HTL anichstraße

Biomedizintechnik · Elektronik · Elektrotechnik · Maschinenbau · Wirtschaftsingenieure

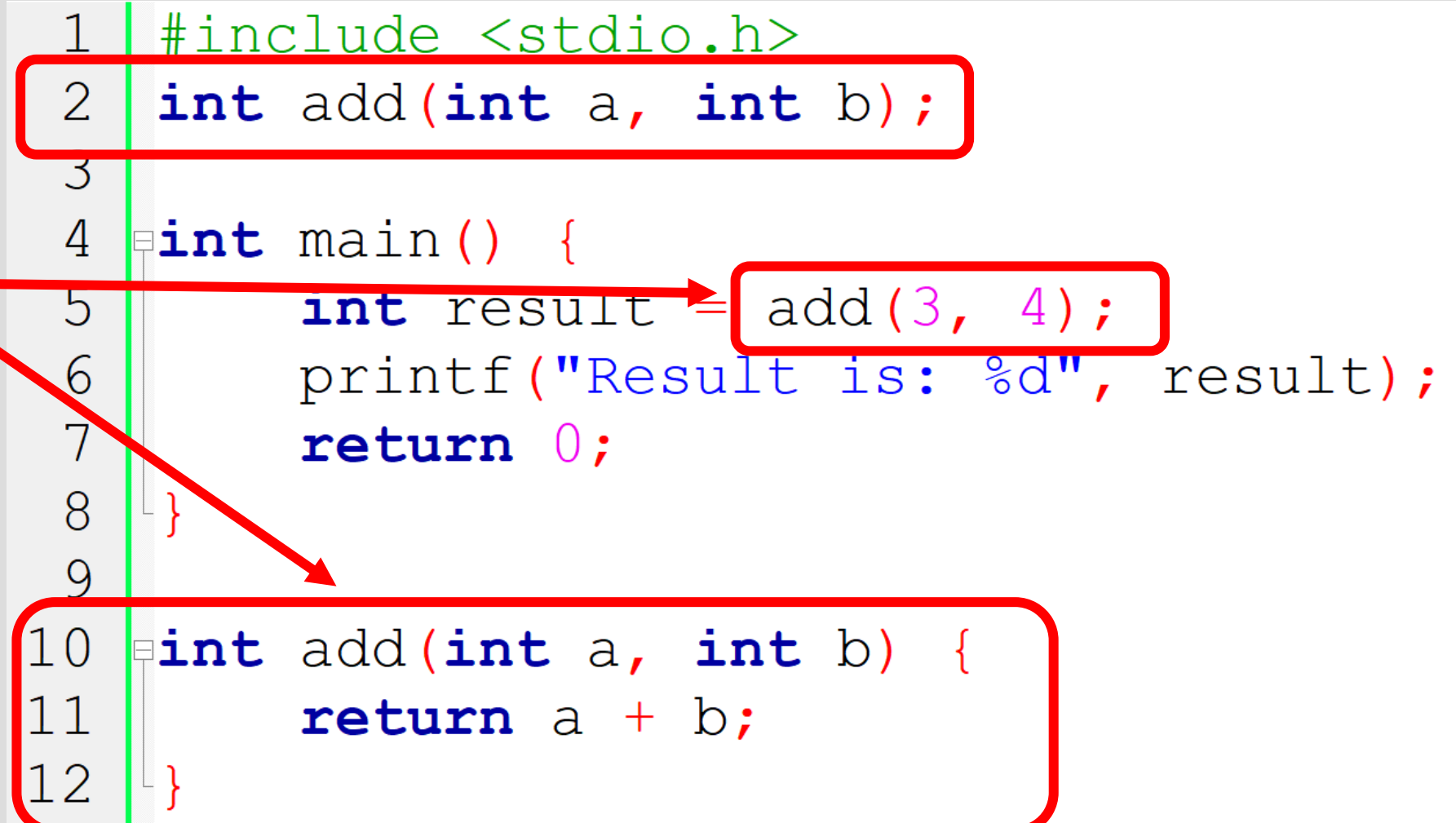
WEITERE THEMEN IN C

Funktionen --- Pointers (Zeiger) --- „call by value vs. call by reference“ --- File IO

FUNKTIONEN IN C

3 Schritte:

1. Deklaration
2. Definition
3. Aufruf (Call)



FUNKTIONEN IN C

- Deklaration ist optional, aber „good practice“, denn:

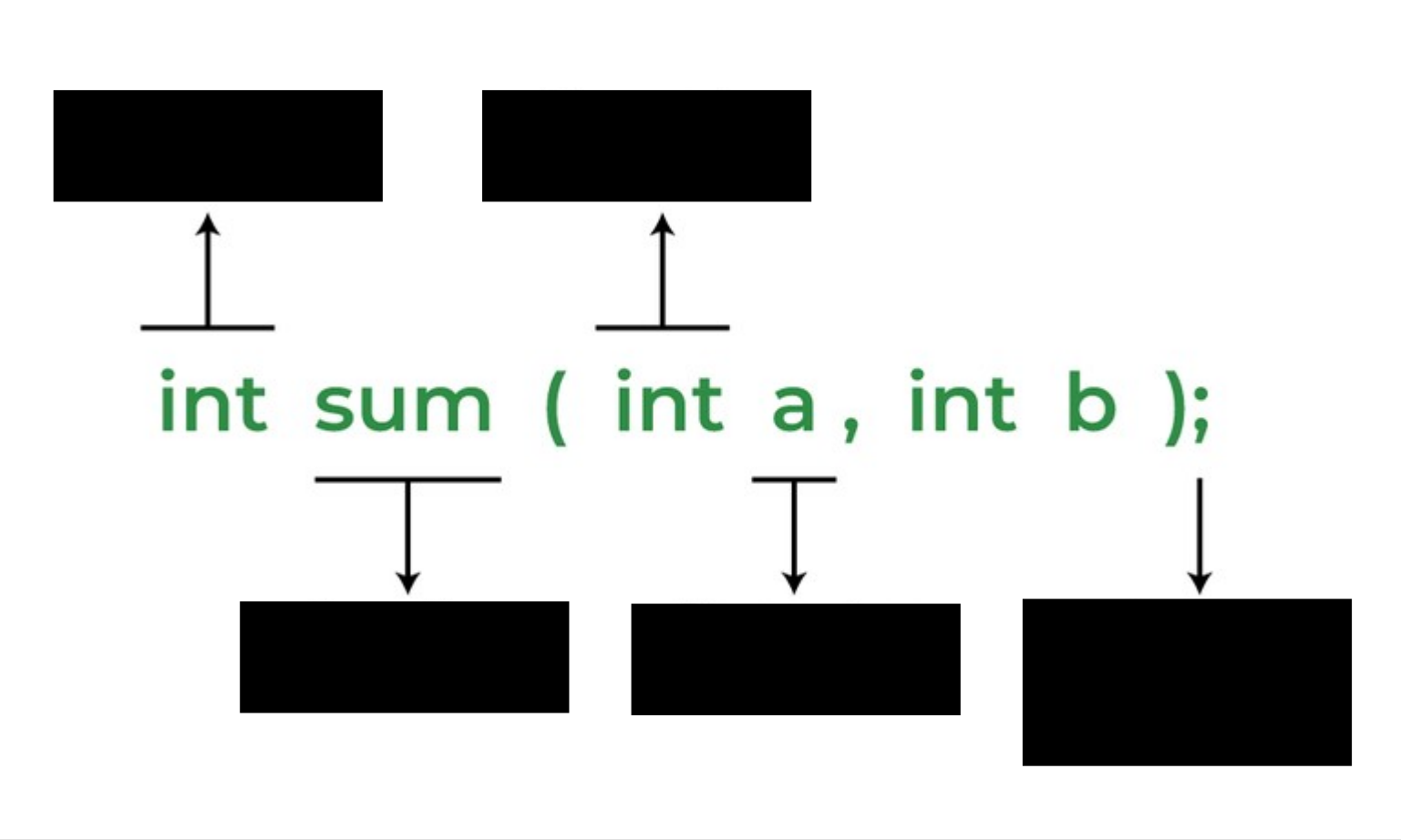
```c
#include <stdio.h>
//int add(int a, int b);

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

int main() {
    int result = add(3, 4);
    printf("Result is: %d", result);
    return 0;
}
```

```c
#include <stdio.h>
//int add(int a, int b);

int main() {
    int result = add(3, 4);
    printf("Result is: %d", result);
    return 0;
}

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

- Funktionen müssen dem Compiler beim Aufruf bekannt sein!

FUNKTIONEN IN C

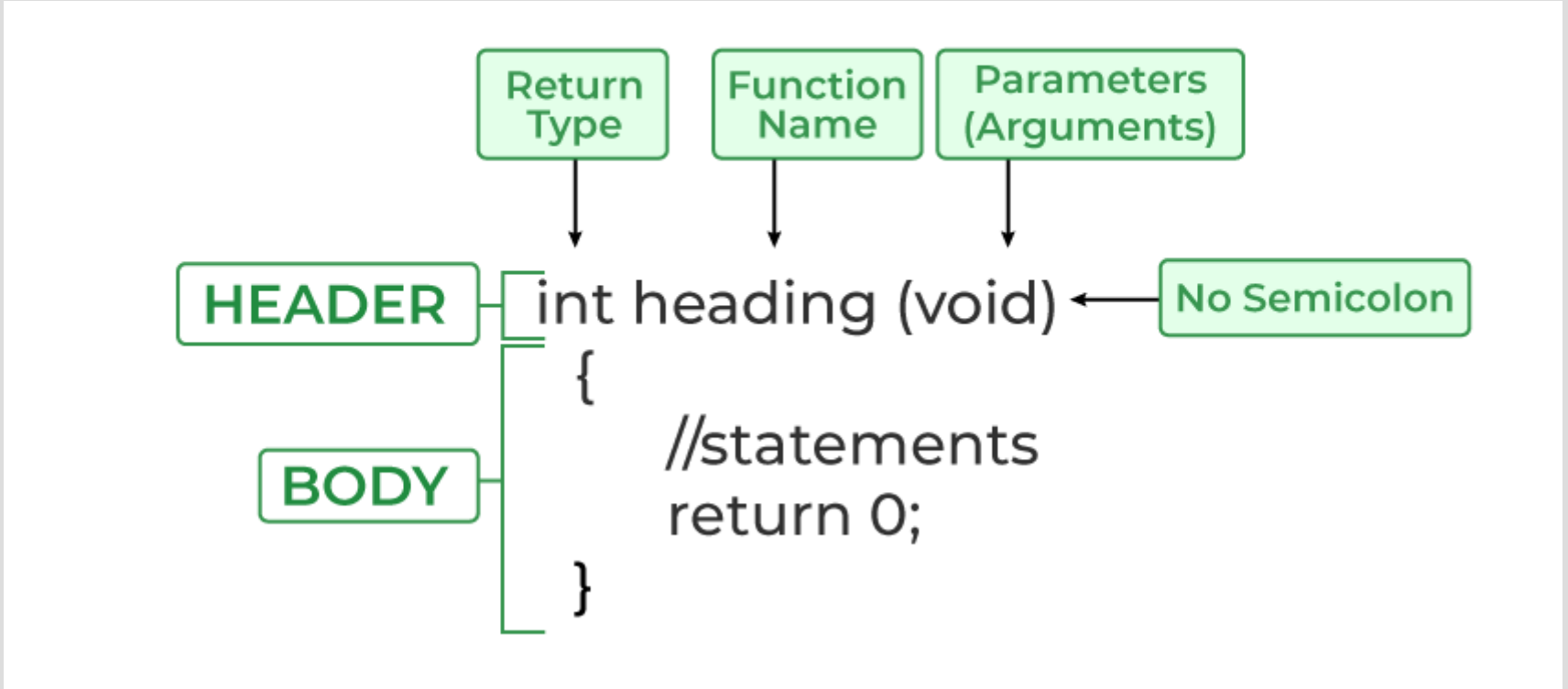
- Deklaration:



Bildquelle: https://www.geeksforgeeks.org/c-functions/ 20.09.2023

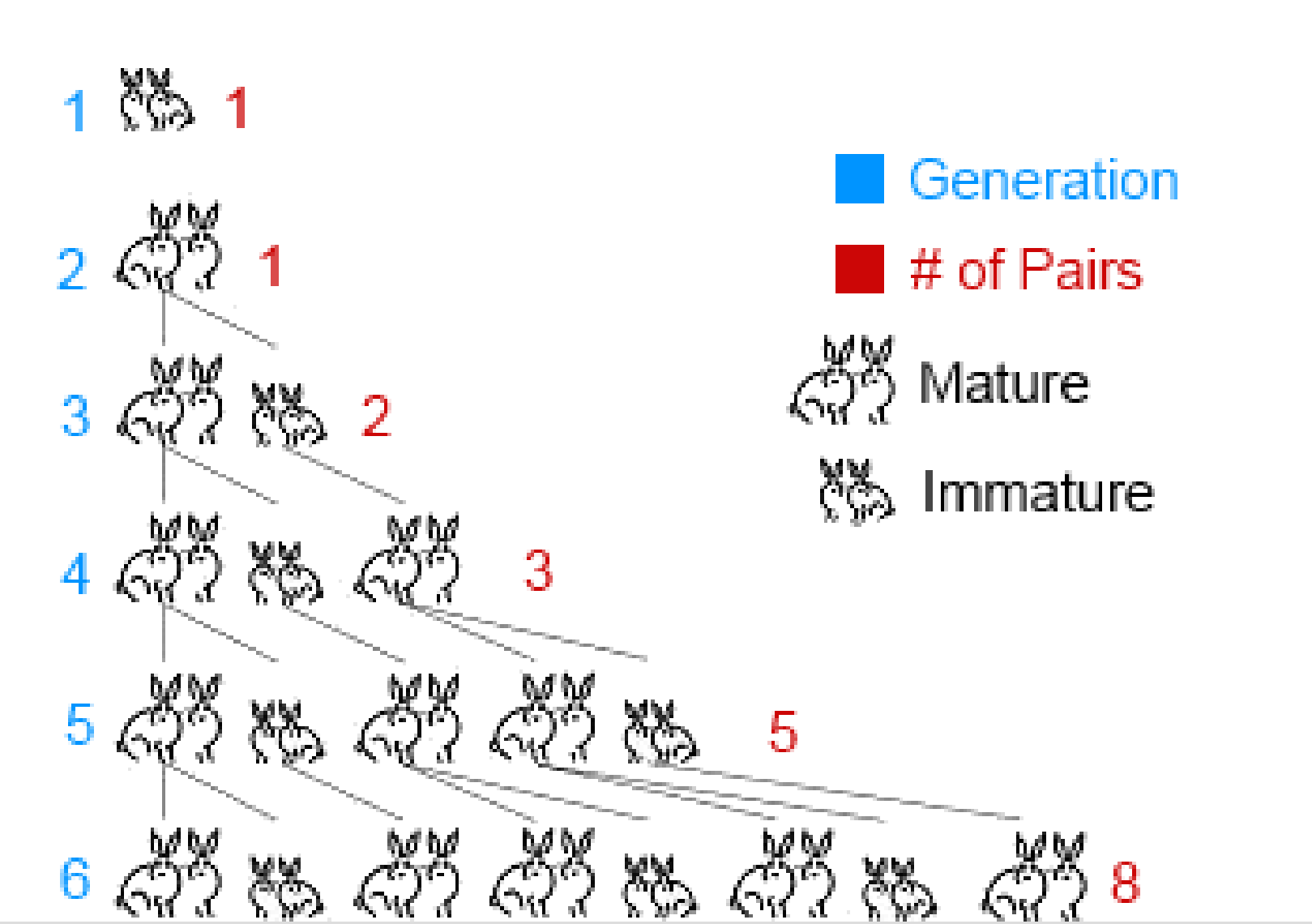
FUNKTIONEN IN C

- Definition:



Bildquelle: https://www.geeksforgeeks.org/c-functions/ 20.09.2023

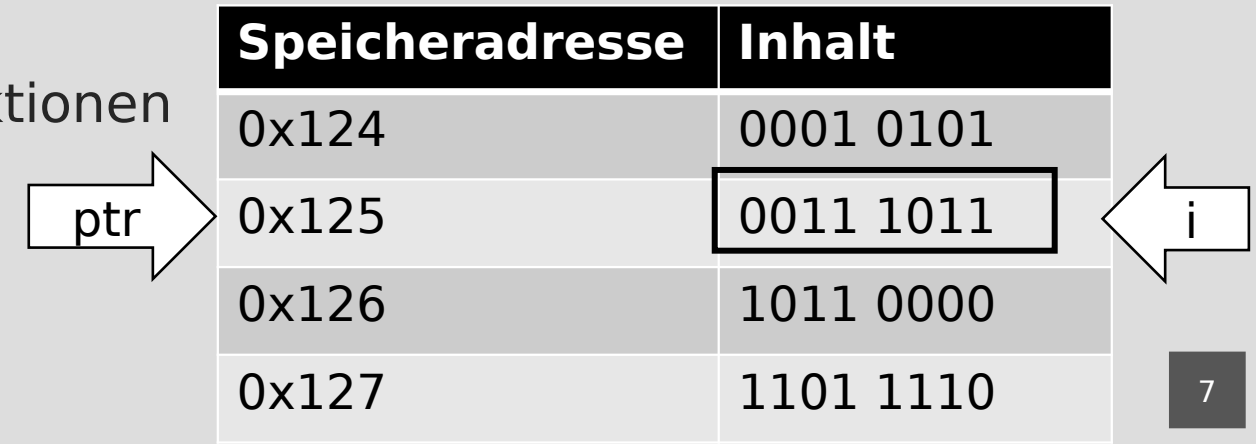
BEISPIEL: FIBONACCI-FOLGE



POINTERS: DEFINITION

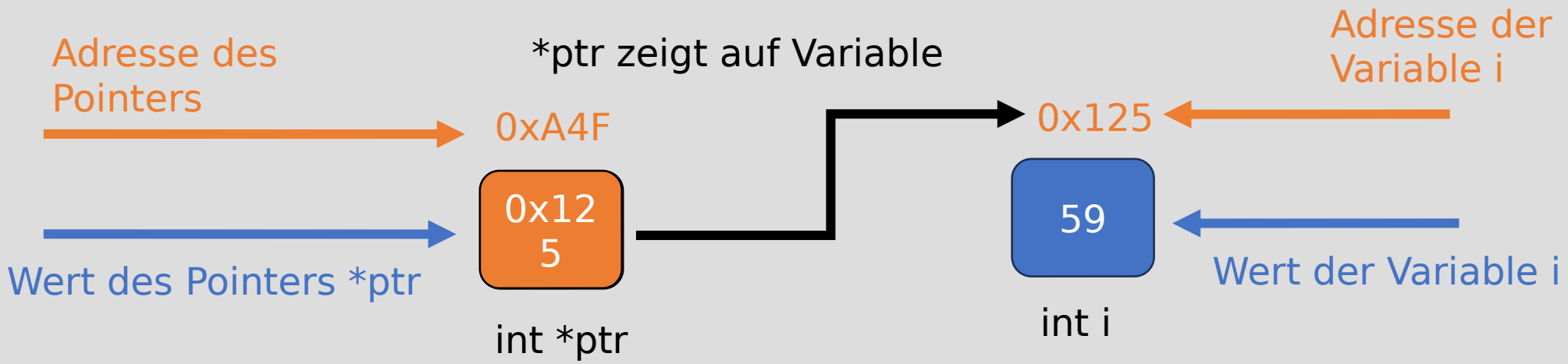
Definition

- Ein Pointer ist eine Variable, in der die Speicheradresse eines Objektes (z.B. einer Variablen) gespeichert ist.
- Dadurch kann man z.B. sehr einfach auf die Speicheradresse zugreifen und die Speicherverwaltung manipulieren.
- Anwendungen:
  - Dynamische Datenstrukturen (zB. Linked Lists, Trees, …)
  - Zugriff auf Arrays
  - Übergabe von Parametern an Funktionen
  - Dynamische Speicherallokation



Speicheradresse	Inhalt
0x124	0001 0101
0x125	0011 1011
0x126	1011 0000
0x127	1101 1110

POINTERS: THEORIE



POINTER DEKLARIEREN

- * Operator für die Deklaration, & Operator für die Initialisierung
  - int * ptr; // Deklaration eines Integer-Pointers
  - int num = 42
  - int *ptr = # // Initialisierung des Pointers mit der Adresse von 'num‘
- Zugriff auf den Wert mit dem **Dereferenzierungsoperator** *
  - int value = *ptr; // 'value' enthält jetzt den Wert von 'num' (42)

POINTER BEISPIEL

```c
// C program to illustrate Pointers
#include <stdio.h>

void pointertest()
{
        int var = 10;
        // declare pointer variable
        int* ptr;
        // note that data type of ptr and var must be same
        ptr = &var;
        // assign the address of a variable to a pointer
        printf("Value at ptr = %p \n", ptr);
        printf("Value at var = %d \n", var);
        printf("Value at *ptr = %d \n", *ptr);
}
int main()
{       pointertest();
        return 0;
}
```

Output

```
Value at ptr = 0x7fff1038675c
Value at var = 10
Value at *ptr = 10
```

ARTEN VON POINTERN

- Viele verschiedene Möglichkeiten, z.B.
  - Integer: `int *ptr;` // Analog: float, char, …
  - Array: `char *ptr =array_name;` // Praktisch für Datenstrukturen
  - Funktionen: `int (*ptr)(int, char);` // Für eine Funktion int foo(int,char)
  - Double Pointers: `datatype **ptr_name;` // Zeigt auf einen Pointer
  - NULL: `datatype *ptr_name = NULL;` // Nicht genutzte Pointer auf „null“

POINTER ARITHMETIC

- Increment, Decrement, Add, Subtract, Compare, Assign

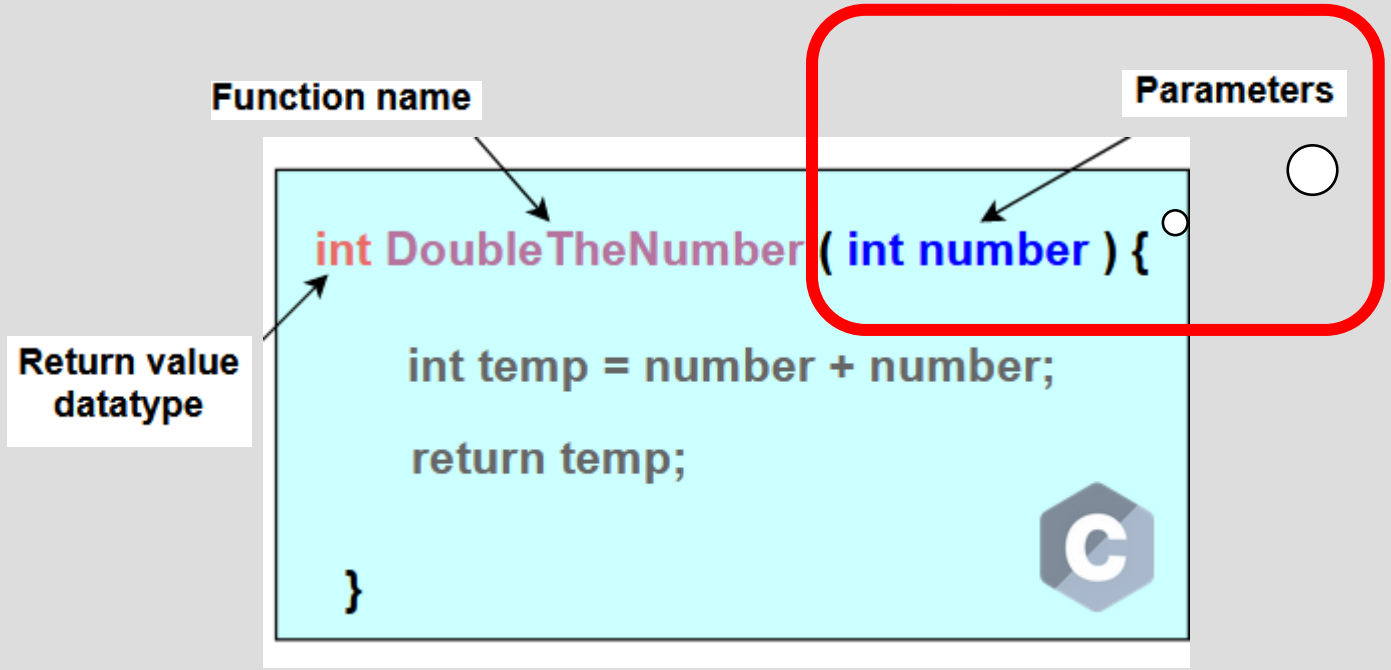
```c
#include <stdio.h>
int main()
{
    // Declare an array
    int v[3] = { 10, 100, 200 };
    // Declare pointer variable
    int* ptr;
    // Assign the address of v[0] to ptr
    ptr = v;
    for (int i = 0; i < 3; i++) {
        // print value at address which is stored in ptr
        printf("Value of *ptr = %d\n", *ptr);
        // print value of ptr
        printf("Value of ptr = %p\n\n", ptr);
        // Increment pointer ptr by 1
        ptr++;
    }
    return 0;
}
```

Output

```
Value of *ptr = 10
Value of ptr = 0x7ffe8ba7ec50

Value of *ptr = 100
Value of ptr = 0x7ffe8ba7ec54

Value of *ptr = 200
Value of ptr = 0x7ffe8ba7ec58
```

CALL BY VALUE VS. CALL BY REFERENCE

- Übergabe von Parametern in Funktionen
  - Als „Kopie“ der Variable
  - Als Referenz auf die Variable (Pointer)



Wie übergebe ich den Parameter an die Funktion?

CALL BY VALUE VS. CALL BY REFERENCE

- **Call by Value**
  - **Kopien der Argumente** werden übergeben
  - Änderungen der Argumente in *modifyValue* haben KEINEN Einfluss auf den Wert von *value* in der main() Funktion
  - Standard bei der Übergabe einfacher Datentypen (z.B. Integer)

```
void modifyValue(int x) {
    x = x + 10;
}

int main() {
    int value = 5;
    modifyValue(value);
    // 'value' bleibt unverändert
    return 0;
}
```

Eine Kopie wird übergeben

CALL BY VALUE VS. CALL BY REFERENCE

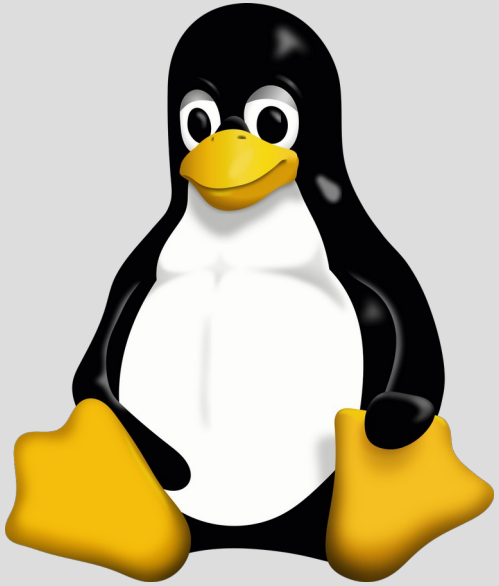
- **Call by Reference**
  - **Zeiger (Referenzen)** auf die ursprünglichen Argumente werden übergeben
  - Die Funktion arbeitet mit den **originalen Variablen**
  - Verwendung: Komplexere Datenstrukturen (z.B. Arrays)

```
void modifyReference(int *x) {
    *x = *x + 10;
}
int main() {
    int value = 5;
    modifyReference(&value);
    // 'value' wird auf 15 geändert
    return 0;
}
```

Eine Referenz auf *value* wird übergeben

FILE IO: ÜBERSICHT

- **Warum will ich mit Files arbeiten?**
  - Permanente Speicherung von Daten in Programmen
  - Verarbeitung großer Datenmengen
  - hardwarenahe Programmierung (Sensordaten)
  - Linux (… everything is a file)

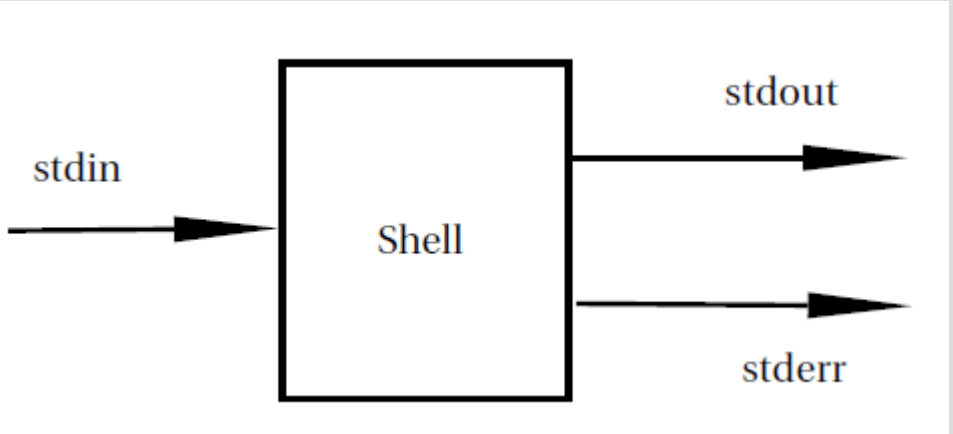


- Zwei Möglichkeiten
  - High-Level über Standard-ANSI-C Bibliothek
  - Zugriff auf I/O Funktionen des Kernels (komplexer)

- High-Level: Bearbeitung über **Dateizeiger** – FILE*
  - Aufruf: FILE *file; // Dateizeiger

FILE IO: FILE-ZEIGER (STREAM)

- Linux: Drei Standard-Streams
  - FILE * stdin (default auf Tastatur)
  - FILE* stdout (default auf Monitor)
  - FILE* stderr (default auf Monitor)



- Streams können **umgeleitet** werden
  - Input aus einem .txt File
  - Output in ein .txt File

FILE IO: OPERATIONEN

- **Filepointer erstellen**
  - FILE *file;
- **File öffnen**
  - file = fopen("output.txt", "w"); // Öffnet myFile.txt im Modus „w“ (write)
  - Filename kann auch den Dateipfad enthalten
  - Viele Modi möglich, z.B. read, write, append, … (siehe Skript)
  - NULL, falls etwas nicht funktioniert – kann überprüft werden

```
if (file == NULL) {
    perror("Die Datei konnte nicht geöffnet werden");
    return 1;
}
```

FILE IO: OPERATIONEN

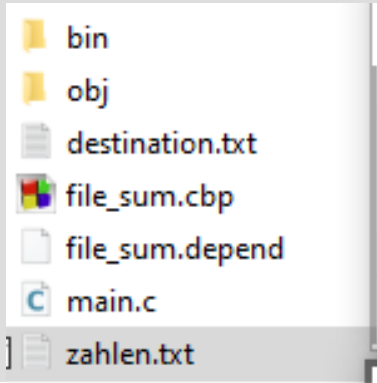
- **Lesen/Schreiben mit Files**
  - fgets() und fputs() ermöglichen zeilenweises Lesen/Schreiben (File Get String)
  - fgetc() und fputc() lesen einzelne Zeichen aus einer Datei (File Get Character)
  - Beispiel: Inhalt eines source-Files in ein destination-File kopieren:

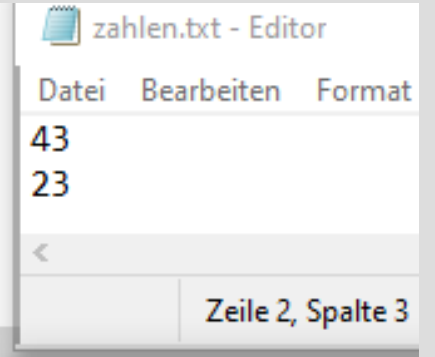
```
char buffer[100];
// Daten aus der Quelldatei lesen und in die Zieldatei schreiben
while (fgets(buffer, sizeof(buffer), sourceFile) != NULL) {
    fputs(buffer, destinationFile);
    printf("%s", buffer);  // Daten auf dem Bildschirm ausgeben
}
```

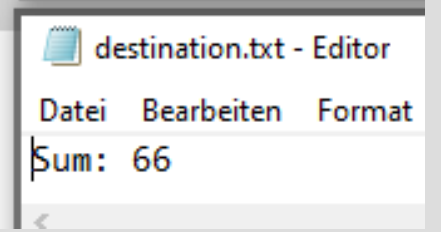
- **File schließen**
  - fclose(file);

AUFGABE

- Erweitere das Programm so, dass das Ergebnis (sum) in ein File „destination.txt“ gespeichert wird.








```
char buffer[100];
// Daten aus der Quelldatei lesen und in die Zieldatei sch
while (fgets(buffer, sizeof(buffer), sourceFile) != NULL)
    fputs(buffer, destinationFile);
    printf("%s", buffer);  // Daten auf dem Bildschirm ausgeben
}
```

Tipp: Verwende die C-Library-Funktion sprintf(…) für den cast von int auf String! https://www.geeksforgeeks.org/sprintf-in-c/

BEISPIEL FILE IO

```c
#include <stdio.h>

int main() {
    // Dateien öffnen
    FILE *sourceFile = fopen("source.txt", "r");
    FILE *destinationFile = fopen("destination.txt", "w");

    if (sourceFile == NULL || destinationFile == NULL) {
        perror("Fehler beim Öffnen der Dateien");
        return 1;
    }

    char buffer[100];
    // Daten aus der Quelldatei lesen und in die Zieldatei schreiben
    while (fgets(buffer, sizeof(buffer), sourceFile) != NULL) {
        fputs(buffer, destinationFile);
        printf("%s", buffer);  // Daten auf dem Bildschirm ausgeben
    }

    // Dateien schließen
    fclose(sourceFile);
    fclose(destinationFile);

    return 0;
}
```