

Kontrollstrukturen

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.

Inhalt

1. Anweisungen und Blöcke.....	3
1.1. Sequenz	5
2. Entscheidungen	7
2.1. if - else	7
2.2. else – if	12
2.2.1. Fehlerquelle in Verbindung mit if	14
2.3. Bedingte Bewertung.....	15

2.4.	switch - case	18
3.	Schleifen	21
3.1.	while Schleifen	21
3.2.	do while Schleifen	25
3.3.	for Schleifen	32
3.3.1.	Äquivalenz von for und while.....	35
3.4.	Kommaoperator	36
3.5.	Sprünge	37
3.5.1.	break	38
3.5.2.	continue	40
3.5.3.	goto	43
4.	Ausnahmebehandlung	46
4.1.	Grundidee der Ausnahmebehandlung	47
4.2.	Ablauf im Fehlerfall	48
4.3.	Allgemeine Form.....	48
4.4.	Ausnahmebehandlung unterschiedlichen Typs.....	51

1. Anweisungen und Blöcke

Einfache **Anweisungen** (statements) werden mit Semikolon **abgeschlossen**.

Die "leere" Anweisung besteht aus nur einem Semikolon.

Typische Beispiele:

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

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.

Typische Beispiele:

```
int a=3, b;
{ int x;           // blocklokale Deklaration
  x = 2;
  cout >> a*x;
}
if ( a>0 ) {      // Block, der nur ausgeführt wird, wenn a>0
  a++;
  cout << a;
}
```

1.1. Sequenz

Die Anweisungen eines Programms werden in Aufschreibungsreihenfolge durchlaufen.

Allgemeine Form:

```
statement1;  
statement2;
```

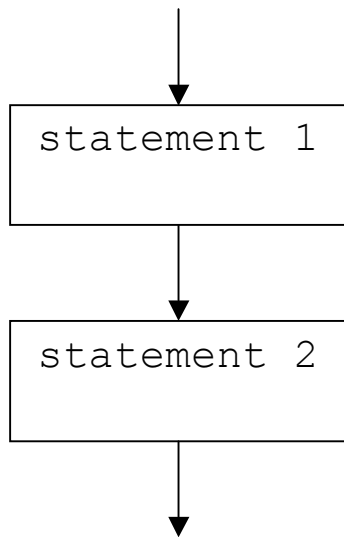
Zuerst wird statement1 ausgeführt, danach statement2.

Beispiele:

```
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.

Ein Flussdiagramm für eine Sequenz aus zwei Anweisungen ist:



2. Entscheidungen

Soll eine Aktion nur ausgeführt werden, wenn gewisse Bedingungen zutreffen, also Entscheidungen programmiert werden sollen, können Verzweigungen verwendet werden.

2.1.if - else

Allgemeine Form:

```
if (expression)
    statement
```

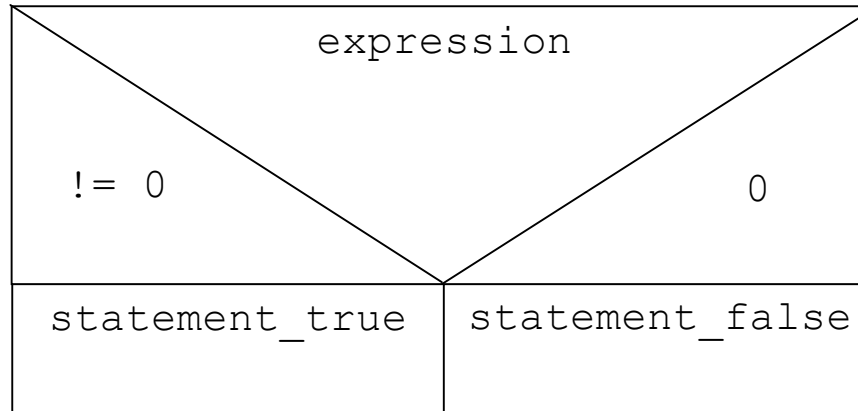
`expression` wird ausgewertet. Ist der Wert `true`, wird `statement` ausgeführt.

```
if (expression)
    statement_true
else
    statement_false
```

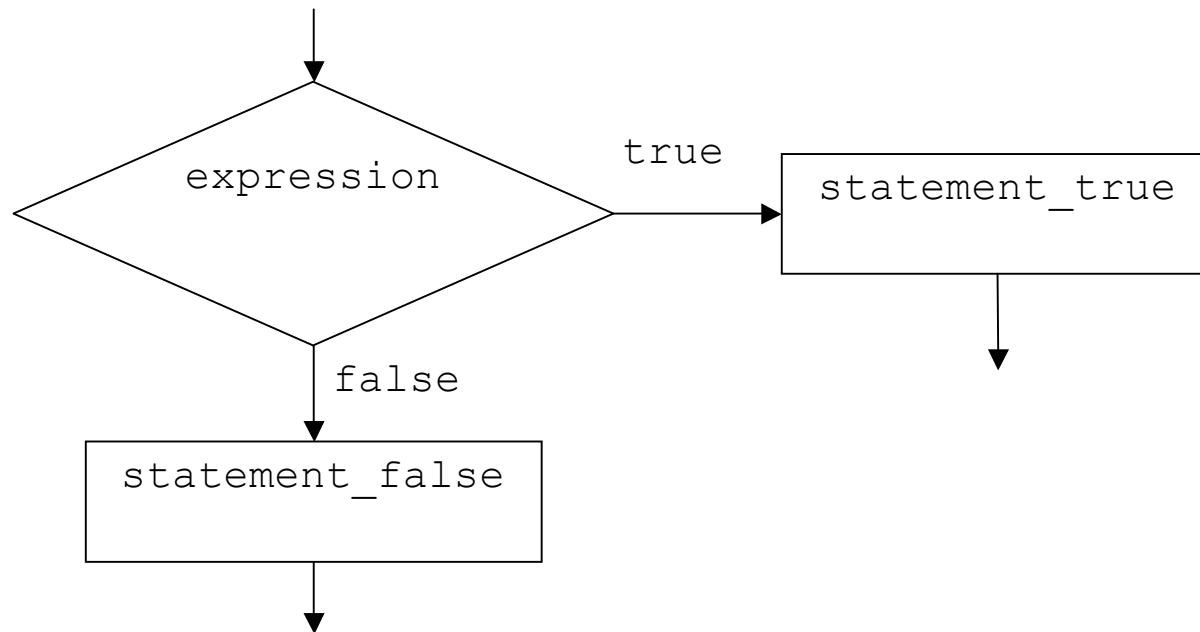
`expression` wird ausgewertet. Ist der Wert `true`, wird `statement_true` ausgeführt, ansonsten `statement_false`.

Dieses Verhalten kann durch ein **Struktogramm** veranschaulicht werden:

```
if (expression)
    statement_true
else
    statement_false
```



Ein Flussdiagramm für diese Entscheidung ist:



Beispiel:

```
$ cat verzweigung.cpp
#include <iostream.h>
main() {
    int x;
    cout << "Eingabe Interger: ";
    cin >> x;
    if ( x%2 == 0 )
        cout << x << " ist gerade Zahl";
    else
        cout << x << " ist ungerade Zahl";
    cout << endl;
}
$
```

Das gleiche Programm in einer eher schwer lesbaren Form (besser **nicht** so programmieren):

```
$ cat verzweigungUnleserlich.cpp
#include <iostream.h>
main() {
    int x;
    cout << "Eingabe Interger: ";
    cin >> x;
    if ( !(x%2) )
        cout << x << " ist gerade Zahl";
    else
        cout << x << " ist ungerade Zahl";
    cout << endl;
}
$
```

D.h.

```
if (expression)
```

ist gleichbedeutend mit

```
if (expression != 0)
```

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.

Beispiel:

richtig	falsch	richtig
<pre>if (n>0) { if (a>b) z = a; else z = b; }</pre>	<pre>if (n>0) if (a>b) z = a; else z = b;</pre>	<pre>if (n>0) { if (a>b) z = a; } else z = b;</pre>

Wie sehen für die richtigen Darstellungen die **Strukto-**
gramme aus?

2.2.else – if

Mehrfachverzeigungen können mit if-else-Ketten programmiert werden.

Allgemeine Form:

```
if ( expression_1 )  
    statement_1  
else if ( expression_2 )  
    statement_2  
else if ( expression_3 )  
    statement_3  
else  
    statement_4
```

Sollte **immer** verwendet werden
(ev. Fehlermeldung)!

Beispiel (Notenberechnung):

```
$ cat mehrfachverzweigung.cpp
#include <iostream.h>
main() {
    int punkte;
    float note;
    cout << "Eingabe Punkte: ";
    cin >> punkte;

    if ( punkte <= 30 )
        note = 5.0;
    else if ( punkte <= 50 )
        note = 4.0;
    else if ( punkte <= 65 )
        note = 3.0;
    else if ( punkte <= 80 )
        note = 2.0;
    else
        note = 1.0;

    cout << "Note: " << note << endl;
}
$
$ mehrfachverzweigung
Eingabe Punkte: 52
Note: 3
$
```

2.2.1. Fehlerquelle in Verbindung mit if

Durch die Verwechslung des Vergleichsoperators mit dem Zuweisungsoperators entstehen Fehler:

Zuweisung: a ist nun immer gleich b
nur wenn b ungleich 0 ist, wird Text ausgegeben

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

Gemeint war:

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

Vergleich

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

Das Semikolon schließt die if-Anweisung ab.

Der Text wird unabhängig vom erfüllt sein der Bedingung ausgegeben.

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

2.3. Bedingte Bewertung

Ein Operator, mit dem Entscheidungen in Ausdrücken realisiert werden können ist die "bedingte Bewertung".

Allgemeine Form:

```
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.

Beispiel:

Anweisung

Ausdruck

```
if (schaltjahr)
    tageImFebruar = 29;
else
    tageImFebruar = 28;
```

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

Beispiel (Ermittlung Schaltjahr):

Ein Jahr ist ein Schaltjahr, wenn es ohne Rest durch 4 teilbar ist. Alle 100 Jahre fällt das Schaltjahr aus, es sei denn, die Jahreszahl ist durch 400 teilbar.

```

$ cat schaltjahr.cpp
#include <iostream.h>
main() {
    int jahr;
    bool schaltjahr;
    cout << "Eingabe Jahreszahl: ";
    cin >> jahr;

    if (jahr%4 == 0) {
        if (jahr%100 == 0) {
            if (jahr%400 == 0) {
                schaltjahr = true;
            } else
                schaltjahr = false;
        } else
            schaltjahr = true;
    } else
        schaltjahr = false;

    cout << jahr << " ist "
         << (schaltjahr ? "ein" : "kein") << " Schaltjahr." << endl;
}
$

```

```

$ schaltjahr
Eingabe Jahreszahl: 1996
1996 ist ein Schaltjahr.
$ schaltjahr
Eingabe Jahreszahl: 2000
2000 ist ein Schaltjahr.
$ schaltjahr
Eingabe Jahreszahl: 1900

```

2.4. switch - case

Fallunterscheidungen können mittels der switch-Anweisung realisiert werden.

Allgemeine Form:

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

expression wird ausgewertet und zu der Stelle *consti* gesprungen, die mit dem *expression*-Wert übereinstimmt, bzw. zu *default*, wenn es keine Übereinstimmung gibt.

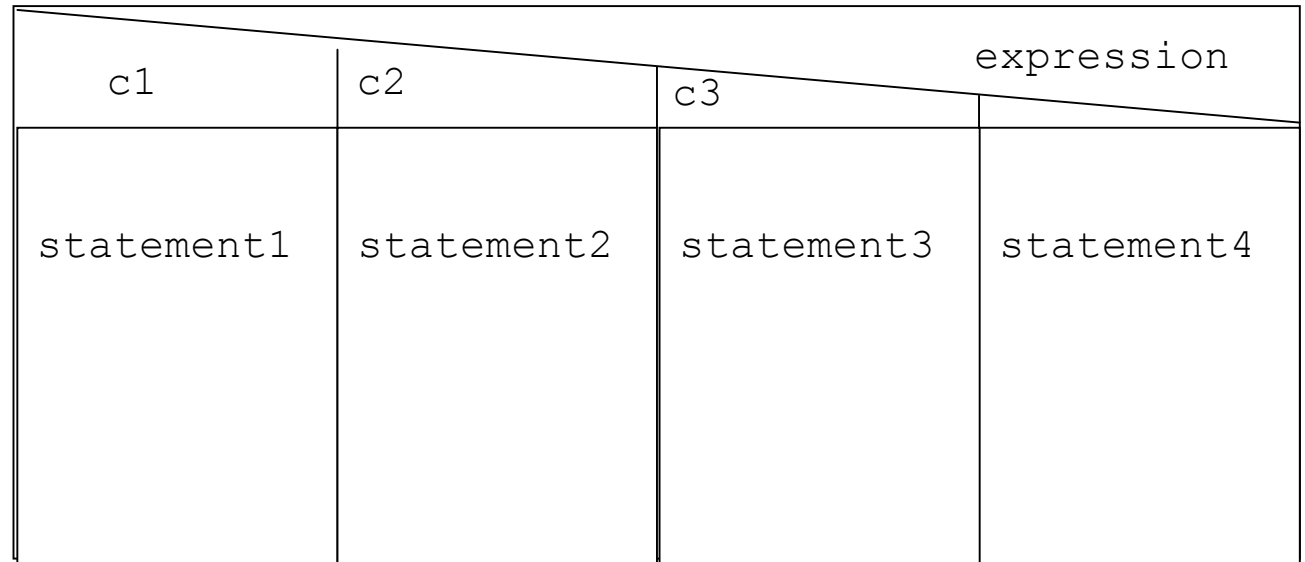
Bei reinen Fallunterscheidungen ist die letzte Anweisung von *statement* eine *break*-Anweisung, die bewirkt, dass das *switch*-Statement terminiert.

Dieses Verhalten kann durch ein Struktogramm veranschaulicht werden:

```

switch (expression) {
  case const1: statement1;
               break;
  case const2: statement2;
               break;
  case const3: statement3;
               break;
  default:    statement4;
}

```



Beispiel (römische Ziffern):

```
$ cat switch.cpp
#include <iostream.h>
int main() {
    int a;
    char c;
    cout << "Zeichen: ";
    cin >> c;

    switch(c) {
        case 'I'      : a=1;      break;
        case 'V'      : a=5;      break;
        case 'X'      : a=10;     break;
        case 'L'      : a=50;     break;
        case 'C'      : a=100;    break;
        case 'D'      : a=500;    break;
        case 'M'      : a=1000;   break;
        default       : a=0;
    }
    if (a > 0)
        cout << a;
    else
        cout <<"keine römische Ziffer!";
    cout << endl;
}
$
```

```
$ switch
Zeichen: M
1000
$
```

3. Schleifen

Sollen Aufgaben wiederholt ausgeführt werden, sind Schleifen-Anweisungen in C++ zu verwenden. Unterschieden werden mehrere Arten von Schleifen:

- Schleifen mit abweisendem Charakter (Prüfung bevor Aktionen ausgeführt werden),
- Schleifen mit nicht abweisendem Charakter und
- Schleifen, bei denen die Schrittweite vorher feststeht.

3.1.while Schleifen

Eine Schleife mit abweisendem Charakter ist die while Schleife.

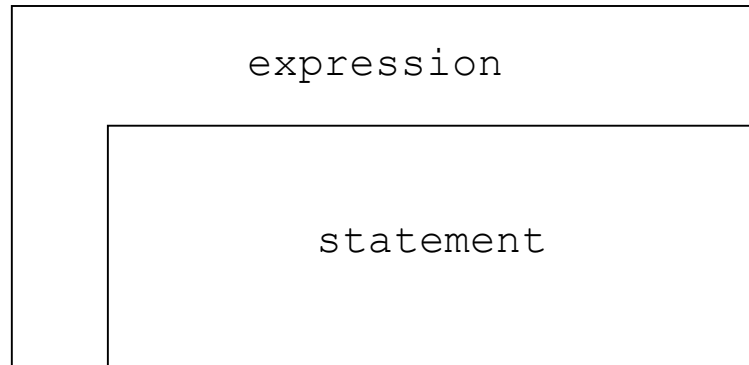
Allgemeine Form:

```
while ( expression )  
    statement
```

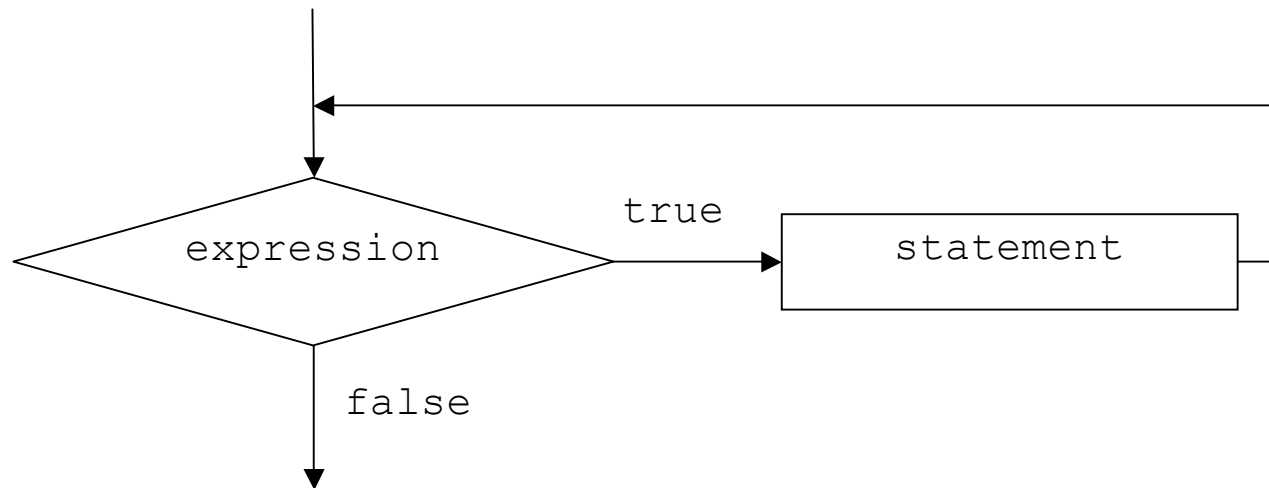
Zuerst wird die Bedingung `expression` geprüft; ist sie zu `true` auswertbar, wird `statement` ausgeführt. Dann wird `expression` wieder geprüft.

Also wird `statement` solange ausgeführt, bis `expression` `false` ist.

Das Struktogramm der Schleife:



Das Verhalten einer Schleife lässt sich durch ein Flussdiagramm veranschaulichen:



Beispiel

Aufgabe: Addieren einer einzulesende Folge von Zahlen mit Endemarke 0.

```
$ cat additionVonZahlenfolge.cpp
#include <iostream.h>
int main() {
    int eingabe=-1;
    int summe=0;

    while (eingabe != 0) {
        cout << "Integer: ";
        cin >> eingabe;
        summe = summe + eingabe;
    }

    cout <<"Summe:" << summe << endl;
}
$
```

```
$ additionVonZahlenfolge.exe
Integer: 1
Integer: 2
Integer: 3
Integer: 4
Integer: 5
Integer: 0
Summe:15
$
```

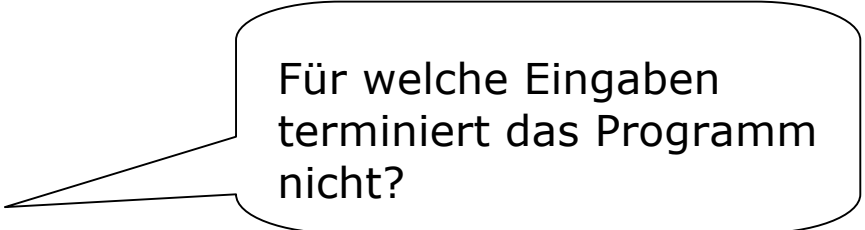
Achtung:

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

Der Programmierfehler ist dann, dass man im Schleifenrumpf nicht dafür sorgt, dass die Schleifenbedingung jemals zu `false` auswertbar ist.

Beispiel (Berechnung Quadratzahlen):

```
$ cat unendlichschleife.cpp
#include <iostream.h>
main() {
    int i;
    cout << "Integer eingeben: ";
    cin >> i;
    while (i*i > 0) {
        cout << "i*i=" << i*i << endl;
        i--;
    }
}
$
```



Für welche Eingaben terminiert das Programm nicht?

Hörsaalübung:

Erweitern Sie das o.a. Programm, so dass es immer terminiert.

3.2.do while Schleifen

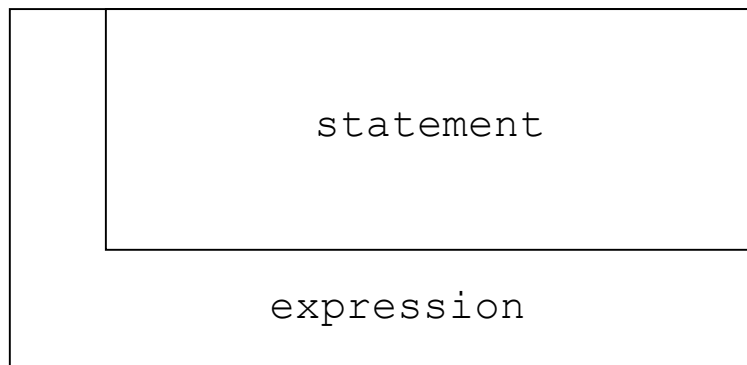
Bei dieser Schleifenart wird zuerst eine Anweisung ausgeführt und erst danach geprüft, ob sie nochmals ausgeführt werden soll.

Allgemeine Form:

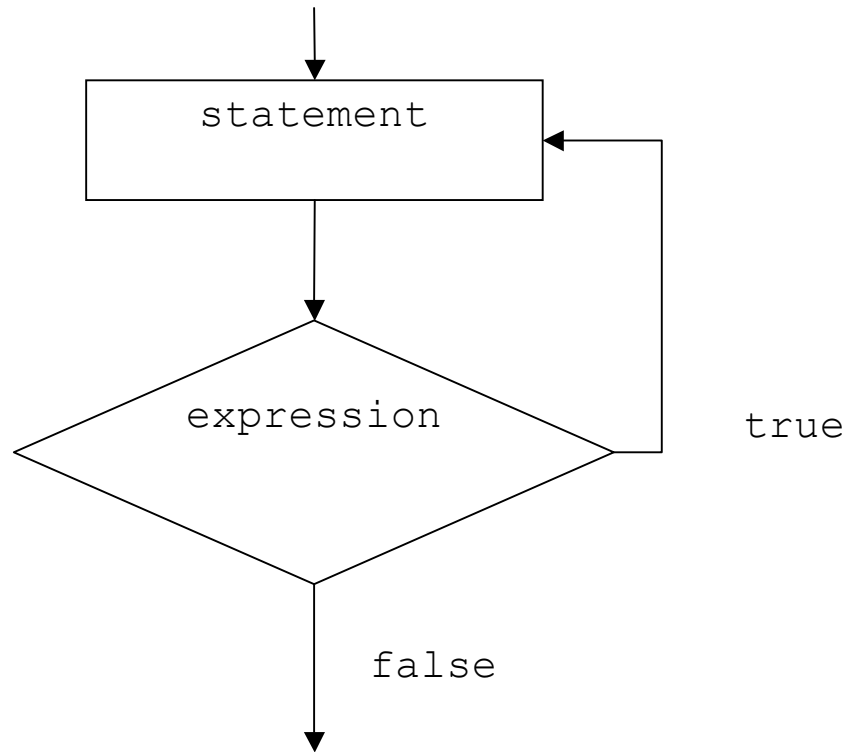
```
do
    statement
while (expression)
```

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

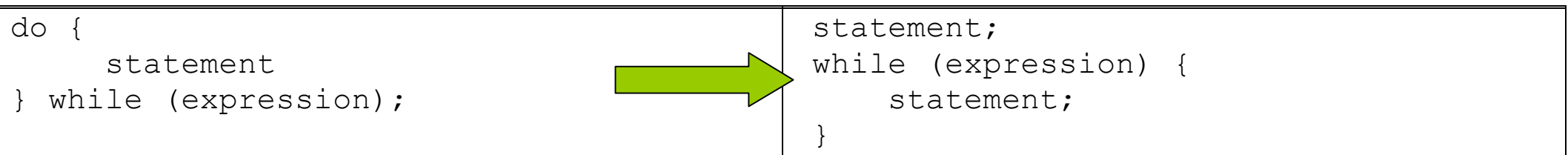
Das Struktogramm der Schleife:



Das Verhalten dieser Schleifenart lässt sich durch folgendes Flussdiagramm veranschaulichen:



Eine do while Schleife kann stets in eine while Schleife umgeformt werden:



Beispiel (Dialog, um nur Werte innerhalb eines gewünschten Bereichs zu erlauben):

```
$ cat doWhile.cpp
#include <iostream.h>
main() {
    const double Minimum = 3;
    const double Maximum = 27;
    double Wert;
    do { // Dialog
        cout << "Bitte geben Sie einen Wert im Bereich "
             << Minimum << ".." << Maximum << " ein: ";
        cin >> Wert;
    } while (Wert < Minimum || Wert > Maximum);

    // Berechnung mit Wert im gültigen Bereich
    cout << "Wurzel von " << Wert << " ist " << sqrt(Wert) << endl;
}
$
```

```
$ doWhile
Bitte geben Sie einen Wert im Bereich 3..27 ein: 1
Bitte geben Sie einen Wert im Bereich 3..27 ein: 3
Wurzel von 3 ist 1.73205
$
```

Die bisherigen Kontrollstrukturen lassen sich auch kombinieren, so ist z.B. auch eine Schleife in einer Schleife möglich.

Beispiel (von Aufgabe über Flussdiagramm zum Programm)

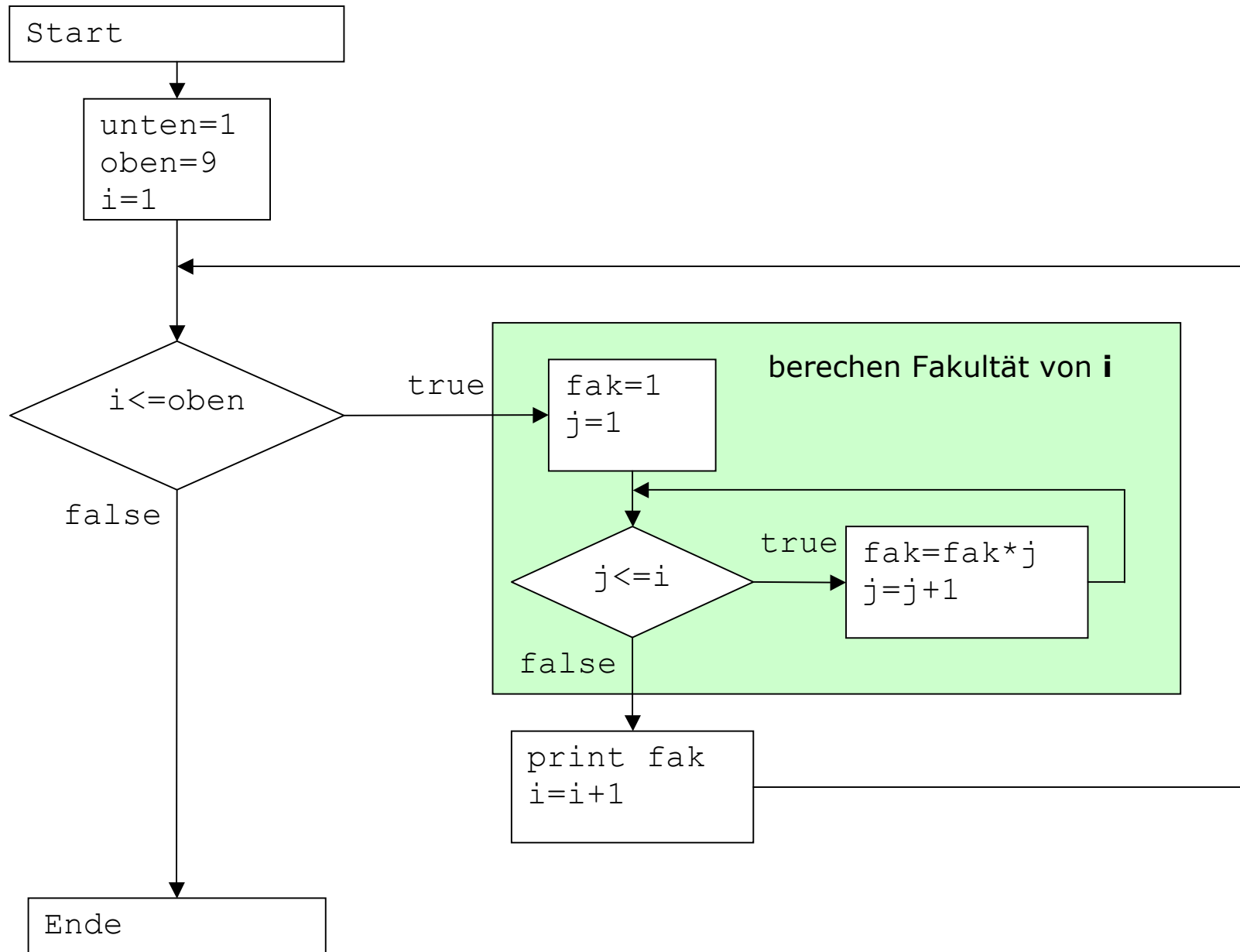
Aufgabe:

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 $n! := 1*2*3* \dots *n$.

Flußgramm:



Programm:

```
$ cat fakultaet.cpp
#include <iostream.h>
main() {
    const int untereGrenze=1;
    const int obereGrenze=9;
    int i=untereGrenze;

    while (i <= obereGrenze) { // für alle Zahlen im Bereich
        unsigned long fak = 1;
        int j = 1;
        while (j <= i) { // berechne Fakultät von i
            fak = fak*j;
            j++;
        }
        cout << "fak(" // Ausgabe Fakultät von i
            << i << ") = "
            << fak << endl;
        i++; // nächste Zahl im Bereich
    }
}
$
```

Hörsaalübung

Entwickeln Sie ein Programm (`maxMinVonFloateingabe.cpp`), das eine Folge von float-Werten zwischen 10 und 1000 einliest und das Maximum und das Minimum ermittelt und ausgibt. Abbruchbedingung ist, dass ein negativer Wert eingegeben wird.

Vorgehen:

1. Flussdiagramm durch Sie
2. C++ Kode von Ihnen
3. Vorführung an der Tafel durch Sie
4. Test und Bewertung durch mich

3.3.for Schleifen

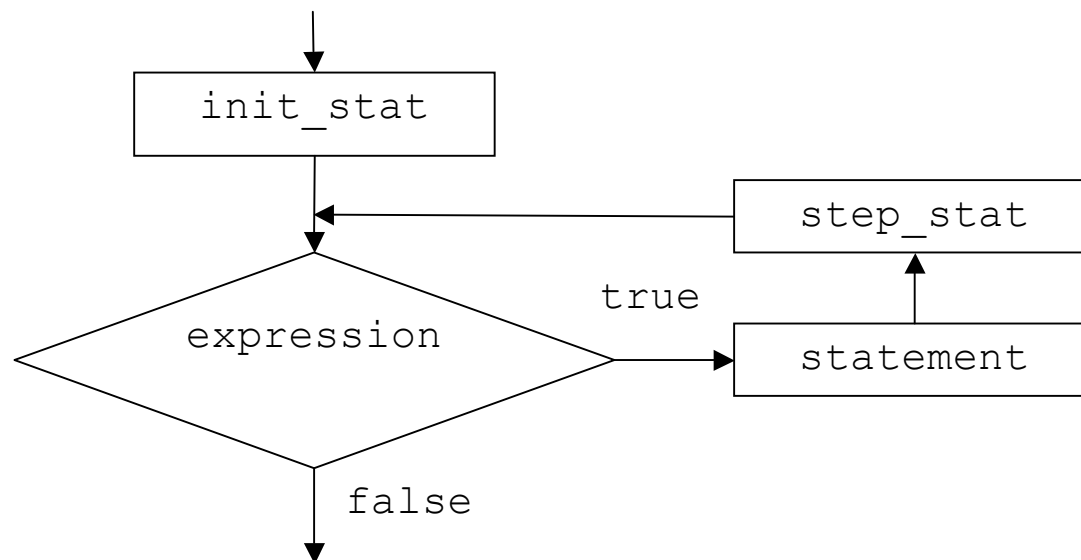
Steht die Anzahl der Wiederholungen vorher fest oder hat man feste Schrittweiten, so wird häufig die for Schleife eingesetzt.

Allgemeine Form:

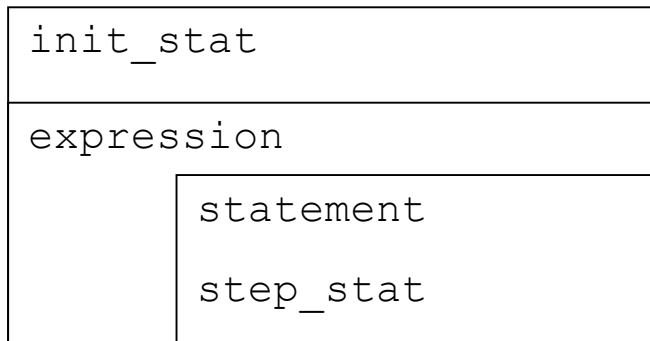
```
for (init_stat; expression; step_stat)
    statement
```

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:



Das Struktogramm dazu:



Beispiel (ASCII Tabelle ausgeben):

```
$ cat ascii.cpp
#include <iostream.h>
main() {
    const int untereGrenze=65;
    const int obereGrenze=70;

    for (int i = untereGrenze; i <= obereGrenze; i++)
        cout << i << " " << char(i) << endl;
}
$
```

```
$ ascii
65 A
66 B
67 C
68 D
69 E
70 F
$
```

Achtung:

Die Variable `i` kann nicht außerhalb des Schleifenrumpfs verwendet werden! 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!


```
$ cat ascii.cpp
#include <iostream.h>
main() {
    const int untereGrenze=65;
    const int obereGrenze=70;

    for (int i = untereGrenze; i <= obereGrenze; i++)
        cout << i << " " << char(i) << endl;

    cout << i << " " << endl;
}
$ make ascii
g++      ascii.cpp      -o ascii
ascii.cpp: In function `int main()':
ascii.cpp:9: name lookup of `i' changed for new ANSI `for' scoping
ascii.cpp:6:   using obsolete binding at `i'
make: *** [ascii] Error 1
$
```

3.3.1. Äquivalenz von for und while

Eine for Schleife entspricht einer while Schleife; es handelt sich eigentlich nur um eine Umformulierung, solange nicht `continue` (-> nächster Abschnitt) vorkommt.

<pre>for (init_stat; expression; step_stat) statement</pre>		<pre>init_stat; while (expression) { statement; step_stat; }</pre>
---	--	--

3.4. Kommaoperator

Der Kommaoperator wird meist in for Schleifen verwendet, um die Bewertungsreihenfolge festzulegen und den Resultatstyp zu bestimmen.

Allgemeine Form:

```
expr1, expr2
```

Zuerst wird `expr1`, dann `expr2` bewertet. Das Resultat hat den Typ von `expr2`.

Beispiel (Summe von 1 bis 10):

```
#include <iostream.h>
main() {
    int i, sum;

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

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

kein Kommaoperator

Kommaoperator

Kommaoperator

Welche Ausgabe hat das folgende Program:

```
#include <iostream>
using namespace std;
int main() {
    int x=1, y=2;
    cout << (x,y) << endl;
}
```

3.5. 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 (oft sinnvoll einsetzbar!)
- Sprünge zu definierbaren Marken (meist schlechter Programmierstil!)

3.5.1. break

`break` haben wir bereits zum Verlassen von `switch`-Alternativen gesehen.

`break` in einer Schleife bewirkt, dass die Schleife verlassen wird.

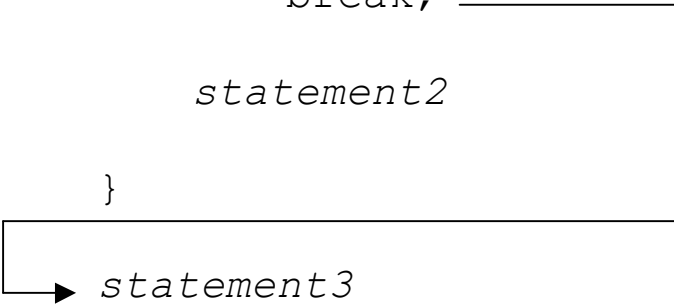
Allgemeine Form:

```
break
```

Die umgebene Schleife wird verlassen.

Verdeutlichung:

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



→ *statement3*

3.5.2. continue

Durch `continue` wird in Schleifen zur erneuten Überprüfung der Bedingung gesprungen.

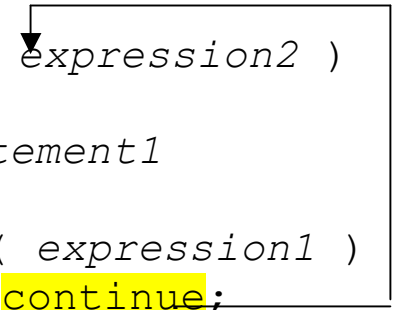
Allgemeine Form:

```
continue
```

Die Bedingung der Schleife wird erneut geprüft.

Verdeutlichung:

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



Beispiel (Menü, das nur gezielt verlassen werden kann):

```

$ cat menue.cpp
#include <iostream.h>
#include <stdlib.h>
main() {
    char auswahl, dummy;
    while (true) {
        system("clear"); // clear screen
        cout << "u      Uhrzeit" << endl;
        cout << "x      Beenden" << endl;
        cout << "Wählen Sie: ";
        cin >> auswahl; // Lesen Benutzereingabe
        if (auswahl == 'u') { // Menuepunkt Uhrzeit
            system("date");
            cout << "Weiter mit bel. Zeichen und RETURN ";
            cin >> dummy;
            continue;
        }
        if (auswahl == 'x') // Menuepunkt Beenden
            break;
        cout << "Falsche Ausgabe, bitte wiederholen" << endl;
        cout << "Weiter mit bel. Zeichen und RETURN ";
        cin >> dummy;
    }
    cout << "Auf Wiedersehen!" << endl;
}
$

```

3.5.3. goto

Mit der Anweisung goto kann zu einer beliebigen Marke gesprungen werden.

Eine Marke muss dabei vorher definiert worden sein.

Allgemeine Form der Markendefinition:

```
name:
```

```
name ist der Name einer Marke.
```

Allgemeine Form der Sprungs:

```
goto name
```

```
Das Programm verzweigt zum Sprungziel, das durch name definiert ist.
```

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.

```
for (...  
    for (...  
        for (...  
            if (Katastrophe)  
                goto fhelerbehebung;  
...  
▶fehlerbehebung: ...
```

um aus sehr tiefen Verschachtelungen sehr schnelle (zur Ausführungszeit) heraus zu kommen, kann es sinnvoll sein, ein goto zu verwenden (in C, in C++ gibt es bessere Alternative vgl. 4).

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

Im Praktikum sind gotos verboten!

Wie kann das nachfolgende Programm abgeändert werden, so dass anstelle einer while-Schleife eine Marke und eine goto-Anweisung verwendet werden?

```
#include <iostream>
using namespace std;
int main() {
    int x;
    cin >> x;
    while (x) {
        cout << x*x << endl;
        x--;
    }
}
```

4. Ausnahmebehandlung

In Anwendungen tritt manchmal ein Fehler auf, d.h. das Programm macht nicht das, was der Benutzer eigentlich erwartet. Dabei sind folgende Fehlerarten unterscheidbar:

1. Bedienungsfehler

der Benutzer verwendet das Programm nicht im Sinne der Bedienungsanleitung

2. Programmfehler

im Programm wird nicht gemäß der Spezifikation reagiert

3. Umgebungsfehler

der korrekte Ablauf des Programms kann nicht sichergestellt werden, weil die Umgebung nicht die Anforderungen erfüllt (Speicher, Plattenplatz etc)

Je nach Fehlerart sind die Reaktionen und Fehlermeldungen unterschiedlich.

Bedienfehler dürfen **nie Programmabbrüche** hervorrufen. Fehlbedienungen müssen dem Benutzer verständlich per Programm erläutert werden (vgl. letztes Menu).

Programmierfehler müssen vom **Programmierer** behoben werden.

Umgebungsfehler sollten **nicht** zu **Programmabbrüchen** führen – dies kann aber nicht immer vermieden werden. Fehlbedienungen müssen einem Anwender verständlich per Programm erläutert werden.

Um das o.a. Verhalten zu erreichen ist neben sauberem Programmentwurf und sorgfältigem Testen das Konzept der **Ausnahmebehandlung** einzusetzen.

4.1. Grundidee der Ausnahmebehandlung

Ein Programm soll so strukturiert werden, dass

- es **getrennte Bereiche** im Code gibt für
 - den normalen Ablauf und
 - den Ausnahmefall bzw. den Fehlerfall und
- **kritische Bereiche** besonders gekennzeichnet werden.

Der **kritische** Programmblock wird „**versuchsweise**“ ausgeführt:

- im Fehlerfall wird der normale Ablauf unterbrochen und
- zum Fehlerbehandlungsblock gesprungen („Fehler goto“);
- tritt kein Fehler ein, wird der normale Ablauf bis zum Ende ausgeführt und
- der Fehlerblock ignoriert.

Im **Fehlerfall** wird dabei ein **Ausnahmeobjekt** erzeugt, das im Fehlerbehandlungsblock auswertbar ist und Daten enthält, die zum Wiederaufsetzen oder zu Fehlermeldungen verwendbar sind.

4.2. Ablauf im Fehlerfall

Der Ablauf kann wie folgt charakterisiert werden:

1. Eine Funktion (bis jetzt `main`) versucht (engl. `try`) den kritischen Block auszuführen.
2. Wenn ein Fehler festgestellt wird, wirft (engl. `throw`) sie eine Ausnahme (engl. `exception`) aus.
3. Die Ausnahme wird vom Fehlerbehandlungsblock aufgefangen (engl. `catch`).

4.3. Allgemeine Form

Das folgende Schema zeigt, wie dies in C++ notiert wird:

```
// Programmcode
// Programmcode
// Programmcode
try {
    // Programmcode
    if (FehlerAufgetreten)
        throw "Info";
    // Programmcode
    if (andererFehler)
        throw "andere Info";
    // Programmcode
}
catch (const char* Text) {
    // ggfs. Reparatur-
    // und Aufräumarbeiten
    cout << Text << endl;
}
```

unkritischer Bereich

kritischer Bereich

Bereich der Fehlerbehandlung

if-Abfrage auf Fehlerbedingung mit throw-Anweisung

- an beliebiger Stelle im "normalen" Programmablauf
- throw kann Objekt eines beliebigen Datentyps auswerfen

Sprung zu nächster passender catch-Anweisung

- Auswahl anhand des Datentyps des Ausnahmeobjekts
- der Rest des "normalen" Ablaufs wird übersprungen

Programmabbruch falls kein passender catch-Block existiert

Beispiel (Wurzel einer negativen Zahl):

```
#include <iostream.h>
main() {
    float zahl;
    cout << "positive Zahl eingeben: ";
    try {
        cin >> zahl;
        if (zahl < 0)
            throw "keine positive Zahl!";
        cout << "Wurzel von " << zahl << " ist "
             << sqrt(zahl) << endl;
    }
    catch (const char * text) {
        cout << "Fehler: " << text << endl;
    }
}
```

```
$ try01
positive Zahl eingeben: 5
Wurzel von 5 ist 2.23607
$
```

```
$ try01
positive Zahl eingeben: -2
Fehler: keine positive Zahl!
$
```

Bei Eingabe von -2 wird die Ausnahme „keine positive Zahl“ geworfen und in den catch-Block gesprungen.

Mit dieser Technik ist eine bessere Alternative zu goto verwendbar, wenn aus tief verschachtelten Schleifen heraus eine Ausnahmebehandlung schnell aktiviert werden muss (vgl. goto sinnvoll).

4.4. Ausnahmebehandlung unterschiedlichen Typs

In einem try-Block können unterschiedliche Ausnahmen geworfen werden, für jeden Datentyp aber höchstens eine.

Dabei wird der erste **passende** catch-Block ausgeführt:

- die Auswahl erfolgt anhand des **Datentyps** des Ausnahmeobjekts
- das Sprungziel ist durch den Datentyp definiert;
- wenn kein passender catch-Block gefunden wird, wird in der „Schachtelungsebene höher gesucht, bis einer gefunden wird oder falls keiner vorhanden ist, wird das Programm beendet.

Dies ermöglicht eine differenzierte Fehlerbehandlung.

Beispiel (unterschiedliche catch-Blöcke):

```

#include <iostream.h>
main() {
    const int maxValue = 100;
    int zahl1, zahl2, ergebnis;
    cout << "2 Zahl eingeben: ";
    try {
        cin >> zahl1 >> zahl2;

        if (zahl1 > maxValue)
            throw zahl1 - maxValue;
        if (zahl2 > maxValue)
            throw zahl2 - maxValue;
        if (zahl2 == 0)
            throw "Nulldivision nicht möglich";
        ergebnis = zahl1 / zahl2;
        cout << zahl1 << " / " << zahl2 << " = " << ergebnis << endl;
    } catch (const int wert) {
        cout << "Fehler: Wert um " << wert << " zu gross!" << endl;
    } catch (const char * text) {
        cout << "Fehler: " << text << endl;
    }
}
}

```

Auf die Problematik der Ausnahmebehandlung wird nochmals eingegangen, wenn die Sprachkonstrukte „Funktion“ und „Klasse“ bekannt sind.

```

$ try02
2 Zahl eingeben: 18 6
18 / 6 = 3
$ try02
2 Zahl eingeben: 18 0
Fehler: Nulldivision nicht möglich
$ try02
2 Zahl eingeben: 18 1001
Fehler: Wert um 901 zu gross!

```

Hörsaalübung

Schreiben Sie ein C++-Programm, das Ihre Matrikel-Nummer einliest und deren Quersumme berechnet!

Verwenden Sie Ausnahmebehandlungen, wenn eine ungültige Matrikelnummer eingegeben wurde ($1 \leq \text{MatrNr} \leq 999999$).

Beispiel:

Mat-Nr: 123123

Quersumme: 12