Konstanten

`const` als Deklarationsangabe bewirkt die Vereinbarung des Namens als Konstante. Allerdings sind bei C-Vektoren die Komponenten und bei Zeigern die Objekte, auf die verwiesen wird, konstant.

Ein solcher Name kann zugleich mit einem Wert initialisiert werden (oder als `extern` vereinbart sein) und darf später nicht verändert werden.

Bei Zeigertypen gilt: Vereinbart T D einen Namen als "Datenstruktur aus T", so vereinbart T `* const` D diesen Namen als "Datenstruktur aus konstantem Zeiger auf T".

Daher können Deklarationen wieder von innen nach außen gelesen werden:

`const int N`	Integerkonstante
`const char s[N]`	`N`-Vektor aus konstanten `char`
`const char *p`	Zeiger auf konstantes `char`
`char * const q`	konstanter Zeiger auf `char`
`const char * const r`	konstanter Zeiger auf konstantes `char`

Bei Initialisierungen (und daher auch bei Parameterübergaben) spielt es im Unterschied zu Zuweisungen keine Rolle, ob die linke oder rechte Seite einem konstanten Datentyp angehört. Allerdings darf ein Zeiger auf einen nichtkonstanten Datentyp *nicht* mit einem Zeiger auf einen konstanten Datentyp initialisiert werden.

Bsp.:

```
char       s[] = "satz";
const char t[] = "text";   // Initialisierung notwendig
char         u = 'x';
const char   v = 'y';      // Initialisierung notwendig
const char  *p = &v;
char * const q = &u;       // Initialisierung notwendig
const char * const r = &v; // Initialisierung notwendig
char *w = &v;              // unzulaessig
```

Zuweisungen an Konstanten sind natürlich nicht zulässig.

Fortsetzung Bsp.:

```
s[3] = 't';
t[2] = 's'; // unzulaessig
*p   = 'n'; // unzulaessig
 p   =  r ;
*q   = 'm';
 q   =  r ; // unzulaessig
```

Bsp.: *Übergabe einer C-Zeichenkette*

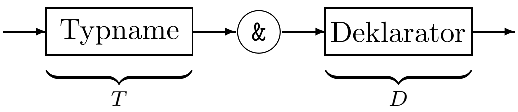
```
      :
void message(char p[])    // besser: void message(const char p[]);
{ cout << "Meldung: " << p << endl; }

int main()
{
  message("Geht das ?");
  return 0;
}
```

Referenzen

Eine Referenz ist ein weiterer Name für eine Variable, die bei der Initialisierung angegeben werden muss (und später nicht mehr geändert werden kann).
Referenzen auf Konstanten können auch mit konstanten Werten initialisiert werden, in diesem Fall wird eine Hilfsvariable angelegt.
C-Vektoren von Referenzen, Zeiger auf Referenzen und Referenzen von Referenzen können nicht gebildet werden.

Vereinbarungssyntax



Vereinbart $T\ D$ einen Namen als "Datenstruktur aus T", so vereinbart T `&` D diesen Namen als "Datenstruktur aus Referenz auf T".

Deshalb können Deklarationen von innen nach außen gelesen werden.

Die der üblicherweise der C++-Syntax zugrundeliegende Idee, dass Vereinbarungen und Ausdrücke die gleiche Gestalt haben, überträgt sich nicht auf Referenzen.

Bsp.:

```
  int  i=5, *ip=&i;
  int& j=i, *jp;  // int& j=5 unzulaessig
  const int& k=4, *kp;
  i++;   // i==6, j==6
  j++;   // i==7, j==7
//k++;      unzulaessig
  j=8;   // i==8, j==8
  jp=&j;
  kp=&k;
  cout << "i="  << i  << " j="  << j  << " k="  << k  << endl
       << "ip=" << ip << " jp=" << jp << " kp=" << kp << endl;
```

Ausgabe:

```
i=8 j=8 k=4
ip=0x7fffef41c650 jp=0x7fffef41c650 kp=0x7fffef41c654
```

Referenzen werden hauptsächlich als (konstante) Referenzparameter und als Funktionswerte im Zusammenhang mit dem Überladen von Operatoren benutzt.

Das Programmiersprachenkonstrukt Referenz lässt sich über konstante Zeiger realisieren.

Zeigerarithmetik

Hat in einem (Teil)ausdruck ein Datenobjekt den Datentyp "C-Vektor aus Typ", so wird dieser automatisch umgewandelt in "Zeiger auf Typ" und der Wert des Datenobjekts ist der Zeiger auf das erste Element des C-Vektors (d.h. die Komponente zum Index 0).

Ausnahmen:
1. Der C-Vektor ist Operand von `sizeof`[1], `&`, `++`, `--`.
2. Der C-Vektor ist linker Operand einer Zuweisung oder Initialisierung.

Beispiel:

```
double a[10], *p;
a=p;       // unzulaessig
p=a;       // zulaessig
p=&a[0];   // aequivalent zur vorigen Zeile
p=&a[3];
p=a+3;     // aequivalent zur vorigen Zeile
```

Zeigt ein Zeiger p auf ein bestimmtes Element eines C-Vektors, dann zeigt $p+1$ auf die folgende Vektorkomponente. (Sogar zulässig, falls der Zeiger auf die letzte Vektorkomponente verweist, allerdings ist die Bildung des Inhaltsoperators dann nicht erlaubt.)
Entsprechend wird $p \pm i$ gebildet.

Für einen Zeiger p und ganzzahliges i sind $p[i]$ und $*(p+i)$ gleichbedeutend, was sogar als $i[p]$ geschrieben werden darf.

Inkrement- und Dekrementoperatoren sind auch auf Zeiger anwendbar.

Beispiel: Vorbesetzen eines C-Vektors mit $a_i = i$

```
double a[N],*p;
p=a;
for (int i=0;i<N;i++) *p++=i;
```

Falsch wäre `*a++=i`, weil die Adresse der Komponente 0 eines C-Vektors eine Konstante ist, die beim Übersetzen festgelegt wird.
Zu beachten ist auch, daß mit der Definition von `p` zwar Speicherplatz z.B. für die Adresse einer double-Variable reserviert wird, nicht aber Speicherplatz für die Variable selbst.

Beispiel:

```
int *ip;
*ip=4;          /* falsch */

int *ip,i
ip=&i; *ip=4;   /* richtig */
```

Auch bei mehrdimensionalen Feldern findet die Umwandlung von C-Vektoren in Zeiger statt, z.B.:

```
double a[M][N];
a[i][j]  ≡  *(a[i]+j)  ≡  *(*(a+i)+j)
```

Allerdings wird `a` wegen Regel 1 oben lediglich in `&a[0]`, aber nicht weiter, umgewandelt.

[1] *Beispiel zur Verwendung von sizeof:*

```
double a[] = {1.0,0.0,3.0,7.0,-3.0};
cout << "Vektor a hat " << sizeof a/sizeof a[0] << "Komponenten" << endl;
```

Übergabe von C-Vektoren

Formale Parameter einer Funktion vom Datentyp "C-Vektor aus Typ" werden in "Zeiger auf Typ" umgewandelt.

Bsp.: Skalarprodukt

```
double skalar(double x[], double y[], int n)
{ double s=0;
  for(int i=0; i<n; i++) s+=x[i]*y[i];
  return s;
}
```

ist äquivalent zu

```
double skalar(double *x, double *y, int n);
{                  ...                      }
```

Aufruf im Hauptprogramm:

```
int main()
{ double a[5],b[5],
     :
  erg=skalar(a,b,5);
     :
}
```

Problematisch ist die Matrixübergabe, da die Umwandlung C-Vektor→Zeiger nicht mehrfach durchgeführt wird.

Bsp.: Kopf und Aufruf einer Funktion zum Lösen eines linearen Gleichungssystems

```
const int M=20,N=30;  // Unbrauchbar fuer Verwendung in einer Programmbibliothek!

bool lingl(double (*a)[N], double *b, double *x, int m, int n)
{                 ...                                        }

int main()
{
  double a[M][N],b[M],x[N];
     :
  lingl(a,b,x,m,n);
     :
}
```

Abhilfe:

```
bool lingl(double *ap[], double *b, double *x, int m, int n)
{                 ...                                      }

int main()
{
  const int M=20,N=30;

  double a[M][N],*ap[M],b[M],x[N];
     :
  for(int i=0; i<m; i++) ap[i]=a[i];
     :
  lingl(ap,b,x,m,n);
     :
}
```

Innerhalb der Funktion `lingl` kann `ap[i][j]` zum Zugriff auf `a[i][j]` verwendet werden.