Eigenschaften von C

C beinhaltet:

- Datentypkonzept
- Blockstruktur
- maschinenunabhängiges Programmieren hardwarenaher Operationen
- leistungsfähiger Mikroprozessor
- Standard Include-Dateien
- getrennte Compilierung -> Modularisierung
- Aufruf von Systemfunktionen

nicht unmittelbar in C enthalten sind:

- Operationen auf zusammengesetzte Datentypen
- Ein-/Ausgaberoutinen
- mathematische Funktionen
- parallele Prozesse
- Prozeßsynchronisation
- Koroutinen
- Garbage Collection

aber:

- C ist ein nackte Sprache
- der Sprachumfang ist relativ klein
- Compiler sind kompakt und schnell
- die fehlenden Funktionen werden durch Bibliotheks. und Betriebssystemfunktionen bereitgestellt
- case-sensitivity
- C ist formatfrei

Eigenschaften einer Programmiersprache

- Integrität: Programmberechnungen werden korrekt ausgeführt
- Klarheit: Lesbarkeit/Verständlichkeit
- Einfachheit: der Programmlösung
- Effektivität: des Compilers, der Ausführungsgeschwindigkeits, der Speicherausnutzung
- Modularität: Zerteilen eines Problems in Unteraufgaben

Definitionen

- Algorithmus: Ein Algorithmus ist eine eindeutige, endliche Beschreibung eines allgemeinen Verfahrens zur Lösung eines Problems.
- Syntax: Grammatikregeln für den Compiler
- Semantik: Bedeutung der einzelnen Anweisungen/Sprachelemente

Variablen

- Namen: Speicherzellen werden mit mnemoisch, sinnvollem Namen angesprochen
- Speicherklasse: Ortsangabe für Variable
- Datentyp: Bestimmt die Art des Inhaltes
- Inhalt: Daten, die in der Variablen gespeichert werden
- keine reservierten Wortsymbole, keine Leerzeichen, Beginn mit Buchstaben
- gleichnamige Variable überschattet im inneren Block eine andere
- Deklaration: Angabe über Datentyp und Speicherklasse
- Definition: Erzeugung; Reservierung von konkretem Speicher
- Vereinbarung: Definition und Deklaration

Datentypen

- **char**
- **short** (16bit), halber **int**-Bereich, 2er Komplement
- **int** (16bit)
- **long** (32 bit)
- **float** (32bit, 0-22bit für Mantisse)
- **double** (64bit, 0-51bit für Mantisse)
- **signed/unsigned**

Speicherklassen

- Lebensdauer: Zeitraum, in welchem die V. im Speicher verfügbar ist
- Gültigkeitsbereich: Orte, an welchen man mit dem Namen auf diese Variable zugreifen kann
- **auto**: Vereinbarung bei Blockeintritt, nur innerhalb eines Blocks ansprechbar, default-Wert
- **register**: wie **auto**, Compiler versucht Varaible im Register des Prozessors unterzubringen
- **static**: Lebensdauer über das gesamte Programm, sagt nix über Gültigkeitsbereich aus
- **extern**: Definition einer Variablen steht in anderem Quelltext

#define

textuelle Ersetzung, z.B. `#define MWST 0.15`

Regeln:

- **#define**-Konstanten werden groß geschrieben
- symbolische Konstanten immer am Programmanfang
- außer 0 und 1 nur symbolische Konstanten

casting

der Wert des Ausdrucks wird berechnet und anschließend in den Datentyp konvertiert

```
int i;
float j;
i= (int) (j*4.0);
```

Seiteneffekt

`b=((a=3)*(a*=3))` -> 27 oder 36, je nach Compiler

Seiteneffekte sind schlechter Programmierstil und verboten

Priorität (Präzedenz)	*Assoziativität*
() [->	l.n.r.
! ~ ++ -- + - * & (type) sizeof	r.n.l.
* / %	l.n.r.
+ -	l.n.r.
>> <<	l.n.r.
< <= > >=	l.n.r., 10
== !=	l.n.r., 9
&	l.n.r.
^	l.n.r.
|	l.n.r.
&&	l.n.r., 5
||	l.n.r., 4
?:	r.n.l.
= *= -= += /= %= &= ^=	r.n.l.
,	l.n.r.

Präzedenz: Ordnung der Operatoren nach Priorität
Assoziativität: Ordnung innerhalb einer Präzendezgruppe

printf

Ausgabe von Zeichen und Formatelementen

'%'-Zeichen: Start eines Formatelementes
optional: '-': linksbündige Ausrichtung
Zahl1: minimale Feldbreite
'.': Trennzeichen
'Zahl2: maximale Feldbreite, sonst Nachkommastellen
'd': dezimale Darstellung
'o': oktale Darstellung
'x': hexadezimale Darstellung
'u': unsigned
'c': einzelnes Zeichen
's': String
'f': float
'lf' double
'e': Expotentialdarstellung: [-]mmm.nnn E[+,]xx

Modifizierer: h für short, l für long, L für long double

"Hello, world"	
<%10s>	<Hello, world>
<%-10s>	<Hello, world>
<%20s>	<____8___Hello, world>
<%-20s>	<Hello, world____8___>
<%20.10s>	<____10____Hello, wor>
<%.10s>	<Hello, wor>

Kommentare

Kommentare zwischen '/*' und '*/' werden vom Compiler ignoriert und können überall stehen, wo Leerzeichen erlaubt sind.
in C++ sind auch '//' für einzeilige Kommentare zugelassen.
Warum? gute Lesbarkeit, Übersichtlichkeit, Wartung, Strukturierung, Programmdokumentation
Regeln: jedes Programm mit ausreichend Kommentar versehen

Blöcke

Ein Block ist eine Folge von Vereinbarungen und Anweisungen. Beginn und Ende werden mit '{' und '}' gekennzeichnet.

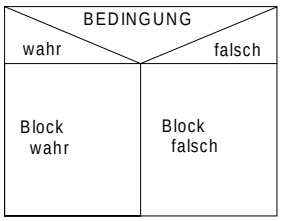
- können beliebig ineinander verschachtelt sein
- als Ganze werden als einzelne Anweisung betrachtet
- regeln Geltungsbereiche von Variablen

scanf

liest von der Standardeingabe(Tastatur)
vorher sollte fflush(stdin) den Puffer leeren
Controlstring: Leerzeichen werden ignoriert
'*' unterdrückt Zuweisung an Varaible (skip)
maximale Feldbreite kann angegeben werden
'&'-Zeichen nicht vergessen, Strings + 1 Zeichen, wg. '\0'

if-then-else

Berechnen der Bedingung, Ergebnis wahr oder falsch



- wahr: Ausführung 1. Block
- falsch: Ausführung else-Block
- Der **else**-Zweig wird jedoch nicht immer benötigt.
- Blöcke immer Klammern und mit Semikolon abschließen!

- Formeln mit int-Ergebnis können als Bedingung aufgefaßt werden.

```
int a = 4;
if(a)
  { ...; };
else
  { ...; };
```

- Bedinungen sollten nicht zu komplex sein.
- Klammern sollten nicht verwendet werden.
- Bedingungen sollten keine Seiteneffekte besitzen.
- Kommentare am Ende der Bedingung verwenden.

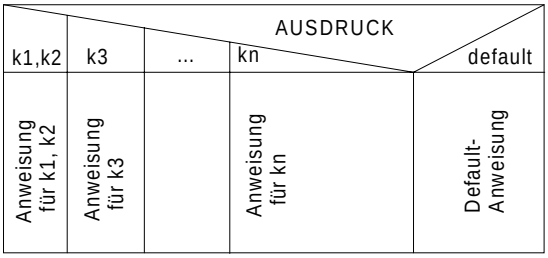
Bedingte Bewertung

```
ergebnis = e1 ? e2 : e3;
```

Es wird der Wert der Bedingung e1 ausgerechnet. Ist e1 wahr, dann wird e2 berechnet und dem Ausdruck zugewiesen, sonst e3.

- Verwendung nur dann, falls e1, e2, e3 einfach sind.
- Bei komplexen Ausdrücken Zwischenergebnisse verwenden.

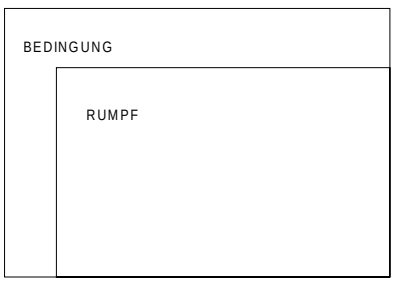
switch



- Erlaubt die Auswahl mehrerer Alternativen
- Wert des Ausdrucks muß **int** oder **char** sein.
- **break**-Anweisung beendet **switch**
- fehlt eine **break**-Anweisung, so wird mit dem nächsten Befehl weitergearbeitet.
- Ist das Ergebnis nicht in der Konstantenliste, so wird der **default**-Zweig ausgeführt.
- Bei Bereichen immer **if-then-else** verwenden, z.B.

```
if ((a>0) && (a<100))
```

while-Schleife



Berechnen der Bedingung. Im wahr-Fall folgt die Ausführung des Schleifenrumpfes, sonst wird die Schleife beendet.

- Es muß auf Terminierung geachtet werden.
- Schleifenkriterium kann schon zu Beginn nicht erfüllt sein -> 0..n mal ausführbar
- Rumpf aus mehreren Anweisungen klammern.
- sinnvoll, wenn die Anzahl der Iterationen nicht bekannt ist.

for-Schleife

(Abbildung bei **while**-Schleife)

- Die Anzahl der Iterationen muß bekannt sein.
- Initialisierung, Kriterium und Reinitialisierung sollten so einfach wie möglich sein.
- Kontrollvariablen können im Rumpf manipuliert werden.

do-while-Schleife

Der Rumpf wird mindestens einmal ausgeführt, danach Test der Bedingung und im wahr-Fall folgt die Wiederholung des Rumpfes.

break

beendet eine **switch**-Anweisung und jede andere Schleife

continue

springt an das Ende des Schleifenrumpfes

Felder

Verarbeitung einer Vielzahl an Datenelementen mit gemeinsamen Eigenschaften unter gleichem Namen (mit Indices).
Eine Feldvariable setzt sich aus einer Menge gleichartiger Teilvariablen zusammen, alle mit dem gleichen Datentyp, dem gleichen Variablennamen, aber mit unterschiedlichen Indices.

- Felder können von allen Datentypen gebildet werden.
- Anfangsindex ist immer 0
- höchster Index ist (KONSTANTE - 1)
- In C findet keine Überprüfung der Feldgrenzen statt
- Bei Felddefinitionen nur KONSTANTE verwenden.

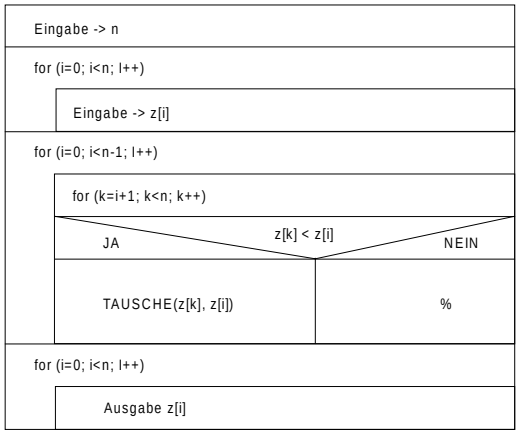
```
int students[40];
char profs[40][40][2];
```

Initialisieren von Feldvariablen

- Initialisierungsliste enthält in geschweiften Klammern durch Komma getrennte Werte (1. Wert -> Nulltes Feldelement, usw.)
- alle Feldelemente, die nicht initialisiert werden bekommen 0 zugewiesen.
- der letzte Index nimmt am schnellsten zu
- Anfangswertgruppen werden in geschweifte Klammern gesetzt
- Ist die Feldgröße nicht angegeben, so wird sie automatisch vom Compiler aufgrund der Initialisierungsliste ermittelt.

```
static int zahlen[5] = {1,2,3};
static char farbe[] = {'R','G','B'};
static int werte[2][2] = {1,2};
static int werte1[4][4] = {{1,2,3}, {4,5,6}};
```

Sortieren von Zahlen



Funktionen

+ keine Codeduplikation bei gleichen Anforderungen
+ Wiederverwendbarkeit
+ schnelle und sichere Realisierung

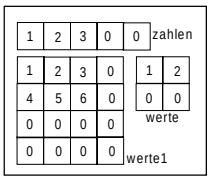
- Funktionen werden nur auf äußerstem Level parallel zu **main()** realisiert
- Die Reihenfolge der Definitionen ist wichtig
- Eine Funktion besteht aus Ergebnistyp (Datentyp, welchen die Funktion zurückliefert, per default auf **int**), Funktionsname, formalen Parametern (variable Daten, welche beim Aufruf mit aktuellen Werten versorgt werden) und dem Funktionsrumpf.

globale Variable
Definition von Variablen parallel zu Funktionen.

- Lebensdauer: gesamtes Programm
- innerhalb einer Funktion kann nach Regeln der Gültigkeitsbereiche auf solche Variablen zugegriffen werden.
- schlechter Programmierstil (Seiteneffekte möglich)

lokale Variable
Funktionsrumpf ist ein Block, d.h. es können Variablen angelegt werden.

- Sie werden beim Aufruf angelegt und beim Verlassen der Funktion vernichtet.
- Variablen mit gleichen Namen können in verschiedenen Funktionen definiert werden, belegen aber unterschiedliche Speicherstellen.



return

- Jede Funktion mit Ergebnis muß mindestens eine **return**-Anweisung enthalten.
- Bei Ausführung von **return** wird der Ausdruck berechnet.
- Die Funktion wird beendet und die Kontrolle wieder an das aufrufende Programm übergeben und der Wert des Ausdrucks zurückgeliefert.
- Fehlt ein **return**-statement, so wird beim Erreichen des Rumpfendes ein implizites **return** ausgelöst.
- guter Programmierstil: jede Funktion besitzt genau eine **return**-Anweisung am Ende

formale Parameter
Bei der Funktionsdefinition kann eine Liste von Variablen angegeben werden, die Platzhalter für die späteren, tatsächlichen Aufrufwerte sind.
- Lebensdauer: Dauer des Aufrufs
- Jeder formale Parameter hat einen eigenen Datentyp.

aktuelle Parameter
sind die tatsächlichen Werte, mit denen die Funktion initialisiert wird
- werden beim Aufruf an die formalen Parameter zugewiesen

Call-by-Value
die Funktion arbeitet immer mit Kopien der aktuellen Parameter
- Wirkung: eine Manipulation an den formalen Parametern hat keine Auswirkungen auf die aktuellen Parameter.

Call-by-Reference
für formale Parameter werden keine Speicherstellen eingerichtet, sondern nur ein anderer Name für den aktuellen Parameter
- &-Zeichen vor dem Parameternamen
- Wirkung: eine Änderung der formalen Parameter bewirkt eine Änderung der aktuellen Parameter (Realisation über Zeiger)

Referenztypen
Bei der Variablendeklaration soll kein eigenes Objekt angelegt werden, sondern lediglich ein anderer Name für ein bestehendes Objekt vergeben werden
- Eine Referenz kann nicht mehr verändert werden.

```
int i = 1;          i = 1
int &ir = i;        i, ir = 1
ir = 3;             i, ir = 3
i++;                i, ir = 4
```

Rekursion
Funktionen können in ihrem Rumpf weitere Funktionen aufrufen. Und eine Funktion kann einen Aufruf von sich selbst beinhalten.
- Bedingungen: rekursive Definition des Algorithmus
  Abbruchbedingung

Fibonacci-Zahlen
```
int fib (int n)
  { switch(n)
    { case 0: return 0;
      case 1: return 1;
      default:
        return fib(n-1) + fib(n-2);
    }
  }
```

Fakultät
```
int fak(int n)
  { if (n==0)
      return 1;
    else
      return
      n*fak(n-1);
  }
```

#define
```
#define NAME Ersetzungstext
```
Alle Vorkommen von NAME werden vor der Compilierung textuell durch den Ersetzungstext ersetzt. Makros sind keine Funktionen.
```
#define MAX(A,B)   ((A)>(B) ? (A) : (B)
x = MAX(i++, j++);  wird zu
x = ((i++) > (j++) ? (i++) : (j++));
```

Prototyping
Bereitstellung zusätzlicher Infos für den Compiler über
- Existenz der Funktion
- Aufruf
- Signatur

```
int translate (int, char, float);
```

Default-Argumente (C++)
In C++ können Funktionen mit default-Aurgumenten vereinbart werden. Fehlt ein Argument im Funktionsaufruf, so wird automatisch das default-Argument verwendet.
```
void f2(int a, int b, int c=3, int d=4) ....;
```
- Default-Argumente müssen immer lückenlos am Ende der Parameterliste stehen.

Overloading (C++)
Aufruf von Funktionen, aber mit unterschiedlichen Parametern. Diese müssen sich in der Argumentliste oder des Rückgabewertes unterscheiden. Der Compiler erkennt beim Aufruf an den Datentypen der Argumente, welche Funktion aufgerufen werden muß.

Zeiger

Der Inhalt einer Zeigervariablen ist eine Adresse auf ein Objekt vom angegebenen Datentyp.
- sind immer an einen Datentyp gebunden
- können von allen Datentypen gebildet werden.

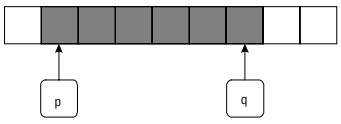
```
int *pv; //Zeiger auf int
```

Adressoperator &
Der Adressoperator darf nur auf Variable angewendet werden. Eine Anwendung auf Konstante, Ausdrücke und Registervariablen ist verboten, wie z.B. &15, &(a+b).

dereferenzieren der Zeigervariablen
- *pv: interpretiere den Inhalt von pv als Adresse und ermittle den tatsächlichen Wert an der Speicherstelle, die auf pv zeigt.
- Einer Zeigervariablen darf keine Konstante außer der 0 (NULL) zugewiesen werden. Die 0 bedeutet, daß die Zeigervariable zur Zeit auf kein gültiges Speicherobjekt verweist.

Verwendung von Zeigern als formale Parameter
```
void swap(int *a, int *b)
  { int temp;
    temp = *a; *a = *b; *b = temp;
  }
swap(&i, &j);
```

Zeiger und Felder
Der Name eines Feldes ist als Zeiger realisiert, welcher konstant ist, d.h. Inhalt bzw. Adresse kann nicht geändert werden.
```
int x[10]; int *pa;
pa = x; oder pa = &x[0];
```
- Wenn pa auf ein Feld zeigt, dann ist pa[0]=44 möglich.
- Der Feldname x zeigt auf das erste Element des Feldes x[0], es gilt x = &x[0].
- Adressrechnung möglich: x+1, pa+i, d.h. der Compiler addiert ja nach Datentyp die richtige Anzahl Bytes dazu.
*(pa+i) = 88; *(x+i) = 88; x[i] = 88; pa++;
- Die Operationen sind nur dann gültig, wenn der Zeiger auf ein gültiges Objekt verweist.



- q-p = 6 (Pointer-Pointer = Integer)
- q-p+1 entspricht der Anzahl der Felder, die die Zeiger voneinander trennen.
- p < q: p zeigt auf ein früheres Feldelement als q

Zeigerarithmetik ist datentypkonsistent

Zusammenfassung
```
pv=&v; //Adreßzuweisung an Zeigervariable
pv=px; //Wertzuweisung von Zeiger an Zeiger, Typ muß
       identisch sein
pv+=4; //ganzzahlige Größe zum Zeiger addieren
pv==pq;//Vergleich zweier Zeigervariablen
```

Zeiger und mehrdimensionale Felder
Ein zweidimensionales Feld ist eine Gruppierung von eindimensionalen Feldern, die nebeneinander liegen. Ein zweidimensionales Feld kann als Zeiger auf eine Gruppe nebeneinanderliegender eindimensionaler Felder definiert werden.

```
int x[10][10];
int (*px)[10];
px = &x[0][0]; //Klammern sind wichtig
```
Zugriff auf einzelnes Element x[2][5] über Zeiger *(*(px+2)+5);

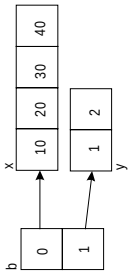
Character-Zeiger
```
char amessage[] = "Hurra";
char *pmessage = "Hurra";
```
- amessage ist ein Feld der richtigen Größe, amessage kann nicht verändert werden, die Feldelemente schon
- pmessage ist ein freier **char**-Zeiger, sowohl die Feldelemente, als auch pmessage können verändert werden.

Zeiger auf Zeiger
```
i = 10;
*zi = 10;
**zzi = 10; // =*(*zzi) = 10
```

typedef-Anweisung
Einführung von Abkürzungen für Datentypen mit komplexer Schreibweise oder Einführung von neuen benutzerdefinierten Datentypen. (der Compiler überprüft im Vergleich zum #define)
Beginnt eine Deklaration mit typedef, so wird der darin enthaltene Bezeichner nicht als Variablenname eingeführt, sondern als ein neuer Name für den gerade spezifizierten Datentyp.
```
typedef long* Plong;      Plong p; // long *p
typedef int(*zfi)();      zfi z1;  // int(*z1)();
```

Zeiger auf Funktionen
Ergebnistyp (*Variablenname)(Parameter);
```
extern int p1(); extern int p2();
int i;
int (*comp)();  //Zeiger auf Funktionen
comp = p1(); i = (*comp)();
comp = p2(); i = (*comp)();
```

Zeiger-Felder
sind speichereffizient, da Felder unterschiedlich lang sein können
```
int x[] = {10, 20, 30, 40};
int y[] = {1, 2};
int *b[2];
b[0] = &x[0];
b[1] = &y[0];
// *(b[0]+2) = 30 im Feld x
```

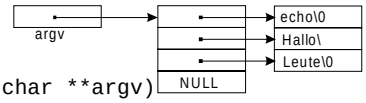


Argumentübergabe in der Kommandozeile

```
main (int argc, char **argv)
```

argc: Anzahl der übergebenen Elemente
argv: Zeiger auf Feld dessen Elemente Zeiger auf **char** sind
argv[0] ist immer der Programmname

```
$echo Hallo Leute
```




```
#include <stdio.h>
void main(int argc, char **argv)
{ while(--argc>0)
   { printf(%s\n", *++argv);  }
}
```

Spezialitäten

```
int* (*p)(int(*a)[])
```

- p ist ein Zeiger auf eine Funktion, die einen **int*** zurückliefert, Parameter ist ein Zeiger auf ein Feld, dessen Elemente ganze Zahlen sind

```
int (*p (char *a))[10]
```

Parameter ist ein Zeiger auf **char**, Ergebnis ist ein Zeiger auf ein ganzzahliges Feld mit 10 Elementen

Strukturen

```
struct Name { Def1; Def2; ... };
```

- Bei der Deklaration eines Strukturtyps werden noch keine Speicherobjekte angelegt. Es wird nur beschrieben, wie ein solches Datenobjekt aussieht.
- Gleiche Gültigkeitsbereiche wie für Variable

Definition

```
Name des Structs Var1, Var2;
```

- Der Zugriff auf structs erfolgt über Punktnotation.
- Sowohl die ganze **struct**, als auch einzelne Elementvariablen können angesprochen werden.

```
P1 = P2;   //bitweises kopieren von P2 nach P1

struct address {  char lastname[40],
                  char firstname [40],
                  char street [30],
                  long zip;} adr[10];

struct rectangle point {P1, P2}
                    //geschachtelte Struktur
rectangle screen;
screen.point.x=44;  //Zugriff auf ersten Punkt
```

Zeiger auf Strukturen

Zeiger auf Strukturen werden wie bei den Basistypen gebildet.

```
(*z_adr).mnr = 622037;
                    //Zugriff über Punktnotation
z_adr->mnr = 622037 //Zugriff über Pfeilnotation
```

- Die Pfeilnotation wird nur beim Zugriff auf Strukturvariablen über Zeiger verwendet, ansonsten gilt die Punktnotation.
- Der Programmierer muß darauf achten, daß der Zeiger auf ein gültiges Speicherobjekt verweist.

```
point *z_p;
z_p->x = 40;        //schreibt ins Nirvana
```

- sollen große Strukturen bearbeitet werden, so ist es effizienter, nur eine Zeigervariable auf diese Struktur zu übergeben

union

Eine **union** faßt wie eine Struktur verschiedene Variable zu einer Einheit zusammen, nur daß hier alle Elementvariable ab der gleichen Speicheradresse beginnen.

- Alle Elemente belegen zusammen nur soviel Speicher, wie die größte Elementvariable benötigt.
- Alle Operationen/Deklarationen auf structs können auch auf unions angewandt werden.

Dynamische Datenstrukturen

Beschaffung von Speicherplatz zur Laufzeit

```
char *malloc(int size)
new
```

- **malloc** beschafft einen zusammenhängenden Speicherbereich von size-Bytes auf dem Heap
- Die size-Angabe ist nie absolut, sondern mit **sizeof**(int) anzugeben
- Das Ergebnis ist unter Umständen zu casten.
- Bei Fehler wird NULL zurückgeliefert.

```
address *z_adr;
z_adr = (address*) malloc(sizeof(address));
z_adr = new address;
```

- bei **new** is keine Angabe der Größe notwendig
- bei **new** wird ein korrekter Zeiger zurückgeliefert

Speicherfreigabe

Der Speicherbereich auf dem Heap wird wieder freigegeben. Die Zeigervariable selbst wird nicht verändert!!!!!!

```
free(z_adr);
delete
```

- **delete** hat Vorteile bei OOP
- jeder dynamisch beschaffte Speicherplatz muß explizit freigegeben werden
- der Programmierer selbst ist für die korrekte Verwendung von **malloc/free** zuständig
- Stackvariablen dürfen nie freigegeben werden!!!!!

```
address a;
z_adr = &a;
free(z_adr);      //falsch
z_Adr = new address;
delete z_adr;
```

Einfach verkettete Listen

```
struct Eliste   { //Nutzdaten
                  Eliste *succ;};

Eliste* gen_elem()
{ Eliste* help;
  help = new Eliste*;
  if (help!=NULL) help->succ=NULL;
  return help;
}
int insert_elem(Eliste** k, Eliste *hilf, Eliste
*z_hl)
{ int fehler = 0;
  if(*k = NULL) //Liste noch leerr
    { *k = z_h1; z_h1->succ = NULL; }
  else
    { if(hilf!=NULL)
        { z_h1->succ = hilf->succ;
          hilf_succ = z_h1; }
      else
        fehler = 1; }
}
int delete_elem(Eliste ** k, Eliste *hilf)
{ Eliste* vorgaenger;
  int gefunden, fehler = 1;
  if(*k == hilf)//erstes Element ausketten
    { *k = hilf->succ; fehler = 0;}
  else //Vorgänger ermitteln
    { vorgaenger = *k; gefunden = 0;
      while (vorgaenger != NULL)
      { if(vorgaenger->succ == hilf)
          { gefunden = 1; break; }}
        else
          vorgaenger = vorgaenger->succ;}
    if (gefunden != 0)
      { vorgaenger->succ = hilf->succ; fehler = 0;}
    }
    if(fehler == 0) free(hilf);
    return fehler;
}
```

Bei doppelt verketteten Listen muß auf erstes Listenelement, Mitte und Ende überprüft werden.

String- und Dateifunktionen

```
strcpy( ziel, quelle);
char ziel[8]; strcpy(ziel,"Hurra");
```

- Vorsicht: Stringendzeichen '\' wird automatisch angehängt!!!

```
i = strln(ziel);
```

- **strlen** zählt die Anzhal der Zeichen bis zum Stringterminator

```
strcat(c1, c2);
```

- c2 wird durch strcat an c1 angehängt

```
int strcmp(s1, s2);
```

- lexikografischer Vergleich zweier Zeichenketten, >0 (s1 ist lexikografisch größer als s2)

```
strcmpi(s1,  s2);  //nicht case-sensitiv
```

Dateiverarbeitung

```
#include <stdio.h>
```

- sequentieller Zugriff: um an den Datensatz n zu gelangen, müssen alle vorherigen n-1 Elemente gelesen werden.
- random access: direkter Zugriff auf gewünschte Position

```
FILE *fp;
FILE *fopen(char *filename, char *mode)
mode: "r" nur lesen, "w" nur schreiben (Datei wird überschrieben, falls sie beschrieben ist), "a" anhängen, "r+" nur lesen (Datei muß existieren)
char* fgets(char *s, int n FILE *fp);
fgets liest entweder bis zum Zeilenende('\n') oder maximal n-1 Zeichen
'\0' wird auf jeden Fall an das Ende von *s gespeichert, im ersten Fall zuvor noch ein '\n'
int fputs(char *s, FILE fp*); //kein '\n' und kein '\0'
int fgetc(FILE *fp); int putc( char c, FILE *fp);
liest das nächste Zeichen aus der Datei und castet nach int
am Dateiende oder im Fehlerfall wird EOF zurückgeliefert
int fscanf( ... ); int fprintf( ... );
wie scanf, RETURN-Wert: Anzahl der fehlerfrei gelesenen, konvertierten und gespeicherten Felder; 0 falls keine Zuweisungen erfolgt sind; sonst EOF
int feof (FILE *fp);
0 Ende erreicht, sonst 1
long ftell(FILE *fp);
aktuelle Dateiposition in Bytes, sonst negativer Returnwert im Fehlerfall
```