Vom Problem zum Programm

- Kontrollstrukturen der prozeduralen Programmierung
  - Grundprinzipien: Sequenz – Alternative – Iteration
- Beschreibungsformen
  - Ablaufpläne – Struktogramme – Pseudocode
- Aussagenlogik

Kontrollfluss

- Bei einem Algorithmus kommt es entscheidend darauf an, in welcher Reihenfolge die einzelnen Anweisungen durchgeführt werden. Üblicherweise werden sie in der Reihenfolge ihres Vorkommens im Programm durchgeführt.
- Am Beispiel der ersten Aufgaben kann man aber sehen, dass es erforderlich ist, Fallunterscheidungen und gezielte Wiederholungen von Anweisungsfolgen zu ermöglichen.
- Dieser Teil führt in die Kontrollstrukturen von C++ ein. Eine Kontrollstruktur definiert, wie die einzelnen Anweisungen ausgeführt werden, z.B.
  - sequentiell,
  - bedingt oder
  - wiederholt.
- Kontrollstrukturen steuern somit den Programmfluss. Den Abschluss bildet die Behandlung von Ausnahmen.

Anweisungen und Blöcke

- Einfache **Anweisungen** (statement) werden mit Semikolon **abgeschlossen**. Die "leere" Anweisung besteht aus nur einem Semikolon.
- Typische Beispiele für Anweisungen:

```
int a, b;       // Deklaration von a und b
a = b;          // Zuweisung
summe = b + c;
zaehler++;      // Inkrement-Anweisung
push(27);       // Funktionsaufruf
;               // leere Anweisung
```


Anweisungen und Blöcke

- Ein **Block** (compound statement) ist eine Zusammenfassung von Anweisungen, die in geschweifte Klammern eingeschlossen sind. Die Deklarationen im Block gelten nur innerhalb des Blocks. Ein Block braucht nicht durch ein Semikolon abgeschlossen zu werden.
- Typisches Beispiel für Blöcke:

```
int a=3, b;

if ( a > 0 )
{    // Block, der nur ausgeführt wird, wenn a>0
  a--;
  cout << a;
}
else
{    // Block, der nur ausgeführt wird, wenn a <=0
  a++;
  cout << a;
}
```

Sequenz

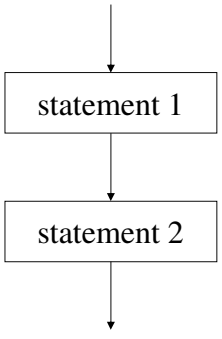
- Die Anweisungen eines Programms werden in der Aufschreibungsreihenfolge durchlaufen.
- Allgemeine Form:
  statement1;
  statement2;
  Zuerst wird statement1 ausgeführt, danach statement2.
- Beispiele sequentielle Abarbeitung

```
int i;
i = 18;
i = i - 6;
cout << i << endl;
```

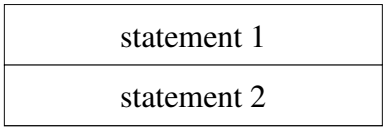
- Durch Flussdiagramme kann der Ablauf eines Programms verdeutlicht werden. Dazu wird jeder Anweisungsform eine Darstellung zugeordnet.

Sequenz

- Ein Flussdiagramm für eine Sequenz aus zwei Anweisungen ist:



- Ein Struktodiagramm für eine Sequenz aus zwei Anweisungen ist:



Bedingte Anweisungen
Selektionsanweisung, Verzweigung

- Soll eine Aktion nur ausgeführt werden, wenn gewisse Bedingungen zutreffen, also Entscheidungen programmiert werden sollen, können Verzweigungen verwendet werden.
- Selektionsanweisungen = Verzweigungen = Entscheidungen = bedingte Anweisungen

- Die bedingte Ausführung einer Anweisungsfolge realisiert man in C durch eine **if**-Anweisung bzw. **if-else**-Anweisung, die folgende Struktur hat:

Bedingte Anweisung
Selektionsanweisung, Verzweigung

Hier steht eine **Bedingung**.

```
if ( expression )
{
  statement1_true
  statement2_true
}
else
{
  statement1_false
  statement2_false
}
```

Wenn diese Bedingung erfüllt ist (d.h. Wert != 0), so wird die in geschweifte Klammern eingeschlossene **Anweisungsfolge** ausgeführt**.**

Ist die Bedingung nicht erfüllt (d.h. Wert = 0), so wird die hier eingeschlossene **Anweisungsfolge** ausgeführt**.**

Selektionsanweisung, Verzweigung

- Beispiele:
  - Berechne das Maximum zweier Zahlen a und b:

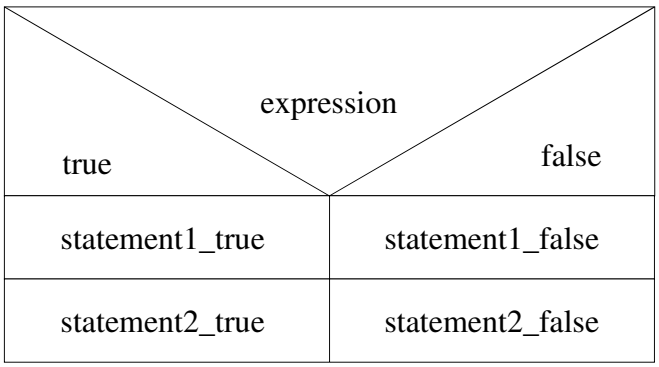
```
if ( a < b ) {
  maximum = b;
}
else {
  maximum = a;
}
```

  - Berechne den Abstand von a und b:

```
if ( a < b ) {
  abstand = b - a;
}
else {
  abstand = a - b;
}
```

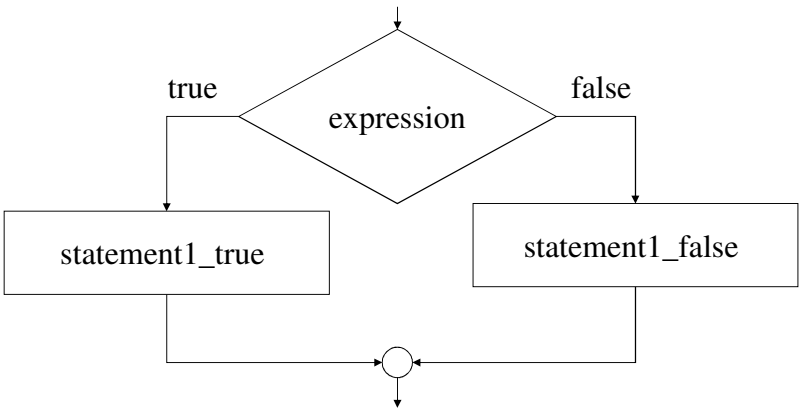

Selektionsanweisung, Verzweigung

- Struktogramm für if-else



Selektionsanweisung, Verzweigung

- Flussdiagramm für if-else



Selektionsanweisung, Verzweigung

- Beispiel - Verzweigung:

```
#include <iostream>
using namespace std;
int  main()
{
  int x;
  cout << "Eingabe Integer: ";
  cin >> x;
  if ( 0 == x % 2 ) {
    cout << x << " ist gerade Zahl";
  }
  else {
    cout << x << " ist ungerade Zahl";
  }
  cout << endl;
  return 0;
}
```

Selektionsanweisung, Verzweigung

- Anweisungen werden in einem Programm nacheinander hingeschrieben. Werden zwei if-Anweisungen nacheinander benötigt, so muss geregelt werden, zu welchem if ein else gehört.
- **Regel:**
  Ein else gehört immer zum **unmittelbar** vorangehenden if; dabei ist die Blockstruktur zu beachten.
  Falls es „unklar“ sein sollte, geschweifte Klammern setzen!
- Beispiele nächste Seite:

Selektionsanweisung, Verzweigung

- Beispiele:

Beispiel 1	Beispiel 2

`if (n > 0 )`
`{`
`  if ( a > b )`
`  {`
`  z = a;`
`  }`
`  else`
`  {`
`  z = b;`
`  }`
`}`
	`if (n > 0 )`
	`{`
	`  if ( a > b )`
	`  {`
	`  z = a;`
	`  }`
	
	`}`
	`else`
	`{`
	`  z = b;`
	`}`

Es fehlen die zugehörigen else-Zweige!!

Selektionsanweisung, Verzweigung

- Mehrfachverzweigungen können mit if-else-Ketten programmiert werden. Bei tiefen Ketten wird häufig auf die sonst vereinbarte Einrückung verzichtet.
- Beispiel:

```
if ( expression_1 ) {
  statement_1
}
else {
  if ( expression_2 ) {
    if ( expression_3 ) {
      statement_2
    }
    else {
      statement_3
    }
  }
  else {
    statement_4
  }
}  // end if ( expression_1 )
```


Mehrfachverzweigung

- **Beispiel Notenberechnung: Mehrfachverzweigung**

```
int main()
{
  int punkte;
  double note;
  cout << "Eingabe Punkte: ";
  cin >> punkte;
  if ( punkte < 40 )
    note = 5.0;
  else if ( punkte <= 46 )
         note = 4.0;
  else if ( punkte <= 63 )
         note = 3.0;
  else if ( punkte <= 80 )
         note = 2.0;
  else
         note = 1.0;
  cout << "Note: " << note << endl;
  return 0;
}
```

Fehlerquellen in Verbindung mit if

- Durch die Verwechslung des Vergleichsoperators (==) mit dem Zuweisungsoperators (=) entstehen Fehler:

```
if ( a = b )
  cout << "a ist gleich b";
```

  Gemeint war:

```
if (a == b)
  cout << "a ist gleich b";
```

- Ein Semikolon zuviel kann bewirken, dass eine Anweisung immer ausgeführt wird:

```
if ( a == b );
  cout << "a ist gleich b";
```

  Das Semikolon schließt die if-Anweisung ab. Der Text wird unabhängig vom erfüllt sein der Bedingung ausgegeben.

Bedingte Bewertung

- Ein Operator, mit dem Entscheidungen in Ausdrücken realisiert werden können ist die "bedingte Bewertung".
- **Allgemeine Form bedingte Bewertung:**
  `cond_expr ? expression_true : expression_false`
  zuerst wird cond_expr ausgewertet; dann wird in Abhängigkeit des Wertes der expression_true oder expression_false das Resultat des Ausdrucks.
- **Anweisung:**

```
if (true == schaltjahr) {
  tageImFebruar = 29;
}
else {
  tageImFebruar = 28;
}
```

- **Ausdruck:**

```
tageImFebruar = schaltjahr ? 29 : 28;
```

- Siehe Beispiel Schaltjahr

Fallunterscheidung

- Fallunterscheidungen können mittels der switch-Anweisung realisiert werden.
- **Allgemeine Form: switch case**

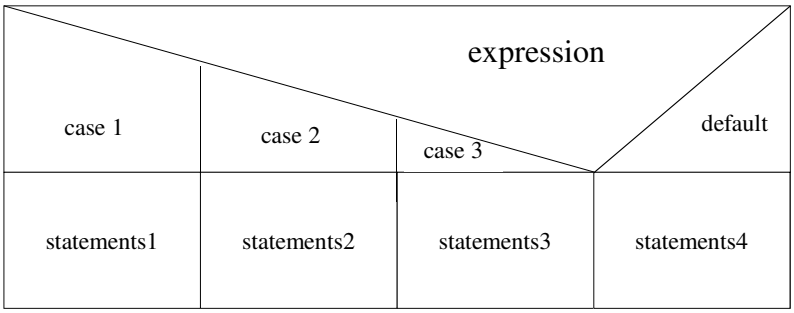
```
switch ( expression )
{
  case const1:
     statements1;
     break;
  case const2:
     statements2;
     break;
  default:
     statements3;
     break;
}
```


Fallunterscheidung

- *expression* wird ausgewertet und zu der Stelle *consti* gesprungen, die mit dem expression-Wert übereinstimmt, bzw. zu *default*, falls es keine Übereinstimmung gibt.
- Der expression-Wert **muss** vom Typ integer oder char sein.
- Bei reinen Fallunterscheidungen ist die letzte Anweisung von *statement* eine break-Anweisung, die das switch-statement beendet.

Fallunterscheidung

- Struktogramm für switch - case



Fallunterscheidung

- Bei reinen Fallunterscheidungen ist die letzte Anweisung von *statement* eine break-Anweisung, die das switch-statement beendet.
- Durch Verzicht auf break wird die Ausführung in die nächsten Case-Anweisungen hinein fortgesetzt, bis entweder ein break oder das Ende der switch-Anweisung erreicht ist.
- Drei Randbedingungen gibt es zu switch Anweisungen
  - Switch kann nur auf Gleichheit testen.
  - Es können keine 2 case Konstanten in demselben switch denselben Wert haben.
  - Wenn in einer switch Anweisung Zeichenkonstanten verwendet werden, werden sie automatisch in Integer verwandelt.
- Siehe Beispiel Römische Zahlen

Wiederholte Befehlsausführung

- Es kann erforderlich sein, in einem Algorithmus eine bestimmte Sequenz von Anweisungen wiederholt zu durchlaufen, bis eine bestimmte Situation eingetreten ist.
- Dies nennen wir Programmschleife oder Schleifen-Anweisungen.
- Unterschieden werden mehrere Arten von Schleifen:
  - Schleifen mit abweisendem Charakter (Prüfung bevor Aktionen ausgeführt werden) (while - Schleife)
  - Schleifen mit nicht abweisendem Charakter (do while - Schleife) und
  - Schleifen, bei denen die Schrittweite vorher feststeht (for - Schleife).
- Versucht man die Anatomie von Schleifen allgemein zu beschreiben, stößt man auf ein immer wiederkehrendes Muster.

Wiederholte Befehlsausführung

- Es gibt eine Reihe von Dingen zu tun, bevor man mit der Durchführung der Schleife beginnen kann.
  Dies nennt man die **Initialisierung** der Schleife.
- Es ist eine Prüfung durchzuführen, ob die Bearbeitung der Schleife abgebrochen oder fortgesetzt werden soll.
  Dies nennt man **Test** auf Fortsetzung der Schleife.
- Bei jedem Schleifendurchlauf sind die eigentlichen Anweisungen durchzuführen.
  Dies nennt man den **Schleifenkörper**.
- Nach Beendigung eines einzelnen Schleifendurchlaufs sind gewisse Operationen durchzuführen, um den nächsten Schleifendurchlauf vorzubereiten.
  Dies nennt man das **Inkrement** der Schleife.

While - Schleife

```
initialisierung
while ( expression )
{
  statement1
  statement2
  statement3

  inkrement
}
```

Die **Initialisierung** wird vor dem erstmaligen Eintritt in die Schleife einmal ausgeführt.

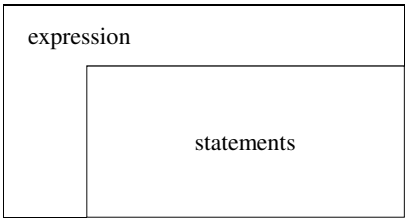
Der **Test** wird vor jedem möglichen Eintritt in den Schleifenkörper ausgewertet. Ergibt sich dabei ein Wert != 0, so wird der Schleifenkörper ausgeführt. Andernfalls wird die Bearbeitung der Schleife abgebrochen.

Der **Schleifenkörper** wird bei jedem Schleifendurchlauf ausgeführt**.**

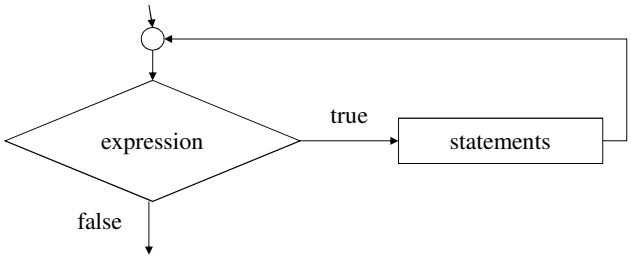
Das **Inkrement** wird – meistens – am Ende des Schleifendurchlaufs ausgeführt**.**

while -Schleife

- Struktogramm für while



- Flussdiagramm für while



while - Schleife

- Beispiel: Zahlen eingeben, aufsummieren und die Summe ausgeben. Schleifenende bei Eingabe von 0.

```
#include <iostream>
using namespace std;
int main ()
{
  int eingabe = -1;
  int summe = 0;
  while ( eingabe != 0 )
  {
    cout << "Integer: ";
    cin >> eingabe;
    summe = summe + eingabe;
  }
  cout << "Summe: " << summe << endl;
  return 0;
}
```


while - Schleife

- **ACHTUNG:**

  Wenn man Schleifen verwendet, können leicht Programme entstehen, die nicht terminieren!

  Der Programmierfehler ist dann, dass man im Schleifen-rumpf nicht dafür sorgt, dass die Schleifenbedingung jemals zu false auswertbar ist.

- Beispiel (Berechnung Quadratzahlen):

while - Schleife

- Beispiel: unendlicheSchleife

```
#include <iostream>
using namespace std;
int main ()
{
  int i;
  cout << "Integer: ";
  cin >> i;

  while ( (i * i) > 0 )
  {
    cout << "i * i = " << i * i << endl;
    i--;
  }
 return 0;
}
```


do while - Schleife

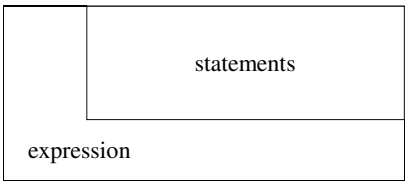
- Bei dieser Schleifenart wird zuerst eine Anweisung ausgeführt und erst danach geprüft, ob sie nochmals ausgeführt werden soll.
- **Allgemeine Form: do while Schleifen**

```
do
    statement
while (expression)
```

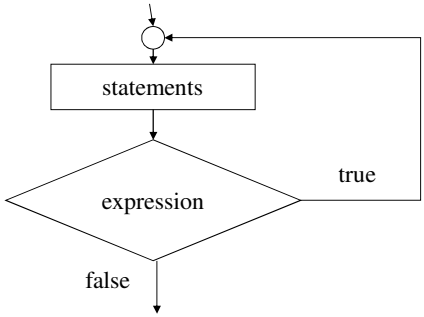
- Zuerst wird *statement* ausgeführt. Dann wird die Bedingung *expression* geprüft; ist sie zu true auswertbar dann wird *statement* wieder ausgeführt.

do while -Schleife

- Struktogramm für do while



- Flussdiagramm für do while



do while - Schleife

- Beispiel: Dialog, um nur "gültige" Werte zu erlauben.

```
#include <iostream>
#include <cmath>
using namespace std;
int main ()
{
  const double Minimum = 3;
  const double Maximum = 27;
  double wert;
  do
  {
    cout << "Bitte Wert zwischen " << Minimum;
    cout << " und " << Maximum << " eingeben: ";
    cin >> wert;
  } while ((wert < Minimum) || (wert > Maximum));
  cout << "Wurzel aus " << wert;
  cout  << " ist " << sqrt(wert) << endl;
 return 0;
}
```

for - Schleife

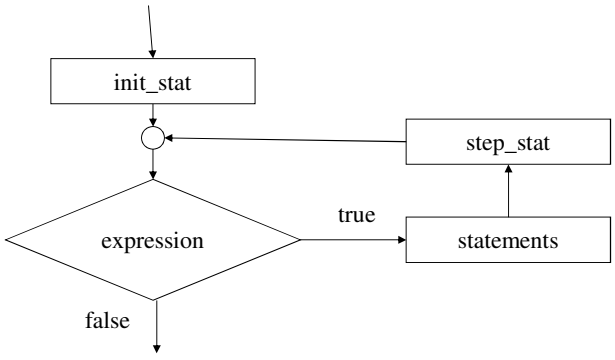
- Steht die Anzahl der Wiederholungen vorher fest oder hat man feste Schrittweiten, so wird häufig die for Schleife eingesetzt.
- Diese Schleifenversion ist die übersichtlichste, da Sie an einer Stelle Startbedingung, Endebedingung und Schrittweite erkennen können.
- **Allgemeine Form: for Schleifen**

```
for ( init_stat; expression; step_stat )
    statement
```

- Zuerst wird *init_stat* ausgeführt; dabei werden i.A. Initialisierungen vorgenommen. Dann wird die Bedingung *expression* geprüft; ist sie zu *true* auswertbar dann wird *statement* wieder ausgeführt danach *step_stat*.
- Das Verhalten dieser Schleifenart läßt sich durch folgendes Flußdiagramm veranschaulichen:

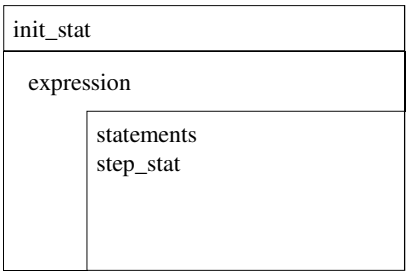
for - Schleife

- Flussdiagramm for - Schleife



for - Schleife

- Struktogramm for-Schleife



for - Schleife

- Beispiel: ASCII-Werte ausgeben.

```
#include <iostream>
#include <cmath>
using namespace std;
int main ()
{
  const int untereGrenze = 65;
  const int obereGrenze  = 70;
  int i;

  for (i = untereGrenze; i <= obereGrenze; i++)
  {
    cout << i << " " << char(i) << endl;
  }
 return 0;
}
```

Kombinierte Schleifen

- Die bisherigen Kontrollstrukturen lassen sich auch kombinieren, so ist z.B. auch eine Schleife in einer Schleife möglich.
- Beispiel (von Aufgabe über Struktogramm zum Programm)

  Aufgabe:
  Von dem Zahlenbereich 1 bis 9 soll für jede Zahl die Fakultät berechnet und ausgegeben werden

  Verfahren:
  Ausgabe von n! für n zwischen 1 und 9
  wobei n! := 1*2*3* … *n.

Kombinierte Schleifen

- Struktogramm Fakultät

// Integer Konstanten festlegen
untereGrenze = 1, obereGrenze = 9

// Integer Variablen deklarieren, definieren und initialisieren
i = untereGrenze, j, (unsigned long) fak

solange (i <= obereGrenze)
Setze fak = 1 (und j = 1)
for (. . ; j <= i; . .)
Berechne: fak = fak * j (Erhöhe j um 1)
Ausgabe: "Fakultaet fak ist " Wert von fak
Erhöhe i um 1

Kombinierte Schleifen

- Programmcode Fakultaet

```
#include <iostream>
using namespace std;
int  main ()
{
  const int untereGrenze = 1;
  const int obereGrenze  = 9;
  int i = untereGrenze;
  int j;
  unsigned long int fak;

  while ( obereGrenze >= i )
  {
     fak = 1;
     for (j = 1; j <= i; j++)
     {
       fak = fak * j;
     }
     cout << "Fakultaet von ("<< i << ") = ";
     cout << fak << endl;
     i++;
  }
 return 0;
}
```


Zusammenfassung Schleifen

- Eine *do while* Schleife kann stets in eine *while* Schleife umgeformt werden, und umgekehrt.
- Eine *for* Schleife entspricht einer *while* Schleife; es handelt sich eigentlich nur um eine Umformulierung.

- In C++ können innerhalb der Selektions- und der Iterationsanweisung lokale Variablen deklariert werden. In C geht das nicht.
- Achtung:
  Die Variable kann dann nicht außerhalb des Schleifenrumpfs verwendet werden, da sie hier innerhalb des Blockes lokal definiert ist! Ältere Compiler erlauben dies zwar, Sie sollten dies dann trotzdem nicht verwenden, da diese Programme nicht portierbar sind, wenn in der Zielumgebung neue Compiler verwendet werden.

while - Schleife

- Beispiel: Erneute Wiederholung der Eingabe

```
#include <iostream>
using namespace std;
int main ()
{
  int eingabe = 1;
  while ( 1 == eingabe )
  {
    cout << "Schleifendurchlauf" << endl;

    cout << "Wollen Sie weitermachen";
    cout << " (1 = ja, jede andere Zahl = nein)? ";
    cin >> eingabe;
  }
  return 0;
}
```


Kommaoperator

- Der Kommaoperator wird meist in *for* Schleifen verwendet, um die Bewertungsreihenfolge zu festzulegen und den Resultatstyp zu bestimmen.
- **Allgemeine Form: Kommaoperator**
  `expr1, expr2`
- Zuerst wird *expr1*, dann *expr2* bewertet.
  Das Resultat hat den Typ von *expr2*.

Kommaoperator

- Programmcode Komma-Operator

```
#include <iostream>
using namespace std;
int main ()
{
  int i, sum;            // KEIN Kommaoperator

  // Kommaoperator
  for ( i = 1, sum = 0; i <= 10; sum += i, i++ )
        ;

  cout << "Summe von 1 bis 10 ist " << sum << endl;
  return 0;
}
```


Sprünge

- Sprünge zu bestimmten Stellen im Programm sind ein Relikt aus der systemnahen Programmierung und haben in Hochsprachen in unterschiedlicher Form Einzug gehalten:
  - kontrolliertes Verlassen von Schleifen (manchmal sinnvoll einsetzbar!)
  - Sprünge zu definierbaren Marken (meist schlechter Programmierstil!)

Sprünge - break

- *break* haben wir bereits zum Verlassen von *switch*-Alternativen gesehen.
- *break* in einer Schleife bewirkt, dass die Schleife verlassen wird.

- Beispiel:

```
while ( expression2 )
{
  statement1
  if ( expression1 )
    break;
  statement2
}

statement3
```

Im Praktikum sind *breaks* zum Verlassen einer Schleife verboten!

Sprünge - continue

- Durch *continue* wird in Schleifen zur erneuten Überprüfung der Bedingung gesprungen.
- *continue* erzwingt die Durchführung der nächsten Schleifeniteration.
- In der for-Schleife wird der nächste bedingte Test und der Inkrementteil der Schleife angesprungen.
- In der while- und der do-while Schleife wird beim Bedingungstest weitergemacht.

Sprünge - continue

- Beispiel:

```
while ( expression2 )
{
  statement1
  if ( expression1 )
    continue;
  statement2
}
statement3
```

Im Praktikum sind *continues* zum Wiederholen einer Schleife verboten!

Sprünge - goto

- Mit der Anweisung *goto* kann zu einer beliebigen Marke (label) gesprungen werden. Eine Marke muss dabei vorher definiert worden sein.

- **Bemerkung:**

  Beim Programmieren kann immer auf „**goto**“ verzichtet werden. Es gibt nur einige wenige Ausnahmen (im Bereich systemnahe Programmierung und Programmierung von Echtzeitanwendungen), bei denen *goto* sinnvoll einsetzbar ist.

  Normalerweise sollte stets auf *gotos* verzichtet werden, da die Programme fehleranfällig und schwer wartbar bzw. erweiterbar sind!

  Im Praktikum sind *gotos* verboten!

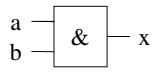
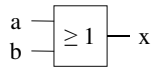
Aussagenlogik

- Unter Aussage versteht man einen Satz, der entweder wahr oder falsch ist.
- Beispiele für Aussagen:
  - Darmstadt liegt in Hessen.
  - Darmstadt hat 120.000 Einwohner.
- Beispiele für "Keine"-Aussagen:
  - Guten Tag, meine Damen und Herren,
  - Wie spät ist es?
- Bei der Programmierung interessiert man sich nur mit präzise formulierten Aussagen wie z.B. wert < 10 oder a + b > c.
- Komplexere Aussagen:
  - Darmstadt liegt in Hessen **und** hat 120.000 Einwohner.
- Das ist richtig, wenn beide Teilaussagen richtig sind.
- Das Wort "und", mit dem die beiden Teilaussagen verbunden sind, bezeichnet man als Junktor.
- Im Folgenden interessiert nicht der Inhalt der Aussagen, sondern eher dass allgemeine Verständnis für die Aussagenlogik bzw. für die Boole'sche Algebra.

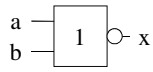
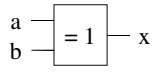
Aussagenlogik - Festlegungen

- Wahrheitswerte: "wahr" bzw. "falsch"
- Symbole: "1" bzw. "0"
- Aussagen (ohne Inhalt): a, b, c
- "a ist wahr " heißt dann "a = 1"
- Umgekehrt: "a = 0" heißt "a ist falsch"
- Die Abhängigkeit einer Gesamtaussage von ihren Teil-aussagen drückt man mathematisch korrekt durch eine Funktion f(a, b, c) aus.
- Man spricht auch von **Schaltalgebra**.
- Man nennt dies eine **boolesche Funktion**.
- Die Funktionstabelle einer booleschen Funktion bezeichnet man als **Wahrheitstafel**.
- Junktoren heißen **logische Operatoren**.

Binäre (Boolesche) Verknüpfungen

Schaltzeichen Symbol	Wahrheits-tabelle	Schaltfunktion, Benennung, Gleichung	Beschreibung Gleichung in C-Notation
	a b x 0 0 0 0 1 0 1 0 0 1 1 1	UND-Funktion (Konjunktion) $x = a \wedge b$	Ausgang nur dann 1-Zustand, wenn sich beide Eingänge im 1-Zustand befinden. x = a && b;
	a b x 0 0 0 0 1 1 1 0 1 1 1 1	ODER-Funktion (Disjunktion) $x = a \vee b$	Ausgang nur dann 1-Zustand, wenn sich mindestens ein Eingang im 1-Zustand befindet. x = a || b;

Binäre (Boolesche) Verknüpfungen

Schaltzeichen Symbol	Wahrheits-tabelle	Schaltfunktion, Benennung, Gleichung	Beschreibung Gleichung in C-Notation
	a x 0 1 1 0	NICHT-Funktion (Negation) $x = \overline{a}$	Ausgang nur dann 1-Zustand, wenn sich der Eingang im 0-Zustand befindet. x = !a;
	a b x 0 0 0 0 1 1 1 0 1 1 1 0	Antivalenz-Funktion (Exklusiv-ODER) $x = (a \wedge \overline{b}) \vee (\overline{a} \wedge b)$ $x = (a \vee b) \wedge \overline{(a \wedge b)}$	Ausgang nur dann 1-Zustand, wenn sich beide Eingänge in unterschiedlichen Zuständen befinden. x = (a || b) && !(a && b);

Logische Operatoren in C

boolesche Algebra	C-Programm	Bedeutung
$\overline{\quad}$	!	logische Negation
$\wedge$	&&	logisches UND
$\vee$	||	logisches ODER

- In der Booleschen Algebra wurde gezeigt, dass diese drei Operatoren ausreichend sind, um alle booleschen Funktionen zu berechnen.
- In C gibt es keinen Wert bool, der nur die Werte wahr oder falsch annehmen kann.
- Der Wert 0 wird als "falsch" bewertet.
- Alle Werte != 0 wird als "wahr" bewertet.
- Es gilt auch hier: ! wird vor && und && vor || ausgewertet.
- In C++ gibt es den Wert bool, der nur die Werte false und true annehmen kann. Er hat die Größe 1 Byte (in älteren Version vom Typ int = 2| 4 Byte).

Logische Operatoren in C

- Es ergeben sich die folgenden Wahrheitstabellen:

a	!a
0	1
1	0

a	b	a && b
0	0	0
0	1	0
1	0	0
1	1	1

a	b	a || b
0	0	0
0	1	1
1	0	1
1	1	1