

# Zeiger

Zeiger (engl. pointer) sind Variablen, die als Inhalt eine Adresse enthalten. Wie andere Variablen auch, haben sie einen Namen und es gibt erlaubte Operationen.

Viele Klassen in C++ und sehr viele Programme in C sind ohne Zeiger nicht realisierbar. Zeiger haben aber einige Tücken:

- sie sind die Ursache von vielen Programmfehlern,
- sie können zu schlecht lesbar und wartbaren Programmen führen.

Aber richtig verwendet ist damit ein effektiver Code erzeugbar.

## Inhalt

1. Zeiger und Adressen .....	3
2. Zeiger und die Argumente von Funktionen .....	10
3. Zeiger und Vektoren .....	16
3.1.1. Unterschied Vektorname und Zeiger .....	20
3.1.2. Vektoren als Parameter.....	20
3.1.3. Zeigerarithmetik .....	24

3.2. Vektoren von Zeigern .....	28
4. Dynamische Datenobjekte .....	29
4.1. Erzeugung von dynamischer Datenobjekten .....	31
4.2. Freigabe von dynamisch erzeugten Datenobjekten .....	36
4.3. Rekursive dynamische Strukturen.....	37
4.3.1. Verkettete Listen.....	39
4.3.2. Suchen in Tabellen - Hashing.....	47
5. Zeiger auf Funktionen .....	53
6. Typedef.....	59
7. Argumente aus der Kommandozeile .....	62

## 1. Zeiger und Adressen

Ein Zeiger muss wie jede Variable vor ihrer ersten Verwendung deklariert werden.

Allgemeine Form:

*Typbezeichnung \* Name*

kennzeichnet *Name* als **Zeiger** auf Speicher, der Objekte vom Typ *Typbezeichnung* enthält

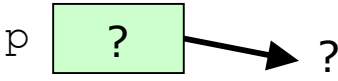
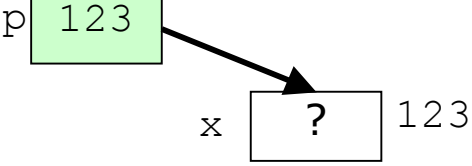
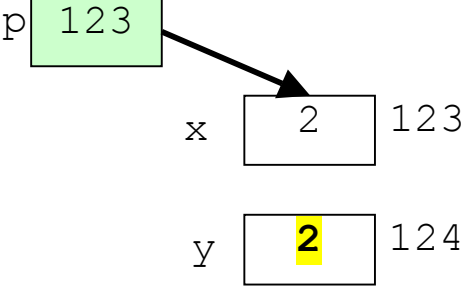
Mit dem **Adressoperator** & kann man die Adresse einer Speicherzelle ermitteln. Der Adressoperator kann auf Variablen und Arrayelement angewendet werden, nicht auf Ausdrücke (sie haben keinen Speicherplatz).

Da ein Zeiger die Adresse eines Objektes enthält, kann auf das Objekt auf zwei Arten zugegriffen werden:

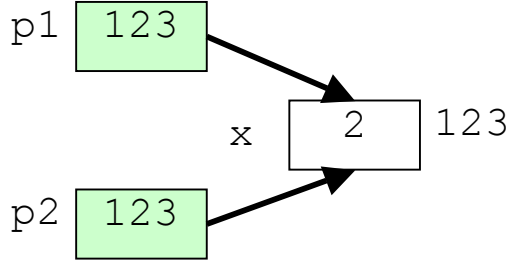
- direkt über den Namen
- indirekt über den Zeiger.

Um indirekt zuzugreifen, wird der Operator \* zum **Dereferenzieren** verwendet:

## Beispiele:

C++ Fragment	Erklärung	Veranschaulichung
<code>int *p;</code>	p ist eine Zeigervariable auf ein int-Objekt. Da keine Initialisierung vorgenommen ist, ist der Inhalt unbestimmt.	
<code>int *p; int x; p = &amp;x;</code>	p wird die Adresse der Variablen x zugewiesen, man sagt p zeigt auf x.	
<code>int x, y; int *p; x = 2; p = &amp;x; y = *p;</code>	p zeigt auf x. Durch <code>y=*p</code> wird derselbe Effekt erzielt, wie <code>y = x</code> .	

**p dereferenzieren:** Inhalt der Speicherstelle, auf die p zeigt.

<pre>int x; int *p1, *p2; x = 2; p1 = &amp;x; p2 = p1;</pre>	<p>Der Variablen <code>p2</code> wird der Wert der Variablen <code>p1</code> zugewiesen; dadurch zeigt <code>p2</code> auf dasselbe Objekt wie <code>p1</code>.</p>	
<pre>int* ip, x; int * ip, x; int *ip, x;</pre>	<p>Alle diese Deklarationen sind <b>äquivalent</b>, da der <code>*</code> sich nur auf den ihm nachfolgenden Namen bezieht.</p>	

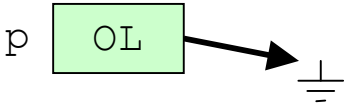
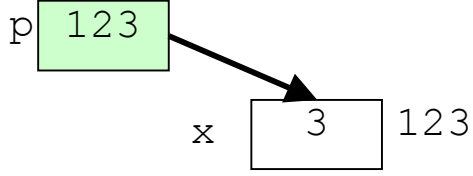
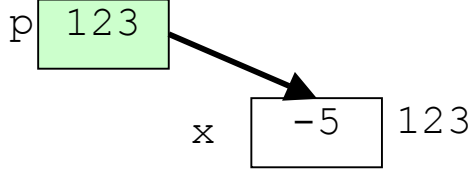
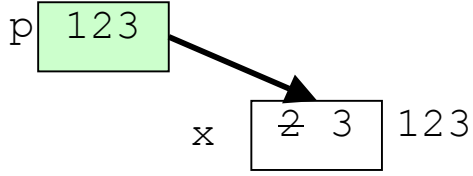
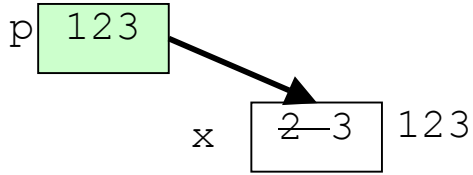
Wird eine Zeigervariable deklariert, so ist sie nicht automatisch initialisiert; sie zeigt irgendwo hin. Will man sie auf einen definierten Wert setzen, aber noch nicht auf ein bestimmtes Objekt, so kann der spezielle Zeigerwert **NULL** verwendet werden.

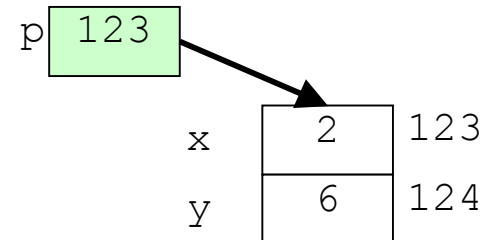
`NULL` wird intern als `0L` dargestellt; man sollte der besseren Lesbarkeit wegen aber den Namen `NULL` verwenden. Die Definition von `Null` ist im Header `<stddef>` enthalten.

In Ausdrücken, wie `y = *p + 1;` haben die Operatoren `*` und `&` **höheren** Rang als arithmetische Operatoren. D.h. **zuerst** wird oben das Objekt ermittelt (**dereferenziert**) dann wird der Wert um 1 erhöht. Achtung: `y = *(p+1)` möglich, hat aber andere Bedeutung (->später).

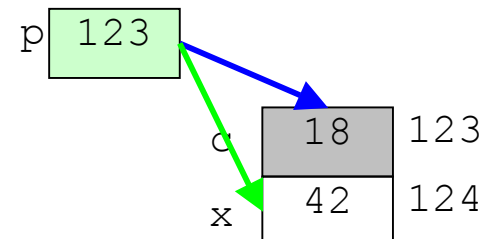
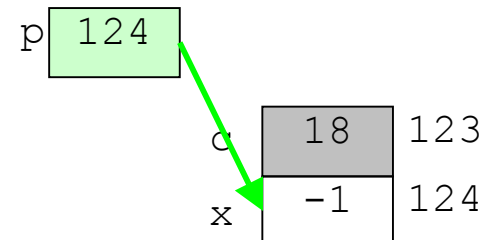
Verweise sind auch links in Zuweisungen möglich.

## Beispiele:

C++ Fragment	Erklärung	Veranschaulichung
<code>int *p=NULL;</code>	p ist eine Zeigervariable auf ein int-Objekt.	
<code>int *p; int x=3; p = &amp;x; x = *p + 1;</code>	Zuerst wird dereferenziert, d.h. der Wert von der Speicherstelle, auf die p zeigt ermittelt (x==3), dann der Wert erhöht (4) und der Variablen x zugewiesen.	
<code>int *p; int x=2; p = &amp;x; *p = -5;</code>	Zuerst die Speicherstelle, auf die p zeigt ermittelt (x), dann der Wert dieser Speicherstelle auf -5 gesetzt.	
<code>int *p; int x=2; p = &amp;x; *p += 1;</code>	Inkrementiert x.	
<code>int *p; int x=2; p = &amp;x; (*p)++;</code>	Inkrementiert x. Die Klammern sind wegen Rang erforderlich.	

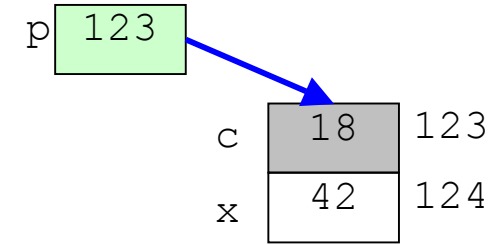
<pre>int *p; int x=2, y=5; p = &amp;x; (*p++)++;</pre>	<p>Inkrementiert das Objekt, das im Speicher auf x folgt!</p> <p><b>Achtung:</b> dies führt zu unleserlichen Programmen und ist extrem fehleranfällig! 💣</p>	 <p>p 123</p> <p>x 2 123</p> <p>y 6 124</p>
--	--	---

Zeiger sind auch als Konstante deklarierbar – sie zeigen dann stets auf dasselbe Objekt. Auch können Zeiger auf Konstanten deklariert werden.

C++ Fragment	Erklärung	Veranschaulichung
<pre>const int x=42; const int c=18; const int *p=&amp;c; //blau ... p=&amp;x;           //grün</pre>	<p>Solche Deklarationen sind von rechts nach links zu lesen:  <b>p ist Zeiger auf eine Int-Konstante.</b>  Der Zeiger kann danach auf anderes <b>Int-Objekt</b> zeigen.</p>	 <p>p 123</p> <p>c 18 123</p> <p>x 42 124</p>
<pre>int x=42; const int c=18; int *const p=&amp;x; *p = -1;</pre>	<p>Solche Deklarationen sind von rechts nach links zu lesen:  <b>p ist konstanter Zeiger auf ein Int-Objekt.</b>  Der Zeiger kann danach <b>nicht</b> verändert werden, aber das Objekt auf das er zeigt.</p>	 <p>p 124</p> <p>c 18 123</p> <p>x -1 124</p>

```
int x=42;
const int c=18;
const int *const p=&c;
```

Solche Deklarationen sind von rechts nach links zu lesen:  
**p ist konstanter Zeiger auf eine Int-Konstante.**  
Der Zeiger kann danach **nicht** verändert werden, das Objekt auf das er zeigt ebenfalls nicht.



Welche Ausgabe erzeugt das nachfolgende Programm?

```
$ cat wasWirdGedruckt.cpp
#include <iostream.h>
main() {
    double zahl = 1;
    double zahl2 = 2;
    const double konstante = 11;
    const double konstante2 = 12;
    double *pd = &zahl;
    cout << zahl - *pd << endl;
    cout << *pd << endl;
    cout << pd << endl;

    const double *pdc = &konstante;
    cout << *pdc << endl;
    cout << pdc << endl;

    double *const cpd = &zahl2;
    cout << *cpd << endl;
    cout << cpd << endl;

    const double *const cpdc = &konstante2;
    cout << *cpdc << endl;
    cout << cpdc << endl;

    pdc = &konstante2;
    cout << *pd + *pdc + *cpd + *cpdc << endl;
}
$
```

## 2. Zeiger und die Argumente von Funktionen

Die Parameterübergabemechanismen haben wir bereits behandelt. Soll die Änderung eines Parameters einer Funktion beim Aufrufer verfügbar sein, musste der Übergabemechanismus „Call by Referenz“ verwendet werden.

Mittels Wertübergabe (Call by Value) kann dieser Effekt erzielt werden, wenn ein Zeiger übergeben wird. Dies **bleibt** Wertübergabe, da der Zeiger selbst nicht verändert wird, lediglich das Objekt auf das er zeigt wird geändert.

Soll mit call by value eine Änderung durch das aufgerufene Unterprogramm erzielt werden, so müssen Zeiger übergeben werden **und** das Unterprogramm muss dereferenzieren.

Beispiel (increment):

```

$ cat inc3.cpp
#include <iostream.h>
void inc(int *p) {
    (*p)++;
    cout <<" in inc:*p = " << *p << endl;
}

main() {
    int x=1;
    int *pi = &x;
    cout << " vor inc3: x = " << x << endl;
    inc3(pi);
    cout << "nach inc3: x = " << x << endl;
}
$

```

Klammer wg.  
Rang erforderlich!

```

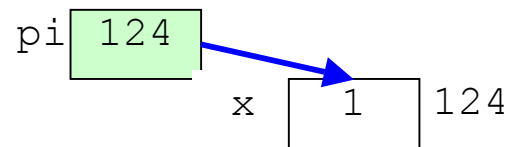
$ inc3

vor inc3: x = 1
in inc3:*p = 2
nach inc3: x = 2

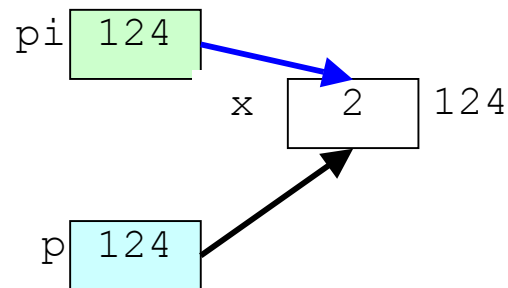
$

```

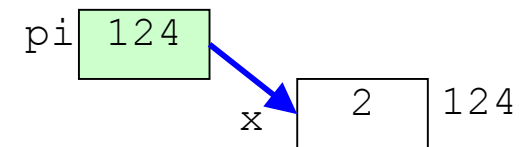
Main vor Aufruf inc



inc nach (\*p)++



Main nach Aufruf inc



Dass sich aber der Zeiger selbst nicht geändert hat, also immer noch Wertübergabe und nicht Referenzübergabe durchgeführt wurde, verdeutlicht die folgende Erweiterung am Programm:

<pre>\$ cat inc4.cpp #include &lt;iostream.h&gt; void inc4(int *p) {     (*p)++;     cout &lt;&lt;"  in inc4: *p = " &lt;&lt; *p &lt;&lt; endl;     cout &lt;&lt;"  in inc4:  p = " &lt;&lt; *p &lt;&lt; endl; }  main() {     int x=1;     int *pi = &amp;x;     cout &lt;&lt; " vor inc4:  x = " &lt;&lt; x &lt;&lt; endl;     cout &lt;&lt; " vor inc4: pi = " &lt;&lt; pi &lt;&lt; endl;     inc4(pi);     cout &lt;&lt; "nach inc4:  x = " &lt;&lt; x &lt;&lt; endl;     cout &lt;&lt; "nach inc4: pi = " &lt;&lt; pi &lt;&lt; endl; } \$</pre>	<pre>\$ inc4  vor inc4:  x = 1 vor inc4: pi = 0x22feb0   in inc4: *p = 2   in inc4:  p = 2 nach inc4:  x = 2 nach inc4: pi = 0x22feb0 \$</pre> <p>} =</p>
--	---

In C ist ausschließlich Wertübergabe möglich. Deshalb sind Zeiger in C der einzige Weg (Arrays werden wir später behandeln und sehen, dass sie eigentlich wie Zeiger behandelt werden), den fehlenden Referenzübergabemechanismus zu überwinden.

Das Ergebnis einer Funktion kann selbst ein Zeiger sein.

## Beispiel (max)

```
$ cat max.cpp
#include <iostream>
int * max(int *a, int *b) {
    if (*a > *b)
        return a;
    else
        return b;
}

main() {
    int x,y;
    int *pi;
    cout << "Zwei Interger eingeben: ";
    cin >> x >> y;
    pi = max(&x, &y);
    cout << *pi << endl;
}
$
```

Oft wird beim Programmieren mit Zeigern der Fehler gemacht, dass eine Funktion einen Wert über den Zeiger zurück geben will, der nach dem Funktionsaufruf nicht mehr lebt, da der Zeiger auf ein auto-Objekt gezeigt hat.

Beispiel (fehlerhafte Verwendung von Zeigern)

```
cat zeigerFehler01.cpp
#include <iostream.h>
int *fkt() {
    int x=1;
    return &x;           // x existiert nach return nicht mehr!
}

main() {
    int *pi ;
    pi = fkt();
    cout << "fkt(): " << *pi << endl;
}
$
```

## Hörsaalübung:

Realisieren Sie je eine void-Funktion zum Addieren von zwei float-Werten, die die beiden zu addierenden Werte als formale Parameter besitzt und einen zusätzlichen Ergebnisparameter hat, der

1. ein Referenzparameter ist
2. ein Zeiger ist.

Eine dritte Funktion zum Addieren soll zwei formale Parameter haben und als Ergebnis einen Zeiger auf den Resultatswert haben.

Für alle drei Funktionen ist ein Testprogramm zu entwickeln.

### 3. Zeiger und Vektoren

Von C her hat C++ eine Erbschaft übernommen; Zeiger und Arrays (häufig auch Vektoren genannt) sind stark verwandt:

Jede Operation mit Vektor-Indizes kann auch mit Zeigern formuliert werden!

Die mit Zeiger formulierte Version ist i.a. effizienter, jedoch manchmal schwerer zu „lesen“.

Durch

```
int a[10]
```

wird ein Vektor mit Namen `a` definiert, der 10 Integer-Komponenten `a[0]`, `a[1]`, bis `a[9]` hat. Dann bezeichnet `a[i]` die Komponente, die `i` Positionen von der ersten Komponente entfernt ist.

Wenn `pa` ein Zeiger auf ein `int`-Objekt ist, also

```
int *pa
```

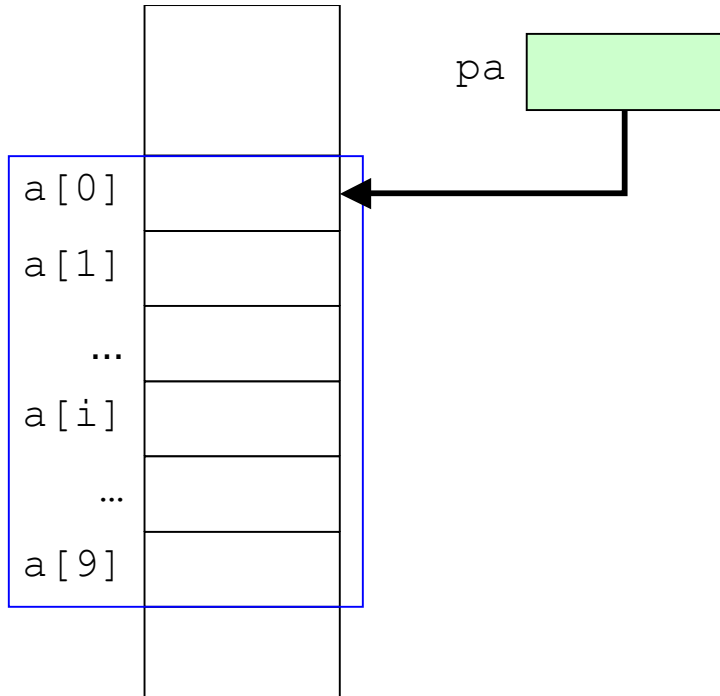
dann kann durch

```
pa = &a[0]
```

`pa` die Adresse der Komponente 0 zugewiesen werden.

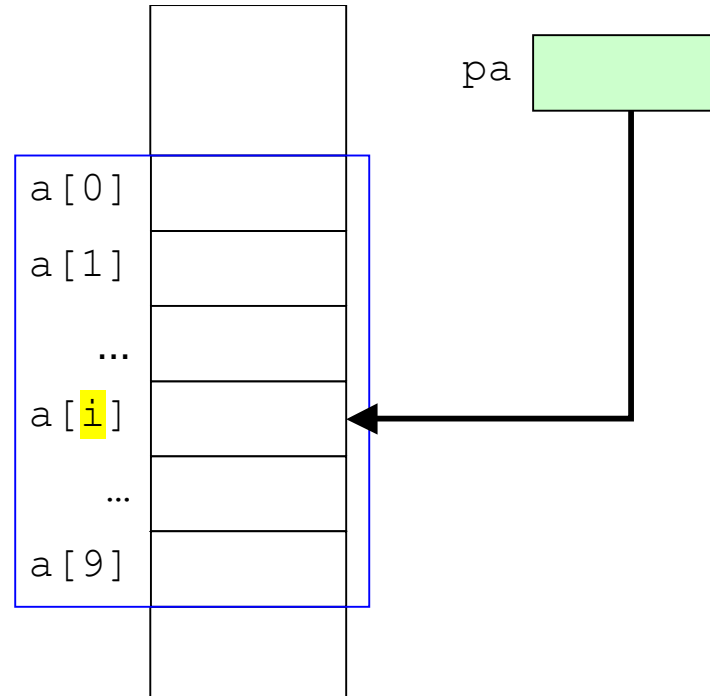
```
int a[10];  
int *pa = &a[0];
```

Speicher



```
int a[10];  
int *pa = &a[i];
```

Speicher



Der C/C++-Compiler verwandelt einen Verweis auf einen Vektor stets in einen Zeiger-Wert auf den Anfang des Vektors. D.h. der **Name eines Vektors** ist die **Startadresse im Speicher**. Daher gilt immer:

$$*(pa+i) == a[i]$$

## Konsequenzen:

Ausdruck	ist äquivalent zu	Erklärung
<code>pa = &amp;a[0]</code>	<code>pa = a</code>	Der Name des Vektors ist Synonym für die Adresse der Komponente 0.
<code>&amp;a[i]</code>	<code>a+i</code>	Adresse der Komponente $i$
<code>a[i]</code>	<code>*(a+i)</code>	Inhalt der Komponente $i$
<code>pa[i]</code>	<code>*(pa+i)</code>	<code>pa</code> ist Synonym für Vektoranfang

Allgemein kann man sagen:

Statt **Vektorname und Indexausdruck** kann man immer einen **Zeiger und den Abstand** zum Anfang angeben und umgekehrt.

Diese Konvention ist **unabhängig vom Typ** des Vektors, da die Addition von 1 zu einem Zeiger definiert ist als das Inkrementieren in Abhängigkeit des Speicherplatzbedarfs des Objektes, auf den der Zeiger zeigt.

## Beispiel (was wird gedruckt?)

```
$ cat wasWirdGedruckt02.cpp
#include <iostream.h>
main() {
    int a[] = {0,1,2,3,4};
    int *p;

    cout << "A: ";
    for (int i = 0; i <= 4; i++)
        cout << a[i] << " ";
    cout << endl;

    cout << "B: ";
    for (p = &a[0]; p <= &a[4]; p++)
        cout << *p << " ";
    cout << endl;

    cout << "C: ";
    int j;
    for (j = 1, p = &a[0]; j <= 5; j++)
        cout << p[j] << " ";
    cout << endl;
}
$
```

### 3.1.1. Unterschied Vektorname und Zeiger

Ein Zeiger ist eine Variable, also sind die Anweisungen

```
int a[10];  
int *pa;  
pa = a;  
pa++;
```

sinnvoll und **möglich**.

Ein Vektorname ist vom Wesen her eine Konstante und kann daher kein L-Wert sein. Somit sind

```
int a[10];  
int *pa;  
a = pa;  
a++;  
pa = &a;
```

**nicht** möglich.

### 3.1.2. Vektoren als Parameter

Wir haben gesehen, dass es unterschiedliche Parameterübergabemechanismen gibt. Wird ein Vektor als Parameter an eine Funktion übergeben, wird in Wirklichkeit die Adresse der Komponente 0 übergeben, da ja der Compiler den Vektornamen durch die Startadresse ersetzt.

Konsequenz:

1. Obwohl **keine Referenzübergabe** (durch &) beim formalen Parameter angegeben ist, ist der **Effekt** einer Änderung in der Funktion beim Aufrufer **ersichtlich**.
2. Die **Größe** eines Vektors kann innerhalb der Funktion **nicht** mit `sizeof` ermittelt werden, da innerhalb der Funktion nur ein Zeiger verwendet wird.

Beispiel (Vektor als Parameter):

```
$ cat vektorToUpper.cpp
#include <iostream.h>
#include <ctype.h>
void toUpper(char str[], int start, int end) {
    for (int i=start; i<=end; i++)
        str[i] =toupper(str[i]);
}

main() {
    char s[]="Jennifer Lopez";
    toUpper(s,0,8);
    cout << s << endl;
}
$
```

```
$ vektorToUpper
JENNIFER Lopez
$
```

Das Ändern des Strings innerhalb der Funktion ist beim Aufrufer nach dem Aufruf ersichtlich, das ja eine Adresse übergeben wird. Aber auch hier ist der Übergabemechanismus immer noch „Call by Value“; der Zeiger selbst wird **nicht** verändert. Die kann man sich zunutze ma-

chen, indem man den Zeiger in der Funktion ändert, um eine Variable in der Funktion zu sparen:

## Beispiel (strlen)

```
$ cat strlen.cpp
#include <iostream.h>
int strlen(char *s) {
    int n;
    for (n=0; *s != '\0'; s++)
        n++;
    return n;
}

main() {
    char str[80];
    cout << "Wort: ";
    cin >> str;
    cout << str << " hat Länge " << strlen(str) << endl;
}
$
```

als formale Parameter ist  
char s[] und char \*s  
gleichbedeutend!

'\0' schließt C-  
Zeichenketten ab!

## Hörsaalübung:

Zu realisieren ist eine Funktion `atoi`, die die als formalen Parameter übergebenen Zeichenkette (String mit Ziffern) in den korrespondierenden Integer konvertiert.

Gesucht sind zwei Lösungen

1. eine Lösung mit Vektoren,
2. eine Lösung mit Zeigern.

Beispiel:

```
cout << atoi("12345") ; gibt den Wert 12345 ais.
```

### 3.1.3. Zeigerarithmetik

Im letzten Beispiel haben wir gesehen, dass man einen Zeiger inkrementieren kann. In C++ ist das „Rechnen“ mit Zeigern möglich.

Vorsicht 💣:

Programme die **Zeigerarithmetik** verwenden können sehr **effizient** sein, aber die Gefahr von **Fehlern** und **nicht** mehr **wartbarem Code** ist extrem groß!

Erlaubte Operationen mit Zeigern sind:

1. Addition oder Subtraktion von Zeiger-Wert und ganzer Zahl
2. Subtraktion von 2 Zeigern
3. Vergleich von Zeigern

D.h. zwei Zeiger können **nicht**:

- addiert,
- multipliziert,
- dividiert oder
- geschiftet

werden.

Die Addition bzw. Subtraktion von Zeiger mit Konstante erfolgt immer in Abhängigkeit des Typs, auf den der Zeiger zeigt.

Beispiel (Zeigerdifferenzen)

```
$  
#include <iostream.h>  
main() {  
    double d[10];  
    double *dp1 = d;  
    double *dp2 = dp1+1;  
    cout << "Elemente zwischen dp1 und dp2: " << dp2-dp1 << endl;           //1  
  
    cout << "Bytes zwischen dp1 und dp2: " << long(dp2) - long(dp1) << endl; //8  
}  
$
```

Zeigerarithmetik

Umwandlung Adresse in long

Kein Zeigerarithmetik, sondern long Arithmetik:

daher Differenz ist Byteanzahl zwischen dp1 und dp2

## Beispiel (strcpy)

```
$ cat strcpy.cpp
#include <iostream.h>
void strcpy(char *s, char *d) { // copy s to d
    while ( (*d = *s) != '\0') {
        s = s+1;
        d++;
    }
}
```

Zeigerarithmetik

```
void strcpy(char *s, char *d) {
    while ( (*d++ = *s++) != '\0')
        ;
}
```

```
void strcpy(char *s, char *d) {
    while ((*d++ = *s++));
}
```

korrekt, aber  
nicht gut lesbar!

## Hörsaalübung:

Realisieren Sie eine Funktion

```
char * strnCAt(char *s1, char * s2, int n)
```

Diese Stringfunktion hängt an den Zielstring s1 die durch den Quellstring s2 bezeichnete, nullterminierte Zeichenfolge an. Dabei werden maximal n Elemente kopiert. Werden mittel n mehr Zeichen angegeben, als in s2 vorhanden sind, endet die Kopie bei der Nullterminierung des Quellstrings.

Als Rückgabewert liefert die Funktion einen Zeiger auf das erste Zeichen des Zielstrings.

Beispiel:

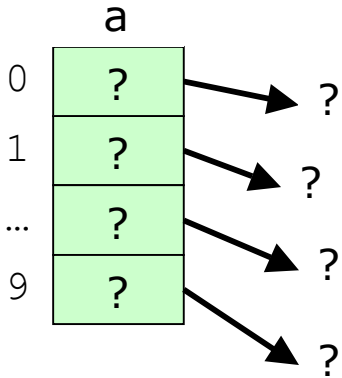
```
$ cat strcatTest.cpp
#include <iostream.h>

int main (void)
{
    char zielstring[255] = "Alle meine Entchen" ;
    char quellstring[30] = " schwimmen auf dem See," ;

    strnCAt(zielstring, quellstring, 9);
    cout << zielstring << endl;
    return 0;
}
$ strcatTest
Alle meine Entchen schwimme
$
```

## 3.2. Vektoren von Zeigern

Zeiger sind normale Datentypen. Deshalb sind auch Arrays von Zeigern möglich.

C++ Fragment	Erklärung	Veranschaulichung
<pre>int *a[10];</pre>	a ist ein Vektor von 10 Zeigern auf jeweils ein Int-Objekt.	

Die Verwendung von solchen Zeigerarrays werden wir im nächsten Abschnitt sehen, wenn wir mit dynamisch erzeugten Speicherbereichen arbeiten.

## 4. Dynamische Datenobjekte

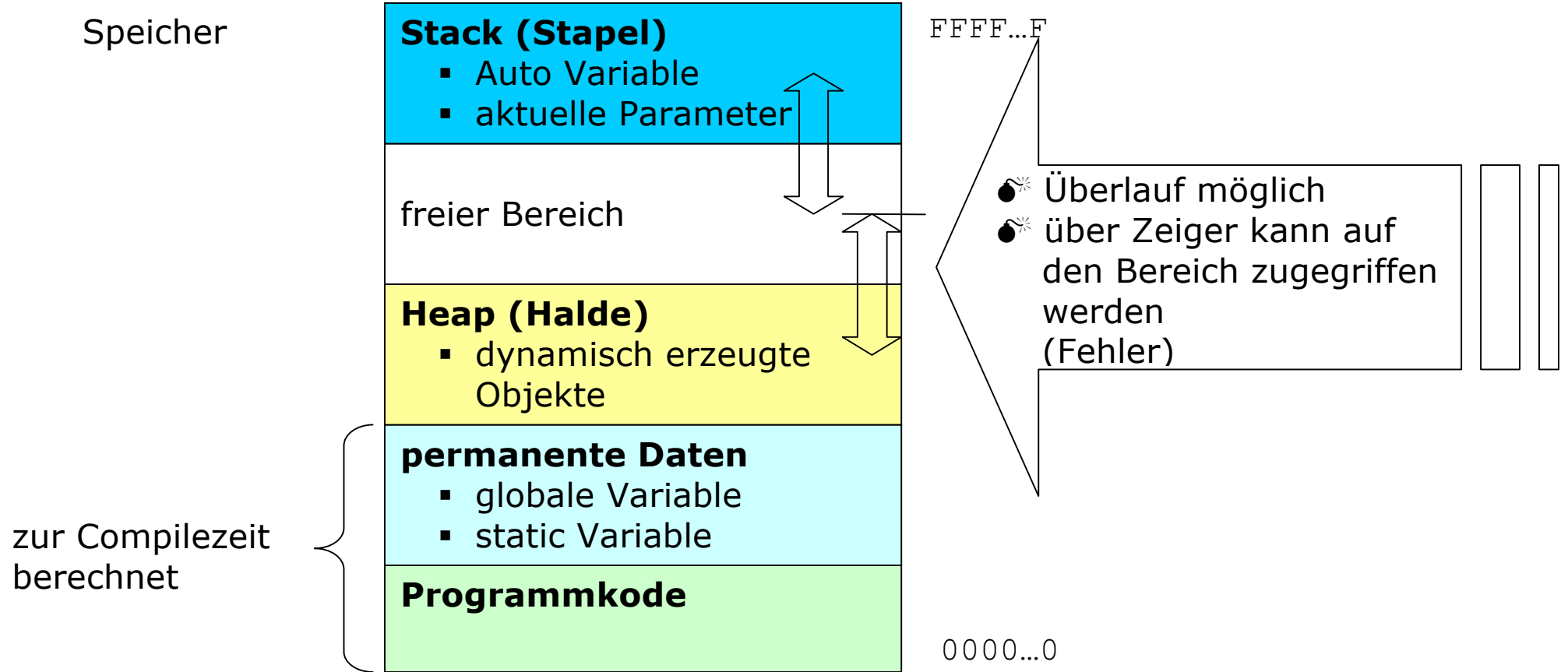
Die bisher behandelten Datentypen waren statisch – der Compiler konnte den Speicherplatzbedarf ermitteln.

In vielen Anwendungen ist der erforderliche Platzbedarf aber erst zur Laufzeit des Programms bekannt; z.B. wenn der Platz von der Anzahl der Benutzereingaben abhängt.

Eine Idee ist es, ein sehr großes Array von Datenobjekten zu deklarieren und dort die Werte zu speichern. Besser wäre es, man könnte Speicher zur Laufzeit, bei Bedarf anfordern (allokieren).

Ein Programm wird vom Betriebssystem (je nach Betriebssystem unterschiedlich) im Speicher etwa wie folgt abgelegt:

Speicher

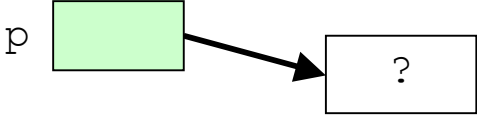
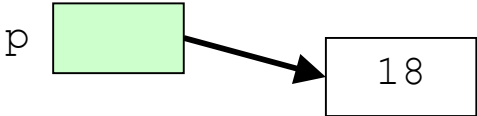
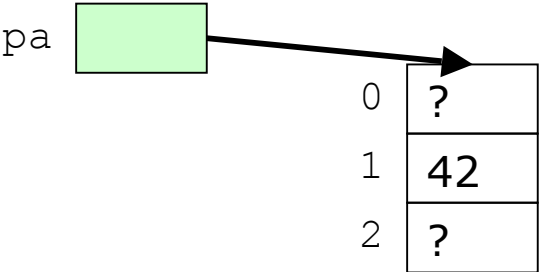


Der Zugriff auf Daten im Stack und den Bereich permanente Daten erfolgt über Namen (der Variablen), der Zugriff auf Elemente des Heap erfolgt über Zeiger.

## 4.1. Erzeugung von dynamischer Datenobjekten

Um ein **neues Element** im Heap ablegen zu könne, verwendet man den **Operator new**.

Der erforderliche Speicherplatz wird durch `new` in Abhängigkeit des Typs automatisch ermittelt.

C++ Fragment	Erklärung	Veranschaulichung
<pre>int *p; p = new int;</pre>	Ein neues Element wird auf dem Heap angelegt, es hat keinen Namen. Der Zugriff erfolgt über den Zeiger <code>p</code> . Der Wert ist undefiniert. Die Größe ist <code>sizeof(int)</code> .	 <p>A green rectangular box labeled 'p' has an arrow pointing to a white rectangular box containing a question mark '?'.</p>
<pre>int *p; p = new int; *p = 18;</pre>	Zuweisung eines Wertes.	 <p>A green rectangular box labeled 'p' has an arrow pointing to a white rectangular box containing the number '18'.</p>
<pre>int *pa; pa = new int[3]; pa[1] = 42;</pre>	Anlegen von 3 Int-Objekten. Zugriff wie auf Arrays über Zeiger und Index.	 <p>A green rectangular box labeled 'pa' has an arrow pointing to a vertical stack of three white rectangular boxes. To the left of the boxes are the indices 0, 1, and 2. The boxes contain the values '?', '42', and '?' respectively.</p>

Ein Programm, mit dem ein **Vektor von Zeigern** definiert ist, bei dem jedes Vektorelement auf ein double-Objekt zeigt ist nachfolgend angegeben:

```
$ cat vektorVonZeigern.cpp
```

```
#include <iostream.h>
```

```
main() {
```

```
    double *pd[10];
```

```
    // Vektor von Zeigern auf double;
```

```
    for (int i=0; i<10; i++) {
```

```
        cout << "Double: ";
```

```
        pd[i] = new double;
```

```
        // Komponente i zeigt auf neues Element im Heap
```

```
        cin >> *pd[i];
```

```
        // Wertzuweisung an neues Element über Zeiger
```

```
    }
```

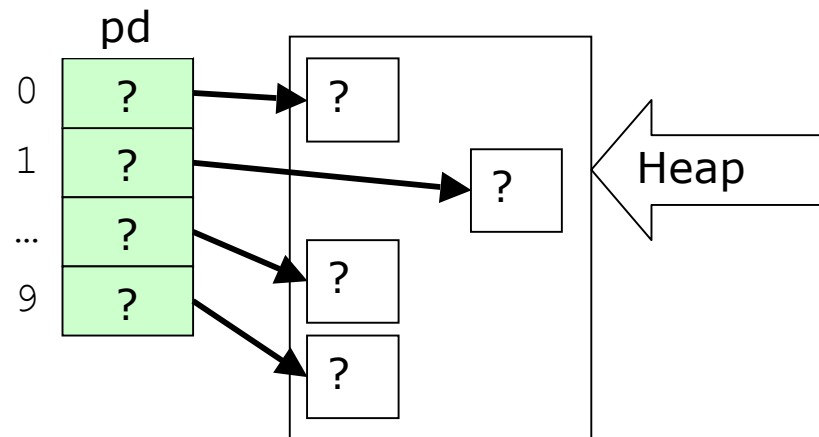
```
    for (int i=9; i>=0; i--)
```

```
        cout << *pd[i] << " ";
```

```
    cout << endl;
```

```
}
```

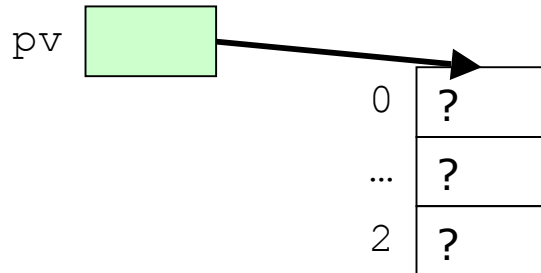
```
$
```



Ein Programm mit einem Zeiger auf ein Array von double-Werten ist:

```
$ cat zeigerAufVektor.cpp
#include <iostream.h>
main() {
    double *pv = new double[10]; // Zeiger auf Vektor von double-Werten;
    for (int i=0; i<10; i++) {
        cout << "Double: ";
        cin >> pv[i] ; // Wertzuweisung an Vektor-Element
    }

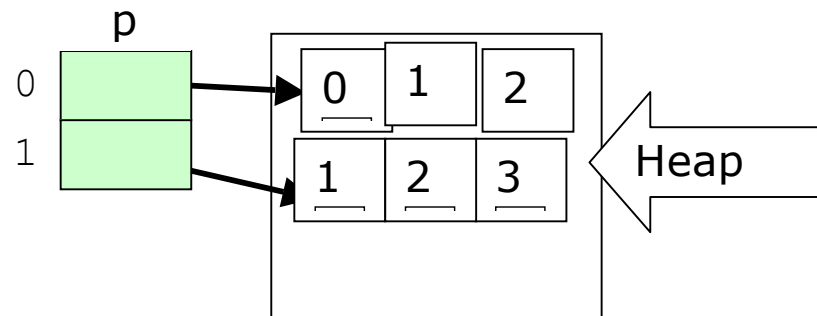
    for (int i=9; i>=0; i--)
        cout << pv[i] << " ";
    cout << endl;
}
$
```



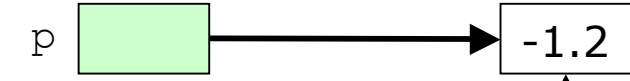
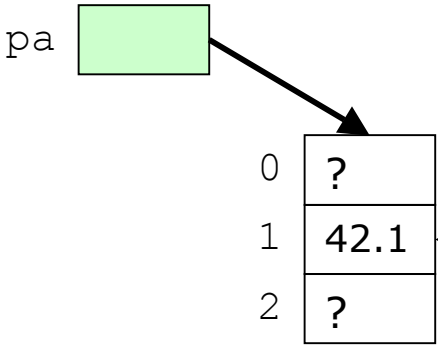
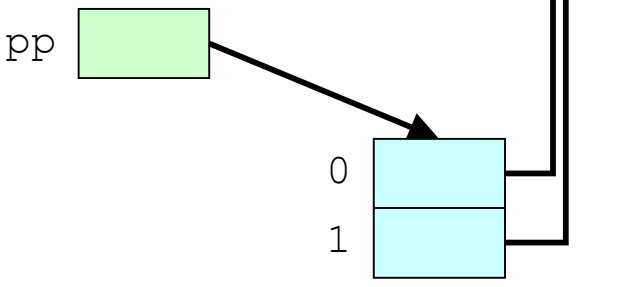
## Ein Programm mit einem Array von Zeigern auf ein Array von int-Werten ist !!!!:

```
$ cat VektorVonZeigernAufIntArray.cpp
#include <iostream>
using namespace std;
main() {
    int *p[2];    // Vektor von Zeigern auf int;
    for (int i=0; i<2; i++) {
        p[i] = new int[3];    // Komponente i zeigt auf neues Int-Array im Heap
        for (int j=0; j<3; j++)
            *(p[i]+j) = i+j;
    }

    for (int i=0; i<2; i++) {
        int *pi = p[i];
        for (int j=0; j<3; j++) {
            // cout << *(p[i]+j); // so oder
            cout << pi[j];    // so moeglich
        }
        cout << endl;
    }
}
$
```

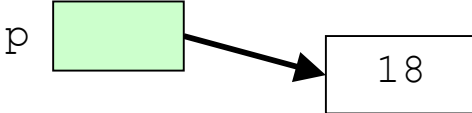
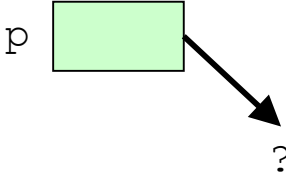


Diese „Spielarten“ können auch gemeinsam verwendet werden, um einen **Zeiger** auf einen **Vektor von Zeigern** auf double-Werte zu haben:

C++ Fragment	Erklärung	Veranschaulichung
<pre>double *p = new double; *p= -1.2;</pre>	<p>Anlegen von double und Zeiger darauf.</p>	 <p>A green box labeled 'p' has an arrow pointing to a white box containing the value '-1.2'.</p>
<pre>double *pa; pa = new double[3]; pa[1] = 42.1;</pre>	<p>Anlegen von 3 double-Objekten.</p>	 <p>A green box labeled 'pa' has an arrow pointing to a white box representing an array. The array has three rows: the first row contains '?', the second row contains '42.1', and the third row contains '?'. The rows are indexed 0, 1, and 2 from top to bottom.</p>
<pre>double **pp = new double*[2];  pp[1] = p;  pp[0] = &amp;pa[1]</pre>	<p>= (double *) *pp, also Zeiger auf Zeiger;</p> <p>pp[1] zeigt auf das Objekt, auf das p zeigt.</p> <p>pp[0] zeigt auf Komponente 1 des Arrays im Heap.</p>	 <p>A green box labeled 'pp' has an arrow pointing to a light blue box representing an array of two pointers. The array has two rows: the first row is indexed 0 and the second row is indexed 1. An arrow from the first row points to the array [?, 42.1, ?]. An arrow from the second row points to the box containing '-1.2'.</p>

## 4.2. Freigabe von dynamisch erzeugten Datenobjekten

Der Operator `delete` kann verwendet werden, um ein vorher mit `new` erzeugtes Objekt wieder freizugeben, damit er erneut allokiert werden kann.

C++ Fragment	Erklärung	Veranschaulichung
<pre>int *p; p = new int; *p = 18;</pre>		 <p>A diagram illustrating the state of memory. On the left, a green rectangular box labeled 'p' represents a pointer variable. An arrow points from this box to a white rectangular box on the right containing the number '18', representing the dynamically allocated memory containing the value 18.</p>
<pre>int *p; p = new int; *p = 18;  delete p;</pre>	Freigabe, auf das Element kann man nicht mehr zugreifen.	 <p>A diagram illustrating the state of memory after the <code>delete</code> operation. On the left, a green rectangular box labeled 'p' represents the pointer variable. An arrow points from this box to a question mark '?' on the right, indicating that the memory has been freed and is no longer accessible.</p>

Ein Array von Datenobjekten wird mit `delete [] pa;` freigegeben.

Einige Besonderheiten sind bzgl. `delete` zu beachten:

- Dynamisch allokiertes Speicher, auf den kein Zeiger mehr zeigt, ist **verloren!**
- C++ **detektiert nicht**, ob ein Speicher noch benutzt wird (wg. Laufzeiteffizienz) [dies ist in Java anders: die JVM überwacht den Heap und führt automatische ein Garbage Collection durch].

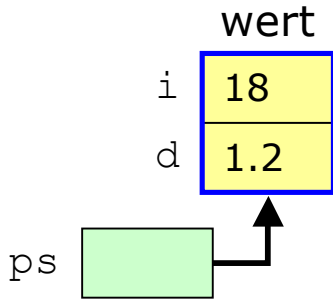
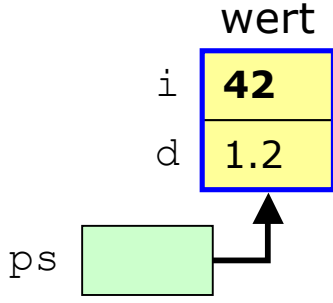
- `delete` darf ausschließlich auf Objekte angewendet werden, die mit `new` erzeugt wurden.
- `delete` auf einen NULL Zeiger ist wirkungslos.
- Mit `new` erzeugte Datenobjekte unterliegen nicht den Gültigkeitsregeln für Variablen –
- sie existieren solange bis sie mit `delete` gelöscht werden.

### 4.3.Rekursive dynamische Strukturen

Zeiger sind nicht nur auf einfache Objekte möglich. Ein Zeiger kann auch auf eine Struktur zeigen. Da ein Element der Struktur selbst wieder ein Zeiger sein kann, entsteht eine rekursive Datenstruktur.

Wenn ein Zeiger auf eine Struktur zeigt, kann eine Komponente selektiert werden durch Verwendung des Selektion- und des Dereferenzierungsoperators:

C++ Fragment	Erklärung	Veranschaulichung
<pre>struct tupel {     int i;     double d; } wert; wert.i = 18; wert.d = 1.2</pre>	<p>Definition einer Variable <code>wert</code>, die eine Struktur von einem <code>int</code> und einem <code>double</code> ist.</p>	<pre>      wert i    18 d    1.2</pre>

<pre>struct tupel *ps; ps = &amp;wert;</pre>	<p>Zeiger auf Struktur <code>tupel</code>;  <code>ps</code> zeigt auf Variable <code>wert</code>.</p>	
<pre>(*ps).i = 42; cout &lt;&lt; ps-&gt;i;</pre>	<p>Komponente <code>i</code> der Struktur selektieren, auf die <code>ps</code> zeigt und den Wert <code>42</code> zuweisen.  Kurzschreibweise für <code>(*ps).i</code> ist:  <b><code>ps-&gt;i</code></b></p> <p>Ausgabe: 42</p>	

### Achtung:

Auch hier ist der Rang der Operatoren wichtig:

`++ps->i;`      Entspricht `++(ps->i)` also wird die Komponente `i` inkrementiert.

`(++ps)->i`      Inkrementiert `ps`, zeigt also auf anders Objekt, dann wird die Komponente `i` selektiert!

### 4.3.1. Verkettete Listen

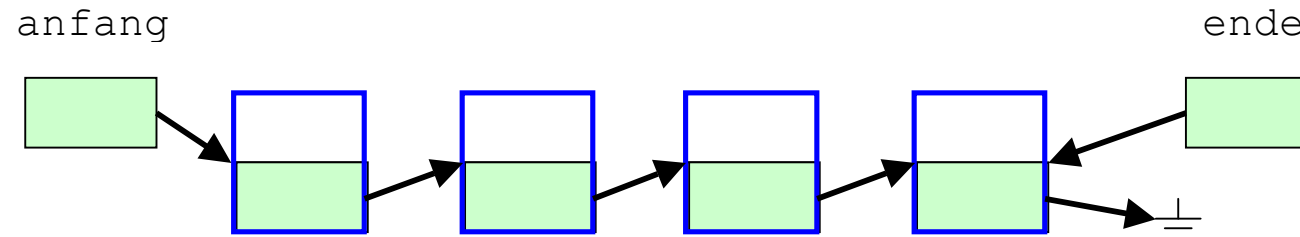
Als erstes Anwendungsbeispiel werden **verkettete Listen** demonstriert.

Eine verkettete Liste ist eine rekursive dynamische Struktur, bei dem ein Strukturelement auf ein Datenobjekt des gleichen Typs zeigt.

D.h. ein Knoten hat die Form:

C++ Fragment	Veranschaulichung
<pre>struct knoten {     int info;     struct knoten *next; };</pre>	

Damit lassen sich dann Listenelemente verketteten:



Wir werden nun Operationen auf der verketteten Liste implementieren:

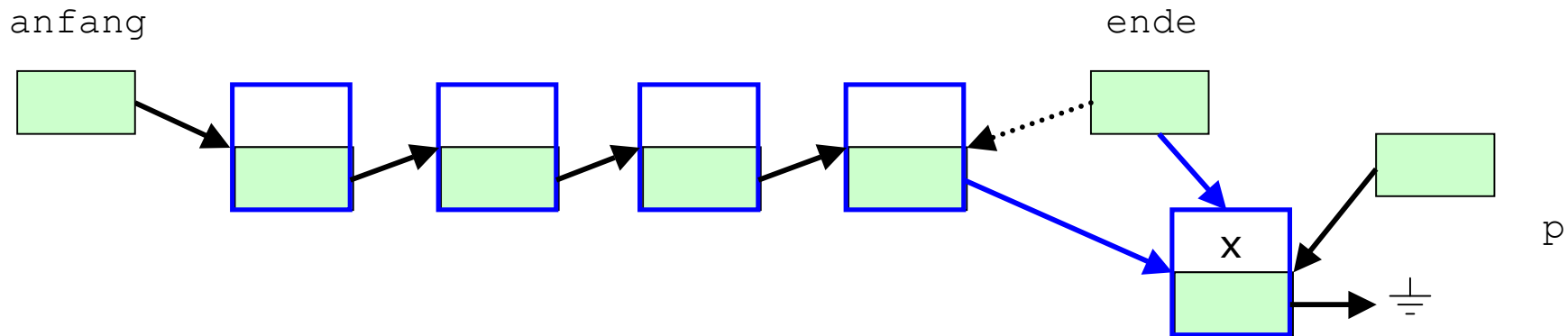
- `void einfuegen(int x)`  
Ein **neues Element** mit Infowert  $x$  soll immer am **Ende** der Liste eingefügt werden.
- `void ausgeben()`  
Die Liste wird vom Anfang an bis zum Ende durchlaufen und der Info-Teil jeden Knotens wird ausgegeben.
- `void loeschen(struct knoten *p)`  
Das Element, auf das  $p$  zeigt soll gelöscht werden.
- `struct knoten * suchen(int x)`  
Das erste Element mit Infowert  $x$  soll gesucht werden Wenn kein solches Element in der Liste ist, soll Ergebnis `NULL` sein.

Die Struktur und die beiden Zeiger werden global definiert.

```
$ cat verketteteListe.cpp
#include <iostream.h>
struct knoten {
    int info;
    struct knoten *next;
};

struct knoten *anfang, *ende; // Zeiger auf Anfang und Ende der Liste
```

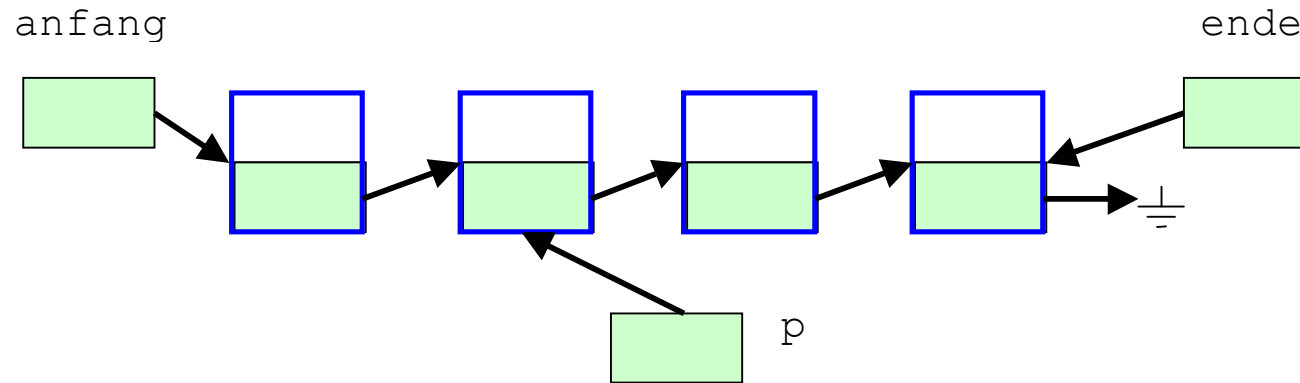
## Einfügen:



```
void einfuegen(int x) {  
    struct knoten *p = new knoten;  
    p->info = x;  
    p->next = NULL;  
    if (anfang == NULL) {  
        anfang = p;  
        ende = p;  
    } else {  
        ende->next = p;  
        ende = p;  
    }  
}
```

```
// einfuegen am Ende der Liste  
// erzeuge neuen Knoten  
// Infowert wird x  
// next-Wert wird initialisiert mit NULL  
// Leere Liste  
  
// Liste nicht leer  
// einfuegen am Ende  
// neues Ende
```

Ausgeben:



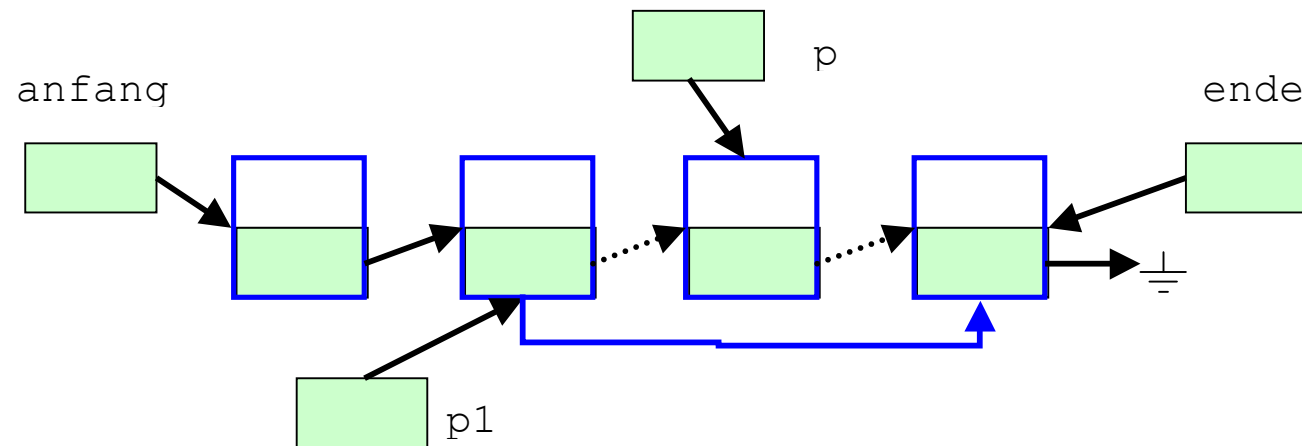
```
void ausgeben() {  
    struct knoten *p = anfang;  
    cout << "Liste: ";  
    while (p != NULL) {  
        cout << p->info << " ";  
        p = p->next;  
    }  
    cout << endl;  
}
```

// ausgeben von Anfang bis Ende  
// p zeigt auf aktuellen Knoten

// Infoteil des aktuellen Knoten ausgeben  
// p auf Folgeelement setzen

```
struct knoten * suchen(int x) { // sequentielles Suchen
    struct knoten *p=anfang;
    while (p != NULL) {
        if (p->info == x)
            return p;
        else
            p = p->next; // p auf Folgeelement setzen
    }
    return NULL; // nicht gefunden
}
```

Löschen:



```
void loeschen(struct knoten *p) {  
    struct knoten *p1;  
    p1=anfang;  
    if (p==anfang) {  
        anfang = anfang->next;           // neuer Anfang  
        return;  
    }  
    while (p1->next != p)                 // suche Vorgaenger von p  
        p1 = p1->next;  
    p1->next = p->next;                   // verkette neu  
}
```

```

main() {
    anfang = ende = NULL;           // Leere Liste
    int eingabe;
    while (true) {
        cout << "Integer (Abbruch mit negativem Wert): ";
        cin >> eingabe;
        if (eingabe<0) break;       // Beende Eingabeaufforderung
        einfuegen(eingabe);
    }
    ausgeben();

    cout << "Integer, der gelöscht werden soll: ";
    cin >> eingabe;
    struct knoten *p;
    p = suchen(eingabe);
    if (p != NULL)
        loeschen(p);
    else
        cout << eingabe << "nicht in Liste vorhanden" << endl;
    ausgeben();
}
$

```

## Hörsaalübung

1. Erweitern Sie die Funktion `loeschen`, so dass der Speicherplatz des gelöschten Knotens freigegeben wird.
2. Erweitern Sie das Programm so dass das Einfügen dafür sorgt, dass die Liste aufsteigend sortiert aufgebaut wird.

### 3. Realisieren Sie eine Funktion zur Ausgabe

```
void printReverse(struct knoten *p)
```

die die Liste ab dem Knoten `p` „rückwärts“, ohne dass die Funktion **zusätzliche** Variable verwendet (weder globale noch lokale). D.h. die in Teil 1 aufgebaute Liste wird durch den Aufruf `printRevers(anfang)` absteigend sortiert gedruckt.

### 4.3.2. Suchen in Tabellen - Hashing

Hashing ist eine Methode zum Realisieren von Mengen, bei denen Operationen wie

- insert und
- isMember

einfach und effizient implementiert werden können.

Hier wird eine Variante der Hash-Verfahren vorgestellt, die mit linear verketteten Listen arbeitet.

Bestandteile von Hashing sind:

1. Die Hash-Tabelle  $H[M]$  als Vektor von Zeigern auf linear verkettete Listen:

```
struct element {
    int info;
    struct elem *next;
} *H[M];
```

2. Die Hash-Funktion

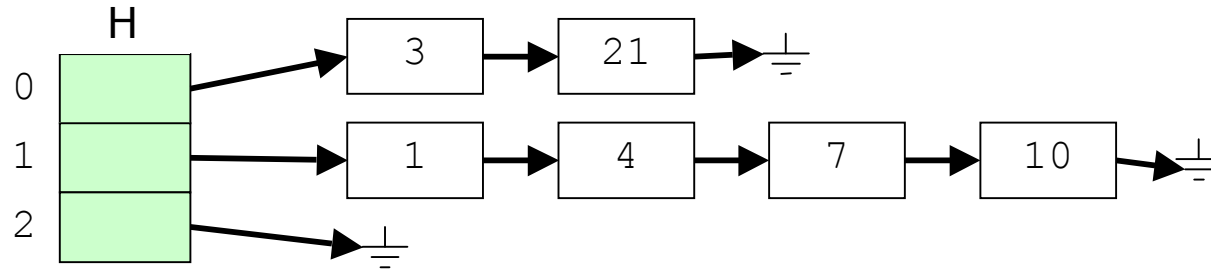
$h:U \rightarrow \{0,1,2,..M-1\}$  wobei  $U$  das Universum ist.

Eine Menge  $S = \{x_1, x_2, \dots, x_n\}$  wird dann durch  $M$  linear verkettete Listen dargestellt, wobei die  $i$ -te Liste alle Elemente  $x_i$  enthält mit  $h(x_i) = i$ .

Beispiel:

$$S = \{1, 3, 4, 7, 10, 21\}$$

$$h(x) = x \bmod 3$$



D.h. die Hash-Funktion soll die Elemente möglichst gleichmäßig auf die einzelnen verketteten Listen verteilen, um kurze Listen zum Einfügen (`insert`) und Suchen (`isMember`) zu haben.

Wir wollen nun das o.a. Hash-Verfahren realisieren.

```

$ cat hash.cpp
#include <iostream.h>
const int M=3;
struct element {
    int value;
    struct element *next;
} *H[M];           // hash table

int h(int x) {     // hash function
    return x%M;
}

void insert (int x) { // insert in front of linked list, determined by hash function
    struct element *p = new element;
    p->value = x;
    p->next = NULL;
    int i = h(x); // derive hash value to determine linked list
    if (H[i] == NULL) // create new linked list
        H[i] = p;
    else { // insert in front of linked list H[h(i)]
        p->next = H[i];
        H[i] = p;
    }
}

```

```

bool isMember(int x) {
    struct element *p;
    p = H[h(x)];          // determine linked list
    while (p != NULL && p->value != x)          // search through the list
        p = p->next;
    if (p != NULL)       // found x
        return (p->value == x);
    else                 // not found
        return false;
}

void print() {          // print out hash table with lists
    for (int i=0; i<M; i++) {
        cout << "H[" << i << "] = " ;
        struct element *p = H[h(i)];
        while (p != NULL) {
            cout << p->value << " ";
            p = p->next;
        }
        cout << endl;
    }
}

```

```

main() {
    int input;
    while (true) {
        cout << "Integer (abort with neg. value): ";
        cin >> input;
        if (input<0) break;
        insert(input);
    }

    print();

    cout << "Integer to search: ";
    cin >> input;
    if (isMember(input))
        cout << input <<" found!" <<
    else
        cout << input <<" NOT found!"
}
$

```

```

$ ./hash
Integer (abort with neg. value): 10
Integer (abort with neg. value): 3
Integer (abort with neg. value): 1
Integer (abort with neg. value): 4
Integer (abort with neg. value): 21
Integer (abort with neg. value): 7
Integer (abort with neg. value): 11
Integer (abort with neg. value): -1
H[0] = 21 3
H[1] = 7 4 1 10
H[2] = 11
Integer to search: 21
21 found!
$

```

## Hörsaalübung

Realisieren Sie eine Funktionen zum Löschen „delete(x)“ eines Elementes x der Menge im o.a. Hash-Programm.

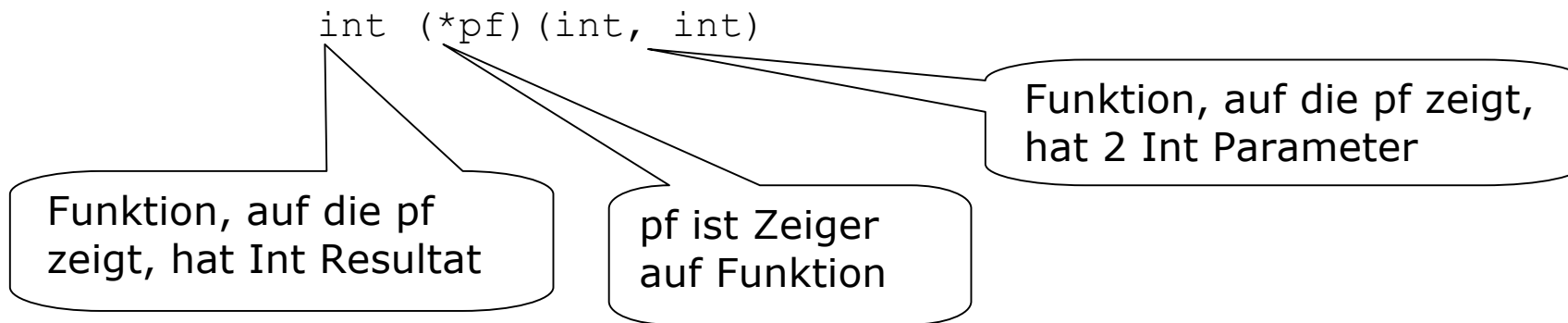
## 5. Zeiger auf Funktionen

Obwohl eine Funktion keine Variable ist, existiert die Möglichkeit, einen Zeiger auf eine Funktion zu definieren. Damit ist es möglich, erst zur Laufzeit eines Programms zu bestimmen, welche Funktion ausgeführt wird (engl. *late binding*, *dynamic binding*).

Dies wird die Grundlage für *Polymorphismus* im objektorientierten C++ im nächsten Semester sein.

Die Verwendung wird am Beispiel gezeigt. Darin wird eine Funktion zum Addieren und eine zum Subtrahieren von zwei Integern verwendet und zur Laufzeit entschieden, welche der beiden Funktionen aufgerufen wird.

Dabei wird **ein Zeiger auf eine Funktion** mit **zwei Int-Parametern**, die **ein Int-Resultat** liefert, definiert durch:



Die Klammerrung (*\*pf*) ist notwendig, ansonsten wäre `int pf (int, int)` eine **Funktionsdeklaration**.

## Beispiel:

```
$ cat addOderSub.cpp
#include <iostream.h>

int add(int, int);           // Funktionsdeklarationen
int sub(int, int);

void main() {
    int zahl1, zahl2;       // Variable für Operanden
    int resultat;
    char operation;        // Variable für Operation

    int (*pf)(int, int);   // Pointer auf Funktion mit 2 Parametern, die Int-Resultat liefert

    cout << "2 Integer: "; // Benutzerdialog
    cin >> zahl1 >> zahl2;
    cout << "+ oder -?: ";
    cin >> operation;
```

```

switch (operation) {
    case '+': pf = add; // pf zeigt auf Funktion add
              break;
    case '-': pf = sub; // pf zeigt auf Funktion sub
              break;
    default: cout << "Ungültige Operation!";
             pf = NULL;
}

if (pf != NULL) {
    resultat = (*pf)(zahl1, zahl2); // Aufruf der Funktion, auf die pf zeigt
    cout << "Ergebnis: " << resultat << endl;
}
}

int add(int x, int y) {
    return x + y;
}

int sub(int x, int y) {
    return x - y;
}
$

```

```

$ addOderSub
2 Integer: 2 5
+ oder -?: +
Ergebnis: 7
$
$ addOderSub.exe
2 Integer: 2 5
+ oder -?: -
Ergebnis: -3
$

```

Ein **Funktionszeiger** kann auch als **Parameter** auftauchen. Damit kann das o.a. Beispiel modifiziert werden, in dem eine Funktion `operation` definiert wird, die sowohl die Operation als auch die Operanden als Parameter erhält:

```
int operation(int (*op)(int, int), int operand1, int operand2) {  
    return (*op)(operand1, operand2);  
}
```

Programm:

```

$ cat addOderSub02.cpp
#include <iostream.h>
int add(int, int);           // Funktionsdeklarationen
int sub(int, int);

int operation(int (*op)(int,int), int, int);

void main() {
    int zahl1, zahl2;       // Variable für Operanden
    int resultat;
    char auswahl;          // Variable für Operation

    int (*pf)(int, int);    // Pointer auf Funktion mit 2 Parametern, die Int-Resultat liefert

    cout << "2 Integer: ";  // Benutzerdialog
    cin >> zahl1 >> zahl2;
    cout << "+ oder -?: ";
    cin >> auswahl;
}

```

```

switch (auswahl) {
    case '+': pf = add;           // pf zeigt auf Funktion add
              break;
    case '-': pf = sub;         // pf zeigt auf Funktion sub
              break;
    default: cout << "Ungültige Operation!";
             pf = NULL;
}

if (pf != NULL) {
    resultat = operation(pf, zahl1, zahl2); // Aufruf Funktion operation,
    cout << "Ergebnis: " << resultat << endl; // mit der Funktion, auf die pf zeigt
}

int operation(int (*op)(int, int), int operand1, int operand2) {
    return (*op)(operand1, operand2);
}

int add(int x, int y) {
    return x + y;
}

int sub(int x, int y) {
    return x - y;
}

```

## 6. Typedef

Komplexe Deklarationen sind (wenn möglich) zu vermeiden; manchmal sind sie aber nicht zu umgehen.

Durch diese Deklaration

```
char **(*seltsam) (double, int) [3];
```

wird ein Zeiger (`seltsam`) auf eine Funktion mit `double` und `int` Parametern, die einen Zeiger auf ein Array von Zeigern auf `char` zurückgibt, definiert.

Diese Deklaration führt zu schwer lesbarem Code.

Um solch komplexe Deklarationen zu vereinfachen, existiert die Möglichkeit, Namen zu definieren.

Mit

```
typedef bestehenderTyp neuerName ;
```

kann dem bestehenden Typ ein neuer Name gegeben werden.

Es wird **kein** neuer Typ definiert, sondern nur ein neuer Name eingeführt.

## Beispiel:

```
typedef float real;
main() {
    real x = 1.2;
    cout << x << endl;
}
```

Komplexe Deklarationen werden durch `typedef` besser lesbar, weil neue Typnamen zur Strukturierung verwendet werden können:

```
typedef char* ArrayVon3CharZeigern [3];
```

Damit kann man dann z.B. eine Variable `p` definieren:

```
ArrayVon3CharZeigern p;
```

entspricht wg. typedef dann

```
char *p[3]
```

Durch

```
typedef ArrayVon3CharZeigern *ZeigerAufArrayVon3CharZeigern;
```

haben wird nun den Namen `ZeigerAufArrayVon3CharZeigern` definiert.

Damit lässt sich nun die unlesbare Deklaration insgesamt definieren als

```
typedef char* ArrayVon3CharZeigern [3];
```

```
typedef ArrayVon3CharZeigern *ZeigerAufArrayVon3CharZeigern;
```

```
ZeigerAufArrayVon3CharZeigern (*seltsam)(double,int);
```

## 7. Argumente aus der Kommandozeile

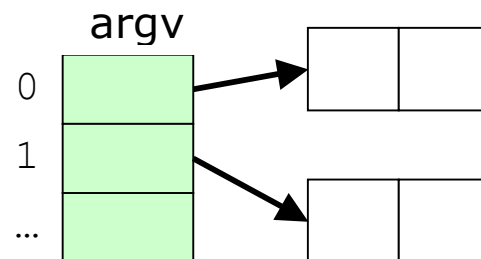
Bisher haben wir die Funktion `main` immer ohne Parameter betrachtet. Als formale Parameter der Funktion `main` sind zumindest zwei in jeder C++ Implementierung vorhanden. Die meisten Compilerhersteller erlauben aber drei Parameter.

Die Funktion `main` wird beim Aufruf eines Programms von der Konsole mit den aktuellen Parametern versorgt – man gibt also beim die Parameter von der Kommandozeile aus mit.

Die Funktion `main` ist definiert als:

```
int main( int argc, char* argv[] ) {...}
```

Anzahl Parameter  
inklusive Programmname



Zeichenkette, enthält  
Programmname

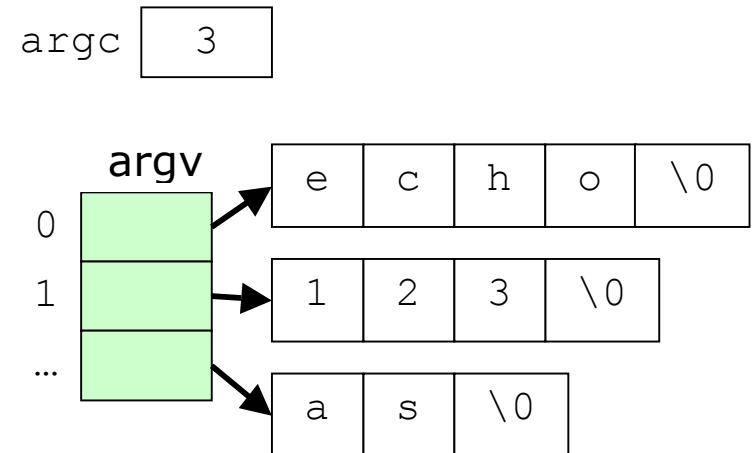
Zeichenkette, enthält  
1. Parameter

Wird ein C++ Programm z.B. mit 2 Parametern aufgerufen, so ist `argc==3`, und `argv[1]`, ... `argv[2]` sind die Parameter.

## Beispiel (Echo der Parameter):

```
$ cat echo.cpp
#include <iostream.h>
int main(int argc, char* argv[]) {
    cout << "Anzahl Argumente: "
         << argc << endl;
    for (int i=0; i<argc; i++)
        cout << i << ": " << argv[i] << endl;
}
$
$ echo 123 as
Anzahl Argumente: 3
0: echo
1: 123
2: as
$
```

Daten in main



## Was druckt das folgende Programm?

```
$ cat UnixEcho.cpp
#include <iostream.h>
int main(int argc, char* argv[]) {
    argv++;
    while ( *argv)
        cout << *argv++ << " ";
    cout << endl;
}
$
```

Der dritte Parameter von `main` zeigt auf die **Aufrufumgebung** (=Umgebungsvariable mit Wert) eines Programms:

```
$ cat env.cpp
#include <iostream.h>
int main(int argc, char* argv[], char *env[]) {
    while ( *env)
        cout << *env++ << endl;
}
$
```

```
$ env
PROCESSOR_ARCHITECTURE=x86
TEMP=/temp
LOGNAME=Alois Schütte
MAKE_MODE=unix
HOME=/home/Alois Schütte
PATH=./usr/local/bin:/usr/bin
...:/bin:./usr/local/bin:/usr/b
in:/bin:/cygdrive/c/WINDOWS/sy
stem32:/cygdrive/c/WINDOWS:/
$
```

## Hörsaalübung

Schreiben Sie ein Programm mit Namen `tr`, das von der Standardeingabe liest und die gelesenen Zeichen auf der Standardausgabe ausgibt. `tr` soll zwei Aufrufparameter haben:

`tr` `suchZeichen` `ersetzZeichen`

Dabei sollen alle Auftreten von `suchZeichen` in der Eingabe durch `ersetzZeichen` in der Ausgabe ersetzt werden.

Beispiel:

```
as@hal4:tr> tr y i  
Hallo Jenny  
Hallo Jenni  
as@hal4:tr>
```