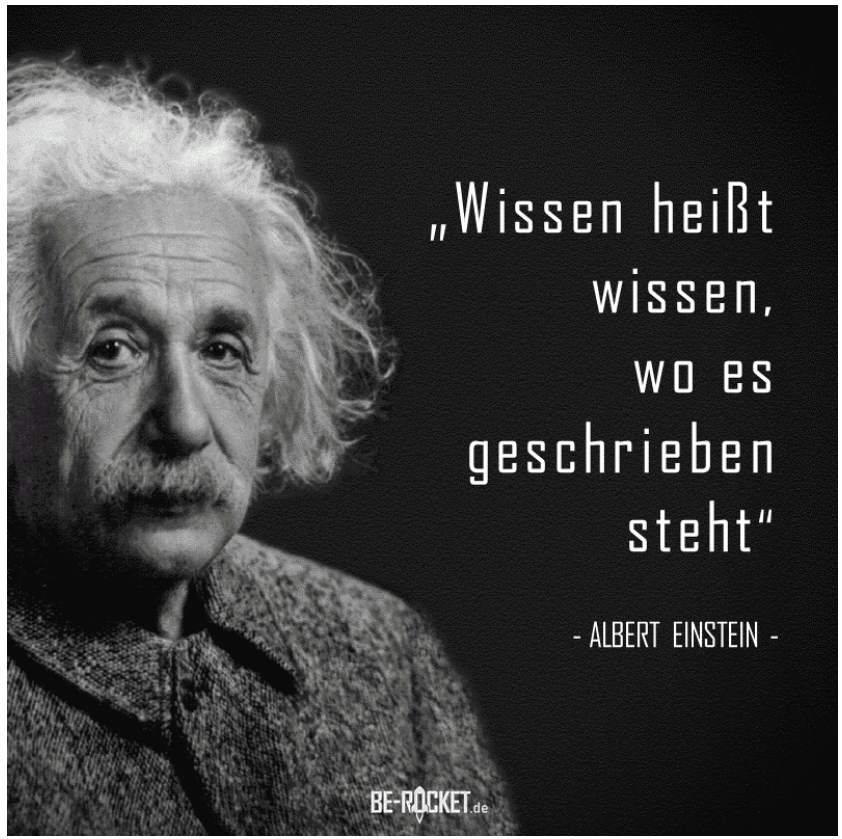
Hardwarenahe Programmierung I

U. Kaiser, R. Kaiser, M. Stöttinger, S. Reith

(HTTP: http://www.cs.hs-rm.de/~kaiser
EMail: robert.kaiser@hs-rm.de)

Wintersemester 2021/2022

10. Zeiger und Adressen

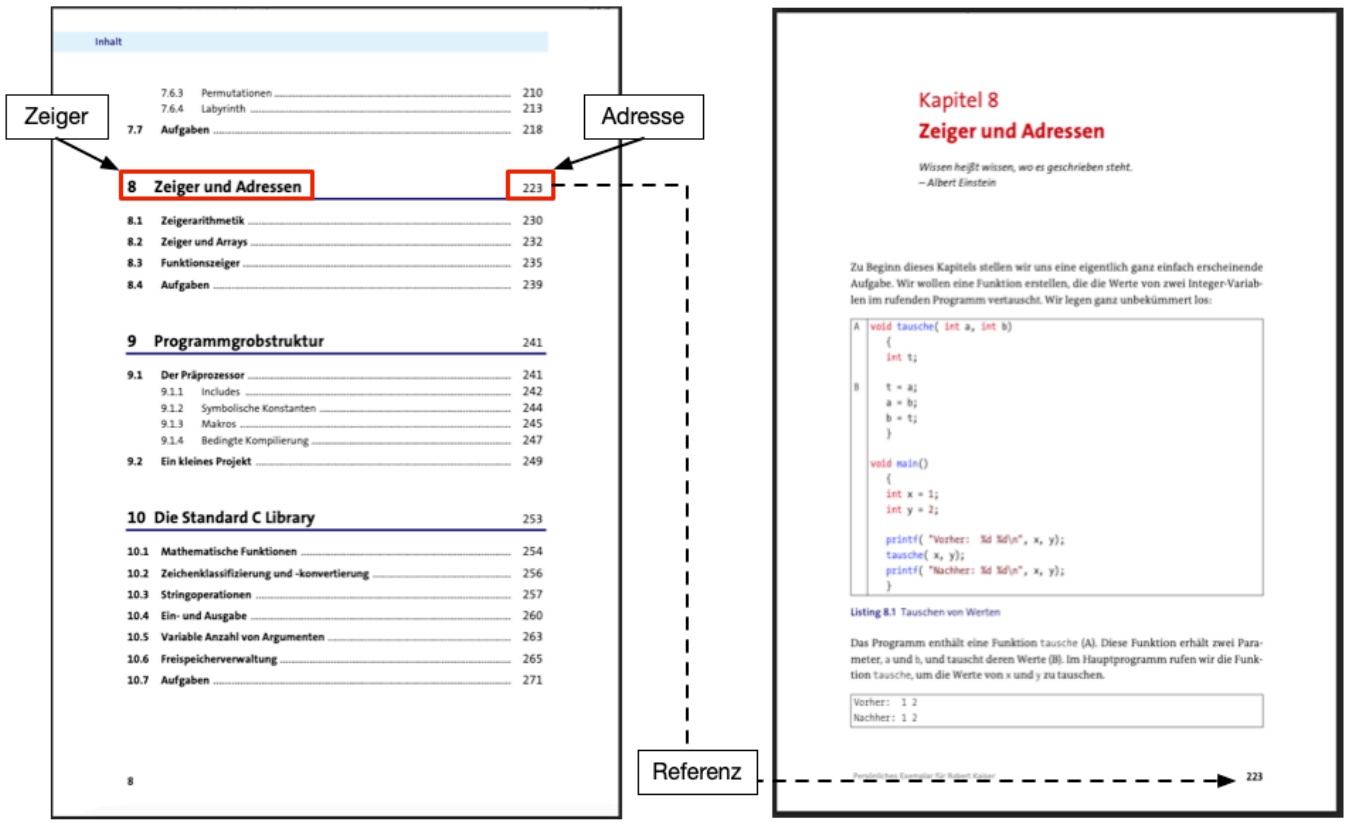


https://be-rocket.de/wp-content/uploads/2018/10/Zitate-Einstein-Wissen.png

Das Prinzip der Indirektion

Hochschule RheinMain

Ein Zeiger enthält nicht die eigentlichen Daten sondern eine Adresse, über die auf die eigentlichen Daten zugegriffen werden kann. Man nennt dies auch eine Referenz.



Motivation

- Erstelle eine Funktion, die Variablenwerte des Hauptprogramms tauscht.

```
void tausche( int a, int b)
{
    int t;

    t = a;
    a = b;
    b = t;
}

void main()
{
    int x = 1;
    int y = 2;

    printf( "Vorher: \%d \%d\n", x, y);
    tausche( x, y);
    printf( "Nachher: \%d \%d\n", x, y);
}
```

- Zugehöriger Ausgabe:

```
Vorher: 1 2
Nachher: 1 2
```

- Fazit: Die Funktion ist wirkungslos, da sie nur auf Kopien der Variablen x und y arbeitet. Die Originale sind nicht betroffen.

Motivation

- Erstelle eine Funktion, die Variablenwerte des Hauptprogramms tauscht.

```
void tausche( int a, int b)
{
    int t;

    t = a;
    a = b;
    b = t;
}

void main()
{
    int x = 1;
    int y = 2;

    printf( "Vorher: \%d \%d\n", x, y);
    tausche( x, y);
    printf( "Nachher: \%d \%d\n", x, y);
}
```

Diese Funktion erhält zwei Parameter, a und b, und tauscht deren Werte.

- Zugehöriger Ausgabe:

```
Vorher: 1 2
Nachher: 1 2
```

- Fazit: Die Funktion ist wirkungslos, da sie nur auf Kopien der Variablen x und y arbeitet. Die Originale sind nicht betroffen.

Motivation

Hochschule RheinMain

- Erstelle eine Funktion, die Variablenwerte des Hauptprogramms tauscht.

```
void tausche( int a, int b)
{
    int t;

    t = a;
    a = b;
    b = t;
}

void main()
{
    int x = 1;
    int y = 2;

    printf( "Vorher: \%d \%d\n", x, y);
    tausche( x, y);
    printf( "Nachher: \%d \%d\n", x, y);
}
```

Diese Funktion erhält zwei Parameter, a und b, und tauscht deren Werte.

Wir rufen die Funktion *tausche*, um die Werte von x und y zu tauschen

- Zugehöriger Ausgabe:

```
Vorher: 1 2
Nachher: 1 2
```

- Fazit: Die Funktion ist wirkungslos, da sie nur auf Kopien der Variablen x und y arbeitet. Die Originale sind nicht betroffen.

Motivation

Hochschule RheinMain

- Erstelle eine Funktion, die Variablenwerte des Hauptprogramms tauscht.

```
void tausche( int a, int b)
{
    int t;

    t = a;
    a = b;
    b = t;
}

void main()
{
    int x = 1;
    int y = 2;

    printf( "Vorher: \%d \%d\n", x, y);
    tausche( x, y);
    printf( "Nachher: \%d \%d\n", x, y);
}
```

Diese Funktion erhält zwei Parameter, a und b, und tauscht deren Werte.

Wir rufen die Funktion *tausche*, um die Werte von x und y zu tauschen

- Zugehöriger Ausgabe:

```
Vorher: 1 2
Nachher: 1 2
```

- Fazit: Die Funktion ist wirkungslos, da sie nur auf Kopien der Variablen x und y arbeitet. Die Originale sind nicht betroffen.

Motivation

- Erstelle eine Funktion, die Variablenwerte des Hauptprogramms tauscht.

```
void tausche( int a, int b)
{
    int t;

    t = a;
    a = b;
    b = t;
}

void main()
{
    int x = 1;
    int y = 2;

    printf( "Vorher: \%d \%d\n", x, y);
    tausche( x, y);
    printf( "Nachher: \%d \%d\n", x, y);
}
```

Diese Funktion erhält zwei Parameter, a und b, und tauscht deren Werte.

Wir rufen die Funktion *tausche*, um die Werte von x und y zu tauschen

- Zugehöriger Ausgabe:

```
Vorher: 1 2
Nachher: 1 2
```

- Fazit: Die Funktion ist wirkungslos, da sie nur auf Kopien der Variablen x und y arbeitet. Die Originale sind nicht betroffen.

Adressen

- Zur Lösung des Tauschproblems übergeben wir der Funktion `tausche` die *Adressen*, an denen die zu tauschenden Werte im Speicher stehen. Die Funktion kann dann über die Adressen direkt auf die Originaldaten zugreifen.

> Die (Speicher-)Adresse einer Variablen erhält man, indem man dem Variablenamen den **Adressoperator** „&“ voranstellt.

- Mit Verwendung des Adressoperators ergibt sich dann der folgende Funktionsaufruf:

```
void main()
{
  int x = 1;
  int y = 2;

  printf( "Vorher: %d %d\n", x, y);
  tausche( &x, &y);
  printf( "Nachher: %d %d\n", x, y);
}
```

- Diese Art der Parameterübergabe kennen Sie bereits von `scanf`. Dort kam es ja auch darauf an, durch das Unterprogramm Variablenwerte im Hauptprogramm zu verändern.
- Die Adresse bezeichnet den „Ort“ eines Datums im Speicher. Über sie erhalten wir Zugriff auf das dort hinterlegte Datum.
- Die Schnittstelle der Funktion muss jetzt allerdings geändert werden, da keine `int`-Werte mehr übergeben werden sondern Adressen an denen im Speicher `int`-Werte stehen.

Adressen

- Zur Lösung des Tauschproblems übergeben wir der Funktion `tausche` die *Adressen*, an denen die zu tauschenden Werte im Speicher stehen. Die Funktion kann dann über die Adressen direkt auf die Originaldaten zugreifen.

> Die (Speicher-)Adresse einer Variablen erhält man, indem man dem Variablenamen den **Adressoperator** „&“ voranstellt.

- Mit Verwendung des Adressoperators ergibt sich dann der folgende Funktionsaufruf:

```
void main()
{
  int x = 1;
  int y = 2;

  printf( "Vorher: %d %d\n", x, y);
  tausche( &x, &y);
  printf( "Nachher: %d %d\n", x, y);
}
```

Hier übergeben wir die Adresse der Variablen `x` und die Adresse der Variablen `y`.

- Diese Art der Parameterübergabe kennen Sie bereits von `scanf`. Dort kam es ja auch darauf an, durch das Unterprogramm Variablenwerte im Hauptprogramm zu verändern.
- Die Adresse bezeichnet den „Ort“ eines Datums im Speicher. Über sie erhalten wir Zugriff auf das dort hinterlegte Datum.
- Die Schnittstelle der Funktion muss jetzt allerdings geändert werden, da keine `int`-Werte mehr übergeben werden sondern Adressen an denen im Speicher `int`-Werte stehen.

Zeiger

Hochschule RheinMain

- Eine Variable, in der die Adresse einer anderen Variablen gespeichert ist, nennen wir eine **Zeigervariable** oder kurz **Zeiger** bzw. **Pointer**.
- Die Variable, deren Adresse im Zeiger gespeichert ist, bezeichnen wir als die durch den Zeiger **referenzierte** oder **adressierte Variable**.
- Über einen Zeiger kann auf die Daten der referenzierten Variablen zugegriffen werden. Wir nennen dies **Indirektzugriff** oder auch **Dereferenzierung**. Zum Zugriff auf die referenzierte Variable verwendet man den Operator „*“ (**Dereferenzierungsoperator**).
- Ist p ein Zeiger, so ist *p die referenzierte Variable.

```
int x;
float y;

int *pi;
float *pf;

pi = &x;
pf = &y;

*pi = 1234;
*pf = *pi + 0.5;

printf( "x: %d\n", x);
printf( "y: %f\n", y);
```

```
x: 1234
y: 1234.500000
```

- Der Dereferenzierungsoperator („*“) ist das Gegenstück zum Adressoperator („&“).
- Mit dem Adressoperator kommen wir von einer Variablen zu ihrer Adresse, mit dem Dereferenzierungsoperator kommen wir über die Adresse zur Variablen.

Zeiger

Hochschule RheinMain

- Eine Variable, in der die Adresse einer anderen Variablen gespeichert ist, nennen wir eine **Zeigervariable** oder kurz **Zeiger** bzw. **Pointer**.
- Die Variable, deren Adresse im Zeiger gespeichert ist, bezeichnen wir als die durch den Zeiger **referenzierte** oder **adressierte Variable**.
- Über einen Zeiger kann auf die Daten der referenzierten Variablen zugegriffen werden. Wir nennen dies **Indirektzugriff** oder auch **Dereferenzierung**. Zum Zugriff auf die referenzierte Variable verwendet man den Operator „*“ (**Dereferenzierungsoperator**).
- Ist p ein Zeiger, so ist *p die referenzierte Variable.

Zwei „gewöhnliche“ Variablen

```
int x;
float y;

int *pi;
float *pf;

pi = &x;
pf = &y;

*pi = 1234;
*pf = *pi + 0.5;

printf( "x: %d\n", x);
printf( "y: %f\n", y);
```

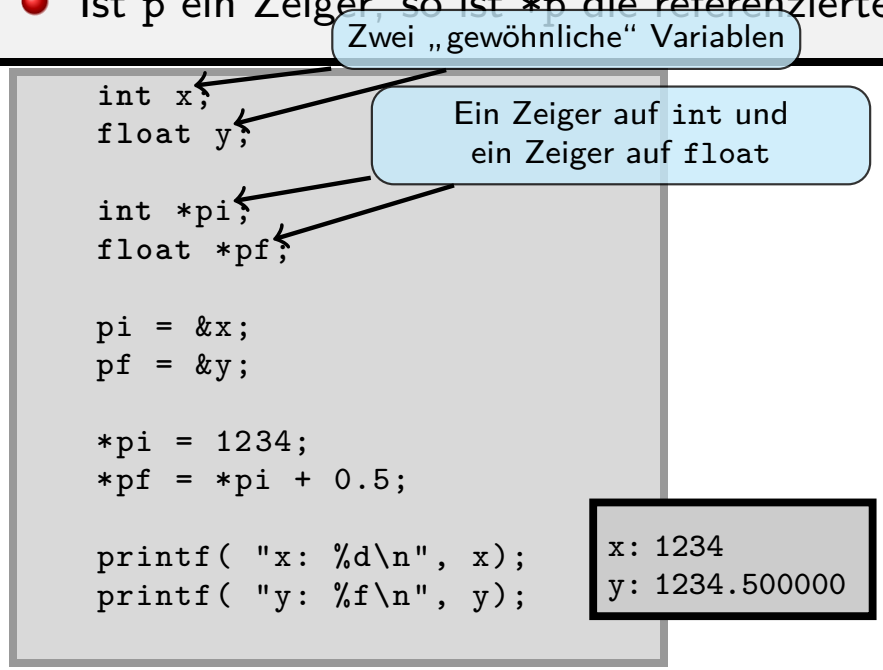
```
x: 1234
y: 1234.500000
```

- Der Dereferenzierungsoperator („*“) ist das Gegenstück zum Adressoperator („&“).
- Mit dem Adressoperator kommen wir von einer Variablen zu ihrer Adresse, mit dem Dereferenzierungsoperator kommen wir über die Adresse zur Variablen.

Zeiger

Hochschule RheinMain

- Eine Variable, in der die Adresse einer anderen Variablen gespeichert ist, nennen wir eine **Zeigervariable** oder kurz **Zeiger** bzw. **Pointer**.
- Die Variable, deren Adresse im Zeiger gespeichert ist, bezeichnen wir als die durch den Zeiger **referenzierte** oder **adressierte Variable**.
- Über einen Zeiger kann auf die Daten der referenzierten Variablen zugegriffen werden. Wir nennen dies **Indirektzugriff** oder auch **Dereferenzierung**. Zum Zugriff auf die referenzierte Variable verwendet man den Operator „*“ (**Dereferenzierungsoperator**).
- Ist p ein Zeiger, so ist *p die referenzierte Variable.

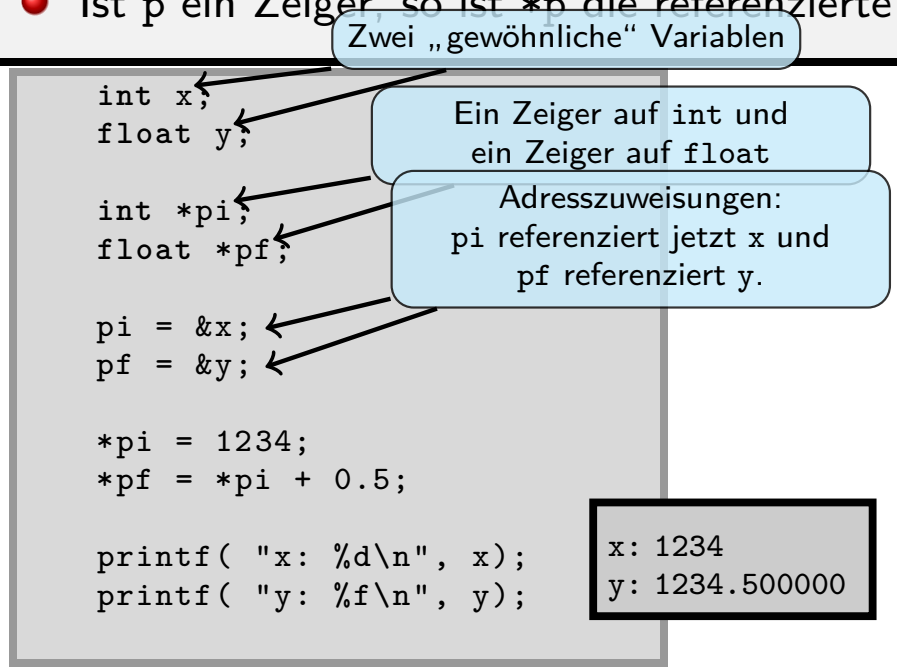


- Der Dereferenzierungsoperator („*“) ist das Gegenstück zum Adressoperator („&“).
- Mit dem Adressoperator kommen wir von einer Variablen zu ihrer Adresse, mit dem Dereferenzierungsoperator kommen wir über die Adresse zur Variablen.

Zeiger

Hochschule RheinMain

- Eine Variable, in der die Adresse einer anderen Variablen gespeichert ist, nennen wir eine **Zeigervariable** oder kurz **Zeiger** bzw. **Pointer**.
- Die Variable, deren Adresse im Zeiger gespeichert ist, bezeichnen wir als die durch den Zeiger **referenzierte** oder **adressierte Variable**.
- Über einen Zeiger kann auf die Daten der referenzierten Variablen zugegriffen werden. Wir nennen dies **Indirektzugriff** oder auch **Dereferenzierung**. Zum Zugriff auf die referenzierte Variable verwendet man den Operator „*“ (**Dereferenzierungsoperator**).
- Ist p ein Zeiger, so ist *p die referenzierte Variable.

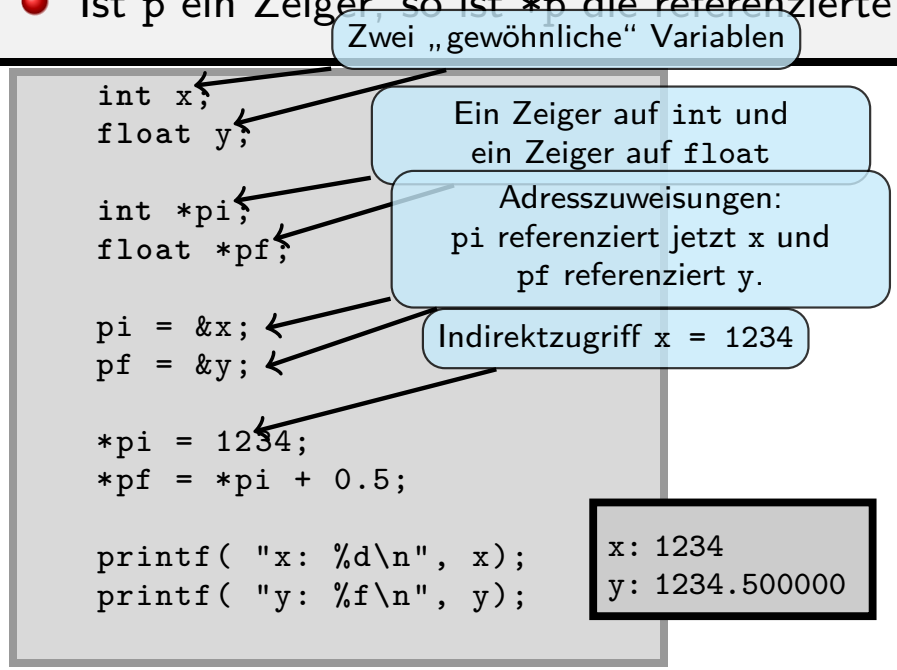


- Der Dereferenzierungsoperator („*“) ist das Gegenstück zum Adressoperator („&“).
- Mit dem Adressoperator kommen wir von einer Variablen zu ihrer Adresse, mit dem Dereferenzierungsoperator kommen wir über die Adresse zur Variablen.

Zeiger

Hochschule RheinMain

- Eine Variable, in der die Adresse einer anderen Variablen gespeichert ist, nennen wir eine **Zeigervariable** oder kurz **Zeiger** bzw. **Pointer**.
- Die Variable, deren Adresse im Zeiger gespeichert ist, bezeichnen wir als die durch den Zeiger **referenzierte** oder **adressierte Variable**.
- Über einen Zeiger kann auf die Daten der referenzierten Variablen zugegriffen werden. Wir nennen dies **Indirektzugriff** oder auch **Dereferenzierung**. Zum Zugriff auf die referenzierte Variable verwendet man den Operator „*“ (**Dereferenzierungsoperator**).
- Ist p ein Zeiger, so ist *p die referenzierte Variable.

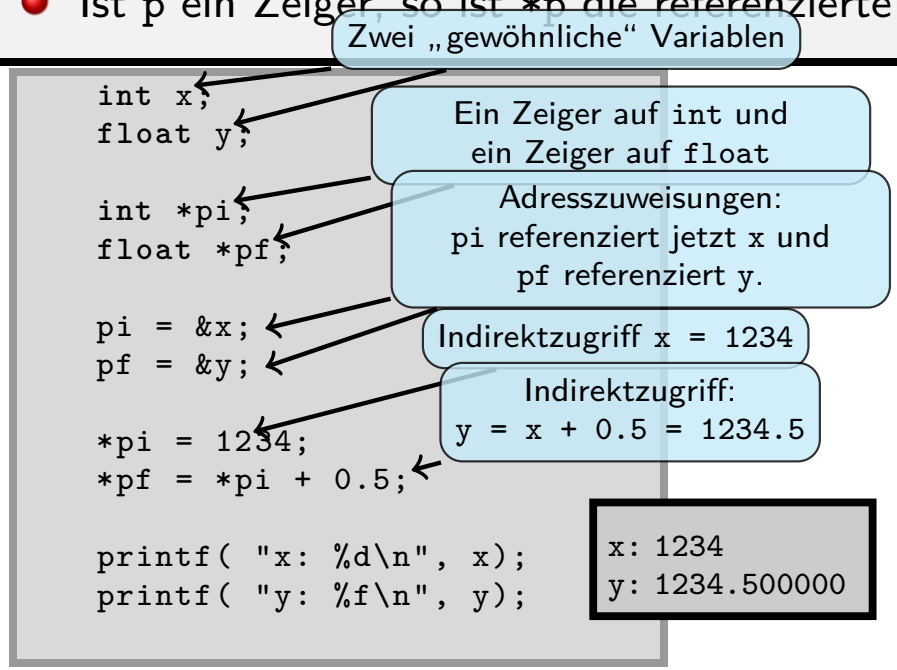


- Der Dereferenzierungsoperator („*“) ist das Gegenstück zum Adressoperator („&“).
- Mit dem Adressoperator kommen wir von einer Variablen zu ihrer Adresse, mit dem Dereferenzierungsoperator kommen wir über die Adresse zur Variablen.

Zeiger

Hochschule RheinMain

- Eine Variable, in der die Adresse einer anderen Variablen gespeichert ist, nennen wir eine **Zeigervariable** oder kurz **Zeiger** bzw. **Pointer**.
- Die Variable, deren Adresse im Zeiger gespeichert ist, bezeichnen wir als die durch den Zeiger **referenzierte** oder **adressierte Variable**.
- Über einen Zeiger kann auf die Daten der referenzierten Variablen zugegriffen werden. Wir nennen dies **Indirektzugriff** oder auch **Dereferenzierung**. Zum Zugriff auf die referenzierte Variable verwendet man den Operator „*“ (**Dereferenzierungsoperator**).
- Ist p ein Zeiger, so ist *p die referenzierte Variable.



- Der Dereferenzierungsoperator („*“) ist das Gegenstück zum Adressoperator („&“).
- Mit dem Adressoperator kommen wir von einer Variablen zu ihrer Adresse, mit dem Dereferenzierungsoperator kommen wir über die Adresse zur Variablen.

Tauschfunktion mit Zeigern

- Mit Adress- und Dereferenzierungsoperator kann man das Tauschproblem lösen:

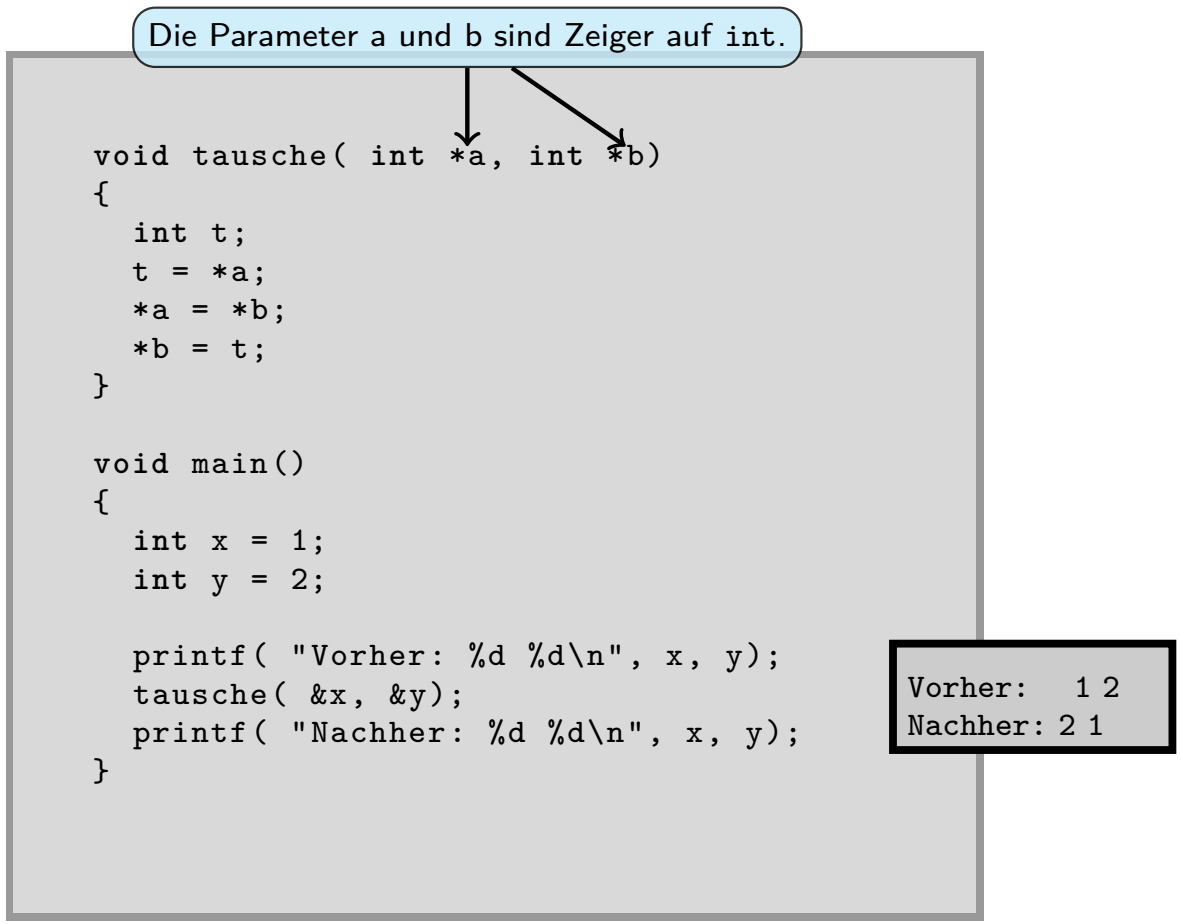
```
void tausche( int *a, int *b)
{
  int t;
  t = *a;
  *a = *b;
  *b = t;
}

void main()
{
  int x = 1;
  int y = 2;

  printf( "Vorher: %d %d\n", x, y);
  tausche( &x, &y);
  printf( "Nachher: %d %d\n", x, y);
}
```

```
Vorher:   1 2
Nachher: 2 1
```

Tauschfunktion mit Zeigern

Hochschule RheinMain

- Mit Adress- und Dereferenzierungsoperator kann man das Tauschproblem lösen:



Tauschfunktion mit Zeigern

Hochschule RheinMain

- Mit Adress- und Dereferenzierungsoperator kann man das Tauschproblem lösen:

Die Parameter a und b sind Zeiger auf int.

```
void tausche( int *a, int *b)
{
  int t;
  t = *a;
  *a = *b;
  *b = t;
}

void main()
{
  int x = 1;
  int y = 2;

  printf( "Vorher: %d %d\n", x, y);
  tausche( &x, &y);
  printf( "Nachher: %d %d\n", x, y);
}
```

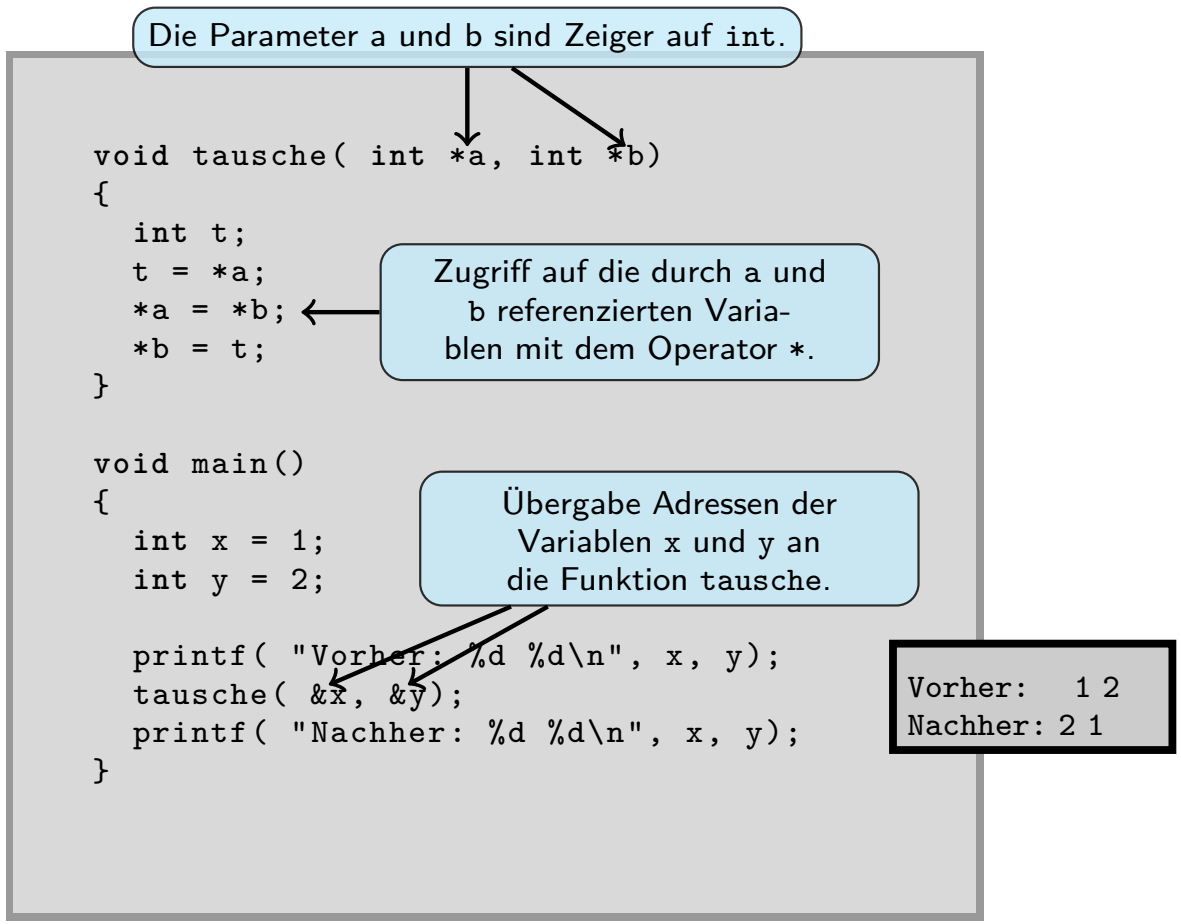
Zugriff auf die durch a und b referenzierten Variablen mit dem Operator *.

```
Vorher:  1 2
Nachher: 2 1
```

Tauschfunktion mit Zeigern

Hochschule RheinMain

- Mit Adress- und Dereferenzierungsoperator kann man das Tauschproblem lösen:



Rückgabe von Werten über Zeiger

- Man kann Zeiger verwenden, wenn man von einer Funktion mehr als einen Rückgabewert erwartet.

- Das rufende Programm stellt Variablen bereit, in die die aufgerufene Funktion Rückgabewerte über Zeiger einträgt.
- An der Schnittstelle werden die Adressen der Variablen übergeben.
- Die nebenstehende Funktion `minmax` bestimmt Minimum und Maximum in einem Array und schreibt die Werte über Zeiger in die Variablen des rufenden Programms.
- Bei `scanf` hatten wir dieses Prinzip schon immer verwendet.

```
void minmax( int anz, int daten[], int *pmin, int *pmax)
{
    int i, min, max;

    min = daten[0];
    max= daten[0];
    for( i = 1; i < anz; i++)
    {
        if( daten[i] < min) min = daten[i];
        if( daten[i] > max) max = daten[i];
    }
    *pmin = min;
    *pmax = max;
}

void main()
{
    int zahlen[8] = {1, -12, 31, 17, -11, 0, 22, 9};
    int min, max;

    minmax( 8, zahlen, &min, &max);

    printf( "Minimum: %d\n", min);
    printf( "Maximum: %d\n", max);
}
```

```
Minimum: -12
Maximum: 31
```

Rückgabe von Werten über Zeiger

- Man kann Zeiger verwenden, wenn man von einer Funktion mehr als einen Rückgabewert erwartet.

- Das rufende Programm stellt Variablen bereit, in die die aufgerufene Funktion Rückgabewerte über Zeiger einträgt.
- An der Schnittstelle werden die Adressen der Variablen übergeben.
- Die nebenstehende Funktion `minmax` bestimmt Minimum und Maximum in einem Array und schreibt die Werte über Zeiger in die Variablen des rufenden Programms.
- Bei `scanf` hatten wir dieses Prinzip schon immer verwendet.

Anzahl und Datenarray

```
void minmax( int anz, int daten[], int *pmin, int *pmax)
{
    int i, min, max;

    min = daten[0];
    max= daten[0];
    for( i = 1; i < anz; i++)
    {
        if( daten[i] < min) min = daten[i];
        if( daten[i] > max) max = daten[i];
    }
    *pmin = min;
    *pmax = max;
}

void main()
{
    int zahlen[8] = {1, -12, 31, 17, -11, 0, 22, 9};
    int min, max;

    minmax( 8, zahlen, &min, &max);

    printf( "Minimum: %d\n", min);
    printf( "Maximum: %d\n", max);
}
```

```
Minimum: -12
Maximum: 31
```

Rückgabe von Werten über Zeiger

Hochschule RheinMain

- Man kann Zeiger verwenden, wenn man von einer Funktion mehr als einen Rückgabewert erwartet.

- Das rufende Programm stellt Variablen bereit, in die die aufgerufene Funktion Rückgabewerte über Zeiger einträgt.
- An der Schnittstelle werden die Adressen der Variablen übergeben.
- Die nebenstehende Funktion `minmax` bestimmt Minimum und Maximum in einem Array und schreibt die Werte über Zeiger in die Variablen des rufenden Programms.
- Bei `scanf` hatten wir dieses Prinzip schon immer verwendet.

Anzahl und Datenarray

Zeiger auf die Variablen für die Rückgabe von Minimum und Maximum.

```
void minmax( int anz, int daten[], int *pmin, int *pmax)
{
    int i, min, max;

    min = daten[0];
    max= daten[0];
    for( i = 1; i < anz; i++)
    {
        if( daten[i] < min) min = daten[i];
        if( daten[i] > max) max = daten[i];
    }
    *pmin = min;
    *pmax = max;
}

void main()
{
    int zahlen[8] = {1, -12, 31, 17, -11, 0, 22, 9};
    int min, max;

    minmax( 8, zahlen, &min, &max);

    printf( "Minimum: %d\n", min);
    printf( "Maximum: %d\n", max);
}
```

```
Minimum: -12
Maximum: 31
```

Rückgabe von Werten über Zeiger

Hochschule RheinMain

- Man kann Zeiger verwenden, wenn man von einer Funktion mehr als einen Rückgabewert erwartet.

- Das rufende Programm stellt Variablen bereit, in die die aufgerufene Funktion Rückgabewerte über Zeiger einträgt.
- An der Schnittstelle werden die Adressen der Variablen übergeben.
- Die nebenstehende Funktion `minmax` bestimmt Minimum und Maximum in einem Array und schreibt die Werte über Zeiger in die Variablen des rufenden Programms.
- Bei `scanf` hatten wir dieses Prinzip schon immer verwendet.

Anzahl und Datenarray

Zeiger auf die Variablen für die Rückgabe von Minimum und Maximum.

Berechnung von Minimum und Maximum in `min` bzw. `max`.

```
void minmax( int anz, int daten[], int *pmin, int *pmax)
{
    int i, min, max;

    min = daten[0];
    max= daten[0];
    for( i = 1; i < anz; i++)
    {
        if( daten[i] < min) min = daten[i];
        if( daten[i] > max) max = daten[i];
    }
    *pmin = min;
    *pmax = max;
}

void main()
{
    int zahlen[8] = {1, -12, 31, 17, -11, 0, 22, 9};
    int min, max;

    minmax( 8, zahlen, &min, &max);

    printf( "Minimum: %d\n", min);
    printf( "Maximum: %d\n", max);
}
```

```
Minimum: -12
Maximum: 31
```

Rückgabe von Werten über Zeiger

Hochschule RheinMain

- Man kann Zeiger verwenden, wenn man von einer Funktion mehr als einen Rückgabewert erwartet.

- Das rufende Programm stellt Variablen bereit, in die die aufgerufene Funktion Rückgabewerte über Zeiger einträgt.
- An der Schnittstelle werden die Adressen der Variablen übergeben.
- Die nebenstehende Funktion `minmax` bestimmt Minimum und Maximum in einem Array und schreibt die Werte über Zeiger in die Variablen des rufenden Programms.
- Bei `scanf` hatten wir dieses Prinzip schon immer verwendet.

Anzahl und Datenarray

Zeiger auf die Variablen für die Rückgabe von Minimum und Maximum.

Berechnung von Minimum und Maximum in `min` bzw. `max`.

Speichern von Minimum und Maximum in den Variablen des rufenden Programms.

```
void minmax( int anz, int daten[], int *pmin, int *pmax)
{
    int i, min, max;

    min = daten[0];
    max= daten[0];
    for( i = 1; i < anz; i++)
    {
        if( daten[i] < min) min = daten[i];
        if( daten[i] > max) max = daten[i];
    }
    *pmin = min;
    *pmax = max;
}

void main()
{
    int zahlen[8] = {1, -12, 31, 17, -11, 0, 22, 9};
    int min, max;

    minmax( 8, zahlen, &min, &max);

    printf( "Minimum: %d\n", min);
    printf( "Maximum: %d\n", max);
}
```

```
Minimum: -12
Maximum: 31
```

Rückgabe von Werten über Zeiger

Hochschule RheinMain

- Man kann Zeiger verwenden, wenn man von einer Funktion mehr als einen Rückgabewert erwartet.

- Das rufende Programm stellt Variablen bereit, in die die aufgerufene Funktion Rückgabewerte über Zeiger einträgt.
- An der Schnittstelle werden die Adressen der Variablen übergeben.
- Die nebenstehende Funktion `minmax` bestimmt Minimum und Maximum in einem Array und schreibt die Werte über Zeiger in die Variablen des rufenden Programms.
- Bei `scanf` hatten wir dieses Prinzip schon immer verwendet.

```
void minmax( int anz, int daten[], int *pmin, int *pmax)
{
    int i, min, max;

    min = daten[0];
    max= daten[0];
    for( i = 1; i < anz; i++)
    {
        if( daten[i] < min) min = daten[i];
        if( daten[i] > max) max = daten[i];
    }
    *pmin = min;
    *pmax = max;
}

void main()
{
    int zahlen[8] = {1, -12, 31, 17, -11, 0, 22, 9};
    int min, max;

    minmax( 8, zahlen, &min, &max);

    printf( "Minimum: %d\n", min);
    printf( "Maximum: %d\n", max);
}
```

Anzahl und Datenarray

Zeiger auf die Variablen für die Rückgabe von Minimum und Maximum.

Berechnung von Minimum und Maximum in `min` bzw. `max`.

Speichern von Minimum und Maximum in den Variablen des rufenden Programms.

Übergabe der Adressen von `min` und `max` an die Funktion minimax.

```
Minimum: -12
Maximum: 31
```

Rückgabe von Adressen

Hochschule RheinMain

- Eine Funktion kann auch eine Adresse zurückgeben.
- Die nebenstehende Funktion `maximum` liefert mehr als nur den größeren von zwei Werten. Sie liefert die Adresse der Variablen, die den größeren Wert enthält.
- Achtung:
  Eine Funktion kann nicht korrekt die Adresse einer internen, lokalen Variable zurückgeben, da diese Variable nach dem Rücksprung aus der Funktion nicht mehr existiert.

```
int *maximum( int *x, int *y)
{
    if(*x > *y)
        return x;
    else
        return y;
}

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

    c= *maximum(&a, &b);
    printf( "a = %d, b = %d, c = %d\n", a, b, c);
}
```

```
a = 1, b = 2, c = 2
```

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

     *maximum(&a, &b) = 3;
    printf( "a = %d, b = %d\n", a, b);
}
```

```
a = 1, b = 3
```

Rückgabe von Adressen

Hochschule RheinMain

- Eine Funktion kann auch eine Adresse zurückgeben.
- Die nebenstehende Funktion `maximum` liefert mehr als nur den größeren von zwei Werten. Sie liefert die Adresse der Variablen, die den größeren Wert enthält.
- Achtung:
  Eine Funktion kann nicht korrekt die Adresse einer internen, lokalen Variable zurückgeben, da diese Variable nach dem Rücksprung aus der Funktion nicht mehr existiert.

Die Funktion gibt einen Zeiger auf int zurück.

```
int *maximum( int *x, int *y)
{
    if(*x > *y)
        return x;
    else
        return y;
}

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

    c= *maximum(&a, &b);
    printf( "a = %d, b = %d, c = %d\n", a, b, c);
}
```

```
a = 1, b = 2, c = 2
```

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

     *maximum(&a, &b) = 3;
    printf( "a = %d, b = %d\n", a, b);
}
```

```
a = 1, b = 3
```

Rückgabe von Adressen

- Eine Funktion kann auch eine Adresse zurückgeben.
- Die nebenstehende Funktion `maximum` liefert mehr als nur den größeren von zwei Werten. Sie liefert die Adresse der Variablen, die den größeren Wert enthält.
- Achtung:
  Eine Funktion kann nicht korrekt die Adresse einer internen, lokalen Variable zurückgeben, da diese Variable nach dem Rücksprung aus der Funktion nicht mehr existiert.

Die Funktion gibt einen Zeiger auf int zurück.

Die Werte der Variablen werden verglichen.

```
int *maximum( int *x, int *y)
{
    if(*x > *y)
        return x;
    else
        return y;
}

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

    c= *maximum(&a, &b);
    printf( "a = %d, b = %d, c = %d\n", a, b, c);
}
```

a = 1, b = 2, c = 2

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

     *maximum(&a, &b) = 3;
    printf( "a = %d, b = %d\n", a, b);
}
```

a = 1, b = 3

Rückgabe von Adressen

- Eine Funktion kann auch eine Adresse zurückgeben.
- Die nebenstehende Funktion `maximum` liefert mehr als nur den größeren von zwei Werten. Sie liefert die Adresse der Variablen, die den größeren Wert enthält.
- Achtung:
  Eine Funktion kann nicht korrekt die Adresse einer internen, lokalen Variable zurückgeben, da diese Variable nach dem Rücksprung aus der Funktion nicht mehr existiert.

Die Funktion gibt einen Zeiger auf `int` zurück.

Die Werte der Variablen werden verglichen.

Die Adresse der Variablen mit dem größeren Wert wird zurückgegeben.

```
int *maximum( int *x, int *y)
{
    if(*x > *y)
        return x;
    else
        return y;
}

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

    c= *maximum(&a, &b);
    printf( "a = %d, b = %d, c = %d\n", a, b, c);
}
```

```
a = 1, b = 2, c = 2
```

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

     *maximum(&a, &b) = 3;
    printf( "a = %d, b = %d\n", a, b);
}
```

```
a = 1, b = 3
```

Rückgabe von Adressen

- Eine Funktion kann auch eine Adresse zurückgeben.
- Die nebenstehende Funktion `maximum` liefert mehr als nur den größeren von zwei Werten. Sie liefert die Adresse der Variablen, die den größeren Wert enthält.
- Achtung:
  Eine Funktion kann nicht korrekt die Adresse einer internen, lokalen Variable zurückgeben, da diese Variable nach dem Rücksprung aus der Funktion nicht mehr existiert.

Die Funktion gibt einen Zeiger auf `int` zurück.

Die Werte der Variablen werden verglichen.

Die Adresse der Variablen mit dem größeren Wert wird zurückgegeben.

Über die zurückgegebene Adresse wird auf den Wert zugegriffen.

```
int *maximum( int *x, int *y)
{
    if(*x > *y)
        return x;
    else
        return y;
}

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

    c= *maximum(&a, &b);
    printf( "a = %d, b = %d, c = %d\n", a, b, c);
}
```

```
a = 1, b = 2, c = 2
```

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

     *maximum(&a, &b) = 3;
    printf( "a = %d, b = %d\n", a, b);
}
```

```
a = 1, b = 3
```

Rückgabe von Adressen

Hochschule RheinMain

- Eine Funktion kann auch eine Adresse zurückgeben.
- Die nebenstehende Funktion `maximum` liefert mehr als nur den größeren von zwei Werten. Sie liefert die Adresse der Variablen, die den größeren Wert enthält.
- Achtung:
  Eine Funktion kann nicht korrekt die Adresse einer internen, lokalen Variable zurückgeben, da diese Variable nach dem Rücksprung aus der Funktion nicht mehr existiert.

```
int *maximum( int *x, int *y)
{
    if(*x > *y)
        return x;
    else
        return y;
}

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

    c= *maximum(&a, &b);
    printf( "a = %d, b = %d, c = %d\n", a, b, c);
}
```

Die Funktion gibt einen Zeiger auf `int` zurück.

Die Werte der Variablen werden verglichen.

Die Adresse der Variablen mit dem größeren Wert wird zurückgegeben.

Über die zurückgegebene Adresse wird auf den Wert zugegriffen.

`a = 1, b = 2, c = 2`

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

     *maximum(&a, &b) = 3;
    printf( "a = %d, b = %d\n", a, b);
}
```

Hier wird über den zurückgegeben Zeiger zugegriffen. Das heißt, der Variablen mit dem größeren Wert wird 3 zugewiesen. Also b = 3..

`a = 1, b = 3`

Zeigerarithmetik

- Mit Zeigern kann gerechnet werden. Die Addition von zwei Zeigern ist nicht sinnvoll, aber zu einem Zeiger kann eine Zahl (offset) addiert werden. Das Ergebnis einer solchen Operation ist aber etwas anders, als auf den ersten Blick vermutet:

```
void main()
{
    int *p = 0;

    printf( "p + 0 = %d\n", p + 0);
    printf( "p + 1 = %d\n", p + 1);
    printf( "p + 2 = %d\n", p + 2);
    printf( "p + 3 = %d\n", p + 3);
}
```

```
p + 0 = 0
p + 1 = 4
p + 2 = 8
p + 3 = 12
```

```
void main()
{
    char *p = 0;

    printf( "p + 0 = %d\n", p + 0);
    printf( "p + 1 = %d\n", p + 1);
    printf( "p + 2 = %d\n", p + 2);
    printf( "p + 3 = %d\n", p + 3);
}
```

```
p + 0 = 0
p + 1 = 1
p + 2 = 2
p + 3 = 3
```

- Wenn man zu einem Zeiger den Wert 1 addiert, erhöht sich der Adresswert des Zeigers um die Größe des Datentyps, auf den der Zeiger zeigt. Der Zeiger zeigt damit nach Addition von 1 auf das nächstmögliche Datum des gleichen Typs.
- Bei Addition oder Subtraktion ganzer Zahlen ändert sich der Adresswert um entsprechende Vielfache der Größe des referenzierten Datentyps.
- Man kann auch die Differenz (p-q) zweier Zeiger des gleichen Zeigertyps bilden. Als Ergebnis erhält man das Offset, das man zu q addieren müsste, um p zu erhalten. Das ist anschaulich die Anzahl der Daten des entsprechenden Typs, die zwischen p und q passen.

Zeigerarithmetik

- Mit Zeigern kann gerechnet werden. Die Addition von zwei Zeigern ist nicht sinnvoll, aber zu einem Zeiger kann eine Zahl (offset) addiert werden. Das Ergebnis einer solchen Operation ist aber etwas anders, als auf den ersten Blick vermutet:

```
void main()
{
    int *p = 0;

    printf( "p + 0 = %d\n", p + 0);
    printf( "p + 1 = %d\n", p + 1);
    printf( "p + 2 = %d\n", p + 2);
    printf( "p + 3 = %d\n", p + 3);
}
```

Der Zeiger wird mit 0 initialisiert. Dann wird 0, 1, 2, 3 zum Zeigerwert addiert und der Adresswert ausgegeben

```
p + 0 = 0
p + 1 = 4
p + 2 = 8
p + 3 = 12
```

```
void main()
{
    char *p = 0;

    printf( "p + 0 = %d\n", p + 0);
    printf( "p + 1 = %d\n", p + 1);
    printf( "p + 2 = %d\n", p + 2);
    printf( "p + 3 = %d\n", p + 3);
}
```

```
p + 0 = 0
p + 1 = 1
p + 2 = 2
p + 3 = 3
```

- Wenn man zu einem Zeiger den Wert 1 addiert, erhöht sich der Adresswert des Zeigers um die Größe des Datentyps, auf den der Zeiger zeigt. Der Zeiger zeigt damit nach Addition von 1 auf das nächstmögliche Datum des gleichen Typs.
- Bei Addition oder Subtraktion ganzer Zahlen ändert sich der Adresswert um entsprechende Vielfache der Größe des referenzierten Datentyps.
- Man kann auch die Differenz (p-q) zweier Zeiger des gleichen Zeigertyps bilden. Als Ergebnis erhält man das Offset, das man zu q addieren müsste, um p zu erhalten. Das ist anschaulich die Anzahl der Daten des entsprechenden Typs, die zwischen p und q passen.

Zeigerarithmetik

Hochschule RheinMain

- Mit Zeigern kann gerechnet werden. Die Addition von zwei Zeigern ist nicht sinnvoll, aber zu einem Zeiger kann eine Zahl (offset) addiert werden. Das Ergebnis einer solchen Operation ist aber etwas anders, als auf den ersten Blick vermutet:

```
void main()
{
    int *p = 0;

    printf( "p + 0 = %d\n", p + 0);
    printf( "p + 1 = %d\n", p + 1);
    printf( "p + 2 = %d\n", p + 2);
    printf( "p + 3 = %d\n", p + 3);
}
```

Der Zeiger wird mit 0 initialisiert. Dann wird 0, 1, 2, 3 zum Zeigerwert addiert und der Adresswert ausgegeben

```
p + 0 = 0
p + 1 = 4
p + 2 = 8
p + 3 = 12
```

```
void main()
{
    char *p = 0;

    printf( "p + 0 = %d\n", p + 0);
    printf( "p + 1 = %d\n", p + 1);
    printf( "p + 2 = %d\n", p + 2);
    printf( "p + 3 = %d\n", p + 3);
}
```

Geänderter Datentyp

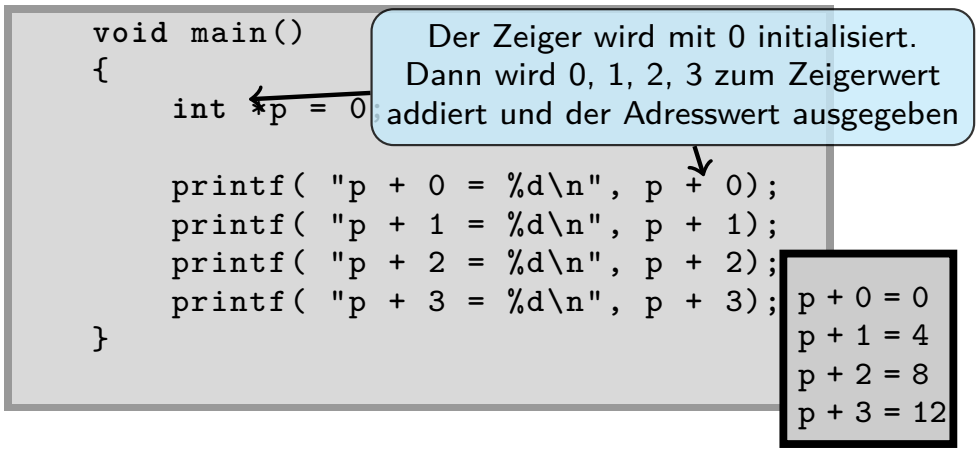
```
p + 0 = 0
p + 1 = 1
p + 2 = 2
p + 3 = 3
```

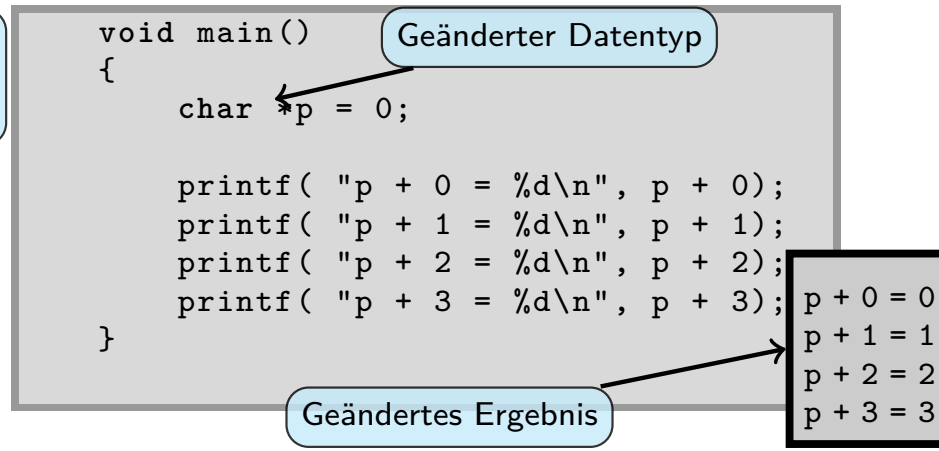
- Wenn man zu einem Zeiger den Wert 1 addiert, erhöht sich der Adresswert des Zeigers um die Größe des Datentyps, auf den der Zeiger zeigt. Der Zeiger zeigt damit nach Addition von 1 auf das nächstmögliche Datum des gleichen Typs.
- Bei Addition oder Subtraktion ganzer Zahlen ändert sich der Adresswert um entsprechende Vielfache der Größe des referenzierten Datentyps.
- Man kann auch die Differenz (p-q) zweier Zeiger des gleichen Zeigertyps bilden. Als Ergebnis erhält man das Offset, das man zu q addieren müsste, um p zu erhalten. Das ist anschaulich die Anzahl der Daten des entsprechenden Typs, die zwischen p und q passen.

Zeigerarithmetik

Hochschule RheinMain

- Mit Zeigern kann gerechnet werden. Die Addition von zwei Zeigern ist nicht sinnvoll, aber zu einem Zeiger kann eine Zahl (offset) addiert werden. Das Ergebnis einer solchen Operation ist aber etwas anders, als auf den ersten Blick vermutet:





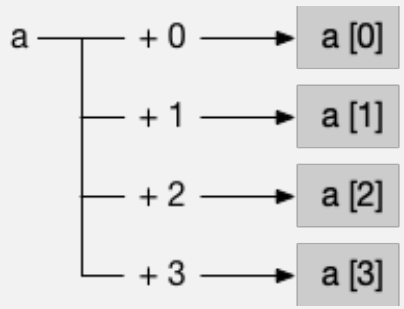
- Wenn man zu einem Zeiger den Wert 1 addiert, erhöht sich der Adresswert des Zeigers um die Größe des Datentyps, auf den der Zeiger zeigt. Der Zeiger zeigt damit nach Addition von 1 auf das nächstmögliche Datum des gleichen Typs.
- Bei Addition oder Subtraktion ganzer Zahlen ändert sich der Adresswert um entsprechende Vielfache der Größe des referenzierten Datentyps.
- Man kann auch die Differenz (p-q) zweier Zeiger des gleichen Zeigertyps bilden. Als Ergebnis erhält man das Offset, das man zu q addieren müsste, um p zu erhalten. Das ist anschaulich die Anzahl der Daten des entsprechenden Typs, die zwischen p und q passen.

Zeiger und Arrays

Hochschule RheinMain

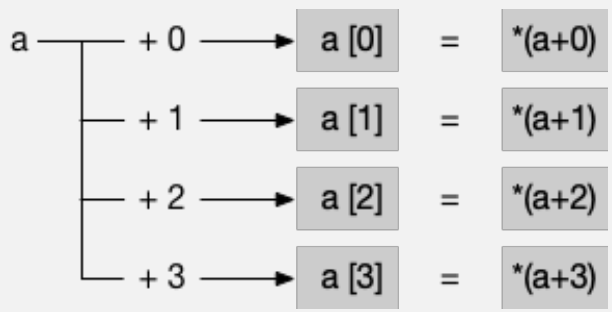
- Zeiger und Arrays sind eng miteinander verwandt.

- Ist a ein Array, so ist a zugleich ein Zeiger auf das erste Element des Arrays.
- Es gilt also `a = &a[0]`
- bzw. `*a = a[0]`



- Wegen der Gesetze der Zeigerarithmetik folgt dann für einen Index i:

- Ist a ein Array, so ist `a+i` ein Zeiger auf das i-te Element des Arrays.
- Es gilt also `a + i = &a[i]`
- bzw. `*(a+i) = a[i]`



Arrays und Strings als Funktionsparameter

- Wird ein Array (oder ein String) an einer Schnittstelle übergeben, so wird ein Zeiger übergeben und die Funktion erhält über diesen Zeiger direkten Zugriff auf die Daten im Array. Das Array wird an der Schnittstelle nicht kopiert, es entsteht nur eine Kopie des Zeigers.
- Beispiel: Zähle die `'a'` in einem String:

```
int zaehle( char *string)
{
    int a;

    for( a = 0; *string != 0; string++)
    {
        if( *string == 'a')
            a++;
    }
    return a;
}

void main()
{
    int anz;

    anz = zaehle( "Panamakanalaal");
    printf( "%d a gefunden\n", anz);
}
```

```
7 a gefunden
```

Arrays und Strings als Funktionsparameter

Hochschule RheinMain

- Wird ein Array (oder ein String) an einer Schnittstelle übergeben, so wird ein Zeiger übergeben und die Funktion erhält über diesen Zeiger direkten Zugriff auf die Daten im Array. Das Array wird an der Schnittstelle nicht kopiert, es entsteht nur eine Kopie des Zeigers.
- Beispiel: Zähle die `'a'` in einem String:

string ist ein Zeiger auf char.
*string ist das erste Zeichen im String.
string++ rückt den Zeiger auf das nächste Zeichen vor.

```
int zaehle( char *string)
{
    int a;

    for( a = 0; *string != 0; string++)
    {
        if( *string == 'a')
            a++;
    }
    return a;
}

void main()
{
    int anz;

    anz = zaehle( "Panamakanalaal");
    printf( "%d a gefunden\n", anz);
}
```

```
7 a gefunden
```

Beispiel - Palindromerkennung

- Ein Palindrom ist ein Wort, das vorwärts und rückwärts gelesen gleich ist.

```
int palindrom( char *string)
{
    char *vorn = string;
    char *hinten = string;

    for(; *hinten != 0; hinten++)
        ;
    hinten--;

    for( ; vorn < hinten; vorn++, hinten--)
    {
        if( *vorn != *hinten)
            return 0;
    }
    return 1;

}

void main()
{
    int ok;

    ok = palindrom( "retsinakanister");
    if( ok == 1)
        printf( "Palindrom erkannt\n");
}
```

```
Palindrom erkannt
```

Beispiel - Palindromerkennung

- Ein Palindrom ist ein Wort, das vorwärts und rückwärts gelesen gleich ist.

Zeiger auf den zu untersuchenden Text

```
int palindrom( char *string)
{
    char *vorn = string;
    char *hinten = string;

    for(; *hinten != 0; hinten++)
        ;
    hinten--;

    for( ; vorn < hinten; vorn++, hinten--)
    {
        if( *vorn != *hinten)
            return 0;
    }
    return 1;

}

void main()
{
    int ok;

    ok = palindrom( "retsinakanister");
    if( ok == 1)
        printf( "Palindrom erkannt\n");
}
```

```
Palindrom erkannt
```

Beispiel - Palindromerkennung

- Ein Palindrom ist ein Wort, das vorwärts und rückwärts gelesen gleich ist.

```
int palindrom( char *string)
{
    char *vorn = string;
    char *hinten = string;

    for(; *hinten != 0; hinten++)
        ;
    hinten--;

    for( ; vorn < hinten; vorn++, hinten--)
    {
        if( *vorn != *hinten)
            return 0;
    }
    return 1;

}

void main()
{
    int ok;

    ok = palindrom( "retsinakanister");
    if( ok == 1)
        printf( "Palindrom erkannt\n");
}
```

Zeiger auf den zu untersuchenden Text

Zwei Hilfszeiger, die auf den Anfang des Textes gesetzt werden.

```
Palindrom erkannt
```

Beispiel - Palindromerkennung

Hochschule RheinMain

- Ein Palindrom ist ein Wort, das vorwärts und rückwärts gelesen gleich ist.

```
int palindrom( char *string)
{
    char *vorn = string;
    char *hinten = string;

    for(; *hinten != 0; hinten++)
        ;
    hinten--;

    for( ; vorn < hinten; vorn++, hinten--)
    {
        if( *vorn != *hinten)
            return 0;
    }
    return 1;

}

void main()
{
    int ok;

    ok = palindrom( "retsinakanister");
    if( ok == 1)
        printf( "Palindrom erkannt\n");
}
```

Zeiger auf den zu untersuchenden Text

Zwei Hilfszeiger, die auf den Anfang des Textes gesetzt werden.

Der Zeiger hinten wird bis zum Terminator vorgeschoben und dann wieder ein Zeichen zurückgesetzt. Damit zeigt er auf das letzte Zeichen.

```
Palindrom erkannt
```

Beispiel - Palindromerkennung

Hochschule RheinMain

- Ein Palindrom ist ein Wort, das vorwärts und rückwärts gelesen gleich ist.

```
int palindrom( char *string)
{
    char *vorn = string;
    char *hinten = string;

    for(; *hinten != 0; hinten++)
        ;
    hinten--;

    for( ; vorn < hinten; vorn++, hinten--)
    {
        if( *vorn != *hinten)
            return 0;
    }
    return 1;

}

void main()
{
    int ok;

    ok = palindrom( "retsinakanister");
    if( ok == 1)
        printf( "Palindrom erkannt\n");
}
```

Zeiger auf den zu untersuchenden Text

Zwei Hilfszeiger, die auf den Anfang des Textes gesetzt werden.

Der Zeiger hinten wird bis zum Terminator vorgeschoben und dann wieder ein Zeichen zurückgesetzt. Damit zeigt er auf das letzte Zeichen.

Die Zeiger laufen vorwärts bzw. rückwärts durch den Text, bis sie sich in der Mitte treffen. Werden dabei Unterschiede festgestellt, ist es kein Palindrom.

```
Palindrom erkannt
```

Beispiel - Palindromerkennung

Hochschule RheinMain

- Ein Palindrom ist ein Wort, das vorwärts und rückwärts gelesen gleich ist.

```
int palindrom( char *string)
{
    char *vorn = string;
    char *hinten = string;

    for(; *hinten != 0; hinten++)
        ;
    hinten--;

    for( ; vorn < hinten; vorn++, hinten--)
    {
        if( *vorn != *hinten)
            return 0;
    }
    return 1;

}

void main()
{
    int ok;

    ok = palindrom( "retsinakanister");
    if( ok == 1)
        printf( "Palindrom erkannt\n");
}
```

Zeiger auf den zu untersuchenden Text

Zwei Hilfszeiger, die auf den Anfang des Textes gesetzt werden.

Der Zeiger hinten wird bis zum Terminator vorgeschoben und dann wieder ein Zeichen zurückgesetzt. Damit zeigt er auf das letzte Zeichen.

Die Zeiger laufen vorwärts bzw. rückwärts durch den Text, bis sie sich in der Mitte treffen. Werden dabei Unterschiede festgestellt, ist es kein Palindrom.

Palindrom erkannt.

```
Palindrom erkannt
```

Minimum oder das Maximum in einem Array

- Über einen Parameter (`modus`) wird gesteuert, welche Funktion (`minimum` oder `maximum`) verwendet werden soll.

```
int suche( int anz, int *daten, int modus)
{
    int i, m;

    m = daten[0];
    for( i = 1; i < anz; i++)
    {
        if( modus == 1)
            m = minimum( m, daten[i]);
        else
            m = maximum( m, daten[i]);
    }
    return m;
}

void main()
{
    int zahlen[8] = {1, -12, 31, 17, -11, 0, 22, 9};
    int min, max;

    min = suche( 8, zahlen, 1);
    printf( "Minimum: %d\n", min);

    max = suche( 8, zahlen, 2);
    printf( "Maximum: %d\n", max);
}
```

```
Minimum: -12
Maximum:  31
```

```
int minimum( int a, int b)
{
    if( a < b)
        return a;
    return b;
}
```

```
int maximum( int a, int b)
{
    if( a > b)
        return a;
    return b;
}
```

Minimum oder das Maximum in einem Array

Hochschule RheinMain

- Über einen Parameter (`modus`) wird gesteuert, welche Funktion (`minimum` oder `maximum`) verwendet werden soll.

```
int suche( int anz, int *daten, int modus)
{
    int i, m;

    m = daten[0];
    for( i = 1; i < anz; i++)
    {
        if( modus == 1)
            m = minimum( m, daten[i]);
        else
            m = maximum( m, daten[i]);
    }
    return m;
}

void main()
{
    int zahlen[8] = {1, -12, 31, 17, -11, 0, 22, 9};
    int min, max;

    min = suche( 8, zahlen, 1);
    printf( "Minimum: %d\n", min);

    max = suche( 8, zahlen, 2);
    printf( "Maximum: %d\n", max);
}
```

Über den Parameter `modus` wird gesteuert, ob das Minimum oder das Maximum berechnet werden soll.

```
Minimum: -12
Maximum:  31
```

```
int minimum( int a, int b)
{
    if( a < b)
        return a;
    return b;
}
```

```
int maximum( int a, int b)
{
    if( a > b)
        return a;
    return b;
}
```

Minimum oder das Maximum in einem Array

Hochschule RheinMain

- Über einen Parameter (`modus`) wird gesteuert, welche Funktion (`minimum` oder `maximum`) verwendet werden soll.

```
int suche( int anz, int *daten, int modus)
{
    int i, m;

    m = daten[0];
    for( i = 1; i < anz; i++)
    {
        if( modus == 1)
            m = minimum( m, daten[i]);
        else
            m = maximum( m, daten[i]);
    }
    return m;
}

void main()
{
    int zahlen[8] = {1, -12, 31, 17, -11, 0, 22, 9};
    int min, max;

    min = suche( 8, zahlen, 1);
    printf( "Minimum: %d\n", min);

    max = suche( 8, zahlen, 2);
    printf( "Maximum: %d\n", max);
}
```

Über den Parameter `modus` wird gesteuert, ob das Minimum oder das Maximum berechnet werden soll.

Die Funktion `suche` wird mit unterschiedlichem `modus` aufgerufen.

```
Minimum: -12
Maximum:  31
```

```
int minimum( int a, int b)
{
    if( a < b)
        return a;
    return b;
}
```

```
int maximum( int a, int b)
{
    if( a > b)
        return a;
    return b;
}
```

Funktionszeiger

- Eine Funktion hat, wie eine Variable, auch eine Adresse. Die Adresse einer Funktion kann man als Parameter an eine andere Funktion übergeben.

```
int suche( int anz, int *daten, int fkt( int, int))
{
     int i, m;

     m = daten[0];
     for( i = 1; i < anz; i++)
         m = fkt( m, daten[i]);
     return m;
}

void main()
{
    int zahlen[8] = {1, -12, 31, 17, -11, 0, 22, 9};
    int min, max;

    min = suche( 8, zahlen, minimum);
    printf( "Minimum: %d\n", min);

    max = suche( 8, zahlen, maximum);
    printf( "Maximum: %d\n", max);
}
```

```
Minimum: -12
Maximum:  31
```

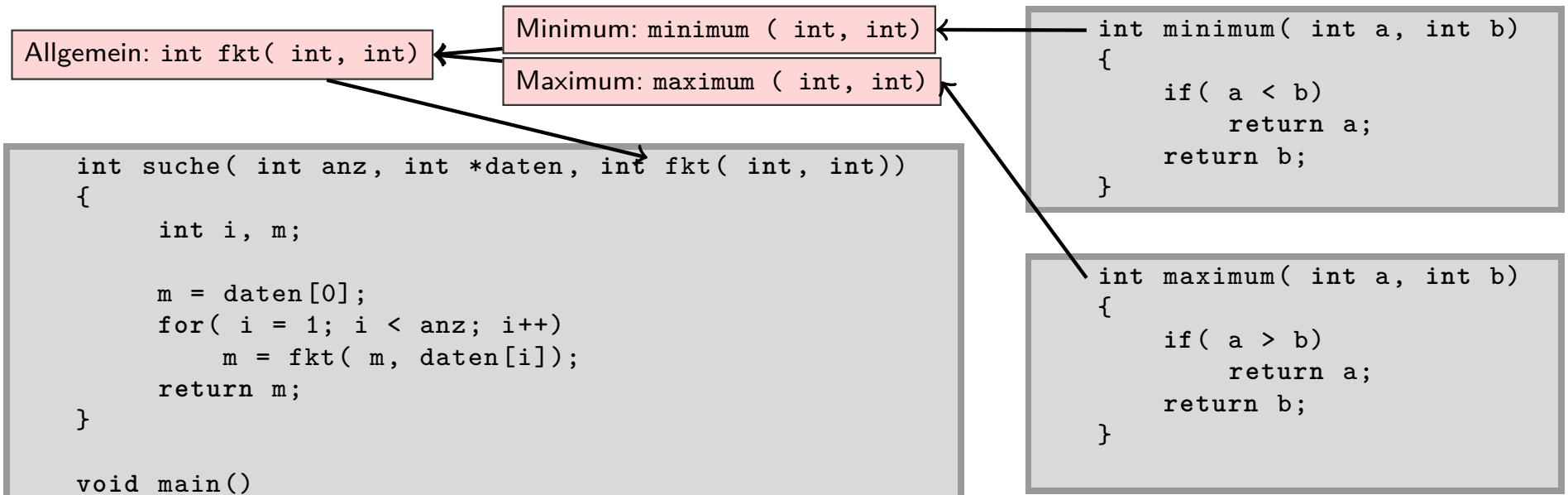
```
int minimum( int a, int b)
{
    if( a < b)
        return a;
    return b;
}
```

```
int maximum( int a, int b)
{
    if( a > b)
        return a;
    return b;
}
```

Funktionszeiger

Hochschule RheinMain

- Eine Funktion hat, wie eine Variable, auch eine Adresse. Die Adresse einer Funktion kann man als Parameter an eine andere Funktion übergeben.



Funktionszeiger

Hochschule RheinMain

- Eine Funktion hat, wie eine Variable, auch eine Adresse. Die Adresse einer Funktion kann man als Parameter an eine andere Funktion übergeben.

Allgemein: `int fkt( int, int)`

Minimum: `minimum ( int, int)`

Maximum: `maximum ( int, int)`

```
int minimum( int a, int b)
{
    if( a < b)
        return a;
    return b;
}
```

```
int maximum( int a, int b)
{
    if( a > b)
        return a;
    return b;
}
```

```
int suche( int anz, int *daten, int fkt( int, int))
{
     int i, m;

     m = daten[0];
     for( i = 1; i < anz; i++)
         m = fkt( m, daten[i]);
     return m;
}

void main()
{
    int zahlen[8] = {1, -12, 31, 17, -11, 0, 22, 9};
    int min, max;

    min = suche( 8, zahlen, minimum);
    printf( "Minimum: %d\n", min);

    max = suche( 8, zahlen, maximum);
    printf( "Maximum: %d\n", max);
}
```

Im Parameter `fkt` wird eine Funktion übergeben, die zwei int-Werte übergeben bekommt und einen int-Wert zurückgibt.

```
Minimum: -12
Maximum:  31
```

Funktionszeiger

Hochschule RheinMain

- Eine Funktion hat, wie eine Variable, auch eine Adresse. Die Adresse einer Funktion kann man als Parameter an eine andere Funktion übergeben.

Allgemein: `int fkt( int, int)`

Minimum: `minimum ( int, int)`

Maximum: `maximum ( int, int)`

```
int minimum( int a, int b)
{
    if( a < b)
        return a;
    return b;
}
```

```
int maximum( int a, int b)
{
    if( a > b)
        return a;
    return b;
}
```

```
int suche( int anz, int *daten, int fkt( int, int))
{
     int i, m;

     m = daten[0];
     for( i = 1; i < anz; i++)
         m = fkt( m, daten[i]);
     return m;
}

void main()
{
    int zahlen[8] = {1, -12, 31, 17, -11, 0, 22, 9};
    int min, max;

    min = suche( 8, zahlen, minimum);
    printf( "Minimum: %d\n", min);

    max = suche( 8, zahlen, maximum);
    printf( "Maximum: %d\n", max);
}
```

Im Parameter `fkt` wird eine Funktion übergeben, die zwei int-Werte übergeben bekommt und einen int-Wert zurückgibt.

Die im Parameter `fkt` übergebene Funktion wird aufgerufen.

```
Minimum: -12
Maximum:  31
```

Funktionszeiger

Hochschule RheinMain

- Eine Funktion hat, wie eine Variable, auch eine Adresse. Die Adresse einer Funktion kann man als Parameter an eine andere Funktion übergeben.

Allgemein: `int fkt( int, int)`

Minimum: `minimum ( int, int)`

Maximum: `maximum ( int, int)`

```
int minimum( int a, int b)
{
    if( a < b)
        return a;
    return b;
}
```

```
int maximum( int a, int b)
{
    if( a > b)
        return a;
    return b;
}
```

```
int suche( int anz, int *daten, int fkt( int, int))
{
     int i, m;

     m = daten[0];
     for( i = 1; i < anz; i++)
         m = fkt( m, daten[i]);
     return m;
}

void main()
{
    int zahlen[8] = {1, -12, 31, 17, -11, 0, 22, 9};
    int min, max;

    min = suche( 8, zahlen, minimum);
    printf( "Minimum: %d\n", min);

    max = suche( 8, zahlen, maximum);
    printf( "Maximum: %d\n", max);
}
```

Im Parameter `fkt` wird eine Funktion übergeben, die zwei int-Werte übergeben bekommt und einen int-Wert zurückgibt.

Die im Parameter `fkt` übergebene Funktion wird aufgerufen.

Im dritten Parameter wird die bei der Suche zu verwendende Hilfsfunktion übergeben.

```
Minimum: -12
Maximum:  31
```

Callback-Funktionen

- Im letzten Beispiel haben wir einen wichtigen Programmierstil kennengelernt:
- **Funktionen, die einer anderen Funktion als Parameter übergeben werden, um von dieser zurückgerufen zu werden, werden auch als Callback-Funktionen bezeichnet.**
- Das Programmieren mit Callback-Funktionen ist eine weit verbreitete Technik, die zum Beispiel bei der systemnahen Programmierung oder auch bei der Programmierung grafischer Benutzeroberflächen intensiv verwendet wird.

Ein-/Ausgabe über Zeiger

- Was eine Adresse genau ist, hat uns im Rahmen der bisherigen Betrachtungen nicht wirklich interessiert.
- Der „Ort“, an dem eine Variable oder eine Funktion im Hauptspeicher hinterlegt ist, wird vom Compiler, vom Linker, bzw. von der Speicherverwaltung des Betriebssystems festgelegt, er entzieht sich somit unserer Kontrolle.

→ Eine Variable wird durch ein Speicherwort[1] repräsentiert, dessen Adresse von der Speicherverwaltung festgelegt wird.

- Jeder Rechner verfügt über **Ein-/Ausgabegeräte**, über die der Rechner mit der Außenwelt in Verbindung treten kann. Beispiele solcher Ein-Ausgabegeräte sind: Tastatur, Bildschirm, Schnittstellen für externe Geräte, etc.
- Beim sogenannten *memory-mapped I/O* werden Ein-/Ausgabegeräte über sogenannte *Register* angesteuert, die -ebenso wie Speicherworte- addressierbar sind.
- Die Adressen dieser Register unterliegen jedoch nicht der Speicherverwaltung, sondern sie sind durch die Art des Anschlusses „hart verdrahtet“.
- Wenn man ihre Adressen kennt, kann man über einen Zeiger, dem man diese Adresse zuweist, auf solche Register zugreifen.

```
/* Function setUserLedOff() */
/* Turns off user LED */
void setUserLedOff(void)
{
   /* clear bit 7 (8th from right) in PORTC register */
  PORTC &= ~( 1 << 7 );
}
```

[1] Vgl. Kapitel 3

Ein-/Ausgabe über Zeiger

Hochschule RheinMain

- Was eine Adresse genau ist, hat uns im Rahmen der bisherigen Betrachtungen nicht wirklich interessiert.
- Der „Ort“, an dem eine Variable oder eine Funktion im Hauptspeicher hinterlegt ist, wird vom Compiler, vom Linker, bzw. von der Speicherverwaltung des Betriebssystems festgelegt, er entzieht sich somit unserer Kontrolle.

→ Eine Variable wird durch ein Speicherwort[1] repräsentiert, dessen Adresse von der Speicherverwaltung festgelegt wird.

- Jeder Rechner verfügt über **Ein-/Ausgabegeräte**, über die der Rechner mit der Außenwelt in Verbindung treten kann. Beispiele solcher Ein-Ausgabegeräte sind: Tastatur, Bildschirm, Schnittstellen für externe Geräte, etc.
- Beim sogenannten *memory-mapped I/O* werden Ein-/Ausgabegeräte über sogenannte *Register* angesteuert, die -ebenso wie Speicherworte- addressierbar sind.
- Die Adressen dieser Register unterliegen jedoch nicht der Speicherverwaltung, sondern sie sind durch die Art des Anschlusses „hart verdrahtet“.
- Wenn man ihre Adressen kennt, kann man über einen Zeiger, dem man diese Adresse zuweist, auf solche Register zugreifen.

```
/* Function setUserLedOff() */
/* Turns off user LED */
void setUserLedOff(void)
{
   /* clear bit 7 (8th from right) in PORTC register */
  PORTC &= ~( 1 << 7 );
}
```

Lösche Bit Nr. 7 in Register PORTC ⇒ LED wird ausgeschaltet

[1] Vgl. Kapitel 3

Ein-/Ausgabe über Zeiger

- E/A-Register erscheinen wie Variablen im Speicher (unter einer speziellen, vorab bekannten Adresse)
- Können ebenso wie solche von C-Programmen gelesen/manipuliert werden.

```
#include <avr/io.h>
void blink( int wie_oft )
{
  DDRC = ( 1 << 7 ); // PORTC Bit Nr. 7 als Ausgang konfigurieren
  while( wie_oft-- )
  {
    PORTC |= ( 1 << 7);    // LED an
    _delay_ms( 250 );
    PORTC &= ~( 1 << 7 ); // LED aus
    _delay_ms ( 250 );
  }
}
```

- (Vereinfachter) Auszug aus `<avr/io.h>`:

```
#define _MMIO_BYTE(mem_addr) (*(volatile uint8_t *)(mem_addr))
#define _SFR_IO8(io_addr) _MMIO_BYTE((io_addr) + 0x20)
#define DDRC _SFR_IO8(0x07)
#define PORTC _SFR_IO8(0x08)
```

- **Allgemein:** `(*(volatile <Typ> *)(<Adresse>))`

Ein-/Ausgabe über Zeiger

- E/A-Register erscheinen wie Variablen im Speicher (unter einer speziellen, vorab bekannten Adresse)
- Können ebenso wie solche von C-Programmen gelesen/manipuliert werden.

DDRC und PORTC „sehen aus“ wie globale Variablen

```
#include <avr/io.h>
void blink( int wie_oft )
{
  DDRC |= ( 1 << 7 ); // PORTC Bit Nr. 7 als Ausgang konfigurieren
  while( wie_oft-- )
  {
    PORTC |= ( 1 << 7);    // LED an
    _delay_ms( 250 );
    PORTC &= ~( 1 << 7 ); // LED aus
    _delay_ms ( 250 );
  }
}
```

- (Vereinfachter) Auszug aus <avr/io.h>:

```
#define _MMIO_BYTE(mem_addr) (*(volatile uint8_t *)(mem_addr))
#define _SFR_IO8(io_addr) _MMIO_BYTE((io_addr) + 0x20)
#define DDRC _SFR_IO8(0x07)
#define PORTC _SFR_IO8(0x08)
```

- **Allgemein:** `(*(volatile <Typ> *)(<Adresse>))`

Ein-/Ausgabe über Zeiger

Hochschule RheinMain

- E/A-Register erscheinen wie Variablen im Speicher (unter einer speziellen, vorab bekannten Adresse)
- Können ebenso wie solche von C-Programmen gelesen/manipuliert werden.

```
#include <avr/io.h>
void blink( int wie_oft )
{
  DDRC |= ( 1 << 7 ); // PORTC Bit Nr. 7 als Ausgang konfigurieren
  while( wie_oft-- )
  {
    PORTC |= ( 1 << 7 );   // LED an
    _delay_ms( 250 );
    PORTC &= ~( 1 << 7 ); // LED aus
    _delay_ms ( 250 );
  }
}
```

DDRC und PORTC „sehen aus“ wie globale Variablen

C-Preprozessor-Macros: Textuelle Ersetzungen:

- z.B. DDRC → _SFR_IO8(0x07)
- _SFR_IO8(0x07) → _MMIO_BYTE((0x07)+0x20)
- _MMIO_BYTE((0x07)+0x20) → (*(volatile uint8_t *)(0x27))

- (Vereinfachter) Auszug aus <avr/io.h>:

```
#define _MMIO_BYTE(mem_addr) (*(volatile uint8_t *)(mem_addr))
#define _SFR_IO8(io_addr) _MMIO_BYTE((io_addr) + 0x20)
#define DDRC _SFR_IO8(0x07)
#define PORTC _SFR_IO8(0x08)
```

- Allgemein: (*(volatile <Typ> *)(<Adresse>))

Ein-/Ausgabe über Zeiger

Hochschule RheinMain

- E/A-Register erscheinen wie Variablen im Speicher (unter einer speziellen, vorab bekannten Adresse)
- Können ebenso wie solche von C-Programmen gelesen/manipuliert werden.

```
#include <avr/io.h>
void blink( int wie_oft )
{
  DDRC |= ( 1 << 7 ); // PORTC Bit Nr. 7 als Ausgang konfigurieren
  while( wie_oft-- )
  {
    PORTC |= ( 1 << 7 );   // LED an
    _delay_ms( 250 );
    PORTC &= ~( 1 << 7 ); // LED aus
    _delay_ms ( 250 );
  }
}
```

DDRC und PORTC „sehen aus“ wie globale Variablen

C-Preprozessor-Macros: Textuelle Ersetzungen:

- z.B. DDRC → _SFR_IO8(0x07)
- _SFR_IO8(0x07) → _MMIO_BYTE((0x07)+0x20)
- _MMIO_BYTE((0x07)+0x20) → (*(volatile uint8_t *)(0x27))

- (Vereinfachter) Auszug aus <avr/io.h>:

```
#define _MMIO_BYTE(mem_addr) (*(volatile uint8_t *)(mem_addr))
#define _SFR_IO8(io_addr) _MMIO_BYTE((io_addr) + 0x20)
#define DDRC _SFR_IO8(0x07)
#define PORTC _SFR_IO8(0x08)
```

- **Allgemein:** `(*(volatile <Typ> *)(<Adresse>))`

Ein-/Ausgabe über Zeiger

- `(*(volatile <Typ> *)(<Adresse>))`
- Allgemein: Ein *Cast* der Form „(<Typ>)<Objekt>“ erzwingt, dass <Objekt> als vom Typ <Typ> aufgefasst wird.
- Speziell wird hier die Zahl vom Typ `int` als Zeiger auf Objekte vom Typ `volatile <Typ>` aufgefasst, der durch den vorangestellten „*“-Operator unmittelbar de-referenziert wird.
- `<Typ>` definiert dabei die Wortgröße des Registers, auf das zugegriffen wird.
  - `char`: 8 Bit
  - `short`: i.d.R. 16 Bit
  - `int`: i.d.R. 16 oder 32 Bit
  - `long`: i.d.R. 32 Bit
  - `long long`: i.d.R. 64 Bit
- Die C-Standardtypen `short`, `int`, `long`, `long long` haben u.U. keine standardisierte Größe.
- Um definierte Wortgrößen für Register portabel beschreiben zu können, definieren C-Programmierumgebungen in der Regel entsprechende Typen, z.B. avr-gcc `<stdint.h>`:
  - `int8_t`: 8 Bit vorzeichenbehaftet
  - `uint8_t`: 8 Bit vorzeichenlos
  - `int16_t`: 16 Bit vorzeichenbehaftet
  - `uint16_t`: 26 Bit vorzeichenlos
  - `int32_t`: 32 Bit vorzeichenbehaftet
  - `uint32_t`: 32 Bit vorzeichenlos

Warum volatile?

- Fallstrick: Codeoptimierung durch den Compiler

```
#include <avr/io.h>
void blink( int wie_oft )
{
  DDRC = ( 1 << 7 ); // PORTC Bit Nr. 7 als Ausgang konfigurieren
  while( wie_oft-- )
  {
    PORTC |= ( 1 << 7);
    _delay_ms( 250 );
    PORTC &= ~( 1 << 7 );
    _delay_ms ( 250 );
  }
}
```

- Aufgrund der (an sich vernünftigen) Annahme, dass Variablenwerte nur infolge von Wertzuweisungen geändert werden können, könnte der Compiler eine der beiden Wertzuweisungen „wegoptimieren“
- Das Attribut `volatile` unterbindet dies.
- Hardwareregister werden in der Regel als `volatile` erklärt, um solche Optimierungen zu vermeiden.

Warum volatile?

- Fallstrick: Codeoptimierung durch den Compiler

```
#include <avr/io.h>
void blink( int wie_oft )
{
  DDRC = ( 1 << 7 ); // PORTC Bit Nr. 7 als Ausgang konfigurieren
  while( wie_oft-- )
  {
    PORTC |= ( 1 << 7);
    _delay_ms( 250 );
    PORTC &= ~( 1 << 7 );
    _delay_ms ( 250 );
  }
}
```

Wertzuweisung an PORTC

- Aufgrund der (an sich vernünftigen) Annahme, dass Variablenwerte nur infolge von Wertzuweisungen geändert werden können, könnte der Compiler eine der beiden Wertzuweisungen „wegoptimieren“
- Das Attribut `volatile` unterbindet dies.
- Hardwareregister werden in der Regel als `volatile` erklärt, um solche Optimierungen zu vermeiden.

Warum volatile?

- Fallstrick: Codeoptimierung durch den Compiler

```
#include <avr/io.h>
void blink( int wie_oft )
{
  DDRC = ( 1 << 7 ); // PORTC Bit Nr. 7 als Ausgang konfigurieren
  while( wie_oft-- )
  {
    PORTC |= ( 1 << 7);
    _delay_ms( 250 );
    PORTC &= ~( 1 << 7 );
    _delay_ms ( 250 );
  }
}
```

Wertzuweisung an PORTC

Keine Verwendungung des Wertes von PORTC

- Aufgrund der (an sich vernünftigen) Annahme, dass Variablenwerte nur infolge von Wertzuweisungen geändert werden können, könnte der Compiler eine der beiden Wertzuweisungen „wegoptimieren“
- Das Attribut `volatile` unterbindet dies.
- Hardwareregister werden in der Regel als `volatile` erklärt, um solche Optimierungen zu vermeiden.

Warum volatile?

Hochschule RheinMain

- Fallstrick: Codeoptimierung durch den Compiler

```
#include <avr/io.h>
void blink( int wie_oft )
{
  DDRC = ( 1 << 7 ); // PORTC Bit Nr. 7 als Ausgang konfigurieren
  while( wie_oft-- )
  {
    PORTC |= ( 1 << 7);
    _delay_ms( 250 );
    PORTC &= ~( 1 << 7 );
    _delay_ms ( 250 );
  }
}
```

Wertzuweisung an PORTC

Keine Verwendungung des Wertes von PORTC

Erneute Wertzuweisung an PORTC

- Aufgrund der (an sich vernünftigen) Annahme, dass Variablenwerte nur infolge von Wertzuweisungen geändert werden können, könnte der Compiler eine der beiden Wertzuweisungen „wegoptimieren“
- Das Attribut `volatile` unterbindet dies.
- Hardwareregister werden in der Regel als `volatile` erklärt, um solche Optimierungen zu vermeiden.