Einführung zur Aufgabengruppe 1

- **Einige Gesichtspunkte zur Softwareentwicklung**
- **Normierte Methoden zur graphischen Darstellung von Programmen**
- **Beispiel-Aufgabe**
- **Strukturierte Datentypen**

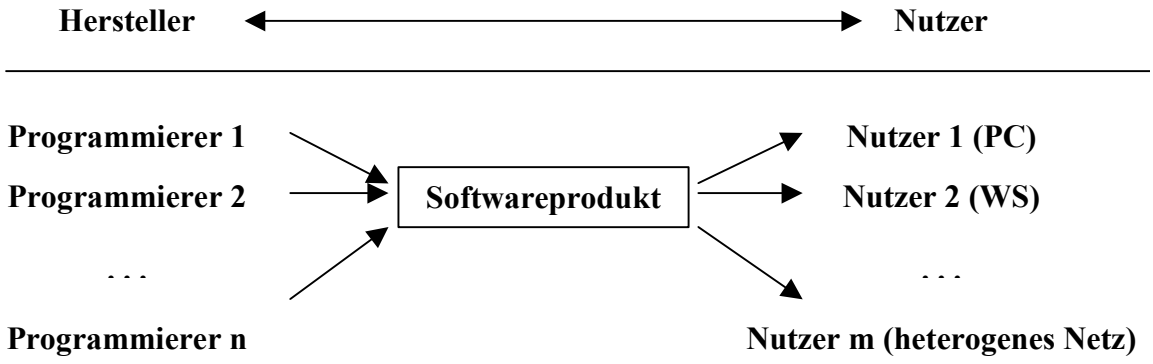
Einige Gesichtspunkte zur Softwareentwicklung

Geschichte
In der Anfangszeit des Programmierens, bis in die sechziger Jahre hinein, gab es keine Programmiermethodik, keine Techniken und keine Regeln für das fachmännische Schreiben größerer Programme; jeder Programmierer hatte vielmehr seine eigene Vorgehensweise.
Dabei wurden jedoch zu dieser Zeit bereits enorm große Programme geschrieben.
Die Komplexität dieser großen Programme wuchs ihren Entwicklern über den Kopf; man konnte sie kaum noch beherrschen. Ein durchaus üblicher Programmquelltext von 100'000 Anweisungen ergibt einen Programmtext von 2'000 Seiten, wenn man eine Anweisung pro Zeile und die Seite mit 50 Zeilen ansetzt. Dabei werden erläuternde Kommentare, die bei der Größe des Programms unabdingbar sind, noch nicht einmal berücksichtigt.
Die Fehlerhäufigkeit war groß; bei der Beseitigung der gefundenen Fehler schlichen sich neue ein.

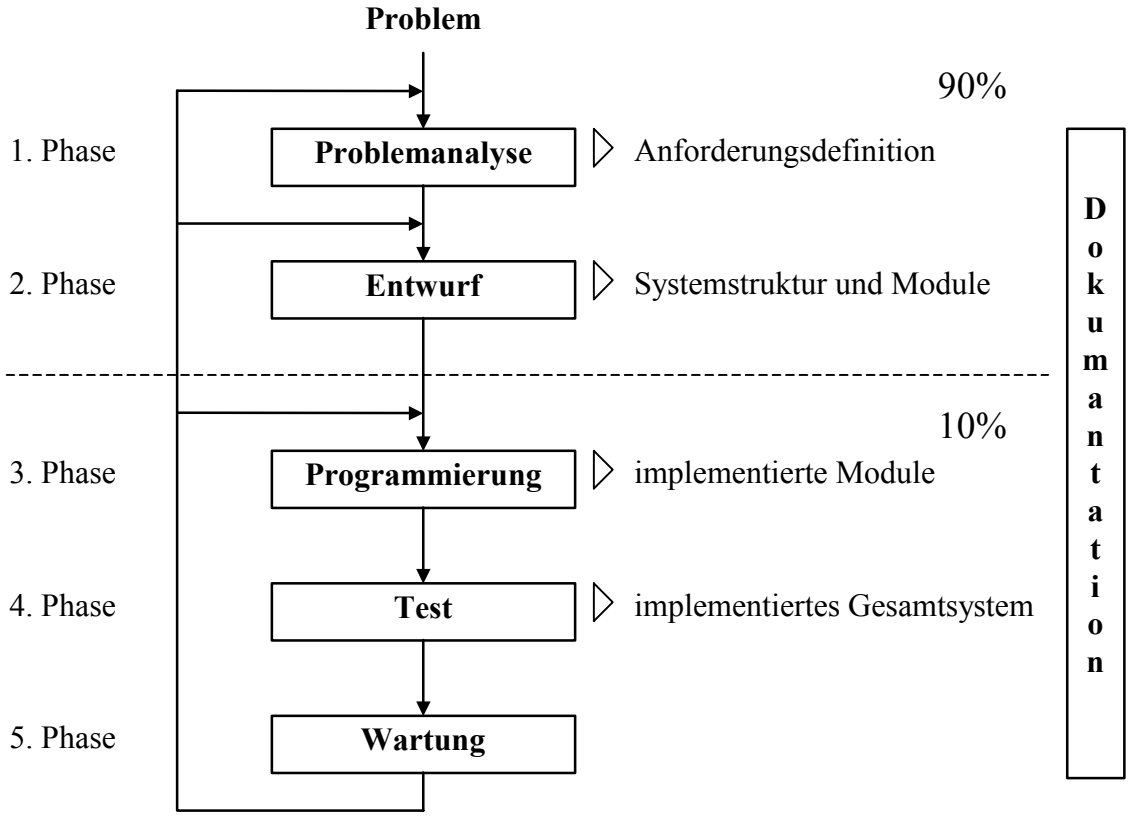
Ende der sechziger Jahre fanden zwei von der NATO ausgerichtete Tagungen zu diesem Thema statt:
- 1968 in Garmisch
- 1969 in Rom

Auf ihnen wurden erstmalig die Probleme der Entwicklung großer Programme explizit benannt und diskutiert. Man stellte fest, dass Software das Ergebnis von Ingenieurtätigkeit ist, die, wie jedes industrielle Produkt, der methodischen Planung, Entwicklung, Herstellung und Wartung bedarf. So entstand ein neues Wissenschaftsgebiet, welches heute unter dem Begriff ***Software Engineering*** bzw. ***Softwaretechnik*** läuft.

Was zeichnet große Programme aus?
- Hersteller und Nutzer sind verschiedenen Personengruppen.
- Viele Programmierer müssen bei der Herstellung eines Softwareproduktes zusammenarbeiten.
- Zuverlässigkeit, Flexibilität und Übertragbarkeit auf andere Maschinen sind wichtige Qualitätskriterien.
- Große Programme haben eine größere Lebensdauer und müssen über längere Zeit gewartet werden.



Software-Lebenszyklus:



Bei Äderungen müssen einige oder gar alle Phasen noch einmal durchlaufen werden.

1. Phase: ***Problemanalyse.*** Das zu lösende Problem wird zusammen mit dem Auftraggeber definiert und analysiert. Das Ergebnis ist die *Anforderungsdefinition* (*Pflichtenheft*):

   - *Leistungsinhalt* und *Leistungsumfang* müssen gemeinsam festgelegt werden.

   - Das Projekt unterliegt, wie jedes andere technische Projekt, einer *Planung* mit Ist-Zustandsanalyse, Systemabgrenzung, Systembeschreibung u. a. m.

2. Phase: ***Entwurf.*** Das Softwaresystem wird als Ganzes entworfen und in Teile zerlegt (*Grobentwurf*), mit den Teilen wird ebenso verfahren (*Feinentwurf*); die dabei entstehenden Schnittstellen werden festgelegt (*Spezifikationen*). Das Ergebnis ist die Systemstruktur und die Spezifikationen der Teile (*Module*):

   - *Entwurfsmethoden*. Die Methode der schrittweisen Verfeinerung propagiert einen Entwurf „von außen nach innen“ oder auch „von oben nach unten“ (*top down*). Man zerlege das Gesamtproblem in Teilprobleme, diese wieder, jedes für sich, in kleinere und setze dieses Verfahren fort, bis die Teilprobleme so klein sind, dass man ihre Lösung in einer Programmiersprache formulieren kann.

3. Phase: ***Programmierung.*** Die Module werden programmiert und jedes für sich getestet:

   - *Modulares Programmieren*. Jeder Modul führt eine in sich geschlossene Aufgabe aus, und wenn er fehlerhaft ist, wird er als Ganzes gegen einen anderen ausgetauscht. Dazu muß jeder Modul eine festgelegte *Schnittstelle* zu den anderen Modulen besitzen, und alle Informationen müssen über diese Schnittstelle laufen.

   - *Defensives Programmieren*. Der Softwareingenieur muß damit rechnen, daß in seinem Programm trotz sorgfältigsten Testens Fehler zurückbleiben, die sich oft erst Jahre nach der Inbetriebnahme herausstellen. Er muß deshalb zusätzlichen Code zur Fehlerdiagnose von vornherein in seine Programme aufnehmen, der auch nach dem Testende im endgültigen Code verbleibt. Zusätzlicher Kommentar erleichtert die spätere Fehlersuche.

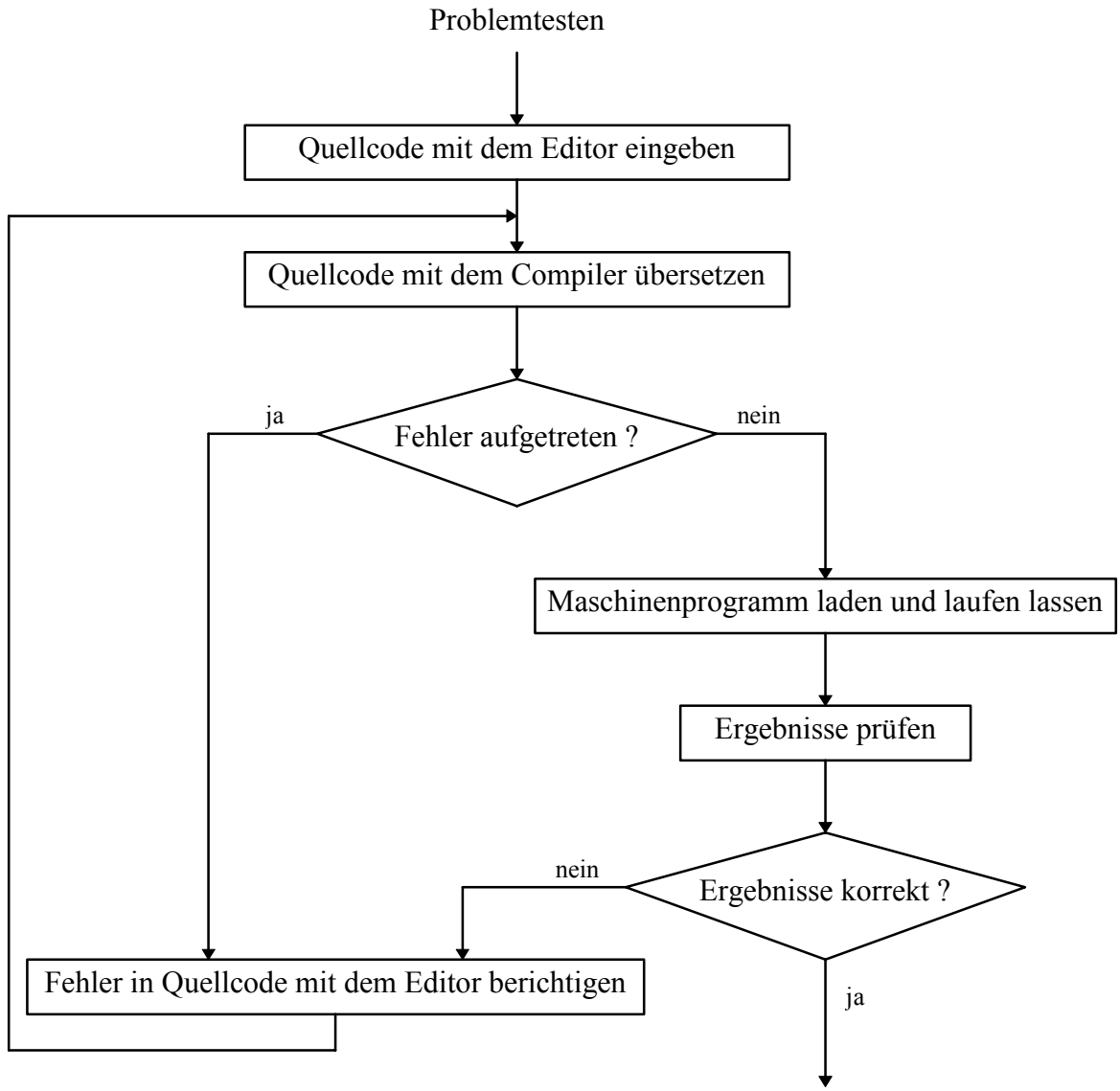
4. Phase: ***Test.*** Die Zusammensetzung der Module zu Gruppen und das Gesamtsystem werden getestet. Das Ergebnis ist das *implementierte Gesamtsystem* mit dem Ziel der Abnahme durch den Auftraggeber:

   - Testen ist das Prüfen eines Programms, ob es den vorgeschriebenen Anforderungen erfüllt, insbesondere daraufhin, ob es noch Fehler enthält. Da das Testen immer nur die An- oder Abwesenheit bestimmter Fehler, niemals aber die vollständige Fehlerfreiheit eines Programm zeigen kann, hat man das Testen auch ausdrücklich als *das Suchen nach Fehlern* definiert.

   - Um eine Prozedur oder eine Gruppe von Prozeduren zu testen, braucht man ein Testprogramm, welches den Prüfling mit geeigneten Testbeispielen als Parameterwerten wiederholt aufruft.

   - Testprogramme und Testbeispiele sollten im System verbleiben. Sie stehen dann bei Wartungsaufgaben wieder zur Verfügung.

Testzyklus:



5. Phase: ***Wartung.*** Im Betrieb entdeckte Fehler werden beseitigt, und das Programmsystem wird den sich verändernden Anforderungen angepasst:

   - Während des Betriebs des Softwaresystems treten i.R. noch Fehler auf. Diese und auch Anpassungsaufgaben sind Aufgaben, die auch nach der Übergabe an den Auftraggeber für den Softwareingenieur anfallen können. Eingebaute Fehlercode und Testprogramme sind hier ein äußerst nützliches Hilfsmittel.

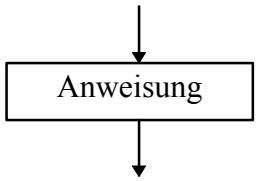
Dokumentation
Software ist, wie alle technischen Produkte, auf Dokumentation angewiesen. Alle Phasen eines Softwareprojekts sollen von Dokumentationen begleitet sein.

Normierte Methoden zur graphischen Darstellung von Programmen

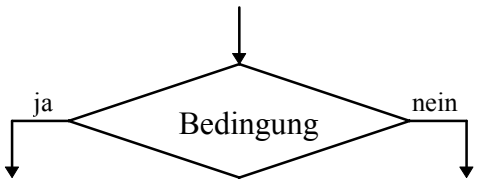
1. Programmablaufpläne (*Ablaufdiagramm, Flußdiagramm*):

Diese sind mit der Maschinenprogrammierung entstanden und heute kaum noch üblich.

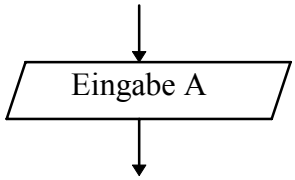
Anweisung:



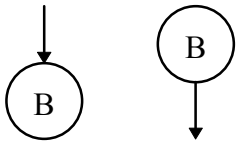
Verzweigung:



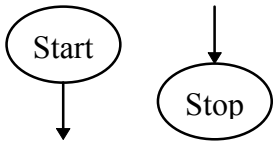
Eingabe bzw. Ausgabe:



Übergangsstellen:



Anfang und Ende:



Vorteile:
- Übersichtliche Darstellung bei kleineren Programmen.
- Ein PAP läßt sich schnell erstellen (evtl. Vereinfachung durch Weglassen einiger Grafiksymbole).

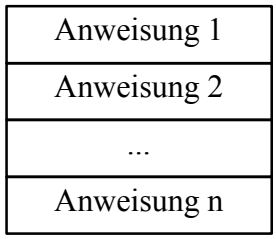
Nachteile:
- Durch die maschinenorientierte Darstellung der Sprungstruktur eines Programmes wird ein PAP schnell unübersichtlich (*Spagetti-Programme*). Es fehlen Darstellungsmittel für Schleifen.

2. Struktogramme (*Nassi-Schneidermann-Diagramm*):

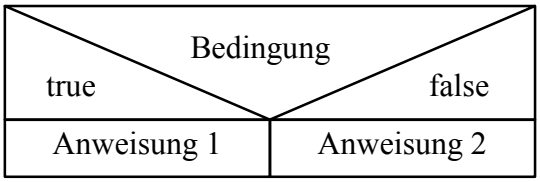
Modernere Darstellung von Programmen.

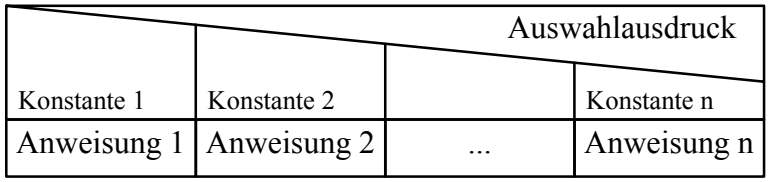
Anweisung:

Zusammengesetzte Anweisungen (Anweisungsblock):

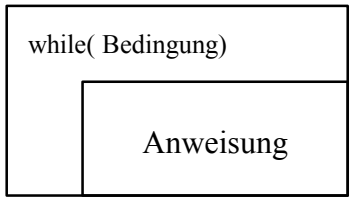


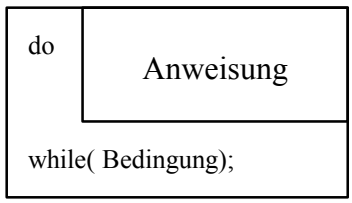
Auswahlanweisungen:

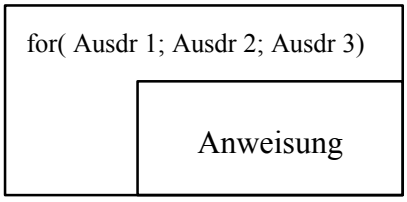




Schleifenanweisungen:







3. Pseudosprachen (*Algorithmenbeschreibungssprache*):

Ablaufdiagramme und Struktogramme sind anschaulich und zu einem gewissen Grade übersichtlich, jedoch durch ihre zweidimensionale Darstellung und die damit verbundene geometrischen Elemente unhandlich. Eine Zwischenstufe zu den grafischen Darstellungen und den Programmiersprachen bilden die Algorithmenbeschreibungssprachen. Diese sind den Programmiersprachen sehr ähnlich, aber einfacher als diese und enthalten keine zeichnerischen Elemente mehr.

Es gibt verschiedene solcher Beschreibungssprachen. Oft verwendet man die leicht verständlichen Befehle der Algorithmenbeschreibungssprache *Adele* (*a*lgorithm *de*scription *l*anguage). Diese ist sehr Pascal ähnlich und damit für C-Programmieranfänger nicht so gut geeignet. Wie verwenden hier eine etwas C nähere Form vom *Adele*. Die wichtigsten, aber noch ausbaufähigen Befehle seien:

Anweisung: *Ausdruck***;**

{ *Anweisung Anweisung ... Anweisung* **}**

Auswahlanweisungen: **if(** *Bedingung* **)** *Anweisung*

if(*Bedingung* **)** *Anweisung* **else** *Anweisung*

Schleifenanweisungen: **while(** *Bedingung* **)** *Anweisung*

do *Anweisung* **while(** *Bedingung* **);**

for(*Ausdruck 1***;** *Ausdruck 2***;** *Ausdruck 3***)** *Anweisung*

Bei der Programmentwicklung kann man alle drei Darstellungsformen kombinieren: Man beginnt oft beim Grobentwurf mit einer übersichtlichen grafischen Darstellung und wechselt im Feinentwurf zur Preudosprache.

Beispiel-Aufgabe

Skalarprodukt zweier Vektoren

Programm zur Berechnung des Skalarproduktes zweier Vektoren im reellen Raum.

Eingabe:
Zwei Vektoren gleicher Länge werden komponentenweise eingegeben. Durch das Dateiendezeichen EOF (^d) wird die Eingabe abgeschlossen.

Ausgabe:
Die beiden Vektoren und ihr Skalarprodukt werden ausgegeben.

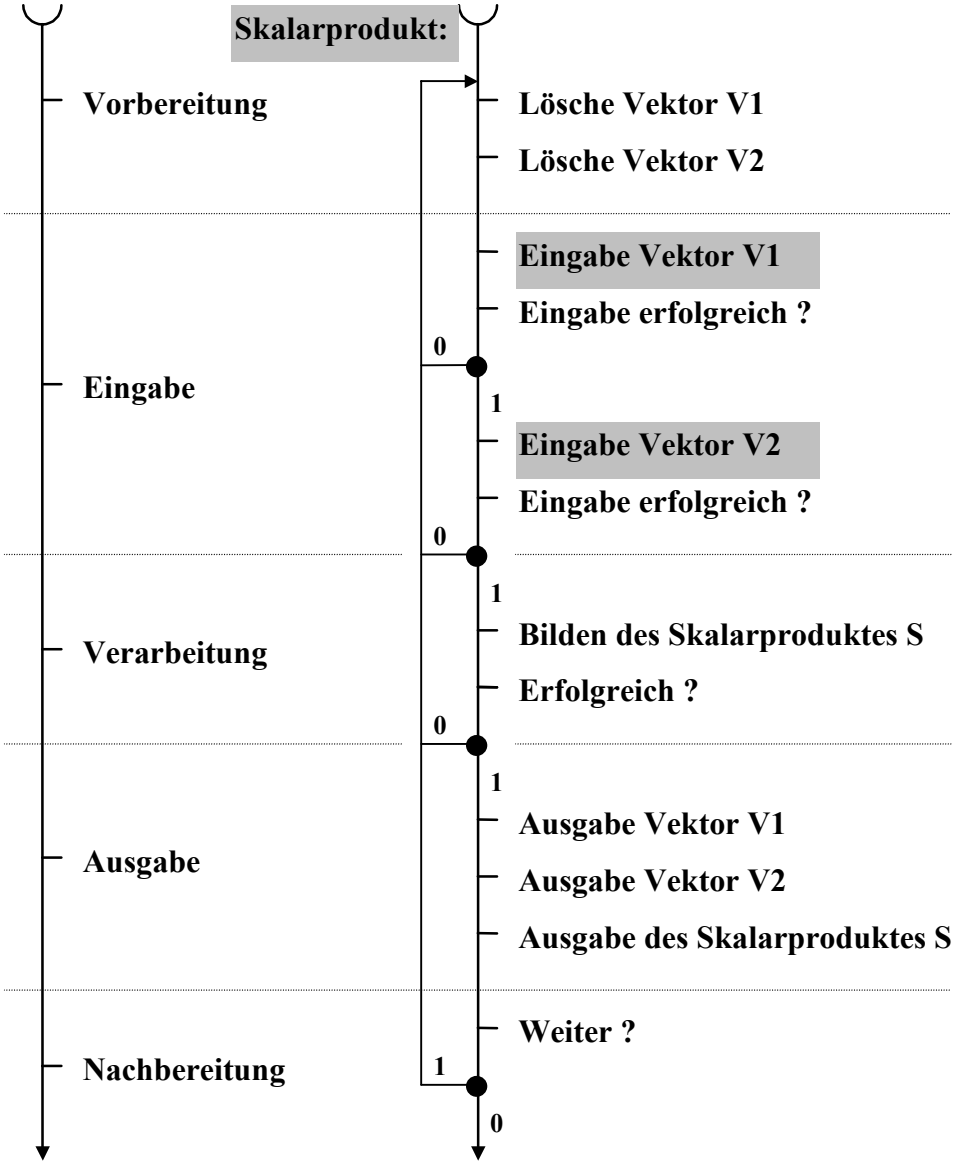
Abbruch:
Das Programm bricht ab, falls der Nutzer keine neue Berechnung wünscht.

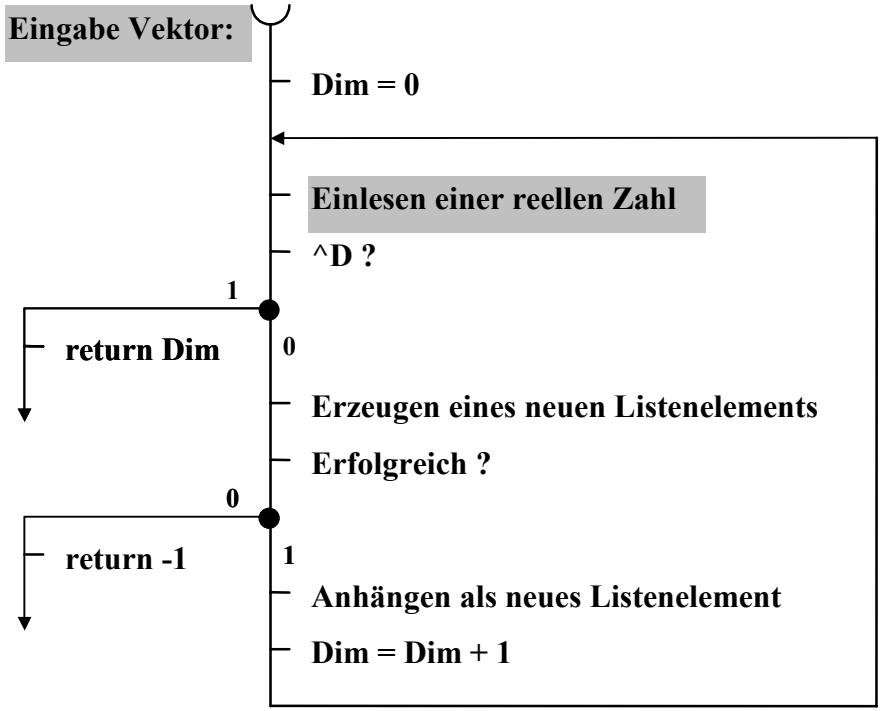
Datentypen:
Die reellen Vektoren werden dynamisch als einfach verkettete Listen abgelegt.

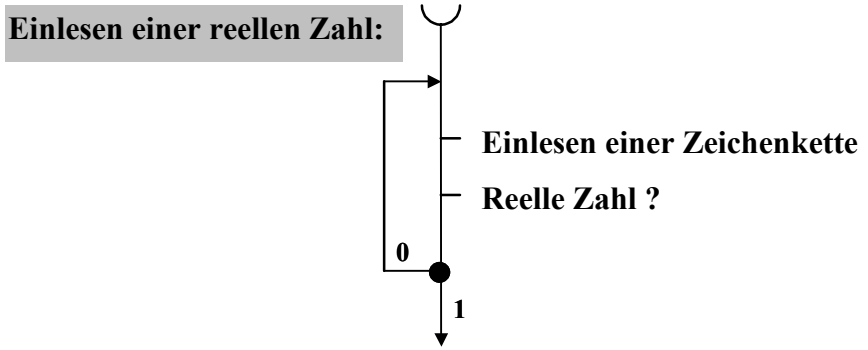
Funktionen:
Eingabe, Ausgabe und Berechnung des Skalarprodukt sollen als Funktionen programmiert werden.
Des Weiteren soll eine Funktion zur Eingabe einer reellen Zahl absichern, dass Fehleingaben nicht zum Absturz des Programms führen. In der Standardbibliothek <stdlib.h> finden Sie entsprechende Umwandlungsfunktionen von einer Zeichenkette in verschiedene Zahlentypen.

Bemerkung zum Verfahren:
Das Skalarprodukt zweier Vektoren $\vec{x}$, $\vec{y}$ eines n-dimensionalen Vektorraumes wird durch $\vec{x} \cdot \vec{y} = \sum_{i=1}^{n} x_i y_i$ mit $\vec{x} = (x_1, \cdots, x_n)$ und $\vec{y} = (y_1, \cdots, y_n)$ berechnet.

PAP zum Beispiel:

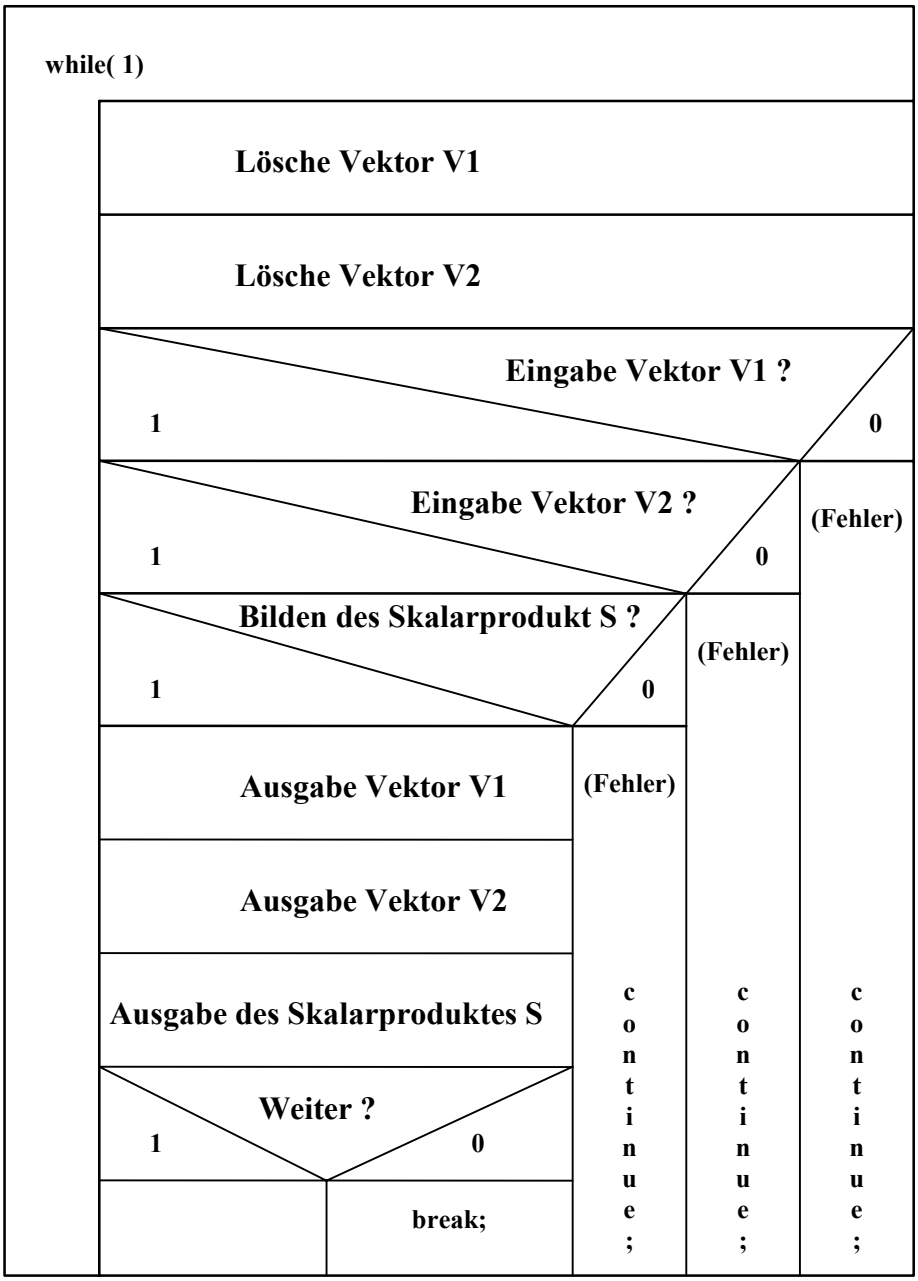






Struktogramm zum Beispiel

Skalarprodukt:



Pseudosprachbeschreibung zum Beispiel

```
while( 1)
{ Lösche_Vektor( V1);
  Lösche_Vektor( V2);
  if( Eingabe_Vektor( V1) fehlerhaft) continue;
  if( Eingabe_Vektor( V2) fehlerhaft) continue;
  if( S = Scalarprodukt( V1, V2) fehlerhaft) continue;
  Ausgabe_Vektor( V1);
  Ausgabe_Vektor( V2);
  Ausgabe_Skalarprodukt( S);
  if( !Weiter()) break;
}
```

Schnittstellen:

Eingabe eines Vektors: **vec.c vec.h**

Name	doubleVektorEingabe
Parameter	**Vektor; Bildschirmtext
Rückgabewerte	Dimension; -1 ... Fehler

Schnittstelle:

Eingabe einer reellen Zahl: **io.c io.h**

Name	doubleEingabe
Parameter	*Zahl; Bildschirmtext
Rückgabewerte	0 ... korrekte Eingabe; 1 ... EOF (^D)

Ausgabe eines Vektors: **vec.c vec.h**

Name	doubleVektorAusgabe
Parameter	*Vektor
Rückgabewerte	kein

Löschen eines Vektors: **vec.c vec.h**

Name	vektorLoeschen
Parameter	**Vektor
Rückgabewerte	Anzahl gelöschter Komponenten

Skalarprodukt zweier Vektoren: **vec.c vec.h**

Name	skalarProdukt
Parameter	*Vektor; *Vektor; *Ergebnis
Rückgabewerte	1 ... korrekt; 0 ... Fehler

Testprogramme und Testbeispiele:

testein.c	doubleEingabe	`3.5; 3e5; -3.4e-5; .0; 3er; ENTER; STRG + d`

testvec.c	doubleVektorEingabe
	doubleVektorAusgabe
	vektorLoeschen
		bel. Vektoren; Vektor der Länge 0:
		`V1=(1, 2, 3); V2=(1, 2);`
		`V3=(1, 1, 1); V=()`

testscal.c	skalarProdukt	Vektoren gleicher Länge zwei Vektoren verschiedener Länge ein oder zwei Vektoren der Länge 0

Module:

Hauptprogramm:	**scalar.c**
Makefile:	**makefile**
Headerfile:	**vec.h io.h**
Implementation:	**vec.c io.c**
Testprogramme:	**testein.c, testvec.c, testscal.c**

Aufteilung:

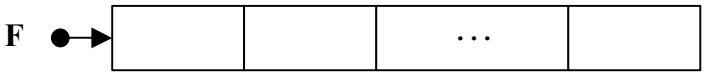
Programmierer 1:	**makefile, vec.h, vec.c, testvec.c, testscal.c**
Programmierer 2:	**makefile, io.h, io.c, testein.c, scalar.c**

Strukturierte Datentypen - Dynamische Feldvereinbarung

`<stdlib.h>: void *malloc( size_t); void free( void *);`

`malloc()` verlangt die Größe des zu reservierenden Speichers als Byteanzahl
`sizeof()` mit dem Datentyp als Operand ermittelt diese

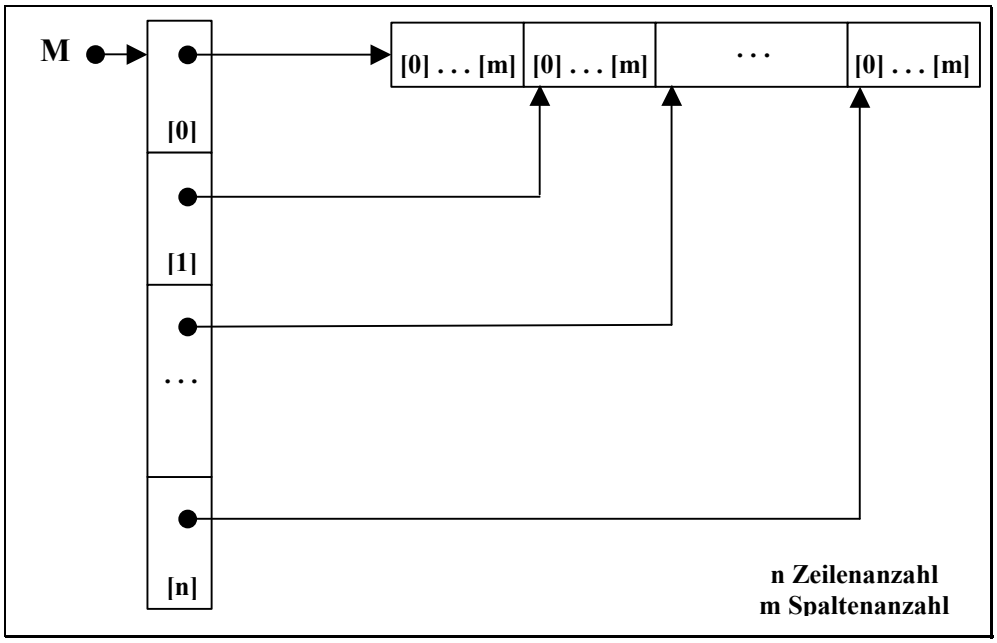
Vereinbarung von Vektoren



F: Zeiger auf ein Feld von Feldkomponenten
Feldkomponente: $f_i \Rightarrow \mathrm{F}[i]$

Standardoperationen **(`ITEM` . . . Datentyp der Komponenten)**
Feld erzeugen: **`ITEM *initFeld( unsigned int);`**
Feld freigeben: **`void freeFeld( ITEM *);`**

Vereinbarung von Matrizen



M: Zeiger auf ein Feld von Zeigern auf die Matrixkomponenten
Matrixkomponente: $m_{ij} \Rightarrow \mathrm{M}[i][j]$

Standardoperationen
Matrix erzeugen:
`ITEM **initMatrix( unsigned int, unsigned int);`
Matrix freigeben: **`void freeMatrix( ITEM **);`**

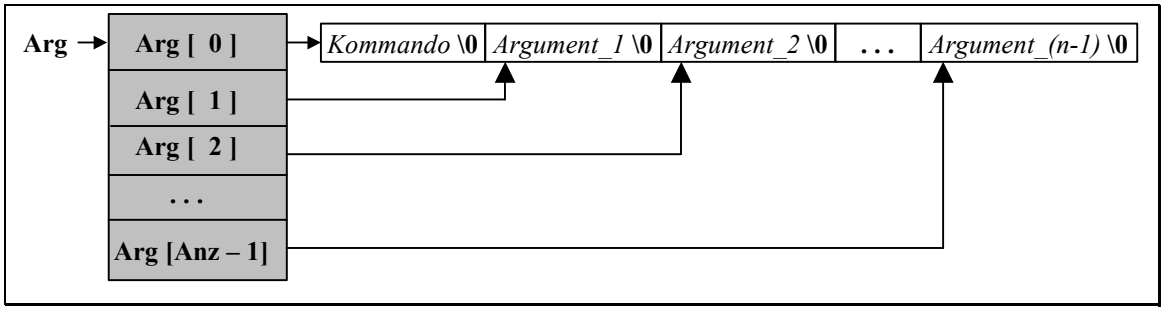
Beispiel für Matrizen - Kommandozeilenparameter

Kommando Argument_1 Argument_2 . . . Argument_(n-1)

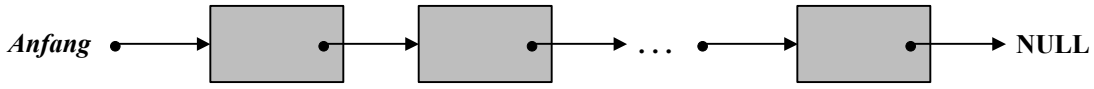
int main ( ) { . . . }
int main (int *Variable*, **char * * Zeiger) { . . . }**

Variable ... Anzahl der Wörter der Kommandozeile
Zeiger ... zeigt auf Anfang des Kommandos

int main (int Anz, char * * Arg) { . . . }

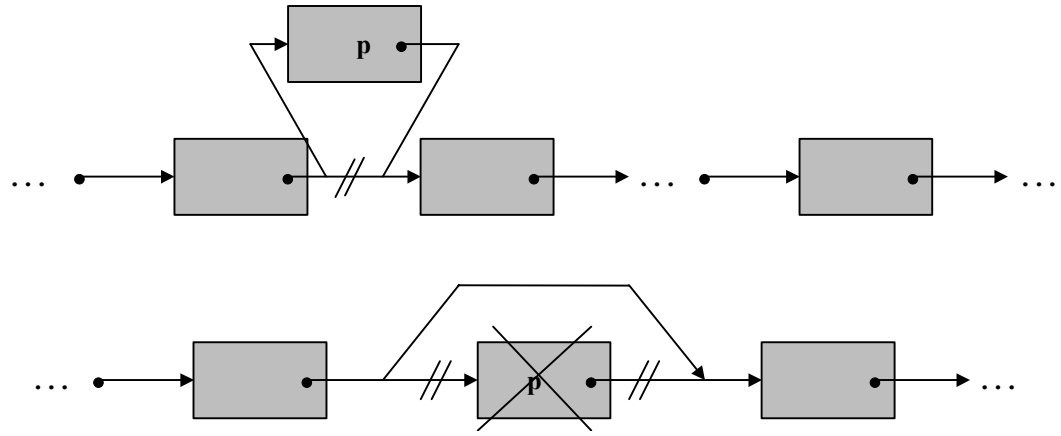


Einfach verkettete lineare Listen



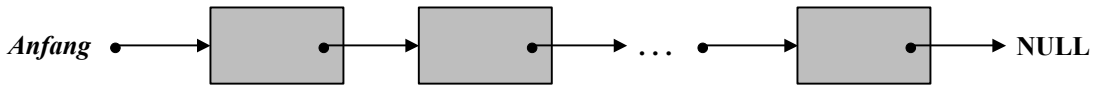
Anfang: Zeiger auf das erste Listenelement
Listenelement: Struktur, bestehend aus Daten und mindestens einem Zeiger
NULL: Nullzeiger, Zeiger auf „Nichts“, Listenende

Standardoperationen
Element suchen
Element einfügen
Element entfernen



Spezielle verkettete lineare Listen

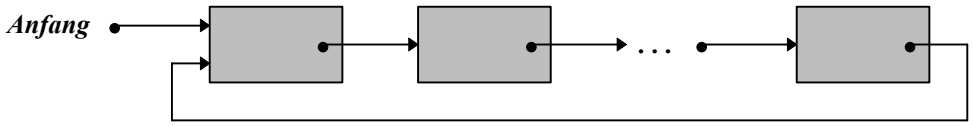
<u>Einfach verkettete lineare Listen</u>



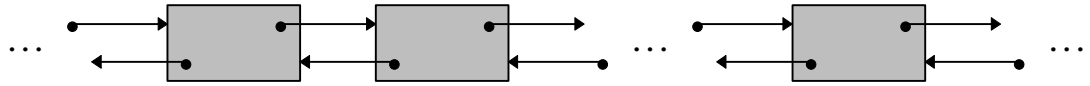
Keller (Stack, lifo-Prinzip, last in first out)

Warteschlange (Queue, fifo-Prinzip, first in first out)

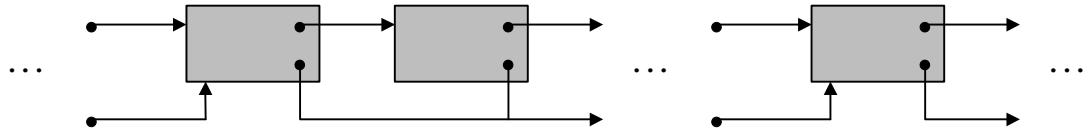
<u>Zyklisch verkettete lineare Listen</u> (Listenende zeigt auf den Anfang):



<u>Doppelt verkettete lineare Listen</u> (Verweis auf Vorgänger und Nachfolger):



<u>Mehrfach verkettete lineare Listen</u> (Verweis auf die nächste Gruppe von Elementen):



<u>Allgemeine Listen</u> erhält man, wenn man als Datentyp der Elemente wiederum Listen zulässt.