Objektorientierte Programmierung
(Winter 2010/2011)

# Kapitel 7: Anweisungen (Statements)

- Wertausdruck als Statement
- Deklaration als Statement, Initialisierung
- Hintereinanderausführung (Block)
- Bedingte Statements: if, switch
- Schleifen: while, for, do
- Sprünge: return, break, continue, goto





## Inhalt







## Statements (1)

- Anweisungen (engl. Statements) sind Teile eines Programms, die den Zustand verändern.

  Z.B. Variablen einen neuen Wert zuweisen, oder eine Ein-/Ausgabe vornehmen. Die sukzessive Änderung des Berechnungszustands ist in imperativen Sprachen das zentrale Programmierkonzept.

- In C/C++ können auch Werteausdrücke (Expressions) den Zustand verändern, hier ist die Trennung zu Statements nicht so klar wie z.B. in Pascal.

  Man spricht von einem Seiteneffekt, wenn ein Wertausdruck den Zustand verändert. Dies ist in vielen Sprachen mindestens verpönt, eventuell auch ganz verboten. In C/C++ ist die Zuweisung (sonst ein klassisches Statement) selbst ein Wertausdruck.





## Statements (2)

- In anderen Sprachen ist auch die Unterscheidung von Statements und Deklarationen üblich:
  - Für Statements wird ausführbarer Code erzeugt (also Befehle für die CPU).

    Diese Befehle werden zur späteren Laufzeit des Programms ausgeführt.

  - Deklarationen verarbeitet der Compiler, indem er Einträge in interne Tabellen vornimmt.

    Wenn man bei Deklarationen überhaupt von “ausführen” sprechen kann, so geschieht es zur Compilezeit: Der Compiler reserviert z.B. Speicherplatz für eine Variable und trägt ihre Adresse in die Symboltabelle ein.

# Statements (3)

- In C++ (nicht C!) sind Deklarationen dagegen formal auch Statements und können beliebig mit ihnen gemischt werden.

  Sonst ist üblich, daß man erst die Deklarationen angeben muß und dann die Statements. In vielen Sprachen können Deklarationen allerdings Initialisierungen enthalten, wofür dann doch ausführbarer Code erzeugt wird. Das ist aber nur eine Abkürzung für ein vorgezogenes Statement.

- Ein Programm setzt sich letztendlich aus Deklarationen und Statements zusammen, strukturiert in Prozeduren/Methoden, Klassen und Module.





# Expression Statement (1)

- Ein Wertausdruck gefolgt von einem Semikolon ";" ist ein Statement.

  Das sogenannte "expression statement".

- Die zwei häufigsten Fälle sind:
  - ◇ Eine Zuweisung:
    ⟨Variable⟩ = ⟨Ausdruck⟩ ;
  - ◇ Ein Prozeduraufruf (oder Methodenaufruf):
    ⟨Prozedurname⟩ ( ⟨Parameter⟩ ) ;

    Auch ⟨Objekt⟩.⟨Methodenname⟩(⟨Parameter⟩); entspricht diesem Fall, ebenso ⟨Objektzeiger⟩->⟨Methodenname⟩(⟨Parameter⟩);. Durch die Operatorsyntax gibt es noch mehr syntaktische Varianten, z.B. ist eigentlich auch `cout << "hallo";` ein Prozeduraufruf.





# Expression Statement (2)

- Wenn man von einem Wertausdruck zu einer Anweisung übergeht,
  - ◇ spielt der berechnete Wert keine Rolle mehr (er wird "vergessen"),
  - ◇ wichtig ist allein die Zustandsänderung ("der Seiteneffekt").
- Z.B. sind "`3;`" oder auch "`n+1;`" zwar legale, aber sinnlose Anweisungen.

  Der Compiler sollte eigentlich eine Warnung ausgeben.





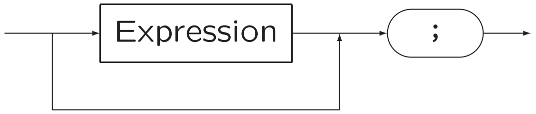
# Expression Statement (3)

- Auch beim Aufruf einer Prozedur, die einen Wert zurückliefert, sollte ein guter Compiler eine Warnung ausgeben, wenn dieser Wert nicht verwendet bzw. abgefragt wird.
- Durch eine explizite Typumwandlung nach "`void`" (leerer Datentyp) kann man klarmachen, daß man den Wert des Ausdrucks $E$ wirklich ignorieren will:
  - ◇ `(void)` $E$`;` (alte C Syntax)
  - ◇ `static_cast<void>(`$E$`);` (neue C++ Syntax)

# Expression Statement (4)

- Syntaxdiagramm:

  Expression Statement:




- Auch ein einzelnes Semikolon (ohne Wertausdruck davor) ist also eine Anweisung ("leere Anweisung").

  Ist syntaktisch manchmal nötig, z.B. in einer Schleife, wenn der Schleifenkopf schon alles macht.





# Inhalt

1. Allgemeines, Expression Statement
2. Blöcke, Deklarationen
3. Bedingte Anweisungen (if, switch)
4. Schleifen (while, do, for)
5. Sprünge (break, continue, goto, return)
6. Zusammenfassung

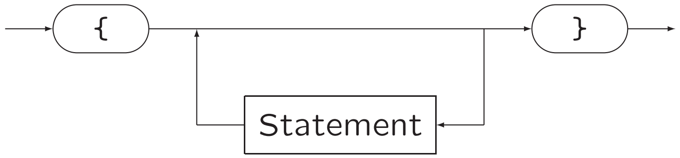




# Block/Sequenz (1)

- Eine Folge von Anweisungen, eingeschlossen in geschweifte Klammern "{" und "}", ist wieder eine Anweisung ("compound statement").
- Syntax-Diagramm:

  Compound Statement:








# Block/Sequenz (2)

- Auf diese Weise kann man an Stellen, an denen syntaktisch nur eine Anweisung erlaubt ist (z.B. als Schleifenrumpf oder von `if` abhängige Anweisung) eine ganze Folge von Anweisungen unterbringen.
- Die in einem "compound statement" zusammengefassten Anweisungen werden sequentiell nacheinander ausgeführt.

  D.h. erst wird die erste Anweisung vollständig ausgeführt, dann die zweite, u.s.w. Der Codeoptimierer könnte die Reihenfoge eventuell umstellen, wenn das Vorteile bringt, aber nur unter der Bedingung, daß sich die Bedeutung des Programms dadurch nicht ändert.

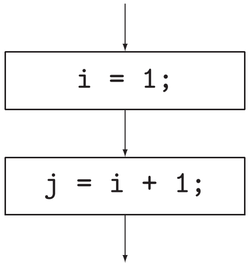
# Block/Sequenz (3)

- Z.B. kann man

  `{ i = 1; j = i + 1; }`

  graphisch als Flußdiagramm so veranschaulichen:

  Statt Flußdiagramm sagt man auch "Programmablaufplan".

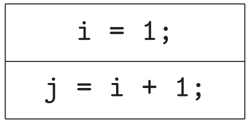







# Block/Sequenz (4)

- Als Alternative zu Flußdiagrammen gibt es auch Struktogramme (Nassi-Shneiderman-Diagramme):




Flußdiagramme sind in DIN 66 001 genormt, Struktogramme in DIN 66 261. Flußdiagramme sind in Verruf geraten, weil sie unstrukturierten Programmen mit beliebigen Sprüngen entsprechen. Sie scheinen mir persönlich aber übersichtlicher zu sein (und zeigen eher, wie die CPU hinterher das Programm abarbeitet). Wenn ich Algorithmen (Berechnungsverfahren) erläutern will, verwende ich aber Pseudocode.





# Block/Sequenz (5)

- In anderen Sprachen (z.B. Pascal) schreibt man
  - "**`begin`**" statt "`{`",
  - "**`end`**" statt "`}`".

    C hatte schon immer einen Hang zur Kürze.
  - Das ganze Konstrukt (Deklarationen plus Anweisungsfolge) nennt man auch einen Block.
  - In Pascal wird das Semikolon zur Trennung der Anweisungen verwendet, in C schließt es Anweisungen ab.

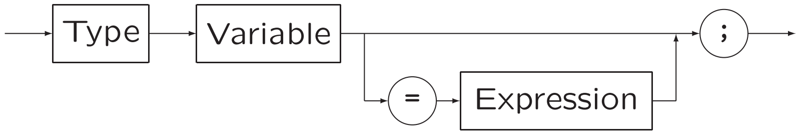
    D.h. in Pascal würde nach der letzten Anweisung kein "`;`" stehen.





# Deklaration (1)

- Eine Deklaration teilt dem Compiler (und dem Leser es Programms) mit, wofür ein Bezeichner steht, der im Programm verwendet wird.
- Eine Variablendeklaration hat (stark vereinfacht) folgenden Aufbau:




- Es gibt später noch ein ganzes Kapitel über Deklarationen.

# Deklaration (2)

- Eine Deklaration wie z.B.

```
int i;
```

  reserviert zwar Speicherplatz für die Variable `i`, aber trägt in diesen Speicherplatz keinen Wert ein.
- Der Wert der Variablen `i` ist dann undefiniert (bis man ihr das erste Mal explizit einen Wert zuweist).
- Man nennt Variablen, denen noch nie ein Wert zugewiesen wurde, "uninitialisierte Variablen".





# Deklaration (3)

- Da die Bits im Hauptspeicher nur 0 oder 1 sein können (nicht "leer"), wird man beim Zugriff auf eine uninitialisierte Variable einen Wert erhalten.
- Dieser Wert ist aber kaum vorhersehbar (er hängt davon ab, was vorher in dieser Speicherzelle stand).

  Das könnte auch ein Wert von einem ganz anderen Typ gewesen sein. Außerdem könnte jederzeit eine neue Compilerversion den Speicherplatz anders aufteilen, man würde dann einen anderen Wert erhalten.
- Der (Lese-)Zugriff auf eine uninitialisierte Variable ist also immer ein Fehler.





# Deklaration (4)

- Leider können Compiler nicht immer feststellen, ob Zugriffe auf uninititialisierte Variable vorkommen.

  Dies hängt ja auch von den Eingabewerten ab. Man kann beweisen, daß diese Frage unentscheidbar ist, d.h. daß es unmöglich ist, ein Computerprogramm zu schreiben, daß andere Computerprogramme auf die korrekte Initialisierung der Variablen für beliebige Eingaben prüft (und immer nach endlicher Zeit eine richtige Antwort ausgibt).
- Selbstverständlich kann ein Compiler Warnungen ausgeben, wenn er vermutet, daß möglicherweise auf eine uninitialisierte Variable zugegriffen wird.

  Diese Warnung muß aber nicht immer stimmen. Umgekehrt finden viele Compiler auch nicht alle Fälle uninitialisierter Variablen.





# Deklaration (5)

- Fehler können klassifiziert werden in:
  - ◇ Fehler, die der Compiler bemerkt:
    Meist leicht zu beseitigen.
  - ◇ Fehler, die bei der Programmausführung zum Abbruch führen (z.B. Division durch 0):
    Nicht zu übersehen, man hat einen Anhaltspunkt.

    Natürlich kann es sein, daß dieser Fehler nur bei bestimmten Eingaben sehr selten auftaucht. Dann wird es auch schwieriger.
  - ◇ Falsches Ergebnis: Eventuell gar nicht bemerkt.
- Uninitialisierte Variablen fallen in die letzte (und schwierigste) Kategorie.

# Deklaration (6)

- Weil uninitialisierte Variablen ziemlich übel sind, ist in vielen Sprachen (so auch C/C++) die Möglichkeit vorgesehen, einer Variablen gleich bei ihrer Deklarationen einen Wert zuzuweisen, z.B.

```
int i = 1;
```

- In vielen Sprachen (auch C) müssen in einem Block erst die Deklarationen stehen, dann die Statements.
- Das führt dazu, daß man oft noch keinen sinnvollen Wert für die Initialisierung hat (er muß erst berechnet werden).





# Deklaration (7)

- Deshalb wurde in C++ die Möglichkeit vorgesehen, Deklarationen und Anweisungen beliebig zu mischen.

  Formal ist eine Deklaration in C++ ein Statement.

- Auf diese Art kann man die Deklaration einer Variablen verschieben, bis man ihr einen sinnvollen Wert zuweisen kann.

  Eventuell geschieht die Initialisierung über einen Prozeduraufruf, etwa bei einer Eingabe. Dann muß man die Variable doch erst mit undefiniertem Wert deklarieren. Aber immerhin kann man das direkt vor der Eingabeanweisung machen. Auch bei Arrays (s.u.) ist eine Initialisierung bei der Deklaration nicht immer sinnvoll möglich.





# Deklaration (8)

- Manche Programmierer (die z.B. von C kommen) finden solche beliebigen Deklarations-Positionen aber unübersichtlich.
- Sie ordnen die Deklaration dann doch immer am Anfang eines Blockes an, eventuell sogar nur des Blocks, der den Prozedurrumpf darstellt.
- Damit nutzen sie die Möglichkeiten der Sprache aber nicht optimal aus.

  Letztendlich ist der Programmierer dafür verantwortlich, daß er nur auf initialisierte Variablen zugreift. Er sollte dazu aber Werkzeuge nutzen, z.B. auch entsprechende Compilerwarnungen anschalten.





# Deklaration (9)

- Deklarationen müssen textuell vor der Verwendung der Variablen stehen.

  Der Compiler muß erst die Deklaration sehen, bevor er Code für Zugriffe auf die Variable erzeugen kann. Der Typ der Variablen bestimmt ja die zu erzeugenden Maschinenbefehle, und eventuell nötige Typumwandlungen.

- Die Deklaration der Variaben gilt nur bis zum Ende des Blockes, in dem sie deklariert ist.

  Sie gilt aber auch in allen darin geschachtelten Blöcken. Gültigkeitsbereiche werden in einem späteren Kapitel genauer diskutiert.

# Inhalt







# If Statement (1)

- Mit der `if`-Anweisung ist es möglich, andere Anweisungen nur beim Vorliegen bestimmter Bedingungen auszuführen ("if" = "wenn", "falls").

  Die `if`-Anweisung gehört zusammen mit der `switch`-Anweisung zur Klasse der bedingten Anweisungen.

- Die `if`-Anweisung gibt es in zwei Varianten: Mit und ohne `else` ("sonst", "andernfalls").
- Beispiel:

```
if(x >= 0)
        y = x;
else
        y = -x;
```





# If Statement (2)

- Es wird zunächst die Bedingung ausgewertet.
- Ist sie verschieden von Null ("true"), wird die vom `if` abhängige Anweisung ausgeführt.

  C hatte noch keinen booleschen Datentyp. Er wurde in C++ eingeführt, aber aus Kompatibilitätsgründen sind beliebige Umwandlungen von/nach `int` möglich. In den meisten anderen Sprachen muß die Bedingung dagegen den Datentyp `bool` haben.

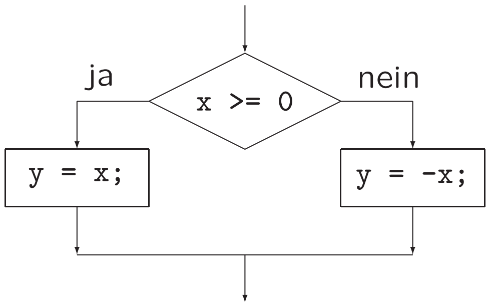
- Ist die Bedingung gleich Null ("false"), so wird die vom `else` abhängige Anweisung ausgeführt (sofern vorhanden).





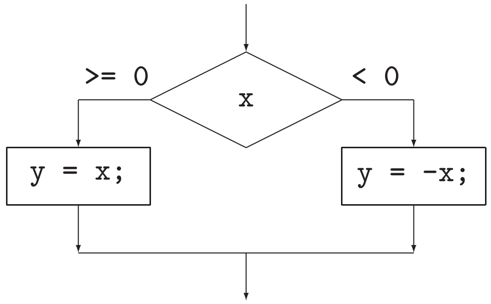
# If Statement (3)

- Auf diese Weise entsteht eine Verzweigung im Programmablauf.
- Flußdiagramm:



# If Statement (4)

- Flußdiagramm (Alternative):



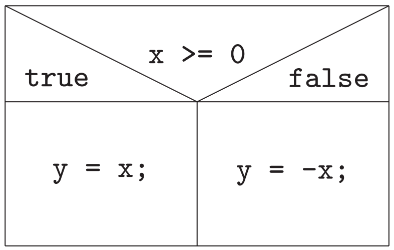

Die Bedingungskanten müssen die Raute nicht immer links und rechts verlassen. Auch auf einer Seite und unten ist möglich.





# If Statement (5)

Struktogramm:




Statt "`true`" und "`false`" kann man auch "`T`" und "`F`" schreiben. Manche Autoren schreiben die Bedingung auch "`if(x >= 0)`".



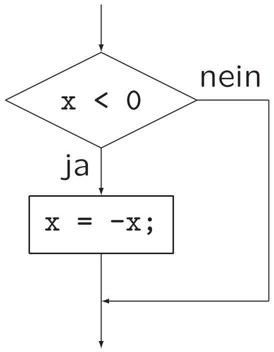


# If Statement (6)

- Beispiel (`if` ohne `else`):

```
if(x < 0) x = -x;
```

- Flußdiagramm:

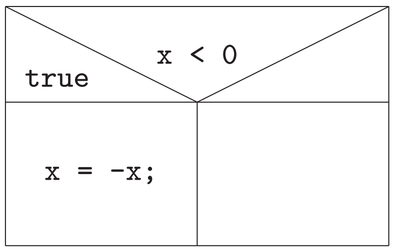







# If Statement (7)

- Struktogramm für den Fall "`if` ohne `else`":



# If Statement (8)

- Die vom `if` abhängige Anweisung wird manchmal auch die `then`-Klausel genannt, weil man in Pascal und ähnlichen Sprachen folgendes geschrieben hat:

```
if x < 0 then        { KEIN C++! }
         x = -x
```

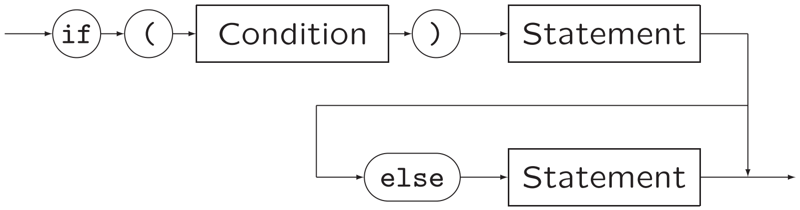
- Hier braucht man keine Klammern um die Bedingung (`if` und `then` sind sozusagen die Klammern).
- In C/C++ muß die Bedingung dagegen immer in Klammern eingeschlossen sein.





# If Statement (9)

- If-Statement:




- Bei `if (A) if (B) C else D` ist
  - ◇ `if (A) { if (B) C else D }` gemeint, und nicht
  - ◇ `if (A) { if (B) C } else D` (falsch).

  (`else` gehört immer zum nächstmöglichen `if`).





# If Statement (10)

- Weil die `if`-Anweisung selbst wieder eine Anweisung ist, lassen sich `else if`-Ketten bilden, indem man eine `if`-Anweisung für die von `else` abhängige Anweisung einsetzt:

```
if(x > 0)
        sign = +1;
else if(x == 0)
        sign = 0;
else // x < 0
        sign = -1;
```

In jedem `else`-Zweig gilt, daß alle vorherigen `if`-Bedingungen falsch sind. Man kann ggf. einen Kommentar benutzen, um dies zu erklären.





# If Statement (11)

- Falls man in einem der Zweige mehrere Anweisungen sequentiell nacheinander ausführen will, muß man sie mit `{ ... }` zusammenfassen.

  Wie oben erläutert, nutzt eine korrekte Einrückung nichts.

- Selbstverständlich darf man immer `{ ... }` setzen (auch bei nur einer abhängigen Anweisung).

  Dem geübten C-Programmierer erscheint das aber umständlich.

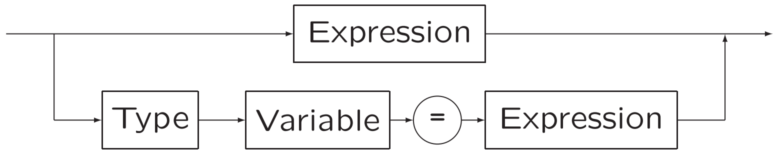
- In der Sprache Algol 68 war das Problem mit schließenden Schlüsselworten für alle Kontrollstrukturen gelöst, z.B. `if ... then ... else ... fi`.

# If Statement (12)

- In der Bedingung kann man wieder eine Variable deklarieren, die dann für die beiden von `if` und `else` abhängigen Anweisungen gilt.

  Da jeder von Null verschiedene Wert als "true" gilt (Bedingung ist erfüllt), muß die Variable nicht unbedingt den Typ `bool` haben. Die Deklaration im Syntaxdiagramm ist wieder vereinfacht.

- Condition:








# Switch Statement (1)

- Es ist manchmal nötig, viele verschiedene Werte für eine Variable getrennt zu behandeln, z.B.

```
if(tag == 1)
        cout << "Montag";
else if(tag == 2)
        cout << "Dienstag";
...
else if(tag == 7)
        cout << "Sonntag";
else
        cout << "FEHLER!";
```





# Switch Statement (2)

- Solche Situationen kann man übersichtlicher mit der `switch`-Anweisung formulieren (Beispiel s.u.).
- Die `switch`-Anweisung ist nicht unbedingt nötig: Man kann damit nichts machen, was man nicht auch mit `if/else if`-Ketten machen könnte.
- Je nach Verteilung der zu betrachtenden Werte erzeugt der Compiler ggf. einen Sprung über eine Tabelle mit den Startadressen der verschiedenen Fälle.
- Das kann effizienter (schneller) sein als die entsprechende `if/else if`-Kette.



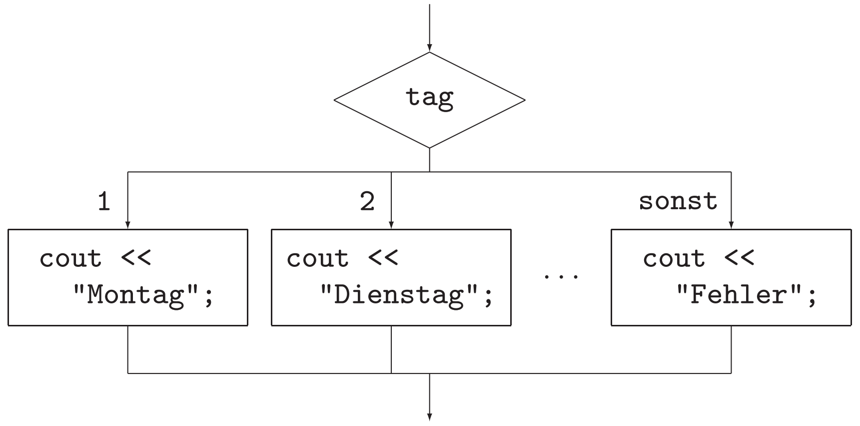


# Switch Statement (3)

```
switch(tag) {
    case 1:
        cout << "Montag";
        break;
    case 2:
        cout << "Dienstag";
        break;
    ...
    case 7:
        cout << "Sonntag";
        break;
    default:
        cout << "FEHLER!";
}
```

# Switch Statement (4)

Flußdiagramm:

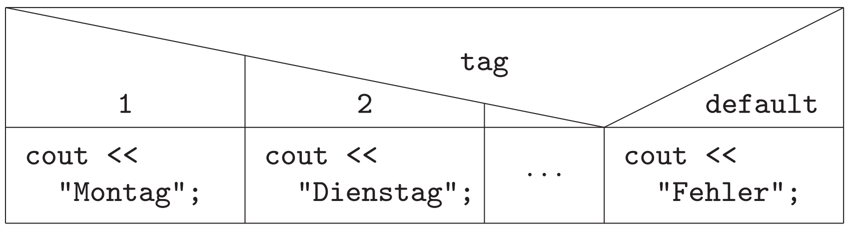







# Switch Statement (5)

Struktogramm:

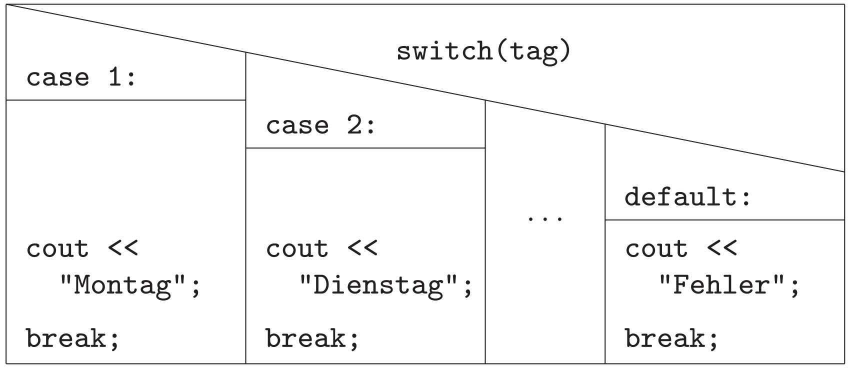







# Switch Statement (6)

Struktogramm (Alternative):








# Switch Statement (7)

- Man kann auch mehrere verschiedene Werte in einem gemeinsamen Fall behandeln:

```
switch(c) {
    case 'a':
    ...
    case 'z':
        ... // Kleinbuchstabe
        break;
    case 'A':
    ...
    case 'Z':
        ... // Grossbuchstabe
        break;
}
```

# Switch Statement (8)

- Wie man am letzten Beispiel sieht, ist es nicht nötig, den `default`-Fall anzugeben.
- Sollte keiner der mit `case` angegebenen Fälle zutreffen, geschieht dann einfach gar nichts.

  D.h. es gibt ein implizites leeres `default`.
- Ein Programm sollte auch mögliche Fehlerfälle abfangen. Selbst wenn es eigentlich nicht passieren dürfte, daß keiner der `case`-Fälle zutrifft, sollte man einen `default`-Fall mit einer Fehlermeldung und ggf. einem Programmabbruch vorsehen.





# Switch Statement (9)

- Falls man das `break` am Ende eines Falls vergißt, wird die Ausführung mit den Anweisungen des folgenden Falls fortgesetzt.
- Das ist gelegentlich nützlich, aber in den meisten Fällen ein Fehler.

  Falls man es wirklich will, sollte man einen Kommentar verwenden, um darauf hinzuweisen, daß kein Fehler vorliegt. Z.B. `/* FALLTHROUGH */` oder `// Drops through`.
- Statt `break` kann man auch `return` verwenden, wenn man gleichzeitig die Prozedur beenden will.





# Switch Statement (10)

- Syntaktisch kann eine `switch`-Anweisung eine beliebige Anweisung enthalten, es müßte theoretisch nicht ein Block sein.
- Switch-Statement:


- Die Bedingung muß von ganzzahligem Typ sein.

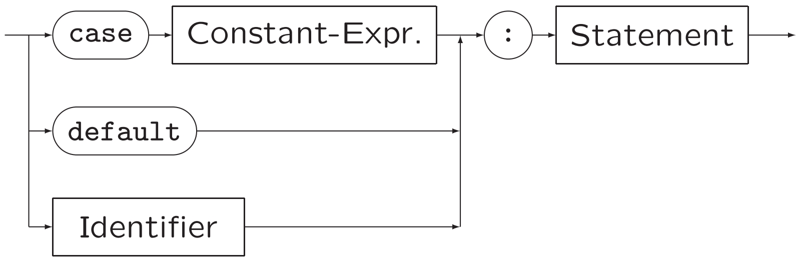
  D.h. `char`, `unsigned char`, `signed char`, `short`, `unsigned short`, `int`, `unsigned int`, `long`, `unsigned long`, `bool`, oder ein Aufzählungs-Typ (`enum`, s.u.).





# Switch Statement (11)

- Man kann an beliebiger Stelle im Innern eines `switch` eine Anweisung mit `case` oder `default` markieren.
- Labeled-Statement:




  Der letzte Fall ist für das `goto`, s.u.

# Switch Statement (12)

- Damit ist es legal, z.B. in das Innere einer bedingten Anweisung oder eine Schleife "hineinzuspringen".

  Man kann nur nicht hinter eine Deklaration mit einer Initialisierung springen, auch nicht in einen Exception Handler (s.u.).

- Nicht alles, was legal ist, ist auch gut!
- Bei Hausaufgaben/Klausur können auch Punkte für schlechten Stil abgezogen werden.

  Es ist zwar gut, eine Sprache vollständig verstanden haben, aber man muß das nicht unbedingt durch irgendwelche Tricks zeigen. Falls es nicht sehr gute Gründe gibt, sollte man eine möglichst übersichtliche, klare, einfache Lösung wählen.





# Switch Statement (13)

- Anweisungen im `switch` vor dem ersten `case` werden niemals ausgeführt. Der Compiler sollte eine Warnung ausgeben.
- `case` und `default` sind nur im Inneren eines `switch` erlaubt.
- Nach dem `case` muß eine "Constant Expression" stehen. Normalerweise ist das eine Konstante, aber man kann bei Bedarf `+`, `-`, `*` etc. anwenden.

  Der Compiler muß in der Lage sein, den Wert schon zur Compilezeit zu berechnen. Das ist wichtig, damit er ggf. die Sprungtabelle aufbauen kann.





# Inhalt







# While Statement (1)

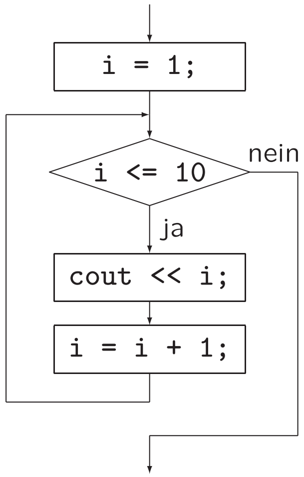
- Die `while`-Anweisung führt eine abhängige Anweisung (den Schleifenrumpf) solange aus, wie eine Bedingung erfüllt ist ("while": u.a. "solange wie").
- While-Statement:


- Beispiel:

```
i = 1;
while(i <= 10) {
        cout << i;
        i = i + 1;
}
```

# While Statement (2)

Flußdiagramm:

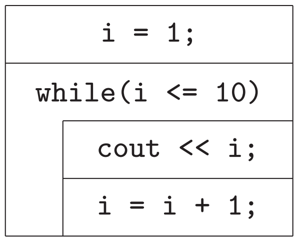







# While Statement (3)

- Struktogramm:




In der Literatur werden für den Schleifenkopf unterschiedliche Notationen verwendet. Manche schreiben z.B. "`Solange`" oder "`DO`" anstelle von "`while`" und lassen die Klammern weg. Manche geben auch nur die Schleifenbedingung an. Da das gleiche graphische Symbol auch für die `for`-Schleife verwendet wird, ist dies aber etwas problematisch.





# While Statement (4)

- Die `while`-Bedingung muß irgendwann falsch werden, sonst erhält man eine Endlosschleife.
- Wenn ein Programm nicht (freiwillig) anhält, sagt man auch, das Programm terminiert nicht.

  Man kann das Programm natürlich immer explizit abbrechen. Wie das genau geht, hängt vom Betriebssystem ab. Meist reicht es, "Ctrl+C" (Steuerung-C) zu drücken. Unter Windows liefert Ctrl+Alt+Del eine Liste aller Prozesse, aus der man den abzubrechenden Prozess aussuchen kann, Unter UNIX zeigt `ps` oder `ps -ef` eine Liste von Prozessen an, der Abbruch geht dann mit `kill` ⟨`Prozess-ID`⟩ bzw. notfalls `kill -9` ⟨`Prozess-ID`⟩.





# While Statement (5)

- Es ist unentscheidbar, ob ein Programm irgendwann anhält.

  Dies ist das bekannte Halteproblem.
- D.h. es wird nie einen Compiler geben, der genau dann eine Fehlermeldung ausgibt, wenn eine Schleife nicht terminieren wird.

  Natürlich muß der Compiler selbst terminieren. Das Problem wäre selbst dann unentscheidbar, wenn man sich nur für eine feste Eingabe interessiert.
- Der Compiler könnte in offensichtlichen Fällen eine Warnung ausgeben, aber die wenigsten tun das.

# While Statement (6)

- Um zu beweisen, daß eine Schleife enden wird, kann man jedem Berechnungszustand eine ganze Zahl zuordnen, die
  - ◇ beim Beginn jedes Schleifendurchlaufs nicht negativ ist, und
  - ◇ bei jedem Schleifendurchlauf um mindestens 1 reduziert wird.
- Im Beispiel wäre das z.B. $10 - \texttt{i}$.

  Durch `i = i + 1;` wird der Wert von $10 - \texttt{i}$ bei jedem Schleifendurchlauf verringert. Durch die Schleifenbedingung `i <= 10` ist sichergestellt, daß der Wert nie negativ wird.





# While Statement (7)

Aufgabe:

- Was halten Sie von diesem Programmstück zur Berechnung der Fakultät ($n! = 1 * 2 * 3 * \cdots * n$)?

```
int n;
cout << "Bitte n eingeben: ";
cin >> n;
int fak = 1;
while(n != 0) {
        fak *= n;
        n = n - 1;
}
cout << "n! = " << fak << "\n";
```





# While Statement (8)

Aufgabe:

- Was ist der Fehler in diesem Programmstück?

```
int n;
int fak = 1;
cout << "Bitte n eingeben: ";
cin >> n;
while(n > 0);
{
        fak *= n;
        n = n - 1;
}
```

- Was passiert, wenn man es ausführt?





# While Statement (9)

- Man beachte, daß eine Schleife auch 0 Mal ausgeführt werden kann (wenn die Schleifenbedingung gleich zu Anfang falsch ist).

  Falls man eine Variable im Rumpf der Schleife initialisiert, kann man sich nach Ende der Schleife nicht darauf verlassen, daß sie einen definierten Wert hat. Eigentlich sollte man so eine Variable erst im Rumpf der Schleife deklarieren, dann wäre sie außen gar nicht zugreifbar.

- Um die Korrektheit eines Programms zu prüfen, ist es wichtig, solche Extremfälle durchzuspielen.

  Wann sollte auch durchdenken, was passiert, wenn die Schleife genau ein Mal durchlaufen wird. Im allgemeinen Fall sind der erste und der letzte Durchlauf der Schleife wichtige Kandidaten für eine manuelle Simulation.

# While Statement (10)

Aufgabe:

- Was berechnet dieses Programmstück?

```
int n;
cout << "Bitte n eingeben: ";
cin >> n;
int m = 2;
while(n > 1) {
        m += m;
        n = n - 1;
}
cout << "Ergebnis: " << m << "\n";
```

- Funktioniert es immer (z.B. $\texttt{n} = 0$)?





# Iteration

- Die wiederholte Ausführung einer Anweisung oder eines Anweisungsblocks nennt man einer "Iteration" (Wiederholung).
- Schleifenanweisungen (`while`, `do`, `for`) sind "Iterative Statements" (iterative Anweisungen).
- Ein Verfahren, das auf einer Schleife basiert, heisst auch "iterativ".

  Besonders, wenn man den Unterschied zu einem "rekursiven" Verfahren betonen will (siehe Kapitel über Funktionen).





# Do Statement (1)

- Manchmal kommt es vor, daß erst am Ende des Schleifenrumpfes feststeht, ob die Schleife nochmal durchlaufen werden muß.

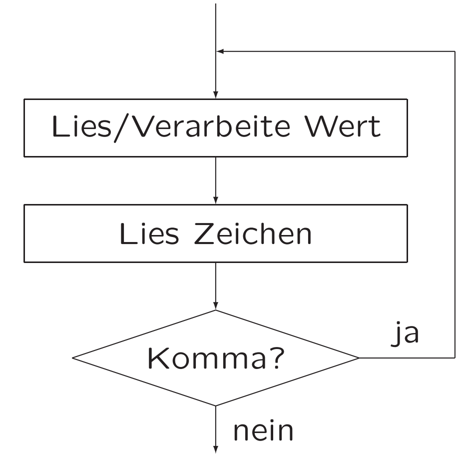
  Dann muß die Schleife natürlich auf jeden Fall mindestens einmal durchlaufen werden.

- Beispiel: Es soll eine durch Kommata getrennte Liste von Werten eingelesen werden (endet mit ".").
  - ◇ Im Schleifenrumpf liest man jeweils einen Wert (und verarbeitet ihn).
  - ◇ Anschließend schaut man sich das nächste Zeichen an (Komma oder Punkt).





# Do Statement (2)



# Do Statement (3)

- Wenn man dieses Verfahren mit einer `while`-Schleife codieren will, muß man das Einlesen und Verarbeiten eines Wertes doppelt aufschreiben:
  - Vor der Schleife (für den ersten Wert), und
  - in der Schleife (für alle folgenden Werte).
- Verdoppelung von Programmcode (Copy&Paste-Programmierung) ist immer schlecht:
  - Der Leser muß längeres Programm verstehen.
  - Wenn man etwas ändert, muß man immer beide Stellen ändern (Vergißt man eine: inkonsistent).





# Do Statement (4)

- Daher gibt es in C/C++ die `do`-Schleife:

```
do {
        Lies/Verarbeite Wert; // Pseudocode
        Lies Zeichen c;       // Kein C++
} while(c == ',');
```

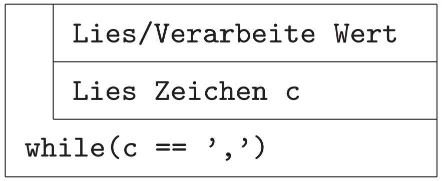
- Während die `while`-Schleife "kopfgesteuert" ist, ist dies eine "fußgesteuerte" Schleife.
- Der Rumpf der `do`-Schleife wird immer mindestens einmal durchlaufen.





# Do Statement (5)

Struktogramm:

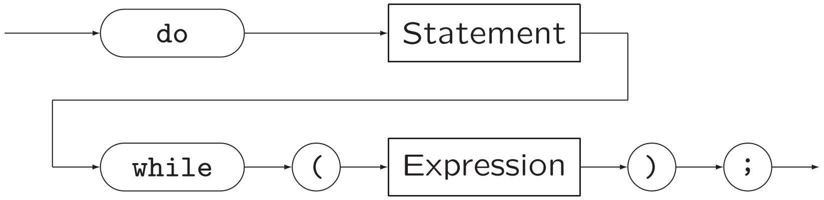







# Do Statement (6)

Syntax-Diagramm (Do-Statement):



# Do Statement (7)

- Viele Leute finden die Syntax des Do-Statements nicht besonders hübsch/übersichtlich.

  Man möchte die wichtige Schleifenbedingung lieber gleich oben sehen. Es besteht eine gewisse Verwechselungsgefahr mit einer `while`-Schleife mit leerem Rumpf, die in C/C++ durchaus vorkommt (deswegen sollte man hier nach "}" keinen Zeilenumbruch machen). In Pascal gibt es deswegen `repeat-until`, wobei hier aber verwirrend ist, daß eine wahre Schleifenbedingung zum Abbruch führt.

- Bjarne Stroustrup schreibt, daß nach seiner Erfahrung `do`-Schleifen häufiger zu Fehlern führen.

  Er sagt, daß auch für den ersten Durchlauf der Schleife (bevor die Bedingung geprüft wird), häufig doch etwas ähnliches wie die Schleifenbedingung gelten muß (damit der Rumpf korrekt funktioniert). Diese etwas abgeschwächte Bedingung wäre aber oft nicht garantiert.





# Do Statement (8)

- Vielleicht sollte man dem Rat von Herrn Stroustrup folgen und auf das `do`-Statement ganz verzichten.

  Oder es jedenfalls nur verwenden, wenn es eindeutig einen Vorteil bringt.

- Die Verdopplung von größeren Stücken Programmcode ist aber sicher schlimmer.
- Falls es sich aber nur um einen Ausdruck/eine Anweisung handelt, ist das unproblematisch.
- Dies kann man durch Einsatz von Prozeduren (s.u.) immer erreichen.





# For Statement (1)

- Die folgende Schleifenstruktur ist typisch:

```
i = 1;                  // Initialisierung
while(i <= 10) {        // Bedingung
        cout << i;
        i = i + 1;      // Schritt (Weiterschalten)
}
```

- Man kann die Schleifensteuerung mit diesen drei Komponenten im Schleifenkopf zusammenfassen:

```
for(i = 1; i <= 10; i = i + 1)
        cout << i;
```

- Die Programmstücke verhalten sich völlig gleich.





# For Statement (2)

- Die Variable `i`, die nacheinander eine leicht zu verstehende Folge von Werten annimmt, und damit den Rest der Schleife steuert, heißt auch die Laufvariable der Schleife.
- Wenn es eine Laufvariable gibt, ist die `for`-Schleife (nach einer gewissen Gewöhnung an die Syntax) übersichtlicher als die entsprechende `while`-Schleife.
- Man kann die `for`-Schleife aber als Abkürzung für die entsprechende `while`-Schleife definieren, sie gibt keine grundsätzlich neuen Möglichkeiten.

# For Statement (3)

- In Pascal sieht die `for`-Schleife so aus:

```
for i := 1 to 10 do     { Kein C++ }
          writeln(i);
```

- Die Syntax ist zunächst übersichtlicher.
- Außerdem terminiert diese Art der Schleife immer.
- Die `for`-Schleife in C/C++ erlaubt dagegen auch ganz andere Arten von Laufvariablen: Nicht nur über Zahlen, sondern z.B. auch über den Elementen einer verketteten Liste (s.u.).





# For Statement (4)

- Es wäre ganz schlechter Stil, wenn die Laufvariable im Innern des Schleifenrumpfes geändert würde.

  Der einzige Vorteil der `for`-Schleife in C/C++ gegenüber der entsprechenden `while`-Schleife ist es, daß man die komplette Schleifenkontrolle gleich zu Anfang sehen kann. Es ist also klar, welche Werte die Laufvariable nacheinander annehmen wird. Denkt man jedenfalls.

  Eine Zuweisung an die Laufvariable im Schleifenrumpf würde diesen Vorteil ins Gegenteil verkehren: Der Leser rechnet damit nicht, sondern nimmt an, daß im Kopf der Schleife alles über die Laufvariable ausgesagt ist, was er wissen muß. Allenfalls könnte vielleicht ein `break`-Statement (s.u.) die Schleife vorzeitig beenden, aber auch das ist etwas problematisch (eventuell in Kommentar ankündigen).

  In Pascal sind Zuweisungen an die Laufvariable im Schleifenrumpf verboten, in C/C++ wären sie legal.





# For Statement (5)

- Man kann die Laufvariable auch gleich in der `for`-Schleife deklarieren:

```
for(int i = 1; i <= 10; i = i + 1)
          cout << i << "\n";
```

- Die Variable `i` ist jetzt nur innerhalb der Schleife bekannt (deklariert).
- Ein Versuch, nach Ende der Schleife auf `i` zuzugreifen, sollte zu einem Fehler führen.

  Bei Microsoft Visual C++ ist es möglich, aber wenn Sie davon Gebrauch machen, kann man Ihre Programme nicht mehr mit dem GNU Compiler übersetzen. Sie sind dann strenggenommen falsch.





# For Statement (6)

- Die drei Teile der `for`-Schleife können (unabhängig von einander) entfallen:
  - ◇ Wenn die Laufvariable z.B. vorher schon initialisiert ist:

```
cin >> n;
if(n < 0) ...; // Fehlerbehandlung
for(; n > 0; n--)
          ...
```

  - ◇ Wenn das Weiterschalten der Laufvariable schon anders geschieht:

```
for(int i = 0; i++ < 100; )
          ...
```

# For Statement (7)

- Optionalität der Komponenten der `for`-Schleife:
  - ◇ Läßt man die Bedingung weg, gilt sie als "true".
  - ◇ Es muß dann also eine andere Art geben, wie die Schleife beendet wird (z.B. eine `break`- oder `return`-Anweisung irgendwo im Schleifenrumpf).

    Dann hat man natürlich nicht mehr den Vorteil der `for`-Schleife, daß man die komplette Schleifenkontrolle sofort sieht. Immerhin ist aber offensichtlich, daß es ein `break` etc. geben muß.
  - ◇ Eine typische "for-ever"-Schleife ist

```
for(;;) {
    ...
}
```





# For Statement (8)

- Es ist auch möglich, zwei Laufvariablen zu verwalten, z.B.

```
for(int i = 1, j = 10; j > 0 && i < 100;
                             i++, j--) {
    ...
}
```

Dieses Beispiel hat keinen besonderen Zweck. Sinnvolle Beispiele mit zwei Laufvariablen gibt es z.B. mit Pointern, etwa beim Kopieren einer Zeichenkette (s.u.). Es könnte aber sein, daß bei mehreren Laufvariablen eine `while`-Schleife übersichtlicher ist, oder man eventuell besser im `for` nur die eine Variable steuert.

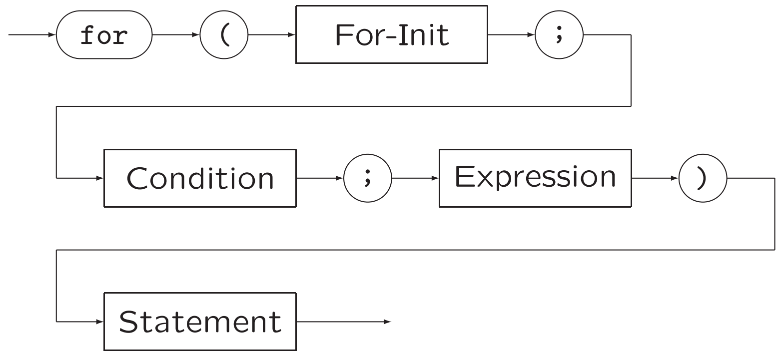
- Im Schritt-Anteil wird hier der Sequenzoperator "," für Ausdrücke verwendet.





# For Statement (9)

Syntaxdiagramm (for-Statement):

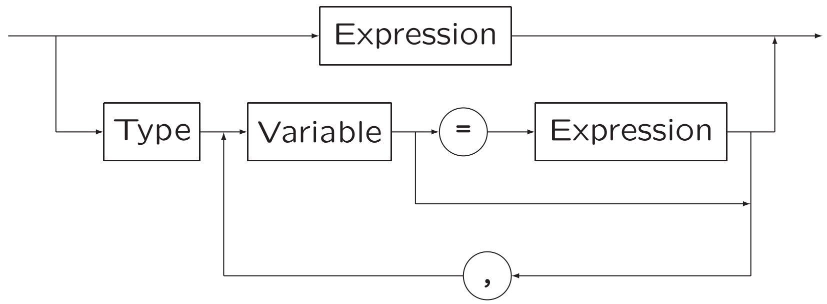







# For Statement (10)

Syntaxdiagramm (for-Init, vereinfacht):



# For Statement (11)

Aufgabe:

- Was gibt dieses Programmstück aus?

```
int n;
cout << "Bitte n eingeben: ";
cin >> n;
for(int i = 0; i < n; i++) {
        for(int j = 0; j <= i; j++)
                cout << '*';
        cout << '\n';
}
```





# Inhalt







# Break Statement (1)

- Die Anweisung “`break;`” beendet die Schleife oder den Switch, in dem es sich befindet.

  Es funktioniert für alle drei Schleifentypen: `while`, `do`, `for`. Falls mehrere Schleifen (oder Switches) geschachtelt sind, wird immer nur die innerste Schleife (bzw. Switch) beendet, in der sich das `break` befindet.

- Auf diese Art kann eine Schleife nicht nur über die Bedingung im Kopf beendet werden, sondern man kann auch an beliebiger Stelle im Rumpf entscheiden, sie zu verlassen.

  Das ist manchmal sehr praktisch. Auf der anderen Seite macht es die Programme unübersichtlicher.





# Break Statement (2)

- Bisher hatte jede Anweisung nur einen Eingang und einen Ausgang.
- Mit dem `break`-Statement kommt jetzt eine weitere Möglichkeit für einen Ausgang hinzu.
- Bisher konnte man bei einer sequentiellen Komposition von Anweisungen (“Block”) sicher sein, daß sie auch wirklich alle ausgeführt werden.

  Von Endlosschleifen abgesehen.

- Jetzt muß man sich die Anweisungen genauer anschauen, ob sie vielleicht ein “`break;`” enthalten.

# Break Statement (3)

- Syntax-Diagramm (break-Statement):


- Es ist ein Syntaxfehler, `break` außerhalb von `while`, `do`, `for`, `switch` zu verwenden.
- Es muß aber nicht direkt in diesen Statements stehen. Es ist z.B. ganz typisch, daß es in einem `if`-Statement steht, das seinerseits im Innern einer Schleife ist.





# Break Statement (4)

Aufgabe:

- Was halten Sie von diesem Programmstück? Was gibt es aus?

```
for(int i = 1; i < 100; i++) {
        cout << "i = " << i << "\n";
        if(i % 2 == 0) {
                break;
                cout << "Aber hallo!\n";
        }
        cout << "Hat nochmal geklappt.\n";
}
```





# Continue Statement

- Mit `continue;` springt man zum Ende des Schleifenrumpfes, d.h. es beginnt sofort der nächste Schleifendurchlauf.

  Bei der `for`-Schleife wird der Schritt-Ausdruck noch ausgewertet, d.h. die Laufvariable wird weitergeschaltet.

- Syntax-Diagramm (continue-Statement):


  Es ist ein Syntaxfehler, `continue` außerhalb von `while`, `do`, `for` zu verwenden. Im Gegensatz zu `break` hat es keine Beziehung zum `switch`-Statement.





# Goto Statement (1)

- Man kann Anweisungen mit einem Label (einem Bezeichner) markieren, z.B.

```
mein_sprungziel: cout << "hallo!\n";
```

- Die Anweisung

```
goto mein_sprungziel;
```

  bewirkt, daß mit der Programmausführung an der mit dem Label markierten Stelle fortgefahren wird.

  Man kann dabei auch in eine tiefe Schachtelungsstruktur hinein oder aus ihr herausspringen. Man darf nur nicht eine Deklaration mit Initialisierung überspringen (oder in einen Exception Handler springen).

# Goto Statement (2)

- Die Verwendung von `goto` ist allgemein als schlechter Programmierstil verpönt ("Spaghetticode" anstelle "strukturierter Programmierung").
- In dieser Vorlesung ist es sehr wahrscheinlich, daß Sie 0 Punkte bekommen, falls Sie in Ihrem Programm `goto` verwenden.
- In automatisch erzeugten Programmen (z.B. von einem Parsergenerator) könnte natürlich `goto` verwendet werden. Solche Programme sind ja nicht zum Lesen gedacht.





# Goto Statement (3)

- Es gibt z.B. einen häufig zitierten Artikel

  Go To Statement Considered Harmful

  von Edsger W. Dijkstra (in den Communications of the ACM, Vol. 11, No. 3, März 1968, S. 147–148).
- Er beginnt mit den Worten: "For a number of years I have been familiar with the observation that the quality of programmers is a decreasing function of the density of go to statements in the programs they produce."

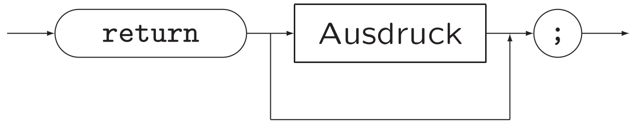




# Return Statement

- Mit dem Return-Statement wird die aktuelle Prozedur verlassen und ggf. ein Rückgabewert festgelegt.

  In gewisser Weise findet hier auch ein Sprung statt: Man geht direkt zum Ende der Prozedur. Pascal hat kein Return-Statement. Das Return-Statement ist allerdings nützlich und allgemein akzeptiert.
- Syntax-Diagramm (Return-Statement):




- Prozeduren werden später besprochen.

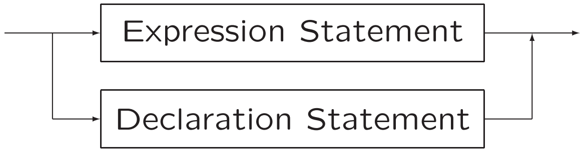




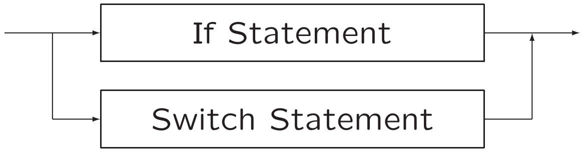
# Inhalt

# Zusammenfassung (1)

- Basic Statement:




- Conditional Statement:

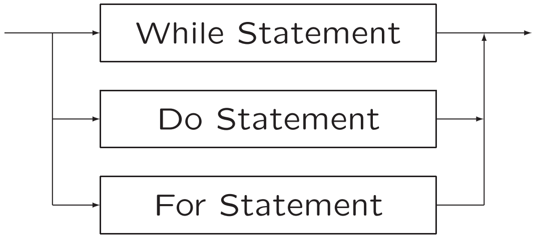







# Zusammenfassung (2)

- Iterative Statement:

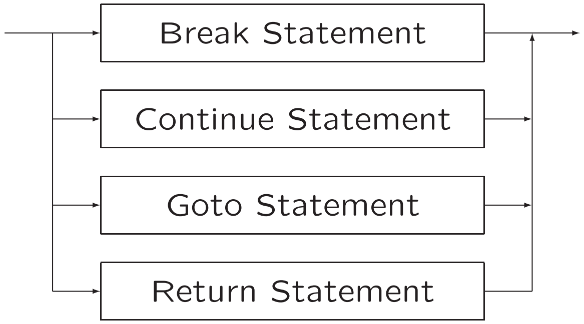







# Zusammenfassung (3)

- Jump Statement:

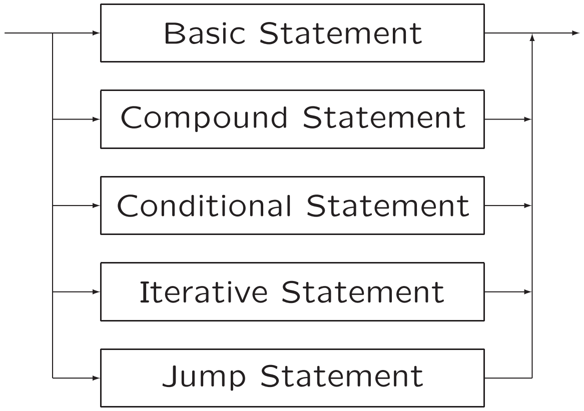






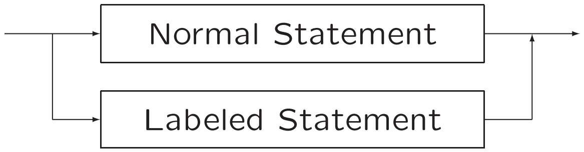

# Zusammenfassung (4)

- Normal Statement:



# Zusammenfassung (5)

- Statement:




- **if**, **switch**, **while**, **do**, **for**, **break**, **continue**, **goto**, **return** werden auch als Kontrollstrukturen bezeichnet.