



# **Einführung in die Programmiersprache C**

---

# Agenda

- Tag 1: Einführung in C
- Tag 2: Vertiefung in C
- Tag 3: Vertiefung in C
- Tag 4: C Projekt
- Tag 5: C Projekt

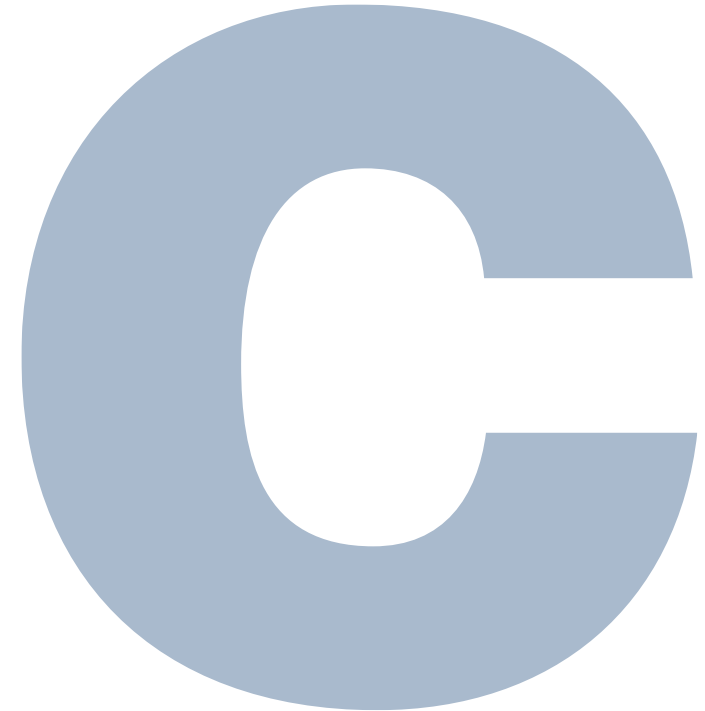


---

# Einführung in die Programmiersprache C

## Was ist C?

- Entwickelt von Dennis Ritchie 1972 bei Bell Labs
- Imperative und prozedurale Programmiersprache
- Basis für viele moderne Sprachen (C++, Java)
- Weit verbreitet für System- und Anwendungsprogrammierung
- Bietet direkte Kontrolle über Speicher und Hardware
- Effizient und performant



# Einführung in die Programmiersprache C

## Einsatzgebiete

- Betriebssysteme
  - Linux
  - Windows
- Embedded Systems
  - Anwendungen in Mikrocontrollern
  - Hardware-naher Programmierung
- Netzwerkprogrammierung
  - Server-Software
  - Protokollen
- Compiler für andere Programmiersprachen zu schreiben
- C wird weltweit in Informatikstudiengängen als Lehrsprache genutzt



# Einführung in die Programmiersprache C

## Imperative Programmiersprache

### Imperativ

```
int sum = 0;

for(int i = 1; i <= 5; i++) {
    sum += i;
}

printf("Summe: %d", sum);
```

### Deklarativ

```
SELECT SUM(value) FROM numbers WHERE value BETWEEN 1 AND 5;
```

# Einführung in die Programmiersprache C


## Prozedurale Programmiersprache

- Strukturierung des Codes durch die Nutzung von Prozeduren oder Funktionen
- Ziel: Wiederverwendung und Struktur durch Modularität zu fördern

```
int add(int a, int b) {  
    return a + b;  
}  
  
int main(void) {  
    int sum = add(5, 10);  
    printf("Summe: %d", sum);  
    return 0;  
}
```

# Einführung in die Programmiersprache C

## Struktur eines C-Programms



```
#include <stdio.h> // Laden der Standard-Eingabe-/Ausgabebibliothek

// Hauptfunktion des Programms
int main(void)
{
    /* Ausgabe der Zeichenkette "Hello, World!" auf dem Bildschirm */
    printf("Hello, World!\n");

    // Rückgabewert 0 bedeutet, dass das Programm erfolgreich beendet wurde
    return 0;
}
```

# Einführung in die Programmiersprache C

## Aufsetzen der Programmierumgebung

<https://code.visualstudio.com/>

- Projektordner anlegen und in VS Code öffnen
- C/C++ for Visual Studio Code Extension installieren

<https://sourceforge.net/projects/mingw/>

- Mingw32-base installieren
- Pfad setzen
  - Umgebungsvariablen für eigenen Account
  - Pfad “C:\MingW\bin” hinzufügen

“Hello World” Programm kompilieren und laufen lassen

```
#include <stdio.h>
int main()
{
    printf("Hello, World! \n");

    return 0;
}
```

```
gcc .\hello-world.c -o hello-world.exe

.\hello-world.exe
```



# Einführung in die Programmiersprache C

## Printf und Scanf

**printf:** Ausgabe von Text und Werten auf dem Bildschirm

**scanf:** Einlesen von Benutzereingaben

%d: Ganzzahl

%f: Gleitkommazahl

%c: Einzelnes Zeichen

%s: Zeichenkette (String)

Hinweis: Benutze das &-Symbol (Adressoperator), um die Speicheradresse der Variablen anzugeben

```
#include <stdio.h>

int main(void)
{
    int zahl;

    printf("Geben Sie eine Zahl ein: ");

    scanf("%d", &zahl);

    printf("Sie haben die Zahl %d eingegeben.\n", zahl);

    return 0;
}
```



# Aufgabenblatt 1

Aufgaben 1 bis 8

<https://github.com/koenig101/vorpraktikum2024>

# Einführung in die Programmiersprache C

## Grundlegende Datentypen in C

- **int** (Integer)
  - Speichert Ganzzahlen: z.B. 158, -10, 0
  - Speichergröße: mindestens 32 Bit
- **float** (Gleitkomma)
  - Speichert Fließkommazahlen: z.B. 3.14, -0.001
  - Unterstützt wissenschaftliche Notation: z.B. 1.7e4
  - Ausgabeformate: %f (normal), %e (wissenschaftlich), %g (C entscheidet)
- **double** (Erweiterte Präzision)
  - Ähnlich wie float, aber mit höherer Präzision
  - Standardmäßig werden Fließkommazahlen als interpretiert
- **char** (Einzelzeichen)
  - Speichert ein einzelnes Zeichen: z.B. 'a', ';', '0', '\n'
- **bool** (Wahrheitswert)
  - Speichert true (1) oder false (0)
  - <stdbool.h> definiert bool, true, false

```
#include <stdio.h>

int main (void)
{
    int        integerVar = 20;
    float       floatingVar = 231.79f;
    double      doubleVar = 8.44e+11;
    char        charVar = 'W';

    _Bool       boolVar = 0;

    printf ('integerVar = %i\n', integerVar);
    printf ('floatingVar = %f\n', floatingVar);
    printf ('doubleVar = %e\n', doubleVar);
    printf ('doubleVar = %g\n', doubleVar);
    printf ('charVar = %c\n', charVar);

    printf ('boolVar = %i\n', boolVar);

    return 0;
}
```

# Einführung in die Programmiersprache C

## Type Specifiers in C: long, short, unsigned und signed

- **long int**
  - Erweitert den Bereich eines int
  - Beispiel: `long int factorial;`
  - Formatierung: `%li`
- **long long int**
  - Mindestens 64-Bit genau
  - Beispiel: `long long int maxAllowedStorage;`
  - Formatierung: `%lli`
- **short int**
  - Speichert kleinere Ganzzahlen, spart Speicher
  - Beispiel: `short int smallNumber;`
  - Format: `%hi`
- **unsigned int**
  - Speichert nur positive Werte, erweitert den Bereich
  - Beispiel: `unsigned int counter;`
  - Format: `%u`
- **signed int**
  - Explizite Deklaration für vorzeichenbehaftete Zahlen
  - Beispiel: `signed int temperature;`
- **Zusätzliche Hinweise:**
  - Kombinationen möglich: z.B. `unsigned long int`, `short unsigned int`.
  - Konstantenbildung: Durch Anhängen von `L`, `LL`, `U`, `UL` usw. (z.B. `20000UL`).

# Einführung in die Programmiersprache C

## Ganzzahl- und Fließkomma-Umwandlungen in C

- **Implizite Umwandlungen**
  - Konvertierung von float zu int schneidet Dezimalstellen ab
- **Ganzzahl-Division**
  - verwirft Dezimalstellen. Gemischte Typ-Division (int/float) erhält Dezimalstellen
- **Typumwandlungs-Operator**
  - Konvertiert temporär Datentypen für Ausdrücke. Verändert den ursprünglichen Variablentyp nicht
- **Zuweisungsoperatoren**
  - Kombiniert Arithmetik mit Zuweisung
  - z. B. `count += 10` ist `count = count + 10`

```
#include <stdio.h>

int main (void)
{
    float  f1 = 123.125, f2;
    int    i1, i2 = -150;
    char   c = 'a';

    i1 = f1;                // floating to integer conversion

    f1 = i2;                // integer to floating conversion

    f1 = i2 / 100;          // integer divided by integer

    f2 = i2 / 100.0;        // integer divided by a float

    f2 = (float) i2 / 100;   // type cast operator

    return 0;
}
```



# Aufgabenblatt 1

Aufgaben 9 bis 12



# Einführung in die Programmiersprache C

## Grundlegende Datentypen in C

- **int** (Integer)
  - Speichert Ganzzahlen: z.B. 158, -10, 0
  - Speichergröße: mindestens 32 Bit
- **float** (Gleitkomma)
  - Speichert Fließkommazahlen: z.B. 3.14, -0.001
  - Unterstützt wissenschaftliche Notation: z.B. 1.7e4
  - Ausgabeformate: %f (normal), %e (wissenschaftlich), %g (C entscheidet)
- **double** (Erweiterte Präzision)
  - Ähnlich wie float, aber mit höherer Präzision
  - Standardmäßig werden Fließkommazahlen als interpretiert
- **char** (Einzelzeichen)
  - Speichert ein einzelnes Zeichen: z.B. 'a', ';', '0', '\n'
- **bool** (Wahrheitswert)
  - Speichert true (1) oder false (0)
  - <stdbool.h> definiert bool, true, false

```
#include <stdio.h>

int main (void)
{
    int      integerVar = 20;
    float    floatingVar = 231.79f;
    double   doubleVar = 8.44e+11;
    char     charVar = 'W';

    _Bool    boolVar = 0;

    printf ('integerVar = %i\n', integerVar);
    printf ('floatingVar = %f\n', floatingVar);
    printf ('doubleVar = %e\n', doubleVar);
    printf ('doubleVar = %g\n', doubleVar);
    printf ('charVar = %c\n', charVar);

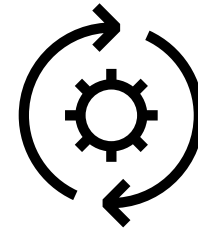
    printf ('boolVar = %i\n', boolVar);

    return 0;
}
```

# Einführung in die Programmiersprache C

## Programmschleifen

- Computer sind hervorragend darin, sich wiederholende Berechnungen effizient durchzuführen
- C bietet spezifische Konstrukte, um Wiederholungen effektiv zu handhaben
  - **for**-Anweisung
  - **while**-Anweisung
  - **do**-Anweisung
  - **break**-Anweisung
  - **continue**-Anweisung
- Helfen, Wiederholungen (Redundanz) im Code zu reduzieren, indem sie die mehrfache Ausführung eines Code-Blocks unter kontrollierten Bedingungen ermöglichen



# Einführung in die Programmiersprache C

## for-Schleife

- **Zweck:** Automatisiert repetitive Aufgaben, indem über eine Reihe von Werten iteriert wird
- **Funktionsweise**
  - `init_expression`: Initialisiert Variablen, bevor die Schleife beginnt
  - `loop_condition`: Führt die Schleife aus, solange diese Bedingung TRUE ist
  - `loop_expression`: Aktualisiert die Variablen nach jeder Schleifeniteration
- **Relationale Operatoren**
  - `<`, `<=`, `>`, `>=`, `==`, `!=`
  - Sorgen für die richtige Ausführung der Schleife durch Überprüfung der Bedingungen
- **Hinweise**
  - Gut für Aufgaben mit einer bekannten Anzahl von Iterationen
  - Verschachtelte Schleifen möglich

```
for (init_expression; loop_condition; loop_expression)
    statement;
```

```
for (n = 1; n <= 10; ++n) {
    triangularNumber += n;
    printf("%i %i\n", n, triangularNumber);
}
```

# Einführung in die Programmiersprache C

## while-Schleife

- **Zweck:** Führt eine Schleife aus, solange eine Bedingung erfüllt ist.
- **Funktionsweise**
  - Die Bedingung innerhalb der Klammern wird ausgewertet
  - Ist die Bedingung TRUE, wird die Anweisung ausgeführt
  - Nach Ausführung der Anweisung wird die Bedingung erneut ausgewertet
  - Die Schleife läuft weiter, bis die Bedingung FALSE wird. Relationale Operatoren
- **Hinweise**
  - Ideal für Situationen, in denen die Anzahl der Iterationen nicht im Voraus bekannt ist
  - Endlosschleifen können auftreten

```
while (Bedingung)
    Anweisung;
```

```
int count = 0;
while (count < 5) {
    printf("%i\n", count);
    count++;
}
```

# Einführung in die Programmiersprache C

## do-Schleife

- **Zweck:** Führt eine Schleife mindestens einmal aus, bevor die Bedingung geprüft wird
- **Funktionsweise**
  - Die Anweisung wird einmal ausgeführt, bevor die Bedingung geprüft wird
  - Wenn die Bedingung TRUE ist, wird die Anweisung erneut ausgeführt
  - Die Schleife läuft weiter, bis die Bedingung FALSE wird
- **Hinweise**
  - Nützlich, wenn der Schleifeninhalt mindestens einmal ausgeführt werden muss, unabhängig von der Bedingung

```
do
    Anweisung;
while (Bedingung);
```

```
int number, right_digit;
printf("Gib deine Zahl ein.\n");
scanf("%i", &number);

do {
    right_digit = number % 10;
    printf("%i", right_digit);
    number = number / 10;
} while (number != 0);
```

# Einführung in die Programmiersprache C

## do-Schleife

- **Zweck:** Führt eine Schleife mindestens einmal aus, bevor die Bedingung geprüft wird
- **Funktionsweise**
  - Die Anweisung wird einmal ausgeführt, bevor die Bedingung geprüft wird
  - Wenn die Bedingung TRUE ist, wird die Anweisung erneut ausgeführt
  - Die Schleife läuft weiter, bis die Bedingung FALSE wird
- **Hinweise**
  - Nützlich, wenn der Schleifeninhalt mindestens einmal ausgeführt werden muss, unabhängig von der Bedingung

```
do
    Anweisung;
while (Bedingung);
```

```
int number;
printf("Gib deine Zahl ein.\n");
scanf("%i", &number);

do {
    printf("%i", number % 10);
    number = number / 10;
}
while (number != 0);
```



# Einführung in die Programmiersprache C

## break und continue Anweisungen

### – break

- Beendet die aktuelle Schleife sofort, unabhängig davon, ob die Schleifenbedingung noch TRUE ist
- **Anwendung:** Nützlich, um eine Schleife vorzeitig zu verlassen, z.B. bei einem Fehlerzustand

### – continue

- Überspringt den Rest des Schleifenkörpers und fährt mit der nächsten Iteration fort
- **Anwendung:** Nützlich, um bestimmte Iterationen einer Schleife zu überspringen, ohne die Schleife zu beenden

### – Hinweise

- leistungsfähige Werkzeuge, sollten jedoch mit Bedacht eingesetzt werden, um die Lesbarkeit des Codes zu erhalten

```
for (int i = 0; i < 10; ++i) {  
    if (i == 5)  
        break; // Schleife endet bei i == 5  
  
    if (i % 2 == 0)  
        continue; // Überspringt gerade Zahlen  
  
    printf("%i\n", i);  
}
```



# Aufgabenblatt 1

Ab Aufgabe 12

# Einführung in die Programmiersprache C

## if-Anweisung

- **Zweck**
  - Ermöglicht die bedingte Ausführung von Code basierend auf eines booleschen Ausdruck
- **Erklärung**
  - Die if-Anweisung überprüft den Ausdruck. Wenn das Ergebnis wahr ist, wird die folgende Anweisung ausgeführt.
- **Beispiel**
  - Die Nachricht wird nur ausgegeben, wenn x größer als 10 ist

```
if (Ausdruck)  
    Anweisung;
```

```
if (x > 10)  
    printf("x ist größer als 10\n");
```

# Einführung in die Programmiersprache C

## else if-Konstruktion

- **Zweck**
  - Erweitert die `if`-Anweisung, um mehrere Bedingungen zu behandeln
- **Erklärung**
  - `else if` ermöglicht das Testen mehrerer Bedingungen nacheinander
  - Die erste wahr-Bedingung führt ihre entsprechende Anweisung aus
- **Beispiel**
  - Hier wird überprüft, ob `x` negativ, null oder positiv ist, und die entsprechende Nachricht ausgegeben

```
if (Ausdruck1)
    Anweisung1;
else if (Ausdruck2)
    Anweisung2;
else
    Anweisung3;
```

```
if (x < 0)
    printf("x ist negativ\n");
else if (x == 0)
    printf("x ist null\n");
else
    printf("x ist positiv\n");
```

# Einführung in die Programmiersprache C

## switch-Anweisung

- **Zweck**
  - Vereinfacht die Mehrwegverzweigung basierend auf dem Wert eines Ausdrucks
- **Erklärung**
  - Die switch-Anweisung überprüft den Ausdruck und führt den passenden case-Block aus
  - Der default-Block wird ausgeführt, wenn kein anderer Fall zutrifft
- **Beispiel**
  - Hier wird der Wert von Tag überprüft, und der entsprechende Wochentag ausgegeben

```
switch (Ausdruck) {  
    case Wert1:  
        Anweisungen;  
        break;  
    case Wert2:  
        Anweisungen;  
        break;  
    ...  
    default:  
        Anweisungen;  
        break;  
}
```

```
switch (Tag) {  
    case 1:  
        printf("Montag\n");  
        break;  
    case 2:  
        printf("Dienstag\n");  
        break;  
    default:  
        printf("Ungültiger Tag\n");  
}
```



# Aufgabenblatt 2

## Aufgaben 1 bis 3



# Einführung in die Programmiersprache C

## Arbeiten mit Arrays

### – Definition

- Ein Array ist eine Sammlung von Daten, die unter einem gemeinsamen Namen gespeichert sind
- Jedes Element in einem Array ist durch einen Index zugänglich

### – Deklaration

- Ein Array muss vor der Verwendung deklariert werden, indem der Datentyp und die Anzahl der Elemente festgelegt werden

### – Zugriff auf Elemente

- Jedes Element in einem Array kann mit einem Index aufgerufen werden
- Der Index beginnt bei 0, das heißt, das erste Element hat den Index 0

### – Initialisierung

- Die Initialisierung eines Arrays legt die Anfangswerte für die Elemente fest

```
// Deklaration
int zahlen[10];

// Initialisierung
zahlen[0] = 5;
printf("%d", zahlen[0]);

// Initialisierung bei der Deklaration
int zahlen[5] = {1, 2, 3, 4, 5};

// Schleife, um Array zu füllen
for (int i = 0; i < 10; i++) {
    zahlen[i] = i * 2;
}

// Multidimensionales Array
int matrix[3][4]; // Ein 3x4-Array (Matrix) mit 3 Zeilen und 4 Spalten
matrix[1][2] = 5; // Setzt das Element in Zeile 2, Spalte 3 auf
```



# Aufgabenblatt 2

## Aufgaben 4 bis 8

# Einführung in die Programmiersprache C

## Grundlagen von Funktionen

- **Was ist eine Funktion?**

- Ein Codeblock, der eine spezifische Aufgabe erfüllt
- Funktionen helfen dabei, große Programme in kleinere, verwaltbare und wiederverwendbare Codeblöcke zu unterteilen.

- **Vorteile**

- Verbessert die Lesbarkeit und Wartbarkeit des Codes
- Erleichtert die Wiederverwendung von Code

```
// Funktion Definition
void printMessage(void) {
    printf("Programmieren macht Spass.\n");
}

// Funktion Aufruf
int main(void) {
    printMessage(); // Aufruf der Funktion
    return 0;
}
```

# Einführung in die Programmiersprache C

## Lokale, globale und statische Variablen

- **Lokale Variablen**

- Werden innerhalb einer Funktion definiert
- Nur innerhalb dieser Funktion zugänglich

- **Globale Variablen**

- Werden außerhalb von Funktionen definiert
- Können von jeder Funktion im Programm verwendet werden

- **Statische Variablen**

- Behalten ihren Wert zwischen Funktionsaufrufen.

```
void myFunction(void) {  
    int localVar = 5; // Lokale Variable  
    printf("%d\n", localVar);  
}
```

```
int globalVar = 10;  
  
void myFunction(void) {  
    printf("%d\n", globalVar);  
}
```

```
void myFunction(void) {  
    static int count = 0; // Statische Variable  
    count++;  
    printf("%d\n", count);  
}  
  
int main(void) {  
    myFunction(); // Gibt 1 aus  
    myFunction(); // Gibt 2 aus  
    return 0;  
}
```

# Einführung in die Programmiersprache C

## Verwenden von Arrays mit Funktionen

### – Übergeben von Arrays

- Arrays werden als Referenz und nicht als Wert übergeben
- Statt das gesamte Array zu kopieren, wird nur die Speicheradresse des ersten Elements übergeben
- Da das Array als Referenz übergeben wird, können Änderungen, die in der Funktion am Array vorgenommen werden, das ursprüngliche Array beeinflussen
- Die Größe des Arrays muss mit angegeben werden
- Wenn die Funktion außerhalb der Grenzen des Arrays zugreift (z.B. durch fehlerhafte Indexierung), kann es zu unerwartetem Verhalten oder Abstürzen des Programms kommen

```
void printArray(int arr[], int size) {  
    for(int i = 0; i < size; i++) {  
        printf("%d ", arr[i]);  
    }  
}
```

```
void printMatrix(int matrix[3][3]) {  
    for(int i = 0; i < 3; i++) {  
        for(int j = 0; j < 3; j++) {  
            printf("%d ", matrix[i][j]);  
        }  
        printf("\n");  
    }  
}
```

# Einführung in die Programmiersprache C

## Daten aus Funktionen zurückgeben

- **Rückgabe eines Werts**
  - Funktionen können einen Wert an die aufrufende Funktion zurückgeben
  - Der zurückgegebene Datentyp wird in der Funktionsdefinition angegeben
- **Rückgabe mehrerer Werte**
  - C erlaubt nicht die direkte Rückgabe mehrerer Werte
  - Verwendung Zeigern oder Strukturen, um dies zu erreichen

```
int add(int a, int b) {  
    return a + b;  
}
```



# Einführung in die Programmiersprache C

## Top-Down-Programmierung

- **Top-Down-Programmierung**
  - Beginnen Sie mit einem high-level Design und teile es in kleinere, detailliertere Funktionen auf
  - Jede Funktion sollte eine spezifische Aufgabe im größeren Programm erfüllen
- **Vorteile**
  - Macht komplexe Programme leichter verständlich und verwaltbar
  - Ermöglicht parallele Entwicklung und Tests

```
int main(void) {  
    initialize();  
    processInput();  
    computeResults();  
    displayResults();  
    return 0;  
}
```

# Einführung in die Programmiersprache C

## Aufrufen von Funktionen innerhalb anderer Funktionen & Rekursion

### – Funktionsaufrufe innerhalb von Funktionen

- Funktionen können andere Funktionen aufrufen, was die Wiederverwendung von Code fördert
- Jede Funktion sollte eine spezifische Aufgabe im größeren Programm erfüllen

```
void calculate(void) {  
    int sum = add(3, 4);  
    printSum(sum);  
}
```

### – Rekursive Funktionen

- Eine Funktion, die sich selbst aufruft
- Vereinfacht manche Probleme wie Berechnung der Fakultät
- Vorsicht: endlose Rekursion vermeiden

```
int factorial(int n) {  
    if (n == 0)  
        return 1;  
    else  
        return n * factorial(n - 1);  
}
```



# Aufgabenblatt 2

## Aufgaben 9 bis 13

# Einführung in die Programmiersprache C

## Schreiben von Dateien

- **Datei öffnen**

- Verwende die Funktion `fopen()` zum Öffnen oder Erstellen einer Datei

- **Daten in die Datei schreiben**

- Verwende `fprintf()` für formatiertes Schreiben

**Vorteile**

- **Datei schließen**

- Schließen die Datei mit `fclose()`, um sicherzustellen, dass alle Daten geschrieben werden

- **Anmerkungen**

- `FILE` ist ein Datentyp in C, der verwendet wird, um Dateien zu handhaben. Es wird durch die `stdio.h`-Bibliothek bereitgestellt
- `*fp` ist ein Zeiger auf eine Struktur vom Typ `FILE`. Dieser Zeiger wird verwendet, um auf die Datei zuzugreifen, die Sie öffnen oder bearbeiten möchten

```
#include <stdio.h>

int main() {
    FILE *fp = fopen("output.txt", "w");
    if (fp == NULL) {
        printf("Fehler beim Öffnen der Datei!\n");
        return 1;
    }

    fprintf(fp, "Hallo, dies ist ein Text in der Datei.\n");
    fclose(fp);

    return 0;
}
```

# Einführung in die Programmiersprache C

## Lesen von Dateien

- **Datei öffnen**
  - Verwende `fopen()` im Lesemodus
- **Daten aus der Datei lesen**
  - Verwende `fscanf()` für formatiertes Lesen
- **Datei schließen**
  - Schließe die Datei mit `fclose()`, um sicherzustellen, dass alle Daten geschrieben werden
- **Anmerkungen**
  - `char buffer[100];` ist ein Array von Zeichen (also ein Zeichenpuffer) mit einer Größe von 100 Zeichen
  - Dieser Puffer wird verwendet, um Zeichenketten (Strings) aus der Datei zu lesen und zu speichern, bevor sie auf dem Bildschirm ausgegeben werden

```
#include <stdio.h>

int main() {
    FILE *fp = fopen("output.txt", "r");
    char buffer[100];

    if (fp == NULL) {
        printf("Fehler beim Öffnen der Datei!\n");
        return 1;
    }

    while (fgets(buffer, 100, fp) != NULL) {
        printf("%s", buffer);
    }

    fclose(fp);

    return 0;
}
```



# Aufgabenblatt 2

Aufgaben 14 bis 18

**ABB**





**Break**  
15 min

