder Rechner hat einen spezifischen Satz von Befehlen, die durch "0" und "1" ausgedrückt werden, die er überhaupt abarbeiten kann. Man spricht vom sogenannten *Instruction Set*.

Speicherauszug den Intel 4004:

Adresse	Speicherinhalt	OpCode	Mnemonik	Bedeutung
0010	1101 0101	1101 DDDD	LD \$5	Lade den Wert 5 in die Recheneinheit
0012	1111 0010	1111 0010	IAC	erhöhe den Wert um 1

Maschinencode lässt sich nicht zwischen verschiedenen Rechnerarchitekturen übertragen. Jede CPU-Familie hat ihr eigenes Instruction Set, das festlegt, welche Befehle sie versteht und wie diese kodiert sind. Ein Programm, das für eine Architektur geschrieben wurde, funktioniert daher nicht automatisch auf einer anderen.

Möchte man so Programme schreiben?

**Vorteil:**

- ggf. sehr effizienter Code (Größe, Ausführungsdauer), der gut auf die Hardware abgestimmt ist

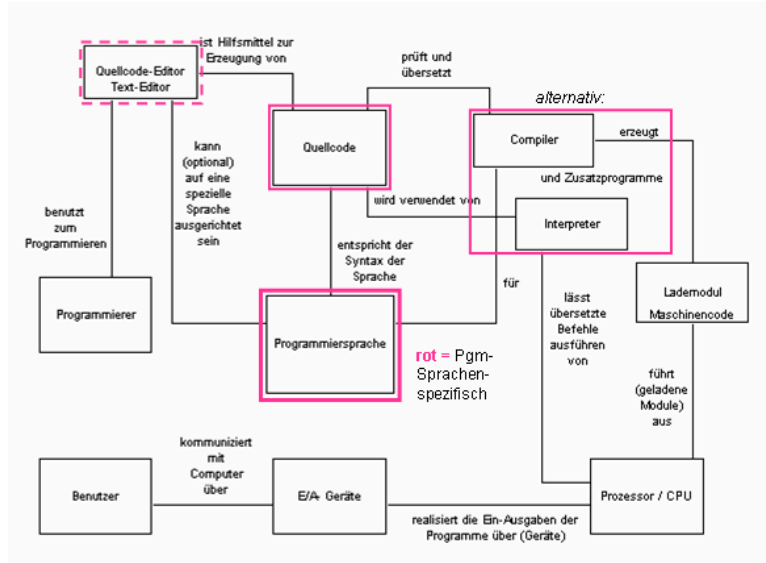
**Nachteile:**

- systemspezifische Realisierung
- geringer Abstraktionsgrad, bereits einfache Konstrukte benötigen viele Codezeilen
- weitgehende semantische Analysen möglich

## Begrifflichkeiten

Um das Verständnis für diesen Vorgang zu entwickeln werden zunächst die Vorgänge in einem Rechner bei der Abarbeitung von Programmen beleuchtet, um dann die Realisierung eines Programmes mit C++ zu adressieren.

## Programmiersprache: Vom Quellcode zur Ausführung im Prozessor



Erzeugung von Programmcode Programmiervorgang und Begriffe (Autor VÖRBY,  
[https://commons.wikimedia.org/wiki/File:Programmiersprache\\_Umfeld.png](https://commons.wikimedia.org/wiki/File:Programmiersprache_Umfeld.png))

Begriff	Erklärung
Compiler	Übersetzt den Quellcode (Sourcecode) einer Programmiersprache in Maschinensprache oder eine andere Zielsprache.
Sourcecode	Der vom Menschen geschriebene Programmtext in einer Programmiersprache.
Editor	Programm zum Schreiben und Bearbeiten von Sourcecode.
Programmiersprache	Formale Sprache mit festgelegter Syntax und Semantik, um Computerprogramme zu formulieren.
Interpreter	Führt den Sourcecode Zeile für Zeile direkt aus, ohne ihn vorher zu übersetzen.

## Einordnung von C und C++

**imperative (befehlsorientierte) Programmiersprachen:** Ein Programm besteht aus einer Folge von Befehlen an den Computer. Das Programm beschreibt den Lösungsweg für ein Problem (C, Python, Java, LabView, Matlab, ...).

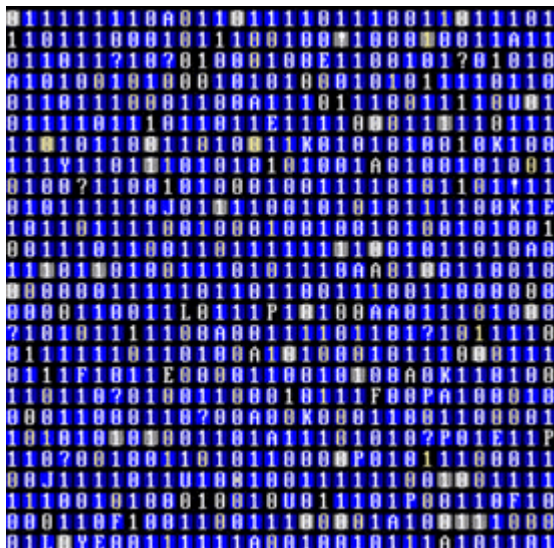
**deklarative Programmiersprachen:** Ein Programm beschreibt die allgemeinen Eigenschaften von Objekten und ihre Beziehungen untereinander. Das Programm beschreibt zunächst nur das Wissen zur Lösung des Problems (Prolog, Haskell, SQL, ...).

#### Fakten zu C++:

- Entwickelt ab 1979 von Bjarne Stroustrup (AT&T Bell Labs)
- Erweiterung von C um objektorientierte Programmierung
- Sehr weit verbreitet in Technik, Wissenschaft, Spieleentwicklung und Embedded Systems
- Kompiliert zu sehr effizientem Maschinencode (hohe Performance)
- Unterstützt prozedurale, objektorientierte und generische Programmierung
- Wird von vielen modernen Compilern (z.B. GCC, Clang, MSVC) unterstützt
- Standardisiert (ISO C++), aktuell C++23/26

## Erstes C++ Programm

Keine Angst vor dem ersten Programm! Sie müssen nicht alles sofort verstehen – wichtig ist, dass Sie sehen, wie ein vollständiges Programm aussieht und wie einfach der Einstieg sein kann.



**"Hello World"**



## HelloWorld.c



```
1 // That's my first C program
2 // Karl Klammer, Oct. 2022
3
4 #include <iostream>
5
6 int main() {
7     std::cout << "Hello World!\n";
8     return 0;
9 }
```

```
Hello World!
Hello World!
```

Zeile	Bedeutung
1 - 2	Kommentar (wird vom Präprozessor entfernt)
4	Voraussetzung für das Einbinden von Befehlen der Standardbibliothek hier <code>std::cout</code> (wird für die Ausgabe gebraucht)
6	Einsprungstelle für den Beginn des Programmes
6 - 9	Ausführungsblock der <code>main</code> -Funktion
7	Anwendung eines Operators <code>&lt;&lt;</code> hier zur Ausgabe auf dem Bildschirm
8	Definition eines Rückgabewertes für das Betriebssystem

**Halt!** Unsere C++ Arduino Programme sahen doch ganz anders aus?





```
1 void setup() {  
2   pinMode(LED_BUILTIN, OUTPUT);  
3 }  
4  
5 void loop() {  
6   digitalWrite(LED_BUILTIN, HIGH);  
7   delay(1000);  
8   digitalWrite(LED_BUILTIN, LOW);  
9   delay(1000);  
10 }
```

Sketch uses 924 bytes (2%) of program storage space. Maximum is 32256 bytes.

Global variables use 9 bytes (0%) of dynamic memory, leaving 2039 bytes for local variables. Maximum is 2048 bytes.

Sketch uses 924 bytes (2%) of program storage space. Maximum is 32256 bytes.

Global variables use 9 bytes (0%) of dynamic memory, leaving 2039 bytes for local variables. Maximum is 2048 bytes.

Die Arduino-Programme haben eine etwas andere Struktur als konventionelle C++ Programme. Das liegt daran, dass das Arduino-Framework bestimmte Funktionen und Abläufe vorgibt, um die Programmierung von Mikrocontrollern zu erleichtern. Es gibt zwei Hauptfunktionen, die in jedem Arduino-Sketch definiert werden müssen: `setup()` und `loop()` - was die grundsätzlichen Nutzungsmuster widerspiegelt.

**Noch mal Halt!** Das klappt ja offenbar alles im Browserfenster, aber wenn ich ein Programm auf meinem Rechner kompilieren möchte, was ist dann zu tun?

## Ein Wort zu den Formalien

## HelloWorld.cpp



```
1 // Karl Klammer
2 // Print Hello World drei mal
3
4 #include <iostream>
5
6 int main() {
7     int zahl;
8     for (zahl=0; zahl<3; zahl++){
9         std::cout << "Hello World!\n";
10    }
11    return 0;
12 }
```

```
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
```

## BadHelloWorld.cpp



```
1 #include <iostream>
2 int main() {int zahl; for (zahl=0; zahl<3; zahl++){ std::cout << "Hel
    World!\n";} return 0;}
```

```
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
```

- Das *systematische Einrücken* verbessert die Lesbarkeit und senkt damit die Fehleranfälligkeit Ihres Codes!
- Wählen sie *selbsterklärende Variablen- und Funktionsnamen*!
- Nutzen Sie ein *Versionsmanagementsystem*, wenn Sie ihren Code entwickeln!
- Kommentieren Sie Ihr Vorgehen trotz "Good code is self-documenting"

# Gute Kommentare

Kommentare sind für Menschen gedacht, die den Code lesen und verstehen müssen. Sie helfen dabei, den Zweck und die Funktionsweise von Codeabschnitten zu erklären, insbesondere wenn dieser komplex oder nicht sofort verständlich ist. Das bezieht auch Sie selbst mit ein 😊

## 1. Kommentare als Pseudocode

```
/* loop backwards through all elements returned by the server
(they should be processed chronologically)*/
for (i = (numElementsReturned - 1); i >= 0; i--){
    /* process each element's data */
    updatePattern(i, returnedElements[i]);
}
```

## 2. Kommentare zur Datei

```
// This is the mars rover control application
//
// Karl Klammer, Oct. 2018
// Version 109.1.12

int main() {...}
```

## 3. Beschreibung eines Algorithmus

```
/**
 * @brief Computes an approximation of pi.
 *
 * The approximation is calculated using
 *  $\pi/6 = 1/2 + (1/2 \times 3/4) 1/5 (1/2)^3 + (1/2 \times 3/4 \times 5/6) 1/7 (1/2)^5 +$ 
 *
 * @param n Number of terms in the sum series.
 *
 * @return the approximate value of pi OR zero on error
 */
double approx_pi(int n);
```

In realen Projekten werden Sie für diese Aufgaben Dokumentationstools verwenden, die die Generierung von Webseite, Handbüchern auf der Basis eigener Schlüsselworte in den Kommentaren unterstützen → [doxygen](#).

## 4. Debugging

```
int main() {
    ...
    preProcessedData = filter1(rawData);
    // printf('Filter1 finished ... \n');
    // printf('Output %d \n', preProcessedData);
}
```

```
// printf("Output %d \n", preprocessedData);  
result=complexCalculation(preProcessedData);  
...  
}
```

## Schlechte Kommentare

Schlechte Kommentare können mehr Schaden als nützen. Sie können den Code unübersichtlich machen, falsche Informationen vermitteln oder einfach nur Zeit verschwenden.

### 1. Überkommentierung von Code

```
x = x + 1; /* increment the value of x */  
std::cout << "Hello World!\n"; // displays Hello world
```



*... over-commenting your code can be as bad as under-commenting it!*

Quelle: [C Code Style Guidelines](#)

### 2. "Merkwürdige Kommentare"

```
// When I wrote this, only God and I understood what I was doing  
// Now, God only knows
```



```
// sometimes I believe compiler ignores all my comments
```



```
// Magic. Do not touch.
```



```
// I am not responsible of this code.
```



```
try {  
  
} catch(e) {  
  
} finally {  
    // should never happen  
}
```

[Sammlung von Kommentaren](#)

## Was tun, wenn es schief geht?

Auch erfahrene Programmierer machen Fehler. Das ist normal und gehört zum Lernprozess dazu. Wichtig ist, dass Sie lernen, wie Sie mit Fehlern umgehen und sie beheben können. Gehen Sie systematisch vor und lassen Sie sich nicht entmutigen! Sofern der Compiler Fehlermeldungen ausgibt, sind diese Ihr bester Freund.

### ErroneousHelloWorld.cpp



```
1 // Karl Klammer
2 // Print Hello World drei mal
3
4 #include <iostream>
5
6 int main() {
7     for (zahl=0; zahl<3; zahl++){
8         std::cout << "Hello World! "
9     }
10    return 0;
11
```

```

main.cpp: In function 'int main()':
main.cpp:7:8: error: 'zahl' was not declared in this scope
   7 |   for (zahl=0; zahl<3;  zahl++){
     |       ^~~~
main.cpp:8:41: error: expected ';' before '}' token
   8 |         std::cout << "Hello World! "
     |                                         ^
     |                                         ;
   9 |   }
     |   ~
main.cpp:10:18: error: expected '}' at end of input
  10 |         return 0;
     |                ^
main.cpp:6:12: note: to match this '{'
   6 | int main() {
     |          ^
main.cpp: In function 'int main()':
main.cpp:7:8: error: 'zahl' was not declared in this scope
   7 |   for (zahl=0; zahl<3;  zahl++){
     |       ^~~~
main.cpp:8:41: error: expected ';' before '}' token
   8 |         std::cout << "Hello World! "
     |                                         ^
     |                                         ;
   9 |   }
     |   ~
main.cpp:10:18: error: expected '}' at end of input
  10 |         return 0;
     |                ^
main.cpp:6:12: note: to match this '{'
   6 | int main() {
     |          ^

```

Methodisches Vorgehen:

- **RUHE BEWAHREN**
- Lesen der Fehlermeldung
- Verstehen der Fehlermeldung / Aufstellen von Hypothesen
- Systematische Evaluation der Thesen
- Seien Sie im Austausch mit anderen (Kommilitonen, Forenbesucher, KI usw.) konkret.

# Compilerfehlermeldungen

## Beispiel 1

### Error.cpp

```
1  #include <iostream>
2
3  int mani() {
4      std::cout << "Hello World!";
5      return 0;
6  }
```

```
/usr/bin/ld: /usr/lib/gcc/x86_64-linux-gnu/11/../../../../x86_64-linux-
gnu/Scrt1.o: in function `_start':
(.text+0x1b): undefined reference to `main'
collect2: error: ld returned 1 exit status
```

```
/usr/bin/ld: /usr/lib/gcc/x86_64-linux-gnu/11/../../../../x86_64-linux-
gnu/Scrt1.o: in function `_start':
(.text+0x1b): undefined reference to `main'
collect2: error: ld returned 1 exit status
```

```
timeout: failed to run command './a.out': No such file or directory
timeout: failed to run command './a.out': No such file or directory
```

## Beispiel 2

### Error.cpp

```
1  #include <iostream>
2
3  int main()
4      std::cout << "Hello World!";
5      return 0;
6  }
```



```

main.cpp:4:3: error: expected initializer before 'std'
    4 |     std::cout << "Hello World!";
      |     ^~~
main.cpp:5:3: error: expected unqualified-id before 'return'
    5 |     return 0;
      |     ^~~~~~
main.cpp:6:1: error: expected declaration before '}' token
    6 | }
      | ^
main.cpp:4:3: error: expected initializer before 'std'
    4 |     std::cout << "Hello World!";
      |     ^~~
main.cpp:5:3: error: expected unqualified-id before 'return'
    5 |     return 0;
      |     ^~~~~~
main.cpp:6:1: error: expected declaration before '}' token
    6 | }
      | ^

```

Manchmal muss man sehr genau hinschauen, um zu verstehen, warum ein Programm nicht funktioniert. Versuchen Sie es!

### ErroneousHelloWorld.cpp

```

1  #include <iostream>
2
3  int main() {
4      std::cout << "Hello World!";
5      std::cout << "Wo liegt der Fehler?";
6      return 0;
7  }

```

```

main.cpp:4:22: warning: missing terminating " character
  4 |         std::cout << "Hello World!';
    |                               ^
main.cpp:4:22: error: missing terminating " character
  4 |         std::cout << "Hello World!';
    |                               ^~~~~~
main.cpp:3:1: error: 'imt' does not name a type; did you mean 'int'?
  3 | imt main() {
    | ^~~
    | int
main.cpp:4:22: warning: missing terminating " character
  4 |         std::cout << "Hello World!';
    |                               ^
main.cpp:4:22: error: missing terminating " character
  4 |         std::cout << "Hello World!';
    |                               ^~~~~~
main.cpp:3:1: error: 'imt' does not name a type; did you mean 'int'?
  3 | imt main() {
    | ^~~
    | int

```

## Und wenn das Kompilieren gut geht?

Logische Fehler kann der Compiler nicht erkennen. Wenn Ihr Programm also ohne Fehlermeldungen kompiliert, bedeutet es noch nicht, dass es auch korrekt ist. Vielmehr müssen wir nun durch Testen und Debuggen sicherstellen, dass das Programm das tut, was wir erwarten.

... dann bedeutet es noch immer nicht, dass Ihr Programm wie erwartet funktioniert.



```

1  #include <iostream>
2
3  int main() {
4      int a[5] = {1, 2, 3, 4, 5};
5      int b[5] = {2, 2, 2, 2, 2};
6      int result = 0;
7
8      // Achtung: Der letzte Index (4) wird nicht adressiert!
9      for (int i = 0; i < 4; i++) {
10         result += a[i] * b[i];
11     }
12
13     std::cout << "Skalarprodukt: " << result << std::endl;
14     return 0;
15 }

```

Skalarprodukt: 20

Skalarprodukt: 20

$$\vec{a} = \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{pmatrix}, \quad \vec{b} = \begin{pmatrix} 2 \\ 2 \\ 2 \\ 2 \\ 2 \end{pmatrix}$$

// Skalarprodukt korrekt berechnet:

$$\vec{a} \cdot \vec{b} = 1 \cdot 2 + 2 \cdot 2 + 3 \cdot 2 + 4 \cdot 2 + 5 \cdot 2 = 30$$

// Fehlerhafte Berechnung (letztes Element fehlt):

$$\vec{a} \cdot \vec{b} \neq 1 \cdot 2 + 2 \cdot 2 + 3 \cdot 2 + 4 \cdot 2 = 20$$

// Unsere Schleife hört zu früh auf!

**Hinweis:** Die Arrays werden wir in einer der kommenden Veranstaltungen im Detail besprechen.

## Warum dann C++?

C++ ist eine weit verbreitete Programmiersprache, die in vielen Bereichen der Softwareentwicklung eingesetzt wird. Sie bietet eine gute Balance zwischen Leistung, Flexibilität und Kontrolle über Systemressourcen. C++ ermöglicht sowohl prozedurale als auch objektorientierte Programmierung, was es

Entwicklern erlaubt, komplexe Anwendungen zu erstellen. Python im Unterschied ist eine interpretierte Sprache, die für ihre Einfachheit und Lesbarkeit bekannt ist. Python eignet sich hervorragend für schnelle Prototypenentwicklung, Datenanalyse und wissenschaftliches Rechnen. Es hat eine umfangreiche Standardbibliothek und viele Drittanbieter-Pakete, die die Entwicklung beschleunigen können. Merke, es gibt nicht die "beste" Programmiersprache – die Wahl hängt immer von den spezifischen Anforderungen und Zielen eines Projekts ab.

Zwei Varianten der Umsetzung ... C++ vs. Python

### HelloWorld.cpp

```
1 #include <iostream>
2
3 int main() {
4     char zahl;
5     for (int zahl=0; zahl<3; zahl++){
6         std::cout << "Hello World! " << zahl << "\n";
7     }
8     return 0;
9 }
```

```
Hello World! 0
Hello World! 1
Hello World! 2
Hello World! 0
Hello World! 1
Hello World! 2
```

```
1 for i in range(3):
2     print("Hallo World!", i)
```

```
Hallo World! 0
Hallo World! 1
Hallo World! 2
Hallo World! 0
Hallo World! 1
Hallo World! 2
```

## Beispiele der Woche

Gültiger C++ Code

**Umfrage:** Welche Ausgabe erwarten Sie für folgendes Code-Schnippelchen?

- ☐ Hello World7
- ☐ Hello World11
- ☐ Hello World 11!
- ☐ Hello World 11 !
- ☐ Hello World 5 !

#### Output.cpp



```
1  #include <iostream>
2
3  int main() {
4      int i = 5;
5      int j = 4;
6      i = i + j + 2;
7      std::cout << "Hello World ";
8      std::cout << i << "!\n";
9      return 0;
10 }
```

```
Hello World 11!
Hello World 11!
```

#### Algorithmisches Denken

**Aufgabe:** Ändern Sie den Code so ab, dass das die LED zwei mal mit 1 Hz blinkt und dann ausgeschaltet bleibt.



## BuggyCode.cpp



```
1 void setup() {  
2   pinMode(LED_BUILTIN, OUTPUT);  
3 }  
4  
5 void loop() {  
6   digitalWrite(LED_BUILTIN, HIGH);  
7   delay(1000);  
8   digitalWrite(LED_BUILTIN, LOW);  
9   delay(1000);  
10 }
```

Sketch uses 924 bytes (2%) of program storage space. Maximum is 32256 bytes.

Global variables use 9 bytes (0%) of dynamic memory, leaving 2039 bytes for local variables. Maximum is 2048 bytes.

Sketch uses 924 bytes (2%) of program storage space. Maximum is 32256 bytes.

Global variables use 9 bytes (0%) of dynamic memory, leaving 2039 bytes for local variables. Maximum is 2048 bytes.

## Quiz

Testen Sie Ihr Wissen! Die folgenden Fragen helfen Ihnen, die wichtigsten Konzepte noch einmal zu reflektieren.

## Einbinden von Bibliotheken

Wie müssen Bibliotheken in C++ eingebunden werden?

- ☐ using iostream
- ☐ import iostream
- ☐ #include <iostream>

## Kommentare

Sind diese Kommentare angebracht?

```
#include <iostream> // I always forget this xD

int main() { // main-function
    char zahl; // I have homework due tomorrow :(
    for (zahl=250; zahl<256; zahl++){ // I really have no idea what this ever
        but whatever... LOL
        std::cout << "Hello World!"; // Prints "Hello World!" in console
    }
    return 0;
}
```

☐ Ja

☐ Nein

## Konventionen und Formalien

Wählen Sie alle Programme aus, die den üblichen Formalien entsprechen.

1.cpp

```
#include <iostream>
int main() {int a = 0; std::cout << "Hello World!" << a; return 0;}
```

2.cpp

```
#include <iostream>

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

### 3.cpp



```
#include <iostream>

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

☐ 1.cpp

☐ 2.cpp

☐ 3.cpp