C

einfach und verständlich für den Anfänger

geschrieben von

Georg zur Horst-Meyer

basierend auf den Vorlesungen, Übungen und dem Skript von

Prof. Dr. Uwe Hartmann

zur Programmierungstechnik I
an der Fachhochschule Stralsund

V 0.9

Vorwort

Dieses Skript soll es den Studenten, die noch keine Programmiererfahrung haben helfen, den Einstieg in die Programmiersprache C zu finden.

Dieses Skript ist NICHT dafür bestimmt, die Übungen oder gar die Vorlesung, geschweige denn das Skript von Prof. Hartmann zu ersetzen.

Es ist dringend empfohlen, die Vorlesungen und die Übungen zu besuchen und die Aufgaben für die Übungen selber zu lösen und zu verstehen!

Auswendig lernen hilft bei Programmierungstechnik nur bedingt!
Nur wer verstanden hat, wie ein Computer Befehle abarbeitet, was wahr und was falsch ist und wie Algorithmen programmiert werden, der hat meiner Meinung nach eine reelle Chance, die Prüfung am Ende des Semesters zu bestehen.

In diesem Sinne,

Viel Erfolg und Spaß beim Lernen der Programmiersprache C und bei der abschließenden Prüfung!

INHALT:

Zum Programmieren allgemein, was beachtet werden sollte!!

Wenn ein Programm geschrieben werden soll, oder man selber eines schreiben möchte und noch kein Vollprofi ist, sollten die folgenden Ratschläge befolgt werden, andernfalls werden die Fehler quasi eingeladen.

1. Wenn ich ein Programm schreiben möchte, sollte ich mir zuerst darüber im Klaren sein, was das Programm können und machen soll.
   Hierfür eignet sich ein Ablaufdiagramm, aus dem unmittelbar erkennbar ist, was das Programm später einmal machen soll, hierbei ist es ratsam, noch keinen Programmiercode zu verwenden.

2. Wenn Punkt 1. berücksichtigt wurde, kann nun das Werk angegangen werden.
   Dabei gelten folgende Faustregeln:

   - Absätze machen, Platz lassen!
     Wenn Ihr einen Programmteil (z.B. Einlesen von Parametern in Variablen oder Felder) fertig programmiert habt, fügt ein paar Absätze ein, damit ihr später leicht erkennen könnt, wo die Abfrage endet.

   - Mehr einzelne Zeilen, schnellere Übersicht!
     Es gibt viele Möglichkeiten, viele Zeilen Programmcode in eine einzige Zeile zu packen. Dem geübten Programmierer erspart das nervige Tipparbeit und er sieht auch sofort alle Abhängigkeiten. Ein Anfänger, der gleich alles in eine Zeile packt, wird sehr schnell an seine Grenzen stoßen, denn er kennt nicht alle Abhängigkeiten und wenn er dann später einen Fehler sucht, hat er es mehr als doppelt so schwer den Fehler zu finden, geschweige denn seinen eigenen Programmcode zu verstehen.
     Deswegen, auch wenn alles in einer Zeile geschrieben werden kann, lohnt es sich, lieber ein paar Zeilen mehr zu schreiben, als eine zu wenig, das spart letztendlich viiiiiiiel Zeit und der Code ist einfacher zu überblicken.

3. Platzhalter verwenden, Text einrücken -> weniger Schleifenfehler!!
   Wenn Schleifen geschrieben werden, rücken viele den Text zwischen den {}-Klammern nicht ein, dies ist schlecht. Warum? Weil hinterher keiner mehr durchblickt, welche Klammer und welche Zeile zu welcher Schleife gehören, auch hier wird sehr schnell der Überblick verloren und es entstehen Endlosschleifen.
   Deswegen, rückt den Text in den {} Klammern immer um einen Tabulator ein (das ist die Taste links vom Q)!
   Wenn ihr eine `for`-Schleife mit nur einer Anweisung in der Schleife habt und die {}-Klammern weg lasst, dann rückt diese eine Anweisungszeile ebenfalls ein!!!

4. Kommentare
   Wenn ihr ein Programm schreibt, ist es Sinnvoll, Kommentare zu verwenden.
   Kommentare werden vom Compiler als Kommentare erkannt, wenn sie hinter // in einer Zeile Stehen.
   Warum Kommentare?
   Ich verstehe doch wohl meinen eigenen Text?
   Das ist richtig, ein anderer aber u.U. nicht, deswegen kommentiert Absätze mit z.B. "`// einlesen der Variablen "` oder Zeilen mit z.B. "`// Aufrufen der Unterfunktion`".
   Ferner fördert es die Übersicht und gibt euch zu einem späteren Zeitpunkt einen besseren Einstieg in das Programm.

Dies ist daher wichtig, da z.T. Programme geschrieben werden und später nach z.B. 50 Tagen noch einmal etwas geändert werden oder ein anderer das Programm weiter entwickeln soll.
Man selber weiß schon nach 2 Wochen nicht mehr im Detail, wie man das eine oder andere Problem gelöst hat und dann muss man Zeile für Zeile das Programm durchgehen um zu verstehen, was wo gemacht wird, ihr werdet es selber merken, wenn ihr euch am Ende des Semesters Programme anschaut, die ihr zu Beginn des Semesters geschrieben habt.
Deswegen, setzt Kommentare bei:

- Zeilen, die schwierig zu verstehen sind
- Als Überschriften für neue Absätze
- zum Erklären schwieriger Zeilen für dritte
- oder als Notizen

5. Gebt euren Variablen Namen, die aussagekräftig sind!
   Wenn ihr euren Variablen immer nur Namen wie z.B. a, z,i,k,q etc. gebt, verliert ihr schnell den Überblick, ferner besteht die Gefahr, dass wenn der Quelltext länger wird, ihr Variablen doppelt verwendet und das Programm dann abstürzt, bzw. ihr Tagelang nach dem Fehler sucht.
   Kurze Variablennamen sind immer dann hilfreich, wenn:

   - das Programm sehr klein ist
   - oder eine Variable nur in einem kleinen Zeitfenster benötigt wird, wie. z.B. eine Zählvariable in einer Schleife, die beim Eintritt in die Schleife Initialisiert wird.

   In allen anderen Fällen ist es ratsam, Variablen, Felder, Strukturen, Zeiger etc. entsprechend ihrer Funktion zu benennen, z.B. `Namensfeld` oder `Indexvariable` etc.
   Wenn dies berücksichtigt wird, ist schon aus dem Namen des Zeigers etc. zu erkennen, was er speichern und wofür er es speichern soll, damit werden die Programme übersichtlicher und leichter zu verstehen.
   Die Gefahr einer doppelten Benutzung einer Variablen etc. sinkt dadurch erheblich.

All diese Sachen werden später beim kompilieren vom Compiler rausgenommen.
Es macht dem PC also nichts aus, wenn Ihr Platz lasst oder Kommentare schreibt. Für den PC steht nachher so oder so Buchstabe neben Buchstabe, egal ob ihr Platz lasst oder nicht.
Das Ganze dient nur, um euch und anderen einen besseren Überblick zu verschaffen und Fehler zu vermeiden.

Die Maschine kann den Code, auch wenn er noch so unübersichtlich ist lesen, aber ein Mensch, der nicht wie eine Maschine arbeitet / arbeiten kann, braucht die Übersicht.

auto

```
auto int i = 0
```

Bedeutet, Autoinitialisierung beim Durchlauf des Programmblockes, ist gleich bedeutend mit

```
int i = 0
```

static

```
static int i = 0
```

Bedeutet, Initialisierung von i nur beim ersten Durchlauf des Programmblockes, bedeutet, dass beim zweiten Durchlauf diese Zeile übersprungen wird

```
for (int z=0, z<=50, z++)
{
     static int i = 0;
     i++;
}
```

register

```
register int f = 0
```

Die Variable behält ihren Wert nur vorübergehend bis zum Ende ihres Gültigkeitsbereiches bei (z.B. bis zum Ende einer Schleife etc.).

`register` ist vom Typ `auto`.

extern

```
int f(int k)
{
     extern int k;
     k++;
}
int k = 23;
void main()
{
     f(k);
}
```

`extern` wird benötigt, um dem Compiler zu sagen, dass die Variable `k` vorhanden ist ABER ERST AN SPÄTERER Stelle deklariert wird!
Dies wird benötigt, wenn Ihr später Funktionen schreibt, in denen Variablen verwendet werden, aber erst im weiteren Programmcode deklariert werden, dazu aber an späterer Stelle mehr.

Wertzuweisungen:

```
i=f=t=k=5;
```

Es wird **IMMER** der Wert von rechts nach links zugewiesen. Bedeutet, erst bekommt k den Wert 5, dann t den Wert von k usw. zugewiesen.

const

```
const int k=3;
```

k wird mit 3 initialisiert und da const davor steht, wird k eine Konstante mit dem Wert 3, welche während des gesamten Gültigkeitsbereiches nicht mehr veränderbar ist.

printf-Funktion

("Text",Argumente) siehe Skript s 26

scanf-Funktion

("Text", Argumente) siehe Skript s 27

getchar ()

```
char c;
c = getchar ();
```

Die Variable c bekommt ein Zeichen aus dem Eingabestrom zugewiesen.
Der Eingabestrom kann von der Tastatur kommen, oder aber beim Aufruf des Programmes in der Konsole dem Programm übergeben werden (Bei Windows auf Start, Ausführen und dann cmd eintippen) (Die Befehle sind Dos ähnlich, einfach mal nach Dos bzw. Windows Konsolen Befehle im Netz suchen!).

putchar ()

```
char c = 'A';
```

Definiert die Variable c als Variable die als Wert einen Buchstaben beinhalten kann und weißt ihr den Buchstaben A als Wert zu

```
putchar (c);
```

Gibt den (ersten) Charakter der Variable c aus -> A

flushall()

`flushall()` wird manchmal benötigt, wenn z.B. ein Zeichen aus dem Eingabestrom gelesen werden soll und schon vorher einmal im Programm ein Zeichen eingelesen wurde.
In den meisten Fällen kommt es dann vor, dass beim zweiten Einlesen keine weitere Abfrage eines Zeichens erfolgt und der Compiler die Abfrage quasi ignoriert und mit dem letzten Zeichen aus der letzten Eingabe weiter arbeitet.

```
char c,b;

c=getchar ();
printf("%c",c);

b=getchar();
printf("%c",b);
```

Hier wird das Zeichen, welches bei der ersten Eingabe eingegeben worden ist, zwei mal ausgegeben.
Wenn nun vor dem zweiten Einlesen `flushall()` verwendet wird, so können wirklich zwei Zeichen eingegeben werden.

```
char c,b;

c=getchar ();
printf("%c",c);

fushall();

b=getchar();
printf("%c",b);
```

system()

Mit system() kann auf die Konsole des Betriebssystems zugegriffen werden.
Damit können Betriebssystem abhängige Kommandos erteilt werden und können einem damit einiges an Arbeit ersparen.
Wenn z.B. das Datum des PC's geändert und hierfür eine Abfrage erfolgen soll, so kann dies wie folgt gemacht werden:

```
system("date");
```

Wenn aber z.B. das ganze durch einen Zeichenstring aus einem Feld erfolgen soll, so kann es wie folgt gelöst werden:

```
char s[]="date 12-04-2009";
system(s);
```

Wenn ihr den Code des letzten Beispiels ausprobiert und ihr Windows verwendet, dann schaut auf die Uhr in der Taskleiste!
Wichtig bei dem System Aufruf ist, dass immer nur ein Argument eingegeben werden kann, sprich, wenn eine Eingabe wie z.B. beim Datum erforderlich ist und dies durch das Programm gemacht werden soll, so sollte für diesen Befehl ein Zeichenstring „gebastelt" werden.
Andernfalls kann / muss der System-Aufruf auch mehrmals ausgeführt werden.
Ein weiteres Beispiel zeigt, wie ihr den Bildschirm löschen könnt:

```
printf("Text1");
system("cls");
printf("Text2")
```

„`Text1`" wird durch `cls` (ClearScreen) gelöscht. Danach wird `Text2` auf den Bildschirm ausgegeben.

Der System Aufruf ist nicht Gegenstand der Vorlesung von Prof. Hartmann, da er aber einem viele Möglichkeiten gibt und sich auch kleine „Viren" und viele andere Späße und auch sinnvolle Programme damit programmieren lassen, möchte ich ihn hier erwähnt haben.

Wer mehr über die Möglichkeiten des System Aufrufes erfahren möchte, sollte sich mit der Konsole und den dazu gehörigen Befehlen seines Betriebssystems auseinander setzen, es lassen sich so wunderbar sortier-Programme für Bilder etc. und vieles mehr schreiben!
Achtung, wie das Beispiel mit der Uhr gezeigt hat, operiert ihr nun direkt am Betriebssystem, bzw. an euren Daten, also Vorsicht! Am Besten ihr macht ein Backup des Systems oder arbeitet nur mit Kopien von Verzeichnissen um Datenverlust zu vermeiden!

Operatoren:

1. **Uniäre Operatoren**

   +
   -

   Sind Vorzeichen, die eine Zahl positiv oder negativ machen.

2. **Arithmetische Operatoren**

+	Addieren von Werten
-	Subtraktion
*	Multiplikation
/	Division
%	Teilt eine Zahl durch eine andere und liefert den Restwert zurück

   `z=(8 % 2)` -> gibt `0` aus (weil 8/2 = 4 ist Rest 0)
   `z=(7 % 5)` -> gibt `2` aus (weil 7/5 = 1 ist Rest 2)

3. **Vergleichsoperatoren** (liefern nur den logischen Wert 1 oder 0 zurück!)

   ```
   if (5==1)
   ```

   Da `5` ungleich `1` ist, liefert der `==` Operator eine `0` zurück und die If-Anweisung wird nicht ausgeführt.

<	kleiner als
<=	kleiner gleich
>	größer als
>=	größer gleich
==	gleich
!=	ungleich

4. **Logische Operatoren** (Ausführung von links nach rechts)

!	nicht-Operator
||	Oder Operator
&&	Und Operator

   ```
   if !(5==1)
   ```

   Da `5==1` den logischen Wert `0` liefert und mit dem `!` Operator negiert wird, wird die Schleife ausgeführt, da `0` negiert `1` ergibt.

   ```
   if ( !(5==1) && (3==3) )
   ```

   Beide Argumente sind erfüllt (betrachtet das Beispiel genau!), daher wird die if-Anweisung ausgeführt (es wird erst der `!(5==1)` Ausdruck geprüft (logisch 1) und dann der `(3==3)` Ausdruck (ebenfalls logisch 1)).

5. **Bitoperatoren** (Achtung, Bitoperatoren arbeiten im binären Zahlensystem!)

```
a= 1011= 11 ; b= 1010 =10
```

~	Komplement	~a = 0100	eine 1 wird in eine 0 und eine 0 in eine 1 gewandelt
&	Und	a&b = 1010	alle 1 werden addiert (nur da wo beide Zahlen eine 1 haben, kommt eine 1 bei raus) (man muss sich zwei Schalter in Reihe geschaltet vorstellen, nur wenn beide geschlossen sind fließt Strom, also eine 1)
|	oder	a|b = 1011	jede Stelle an der eine der beiden Zahlen eine 1 hat, wird eine 1 geschrieben (z.B. zwei Lichtschalter, jeder kann das Licht einschalten (1)) (1|0 -> 1 ; 0|0 -> 0; 1|1 ->1)
^	exklusiv oder	a^b = 0001	nur wenn die beiden Zahlen an der Stelle verschieden sind, wird eine 1 gesetzt (1^0 -> 1 ; 0^0 -> 0 ; 1^1 -> 0)
>>n	Rechtsver-schiebung um n Bit	15>>3 =0001	Bei Rechtsverschiebung wird um n mal durch 2 geteilt
<<n	Linksver-schiebung um n Bit	15<<3 = 1111000	Bei Linksverschiebung wird um n mal mit 2 multipliziert

6. Zuweisungsoperatoren

```
a= 1011= 11 ; b= 1010 =10
```

a = b	a=b	a bekommt den Wert von b =10	
a+= b	a=a+b	a bekommt den Wert von a+b =21	
a-= b	a=a-b	a bekommt den Wert von a-b =1	
a*= b	a=a*b	a bekommt den Wert von a*b =110	
a/= b	a=a/b	a bekommt den Wert von a/b =1	der Rest von 11 /10 wird weggelassen
a%= b	a=a%b	a bekommt den Wert von a%b =10	es wird nur der teilbare Teil der Division von 11/10 ausgegeben ->10
a&= b	a=a&b	a bekommt den Wert von a&b =10	es wird auf binärer ebene gerechnet!
a|= b	a=a|b	a bekommt den Wert von a|b =11	
a^= b	a=a^b	a bekommt den Wert von a^b =1	
a>>=b	a=a>>b	a bekommt den Wert von a>>b =0	
a<<=b	a=a<<b	a bekommt den Wert von a<<b =11264	

7. Inkrement- / Dekrementoperator

`++a`	`a` wird um 1 erhöht `=12`	dies ist Präfix-Operator, das bedeutet, dass wenn `a=10` und `printf("%d",++a)` dann wird a erst um eins erhöht und dann ausgegeben, in diesem Fall also die `11`
`--a`	`a` wird um 1 erniedrigt `=10`	
`a++`	`a` wird um 1 erhöht `=12`	Dies ist ein Postfix-Operator, dass bedeutet bei `a=10` und `printf("%d",a++)`, dass erst a ausgegeben wird (also die `10`) und dann a um eins erhöht wird, a danach den Wert `11` enthält -> Merksatz Postfix, Die Post kommt zu spät!
`a--`	`a` wird um 1 erniedrigt `=10`	

Achtung!
`a++` ist zwar vergleichbar mit `a+1`, doch wenn ihr `a` um `1` erhöhen wollt, dann geht nur `a++`, denn `++` ist ein Inkrement**operator**, `a+1` kann `a` aber nur um `1` erhöhen, wenn es sich um eine Wertzuweisung, also `a=a+1` handelt!!! Deswegen ist `a+1` falsch!
Dies ist bei Anfängern eine geliebte, wenn auch kleine Fehlerquelle.

8. sizeof Operator

```
x=sizeof (int b)
```

x bekommt den Wert des erforderlichen Speicherplatzes von `int b` in Byte zugewiesen
daher `x=4` (weil maximaler `int` wert `2147483647` = 4 Byte Speicherplatz benötigen)
(Testprogramm dafür: `int a=1; while (a>0) a++; printf("%d",--a);`)

9. Typecast Operator

```
int a = 3, b=5;
float c;
c = (float) a/ (float) b
```

`c` ist vom Typ `float`, daher müssen `a` und `b` in den Typ `float` umgewandelt / getypecastet werden (das Umwandeln von Variablen in einen anderen Variablentyp nennt man Typecasting).

Hinweis: Wenn ihr später ein Programm schreibt und es ein Problem mit dem Inhalt einer Variable bei einer Wertzuweisung gibt und der Compiler meckert, dann ist es häufig eine fehlerhafte Zuweisung von Variablenwerten, wo getypecastet werden muss!

10. Adressoperatoren

`&`	gibt die Adresse eines Elementes im Arbeits-Speicher (RAM) zurück
`*`	gibt den Inhalt eines Elementes aus dem Arbeits-Speicher zurück
`->`	Der Pfeiloperator wird verwendet um auf Elemente innerhalb einer Struktur zuzugreifen, wenn auf die Struktur durch einen Zeiger adressiert wird (siehe Zeiger & Strukturen) (Später wird darauf näher eingegangen, der Vollständigkeit wurden die Adressoperatoren hier genannt)

Steueranweisungen

Wann wähle ich welche Steueranweisung? (`if/ switch/while`)

Wenn ich einen Fall habe, wie wenn z.B. meine Variable `0` ist und ich diesen Fall ausschließen möchte, dann verwende ich eine `if`-Anweisung z.B. `if (Variable ==0) tue irgendwas;`

Wenn sich aus einer Texteingabe mehrere Möglichkeiten ergeben (z.B. ein Menü), weiter zu verfahren, dann wähle ich eine `switch`-Anweisung.

Wenn ich nun einen Algorithmus habe, bei dem ich weiß, wann die Berechnung abgebrochen werden soll, dann wähle ich eine `while`-Schleife.

Wenn ich das gleiche Problem habe, aber die Schleife mindestens einmal durchlaufen werden muss, dann wähle ich eine `do-while`-Schleife.

Wenn ich ganz genau weiß, wie oft meine Schleife durchlaufen werden soll, dann wähle ich eine `for`-Schleife.

If-Anweisung

```
if ()
{
      printf("hallo");
}
```

Die If-Anweisung wird immer ausgeführt, da das Argument in der Klammer als erfüllt betrachtet wird. Alles in den geschweiften Klammern wird bedingt durch die If-Anweisung ausgeführt.

```
if (a>=3)
      printf("hallo");

else
{
      printf("anderer Fall");
}
```

Hier wird die `If-Anweisung` ausgeführt, wenn `a` größer ist als `3.`
Da keine geschweiften Klammern vorhanden sind, wird nur der Teil von der `if`-Anweisung bedingt ausgeführt, der vor dem nächsten Semikolon (`;`) steht.
Das `else` wird benötigt, falls `a<=3` ist.
Wenn z.B. `a=2` ist, dann werden die Anweisungen in den `{}`-Klammern abgearbeitet.

Eine andere Schreibweise für `If-`Anweisung isr: `? :`

```
(a>=3) ? printf("hallo") : printf("anderer Fall");
```

Vergleicht das Beispiel mit der oberen `if-Anweisung`!!

switch-Anweisung

```
char c=getchar ();
switch(c)
{
     case 'E' : printf ("Lampe ein");
     case 'A' : printf ("Lampe aus");
     default : printf ("nichts passiert");
}
```

In der ersten Zeile wird ein Buchstabe mittels `getchar` in eine Variable eingelesen. Die `switch`-Anweisung ist im Prinzip ein Schalter, der entsprechend der Anweisungen in den `{}`-Klammern schaltet und anschließend alle folgenden Anweisungen in den {}-Klammern abarbeitet.
Bei `E` würde „`Lampe einLampe aus nichts passiert`“ und bei `A` würde er „`Lampe ausnichtspassiert`“ auf den Bildschirm ausgegeben werden. Bei allen anderen Zeichen würde er „`nichts passiert`“ auf dem Bildschirm ausgeben, da die `default`-Anweisung der `switch`-Anweisung die Möglichkeit gibt, eine Entscheidung zu treffen, wenn kein `E` oder `A` eingegeben wurde (vergleichbar mit `else` bei einer `for`-Schleife!!).

Wenn nun nur „Lampe ein“, „ Lampe aus“ und „nichts passiert“ ausgegeben werden soll, dann muss nach jedem Fall ein break stehen, damit die weitere Abarbeitung in den {}-Klammern abgebrochen wird.

```
char c=getchar ();
switch(c)
{
     case 'E' : printf ("Lampe ein");
          break;
     case 'A' : printf ("Lampe aus");
          break;
     default : printf ("nichts passiert");
}
```

while-Schleife

```
int c=5;

while (c>0)
{
     c-1;
}

printf ("%d",c);
```

Eintrittsbedingung in die Schleife ist `c>0`. Die Schleife wird solange ausgeführt, bis `c` kleiner als `0` ist.
Es wird alles in den `{}`-Klammern ausgeführt. Danach wird mittels der `printf`-Funktion der Wert von `c` ausgegeben (`0`).

do-while-Schleife

```
int c=5;
do
{
     c--;
} while (c>0);
printf ("%d",c);
```

Bei der `do-while`-Schleife wird die Schleife beim ersten Durchlauf ausgeführt, ohne eine Bedingung zu überprüfen. Erst nachdem die Schleife einmal durchlaufen wurde, wird überprüft, ob sie noch einmal durchlaufen werden soll.

for-Schleife

```
int i;
for (i=0 ; i<5 ; i++)
{
     printf ("Text");
     printf ("\t %d \n", i );
}
```

Bei der `for`-Schleife hat man die Möglichkeit, anzugeben, wie oft die Schleife durchlaufen werden soll.
So gilt:
`for(`*Anweisung beim Eintritt in die Schleife*`;` *Bedingung die vor Durchlauf und nach jedem Durchlauf überprüft wird* `;` *Anweisung, die nach jedem Durchlauf der Schleife ausgeführt wird*`) {` *normaler Anweisungsteil*`;}`
Beim Eintritt wird `i = 0` gesetzt, es wird geprüft, ob `i` kleiner als `5` ist, damit die die Schleife ausgeführt, bzw. ein weiteres Mal ausgeführt werden kann, nach jedem Durchlauf wird `i` um `1` erhöht.

Frage, was wird auf dem Bildschirm ausgeben?

```
Text  0
Text  1
Text  2
Text  3
Text  4
```

break – Anweisung

Die break – Anweisung beendet vorzeitig `for`, `while` und `do-while` Schleifen und Switch Anweisungen.

```
for (i=0 ; i<10 ; i++)
     if (i==5)
          break;
printf ("Ende");
```

Hier soll die `for`-Schleife 10x durchlaufen werden.
Was passiert wenn `i` den Wert `5` hat?
Die Schleife wird abgebrochen und „`Ende`“ auf dem Bildschirm ausgegeben.

continue –Anweisung

Die `continue`-Anweisung wird bei `while`-, `do-while`- und `for`-Schleifen angewendet.
Wenn `continue` ausgeführt wird, wird zum Ende der Schleife gesprungen, also zur `}`-Klammer. Danach wird die Durchlaufbedingung der Schleife überprüft.

```
int f[10]={2, -5, 17, 3, -22, 37, 9, -81, 52, -39};
int summe=0, anzahl=0;
for(int i=0; i<10; i++)
{
      if(f[i]<0) continue;
      summe+=f[i];
      anzahl++;
}
printf("Anzahl:%d \t Summe:%d", anzahl, summe);
```

Dieses kleine Programm addiert alle positiven Elemente des Feldes `f`. Ferner wir gezählt, wie viele positive Elemente im Feld enthalten sind.
Ist nun ein Feld kleiner `0` dann gilt `f[i]<0`. Es werden nun die anderen Zeilen in der Schleife bis zur `}`-Klammer übersprungen.

Frage, was gibt das Programm aus?

```
Anzahl:6   Summe:120
```

Felder

```
int k[10]={1,2,3,4,5,6,7,8,9,10};
printf("%d", k[3]);
```

In den `[]`-Klammern steht, wie viele Elemente unser Feld haben soll, in den `{}` erfolgt die Zuweisung von Daten dem jeweiligen Feld.
In den `[]`-Klammern steht quasi der Index des Feldes, vergleichbar mit dem Index von Matrizen in der Mathematik. Auf diesen Index kann auch mit Variablen zugegriffen werden, wie das folgende Beispiel zeigen wird.
Bei `k[3]` wird auf das 4. Element des Feldes zugegriffen. Warum? Weil das erste Element immer das 0. Element ist, in diesem Fall ist es das Element mit dem Wert `1`.
Bei `printf("%d",k);` wird die `1` ausgegeben, da automatisch das erste Element des Feldes gewählt wird, wenn nicht anders mittels Index angegeben ist.

Wichtig!
Wenn mit einem Index auf ein Element innerhalb eines Feldes zugreift, spricht man von **Indexnotation**!!!

```
int k[]={9,3,4,6,12,20,3};
for (i=0 ; i<5; i++)
     printf("%d",k[i]);
```

Wenn die `[]`-Klammern bei der Deklaration des Feldes leer gelassen und dabei gleich das Feld mit Werten initialisiert wird, dann wird die Länge des Feldes automatisch gesetzt.
Interessant bei diesem Beispiel ist auch in der `printf`-Funktion der Zugriff auf einzelne Elemente des Feldes mit der Variablen `i`.

Was wird ausgegeben?

```
9 3 4 6 12
```

struct

Strukturen sind an und für sich wie Felder zu handhaben, es ist klug, Strukturen vor der Main-Schleife zu deklarieren, da sonst in der `main`-Funktion bei jedem Aufruf einer Variable in der Struktur mit einem `extern` gekennzeichnet werden muss, damit der Compiler weiß, dass die Funktion später deklariert wird Bsp.: `extern char Person.Name`

```
struct irgendwas
{
     char Name[10];
     struct
     {
          char Ortsname[10];
          char Straßename[20];
          int Hausnummer;
     } Wohnung[2];
     int Alter;
} Person[10], *zeiger=Person;
```

`irgendwas` ist der Strukturtyp (man kann einen beliebigen Namen wählen).
Der Strukturtyp wird gebraucht, wenn mit Zeigern gearbeitet werden muss /soll.
(Wenn nun ein Zeiger erstellt werden soll, der auf diese Struktur zeigen soll, so muss er mit "`irgendwas *irgendeinzeiger`" deklarieret werden(`irgendwas` ist also nun ebenfalls ein Typ, genau wie `int`, `char` etc.)).
Person ist der Name der Struktur. Mit dem Index `10` wird angegeben, dass wir 10 mal diese Struktur haben möchten.

Jede Person hat einen Namen mit max 10 Buchstaben.
Jede Wohnung hat einen Ortsnamen mit max 10 Buchstaben.
Jede Wohnung hat einen Straßennamen mit max 20 Buchstaben.
Jede Wohnung hat eine Hausnummer.
Jede Person kann maximal 2 Wohnungen haben (siehe Index).
Jede Person hat ein Alter.
Unsere Struktur kann also maximal 10 Personen speichern.

Wie dem aufmerksamen Beobachter sicherlich aufgefallen ist, beinhaltet die Struktur noch eine weitere Struktur. Dies ist durchaus normal und kann beliebig weiter geführt werden (Vergleichbar mit den Ordnern auf der Festplatte von z.B. einer Musiksammlung, denn je

nach dem, wie genau die Musiksammlung sortiert ist, gibt es entsprechend viele Unterordner).

Wenn nun der Name der 2. Person ausgegeben werden soll:

```
printf("%s",Person[1].Name);
```

Der Index `1` gibt an, dass es sich um die zweite Person handelt (`0` wäre für die erste Person), der Punkt hinter dem Index gibt an, dass wir auf ein Element innerhalb der Struktur zugreifen möchten. In diesem Fall möchten wir auf die Variable Name in der Struktur Person zugreifen.

Für den Ort der 5. Person:

```
printf("%s",Person[4].Wohnung.Ort);
```

Die `[]`-Klammern bei Wohnung können entfallen, da wir die erste Wohnung haben möchten und da `Wohnung[0]` eine andere Schreibweise für Wohnung ist und das gleiche bedeutet.

enum

(engl. enumeration - Aufzählung)

```
enum {a,b,c,d} x;
```

`Enum` weist den in den `{}`-Klammern stehenden Variablen aufsteigende Werte zu (falls nicht anders durch den Programmier festgelegt wie im zweiten Beispiel), `x` kann nur die Werte von den Variablen annehmen Die Variablen besitzen nun die Werte `a=0 ; b=1 ; c=2 ; d=3;`.
X kann also nur `0` oder `1` oder 2 oder 3 sein.

```
enum {einer=1, zweier, fünfer=5, zehner=2*fünfer, fünfziger =
5* zehner } geld;
```

Hier wird Geld definiert mit den möglichen einzelnen Werten, die Geld annehmen kann, wie ein 1€ Stück, 10€-Schein etc.

Zeiger

Es ist hierbei wichtig, sich S.55 im Skript anzusehen und zu verstehen!!!

```
char c, *PC;
c='A';
PC=&c;
*PC='B';
```

Es wird eine Variable c und ein Zeiger *PC deklariert, der Zeiger muss vom selben Typ sein, wie das /die Variable, auf die er später zeigen soll (in diesem Fall vom Typ char).
Nun wird der Variablen c der Buchstabe 'A' zugewiesen.
Danach wird dem Zeiger die Adresse der Variablen c zugewiesen, der Zeiger PC zeigt also nun auf die Variable c.
Mit *PC (der Inhalt worauf der Zeiger zeigt (Dereferenzierung (siehe Adressoperatoren!!))) wird der Variablen c der Buchstabe 'B' zugewiesen, da der Zeiger auf die Variable c zeigt und mit *PC auf den Inhalt der Variablen c zugegriffen wird.

Merke:

Mit * können wir auf dass zugreifen, worauf der Zeiger zeigt.
Mit & können wir die Adressen von Variablen, Feldern etc. auslesen und in einen Zeiger speichern

```
char feld[3]={A,B,C} , *z, a;
z=&feld[0];
a=*(z+2);
```

Der Zeiger bekommt die Anfangsadresse des Feldes feld zugewiesen (alternativ könnte hier auch z=&feld; stehen). Danach wird die Adresse von z um 2 erhöht (ohne dabei z zu verändern) und mit * dereferenziert. Die Variable a enthält nun also das Zeichen C.

Man sollte noch beachten, dass bei *z+1 das, worauf der Zeiger zeigt, um 1 erhöht wird!
Im Beispiel würde dann im ersten Element des Feldes feld der Buchstabe A um eins erhöht werden, es würde dann also ein B dort stehen!!!

Zum besseren Verständnis des Beispiels und weil es elementar ist in der Programmierung, sollte man sich bei Wikipedia über den ANSII-Code informieren!

Merkzettel fürs Programmieren allgemein:

Buchstaben werden intern im PC als Zahlenwerte gespeichert.
Jeder Buchstabe besteht aus 8 Bit = ein Byte bzw. der Zahl 256 im Dezimalsystem oder maximal FF im hexadezimalen System (Zahlensystem mit der Basis 16) (Siehe ACSII-Code bei Wikipedia).
Alles in einem Computer basiert darauf, dass Wahrheitswerte und Nichtwahrheitswerte (so genannte Bool'sche Ausdrücke) sowie Buchstaben und Adressen in 1 und 0 gespeichert werden.

Wenn wir nun eine Variable definieren, sagen wir, die Variable hat einen Namen `x`, und einen Typ, z.B. `int`.
Dies bedeutet, dass die Variable Speicherplatz benötigt um eine Zahl vom Typ `int` zu speichern, also 4 Byte.
Wenn wir nun der Variablen `x` einen Wert (eine Zahl) zuweisen, (z.B. `3820`) wird diese Zahl in Binärcode (1 und 0) umgewandelt und in die 4 Byte Speicherplatz geschrieben (00 00 0E EC im hexadezimalen Code).

Wenn wir dies nun mit einem Buchstaben machen und einer Variablen vom Typ `char`, so hat auch diese Variable einen Namen, Speicherplatz der zur Variablen gehört und der PC weiß, dass durch `char v` die Werte im Speicher als Buchstaben interpretiert werden sollen.
Wenn wir der Variablen `v` einen Buchstaben zuweisen mit `v ='A'` dann wird A in eine Hex Zahl umgewandelt und in der Variablen gespeichert.
Wenn wir auf die Variable zugreifen, z.B. mit `printf ("%c", v)` dann wird der gespeicherte Wert der Variablen `c` durch das `%c` zurück in einen Buchstaben gewandelt und auf dem Bildschirm ausgegeben.

Auf diesem System basieren auch alle anderen Funktionen, Variablen und Felder etc.

Bei Zeigern z.B. (siehe Zeiger) wird ein Zeiger und eine Variable erzeugt, beide müssen vom gleichen Typ (`char`) sein, denn der PC weiß, `*PC` ist ein Zeiger enthält also auf seinem Speicherplatz eine Adresse von einer anderen Variablen.
Wenn nun mit `PC=&c;` PC die Adresse von der Variablen `c` zuwiesen wird, dann wird durch den Operator & die Variable `c` in Ihre Adresse "aufgelöst". Es steht dann also z.B. `PC= 0f3e4a` (im hexadezimalen System).
Da PC ein Zeiger ist, wird dieser Wert auf dem Speicherplatz von PC gespeichert und kann jederzeit als Adresse wieder abgerufen werden, da PC ein Zeiger ist und wir mit `&c` die Adresse von `c` abgerufen haben.

Wenn wir nun `*PC= 'B'` haben, dann wird der Buchstabe B in eine Hex Zahl umgewandelt, und an `*PC` weitergegeben.
Da `*PC` vom Typ `char` ist, weiß der PC, "aha, darauf, worauf der Zeiger zeigt, handelt es sich um Buchstaben".
Da `*PC` ein Zeiger ist, der wird mit dem `*` Operator die in `*PC` gespeicherten Adresse von `c` aufgerufen. Intern steht da also nun " `c='a'` " bzw. "`c=` (wert vom Buchstaben `a` in hex)".
Durch `*PC ='a'` wird also der Variablen `c` der Wert des Buchstaben `a` zugewiesen.

Zeiger und Strukturen

Zeiger (->)werden verwendet um auf Elemente innerhalb einer Struktur zuzugreifen, wenn die Struktur über einen Zeiger adressiert wird (siehe Beispiel).

```
struct typperson
{
      char Name[10];
      struct
      {
            char Ort[10];
            char Straße[20];
            int Nummer;
      } Wohnung;
      int Alter;
} Person[10], *pPer = Person;

for(int i=0; i<10; i++)
{
      printf("%s %d", pPer->Name, pPer->Wohnung.Nummer);

      pPer++;
}
```

Es wird eine Struktur vom Typ `typperson` erzeugt (für weiteres siehe Strukturen).

Interessant bei diesem Beispiel ist allerdings, dass hier auf die Elemente der Struktur zugegriffen werden kann, ohne Indexnotation zu verwenden.

Dies ist dadurch möglich, weil ein Zeiger `pPer` vom Typ `typperson` existiert, der auf das erste Element der Struktur Person zeigt.
Wenn nun die `for`-Schleife näher betrachtet wird, kann leicht erkennt werden, dass die Schleife 10mal durchlaufen wird.

Bei der `printf`-Funktion festgestellt werden, dass der Pfeiloperator (Zeiger (`->`)) zwischen dem Zeiger `pPer` und einem Element der ersten Struktur steht.
Dies bedeutet, dass wenn auf ein Element zugegriffen werden soll, welches sich innerhalb einer Struktur befindet und die Struktur mit Zeigern adressiert wird, einen Pfeiloperator verwenden muss um auf dieses Element zugreifen zu können.
Soll nun auf ein Element einer Struktur zugegriffen werden, die sich innerhalb der vom Zeiger adressierten Struktur befindet (in diesem Fall ein Element innerhalb der Struktur `Wohnung`), so wird der Zeiger `pPer` verwendet und lässt ihn mittels Pfeiloperator `->` auf den Namen der inneren Struktur zeigen (welche ein Element der ersten Struktur ist).
Mit einem Punkt und dem Namen des Elements innerhalb der zweiten Struktur kann nun auf dieses Element zugegriffen werden (`pPer->Wohnung.Nummer`).

Funktionen

Funktionen sind quasi kleine Programme, die vom Programmierer geschrieben werden können und innerhalb des Programmcodes mit Parametern aufgerufen werden, die Parameter entsprechend der Anweisungen innerhalb der Funktion berechnen und das Ergebnis als vorher definierten Rückgabewert an die aufrufende Funktion zurück liefern.

```
int _multi (int k, int i)
{
      int t;
      t =k*i;
      return t;
}

void main (void)
{
      int k=4, i=3, z=0;
      z=_multi(k, i)
      printf("%d", z);
      return 0;
}
```

Die hier programmierte Funktion heißt `_multi` und liefert einen Wert vom Typ `int` zurück (es ist ratsam, Funktionen mit einem Unterstricht (_) am Anfang zu benennen, da sie so leichter von Variablennamen unterschieden werden und somit leichter Fehler vermieden werden können).
Ferner wird unserer Funktion beim Aufruf zwei Parameter vom Typ `int` (`k` und `i`) übergeben.
In den geschweiften Klammern stehen nun die Anweisungen, welche die Funktion ausführen soll.
Mit `return` wird der Wert von `t` (`12`) an die aufrufende Funktion übergeben.
Ein `return` muss immer vorhanden sein.
Soll kein Wert zurückgeben, so muss `return 0` verwendet werden.

Die nun folgende `main`-Funktion ist die Hauptfunktion die in jedem Programm vorhanden ist und von der aus alle anderen Funktionen aufgerufen werden.

Innerhalb der `main`-Funktion werden nun drei Variablen deklariert und definiert.
In der folgenden Zeile wird die Funktion `_multi` aufgerufen mit den Werten 4 und `3.`
Der Rückgabewert der Funktion, welchen die Funktion zurückliefert, wird der Variablen `z` zugewiesen.
Ferner ist zu beachten, dass `z` vom gleichen Typ (`int`) ist wie der Rückgabetyp der Funktion (andernfalls ist Typecasting erforderlich!!!).
Danach wird der Wert der Variablen `z` ausgegeben (`12`).

Wenn die Funktion nach ihrem Aufruf keinen Werte zurück geben, oder bei ihrem Aufruf Parameter übergeben bekommen soll, so wird `void` vor die Funktion oder in die Klammern hinter dem Funktionsname geschrieben.

Warum werden Funktionen überhaupt gebraucht?
Wenn z.B. der Betrag eines Vektors berechnet werden und nicht jedesmal im Quelltext der komplette Berechnungsalgorithmus einfügt / geschrieben werden soll, dann wird einfach eine Funktion geschrieben, in der der Algorithmus steht und welche an den entsprechenden Stellen mit den aktuellen Parametern aufrufen wird um als entsprechenden Rückgabewert den Betrag des Vektors zu bekommen.

Es ist klug, Funktionen zu programmieren, denn damit hält wird der Quelltext kurz und übersichtlich gehalten, ferner ist der Quelltext einfacher nachzuvollziehen.

main() Funktion-.

Siehe Skript S.69

Stringfunktionen-.

Siehe Skript Seite 64ff

Achtung, für Stringfunktionen muss die `string.h` am Anfang des Programms includiert werden!!!!

Es ist sehr wichtig, sich die Stringfunktionen anzuschauen, denn wer sie kennt, kann sich sehr viel Arbeit und Zeit sparen! (sind aber in der Abschlussklausur nicht erlaubt).

Dateifunktionen-.

Siehe Skript S. 67

Dynamische Speicherverwaltung:

Wozu brauche ich dynamische Speicherverwaltung?

Wenn Programme für normale Computer geschrieben werden, die recht üppig mit Speicherplatz versehen sind, ist es eigentlich nicht soo wichtig, dynamischen Speicherplatz zu verwenden.
Doch wenn für einen Mikroprozessor programmiert werden soll, der nur wenige Kilobytes Speicherplatz hat, dann ist es sehr wichtig, sparsam mit dem Speicherplatz umzugehen.
Aus diesem Grunde wird der Speicherplatz dynamisch verwendet.
Es wird nur soviel Speicherplatz reserviert, wie benötigt. Speicherplatz nicht mehr benötigt wird, wird wieder freigegeben.
Wenn dies ignoriert wird, kann es vorkommen, dass der PC etc. nicht mehr richtig funktioniert, weil der gesamte Speicher voll ist.
In diesem Fall hilft dann meistens nur ein Neustart der gesamten Maschine. Da dies aber in der Industrie u.U. mit enormen Kosten verbunden ist, ist es ratsam, sich schon jetzt an die dynamische Speicherverwaltung zu gewöhnen.
Wer dies beachtet, wird später weniger bis keine Probleme mit Programmabstürzen aufgrund von fehlerhafter Speicherverwaltung haben.

1. sizeof()

Die `sizeof()`-Funktion liefert die benötigte Größe des Speicherplatzes, des in den Klammern stehenden Ausdrucks in Byte an die aufrufende Funktion.

```
int z=0;
z=sizeof(char);
printf("%d",z);
```

In diesem Beispiel wird die Größe des benötigten Speicherplatzes in Bytes (8 Bit) eines Zeichens vom Typ `char` in `z` geschrieben. In der folgenden Zeile wird die Größe des benötigten Speicherplatzes ausgegeben (`1`).
Probiert es selbst und testet mal, wie viele Bytes ein `int`-Wert, ein `long int`-Wert etc. haben.

2. malloc()

Die `malloc`-Funktion reserviert n*Bytes Speicherplatz im Heap (der Arbeitsspeicher des PC's (RAM) ist in Heap (Kurzzeitgedächtnis) und Stack (längeres Kurzzeitgedächtnis) geteilt). Die Größe des Speicherplatzes wird durch eine Zahl in den `( )` Klammern bestimmt.
Die Funktion liefert die Anfangsadresse des Speicherplatzes zurück an die aufrufende Funktion oder `0`, wenn kein Speicherplatz reserviert wurde.
Wenn die Anfangsadresse an einen Zeiger übergeben werden soll, ist ggf. Typecasting erforderlich.

```
char *pc;
pc=(char *) malloc(10);
```

Bei diesem Beispiel werden 10 Bytes Speicherplatz reserviert. Da `pc` ein Zeiger ist, der vom Typ `char` ist, muss die Adresse, welche `malloc` zurück gibt, getypecastet werden. Der Zeiger `pc` enthält nun also die Anfangsadresse eines 10 Bytes großen Speicherplatzes vom Typ `char`.

Eine andere Möglichkeit wäre auch:

```
pc=(char *) malloc(sizeof(10*char))
```

Hier wird zunächst die Größe des benötigten Speicherplatzes von 10 Zeichen berechnet, dann reserviert (man spricht dann von allokieren), getypecastet und die Adresse an den Zeiger pc übergeben.

3. realloc()

Die realloc Funktion funktioniert wie die malloc Funktion, nur dass sie keinen Speicherplatz neu erstellt, sondern die Größe des bereits erstellen Speicherplatzes verändert.

```
char *pc;
pc=(char *) malloc(10);

pc=(char *) realloc(15);
```

Bei diesem Beispiel wird zunächst Speicherplatz für 10 Zeichen erstellt. Danach wird der Speicherplatz um 5 Zeichen mittels realloc erweitert.
Der Zeiger PC zeigt also nun auf die Anfangsadresse einer Zeichenkette von 15 Zeichen.

4. free()

Mit free() wird allokierter Speicherplatz wieder freigegeben.

```
free(pc);
```

Es wird der Speicherplatz des Zeigers pc wieder freigegeben.

5. new – Operator

Der new – Operator reserviert, ähnlich wie malloc, Speicherplatz im Heap.
Allerdings liefert / berechnet new automatisch den benötigten Speicherplatz und gibt die Anfangsadresse darauf zurück und somit ist kein Typecasting erforderlich.

```
int *Zahl;
char *pc;
Zahl=new int;
pc=new char[100];
```

Bei diesem Beispiel wird die Adresse des Speicherplatzes vom Typ int im Zeiger Zahl gespeichert.
Danach bekommt der Zeiger pc die Anfangsadresse von einem char-Feld mit 100 Zeichen zugewiesen.

6. delete – Operator

Der delete - Operator funktioniert im Prinzip wie die free-Funktion, nur dass die ()-Klammern fehlen und bei zu löschenden Feldern die []-Klammern vor dem Namen des Feldes stehen müssen.

```
Zahl=new int;
pc=new char[100];

delete Zahl;
delete[ ] pc;
```

Verkettete und ringverkettete Listen

Bei verketteten Listen handelt es sich um nichts weiter, als um eine Struktur, in der ein Inhalt einer Struktur einen Zeiger mit der Adresse auf die nächste Struktur desselben Typs enthält.
Bei ringverketteten Listen ist es das gleiche Prinzip, nur dass der Zeiger des letzten Elementes die Adresse der ersten Struktur beinhaltet.
In diesem Fall ist es ratsam, die Adresse der ersten Struktur in einer Variable zwischen zu speichern (diese Variable wird meistens als Anker bezeichnet) um nach dem Erstellen der Kette dem letzten Zeiger der letzten Struktur diese Adresse zu zuweisen(damit wäre dann der Kreis / Ring geschlossen).

```
struct element
{
     int wert;
     element *next;
};

element *anfang, *zeiger;
zeiger=(element *) malloc(sizeof(element));
anfang=zeiger;

for(int i=0;i<4;i++)
{
     printf("Wert: ");
     scanf("%d",&zeiger->wert);

     zeiger->next=(element *) malloc(sizeof(element));
     zeiger=zeiger->next;
}

printf("Wert: ");
scanf("%d",&zeiger->wert);
zeiger->next=anfang;
```

Als erstes wird eine Struktur vom Typ `element` erstellt.
Die Struktur enthält eine Variable vom Typ `int`, die `wert` heißt und einen Zeiger `next`, welcher vom Typ `element` ist.
Danach werden die Zeiger `*anfang` und `*zeiger` mit dem Typ `element` deklariert.

In der darauf folgenden Zeile wird zunächst die mittels `sizeof` die Größe des benötigten Speicherplatzes der Struktur ermittelt. Danach wird Speicherplatz allokiert (reserviert), die Adresse welche `malloc` liefert getypecastet und anschließend im Zeiger `zeiger` gespeichert.
Da dies das erste Element der Kette ist, kopieren wir die Adresse von `zeiger` in den Zeiger `anfang` (wir setzen also den Anker).

In der folgenden `for`-Schleife werden weitere Kettenglieder erzeugt. Dabei werden mittels `scanf` die Werte für die Variable `wert` eingelesen und anschließend bekommt der Zeiger `next` die Adresse des nächsten Elements der Kette, bzw. der nächsten Struktur. Anschließend bekommt der Zeiger `zeiger` die Adresse des nächsten Elements.
Dies geschieht 4 mal.
Danach wird in noch einmal dem letzten in der Kette erzeugten Element ein Wert eingelesen und dem Zeiger `next` die Adresse des ersten Elements der Kette zugewiesen.
Die Ringverkettete Liste ist nun vollständig.

Wenn es keine Ringverkettete Liste sein soll, sondern nur eine verkettete Liste, müsste in der letzten Zeile "`zeiger->next=NULL;`" stehen.

Warum brauche ich ringverkettete Listen?

Ringverkettete Listen sind ähnlich wie dynamisch verwaltete Variablen / Felder / Strukturen zu betrachten.
In einer ringverketteten Liste können einzelne Elemente gelöscht, hinzugefügt und beliebig verändert und damit der Speicherplatz verwaltet werden.
Dies erleichtert den Zugriff auf einzelne Elemente der Kette, sowie die Verwaltung der gesamten Kette.
Der Aufwand eine ringverkettete Liste zu verwalten ist um einiges geringer, als wenn das ganze mit konventionellen Strukturen / Feldern realisiert werden müsste.
Es gibt einfach mehr Möglichkeiten, bei vergleichsweise weniger Aufwand.

Rekursive Funktionen

Rekursive Funktionen beruhen auf dem Prinzip, dass sie sich selber aufrufen (vergleichbar mit einer Kamera, die filmt, was sie sieht), bis ein gewisser Punkt erreicht wird (bei der Kamera ist die Auflösung endlich, deswegen wäre dies der Punkt zum aufhören).
Wenn dies der Fall ist, werden die aufgerufenen Funktionen geschlossen und dabei die zurückgegebenen Werte verarbeitet. Auf diese Art und Weise kann mit sehr wenigen Programmzeilen, eine sonst schwierig zu programmierbare Funktion gelöst werden.

Beispiel:

Schreiben Sie eine rekursive Funktion, welche die Summe von (`i=1`) bis `n` berechnet. (1+2+3+4+5+...+n)

```
int sum (int n)
{
      int x=n;

      if (n==0)
            return 0;

      x = x + sum(n-1);

      return x;
}
```

Der Funktion `sum` wird beim Aufrufen der Parameter `n` übergeben. Nach Beenden der Funktion gibt sie einen `int` Wert zurück.

Dies sind die Randdaten, die wir wissen müssen, um unsere Funktion zu schreiben.

Zunächst wird der Variablen x der aktuelle Wert von n zugewiesen.
Danach wird die Abbruchbedingung geschaffen, wenn n bei 0 ist. Wenn dem so ist, wird eine Null zurück gegeben.
Anschließend kommt die charakteristische Zeile einer rekursiven Funktion.
In dieser Zeile wird zu x der Wert addiert, der von der neu aufgerufenen Funktion sum mit dem Übergabeparameter n-1 zurückgegeben wird.

Die Funktion ruft sich also solange selber auf, biss die Abbruchbedingung n==0 erfüllt ist. Wenn dem so ist, wird der vorige x-Wert mit 0 addiert, der Wert dann mit dem x-Wert darüber usw.

Beispiel: n=5

	1.Aufruf	2.Aufruf	3.Aufruf	4.Aufruf	5.Aufruf	6.Aufruf
z= sum(5);	x=5; x=x+ sum(4);	x=4; x=x+ sum(3);	x=3; x=x+ sum(2);	x=2; x=x+ sum(1);	x=1 ; x=x+ sum(0);	x=0; Abbruch
z=15; <-	x=5+10	x=4+6 <-	x=3+3 <-	x=2 +1 <-	x=1+0 <-	0 <-

Die Variable z bekommt den Wert 15 zugewiesen.

Grundsätzlich gilt, dass bei einer rekursiven Funktion das letzte Ergebnis (hier 15) aus vielen kleinen Teilergebnissen von vielen kleinen Funktionsaufrufen besteht.
Oder mit anderen Worten bei rekursiven Funktion wird Rückwärts auf das Ergebnis zugegangen, normalerweise rechnet ein Mensch mit 1+2+3+4+...+n.
Bei dieser rekursiven Funktion wurde n zunächst zerteilt 5 4 3 2 1 0 und dann von 0+1 =1+2=3+3=6+4=10+5 zusammen addiert.
Also erst rückwärts gegangen und dann vorwärts addiert.

Iterative Funktionen

Iterative Funktionen sind Funktionen, wie wir sie als normal bezeichnen würden.
Es sind Funktionen, die mit for-, while- Schleifen etc. arbeiten.

Iterativ würde die Funktion aus dem Beispiel der rekursiven Funktionen wie folgt aussehen:

```
int sum(int n)
{
      int k=0;

      if (n==0)
            return 0;
      for (int i=0; i<=n; i++)
            k=k+i;

      return k;
}
```

Bei diesem Beispiel zu der Variable das aufwärts zählende i solange dazu addiert, bis i größer ist als n.

Wenn `i` dann größer ist, wird die `for`-Schleife abgebrochen und der Wert von `k` an die aufrufende Funktion zurück gegeben.
Die `if`-Schleife dient als Sicherheit falls einmal eine `0` eingegeben werden sollte.

Achtung!
Immer darauf achten, alle möglichen Eventualitäten mit in die Funktion einzubeziehen, andernfalls sind Programmabstürze quasi vorprogrammiert, und in einem längeren Quelltext einen solchen Fehler zu finden, ist sehr, sehr Zeitaufwendig und damit teuer!!!!