



**WBS  
TRAINING**

## Programmierung(1)

# Agenda

- **Datenspeicherung** auf dem Rechner: Eine vereinfachte Darstellung
- Maschinensprache versus „Höhere Sprache“: **Binärcode** versus **Quellcode**
- Programmier-Paradigma: **Imperative**, **Deklarative** oder **Objekt-orientierte** Sprache
- Entwicklungsumgebung **Code::Blocks** (**installieren!?**): **Editor**, **Compiler**, **Debugger**, **Linker**, **Ausgabe-Konsole**
- **Der erste eigene Code**
  - Aufbau des Quellcodes
    - Hauptprogramm: **main()**
      - Anweisungsblock
      - Anweisung
    - Präprozessoranweisung: **include<...>**
  - Variable: **Deklarieren**, **Definieren**, **Initialisieren**
  - Zuweisung: **Literale** und/oder **Ergebnisse von Rechenoperationen**
  - Ausgabe: **printf()**
    - **Formatierung:**
      - String-Literale
      - Typen
      - Layout

# Datenspeicherung auf dem Rechner

- Die kleinste Speichereinheit auf einem Computer wird als **Bit** bezeichnet. Es handelt sich dabei technisch betrachtet um eine Art Schalter, der nur die beiden Zustände: ...
  - „geschlossen“(Strom fließt) oder
  - „offen“(Strom fließt nicht)... annehmen kann.

Diese beiden Zustände werden der Kürze wegen gerne mit den Ziffern **1** (für geschlossen) und **0** (für offen) dargestellt.
- Fasst man 8 Bit zusammen, so wird diese Speicherstelle als **Byte** bezeichnet. Da jedes Bit 2 Zustände annehmen kann, können durch ein Byte demnach „2 hoch 8“ = 256 Zustände unterschieden werden. Binäre Darstellung: 00000000, 00000001, 00000010, 00000011, ... , 11111100, 11111101, 11111110, 11111111
- Solche **Bitfolgen** können ganz unterschiedlich interpretiert werden. Sie können je nach Bedarf (bzw. nach gewünschter Festlegung) als Zeichen, Zahl oder auch Befehl verstanden werden. Letzteres werden wir uns nun anschauen, wenn es im Folgenden um Befehle (einer Maschinensprache) geht, die von einem Prozessor ausgeführt werden sollen:

# Maschinensprache versus „Höhere Sprache“

- Eine **Maschinensprache** (bzgl. eines bestimmten Prozessortyps) bezeichnet den gesamten Umfang all jener Befehle, die der entsprechende Prozessor unmittelbar ausführen kann.
- Maschinensprachen-Befehle werden in der Regel als **Bitfolge** dargestellt, deren Länge sich nach dem Prozessortyp richtet (für gewöhnlich handelt es sich bei der Länge um ein Vielfaches von 8).
- Maschinensprache-Befehle führen allerdings üblicherweise nur sehr elementare Prozesse aus. Entsprechend arbeitet ein in Maschinensprache geschriebenes Programm vergleichsweise **klein-schrittig**. Daher ist die Erstellung eines solchen Programms oft recht aufwendig.
- „**Höhere**“ **Sprachen** umgehen dieses Problem, indem sie den Programmierer zunächst einen sogenannten **Quellcode** erzeugen lassen, der zum einen wesentlich größere Arbeitsschritte erlaubt, und zum anderen deutlich leichter lesbar ist. Da dieser Quellcode aber für einen Prozessor nicht ausführbar ist, wird er **vor** der Programm-Ausführung (mittels **Compiler**) oder **während** der Ausführung (mittels **Interpreter**) in Maschinensprache übersetzt.

## Hinweis:

- Die hohe Arbeitsgeschwindigkeit heutiger Computer macht das Codieren in Maschinensprache oft überflüssig. Immer dort aber, wo extreme Performance und/oder geringer Speicherplatzbedarf von großer Bedeutung ist, macht es Sinn, Programme (oder Teile von Programmen) in Maschinensprache zu schreiben.

# Programmier-Paradigma (bzw. Philosophie, Konzept oder „Art und Weise“)

- Die älteste (und vielleicht auch noch verbreitetste) Art des Programmierens ist die sogenannte **imperative** (alternative Bezeichnung: **prozedurale**) Programmierung. Sie arbeitet – wie der Name schon sagt – mit einer Abfolge von **Befehlen** und erinnert an ein Kochrezept, bei dem man der Reihe nach (bzw. „von oben nach unten“) alle Handlungsschritte auflistet. Dies wird jenes Verfahren sein, mit dem wir uns (an Hand der Programmiersprache **ANSI C**) in diesem Baustein befassen werden.
- Eine andere Philosophie des Programmierens verfolgen die sogenannten **deklarativen** Sprachen. Diese verlangen nicht, dass man Schritt für Schritt beschreibt, „wie“ ein Programm zum gewünschten Ziel gelangt. Statt dessen genügt es, dieses Ziel zu beschreiben, da der Compiler/Interpreter anschließend eigenständig entscheidet, welche Arbeitsschritte zum Erreichen des Ziels nötig sein werden. (Beispiel für eine deklarative Sprache in dieser Umschulung ist die Datenbanksprache **SQL**)
- Ein deutlich jüngeres (aber zunehmend an Bedeutung gewinnendes) Programmier-Konzept ist das der OOP (**Objekt-Orientierte-Programmierung**). Es arbeitet mit einem „Ordnungs-Prinzip“, dessen Grundlagen (siehe: „Strukturen“) wir auch bei ANSI C kennenlernen werden. (Beispiel für eine OOP-Sprache ist **C#**)
- Neben vieler weiterer Paradigmen, seien exemplarisch nur noch die „**funktionale** Programmierung“(basiert auf Rekursion) und „**logische** Programmierung“(insbesondere für mathematische Beweise) genannt.

# Entwicklungsumgebung (IDE = Integrated Development Environment)

- Wer einen Quellcode schreibt, der könnte dies in einem ganz gewöhnlichen **Editor** tun. Anschließend müsste er mit einem **Compiler** (gegebenenfalls unter Verwendung eines **Präprozessor**) diesen Code übersetzen lassen, mit einem **Linker** die einzelne Module seines Programms zusammenführen, eventuell mittels **Debugger** nach Fehlern suchen und die aus all diesen Vorarbeiten resultierende Datei schließlich noch *abspeichern*, um sie anschließend *starten* bzw. *ausführen* zu können.
- Da dies lästig ist, wurden Entwicklungsumgebungen entwickelt, mit denen all diese Funktionalitäten in einem einzigen Programm vereinigt wurden. In der Regel reicht dann ein einziger „Klick“ um die obigen Arbeitsschritte der Reihe nach abarbeiten zu lassen.
- Wir wollen in diesem Baustein mit der IDE **Code::Blocks** arbeiten, die Sie bitte **herunterladen**, sofern sich diese nicht bereits auf ihrem Rechner befindet.
  - **Hinweis:** Nach der Installation wird möglicherweise der Compiler nicht gefunden. In diesem Fall navigieren Sie bitte bis: **Settings/Compiler/Toolchain executables** klicken dort auf den Button **Auto-detect** und bestätigen dies anschließend mit „OK“

# Der erste eigene Code – Aufbau des Quellcodes

## Vorbemerkungen:

- Der Quellcode besteht aus den im Folgenden dargestellten Abschnitten, die wir uns der Einfachheit halber „graphisch animiert“ anschauen werden.
- Selbstverständlich werden wir aber nach dieser PowerPoint-Präsentation auch eine gemeinsame praktische Übung absolvieren, bevor Sie anschließend durch weitere Einzelübungen Gelegenheit bekommen, das hier Vorgestellte ausführlich zu trainieren.
- Darüber hinaus gilt, dass die am heutigen Tag vermittelten Inhalte bei praktisch allen kommenden Codierungsaufgaben von Bedeutung sein werden. Damit ist dann aber auch deren regelmäßige Wiederholung gesichert.

# Der erste eigene Code – Aufbau des Quellcodes

Wir beginnen mit dem eigentlichen **Hauptprogramm**, das in ANSI C stets mit dem Funktionsnamen **main()** versehen wird:

```
#include<...>
...
#define ...
...
main()
{
    Anweisung1;
    Anweisung2;
    ...
    ...
}
```



# Der erste eigene Code – Aufbau des Quellcodes

Anfang und Ende des Anweisungsblocks werden (wie schon beim Pseudocode üblich) durch **geschwungene Klammern** dargestellt

```
#include<...>
...
#define ...
...

main()
{
    Anweisung1;
    Anweisung2;
    ...
    ...
}
```

# Der erste eigene Code – Aufbau des Quellcodes

Hier **beginnt** der Anweisungsblock der main-Funktion:

```
#include<...>
...
#define ...
...

main()
{
    Anweisung1;
    Anweisung2;
    ...
    ...
}
```

# Der erste eigene Code – Aufbau des Quellcodes

```
#include<...>
...
#define ...
...


main()
{
    Anweisung1;
    Anweisung2;
    ...
    ...
}
```

Hier **endet** er:

# Der erste eigene Code – Aufbau des Quellcodes

Alle **Anweisungen** (Befehle) werden **engerückt** notiert:

```
#include<...>
...
#define ...
...

main()
{
    Anweisung1;
    Anweisung2;
    ...
    ...
    
    Breite der Einrückung:  

    1 Tabulator-Schritt
}
```

# Der erste eigene Code – Aufbau des Quellcodes

Jede Anweisung wird mit einem **Semikolon** abgeschlossen:

```
#include<...>
...
#define ...
...

main()
{
    Anweisung1;
    Anweisung2;
    ...
    ...
}
```

# Der erste eigene Code – Aufbau des Quellcodes

Um komplizierte Abschnitte eines Codes für den Leser erläutern zu können, gibt es die Möglichkeit, Kommentare in einen Quellcode einzutragen, die vom Compiler beim Übersetzen ignoriert werden.

Falls ein Code nur eine einzige Zeile benötigt, so reicht die Notation zweier **Slash-Symbole** unmittelbar vor dem Kommentar:

```
#include<...>
```

```
...
```

```
#define ...
```

```
...
```

```
main()
```

```
{
```

```
    Anweisung1;
```

```
    Anweisung2;
```

```
    ...
```

```
    // einzeliger Kommentar
```

```
}
```

# Der erste eigene Code – Aufbau des Quellcodes

Mehrzeilige Kommentare starten mit `/*` und enden mit `*/`

```
#include<...>
...
#define ...
...

main()
{
    Anweisung1;
    Anweisung2;
    ...
    /*
        mehrzeiliger
        Kommentar
    */
}
```

# Der erste eigene Code – Aufbau des Quellcodes

Oberhalb der main-Funktionen werden die  
**Präprozessor-Anweisungen** notiert.  
(Sie beginnen stets mit dem **Hash-Symbol**)

```
#include<...>
...
#define ...
...

main()
{
    Anweisung1;
    Anweisung2;
    ...
    ...
}
```



# Der erste eigene Code – Aufbau des Quellcodes

Mit **include** werden benötigte Bibliotheken eingebunden  
(dazu heute noch mehr)

```
#include<NameDerBibliothek>
```

```
...
```

```
#define ...
```

```
...
```

```
main()
```

```
{
```

```
    Anweisung1;
```

```
    Anweisung2;
```

```
    ...
```

```
    ...
```

```
}
```

# Der erste eigene Code – Aufbau des Quellcodes

Die Präprozessor-Anweisung **define**  
werden wir erst später behandeln

```
#include<...>
...
#define ...
...

main()
{
    Anweisung1;
    Anweisung2;
    ...
    ...
}
```

# Der erste eigene Code – Aufbau des Quellcodes

Der Bereich unterhalb der Präprozessor-Anweisungen, aber oberhalb vom main kann für (eigene) **Funktionen** verwendet werden.  
Da wir dieses Thema aber erst deutlich später behandeln, werden wir diesen Bereich zunächst ungenutzt lassen.

```
#include<...>
...
#define ...
...
main()
{
    Anweisung1;
    Anweisung2;
    ...
    ...
}
```

# Der erste eigene Code – Aufbau des Quellcodes

Aus genau diesem Grund wollen wir auch die main-Funktion zunächst so „**sparsam**“ wie möglich notieren. Gemeint ist ...

```
#include<...>
...
#define ...
...

main()
{
    Anweisung1;
    Anweisung2;
    ...
    ...

}
```

# Der erste eigene Code – Aufbau des Quellcodes

... dass wir die hier farblich markierte Schreibweise zunächst bewusst unterlassen werden, da wir weder **Parameter**, noch den **Typ des Rückgabewertes**, geschweige denn den **Return-Befehl** bereits behandeln wollen.

```
#include<...>
...
#define ...
...

int main(int argc, char *argv)
{
    Anweisung1;
    Anweisung2;
    ...
    ...

    return 0;
}
```

# Der erste eigene Code – Aufbau des Quellcodes

Der Compiler reagiert dann zwar mit dem folgenden Hinweis:

```
warning: return type defaults to 'int' [-Wimplicit-int]
```

... dies ist aber nur eine **Warnung** (also keine Fehlermeldung) und wird von uns bis auf weiteres ignoriert. Erst wenn wir das Thema der *Funktionen* behandelt haben werden, wird es Sinn machen (zumindest Teile) der angesprochenen Notation zu ergänzen.

```
#include<...>
...
#define ...
...

main()
{
    Anweisung1;
    Anweisung2;
    ...
    ...
}
```

# Der erste eigene Code – Variablen

```
main()
{
    int zahl=5;
    char zeichen='w';
    float x=2.355;

}
```

# Der erste eigene Code – Variablen - Deklarieren

```
main()
{
    int zahl=5;
    char zeichen='w';
    float x=2.355;

}
```



# Der erste eigene Code – Variablen - Definieren

```
main()
{
    int zahl=5;
    char zeichen='w';
    float x=2.355;

}
```

# Der erste eigene Code – Variablen - Initialisieren

```
main()
{
    int zahl=5;
    char zeichen='w';
    float x=2.355;

}
```

# Der erste eigene Code – Variablen – Definieren+Deklarieren

```
main()
{
    int zahl=5;
    char zeichen='w';
    float x=2.355;
```

Alle Variablen **müssen** in einem Quellcode **vor** der ersten Verwendung deklariert und definiert werden!

```
}
```

# Der erste eigene Code – Variablen – Definieren+Deklarieren

```
main()
{
    int zahl=5;
    char zeichen='w';
    float x=2.355;
```

Eine sofortige Initialisierung ist aber nicht zwingend gefordert:

```
float kommazahl;
```

```
}
```

# Der erste eigene Code – Variablen – Zuweisung

```
main()
{
    int a=1;
    int b=2;
    int c=3;
}
```

Erste Zuweisungen werden  
als Initialisierung bezeichnet

# Der erste eigene Code – Variablen – Zuweisung

```
main()
{
    int a=1;
    int b=2;
    int c=3;

    a=1;
    b=2;
    c=3;
}
```

Erst beim **Überschreiben** wird von einer Zuweisung gesprochen

# Der erste eigene Code – Variablen – Zuweisung

Die Definition (und Deklaration) jeder einzelnen Variable geschieht natürlich stets nur **einmalig**

```
main()
{
```

```
    int a=1;
    int b=2;
    int c=3;
```

```
    a=1;
    b=2;
    c=3;
```

```
}
```

# Der erste eigene Code – Variablen – Zuweisung

```
main()
{
    int a=1;
    int b=2;
    int c=3;
```

Keine erneute Definition!

```
} a=1;
  b=2;
  c=3;
```

```
}
```



# Der erste eigene Code – Variablen – Zuweisung

```
main()
{
    int a=1;
    int b=2;
    int c=3;

    a=1;
    b=2;
    c=3;

    a= a+1 ;
    b= 2*b ;
    c= a+b ;

}
```

Und natürlich können auch die Resultate von Rechnungen zugewiesen werden:

# Der erste eigene Code – Ausgabe – String

## Code-Ausschnitt

```
printf("Hallo Welt");
```

Mit der Funktion **printf** können Texte auf der Konsole ausgegeben werden. Solche Texte werden auch als „Zeichenkette“ oder „String“ bezeichnet.

## Konsolen-Ausgabe

Hallo Welt

# Der erste eigene Code – Ausgabe – String

## Code-Ausschnitt

```
printf("Hallo Welt");
```

Um dem Compiler deutlich zu machen, wo ein String anfängt und wo er aufhört, muss er stets in (doppelten) **Anführungszeichen** notiert werden!

## Konsolen-Ausgabe

```
Hallo Welt
```

# Der erste eigene Code – Bibliothek – **stdio.h**

## Code-Ausschnitt

```
printf("Hallo Welt");
```

Die Funktion **printf** ist (zumindest der reinen Lehre nach) dem Compiler unbekannt. Der Compiler weiß also nicht, was er zu tun hat, wenn er den Befehl printf kompilieren soll. Daher muss zunächst eine **Bibliothek** eingeführt werden, in der dies „erklärt“ wird.

Es gibt eine große Anzahl von Bibliotheken, die die unterschiedlichsten Funktionen erklären. Die Bibliothek, in der printf erklärt wird, heißt **stdio.h** (Abkürzung für STanDard Input Output)

## Quellcode

```
#include<stdio.h>
```

```
main()  
{  
    printf(...)
```

```
}
```

## Konsolen-Ausgabe

```
Hallo Welt
```

# Der erste eigene Code – Ausgabe-Formate: Zeilenumbruch

## Code-Ausschnitt

```
printf("Hallo\nWelt");
```

Das Format-Zeichen `\n` (= „next Line“) sorgt für einen Umbruch

## Konsolen-Ausgabe

```
Hallo  
Welt
```

# Der erste eigene Code – Ausgabe-Formate: Tabulator

## Code-Ausschnitt

```
printf("\tHallo Welt");
```

Das Format-Zeichen `\t` sorgt für einen Tabulator-Sprung (also ein „Einrücken“)

## Konsolen-Ausgabe

Hallo Welt

# Der erste eigene Code – Ausgabe-Formate: mehrere

## Code-Ausschnitt

```
printf("\tHallo\nWelt");
```

Natürlich können auch mehrere Format-Symbole gleichzeitig genutzt werden

## Konsolen-Ausgabe

```
Welt      Hallo
```

# Der erste eigene Code – Ausgabe-Formate: Variablen

## Code-Ausschnitt

```
int zahl=123;  
printf("Die Zahl %d ist einfach toll :-D", zahl);
```

%d (= dezimal) ist der Platzhalter für ganze Zahlen

## Konsolen-Ausgabe

Die Zahl 123 ist einfach toll :-D



# Der erste eigene Code – Ausgabe-Formate: Variablen

## Code-Ausschnitt

```
int zahl=123;  
printf("Die Zahl %d ist einfach toll :-D", zahl);
```

Hinter dem String und durch Komma getrennt  
werden die auszugebenden Variablen notiert

## Konsolen-Ausgabe

```
Die Zahl 123 ist einfach toll :-D
```

# Der erste eigene Code – Ausgabe-Formate: Variablen

## Code-Ausschnitt

```
int i=100;  
char c='g';  
float f=5234.57;  
printf("%d%c Gold kosten %f Euro", i, c, f);
```

Natürlich können auch mehrere Variablen  
(beliebigen Typs) ausgegeben werden

## Konsolen-Ausgabe

100g Gold kosten 5234.570000 Euro

# Der erste eigene Code – Ausgabe-Formate: Variablen

## Code-Ausschnitt


```
int i=100;  
char c='g';  
float f=5234.57;  
printf("%d%c Gold kosten %.2f Euro", i, c, f);
```

Die **Anzahl** der ausgegebenen **Nachkommastellen** kann festgelegt werden. (Es wird mathematisch korrekt gerundet)

## Konsolen-Ausgabe

100g Gold kosten 5234.57 Euro

# Der erste eigene Code – Gemeinsame Übung A\_01\_02\_01



## Aufgabe\_01\_02\_01

Gegeben sei das folgende PAP:

```
graph TD
    Start([Start]) --> A1[Ausgabe()]
    A1 --> A2[Ausgabe()]
    A2 --> A3[/Ausgabe: "Welche Meinung?"  
zeichen "ist der" position" Buchstabe im Alphabet/]
    A3 --> Ende([Ende])
```

Hinweis:  
"tester Text" wird hier in Anführungszeichen notiert  
... Variablen (bzw. deren Werte) stehen außerhalb der Anführungszeichen


**Aufgabenstellung:**  
Bitte erstellen Sie dazu einen geeigneten **Quellcode** in ANSI C.  
Versuchen Sie dabei bitte aus Trainingsgründen, die gesamte Ausgabe (entsprechend des dargestellten Layouts) mit einem einzigen printf-Befehl ausführen zu lassen.

**Hinweis:**  
Der besseren Lesbarkeit des Quellcodes wegen, kann es in der Praxis sinnvoll sein, diese Aufgabe mit mehreren printf-Befehlen zu lösen, da auf diese Weise im Quellcode deutlich besser ablesbar sein wird, „was“ (und in welchem Layout) ausgegeben werden soll.

WBS TRAINING AG  
Lorenzweg 5  
D-12099 Berlin  
Amtsgericht Berlin HRB 68531  
Sitz der Gesellschaft: Berlin

Vorstand:  
Heinrich Kronbichler,  
Joachim Giese  
Aufsichtsrat (Vorsitz): Dr. Daniel Stadler  
USt-IDNr.: DE 209 768 248

GLS Gemeinschaftsbank eG  
IBAN: DE18 4306 0967 1146 1814 00  
BIC: GENODEM33GLS



GLS Gemeinschaftsbank  
eG (In der Form einer AG) Reg. Nr. 310304 (HRB)  
Zulassung nach AGG Nr. 40/2010 (AGG)

**VIELEN DANK  
FÜR IHRE  
AUFMERKSAMKEIT!**