# Funktionen in C

# Lernziele

- Sie können Funktionen definieren, implementieren und aufrufen.
- Sie kennen den Gültigkeitsbereich von Variablen.
- Sie verstehen das Prinzip der Parameterübergabe Call By Value.

# Was ist eine Funktion?

Eine Funktion ist eine Verarbeitungseinheit, die übergebene Daten verarbeitet und den berechneten Funktionswert als Ergebnis zurückgibt:

- ✓ fasst ein Stück Code als eine Einheit zusammen.
- ✓ Hat einen Namen
- ✓ Kann Daten via Parameter übergeben
- ✓ Kann ein Ergebnis zurückliefern

Beispiel:

Aus der Mathematik:

$f(x) = x^2$

Funktion in C:

```
double f(double x){
   return x*x;
}
```

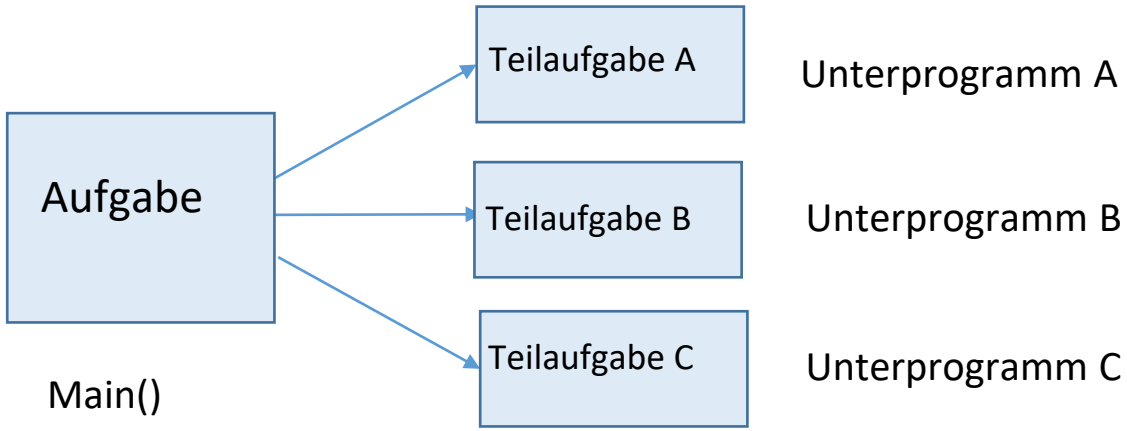
# Wozu brauchen wir Funktionen?

- Fundamentaler Baustein einer prozeduralen Programmiersprache

  C Programm: main() und weitere Funktionen

- Strukturiertes Vorgehen:

- ✓ Strukturiert, besser lesbar
- ✓ Wiederverwendbarkeit (DRY)



# Funktion ohne Parameter, kein Return-Wert

Definition:

- Funktionskopf: void Funktionsname()
- Funktionskörper: Mit *return* wird die Funktion beendet.

```
void Hallo() {
    printf("In der Funktion\n");
    return;
}
```

Aufruf:

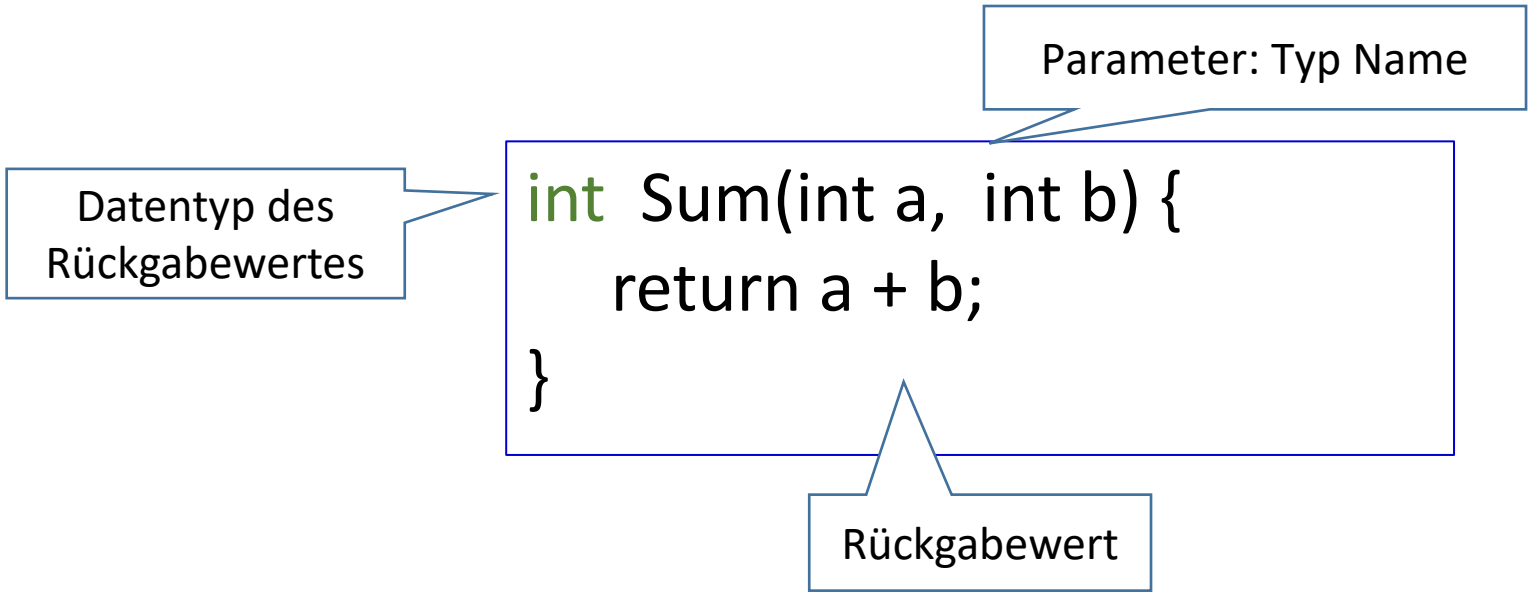
```
int main()
{
        printf("Vor der Funktion\n");
        Hallo();
        printf("Nach der Funktion\n");
        return 0;
}

        //fktNoParam.cpp
```

# Funktion mit Parameter und Return-Wert

## Definition

- Funktionskopf : Returntyp Funktionsname (Parameterliste)

  Man kann einer Funktion Werte übergeben, sogenannte Parameter. Für jeden Parameter muss ein Datentyp festgelegt werden.
- Funktionskörper: *return* beendet die Funktion und gibt den Rückgabewert zurück.

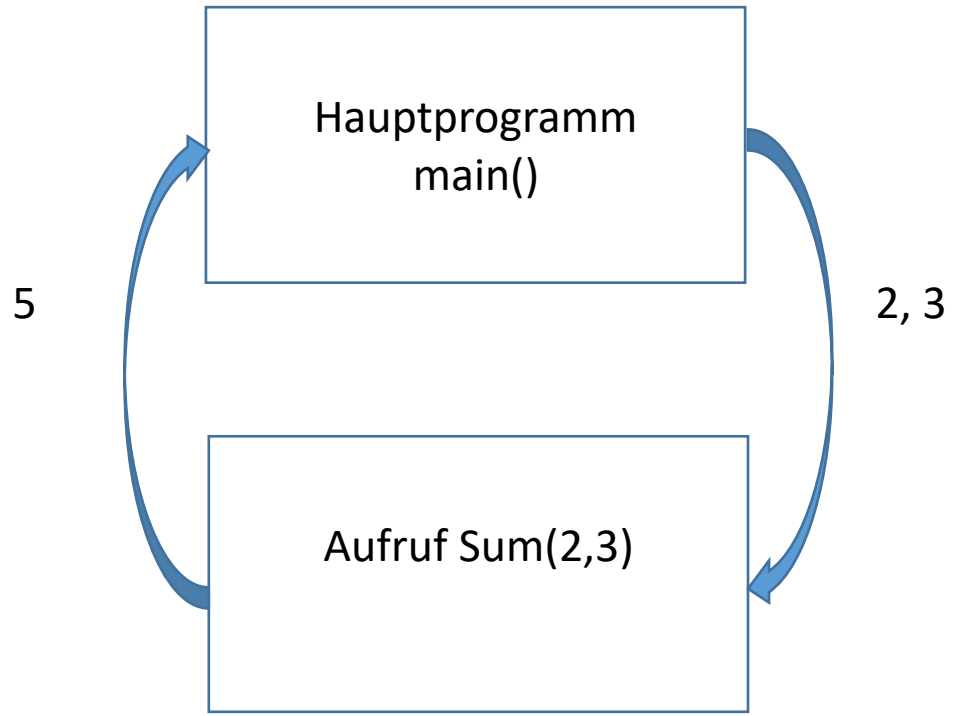


# Funktion mit Parameter und Return-Wert

## Aufruf

Die Argumente müssen mit der richtigen Reihenfolge und auch mit dem richtigen Typen des formalen Parameters übereinstimmen.

```
int Sum(int a, int b) {
    int c = a + b;
    return c;
}

int main()
{
    int summe;

    summe = Sum(2, 3);

    printf("Die Summe ist %d\n", summe);
    return 0;
}                                   //sum.cpp
```



# Namensgebung für Funktionen

Regeln der Namensgebung für Funktionen:

- ✓ Der Name sagt aus, was die Funktion leistet. Es beginnt häufig mit einem Verb.
- ✓ Alle Bezeichner in ANSI-C sind Case-Sensitive. sum() und Sum() sind verschiedene Funktion.
- ✓ GIBZ-Namensgebung: Der Name einer eigen erstellten Funktion soll immer mit einem Grossbuchstaben beginnen. Es wird die sogenannte UpperCamelCase-Notation verwendet.

Beispiele:

- BerechneSumme()
- WriteNameToConsole()
- BerechneMaximalWert()

# Funktionsdeklaration

```
// Funktionsdeklaration (Prototypen)
int CalculateFaculty(int number);

int main()  {
    int number = 4; //
    printf("The faculty of the number %d is %d.", number, CalculateFaculty(number));
    return 0;
}
//Funktionsdefinition
int CalculateFaculty(int value) {
            int faculty = 1;
            while (value > 1)  {
                faculty *= value;
                value--;
            }
            return faculty;
}
```

**Guter Programmierstil:**
Funktionen sollten am Anfang des Programms (zwischen #include-Anweisungen und main()) mittels Prototypen deklariert werden.

# Lokale und globale Variablen

## Globale Variablen

- Alle Variablen, die ausserhalb einer Funktion definiert werden, werden als "globale Variablen" bezeichnet.
- Diese sind während der ganzen Programm Ausführung verfügbar und können von allen Funktionen verwendet werden.
- Alle globalen Variablen werden nach ANSI C mit dem Wert 0 initialisiert.

```
/* Globale Variable, im ganzen Programm gültig */
float summe;

int main() {
  summe = 3;
}

void Summe_bilden {
  summe = 4;
}
```

# Lokale und globale Variablen

## Lokale Variablen

- Lokale Variablen sind nur im Bereich gültig,  in dem sie deklariert wurden (main, Funktionen). Lokale Variablen werden beim Aufruf der Funktion erzeugt und am Ende dieser zerstört.
- Lokale Variablen werden nicht automatisch initialisiert.

```
int doppel (int n);

int main(){
  int m = 2;
  int res;

  res = doppel(m);     // Aufruf der Funktion
  //Hier haben Sie keinen Zugriff auf lokale Variable res.
  return 0;
}

int doppel (int n) {
   int res = 2 * n;    //Die lokale Variable res wird hier erzeugt.
   return res;         //res wird ungültig und verliert ihren Inhalt.
}
```

# Lokale und globale Variablen

## Statische Lokale Variablen

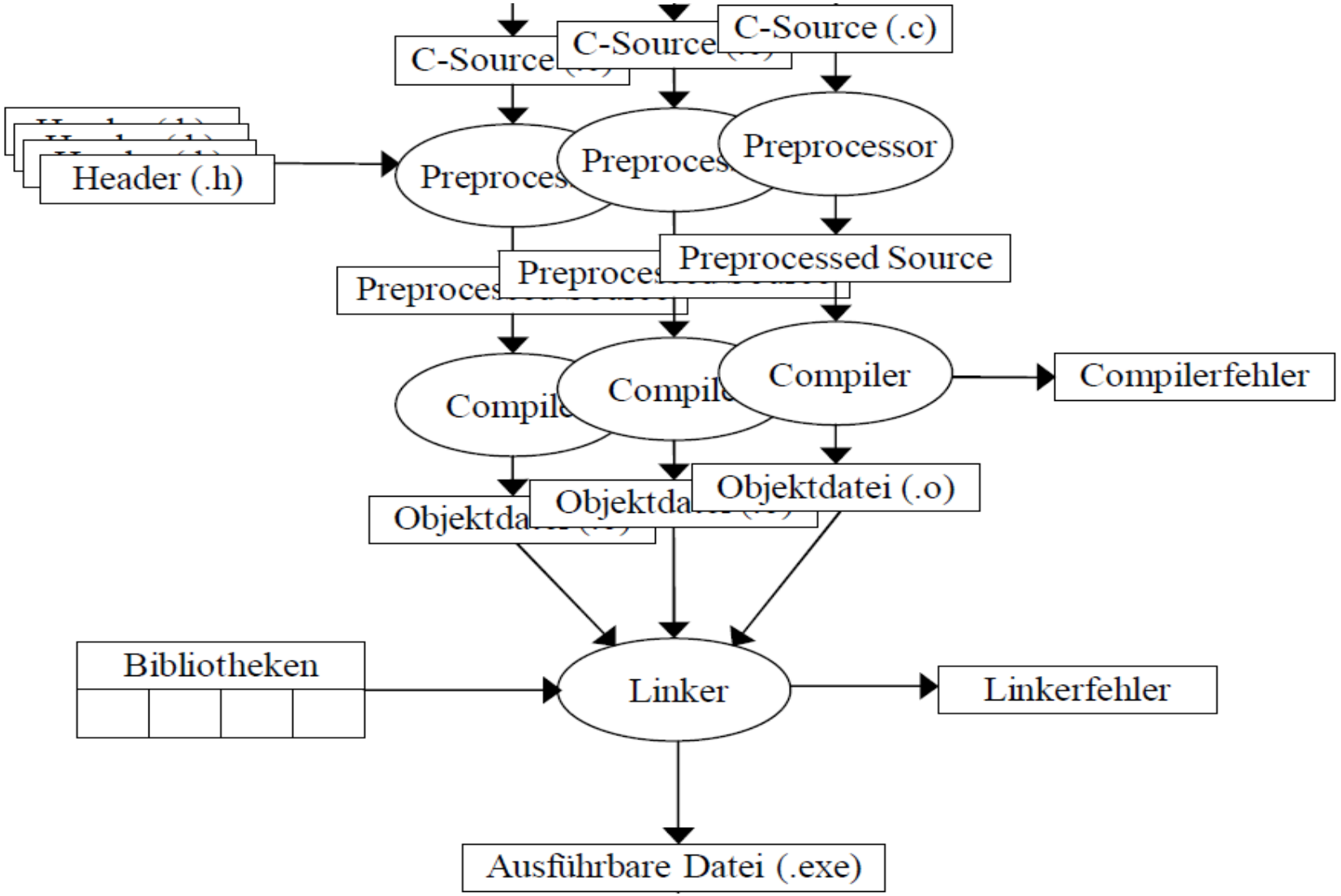
- Wenn man eine lokale Variable als *static* definiert, ist sie zwar nach wie vor nur innerhalb ihres Blocks zugreifbar, aber sie behält ihren Wert bis zum nächsten Funktionsaufruf.

- Die Initialisierung von static-Variablen wird nur einmal zu Beginn des Programms ausgeführt.

```
int Counter(void);

int main()
{
   for (int  i= 0; i <10; i++ ){
        printf("Aktueller Wert von count: %d\n", Counter());
   }
}

int Counter(void)
{
   static int count = 0; /* Statische lokale Variable */
   count++;
   return count;
}
```

Counter.cpp

# Aufbau eines C-Projektes



# Bibliotheken einbinden

Damit Bibliotheksfunktionen verwendet werden können, muss die Schnittstelle (Headerdatei) dieser Funktion bekannt gemacht werden.

Der Präprozessor-Befehl zum Einbinden einer Bibliothek:

```
#include <stdio.h> // Einbindung der Standardbibliothek
#include <math.h>
```

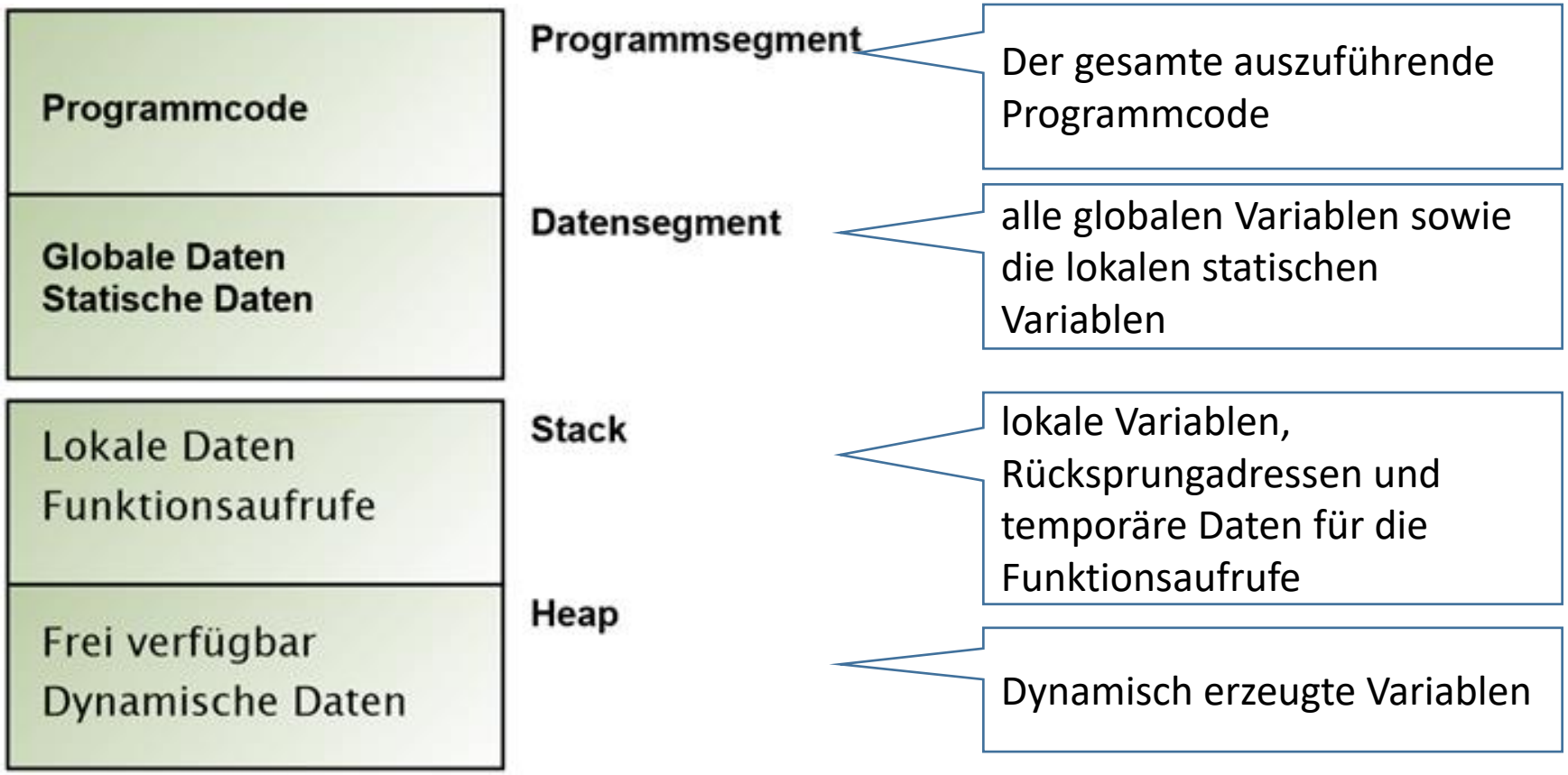
Beispiel: div_Fkt.cpp

# Alternative zu Funktionen: die Macros

```
#include <stdio.h>
#define PI 3.1415

int main()
{
   float radius, area;
   printf("Enter the radius: ");
   scanf_s("%f", &radius);

   // Notice, the use of PI
   area = PI*radius*radius;

   printf("Area=%.2f",area);
   return 0;
}
```

```
#include <stdio.h>
#define PI 3.1415
#define circleArea(r) (PI*(r)*(r))

int main()
{
   float radius, area;

   printf("Enter the radius: ");
   scanf_s("%f", &radius);
   area = circleArea(radius);
   printf("Area = %.2f", area);

   return 0;
}
```

# Speichermodell eines Prozesses



# Parameterübergabe

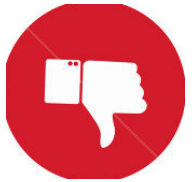
Die Übergabe von Daten vom Hauptprogramm an die Funktion zur Weiterverarbeitung sollte immer über Parameter erfolgen.

Wertübergabe über Parameter

```
#include <stdio.h>

int Zaehlen(int c) {
    return c+1;
}

int main() {
        int counter = 0;
        counter=Zaehlen(counter);
        return 0;
}
```

Wertübergabe über globale Variablen

```
#include <stdio.h>

int counter;
void Zaehlen() {
      counter += 1;
}

int main() {
        counter = 0;
        Zaehlen();
        return 0;
}
```



# Übergabe der Parameter mittels "Call by Value"

## Call By Value:

- ✓ Nur eine Kopie des Argumentes wird vom Aufrufer an die Funktion übergeben.
- ✓ Eine Veränderung des Parameters in der Funktion hat keine Auswirkung auf den Wert des Argumentes.

```
int Halbieren(int a);
int main () {
  int b = 4, z;
 //b=?
  z = Halbieren(b);
 //b=? z=?
  return 0;
}

int Halbieren(int a) {
    //a = ?
    a = a / 2;
    //a = ?
   return a;
}  // Halbieren.cpp
```

a erhält eine Kopie von b

=> Änderung von Parameter a innerhalb der Funktion Halbieren() hat keine Auswirkung auf Argument b in main()! Warum?

# Übergabe der Parameter mittels "Call by Value"

Kontrollfrage:

Funktioniert folgende Lösung richtig?

```
void GetMax(int z1, int z2);

int main () {
  int i1=1, i2=5;
  GetMax(i1, i2);
  //i1=  i2=
  return 0;
}

void GetMax (int z1, int z2) {
     z1= (z1 > z2) ? z1 : z2;          //? : Conditional Operator
    return;
}
```