12. Konzepte der Computerprogrammierung

- Algorithmen und Beschreibungsformen
- Strukturierung von Programmabläufen
- Typen
- Abstraktionsmechanismen
- Anforderungen an eine maschinennahe Programmiersprache

12.1 Algorithmen und Ablaufstrukturierung

12.1.1 Intuitive Einführung

- Algorithmen sind detaillierte Vorschriften zur schrittweisen Lösung von Problemen.

- Algorithmen sind allgemeine Lösungsverfahren und nicht an eine bestimmte Ausführungsform gebunden (z.B. eine bestimmte Programmiersprache).

- Algorithmen bestehen aus
  - einer Abfolge von Arbeitsschritten,
  - Ein- und Ausgaben von Daten,
  - bedingten Verzweigungen,
  - Schleifen,
  - Unteralgorithmen.

- Jeder Schritt eines Algorithmus basiert auf elementaren Handlungen.
  Elementare Handlungen zeichnen sich dadurch aus, das sie wohldefiniert (zweifelsfrei) sind und keiner weiteren Erläuterung bedürfen.

Beispiel: Textuelle Beschreibung eines Algorithmus

Beispiel: Finden des "größten gemeinsamen Teilers" (GGT) zweier Zahlen

Gegeben seien zwei Zahlen x und y.

Wenn x größer y ist, **dann** subtrahiere von x den Wert y,
wenn nicht, dann subtrahiere von y den Wert x.

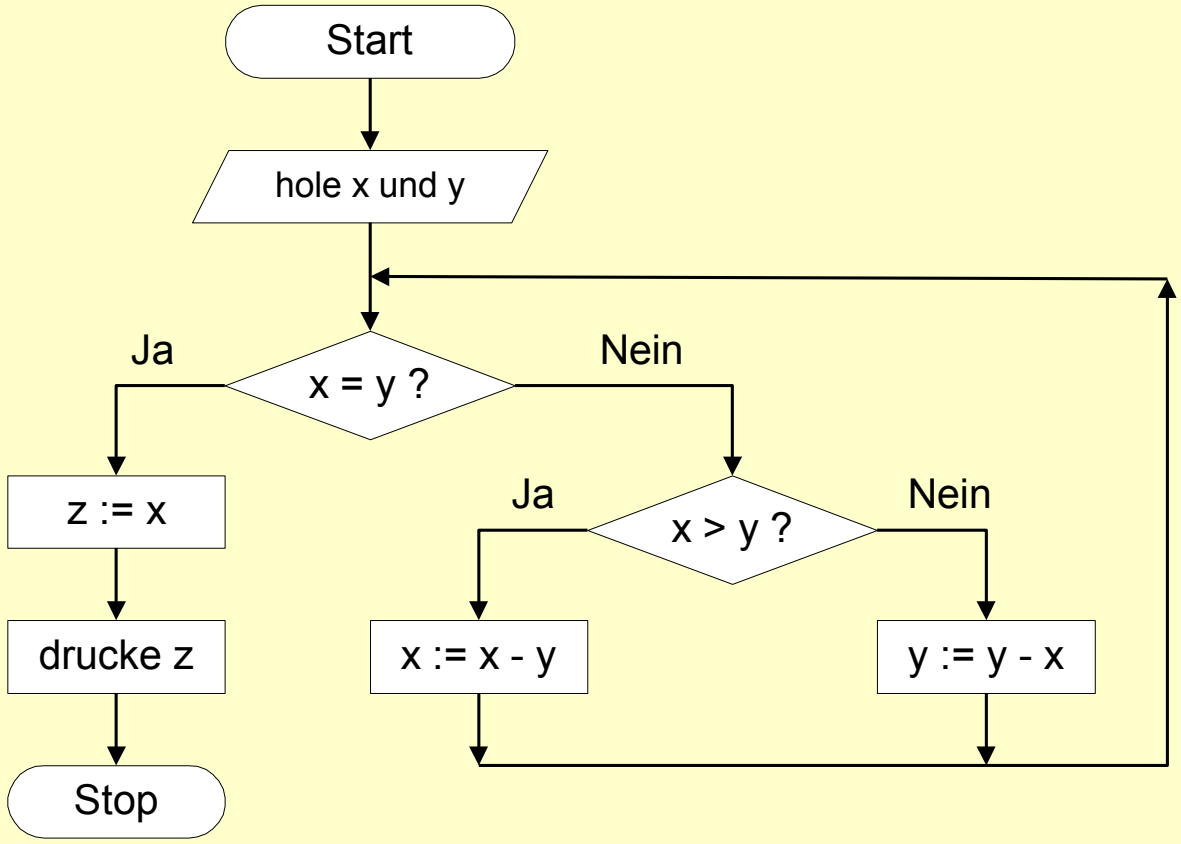
Wenn x und y nicht gleich sind, **dann wiederhole** den letzen Schritt, **ansonsten** drucke das Ergebnis aus.

z.B.: x = 27, y = 15

```
27 - 15 = 12    // x:=x-y (=12)
15 - 12 = 3     // y:=y-x (= 3)
12 -  3 = 9     // x:=x-y (= 9)
 9 -  3 = 6     // x:=x-y (= 6)
 6 -  3 = 3     =>   GGT = 3
```

12.1.2 Darstellungsform: Flussdiagramm

Beispiel: Finden des “größten gemeinsamen Teilers” (GGT) zweier Zahlen



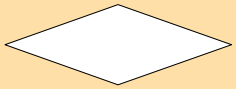
- älteste aller Darstellungsmethoden (für 1. Sprach-Generation: Assembler)
- basiert auf nur 5 Symbolen
- **Assembler-geeignet (Darstellung unstrukturierter Algorithmen)**
- genormt (DIN66001 u. ISO-Norm 5807)

Symbole des Flussdiagramms

Programmstart oder -ende

Aktion

Ein- und Ausgabe



Bedingung

Fortsetzung an anderer Stelle

Flussdiagramm – Vor- und Nachteile

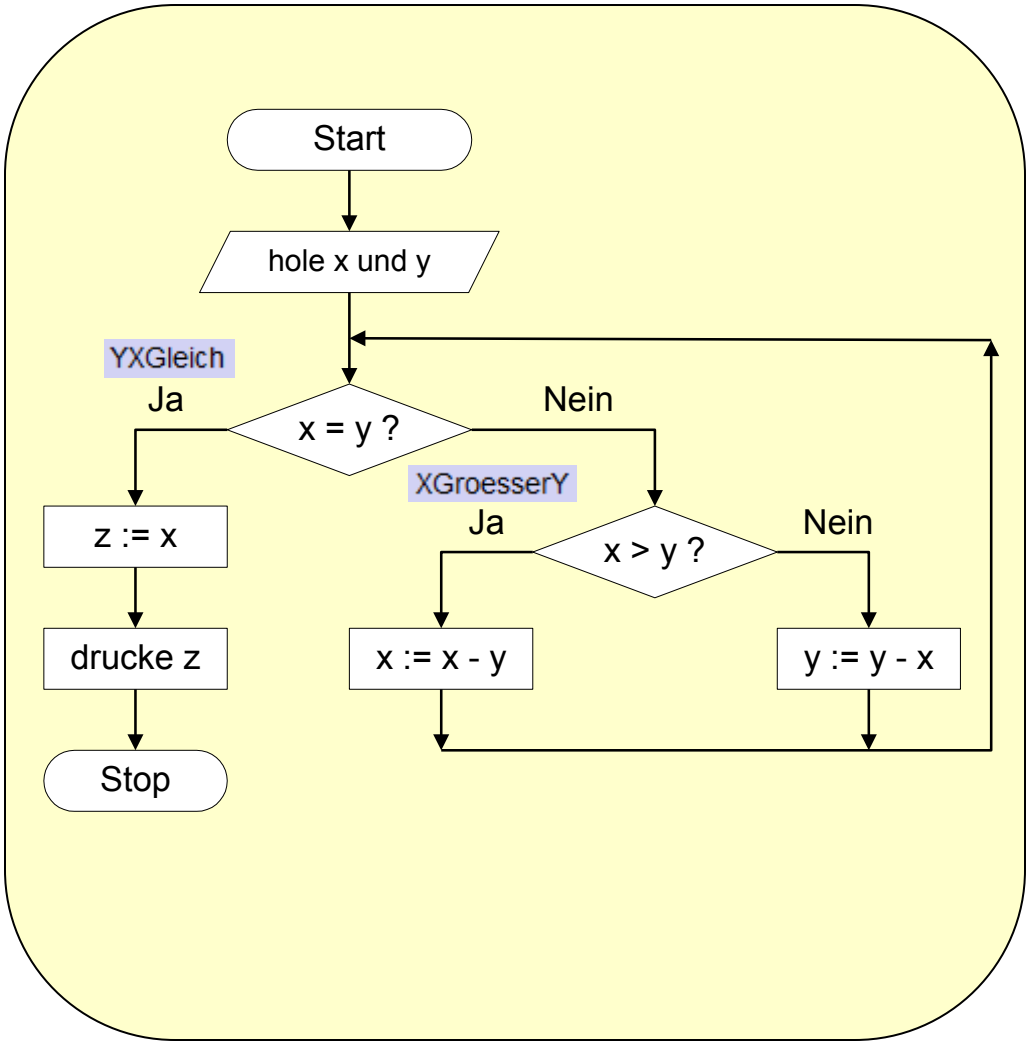
Vorteile:
- sehr intuitiv und einfach erlernbar
- genormte Symbole
- kann leicht ergänzt werden

Nachteile:
- verleitet zu unstrukturiertem Denken (“Spaghetti-*Programmierung*”) → s. Beispiel

Bewertung:
- Die Urform des Flußdiagramms ist “*veraltet*” aber
- eine stark erweiterte Variante (*Aktivitätsdiagramm*) ist Teil der UML 2.

Anmerkung:
- Das Flußdiagramm ist eine häufig verwendete Beschreibungsform für Assemblerprogramme (z.B. Dokumentation von Altcode).
  Neue Assemblerprojekte sollten immer strukturiert geplant werden (s.u.).

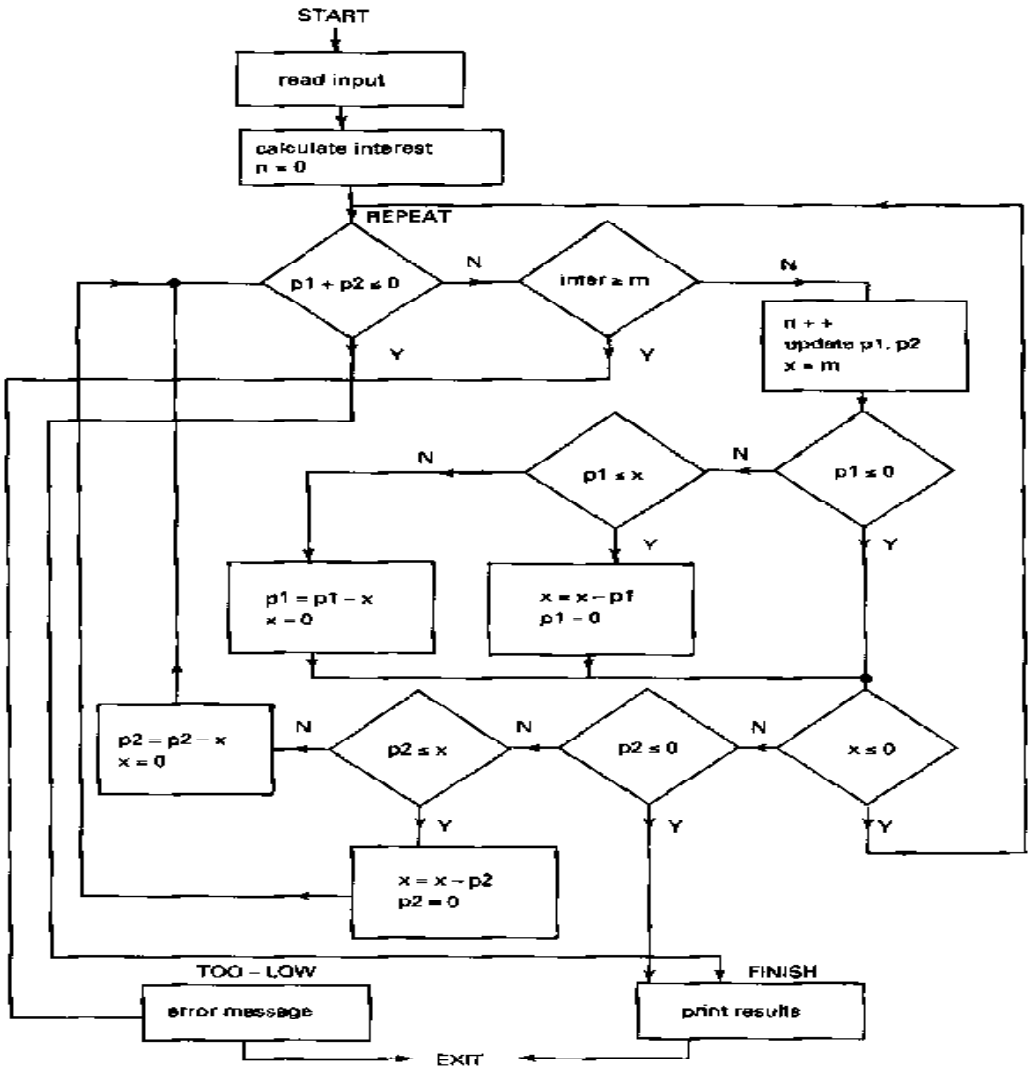
Beispiel: Flussdiagramm und Realisierung mit unstrukturierter Sprache



Beispiel einer unstrukturierten Sprache

```
PROGRAM GGT(x,y)
LABEL  naechsterLauf
IF x=y THEN GOTO  YXGleich
IF x>y THEN GOTO  XGroesserY
y=y-x
GOTO  naechsterLauf
LABEL  XGroesserY
x=x-y
GOTO  naechsterLauf
LABEL  XYGleich
z=x
PRINT(z)
END PROGRAM
```

Beispiel: Spaghetti-Programm (GoTo-Programmierung)



Selbst einfache Zusammenhänge werden sehr schnell unübersichtlich !

12.1.3 Strukturierte Programmierung

Idee entstand Ende der 60'er Jahre als Folge der sog. "*1. Softwarekrise*"*:*
Problem: → *"unentwirrbare Programmabläufe" (Spaghetticode)*

Untersuchungen von *Nassi-Shneiderman* haben gezeigt:

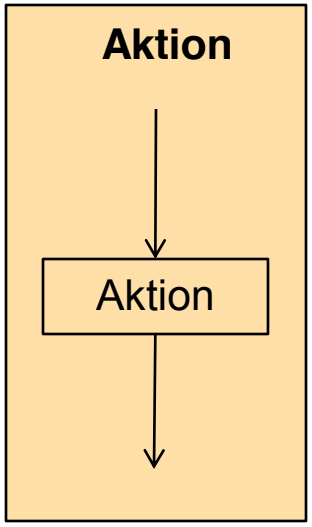
Anwendungen sind (goto-frei) durch nur 7 Strukturblockarten darstellbar !

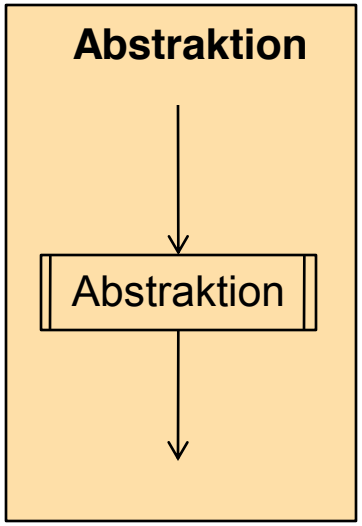
Strukturblöcke sind aus Kombinationen von Bedingungen und Aktionen darstellbar !

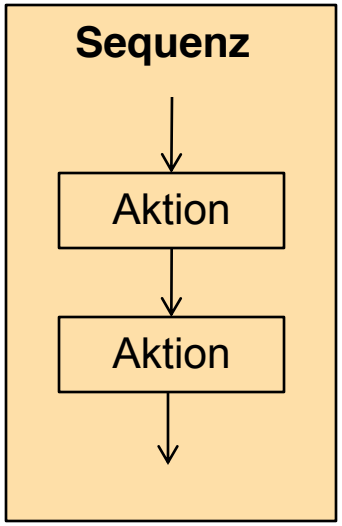
⇒ **Strukturierte Programmierung**

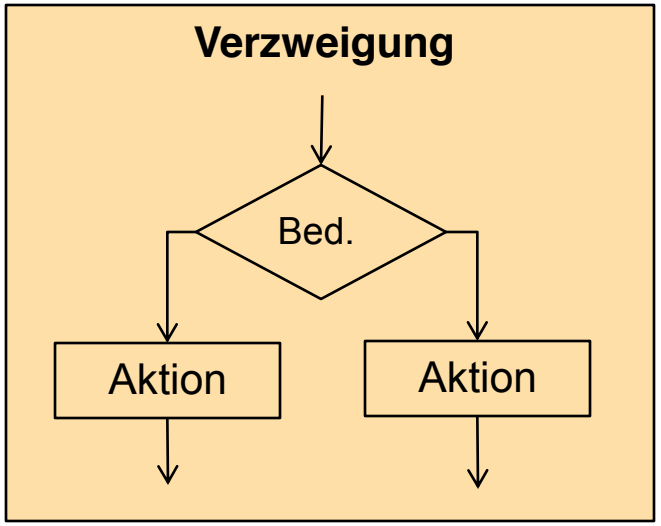
Fazit: Unstrukturierte Entwürfe können immer in strukturierte Entwürfe umgeformt werden !

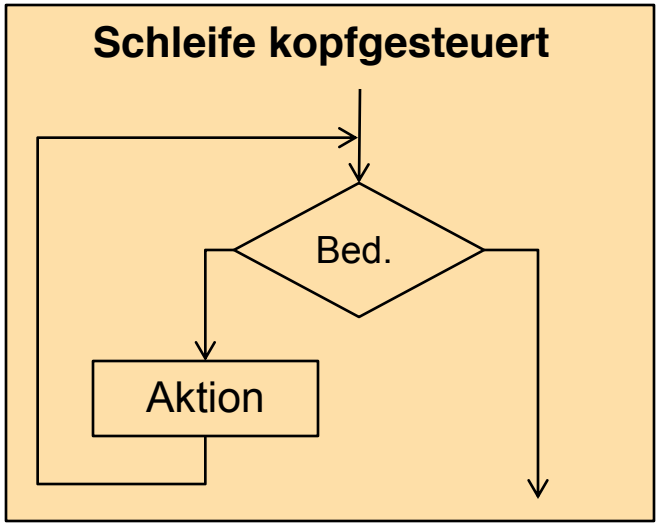
Strukturblockarten nach Nassi-Shneiderman

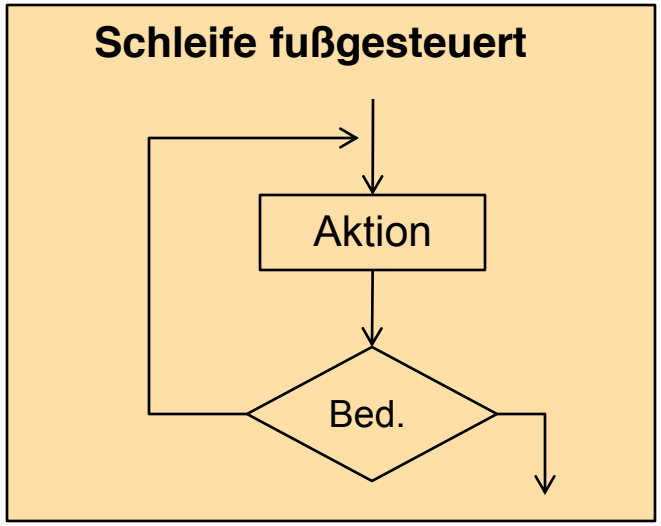










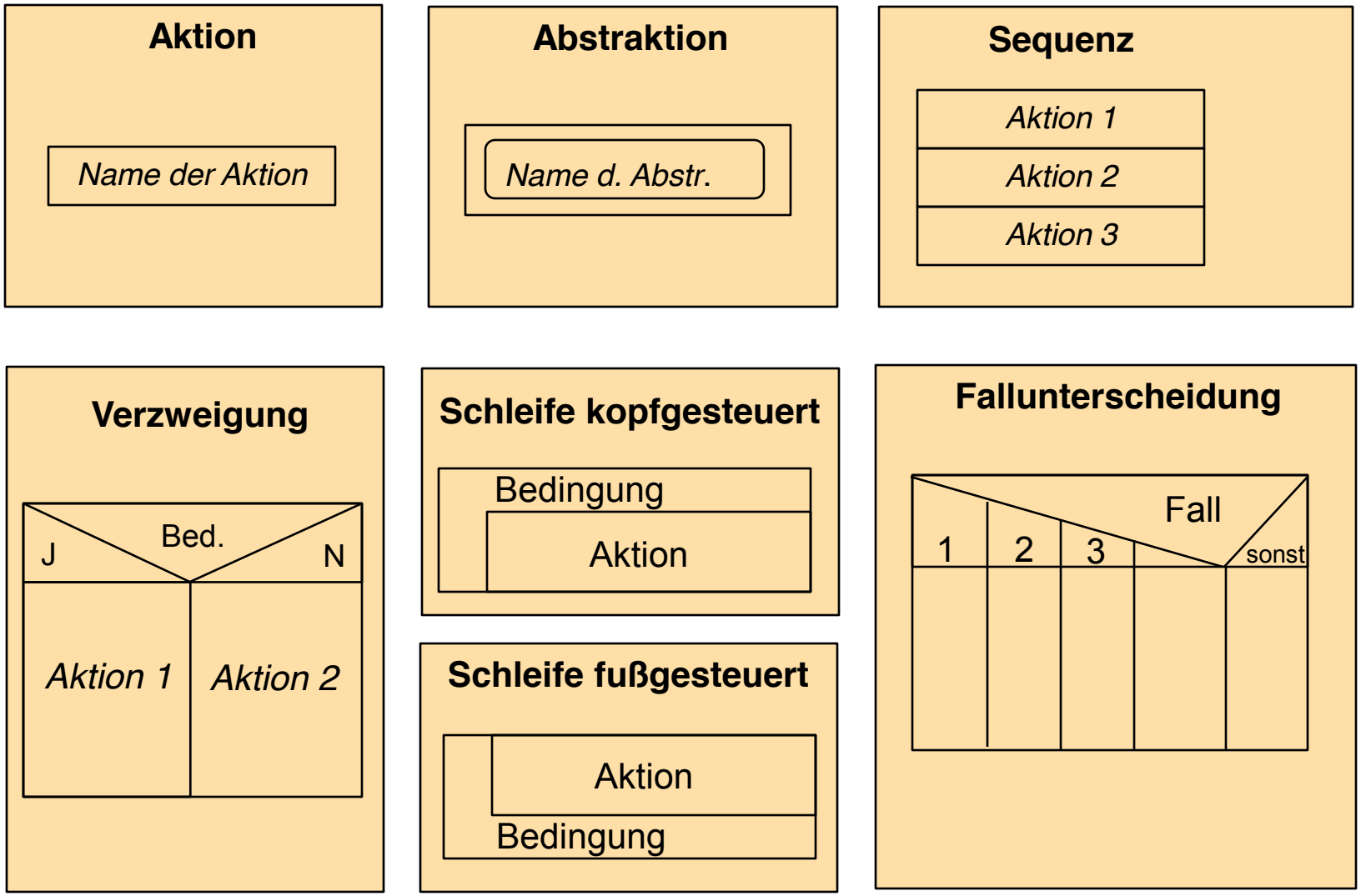


+

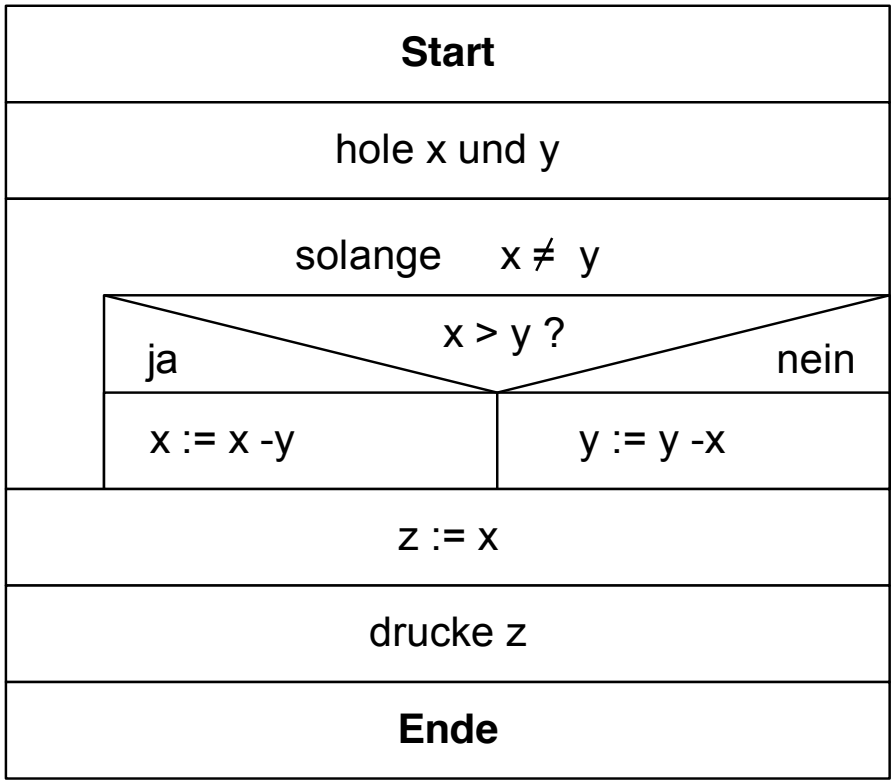
mehrfache Verzweigung

(= Fallunter-scheidung)

Struktogramm nach Nassi-Shneiderman



Beispiel: GGT-Algorithmus als Struktogramm



Strukturblöcke können nur

gestapelt oder
geschachtelt

werden !

Unstrukturierte Algorithmen-entwürfe müssen entsprechend umgebaut werden !

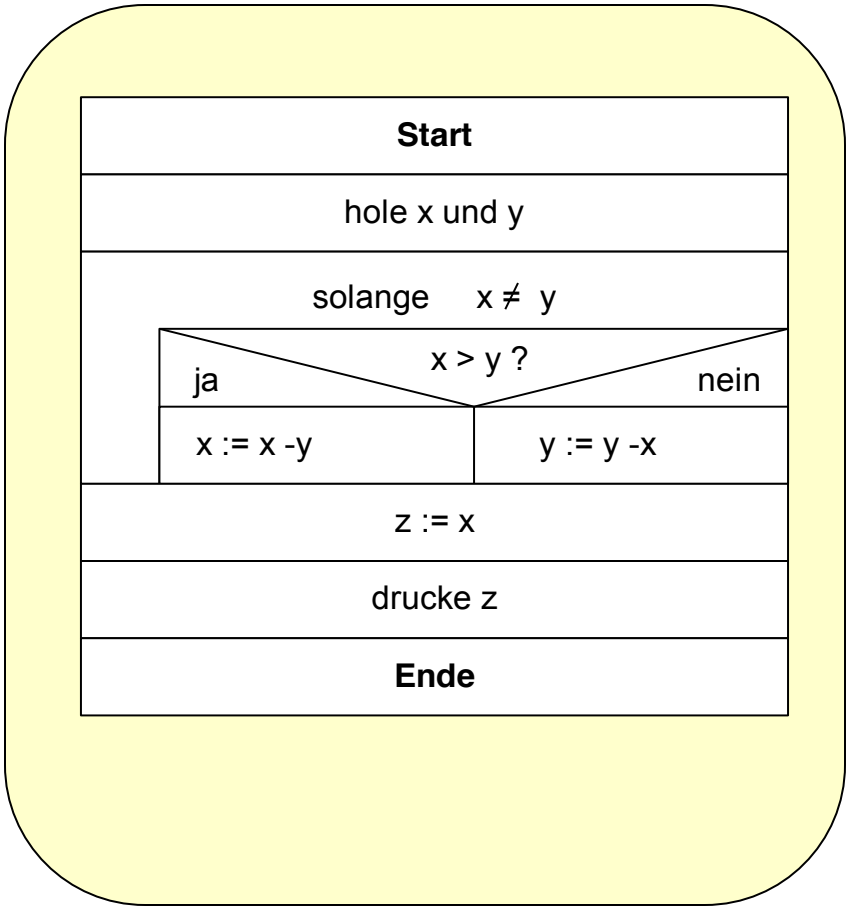
Struktogramm – Vor- und Nachteile

Vorteile:
- grundsätzlich strukturiert (goto-frei)
  → führt zu wesentlich übersichtlicheren und verständlichere Programmen
- Algorithmenentwurf hierarchisch gliederbar (Top-Down- bzw. Bottom-up-Entwurf)
- genormt (DIN 66261)

Nachteile:
- mit Papier und Bleistift ist der Entwurf ziemlich umständlich
- nur mit geeigneten Werkzeugen einsetzbar

Bewertung:
- Erfüllt mit geeigneten Werkzeugen durchaus seinen Zweck,
- ist "***veraltet***" und wird kaum noch verwendet,
- Grund: Der im Folgenden beschriebene *Pseudocode* (s.u.) besitzt die gleichen Vorteile wie Struktogramme und vermeidet dessen Nachteile

Beispiel: Struktogramm und Realisierung mit strukturierter Sprache



Beispiel einer strukturierten Sprache

```
PROGRAM GGT(x,y)
WHILE  ( x ungleich y )
        IF (x > y)
                x=x-y
        ELSE
                y=y-x
        ENDIF
ENDWHILE
z=x
PRINT(z)
END PROGRAM
```

Vergleich: Unstrukt. Programmiersprache vs. strukt. Programmiersprache

```
PROGRAM GGT(x,y)
LABEL naechsterLauf
IF x=y THEN GOTO YXGleich
IF x>y THEN GOTO XGroesserY
y=y-x
GOTO  naechsterLauf
LABEL  XGroesserY
x=x-y
GOTO  naechsterLauf
LABEL  XYGleich
z=x
PRINT(z)
END PROGRAM
```

```
PROGRAM GGT(x,y)
WHILE  ( x ungleich y )
        IF (x > y)
                x=x-y
        ELSE
                y=y-x
        ENDIF
ENDWHILE
z=x
PRINT(z)
END PROGRAM
```

12.1.4 Darstellungsform: Pseudocode

Die Algorithmusbeschreibung ist Programmiersprachen-ähnlich.

Die Strukturierungselemente sind angelehnt an die des Struktogramms.
Es gibt einige praktische Erweiterungen.

Strukturierungselemente des Pseudocode *(hier: LaTEX-Style: algoritmicx)*

a) Zuweisung

$$x \leftarrow x + 1$$

b) Kopfgesteuerte Schleife

```
1: while <text> do
2:     <body>
3: end while
```

Beispiel

```
1: sum ← 0
2: i ← 1
3: while i ≤ n do
4:     sum ← sum + i
5:     i ← i + 1
6: end while
```

→ while-Bedingung ist eine Laufbedingung (in der Schleife bleiben, **solange** …. gilt)

c) Kopfgesteuerte Zählschleife (Erweiterung)

```
1: for <text> do
2:     <body>
3: end for
```

Beispiel

```
1: sum ← 0
2: for i ← 1, n do
3:     sum ← sum + i
4: end for
```

Fussgesteuerte Schleife

```
1: repeat
2:     <body>
3: until <text>
```

Beispiel

```
1: sum ← 0
2: i ← 1
3: repeat
4:     sum ← sum + i
5:     i ← i + 1
6: until i > n
```

→ until-Bedingung ist eine Abbruchbedingung (bleibe in der Schleife **bis** …… gilt)

Iteration über eine Elementemenge (Erweiterung)

```
1: for all <text> do
2:     <body>
3: end for
```

Alternative

```
1: if <text> then
2:     <body>
   [
3: else if <text> then
4:     <body>
   ...
   ]
   [
5: else
6:     <body>
   ]
7: end if
```

Beispiel

```
1: if quality ≥ 9 then
2:     a ← perfect
3: else if quality ≥ 7 then
4:     a ← good
5: else if quality ≥ 5 then
6:     a ← medium
7: else if quality ≥ 3 then
8:     a ← bad
9: else
10:    a ← unusable
11: end if
```

Abstraktion durch Prozeduren

```
1: procedure <NAME>(<params>)
2:     <body>
3: end procedure
```

Beispiel

```
1: procedure GGT(x, y)
2:     while x ≠ y do
3:         if x > y then
4:             x ← x − y
5:         else
6:             y ← y − x
7:         end if
8:     end while
9:     return x
10: end procedure
```

Pseudocode – Vor- und Nachteile

Vorteile:

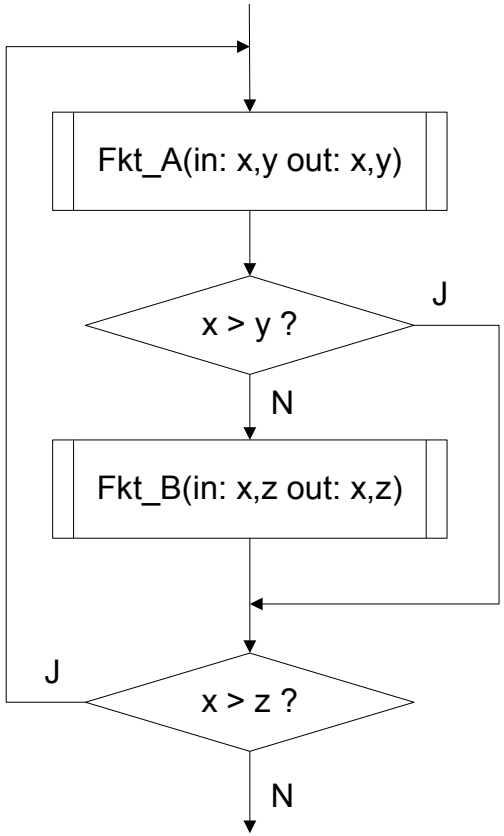
- grundsätzlich strukturiert (goto-frei)
  → führt zu übersichtlicheren und verständlichere Programmen
- Algorithmenentwurf hierarchisch gliederbar (Top-Down- bzw. Bottom-up-Entwurf)
- sprachunabhängig aber trotzdem implementierungsnah
- leicht als Kommentartext in Programme integrierbar
- leicht zu erstellen und zu ändern

Nachteile:

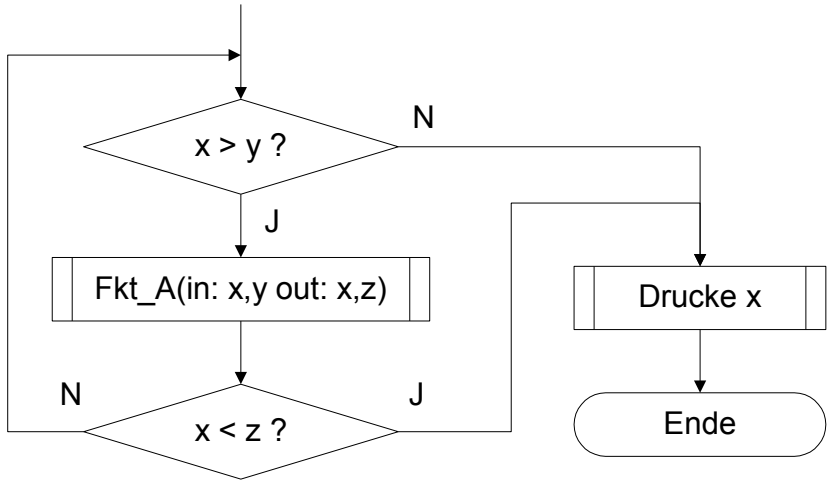
- Gefahr der zu unpräzisen Algorithmusbeschreibung

ÜBUNG: Umwandeln Flussdiagramm -> Pseudocode

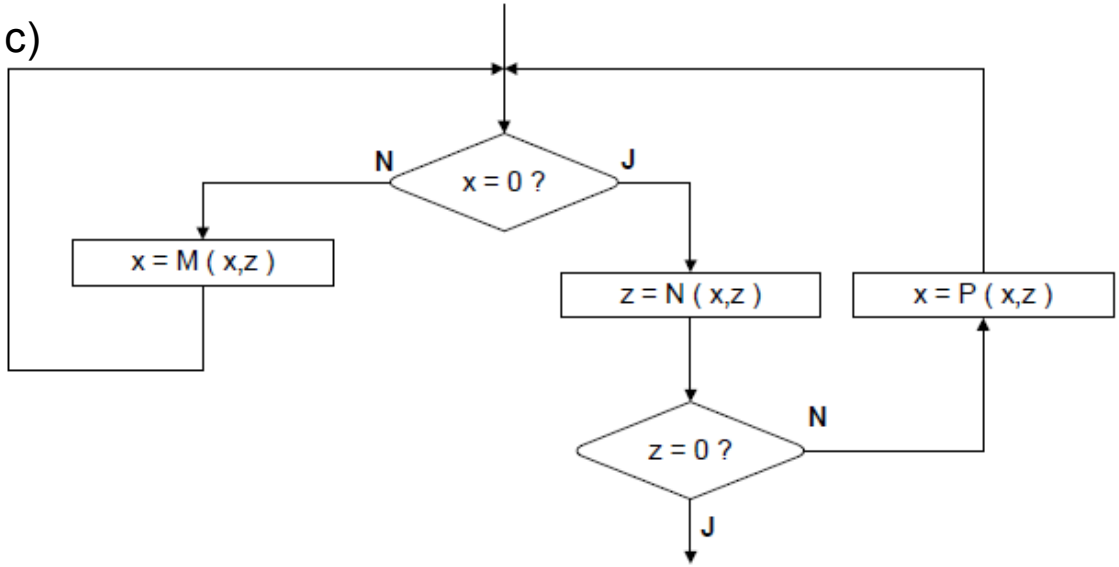
a)



b)



c)



12.1.5 Konsequenzen für Programmiersprachen

- Die meisten **Assemblersprachen** stellen für die Ablaufgestaltung lediglich bedingte Sprünge zur Verfügung (*Branch if zero, Branch if not carry, u.ä.*).
  Strukturierung wird daher nicht erzwungen sondern kann **nur durch Programmier-disziplin** (per Konvention) erreicht werden.
  Fazit: Assembler besitzt i.a. keine Sprachkonstrukte zur Ablaufstrukturierung

- Alle **höheren** prozeduralen und viele objektorientiere **Programmiersprachen** stellen Sprachkonstrukte für die Ablaufstrukturierung im Sinne der Nassi-Shneiderman-Strukturblockarten zur Verfügung, z.B. :

  - **if** (*Bedingung*) **then** {*Programmsequenz*} **else** {*Alternativsequenz*} **endif**
  - **while** (*Bedingung*) **do** {*Programmsequenz*} **endwhile**
  - **repeat** {*Programmsequenz*} **until** (*Bedingung*) **endrepeat**

12.2 Typen

- Typen klassifizieren Daten aufgrund ihrer Eigenschaften und ihrer Verwendungsmöglichkeiten.

- Typen sollen die unbeabsichtigte oder falsche Interpretation oder Verwendung von Daten verhindern.

- Eine Typüberprüfung stellt die Kompatibilität zwischen Operator und seinen Operanden sicher.

- Programmiersprachen in denen es Typen gibt, heißen „*typisiert*“.

- Eine Programmiersprache heißt „*statisch typisiert*“, wenn die Typüberprüfung zur Compilezeit überprüft wird (z.B. C).

- Eine Programmiersprache heißt „*stark typisiert*“, wenn der Compiler die Typkonsistenz garantiert (*strong typing*).

  Beispiel : Assembler → nicht typisiert

12.3 Abstraktionsmechanismen

12.3.1 Prozedurale Programmiersprachen → ablauforientiert

Abstraktion hilft Komplexität zu beherrschen.

Ein wichtiges Abstraktionsmittel sind *Prozeduren*. Eine Prozedur ist eine Softwareeinheit mit einer Ein- und Ausgabeschnittstelle sowie einem beschreibbaren Verhalten. Das Wesen der Prozedur ist, das das der Anwender nur das Schnittstellenverhalten kennen muß. Die Implementierungsdetails sind für die Verwendung unwesentlich.

Programmiersprachen, die hauptsächlich durch die Untergliederung in verschiedene Prozeduren strukturiert sind, heißen *prozedurale Programmiersprachen*.

In prozeduralen Programmiersprachen ist die Verarbeitung strukturiert, aber nicht zwangsläufig die Daten !

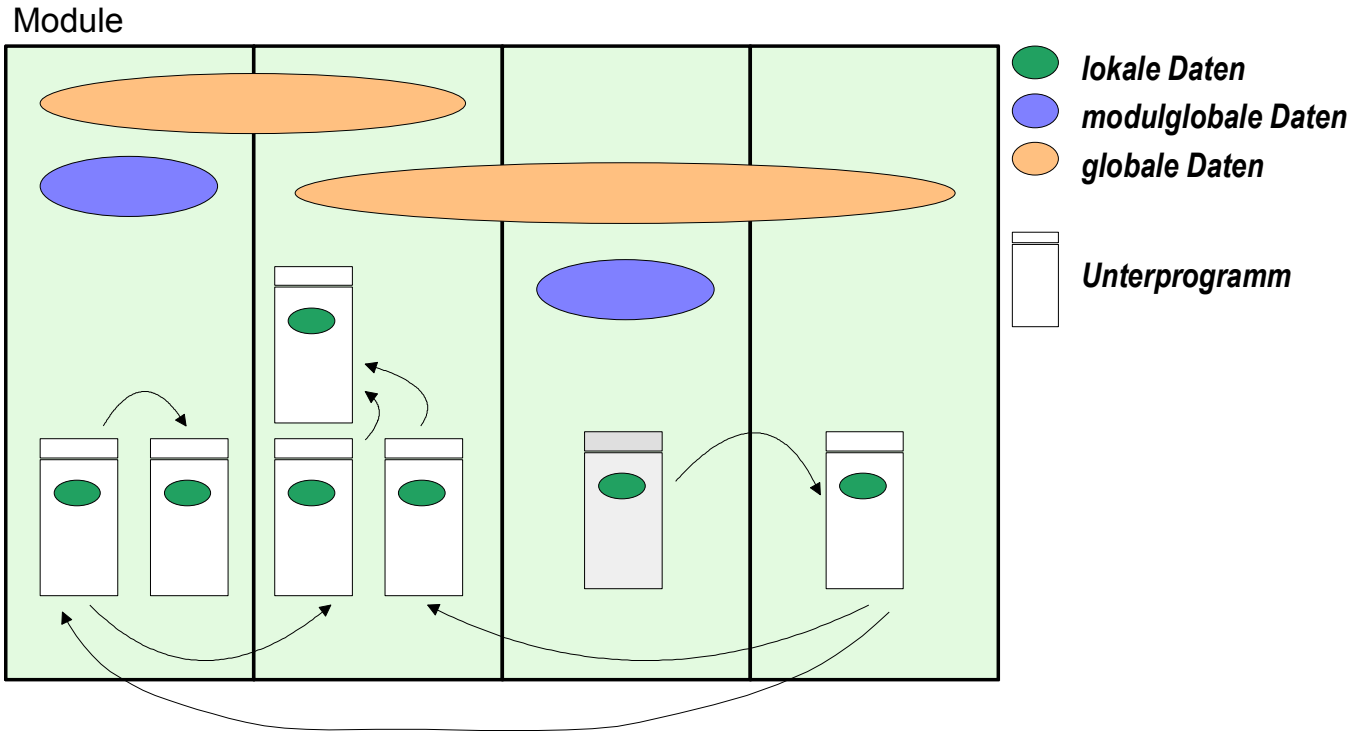
In statisch typisierten Sprachen wird die Typkorrektheit der Ein- und Ausgabewerte der Prozeduren zur Complilezeit überprüft.

Bsp.: Assemblersprachen stellen i.allg. Mechanismen für die Bildung von Prozeduren zur Verfügung (bl, bx).
Eine Typüberprüfung der Ein- und Ausgabewerte findet aber nicht statt.

Grenzen prozeduraler Programmiersprachen

In prozeduralen Programmen gibt es keine feste Kopplung zwischen den Prozeduren und Daten.

Änderungen/Erweiterungen an den Datenstrukturen oder Prozeduren führen daher häufig zu schwer überblickbaren Änderungen am gesamten Softwaresystem.

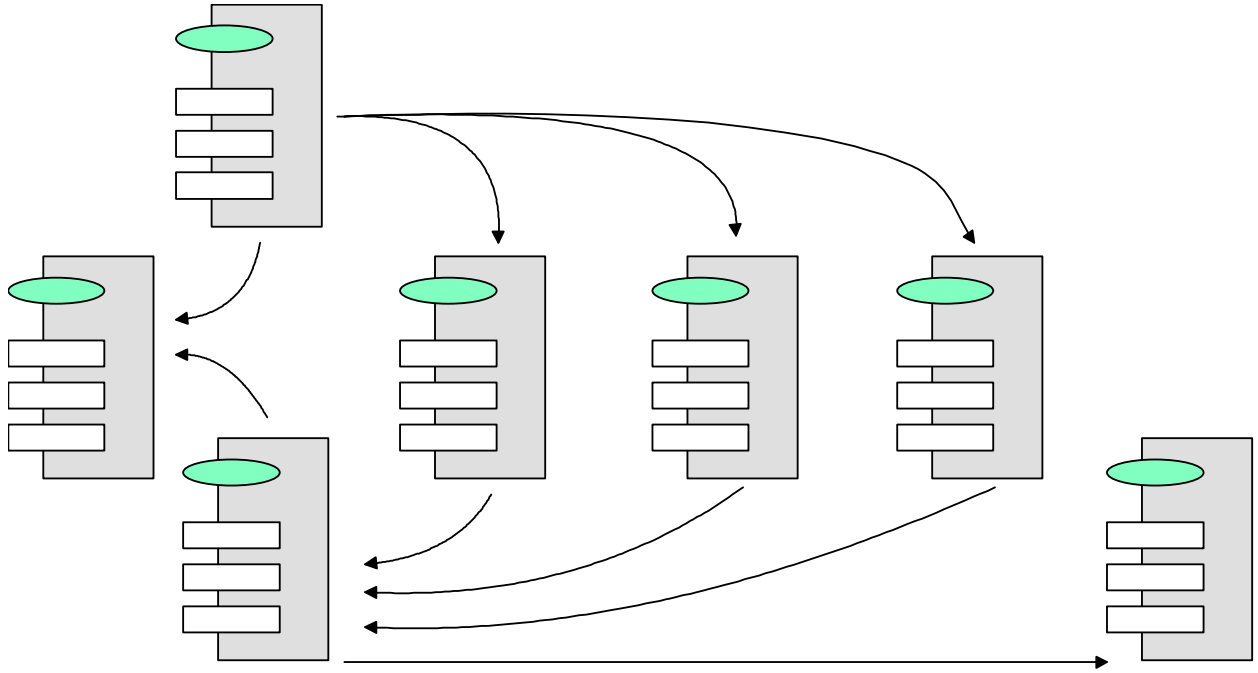


In Zeiten zunehmender Softwarekomplexität (Anfang der 90er Jahre) war dieser Sachverhalt eine zunehmendes Implementierungshindernis.

→„*Wiederverwendungskrise*“ der Softwareentwicklung

12.3.2 Objektorientierte Programmiersprachen → interagierende Objekte

Objektorientierte Programmiersprachen vermeiden das geschilderte Problem u.a. durch feste Kopplung der Daten und der mit ihnen arbeitenden Prozeduren.



Ziele objektorientierter Programmiersprachen sind also eine verbesserte Änderbarkeit, Wartbarkeit und Wiederverwendbarkeit von Softwareteilen.

12.4 Mindestanforderungen an höhere, maschinennahe Sprachen

effizient

Gutes Laufzeitverhalten, wenig Overhead.

typisiert

Operationen und Unterprogrammaufrufe werden zur Compilezeit daraufhin überprüft, ob die Parameter den vereinbarten Typ haben.

strukturiert

Es stehen Sprachkonstrukte im Sinne der Nassi-Shneiderman-Strukturblockarten zur Verfügung.

prozedural

Das Programm kann in benennbare Funktionsblöcke zerlegt werden. Von einer einmal geschriebenen Funktion muss nur noch das Schnittstellenverhalten bekannt sein.

modular

Der Quellcode kann in sinnvolle Einheiten unterteilt werden, z.B. mit dem Ziel der Wiederverwertbarkeit von Codeteilen. → Bibliotheken
Beispiele: Mathematische Funktionen, Grafikfunktionen, …..

13. Programmiersprache C

13.1 Historie

- C wurde 1972 von Dennis Ritchie bei den AT&T Bell Lab als Systemprogrammier-sprache zur Implementierung von UNIX für die PDP-11 entwickelt

- Ziel von C war die Entwicklung einer Hochsprache für lesbare und portable Systemprogramme, aber einfach genug, um auf die zugrunde liegende Maschine abgebildet zu werden.

- In 1973/74 wurde C von Brian Kernighan verbessert. Daraufhin wurden viele Unix-Implementierungen von Assembler nach C umgeschrieben.

- Um die Vielzahl der entwickelten Compiler auf einen definierten Sprachumfang festzulegen, wurde C 1983 durch die amerikanische Normbehörde ANSI normiert. Diese Sprachnorm wird als ANSI-C bezeichnet. Die aktuelle Norm stammt aus dem Jahr 1999 (ANSI = **A**merican **N**ational **S**tandardisation **I**nstitute).

13.2 Einführende Anmerkungen

13.2.1 Stärken und Schwächen

Stärken:

- standardisiert (ANSI)
- sehr effizient (hardwarenah implementiert)
- universell verwendbar
- sehr weit verbreitet, speziell in technischen Anwendungen
- Typüberprüfung (weitgehend strong typing)

Schwächen:

- Code kann beliebig unlesbar geschrieben werden, da Fragen des Programmierstils nur in (freiwilligen) Konventionen festgelegt sind
  --> Erfahrung und persönlicher Stil haben entscheidenden Einfluß auf die Softwarequalität !
- teilweise etwas kryptische Notation

13.2.2 Die größte Gefahr: Stilloser Code

```
/* ASCII to Morsecode                                          */
/* Obfuscated C Code Contest :   „Best“ small Program */

#include<stdio.h>
#include<string.h>

main()
{
        char*O,l[999]="`acgo\177~|xp .-\0R^8)NJ6%K4O+A2M(*0ID57$3G1FBL";
        while(O=fgets(l+45,954,stdin)){
                *l=O[strlen(O)[O-1]=0,strspn(O,l+11)];
                while(*O)switch((*l&&isalnum(*O))-!*l){
                case-1:{char*l=(O+=strspn(O,l+12)+1)-2,O=34;
                        while(*l&3&&(O=(O-16<<1)+*l---'-')<80);
                        putchar(O&93?*l&8||!(l=memchr(l,O,44))?'?':l-l+47:32);
                        break;
                case 1:     ;}*l=(*O&31)[l-15+(*O>61)*32];
                        while(putchar(45+*l%2),(*l=*l+32>>1)>35);
                case 0:     putchar((++O,32));}
        putchar(10);}
}
```

13.2.3 Eigenschaften

C ist .....

klein

- 32 Schlüsselwörter und
- 40 Operatoren

modular

- alle Erweiterungen stecken in Funktionsbibliotheken
- unterstützt das Modulkonzept

maschinennah

- geht mit den gleichen Objekten um wie die Hardware:
  Zeichen,
  Zahlen,
  Adressen,
  Speicherblöcke.

13.3 Ein erster Blick auf (einfache) C-Programme

13.3.1 Beispiel eines einfachen C-Programms

```
#include <stdio.h>

int main( )
{
    printf("Hello World\n");
    return 0;
}
```

Eine **main-Funktion** wird immer benötigt, damit der Compiler den Beginn des Hauptprogramms erkennt. Das Programm steht zwischen { ... }.

Der Rückgabewert der Funktion ist "**int**„ (ganze Zahl). In diesem Programm ist der Rückgabewert der **main()** - Funktion 0:

return 0;

Dies bedeutet, dass das Programm sauber beendet wurde.

13.3.2 Komponenten eines C-Programms

Präprozessor

`#include <stdio.h>` ist kein direkter Bestandteil der Sprache C, sondern ein Befehl des sogenannten **Präprozessors**. Er führt vor dem eigentlichen Übersetzungsvorgang Textersetzungen durch und erlaubt die Steuerung des Übersetzungsvorganges. Präprozessorbefehle erkennt man am **#** in der ersten Spalte.

`#include` kopiert vor der Übersetzung den Text der Datei `<stdio.h>` vor den Programmtext.

Ergebnisausgabe

Mit `printf("Hello World\n")` wird der in " " stehende Text ausgegeben. `"\n"` ist eine sog. Escape-Sequenz und bedeutet „Zeilenumbruch".

Zeilenende ";"

Alle Kommandos werden mit einem ";" abgeschlossen.

Kommentare /* ..... */

Kommentare sind in `/* Kommentar ... */` eingeschlossen.

13.3.3 Einfache Datentypen und formatierte Terminalausgabe

```
#include <stdio.h>
int main ()  /* Beginn des Hauptprogramms */
{
      int       i = 10;
      double    db = 1.23;
      printf("Zahl=%d",i);
      printf("\n");
      printf("%10.2lf",db);
      return 0;
}
```

Variablendeklaration

Alle verwendeten Variablen müssen deklariert werden, d.h. der Datentyp und Name der Variablen werden bekannt gemacht.

`double` deklariert z.B. eine reelle Zahl. Das Dezimaltrennzeichen ist der ".".

Formatierte Ergebnisausgabe

Ergebnisse können durch Formatbeschreiber (%d, %8.3lf) im Formatbeschreiber-string formatiert ausgegeben werden.

%2d bedeutet, eine Integerzahl wird in einem Feld von 2 Zeichen ausgegeben.
%8.3lf bedeutet, eine double-Zahl wird in einem Feld von 8 Zeichen, mit 2 Nachkommastellen ausgegeben.

13.3.4 Tastatureingabe

```
int main ()
{
      int i;
      printf("Bitte geben Sie eine Zahl ein : ");
      scanf("%d",&i);       /*Wartet auf Eingabe*/
      printf("Die Zahl die Sie eingegeben haben war %d\n",i);
      return 0;
}
```

Mit scanf() können Texte von der Tastatur eingelesen werden.
Der Formatbeschreiber (hier "%d") gibt den Typ der eingelesenen Werte an.

Die Variable muß den Adressoperator & vorangestellt haben.

13.4 Sprachelemente von C

13.4.1 Übersicht

C besitzt 6 **Wortklassen**:

- Bezeichner
- reservierte Worte
- Konstanten
  - Ganzzahlkonstanten
  - Gleitkommakonstanten
  - Zeichenkonstanten
- Strings
- Operatoren
- Trenner (Leerstellen, Zeilentrenner, Kommentare)

*13.4.2 **Bezeichner** (vom Programmierer vergebbare Namen)*

Bezeichner (engl.: Identifier) benennen auf eindeutige Weise

- Variablennamen,
- Konstantenbezeichner,
- Typbezeichner und
- Funktionsnamen.

Syntax:

identifier : *buchstabe* { *buchstabe* | *ziffer* } .

Unter Buchstaben wird die Menge der grossen und kleinen Buchstaben des lateinischen Alphabetes verstanden; zuzüglich des Zeichen '_' (*underscore*)

- C unterscheidet zwischen Gross- und Kleinschreibung (***case-sensitiv***).
- Die Namen können beliebig lang sein, wobei mindestens die ersten 31 Zeichen signifikant sind.
- Schlüsselwörter dürfen nicht verwendet werden (s.u.)

13.4.3 Reservierte Worte

Folgende Bezeichner gehören in C zur Menge der **reservierten Wörter:**

auto	default	float	long	sizeof	union
break	do	for	register	static	unsigned
case	double	goto	return	struct	void
char	else	if	short	switch	volatile
const	enum	int	signed	typedef	while
continue	extern				

Darüberhinaus kann es Compiler-spezifische reservierte Wörter geben.

ÜBUNG Bezeichner

Welche der folgenden Bezeichner sind nicht erlaubt:

- `Mitgliedsnummer`
- `4U`
- `Auto`
- `Read_Me`
- `double`
- `Laurel&Hardy`
- `Ergänzung`
- `a`
- `ist_Null_wenn_die_Linie_laenger_als_MAXLEN_ist`

13.4.4 Konstanten

13.4.4.1 Ganzzahlkonstanten

Ganzzahlkonstanten (Datentyp ***int)*** werden nach folgender Syntax geschrieben

Syntax:

dez-integer-konstante : ['+' | '-'] *ziffer-ohne-0* { *ziffer* } .

ziffer-ohne-0 : '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' .

ziffer-ohne-0 bezeichnet die dezimalen Ziffer-Symbole von '1' bis '9'

ziffer : '0' | *ziffer-ohne-0* .

ziffer bezeichnet die dezimalen Ziffer-Symbole von '0' bis '9'.

Beginnt die Zahlenfolge mit **0x**, so wird sie hexadezimal interpretiert. In diesem Fall zählen auch A,B...F (bzw. a,b,...f) zu den erlaubten Zeichen.

Beispiele:

```
78,  +675,  -10,   0xaf33,   0x123F
```

13.4.4.1 Ganzzahlkonstanten (Fortsetzung)

		Wertebereich	Wertebereich ber.
int (=signed int) **unsigned int**	z.B. 4 Byte (s.u.) "	-2^{31} .... $2^{31}-1$ 0 .... $2^{32}-1$	-2147483648 .... 2147483647 0 4294967295
long int **unsigned long int**	meist 4 Byte "	-2^{31} .... $2^{31}-1$ 0 .... $2^{32}-1$	-2147483648 .... 2147483647 0 4294967295
short int **unsigned short int**	meist 2 Byte "	-2^{15} .... $2^{15}-1$ 0 .... $2^{16}-1$	-32768 .... 32767 0 65535
char (=signed char) **unsigned char**	1 Byte "	-2^{7} .... $2^{7}-1$ 0 .... $2^{8}-1$	-128 .... 127 0 255

Laut ANSI muss die ***int***-Variable mindestens eine Größe von 16 Bit aufweisen
Größe von **int** ist maschinenabhängig:

- auf 16-bit-Maschinen (z.B. 68000) ist **int** i.Allg. 16 Bit groß,
- auf 32-bit-Maschinen (z.B. PowerPC, ARM) ist **int** i.Allg 32 Bit groß.

Hintergrund: um C als Implementierungssprache auf den verschiedensten Rechnersystemen (Micro-Controller, Grossrechner) einsetzen zu können.

Anm.: die gültigen Größen stehen in "*limits.h*"

Beispiel: Auszug aus `limits.h` (Gnu ARM)

```
....
....

#define SHRT_MIN   (-32767-1)        /* minimum signed   short value */
#define SHRT_MAX     32767           /* maximum signed   short value */
#define USHRT_MAX    65535U          /* maximum unsigned short value */

#define LONG_MIN   (-2147483647L-1)  /* minimum signed   long value */
#define LONG_MAX     2147483647L     /* maximum signed   long value */
#define ULONG_MAX    4294967295UL    /* maximum unsigned long value */

#define INT_MIN      LONG_MIN        /* minimum signed   int value */
#define INT_MAX      LONG_MAX        /* maximum signed   int value */
#define UINT_MAX     ULONG_MAX       /* maximum unsigned int value */
....
....
```

13.4.4.2 Gleitkommakonstanten

Gleitkommakonstanten (Datentyp ***float***) gehorchen der folgenden Syntax:

Syntax:

gleitpunkt-konstante : ['+' | '-'] (*dez-darstellung* | *exp-darstellung*) .

dez-darstellung : *ziffer* { *ziffer* } '.' { *ziffer* } | { *ziffer* } '.' *ziffer* { *ziffer* } .

exp-darstellung : (*ziffer* { *ziffer* } | *dez-darstellung*) ('e' | 'E') ['+' | '-'] *ziffer* { *ziffer* }

Unter Ziffer wird die Menge der dezimalen Ziffer-Symbole von '0' bis '9' verstanden.

Beispiele:

```
78.4,  +0.675,  -1045.125576,  5. ,  .45
0.784e2,   +67.5E-2,   -0.1045125576e+4,  3e5
```

13.4.4.2 Gleitkommakonstanten (Fortsetzung)

		Wertebereich	signifikante Stellen
float	meist 4 Byte	$3.4*10^{-38}$ ..... $3.4*10^{38}$	7
double	meist 8 Byte	$1.7*10^{-308}$ ..... $1.7*10^{308}$	16
long double	meist 10 Byte	$3.4*10^{-4932}$ ..... $3.4*10^{4932}$	19

Anm.:

- Die gültigen Größen stehen in "*float.h*".
- Viele Maschinen besitzen eine 64 Bit Floatingpoint-Einheit, so dass der Datentyp **float** keine Rechenzeitvorteile bringt.
  Der Datentyp **double** ist daher i.allg. die bessere Wahl.
- **long double** ist vor allem dann zweckmäßig, wenn eine hohe numerische Genauigkeit erforderlich ist (z.B. sog. schlecht konditionierte Gleichungs-systeme).

Beispiel: Auszug aus `float.h` (Borland C++-Builder)

```
......
......
#define   DBL_MANT_DIG          53
#define   FLT_MANT_DIG          24
#define   LDBL_MANT_DIG         64

#define   DBL_EPSILON           2.2204460492503131E-16
#define   FLT_EPSILON           1.19209290E-07F
#define   LDBL_EPSILON          1.08420217248550443 4e-019L

/* smallest positive IEEE normal numbers */
#define   DBL_MIN               2.2250738585072014E-308
#define   FLT_MIN               1.17549435E-38F

#define   DBL_MAX_10_EXP       +308
#define   FLT_MAX_10_EXP       +38
#define   LDBL_MAX_10_EXP      +4932
.....
```

13.4.4.3 Zeichenkonstanten

Zeichenkonstanten (Datentyp ***char***) =
einzelne in einfache Anführungsstriche eingeschlossene Zeichen:

Beispiele: `'a', 'R', '1', '8'`

Der Wert der Konstante ist der numerische Wert des Zeichens im Zeichensatz der jeweiligen Maschine.

Darüberhinaus sind sog. Escape-Sequenzen erlaubt, z.B. (s. ASCII-Tabelle):

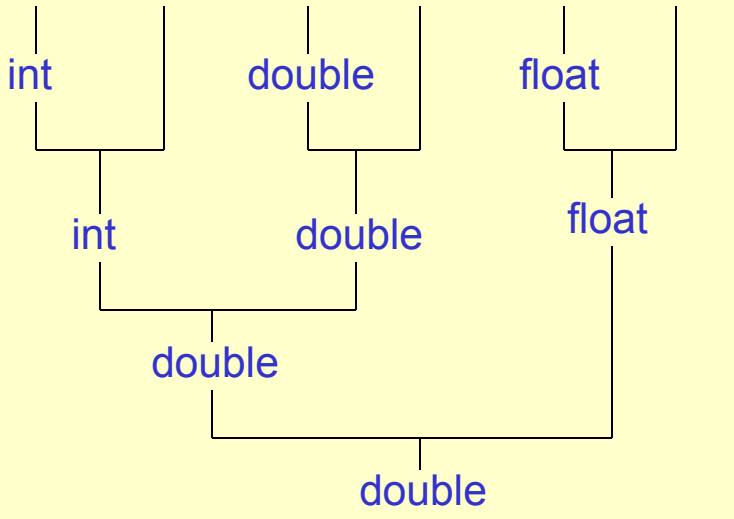
`\n`	Zeilenumbruch (CR)
`\r`	Wagenrücklauf (LF)
`\f`	Seitenwechsel (FF)
`\b`	Backspace (BS)
`\0`	Nullzeichen (NUL)
...	...

13.4.4.4 Implizite Typkonvertierungen

Werden in math. Ausdrücken verschiedene Typen kombiniert, wird bei jeder Operation der "niedrigere" Typ in den höheren konvertiert.

Beispiel:

```
char   ch;
int    i;
float  f;
double d;
...
result = (ch / i) + (f * d) - (i + f);
```



13.4.4.5 Explizite Typkonvertierungen (= casting)

Gelegentlich werden auch explizite Typvereinbarungen benötigt. Hierzu wird im math. Ausdruck vor die zu konvertierenden Variablen der Zieltyp in Klammen gesetzt:

Beispiel:

```
int    a=7;
int    b=2;

double Erg1, Erg2;

...

Erg1 = (double)a/(double)b;   /* Erg1 = 3.5     */

Erg2 = a/b;                   /* Erg2 = 3.0 !!! */
```

13.4.5 Strings

String (Zeichenkette = Folge von Zeichen umgeben von Anführungszeichen ("...") .

Intern wird der Zeichenkette ein NUL-Zeichen (\0) angehängt, wodurch das Ende der Zeichenkette markiert wird.

Beispiel: `"ABC 123 \n"`

Im Speicher steht dann:

0x41	*0x42*	*0x43*	*0x20*	*0x31*	*0x32*	*0x33*	*0x20*	*0x0D*	*0x00*

ASCII-Merkregel:

Zahlen beginnen bei 0x30.
Großbuchstaben beginnen bei 0x41.
Kleinbuchstaben beginnen bei 0x61.
Leerzeichen ist 0x20.

13.4.6 Operatoren

13.4.6.1 Binäre Operatoren: Übersicht

+ - * / %	Addition Subtraktion Multiplikation Division Modulo-Division (Rest)	Arithmetik	Zahlen (u. teilw. Zeiger) " " " ganze Zahlen
< <= == != >= >	kleiner kleiner gleich gleich nicht gleich größer gleich größer	Vergleich	alle Typen " " " " "
& | ^ << >>	bitweise AND bitweise OR bitweise XOR bitw. linksschieben bitw. rechtsschieben	Bitoperationen	ganze Zahlen " " " "
&& ||	log. AND log. OR	Logik	boolsche Werte "

13.4.6.2 Binäre Operatoren: Arithmetische Operatoren

Ausdrücke mit arithmetischen Operatoren liefern einen numerischen Wert.
Bei der Auswertung eines Ausdruckes wird der Vorrang der Operatoren beachtet.

- 3 + 5 * 4 – 2 liefert den Wert 21, weil der Vorrang von * beachtet wird.
- (3 + 5) * (4 – 2) liefert den Wert 16, weil durch die Klammern die Addition und die Subtraktion Vorrang vor der Multiplikation erhalten.

Vorrangstufen der numerischen Operatoren

Stufe	Operanden	Erläuterung	Auswertung
5	( )		von links
4	+ -	unären Operatoren (Vorz.)	von rechts
3	* % /		von links
2	+ -	binären Operatoren	von links
1	=	Zuweisung	von rechts

13.4.6.2 Binäre Operatoren: Anmerkung zur Auswertereihenfolge

Stehen mehrere (bezüglich der Vorrangstufe) gleichwertige Operationen hintereinander, sind die Reihenfolgeregeln anzuwenden:

Beispiel: Die ganzzahlige Berechnung von "3 * 11 / 4" ergibt:

a) von links ausgewertet (3 * 11) / 4 = 33 / 4 = 8 !
b) von rechts ausgewertet 3 * (11 / 4) = 3 * 2 = 6 !

--> Das Assoziationsgesetz gilt nicht mehr !!!!
C würde Lösung a) liefern.

Zur Vermeidung solcher schwer durchschaubarer Rechenregeln sollte man entweder

a) Klammern : Klammern machen zweifelsfrei die Reihenfolge deutlich
b) oder einen höheren Zahlentyp wählen (z.B. double).
(in diesem Fall wäre das Ergebnis 8.25)

ÜBUNG: Arithm. Operatoren

Gegeben seien:

```
char    c1='5', c2=25, c3, c4=-1;
int     i=5, j=9, k=-15, m, n, p, q;
double  d1=12.5, d2=2.0E-3, d3=-100, d4, d5, d6;
```

Berechnen Sie

```
m  = d1*i;
d4 = 9/2;
n  = k%4;
j  = j+1;
p  = d2*750 + 0.1;
d5 = (double)j/i + 0.2;
q  = (unsigned char)c4;
d6 = 22%5*3%2;
c3 = (c1-0x30)+c2;
c3 = 64*8;
```

Welche Ausdrücke zeugen von schlechtem Stil ?

13.4.6.3 Binäre Operatoren: Vergleichsoperatoren

Vergleichsoperatoren liefern ebenfalls ein numerisches Ergebnis:

- Vergleich falsch: 0
- Vergleich richtig: 1

Bei der Auswertung ist auf den Vorrang zu achten.

Vorrangstufen der numerischen Operatoren

Stufe	Operanden	Erläuterung	Auswertung
7	( )		von links
6	+ -	unären Operatoren	von rechts
5	* % /		von links
4	+ -	binären Operatoren	von links
3	< <= => >		von links
2	== !=		von links
1	=	Zuweisung	von rechts

Beispiele

3 < 5 - 4	liefert den Wert 0	
(3 < 5) - 4	liefert den Wert - 3	(Unsinn: Typ-Kuddelmuddel)
3 < 5 < 4 < 2	liefert den Wert 1	(Unsinn: “ “)

13.4.6.4 Binäre Operatoren: Logische Operatoren

Logische Operatoren (&&, ||, !) liefern ebenfalls ein numerisches Ergebnis:

- Aussage falsch (false): 0
- Aussage richtig (true): 1

Die logischen Operatoren stehen in engem Zusammenhang mit den Vergleichsoperatoren.

Bei der Auswertung ist auf den Vorrang zu achten .

Vorrangstufen der log. Operatoren und Vergleichsoperatoren

Stufe	Operanden	Erläuterung	Auswertung
7	!		
6	< <= => >		von links
5	==, !=		von links
4	&&	log. Operatoren (AND)	
3	||	" " (OR)	
1	=	Zuweisung	von rechts

In C wird jeder Ausdruck **ungleich 0** logisch interpretiert als wahr (true) betrachtet !!

ÜBUNG Vergleichsoperatoren, log. Operatoren

Was ergeben folgende Ausdrücke:

```
char  b1, b2, b3, b4, b5, b6, b7;
int   Zahl=7, Z2=25;

b1 = !(Zahl > 10) && !(Zahl < 5);
b2 = (Zahl <=10) && (Zahl >= 5);
b3 = Zahl <= 10 && Zahl >= 5;
b4 = (Zahl - 10) && (Zahl - 7);
b5 = 5 <= Z2 <= 10;
b6 = 5 <= Z2 && Z2 <= 10;
b7 = !(Zahl=7);
b8 = !(Zahl < 10) || Zahl==7 || !(Zahl-7);
```

Welche Ausdrücke zeugen von schlechtem Stil ? Welche sind unsinnig ?

ÜBUNG Schaltjahrberechnung

Zu einer gegebenen Jahreszahl ist zu berechnen, ob es sich um ein Schaltjahr handelt. Die Regel für Schaltjahre lautet:

Ein Schaltjahr liegt dann vor, wenn

- die Jahreszahl durch 4 teilbar ist,
- ausser sie ist durch 100 teilbar.
- Einzige Ausnahme: Ist die Jahreszahl durch 400 teilbar, liegt jedoch trotzdem ein Schaltjahr vor.

Sind die Jahre 1800 und 2000 Schaltjahre ?

Geben Sie einen Ausdruck an, der den log Wert 1 (true) ausgibt, wenn ein Schaltjahr vorliegt.

13.4.6.5 Unäre Operatoren: Übersicht

& *	Adresse von ... Inhalt von ...	Referenzierung Dereferenzierung	alle Typen Zeiger
+ -	pos. Vorzeichen neg. Vorzeichen	Arithmetik "	Zahlen "
~	bitweise invertieren	Bitoperation	ganze Zahlen
!	log. invertieren	Logik	boolesche Werte
(Zieltyp)	Typumwandlung		
sizeof	Speicherbedarf		Ausdrücke u. Typen
++ --	Incrementierung Decrementierung	Prä- und Post-increment/-decrement	ganze Zahlen und Zeiger

13.5 Variablen und Vereinbarungen

Variablen = Datenobjekte, deren Wert im Programmverlauf geändert werden kann.
Alle Variablen müssen vor ihrer ersten Benutzung in einem Programm deklariert werden.

Datenobjekte werden nach folgender Syntax vereinbart.

Syntax:

declaration : *type-spezifier* *init-declarator* { ',' *init-declarator* } ';' .

type-spezifier : 'int' | 'float' | 'double'|.....usw... . Anm.: unvollständig !

init-declarator : *variablen-bezeichner* ['=' *initialwert*] .

Beispiele:

```
int     zaehler, increment = 5 ;
float   pi = 3.1415926 ;
```

13.6 Anweisungen

Sequenzen

- Verbund-Anweisung
- Ausdrücke als Anweisungen
- Funktionsaufrufe

– **Verzweigungen**

- if-else - Anweisung
- switch - Anweisung

– **Schleifen**

- while - Anweisung
- do-while - Anweisung
- for - Anweisung

13.6.1 Verbund-Anweisung

Die Verbundanweisung (*compound-Statement*) – auch Block genannt - eine mit '{' und '}' geklammerte Folge von einzelnen Vereinbarungen und Statements.

Syntax:

compound statement = '{' { *declaration* } { *statement* } '}' .

Der Verbund dient dazu einzelne Anweisungen zu gruppieren, um sie syntaktisch wiederum als eine Anweisung betrachten zu können.

Der Verbund ist auch ein Namensraum/Gültigkeitsbereich für die in ihm vereinbarten Datenobjekte.

Vereinbarungen können an jeder Stelle getroffen. Es ist sinnvoll sie zu Beginn des Blockes vorzunehmen.

Der Kontrollfluss in der Verbundanweisung ist der einer Sequenz.

```
{
    int a, b, c; a=1; b=2; c=a+b;
}
```

13.6.2 IF-ELSE-Anweisung

Syntax:

'**if**' '(' *expression* ')' *statement* ['**else**' *statement*]

Beispiele:

```
if ( a < b ) b = a; else a = b;

if ( a >= b) {
          h = sin(b);
          a = b*b;
}
```

Anmerkung: In C gilt jeder Wert ungleich 0 als "Wahr" !

13.6.2 IF-ELSE-Anweisung (Fortsetzung)

Ein häufig angewendeter Konstrukt ist die sog. *Else-If-Kette* :

'if' '('*expression*')'
    statement
'else' 'if' '('*expression*')'
    statement
'else' 'if' '('*expression*')'
    statement
...

...

'else'
    statement

Beispiel:

```
if(a<0){
        V=-1;
        printf("negative! \n");
}
else if(a==0){
        V=0;
        printf("zero! \n");
}
else{
        V=1;
        printf("positive! \n");
}
```

Wichtig : Ein *else* gehört immer zum (rückwärtsgehend) letzten *if*, welches noch kein *else* hat.

ÜBUNGEN: if-else-Anweisung

Welcher Wert wird ausgegeben? (Achtung: C-Puzzles, ganz schlechter Code)

Aufg. 1

```
int i=0, a=0, b=2;
…
if (i==0)
    a=7;
else
    b=15;
    a=b+1;
printf("%d",a);
```

Aufg. 2

```
int i=0,b=10,a;
…
if (i=0)
    a=10;
else if (i=2){
    b=15;
    a=b+1;
}
else a=0;
printf("%d",a);
```

Aufg. 3

```
int i=1,a=5,b=10;
…
if (i==0);
{
    a=10;
    b=a+1;
}
printf("%d",a);
```

ÜBUNG: if-else-Anweisung - Wahl der richtigen Bedingung

(Achtung: C-Puzzles, ganz schlechter Code)

1. Vereinfachen Sie:

```
if (a){
   if(b)
      if(c) D;
}else
   if(b)
      if(c) E;
      else  F;
   else;
```

2. Was wird gedruckt:

```
x = -1;
y = -1;

if (x>0)
    if(y>0)
        print("1");
else
    print("2");
```

Anm.: a,b,c sind beliebige expressions
D,E,F sind beliebige statements

13.6.3 Switch-Anweisung

Diese Kontrollstruktur ist als Mehrfachverzweigung oder Verteiler gedacht.

'switch' '(' *expression* ')'
{ [**'case'** constant-expression ':' [*statement*]... [**'break'**]] ...
[**'default'** : [*statement*] ...]
}

Beispiel:

```
switch ( character )
{
   case 'a' :
   case 'A' : alpha = 'a';  break ;
        ....
   case 'z' :
   case 'Z' : alpha = 'z';  break ;

   default : alpha = '0';
}
```

Achtung: Das *break* ist notwendig, damit nicht die folgenden Fälle abgearbeitet werden!

ÜBUNGEN: Switch-Anweisung

1. Was wird ausgegeben?

```
int in=2;
...
switch(in){
        case 1:
        case 2:
        case 4:
        case 6:
                printf("A");
        case 9:
                printf("u");
                printf("t");
        case 10:
                printf("o");
                break;
         case 11:
                printf("ma");
        default:
                printf("t \n");
}
```

2. Geben Sie zu folgendem Ausdruck eine äquivalente switch-Anweisung an:

```
if ((num<=8) && (num>=5))
        printf("schlecht \n");

else if ((num==3) || (num==4))
        printf("mittel \n");

else if ((num<=2) && (num>0))
        printf("gut \n");

else
        printf("unmoeglich \n");
```

13.6.4 WHILE-Anweisung (kopfgesteuerte Schleife)

Syntx:

'**while**' '(' *expression* ')' *statement*

Die Anweisung wird ausgeführt solange der Prüfausdruck wahr ist (!= 0).

Beispiele:

```
while( a < b )
      a = a + 2;

while( y<=10 ){
      x=x+1;
      y=y+1;
}
```

13.6.5 DO-WHILE-Anweisung (Fußgesteuerte Schleife)

Syntax:

'do' *statement* **'while'** '(' *expression* ')' ;

Die Anweisung wird ausgeführt solange der Prüfausdruck wahr ist (!= 0).

Beispiel:

```
do{
      summe = summe +1;
      i = i + 1;
}while (i <= 100);
```

13.6.6 FOR-Anweisung

'for' '(' [*expression 1*]; [*expression 2*]; [*expression* 3] ')' *statement*

Üblicherweise:

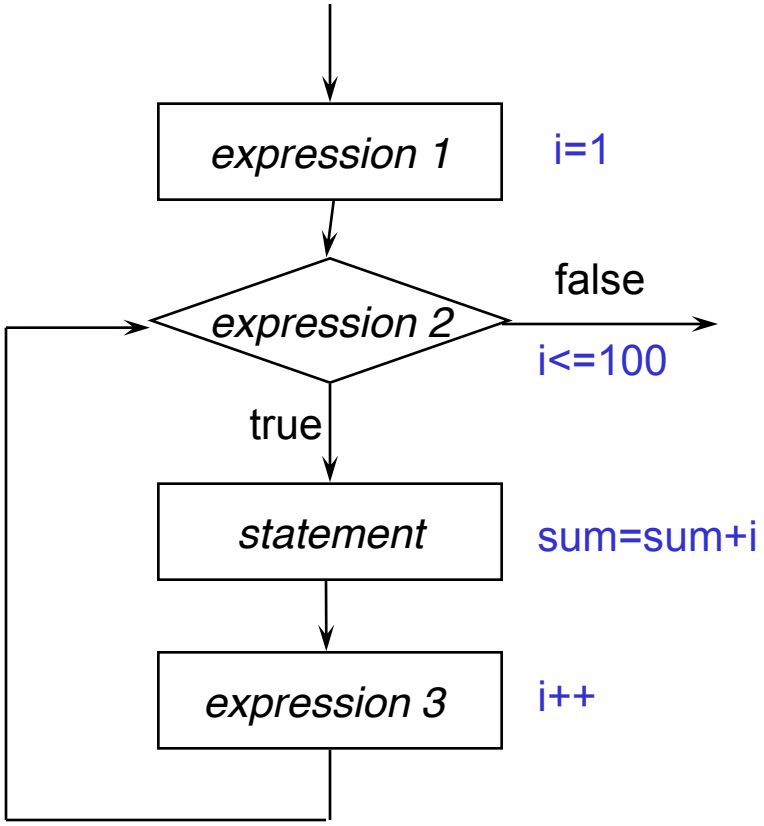
expression 1: Initialisierung

expression 2: Bedingung

expression 3: Incrementierung / Decrementierung

Beispiel:

```
sum=0;
for (i=1; i<=100; i++)
    {
      sum = sum + i;
    }
```



ÜBUNGEN: for-Schleife

1. Schreiben Sie ein Programm, welches alle Zahlen des "2-aus-5-Code" tabellarisch ausgibt, also:

```
00011
00101
00110
...
11000
```

2. Was wird ausgegeben?

```
char ch2[]="Irgendein Text.";
char ch1[]="Irgendein Text.";
int  Len,i,k;

Len=strlen(ch1); /* liefert die Stringlaenge */
for(i=0, k=Len-1;  i<Len;  i++,k--){
        ch2[k] = ch1[i] ;
}
printf("%s \n", ch2);
```

13.6.7 break und continue

break dient dazu eine Schleife (while, do, for) vorzeitig zu verlassen.

Typischer Anwendungsfall ist der Abbruch einer Schleife, wenn besondere Bedingungen vorliegen.

I.allg. sollte eine strukturierte Lösung dem *break* vorgezogen werden. Gelegentlich kann der Code durch *break* auch übersichtlicher werden.

Darüberhinaus beendet *break* ebenfalls switch-Anweisungen. Hier ist *break* sinnvoll und üblich.

continue dient in Schleifen (while, do, for) dazu, die nächste Wiederholung der umgebenden Schleife sofort zu beginnen.

Beispiel:

```
/* druckt alle Zahlen von 0 ... 99, mit Ausnahme */
/* der durch 5 teilbaren Zahlen.                 */
for(i=0; i<100; i++){
    if(i%5 == 0) continue;
    printf("%2d \n",i);
}
```

Frage: Wie könnte man das Programm besser (ohne *continue*) schreiben?

ÜBUNG: Wahl der richtigen Kontrollstruktur

1. Vereinfachen Sie:

```
int i=0;
while(i<100) {
   if((x=x/2) > 1) {
      i++;
      continue;
   }
   break;
}
```

2. Vereinfachen Sie:

```
int i,total=0,count=0;
/* Zufallszahl von 0..100 */
while((i=random(100)) != 0) {
   total++;
   if(i==50)   break;
   if(i%3==0)  continue;
   if(i%5==0)  count++;
}
```

13.7 Funktionsdefinitionen

Funktionen sind die Träger von Algorithmen bzw. Berechnungen.
Funktionen werden nach folgender Syntax vereinbart.

SYNTAX:

funktionsdefinition : *fkt-kopf* *fkt-rumpf* .

fkt-kopf : res-*type-spezifier* *fkt-bezeichner* '(' *formale-argumente* ')' .
fkt-rumpf : *verbund-anweisung* .

res-type-spezifier : 'void' | 'int' | 'float' . Anm.: unvollständig !
formale-argumente : 'void' | *argumentenliste*

argumentenliste : *type-spezifier* *variablen-bezeichner*
{ ',' *variablen-bezeichner* }
{ ',' *type-spezifier* *variablen-bezeichner*
{ ',' *variablen-bezeichner* } } .

Beispiele:

```
void wenig ( void ) { .... }
int inkrementiere ( int zahl, int incr ) { .... }
void update ( float druck, temp, int wert ) { .. }
```

13.7 Funktionsdefinition (Fortsetzung)

Deklaration

= Bekanntgabe von Name und Typ eines Objektes an den Compiler.

Definition

= Detailbeschreibung eines Objektes.

```
/* Deklaration von Variablen*/
int  i, j, k;
char txt;

/* Deklaration von Funktionen */
int QuadSum (int x, int y);


int main(){
   ...
   k=QuadSum(m,n);   /*Aufruf*/
   ...
}
```

```
/* Definition einer Funktion */


int QuadSum (int x, int y)
{
    int res;
    res = x * x + y * y;
    return res;

}
```

13.8 Return-Statement

Das Return - Statement beendet die Ausführung einer Funktion und gibt, sofern für die Funktion ein Ergebnistyp vereinbart wurde, das erarbeitete Ergebnis an seine Aufrufumgebung zurück.

Syntax:

return [*expression*] ;

Der Wert des optionalen Ausdrucks liefert den Return-Wert der Funktion und muss zum Ergebnistyp der Funktion kompatibel sein.

Beispiel:

```
return ( a + b )/2 ;
```

ÜBUNG Funktionsdefinition / Funktionsdeklaration

Schreiben Sie eine C-Funktion `XNorm()` zur Berechnung von:

$$x_{Norm} = \left| \frac{x}{\sqrt{x^2 + y^2}} \right|$$

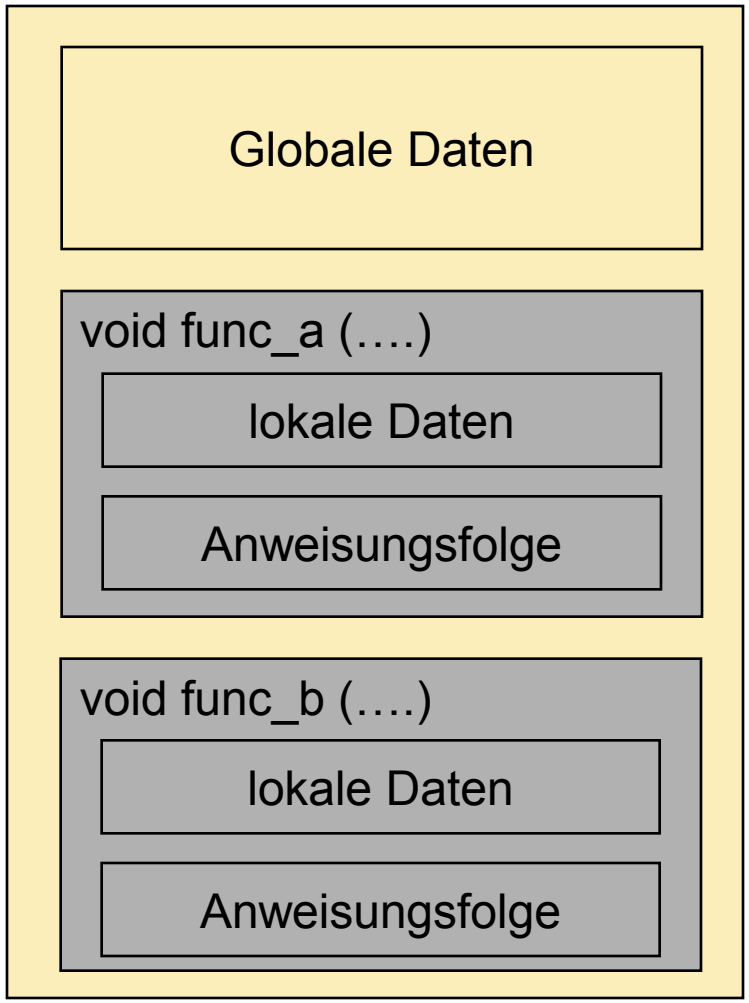
x und y sind vom Typ `int`, der Rückgabewert sei vom Typ `double`.
Im Fall x=y=0 soll der Rückgabewert 0.0 sein.

Es können folgende Bibliothekskunktionen verwendet werden:

- `double sqrt(double x)`
- `double fabs(double x)`

Schreiben Sie ein Testprogramm, welches die Funktion aufruft.
Das Testprogramm und die Funktion `XNorm()` sollen in verschiedenen Dateien realisiert werden (`Test.c`, `Norms.c`).

13.9 Gültigkeitsbereiche und Speicherklassen



Dargestellt ist ein Modul (Texteinheit).

Die globalen Daten sind in allen Funktionen bekannt und die lokalen Daten nur in ihren Funktionen.

Die globalen Daten sind für die Funktionen extern.

Die lokalen Daten haben die gleiche Lebesdauer wie ihre Funktionen.

Wenn die Funktionen aufgerufen werden, werden die Datenobjekte angelegt. Sie werden aufgelöst, wenn die Funktion endet.

In diesem Sinn unterscheidet C zwischen statischen und automatischen Datenobjekten.

13.9 Gültigkeitsbereiche und Speicherklassen (Fortsetzung)

Es gibt vier Speicherklassen:

auto Lokale Variablen einer Funktion werden erzeugt wenn die Funktion aufgerufen wird und gelöscht, wenn die Funktion verlassen wird. Solche Variablen heißen "automatische Variablen".

static Innerhalb einer Funktion deklariert: Lokale Variablen, die ihren Wert zwischen den Funktionsaufrufen behalten.
Ausserhalb der Funktionen deklariert: Modul-globale Variablen, d.h. innerhalb des Moduls global, ausserhalb des Moduls unbekannt.

extern Außerhalb der Funktionen deklarierte Variablen stehen allen Funktionen des Moduls zur Verfügung (= globale Variablen).
Mit "extern" können diese Variablen auch über Modulgrenzen hinweg bekannt gemacht werden. (`extern` → „*Variable ist dem Linker bekannt*“)

register Variablen, die möglichst in den Prozessorregistern gehalten werden sollten (Geschwindigkeitsvorteil).

ÜBUNG: (Gültigkeitsbereiche)

```c
/* Modul_1.c */
#include "Modul_2.h"

int calc_i();  /* Fkt.-Deklaration */

static int i=0;

int main(){

    int i=1, m;
    printf("i=%d \n", i);
    printf("i=%d \n", calc_i());
    printf("i=%d \n", calc_i_ext());
    printf("k=%d \n", k);

    return 0;
}
```

```c
int calc_i(){
    return i;
}
```

Was wird ausgedruckt ?
Ist etwas falsch ?

```c
/* Modul_2.c */


int k=15;
static int i=2;

int calc_i_ext(){
    return i;
}
```

```c
/* Modul_2.h */

extern int k;
int calc_i_ext();
```

```
/* Modul_1.c */

   /* Modul_2.h */
   extern int k;
   int calc_i_ext();

int calc_i();  /* Fkt.-Deklaration */

static int i=0;

int main(){

    int i=1, m;          /* lokal */
    printf("i=%d \n",i);
    printf("i=%d \n",calc_i());
    printf("i=%d \n",calc_i_ext());
    printf("k=%d \n",k);

    return 0;
}

int calc_i(){
    return i;
}
```

```
/* Modul_2.c */


int k=15;
static int i=2;

int calc_i_ext(){
    return i;
}
```

```c
/* Modul_1.c */

   /* Modul_2.h */
   extern int k;
   int calc_i_ext();

int calc_i();  /* Fkt.-Deklaration */

static int i=0;      /* Modulglobal */

int main(){

    int i=1, m;
    printf("i=%d \n",i);
    printf("i=%d \n",calc_i());
    printf("i=%d \n",calc_i_ext());
    printf("k=%d \n",k);

    return 0;
}

int calc_i(){
    return i;
}
```

```c
/* Modul_2.c */


int k=15;
static int i=2;

int calc_i_ext(){
    return i;
}
```

```c
/* Modul_1.c */

   /* Modul_2.h */
   extern int k;
   int calc_i_ext();

int calc_i();  /* Fkt.-Deklaration */

int i=0;

int main(){

    int i=1, m;
    printf("i=%d \n",i);
    printf("i=%d \n",calc_i());
    printf("i=%d \n",calc_i_ext());
    printf("k=%d \n",k);

    return 0;
}

int calc_i(){
    return i;
}
```

```c
/* Modul_2.c */


int k=15;
static int i=2;

int calc_i_ext(){
    return i;
}
```

```
/* Modul_1.c */
   /* Modul_2.h */
   extern int k;
   int calc_i_ext();

int calc_i();  /* Fkt.-Deklaration */

int i=0;

int main(){

    int i=1, m;
    printf("i=%d \n",i);
    printf("i=%d \n",calc_i());
    printf("i=%d \n",calc_i_ext());
    printf("k=%d \n",k);

    return 0;
}

int calc_i(){
    return i;
}
```

Globale Variable !
Möglichst nicht verwenden !

```
/* Modul_2.c */


int k=15; /*Global*/
static int i=2;

int calc_i_ext(){
    return i;
}
```