```
void clear(int size, int *array, int value=0){
  for (int i=0; i < size; i++ )
     array[i] = value;
}
```

Ein Defaultargument muß einen konstanten Wert besitzen, der zur Compilezeit feststeht und eindeutig ist. Das bedeutet, daß dieser in jeder Übersetzungseinheit nur einmal spezifiziert werden darf. Wenn sichergestellt ist (z.B. mit include guards im header-file), daß der Compiler nur eine Deklaration der Funktion sieht, dann ist der beste Platz, um den Defaultwert festzulegen, in einer Funktionsdeklaration in einem header-file. Ansonsten sollte dieser Wert bei der Definition der Funktion festgelegt werden.

2.5.6 Überladen von Funktionen

Im Gegensatz zu C und einigen anderen Sprachen sind in C++ die Argumenttypen zusätzlich zum Funktionsnamen relevant bei der Auswahl der aufzurufenden Funktion. Es ist möglich, eine `print`-Funktion für `int` zu haben, eine für `double` und eine weitere für `struct Date` {...}. Die Mehrfachverwendung des Funktionsnamens erlaubt, diesen von Typinformation freizuhalten und sich auf die Funktion zu beschränken. Als C-Gegenbeispiel können hier `sprintf` und `fprintf` dienen, in denen `f` und `s` den Typ des ersten Argumentes angeben. Diese Häßlichkeit (die darüberhinaus noch zu Fehlern führen kann, wenn beispielsweise der `FILE *` einmal durch `char *` ersetzt wird) wird in C++ vermieden; der Compiler wählt automatisch die richtige Funktion aus.

Man kann sich vorstellen, daß der Compiler dafür sorgt, daß gewissermaßen programmintern komplexere Namen für Funktionen verwendet werden, die neben dem Funktionsnamem auch noch die Argumente enthalten.

```
void swap(int &a, int &b){   // internal name e.g. swap_int_int
  int tmp = a;
  a = b; b = tmp;
}

void swap(double &a, double &b){     // swap_double_double
  double tmp = a;
  a = b; b = tmp;
}

void f(){
  double a, b;
  int    i, j;
```

```
    // ...
    swap(a, b);  // double version
    swap(i, j);  // int    version
  }
```

Um zu verhindern, daß diese internen Namen auch für Funktionen erzeugt werden, die aus C-Bibliotheken geladen werden, müssen diese für den Compiler sichtbar als `extern "C"` deklariert werden.

```
extern "C" double sqrt(double);
```

Dieser Mechanismus erlaubt dem Compiler sowohl das Erzeugen der richtigen Namen als auch die Berücksichtigung anderer Eigenarten der Sprache, deren Prozeduren der Linker an das Programm binden soll. Bei einem Funktionsaufruf kann z.B. die Reihenfolge, in der Parameter auf dem Stack abgelegt werden eine andere als die in C++ sein.

Durch Überladen läßt sich ein ähnlicher Effekt erzielen wie mit Defaultargumenten, nämlich, daß nachträglich weitere Versionen der "gleichen" Funktion mit erweiterten Argumentlisten bereitgestellt werden können. Nachteil des Überladens zu diesem Zweck ist, daß dabei der Programmcode dupliziert wird, was zu größeren Programmen führt und größere Disziplin bei der Unterhaltung des verdoppelten Codes erfordert (Fehler (bugs) müssen in beiden Versionen entfernt werden).

Falls `@` für einen eingebauten Operator steht, so erlaubt C++ auch, diesen als `operator@()` anzusprechen. Der Ausdruck `3 * z + 5` könnte also auch, weniger intuitiv, geschrieben werden als `operator+(operator*(3,z),5)`.

In dieser Syntax lassen sich Operatoren genau wie Funktionen überladen. Dies kann dazu dienen, eine natürliche Syntax für benutzerdefinerte Typen wie z.B. Matrizen oder komplexe Zahlen einzuführen.

2.6 Zeiger, Zeiger-Feld-Dualität, und Referenzen

2.6.1 Zeiger

Vom Standpunkt des Computers betrachtet ist jede erreichbare Speicherzelle im (virtuellen) Speicher durch eine eindeutige Zahl gekennzeichnet (Teile des virtuellen Adressbereiches können auf Harddisk gehalten sein, andere können im RAM liegen, wieder andere, gegenwärtig unbenutzte, müssen vom Betriebssystem gar nicht repräsentiert sein). Auf regulären PC's liegen diese Adressen im Wertebereich normaler `int` Variablen, auf anderen

Architekturen können sie darüber hinaus gehen. C/C++ bietet Möglichkeiten diese "Namen" oder **Adressen** der Speicherstellen für Programmierzwecke nutzbar zu machen. An Übersichtlichkeit gewinnt das Programm dadurch, daß man die Adressen selbst benennen darf. Das Programmsegment

```
int    i=5;
double a=12.34;
char   *c1="hello";   // declaration of an array of char
int    j=0;
```

erzeugt beispielsweise die in Abb. 2.1 abgebildeten Speichereinträge (linke Spalte). Nur die Speichereinträge benötigen wirklich Platz im Computer die Daten enthalten. Die Adressen bzw. Namen belegen nicht mehr Platz wenn man im Programm anstelle von `i` den weniger kryptischen Namen `counter` verwenden würde.

Speichereintrag	Adresse (computerdef.)	Variablenname (benutzerdef.)
5	0xfd10	**i**
12.34	0xfd18	**a**
0xff10	0xfd20	**&c1**
0	0xfd28	**j**
	...	
h	0xff10	**c1[0]**
e	0xff11	**c1[1]**
l	0xff12	**c1[2]**
l	0xff13	**c1[3]**
o	0xff14	**c1[4]**
\0	0xff15	**c1[5]**
	...	

Abbildung 2.1: Speicherbelegung (schematisch) mit realem Speichereintrag (Links), einer willkürlich gewählten, Computer-internen Adresse (Mitte) und dem benutzerdefinierten Variablennamen (Rechts).

Im allgemeinen besteht leider keine Garantie, daß hintereinander definierte Variablen auch direkt nacheinander im Speicher liegen (Ausnahme: Felder). Die Variable `c1` vom Typ `char` enthält einen Verweis auf einen ganz anderen Speicherbereich, in dem die gespeicherten Zeichen abgelegt sind. Die Definition von `*c1` reserviert im Speicher einen Bereich, der genügt um das Wort `"hello"` abzulegen. Auf die einzelnen Elemente `'h'`, `'e'`, `'l'`, `'l'` und `'o'` kann man allerdings auch mit der Formulierung `c[0]`, `c[1]`, `c[2]`, `c[3]`, und `c[4]` direkt zugreifen. Man beachte dabei daß Numerierungen in C++ immer mit `0` beginnen.

Wenn z.B. nötig wird Daten in eine geordnete Liste schnell einzufügen oder zu löschen, so ist diese Operation nicht durch die Anordnung der Zeichen in einem derartigen linearen Feld zu erreichen. Ein *lineares Feld* ist dadurch ausgezeichnet, daß alle Daten in aufeinanderfolgenden Speicherbereichen liegen (die genaue Lokalisierung wird vom Computer übernommen). Sowohl das Einfügen als auch das Entfernen einer Zahl aus diesem Feld erfordert das Umkopieren aller "oberhalb" des modifizierten Elementes liegenden Daten, im Mittel also etwa halbsoviele Operationen wie das Feld Elemente aufweist. Dieser Aufwand ist nur bei kleinen Feldern akzeptabel. Eine Teillösung dieses Problems liegt in einer *verketteten Liste*, in der jedes Element noch die Information erhält "wo" das jeweils nächste im Speicher liegt (d.h. welches der Name der Speicherzelle ist, die den Wert des Elementes speichert). Die tatsächliche Position des Elementes spielt keine Rolle, so daß immer nur die Speicherzellennamen manipuliert werden müssen. Um zum Beispiel ein Element x zu entfernen, muß nur der Namenseintrag des Vorgängers von x in der Liste so geändert werden, daß er nicht mehr auf x sondern auf dessen Nachfolger zeigt. Kopieren anderer Daten ist nicht mehr nötig, allerdings benötigt eine verkettete Liste evtl. mehr Speicher als eine lineare Liste, da für eine Zahl sowohl ihr Wert als auch eine Adresse abgelegt werden muß.

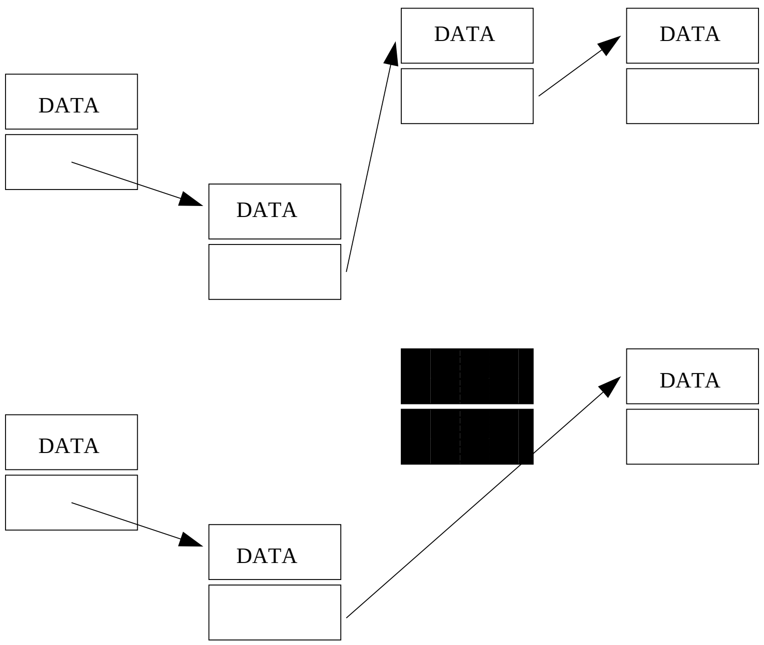

Abbildung 2.2: Benutzung von Zeigern am Beispiel einer verketteten Liste. Oben sieht man die Liste vor Entfernen eines Elementes. Unten ist es nicht in der Liste vorhanden.

Um den Umgang mit solchen Speicherzellennamen unabhängig vom jeweiligen Rechner zu gestalten, hält C/C++ den Zeiger/Pointer-Typ bereit. Ein Zeiger kann auf einen beliebigen Datentyp verweisen. Dieser Typ kann auch wieder ein Zeiger sein. Zur Deklaration benutzt man einen `*`. Um die Addresse einer Variablen zu erhalten, verwendet man den `&` Operator. Die Deklarationen im Programmsegment

```
// Declaration part
  int     i = 5;        // initialisation
  int  *p_i = &i;       // p_i is a pointer to an int,
                        //   here the variable i
  int **pp_i = (&p_i);  // pointer to pointer to int

// How to use * and & legally in the program
    *p_i = 1;            // set i to 1, p_i remains unchanged
  **pp_i = 2;            // set i to 2, p_i remains unchanged
   int j = 7;            // declare another integer j
   *pp_i = &j;           // change p_i to point on j, *p_i != i
     p_i = &i;           // let p_i point to i again
```

führen zu der in Abb. 2.3 abgebildeten Speicherbelegung.

5	0xfa10	**i**
*	...	
0xfa10	0xfc28	**pi**
*	...	
0xfc28	0xfe08	**ppi**
	...	

Abbildung 2.3: Speicherbelegung: Bedeutung der Spalten wie in Abb. 2.1.

Um den Wert einer Speicherstelle, auf die ein Zeiger verweist, zu ändern oder in einer Zuweisung oder Rechnung zu benutzen, benutzen wir ebenfalls den `*` Operator.

```
  *p_i = 27;               // i is 27 now
  *p_i = 2 * (*p_i) + 5;   // equivalent to:  i = 2 * i + 5;
```

Konstante Pointer werden durch das Setzen von `const` rechts vom Pointersymbol `*` deklariert. Sie müssen bereits bei der Deklaration initialisert werden. Jede weitere Zuweisung auf den Pointer wird vom Compiler als Fehler erkannt.

```
const int *cp_i = &i;                   // ok, i cannot be changed
                                        //     through use of cp_i
                                        //     cp_i can be changed
const char newline = '\n';
const char * const p_nl = &newline;     // const pointer to const char
                                        // note: p_nl = '0'; -> error!
```

Der `*` Operator hat damit zwei Bedeutungen. Steht er in einer Variablendeklaration, d.h. bei einem Typ, so kennzeichnet er einen Pointer. Vor einer Variablen (vom Typ Pointer) bewirkt er den Zugriff auf den Inhalt der Speicherstelle auf die der Pointer zeigt.

Analog hat der `&` Operator zwei Bedeutungen. Bei einer Variablendeklaration, d.h bei einem Typ kennzeichnet er eine Referenz. Vor einer Variablen ist es der sog. `adress of` Operator, d.h. er liefert die Adresse der Variablen.

Mit Zeigern kann Arithmetik getrieben werden; im obigen Beipiel läßt z.B. `p_i = p_i + 5` den Zeiger um 5 Speicherstellen für `int` weiterwandern. In dieser Operation benötigt der Compiler die Information, wie groß der Datentyp ist, auf den verwiesen wird, um richtig inkrementieren zu können. Ebenfalls kann die Differenz von Pointern gebildet werden, wobei das Ergebnis ein `integer` ist. Ein definiertes Ergebnis hat diese Operation jedoch nur, wenn die beiden pointer in ein und dasselbe Datenfeld verweisen, s.u.. Wenn der Wert einer Speicherstelle benötigt wird, die man durch Fortschalten des Pointers um `n` Speicherstellen erhält, so gibt es dafür eine kurze Notation mittels `[]`:

```
int p[30];          // declare a field of int
*(p_i + 25)  = 15;  // the value of the 25th entry (past the one
                    // pointed to by p_i) is set to 15
p_i[25]      = 15;  // equivalent ...
```

Die Sprache garantiert, daß 0 niemals als Adresse eines wirklichen Datenelementes vorkommt. Dieser Wert läßt sich also verwenden, um irgendwie geartete Fehler anzuzeigen, die bei einer Operation mit Pointern aufgetreten sind.

Zeiger auf void

Der Typ Zeiger auf void ist eine generische Zeigervariable. Es ist garantiert, das jeder Zeiger auf einen beliebigen Datentyp in einen Zeiger auf `void` umgewandelt werden kann, ohne daß Informationsverlust auftritt. In C wurden daher `void *` häufig dazu eingesetzt, Information an Routinen zu übergeben, die mit beliebigen Pointern arbeiten konnten (z.B. Sortierroutinen wie `quicksort`, Rückgabewert der Speicheralloziierungsroutine `malloc`). In C++ sollten in solchen Fällen `template`-Funktionen zum Einsatz kommen, die die Typsicherheit des Programmes erhalten.

2.6.2 Zeiger-Feld-Dualität

Zwischen Zeigern und Feldern besteht in C/C++ ein enger Zusammenhang (Zeiger-Feld-Dualität), der sich bereits oben in der Verwendung von `[]` zur "indirekten" Dereferenzierung von Pointern andeutete. Ein Feld `a[..]` wird intern durch den Compiler sofort in einen Pointer auf dessen erstes Element verwandelt. Der Feldzugriff wird dann vollständig nach den Regeln der Pointerarithmetik abgewickelt. Dies ist auch der tiefere Grund dafür, daß C/C++-Felder immer bei 0 beginnen (man spart dabei Rechenoperationen bei der Lokalisierung von Adressen).

```
int a[20];
int *p = a;          // ok, a is name of the field and can
                     //     be used as address to the field
int *p = &(a[0]);    // ... equivalent

p[5] = 4;            // access to a through pointer arithmetic,
                     //     a[5] is set to the value of 4
a[5] = 4;            // ... equivalent

int b[20][30];       // b is now an array of 20 arrays of
                     //     30 elements of type int
                     //     int *p = b;  error! wrong type, since
                     //     b is of type pointer to 30 int here
int *r = b[0];       // ok, r points to the first int
                     //     in the first (of 20) array(s) of 30 ints
int (*q)[30] = b;    // ok, q points to the first set of 30 int's,
                     //     q and b can now be used synonymous
q[12][15]  = 26;     // ok, set element (13,16) to the value 26
(*(q+12))[15] = 26;  // ... equivalent
*(*(q+12)+15) = 26;  // ... equivalent
b[12][15]  = 26;     // ... equivalent
```

2.6.3 Feldvariablenübergabe an Funktionen, dynamische Speicherverwaltung

Um ein Feld an eine Funktion (vgl. Abschnitt 2.5) zu übergeben, muß man einen entsprechenden Zeiger übergeben. In einer Dimension ist das einfach ein Zeiger auf das erste Datenelement, in zwei Dimensionen ein Zeiger auf ein Feld der passenden (festen) Anzahl von Elementen. Nur die erste Dimension in der Deklaration kann dabei entfallen, alle anderen müssen zur Compilezeit feststehen, damit der Compiler den für die Zeigerarithmetik nötigen Code erzeugen kann. Dies macht C-Felder sehr unpraktisch im Gebrauch, wenn variable Dimensionen erforderlich sind.

```
void f( int b[][30] )
void g( int (*q)[30] )

int  b[20][30];
int  c[10][30];
int  d[15][31];

f(b); g(b); f(c); g(c); // ok
f(d); g(d);             // errors, since second dimension not ==30
```

Wir wollen hier kurz auf die dynamische Speicherverwaltung vorgreifen und eine weitere Technik zur dynamischen Erzeugung mehrdimensionaler Felder schildern. Der `operator new` *type*`[`N`]` gibt einen Zeiger auf das erste Element eines linearen Speicherbereiches von `N` Variablen des Typs *type* zurück. Um ein zweidimensionales Feld von, sagen wir, Dimension N×M zu erzeugen, kann man zunächst einen Speicherbereich von N×M Elementen anfordern. Dann kann man ein Hilfsfeld von Pointern einrichten, die auf die Positionen der Variablen `a[i][0]` verweisen, `i` $= 0..N-1$. Ein weiterer Zeiger auf die erste Position dieses Hilfsfeldes kann syntaktisch wie ein zweidimensionales Feld behandelt werden. Dieser kann dann auch unproblematisch an Unterprogramme weitergereicht werden und erlaubt damit die Übergabe dynamisch angelegter Felder.

```
void f(int **a);        // function expecting 2D array

// ...
int  nx, ny;
std::cin >> nx >> ny;   // read dimensions
int **array;            // pointer to first element of 'secondary' array
array = new int *[nx];  // place for nx pointers to int
array[0] = new int[nx*ny]; // nx * ny elements in the entire array,
                        // array[0] is a pointer to the first ``row'' of
                        // ny elements and equals the pointer to the
                        // beginning of the entire array
for(int i=1; i < ny; i++ )
  array[i] = array[0] + i * ny;  // points to i'th segment of ny integers

if ( nx > 23 && ny > 10 )
  array[23][10] = 17;   // ok if 23 < nx and 10 < ny

f( array );             // ok, array is int **, but subscripting behaves
                        //  as if it were a 2D field
```

Diese Technik läßt sich für Felder mit beliebig vielen Indizes anwenden und führt dann zu entsprechend hohem Grad der verwendeten Zeiger.

2.6.4 Referenzen

Eine Referenzvariable etabliert einen zweiten Namen für eine bereits bestehende Variable gleichen Typs. Daher ist klar, daß eine Referenzvariable bei der Deklaration initialisiert werden muß. Danach ändern Zuweisungen nur den den Inhalt der referenzierten Variable, aber nicht die Variable selbst. Viele Compiler implementieren den zugrundeliegenden Mechanismus mit Hilfe von Pointern und wir wollen zwei Programmsegmente, die die gleiche Aufgabe mit Pointern und mit Referenzen erledigen, gegenüberstellen. Es kann jedoch nicht genug betont werden, das Referenzen keine Pointer sind und die jeweilige Implementation vollständig dem Compilerbauer überlassen ist.

```
int  i;                         |   int i;
int &i_2 = i;                   |   int * const i_2 = &i;
i_2  = 5;     // i is now 5     |   *i_2 = 5;
i = i_2 * i + 5; // i == 30     |   i = (*i_2) * i + 5;
std::cout << i_2;               |   std::cout << *i_2;
```

Man sieht, daß sich eine ähnliche Funktionalität auch mit konstanten Zeigern erreichen läßt, daß aber die Benutzung des “value of” Operators `*` entfällt. Referenzen sind daher einfacher zu verwenden als konstante Zeiger und daher auch weniger fehlerträchtig.

Es ist entscheidend, daß Referenztypen auch von Funktionen (s.d.) zurückgeliefert werden können und damit Funktionen sinnvoll auf der linken Seite von Zuweisungen stehen können.

```
int &access( int a[], int i ){ return a[i]; }

void f(){
   int a[20];
   access(a,10) = 5;  // ok, a[10] is 5
}
```

Der Sinn dieses Beipieles wird klarer, wenn man sich vorstellt, daß in `access()` Bereichsgrenzentests oder ähnliches stattfinden können.

Konstante Referenzen können als Funktionsargumente dienen, wenn die Funktion nur lesend auf die Variable zugreift. In C++ kann die Übergabe als `const &` häufig das aufwendige Kopieren einer Klassenvariable verhindern und laufzeiteffektiveren Code erzeugen.

Die Übergabe als nichtkonstante Referenz wie die Übergabe von Information per Zeiger verhindert häufig Optimierungen (Stichwort: aliasing von Variablen) und sollte nur dort verwendet werden, wo sie wirklich nötig ist. Auch wird der Compiler nicht ohne Warnung erlauben, temporär erzeugte Variablen, wie die Werte von Ausdrücken oder Zwischenergebnisse einer Rechnung, als nichtkonstante Referenzen an Funktionen zu übergeben.

2.6.5 typedef

`typedef`'s erlauben, neue Namen für bestehende Typen einzuführen, die sich exakt wie der ursprüngliche Typ verhalten.

```
typedef   int    Int32;
typedef   struct { int lower, upper } TwoInt;
```

Sie können dazu dienen, die etwas kryptische Deklarationssyntax von C++ freundlicher zu gestalten. Das folgende zeigt die Definition eines Feldes von Zeigern auf Funktionen.

```
double  (*a[20])(double);     // declaration without typedef

typedef   double DblFctDbl(double);
typedef   *DblFctDbl  Ptr_DblFctDbl;
Ptr_DblFctDbl  a[20];         // array of 20 function pointers
```

2.7 Bezugsrahmen von Bezeichnern

`Scope` bedeuted Bezugsrahmen von Variablen, Funktionen usw.

Die Sprache C++ unterscheidet fünf verschiedene Bezugsrahmen
1. Datei
2. Lokal
3. Funktion
4. Klasse
5. Namensraum

Dateibezugsrahmen Ein Projekt besteht im Normalfall aus mehreren Files. Jedes Modul besteht aus `*.cc` und `*.h`. Bei ungewollten Namenskonflikten besteht die Möglichkeit die entsprechende Funktion `static` zu deklarieren. Ansonsten die Möglichkeit mit `extern` eine "gewollten" Namenskonflikt zu kennzeichnen

Lokaler Bezugsrahmen Hier gilt, daß lokale Namen globale verdecken. Will man explizit auf globale Namen zugreifen, so verwendet man den `scope operator ::`.

```
goto label;
{
  // hier eingefuehrte Variablen sind lokal
  // dieser Block hilft evtl. dem Compiler
}
label:
```

Eine Alternative zu Variablen mit Filebezugsrahmen sind lokale Variablen in Funktionen, die static deklariert sind. Sie haben einen lokalen Bezugsrahmen, aber eine Lebensdauer, die über den Funktionsaufruf hinausgeht.

Klassenbezugsrahmen Bezeichner im Klassenbezugsrahmen werden alternativ durch `.name`, `->name` oder für `static` Variablen `Klassenname::name` angesprochen. Static Variablen existieren unabhängig von der Instanzbildung nur einmal für die ganze Klasse, d.h. alle Objekte/Instanzen einer Klasse teilen sich eine Variable oder Funktion

Namensräume Namensräume sind dazu gedacht, größere Einheiten wie z.B. die gesamte Standardbibliothek zusammenzufassen. Durch Ihre Benutzung wird vermieden, daß es bei Benutzung von mehreren großen Programmpaketen zu Problemen wegen doppelt vergebener Namen kommt. Bei der Benutzung von Teilen dieser Pakete spezifiziert man den Namen mit vorangestelltem `namespace Name{...}` oder importiert diese Teile mit `using` Direktiven.

```
namespace Name{
  ....
}
```

Durch `using` Direktiven kann man einzelne Elemente oder auch einen ganzen Namensraum "importieren", d.h. auf seine Elemente zugreifen ohne explizit anzugeben, in welchem Namensraum sie sich befinden. Allerdings kann man so nur einen Namensraum importieren.

2.8 Ein- und Ausgabe

Die Ein- und Ausgabe von C++ wurde gegenüber C vollständig überarbeitet. Dies war unter anderem deshalb notwendig, weil in C die Ein- und Ausgabefunktionen variable