Kontrollstrukturen

- Kontrollstrukturen dienen der Steuerung des Instruktionsflusses
- Grundelemente sind die Verzweigung und die Schleife
- Sprünge sind zulässig, sollten aber vermieden werden

if Anweisung 1

- Syntax:

if (*C*) *A1 [* else *A2]*

  - A1 wird ausgeführt, falls der Ausdruck C einen Wert ungleich 0 hat
  - ist der “else”-Zweig vorhanden, so wird andernfalls A2 ausgeführt

if Anweisung 2

- Beispiel:

```
if (1)
  printf("wahr\n");
else
  printf("falsch\n");

if ((x-y<=0) && (y-x<=0))
  printf("x ist gleich y\n");
```


if Anweisung 3

- die Anweisungen A1 und A2 können auch Blöcke sein, also

```
if ( C ) {
      A11 ... A1n
} else {
      A21 ... A2m
}
```

if Anweisung 4

```
int x[10];

if ((x[2]=5) & 1) {
     x[1]=-67;
     printf(”x[2]==%d\n”,x[2]);
} else {
     if (x[0]==-1)
          printf(”hallo\n”);
     else
          printf(”tschuess\n”);
}
```


switch Anweisung 1

- Beispiel:
```
if (C==C1) A1;
if (C==C2) A2;
...
if (C==Cn) An;
```
  hier werden alle Ausdrücke C und C1,...,Cn ausgewertet
- semantisch gleich:
```
if (C==C1) A1;
else if (C==C2) A2;
...
else if (C==Cn) An;
```
  hier werden nur die Ausdrücke C und C1..Ci ausgewertet, für die C1,..,C(i-1) != C und C==Ci ist; C wird i-mal ausgewertet

switch Anweisung 2

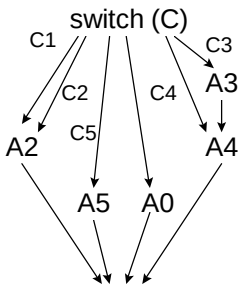
- syntaktisch und semantisch eleganter: die switch-Anweisung

```
switch (C) {
        case C1: A1
        case C2: A2 break;
        ...
        case Cn: An break;
        default: A0
}
```


switch Anweisung 3

- die “break” Anweisung bewirkt das Verlassen des aktuellen Blockes

```
switch (C) {
        case C1:
        case C2: A2 break;
        case C3: A3
        case C4: A4 break;
        case C5: A5 break;
        default: A0
}
```



switch Anweisung 4

```
typedef struct { int x,y; } Point;
typedef struct { int x,y,r; } Circle;
typedef struct { Point A,B; } Rectangle;
typedef union { Point point;
                Circle circle;
                Rectangle rect; } GeometryElement;
typedef enum {POINT,CIRCLE,RECTANGLE} Gtype;
typedef struct {
  Gtype type;
  GeometryElement ge;
} Geometry;
```


switch Anweisung 5

```
Geometry g;
g.type=RECTANGLE;
...
switch (g.type) {
  case POINT:
       printf("Point(%d,%d)\n",g.ge.point.x,
              g.ge.point.y);
       break;
  case CIRCLE:
       printf("Circle(%d,%d,%d)\n",g.ge.circle.x,
              g.ge.circle.y,g.ge.circle.r); break;
  case RECTANGLE:
       printf("Rectangle((%d,%d),(%d,%d))\n",
              g.ge.rectangle.A.x,g.ge.rectangle.A.y,
              g.ge.rectangle.A.x,g.ge.rectangle.A.y);
}
```

Schleifen

- Schleifen dienen der wiederholten Ausführung einer Anweisung
- C kennt drei Typen von Schleifen
  - abweisende Schleifen
    - » do-while-Schleife
    - » for-Schleife
  - akzeptierende Schleife
    - » while-Schleife

do-while Schleife 1

- Syntax:

  do *A* while (*C*) ;

  - die Anweisung A wird ausgeführt
  - hat der Ausdruck C danach einen Wert ungleich 0, so beginnt die Abarbeitung von vorn

do-while Schleife 2

- Beispiel:

```
int i=0;
do printf(”>%d\n”,i); while (i++<3);
```

```
>0
>1
>2
>3
```

Durchlauf	i	i++	i
1	0	0	1
2	1	1	2
3	2	2	3
4	3	3	4
5	-	-	-

do-while Schleife 3

```
float x,xn,y,yx,eps=1e-10;
int fehler=0;
xn=2.0;
do {
  x=xn;
  y=sin(x);      /* Funktionswert */
  yx=cos(x);     /* Wert der Ableitung */
  if ((yn>1e-20) || (yn<-1e-20))
    xn=x-y/yn;
  else
    fehler=1;
} while ( fabs(x-xn)>eps) && (!fehler))
if (!fehler) printf(”X0=%f\n”,xn);
  else print(”f’(%f)=0\n”,xn);
```

while Schleife 1

- Syntax:

while (*C*) *A*

  - hat der Ausdruck C einen Wert ungleich 0, so wird A ausgeführt
  - nach Abarbeitung von A beginnt der Zyklus von vorn

while Schleife 2

- Beispiel:

```
int i=0;
while (++i<3) printf(">%d\n",i);
```

```
>1
>2
```

Durchlauf	i	++i	i
1	0	1	1
2	1	2	2
3	2	3	-

for Schleife 1

- Syntax:

  for (A0 ; C ; I) A

  - vor Eintritt in die Schleife wird A0 ausgeführt
  - Test: vor jeder Iteration wird geprüft, ob C einen Wert ungleich 0 hat, wenn ja, wird A ausgeführt
  - nach A wird I ausgeführt und die Iteration beginnt erneut beim Test

for Schleife 2

- Beispiel:

```
int i;
for ( i=0 ; i<3 ; i++ )
    printf("%d ",i);
```

  produziert die Ausgabe von “0 1 2”

- alternative Schreibweise:

```
for (i=0 ; i<3 ; printf("%d ",i++));
```

for Schleife 3

- Beispiel (for mit ","-Ausdruck):

```
for (i=n-1;x[i]=0,i>=0;i--) ;
```

ist äquivalent zu

```
for (i=n-1;i>=0;i--) x[i]=0;
```


Sprünge

- C kennt Marken und Sprünge

```
label :
goto label ;
```

ES GIBT KEINEN CODE, DER DIE VERWENDUNG VON SPRÜNGEN ZWINGEND ERFORDERT!

Blockkonzept 1

- ein Block in C ist keine Block-Anweisung im Sinne von Pascal, kann aber äquivalent genutzt werden

C	Pascal
{	begin
A1 ... An	A1 ; ... ; An
}	end

Blockkonzept 2

- ein Block ist der Gültigkeitsbereich einer Variablen
  - `ig` ist im äußeren und im inneren Block gültig
    `ig` heißt "global" bzgl. des inneren Blocks
  - `il` ist nur im inneren Block gültig
    `il` heißt "lokal" bzgl. des inneren Blocks
  - das `x` des äußeren Blocks wird im inneren verdeckt

äußerer B.
innerer B.

```
int ig,x;
{
    int il,x;
}
```

Blockkonzept 3

```
int x[10],y[10],i;
int n=10;

for (i=0;i<n;i++){
  int i,s=0;
  for (i=0;i<n;i++) {
    x[i]=...;
    s=s+x[i];
  }
  y[s]=...;
}
```


Funktionen 1

- Funktionen sind die grundlegenden strukturellen Einheiten eines C Codes
- F. erlauben das Aufspalten großer Aufgaben in kleinere
- F. verbergen ihre Implementation
- F. können wiederverwendet werden

Funktionen 2

- Funktionen berechnen einen Wert und nutzen dazu übergebene Argumente; im ANSI Standard wird dies mit *Funktionsprototypen* ausgedrückt
- Funktionsprototyp:

Typ Funktionsname (Argumentliste) ;

Funktionen 2

- Beispiel:

```
int abs(int);
```

Rückgabetyp Funktionsname Argumenttyp

- die Funktion "`abs`" berechnet aus einem ganzzahligen Argument den Betrag, der ebenfalls ganzzahlig ist
- in math. Notation abs: $Z \rightarrow Z$

Funktionen 3

- weiteres Beispiel:

```
float scalar(float[],float[],int);
```

- generell ist jeder vorher definierte Datentyp sowohl als Argumenttyp, als auch als Rückgabewert zulässig

Funktionen 4

- C kennt einen speziellen Typ “void” um das Fehlen eines Rückgabewertes auszudrücken

```
void f1();
```

  ist äquivalent zu

```
void f1(void);
```

- void darf anstelle einer Typliste stehen

Funktionen 5

- nachdem eine Funktion mittels ihres Prototypes deklariert wurde, muß sie definiert werden
- ein F.-Def. besteht grundsätzlich aus dem Funktionskopf mit dem Funktionskörper

fct_head { *fct_body* }

Funktionen 6

- der Funktionskopf definiert den Typ des Rückgabewertes, den Funktionsnamen und die Argumente

  Beispiel:

```
int abs(int x)
```

Funktionen 7

- im Funktionskörper wird die Vorschrift zur Bildung des Funktionswertes angegeben; der Funktionskörper ist ein Block
- die Rückgabe des Funktionswertes erfolgt mit der “return”-Anweisung
- “return” beendet die Abarbeitung der Funktion

Funktionen 8

- Beispiel:
- **y** ist nur innerhalb der Funktion gültig, ebenso **x**

```
int abs(int x) {
   int y;
   if (x<0)
     return -x;
   else
     y=x;
   return y;
}
```

Funktionen 9

- der Aufruf einer Funktion erfolgt durch ihren Namen mit Parametern eines passenden Typs

```
int x,y;
...
x=-19;
y=abs(x)+2;
```


Funktionen 10

- Funktionsaufrufe sind Ausdrücke, d.h. sie können gemäß ihrem Rückgabetyp mit anderen Ausdrücken verknüpft werden
- Beispiel:

```
float x,y;
float exp(float);
float sin(float);
y=exp(-x*x)*sin(x);
```

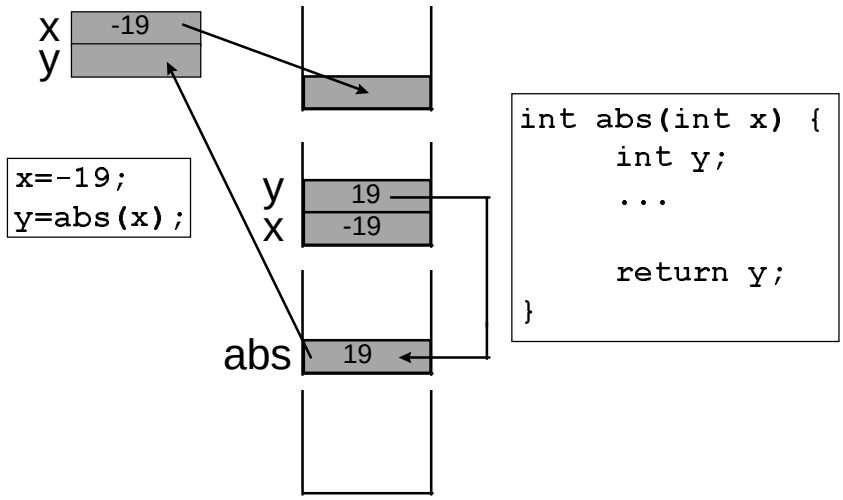
Funktionen - Stackprinzip

- C kennt zwei grundlegende Speichertypen
  - den Freispeicher (Heap), den der Programmierer selber verwalten muß
  - den Kellerspeicher (Stack), der nur dem Laufzeitsystem zugänglich ist
- der Stack wird zur Übergabe von Parametern an F. und zur Speicherung des Ergebnisses verwendet

Funktionsaufrufmechanismus 1

- wird eine Funktion gerufen, so werden zunächst von allen Argumenten Kopien auf dem Stack angefertigt
- danach wird die F. angesprungen und es wird Speicherplatz für alle lokalen Variablen der Funktion auf dem Stack bereitgestellt
- die Funktion wird abgearbeitet
- die “return”-Anweisung veranlaßt die Freigabe des belegten Kellerspeichers und speichert im Anschluß den Funktionswert auf dem Stack
- im rufenden Code wird der Wert des obersten Stack-Elements als Funktionswert verwendet und anschließend freigegeben

Funktionsaufrufmechanismus 2



Funktionsparameter

- eine Funktion arbeitet grundsätzlich mit einer Kopie der aktuellen Parameter
- durch die Verwendung von Referenzen auf Argumente kann aber auch direkt mit den aktuellen Parametern gearbeitet werden

Referenzen als Parameter 1

- alle Argumente, die in der Form

  Typbezeichner ***Argumentname***

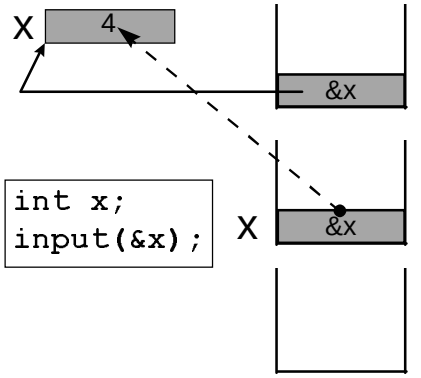
  erklärt werden und deren Wert innerhalb der Funktion nicht geändert wird, wohl aber der referenzierte Wert, können als Variablenparameter interpretiert werden

Referenzen als Parameter 2

- Beispiel:

```
void input(int *x);
int main() {
  int x;
  input(&x);
  return 0;
}
void input(int *x) {
  *x = 4;
}
```

Referenzen als Parameter 3



Funktionen und Ellipsen 1

- eine Funktion kann eine variable Anzahl von Argumenten haben
- mindestens ein Argument muß vorhanden sein

Typ FktName (*ArgTyp1* , ...) ;

Ellipse

Funktionen und Ellipsen 2

```
#include <stdarg.h>
#include <stdio.h>
void minprintf(int n, ...) {
  va_list args;
  int i,ival;
  va_start(args,n);
  for (i=0;i<n;i++) {
    ival=va_arg(args,int);
    printf("%d ",ival);
  }
  va_end(args);
}
```


Funktionen und Ellipsen 3

```
void minprintf(int, ...);

main() {
  minprintf(1,1);
  minprintf(5,1,2,3,4,5);
}
```

Funktionen 11

- Funktionen können nicht geschachtelt werden:

```
int f1() {
      int f2() {
            ...
      }
      ...
      f2();
}
```

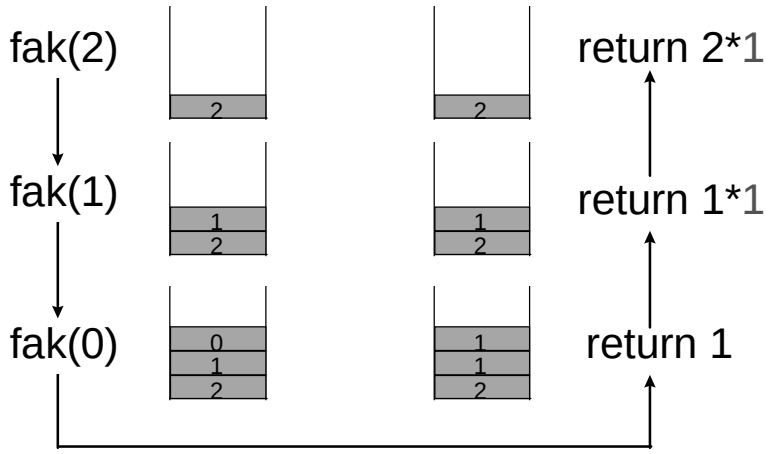
ist nicht zulässig!

Funktion ruft Funktion 1

- Stack ist Grundlage für das Prinzip der Rekursion (Funktion ruft sich selbst)
- viele Algorithmen lassen sich rekursiv sehr elegant formulieren
- Beispiel:

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

Funktion ruft Funktion 2



Funktionstyp 1

- auch Funktionen können typisiert werden (Funktionenzeiger)

```
float MaxNorm(float[],int);
float EukNorm(float[],int);
typedef float (*NormFct)(float[],int);
...
NormFct norm; float n,x[3];
norm=MaxNorm;
...
n=norm(x,3);
```

Funktionstyp 2

- Beispiel:

dieses Vorgehen erlaubt die Zuordnung von Funktionen zu Daten

```
typedef void (*Constructor)(void*);

typedef struct {
  Constructor Init;
} Object;

Object New(Constructor c);
void Object_A Object*);
void Object_B Object*);

main() {
  Object i1,i2;
  i1=New((Constructor)Object_A);
  i2=New((Constructor)Object_B);
  i1.Init(&i1); i2.Init(&i2);
}
```