Technisches Programmieren in C++

Ausdrücke und Befehle

Karl Riedling
Institut für Sensor- und Aktuatorsysteme

Ausdrücke und Befehle

- Initialisierungen
- *L-Values* und *R-Values*
- Funktionen
- Operatoren
- Befehle (*Statements*)

Ausdrücke und Befehle

- **Initialisierungen**
- *L-Values* und *R-Values*
- Funktionen
- Operatoren
- Befehle (*Statements*)

Ausdrücke und Befehle

- Initialisierungen
  - **Allgemeines**
  - Fundamentale Variable
  - Felder; Klassen-Typen ohne Konstruktoren
  - Klassen-Typen mit Konstruktoren
  - Referenzen
  - Programmverzweigungen und Initialisierungen

Initialisierungen — Allgemeines

- Der Ablauf bei der Initialisierung hängt ab von:
  - Speicherklasse (statisch oder automatisch);
  - Klassen- oder Nicht-Klassen-Objekt;
  - Initialisierung mit einer Konstanten oder mit einem nicht-konstanten Ausdruck.

Initialisierungen — Allgemeines

Speicher-klasse	Konstruk-tor?	mit Konstante?	Zeitpunkt und Art der Initialisierung
statisch	nein	nein	Vor Eintritt in den Block, in dem das Objekt erstmals gültig wird; dynamisch
automat.	nein	nein	wie oben
statisch	ja	nein	wie oben
automat.	ja	nein	wie oben
statisch	nein	ja	Beim Laden des Programms
automat.	nein	ja	Vor Eintritt in den Block, in dem das Objekt erstmals gültig wird; dynamisch
statisch	ja	ja	wie oben
automat.	ja	ja	wie oben

Initialisierungen — Allgemeines

- Alle *statischen* Objekte, für die kein Konstruktor existiert und die *nicht* explizit initialisiert werden, werden automatisch auf "0" gesetzt.
- Globale Objekte, für die *kein* Konstruktor existiert, können grundsätzlich nur in *einem* Modul mit einer Konstanten initialisiert werden.
- C++ erlaubt eine Initialisierung globaler Objekte mit einem Ausdruck oder mittels des Konstruktors einer Klasse; in diesem Fall versagen die Schutzmechanismen beim Zusammenbinden.

Initialisierungen — Allgemeines

- Um undefinierte Ergebnisse bei der Definition globaler Objekte zu vermeiden, sollte folgendermaßen vorgegangen werden:
- Globale Variable weitgehend vermeiden. Alternativen:
  - Zusammenfassung global benötigter Programmdaten in einem Klassen-Objekt (Struktur oder Klasse);
  - Definition statischer Daten mit `static`, die sie in ihrer Gültigkeit auf die vorliegende Übersetzungseinheit beschränkt.

Initialisierungen — Allgemeines

- Globale Objekte grundsätzlich in *einer* Übersetzungseinheit definieren und *explizit*, möglichst mit einer Konstanten, *initialisieren*;
- In allen anderen Übersetzungseinheiten sollten sie grundsätzlich mit `extern` *nur deklariert* und *nicht definiert* werden.

Ausdrücke und Befehle

- Initialisierungen
  - Allgemeines
  - **Fundamentale Variable**
  - Felder; Klassen-Typen ohne Konstruktoren
  - Klassen-Typen mit Konstruktoren
  - Referenzen
  - Programmverzweigungen und Initialisierungen

Initialisierungen — Fundamentale Variable

- Objekte der fundamentalen C++-Typen können auch mit nicht-konstanten Ausdrücken initialisiert werden.
- Diese müssen jedoch zum Zeitpunkt der Initialisierung definiert sein.
- Das folgende Programm ist in C++ korrekt, nicht aber in ANSI-C:

Initialisierungen — Fundamentale Variable

```
#include <iostream.h>

int i = 3;                  // in ANSI-C Ok
  // Global; statisch und public, initialisiert mit
  // der Konstante 3. i darf in keinem anderen Modul
  // definiert (wohl aber deklariert) werden.

int j = i;                  // in ANSI-C verboten
  // Global; statisch und communal (d.h., j darf in
  // einem anderen Modul definiert und genau einmal
  // mit einer Konstanten initialisiert werden). j
  // wird vor der Ausführung von main() dynamisch
  // auf den aktuellen Wert von i gesetzt.
```

Initialisierungen — Fundamentale Variable

```
void func ()
{
  int k = 5;                // in ANSI-C Ok
    // Am Stack allokiert und am Beginn von func()
    // dynamisch auf 5 gesetzt.
  int l = 7 * i;            // in ANSI-C Ok
    // Am Stack allokiert und am Beginn von func()
    // dynamisch auf das Ergebnis von i * 7 gesetzt.
  static int m = 11;        // in ANSI-C Ok
    // Statisch, initialisiert mit der Konstanten
    // 11. Behandelt wie eine globale Variable, aber
    // mit einem eindeutig "dekorierten" Namen.
  ...
```


Initialisierungen — Fundamentale Variable

```
  ...
  static int n = 13 * i;    // in ANSI-C verboten
    // Statisch, initialisiert mit Hilfe einer
    // statischen Hilfsvariablen: Hilfsvariable wird
    // beim Laden des Programms auf 0 und bei der
    // Initialisierung von n auf 1 gesetzt;
    // Initialisierung nur, wenn Hilfsvariable == 0
    // – sichert einmalige Initialisierung.
  ...
}
```


Initialisierungen — Fundamentale Variable

- Die *Initialisierung* eines Objekts ist semantisch von der *Zuweisung* eines Wertes verschieden.
- Die folgenden zwei Code-Fragmente sind ihrer Philosophie nach verschieden, obwohl sie identischen *Object-Code* erzeugen:

Initialisierungen — Fundamentale Variable

- Initialisierung:

```
int main ()
{
  int i = 5;
  ...
}
```

- Zuweisung:

```
int main ()
{
  int i;
  i = 5;
  ...
}
```


Initialisierungen — Fundamentale Variable

- In C++ werden auch bei der Initialisierung von Objekten alle erforderlichen automatischen Typkonversionen vorgenommen;

  `int i = 3.14;`

  ist daher kein Fehler, sondern identisch zu

  `int i = 3;`.

Ausdrücke und Befehle

- Initialisierungen
  - Allgemeines
  - Fundamentale Variable
  - **Felder; Klassen-Typen ohne Konstruktoren**
  - Klassen-Typen mit Konstruktoren
  - Referenzen
  - Programmverzweigungen und Initialisierungen

Initialisierungen — Felder; einfache Klassen

- Initialisierung mit einer Liste von Anfangswerten für alle Elemente des Objekts für:
  - Felder;
  - Klassen-Typen (Strukturen oder Klassen) ohne:
    - Konstruktoren;
    - Elemente, die nicht `public` sind;
    - Basisklassen und virtuelle Funktionen.

Initialisierungen — Felder; einfache Klassen

- Die Liste darf auch *weniger* Anfangswerte als Elemente im Objekt enthalten.
- Die nicht explizit initialisierten Elemente werden auf "`0`" gesetzt.
- Es dürfen jedoch nicht *mehr* Anfangswerte als Elemente angegeben werden.

Initialisierungen — Felder; einfache Klassen

```
struct RCPrompt
{
  short nZeile;
  short nSpalte;
  char *szPrompt;
};
RCPrompt rcWeiter = { 24, 0, "Weiter (J/N)?" };
int nPrim[5] = { 1, 2, 3, 5, 7 };
// Für ein eindimensionales Feld mit Initialisierung
// braucht die Dimension nicht angegeben zu werden:
// int nPrim[] = { 1, 2, 3, 5, 7 };
```


Initialisierungen — Felder; einfache Klassen

- Mehrdimensionale Felder, Felder von Klassen, Felder innerhalb einer Klasse oder ineinander geschachtelte Klassen-Typen: Initialisierungsliste mit ineinander geschachtelten geschwungenen Klammern ("`{ }`").
- Innere "`{ }`" sind optional, fördern aber die Lesbarkeit!

Initialisierungen — Felder; einfache Klassen

```
struct RCPrompt
{
  short nZeile;
  short nSpalte;
  char *szPrompt;
};
RCPrompt Menu[] = {
  { 4, 7, "Vorhandene Optionen:" },
  { 5, 9, "1. Hauptmenü" },
  { 6, 9, "2. Druckerausgabe" },
  { 7, 9, "3. Programm beenden" }
};
```


Initialisierungen — Felder; einfache Klassen

- Felder vom Typ `char` initialisiert durch eine Liste von Zeichenkonstanten oder durch einen String:
  - `char abc[4] = {'a','b','c','d'};`
  - `char ABC[5] = "abcd";`
- In C++ auch Initialisierung für Felder und Strukturen mit automatischer Speicherklasse (also für nicht-statische Objekte)!
- Das folgende Beispiel ist daher in C++ korrekt, wäre aber in ANSI-C nicht zulässig:

Initialisierungen — Felder; einfache Klassen

```
#include <iostream.h>
int main ()
{
  int ai[5] = { 3, 5, 7, 11, 13 };
  char fox[] = "The quick brown fox\n";
  for (int i = 0; i < 5; i++)
    cout << "ai [" << i << "] = " << ai[i] << "\n";
  cout << fox;
}
```


Initialisierungen — Felder; einfache Klassen

- Initialisierung von `unions`:
  - Mit einer `union` gleichen Typs;
  - Mit einem Initialisierungs-Ausdruck in geschwungenen Klammern ("`{ }`") für das erste Element der `union`.

Ausdrücke und Befehle

- Initialisierungen
  - Allgemeines
  - Fundamentale Variable
  - Felder; Klassen-Typen ohne Konstruktoren
  - **Klassen-Typen mit Konstruktoren**
  - Referenzen
  - Programmverzweigungen und Initialisierungen

Initialisierungen — Klassen mit Konstruktoren

- *Konstruktoren* werden mit der Definition einer Klasse festgelegt.
- Ihr Name ist der Name der Klasse.
- *Überladen* von Funktionen (*Function Overloading*) erlaubt die Definition unterschiedlicher Konstruktoren, die sich durch Anzahl und/oder Type ihrer Argumente voneinander unterscheiden müssen.
- Wenn für eine Klasse mindestens ein Konstruktor definiert wurde, sollten auch ein *Default*-Konstruktor und ein *Kopier*-Konstruktor definiert werden.
- Der Compiler generiert bei Bedarf rudimentäre Default- und Kopier-Konstruktoren.

Initialisierungen — Klassen mit Konstruktoren

- Ein *Default-Konstruktor* hat *keine Argumente*; er erzeugt ein Standard-Objekt der Klasse.
- Ein *Kopier-Konstruktor* hat *ein einziges Argument*, dessen Typ eine *Referenz auf die selbe Klassen-Type* sein muss. Er dient dazu, Klassen-Objekte zu kopieren.
- C++ prüft nicht, *was* ein bestimmter Konstruktor *wirklich* tut. Es sind daher auch zusätzliche oder abweichende Funktionsmechanismen möglich.
- Vorsicht bei der Initialisierung globaler Klassen-Objekte!

Initialisierungen — Klassen mit Konstruktoren

```
// Datei point.h (Definition der Klasse Point)
class Point
{
public:
  Point () { _x = 13; _y = 17; }
    // Default-Konstruktor (#1)
  Point (const Point& pt){_x = 2*pt._x; _y = 2*pt._y;}
    // Kopier-Konstruktor (#2)
  Point (short x, short y) { _x = x; _y = y; };
    // Konstruktor #3
  Show ()
    { cout << "x = " << _x << ", y = " << _y << "\n";}
private:
   short _x, _y;
};
```

Initialisierungen — Klassen mit Konstruktoren

```
// Datei A.cpp (Übersetzungseinheit A)
#include <iostream.h>
#include "point.h"
void func ();     // Vorwärts-Deklaration
extern Point pt1; // HIER kein Konstruktor-Aufruf!
Point pt2;        // Konstruktor #1
Point pt3 (3, 5); // Konstruktor #3
```


Initialisierungen — Klassen mit Konstruktoren

```
// Datei A.cpp (Übersetzungseinheit A) - Fortsetzung
int main ()
{
  Point pt4;           // Konstruktor #1
  Point pt5 (7, 11);   // Konstruktor #3
  pt1.Show (); pt2.Show (); pt3.Show ();
  pt4.Show (); pt5.Show ();
  pt4 = pt3;       // bitweise Kopie, KEIN Konstruktor
  Point pt6 = pt3;     // Konstruktor #2
  pt4.Show (); pt6.Show ();
  func ();         // Aufruf der Funktion aus B.cpp
}
```


Initialisierungen — Klassen mit Konstruktoren

```
// Datei B.cpp (Übersetzungseinheit B)
#include <iostream.h>
#include "point.h"
Point pt1 (23, 29);     // Konstruktor #3
Point pt2 (31, 37);     // Konstruktor #3
void func (void)
{
    cout << "*****************\n";
    pt1.Show ();
    pt2.Show ();
}
```


Initialisierungen — Klassen mit Konstruktoren

- Das Programm erzeugt die Ausgabe:

```
x = 23, y = 29        (pt1 — aus #3, B.cpp)
x = 13, y = 17        (pt2 — aus #1, A.cpp)
x = 3, y = 5          (pt3 — aus #3, A.cpp)
x = 13, y = 17        (pt4 — aus #1, A.cpp)
x = 7, y = 11         (pt5 — aus #3, A.cpp)
x = 3, y = 5          (pt4 — Kopie von pt3)
x = 6, y = 10         (pt6 — aus #2, A.cpp)
*****************
x = 23, y = 29        (pt1 — aus #3, B.cpp)
x = 13, y = 17        (pt2 — aus #1, A.cpp)
```


Initialisierungen — Klassen mit Konstruktoren

- Eine Initialisierung wie bei fundamentalen Variablen ist nur dann (mit dem Kopierkonstruktor) möglich, wenn *genau ein* Wert für die Initialisierung benötigt wird.
- Dann sind die "klassische" Initialisierung (mit "=") und die Initialisierung mit funktionsartiger Syntax ("( )") äquivalent:

Initialisierungen — Klassen mit Konstruktoren

```
#include <iostream.h>
class Konto
{
public:
  Konto () { Stand = 0; }
    // Default-Konstruktor (#1)
  Konto (const double& s) { Stand = s; }
    // Kopier-Konstruktor (#2)
  Konto (double s) { Stand = s; };
    // Konstruktor #3
  Show () { cout << "Kontostand: " << Stand << "\n"; }
private:
  double Stand;
};
```

Initialisierungen — Klassen mit Konstruktoren

```
Konto k1;                // Konstruktor #1
Konto k2 = 123.45;       // Konstruktor #2
Konto k3 (234.56);       // Konstruktor #3

int main ()
{
  Konto k4;              // Konstruktor #1
  Konto k5 = 654.32;     // Konstruktor #2
  Konto k6 (765.43);     // Konstruktor #3

  k1.Show (); k2.Show (); k3.Show ();
  k4.Show (); k5.Show (); k6.Show ();
}
```


Ausdrücke und Befehle

- Initialisierungen
  - Allgemeines
  - Fundamentale Variable
  - Felder; Klassen-Typen ohne Konstruktoren
  - Klassen-Typen mit Konstruktoren
  - **Referenzen**
  - Programmverzweigungen und Initialisierungen

Initialisierungen — Referenzen

- Variable vom Referenz-Typ müssen bei ihrer Definition mit einem Objekt initialisiert werden, dessen Type
  - der Basistype der Referenz entspricht, oder
  - sich in die Type der Referenz umwandeln lässt.
- Bei Typumwandlung muss der Compiler das ursprüngliche Objekt in einem separaten internen Hilfsobjekt speichern.
- In diesem Fall ist die Initialisierung einer Referenz-Variablen nur mit dem Attribut `const` möglich:

Initialisierungen — Referenzen

```
#include <iostream.h>

short sVar = 3;
long lVar = 5;

int main ()
{
  short& sRef1 = sVar;        // "normale" Referenzen
  long& lRef1 = lVar;
  const short& sRef2 = lVar;  // Referenzen auf
  const long& lRef2 = sVar;   // Hilfsobjekte

  cout << "sVar = " << sVar << ", lVar = " << lVar << "\n";
  cout << "sRef1 = " << sRef1 << ", lRef1 = " << lRef1 <<
   ", sRef2 = " << sRef2 << ", lRef2 = " << lRef2 << "\n";
```


Initialisierungen — Referenzen

```
  // Fortsetzung...
  sRef1++;                  // inkrementiere sVar
  lRef1++;                  // inkrementiere lVar
                  // sRef2 und lRef2 dürfen wegen
                  // "const" nicht geändert werden
  cout << "sVar = " << sVar << ", lVar = " << lVar << "\n";
  cout << "sRef1 = " << sRef1 << ", lRef1 = " << lRef1 <<
   ", sRef2 = " << sRef2 << ", lRef2 = " << lRef2 << "\n";
}
```

- Die Ausgabe des Programms lautet:

```
sVar = 3, lVar = 5
sRef1 = 3, lRef1 = 5, sRef2 = 5, lRef2 = 3
sVar = 4, lVar = 6
sRef1 = 4, lRef1 = 6, sRef2 = 5, lRef2 = 3
```


Initialisierungen — Referenzen

- Konstante Werte, die an eine *Funktion mit Referenztyp-Argumenten* übergeben werden, werden zunächst in eine interne Hilfsvariable kopiert, deren Adresse dann an die Funktion übergeben wird.

Ausdrücke und Befehle

- Initialisierungen
  - Allgemeines
  - Fundamentale Variable
  - Felder; Klassen-Typen ohne Konstruktoren
  - Klassen-Typen mit Konstruktoren
  - Referenzen
  - **Programmverzweigungen und Initialisierungen**

Programmverzweigungen und Initialisierungen

- Bei der Definition von Objekten ist sicher zu stellen, dass eine Initialisierung immer dann erfolgt, wenn tatsächlich ein Zugriff auf das Objekt stattfindet.
- Im folgenden Beispiel kann unter bestimmten Voraussetzungen eine Variable wohl definiert, jedoch nicht initialisiert sein:

Programmverzweigungen und Initialisierungen

```
#include <iostream.h>
void func (int param)
{
  int i;
  if (param)
    i = 7;
  cout << "i = " << i << "\n";
}
int main ()
{
  func (0);
  func (1);
}
```


Ausdrücke und Befehle

- Initialisierungen
- ***L-Values* und *R-Values***
- Funktionen
- Operatoren
- Befehle (*Statements*)

L-Values und *R-Values*

- Voraussetzungen für Zuweisung des Ergebnisses eines Ausdrucks an einen anderen Ausdruck:
  - Veränderung des Wertes des zweiten Ausdrucks muss physikalisch und semantisch *möglich* sein;
  - Es muss sich also um eine *Variable* oder ein *nicht-konstantes Objekt* handeln.
- Nur solche Ausdrücke dürfen auf der *linken* Seite eines Zuweisungsoperators (daher "*L-Value*") vorkommen, die eine *Variable* oder ein *nichtkonstantes Objekt* beschreiben und die eine andere Type als `void` haben.

L-Values und *R-Values*

- Alle anderen Ausdrücke, insbesondere Konstanten, die (nur) auf der *rechten* Seite eines Zuweisungsoperators vorkommen können, werden gelegentlich als "*R-Values*" bezeichnet.
- Alle *L-Values* sind auch *R-Values*, aber nicht alle *R-Values* sind *L-Values*.

L-Values und *R-Values*

Ausdruck:	***L-Value:***
Variable, Element eines Objekts	ja
`const` Variable oder Objekt	nein
Funktion mit Nicht-Referenztyp	nein
Referenz-Typen (auch Funktionen)	ja
arithmetische oder logische Ausdrücke	nein
Ausdrücke mit dem Operator für bedingte Programmausführung	ja

Ausdrücke und Befehle

- Initialisierungen
- *L-Values* und *R-Values*
- **Funktionen**
- Operatoren
- Befehle (*Statements*)

Ausdrücke und Befehle

- Funktionen
  - **Formale und aktuelle Argumente**
  - Deklaration und Definition
  - *Function Overloading*
  - Funktionen, Makros und Nebenwirkungen

Formale und aktuelle Argumente

- Funktionen können beliebig viele *Argumente* (oder *Parameter*) haben, die innerhalb der Funktions-klammern, getrennt durch Kommas (","), stehen:
  - *Formale Argumente:* Argumente einer Funktion bei der Deklaration und Definition;
  - *Aktuelle Argumente:* Ausdrücke, die beim *Aufruf* der Funktion an ihrer Stelle stehen.

Formale und aktuelle Argumente

```
#include <iostream.h>
int myfunc (int nP, int *npP, int& nrP);
     // Deklaration ("Prototyp") - ";" am Ende

int main ()
{
  int i = 1; int j = 2; int k = 3; int l;
  l = myfunc (i, &j, k);
            // i, &j und k sind aktuelle Argumente
  cout << "i = " << i << ", j = " << j << ", k = "
    << k << ", l = " << l << "\n";
  return 0;
}
```


Formale und aktuelle Argumente

```
// Definition ("Körper") - "{ }"
// nP, npP und nrP sind formale Argumente
int myfunc (int nP, int *npP, int& nrP)
{
  nP++;                 // inkrementiere die Kopie
  (*npP)++;             // inkrementiere das Objekt
  nrP++;                // inkrementiere das Objekt
  cout << "nP = " << nP << ", *npP = " << *npP
    << ", nrP = " << nrP << "\n";
  return *npP++;
  // gib Wert des Objekts zurück, inkrementiere Zeiger
}
```

Formale und aktuelle Argumente

- Beim Aufruf einer Funktion:
  - *Aktuelle* Argumente werden ausgewertet;
  - Die korrespondierenden *formalen* Argumente werden entsprechend den Ergebnissen (unter Berücksichtigung von Typkonversionen) initialisiert.
- In C++ keine festgelegte Reihenfolge, in der die Argumente ausgewertet werden:

Formale und aktuelle Argumente

```
void func (int k, int l, int m);
int i = 1;
...
func (i++, i++, i++);
// Auswertung von rechts nach links:
// m => 1, l => 2, k => 3, i (nach Aufruf) => 4
// Auswertung von links nach rechts:
// k => 1, l => 2, m => 3, i (nach Aufruf) => 4
```


Formale und aktuelle Argumente

- Alle Argumente werden ausgewertet und alle Nebenwirkungen der Auswertung abgeschlossen, bevor der Eintritt in die Funktion erfolgt.
- An die Funktion werden *Kopien* der aktuellen Argumente übergeben (Ausnahme: Argumente vom Referenz-Typ).
- Die Funktion kann den Wert eines Arguments beliebig verändern, ohne dass dies Rückwirkungen auf die aufrufende Funktion hat.
- Objekte der aufrufenden Funktion können über Zeiger- oder Referenz-Argumente beeinflusst werden.

Ausdrücke und Befehle

- Funktionen
  - Formale und aktuelle Argumente
  - **Deklaration und Definition**
  - *Function Overloading*
  - Funktionen, Makros und Nebenwirkungen

Ausdrücke und Befehle

- Funktionen
  - Deklaration und Definition
    - **Allgemeines**
    - Funktionen ohne Argumente
    - Funktionen mit variabler Argumente-Zahl

Deklaration und Definition — Allgemeines

- Die *Definition* einer Funktion erfolgt durch die Angabe ihres *Funktionskörpers*, also des eigentlichen Programmcodes, der die Funktion ausmacht.
- Zum Zeitpunkt des Aufrufes einer Funktion muss diese zwar *deklariert*, aber nicht unbedingt *definiert* sein (Bibliotheksfunktionen!).
- *Deklaration* einer Funktion über *Funktions-Prototypen*, die oft in *Header-Dateien* zusammengefasst und über `#include`-Befehle in die Übersetzungseinheit eingebunden werden.

Deklaration und Definition — Allgemeines

- In der *Deklaration* (*nicht* in der *Definition*) einer Funktion sind *abstrakte Deklaratoren* zulässig (d.h. Typen-bezeichnungen ohne Objektnamen).
- Die folgenden beiden Prototypen sind daher zulässig und äquivalent:

```
int myfunc (int nP, int *npP, int& nrP);
int myfunc (int, int *, int&);
```

- Die Verwendung deskriptiver Namen für die formalen Argumente in der Deklaration fördert die Lesbarkeit des Programmcodes.

Ausdrücke und Befehle

- Funktionen
  - Deklaration und Definition
    - Allgemeines
    - **Funktionen ohne Argumente**
    - Funktionen mit variabler Argumente-Zahl

Funktionen ohne Argumente

- Funktionen, die keine Argumente benötigen, können folgendermaßen deklariert und definiert werden:

```
int func1 (void);     oder    int func2 ();
int func1 (void)              int func2 ()
{                             {
     ...                         ...
}                             }
```

- Das Schlüsselwort `void` darf nur als einziges formales Argument in der Deklaration oder Definition einer Funktion vorkommen.

Funktionen ohne Argumente

- Der Aufruf muss in jedem Fall in der Form

```
resultat1 = func1();
resultat2 = func2();
```

  erfolgen.
- Die Zuweisung des Ergebnisses einer Funktion ist optional.

Ausdrücke und Befehle

- Funktionen
  - Deklaration und Definition
    - Allgemeines
    - Funktionen ohne Argumente
    - **Funktionen mit variabler Argumente-Zahl**

Funktionen mit variabler Argumente-Zahl

- Zwei Möglichkeiten in C++ für Übergabe einer beliebige Anzahl von Argumenten:
  - Standard-C-Methode: Auslassungspunkte ("`...`")
  - Voreingestellte Argumente

Funktionen mit variabler Argumente-Zahl

- Standard-C-Methode:
  - Auslassungspunkte am Ende einer Argumentliste geben an, dass weitere Argumente mit beliebigen Typen folgen *können*:

```
extern "C" int printf (const char *, ...);
```

  - Eine derartige Deklaration unterläuft die Typen-prüfung von C++ und ist daher möglichst zu vermeiden.

Funktionen mit variabler Argumente-Zahl

- Standard-C-Methode:
  - Die weiteren Argumente werden den folgenden automatischen Typenumwandlungen unterworfen:
    - `float` ↛ `double`;
    - `char`, `short`, Aufzählungen, Bitfelder ↛ `int` oder `unsigned int`;
    - Klassen-Objekte (`struct`, `union`, `class`) werden binär kopiert als Datenstrukturen übergeben.

Funktionen mit variabler Argumente-Zahl

- Voreingestellte Argumente:
  - Funktionsargumente, für die fast immer die gleichen Werte verwendet werden, können bei der Deklaration einer Funktion mit einer Konstanten *voreingestellt* werden.
  - Dieser Wert wird bei Funktionsaufrufen dann verwendet, wenn kein aktuelles Argument übergeben wurde:

Funktionen mit variabler Argumente-Zahl

```
#include <iostream.h>
void showval (int i, int j = 10, int k = 20);
int main ()
{
  showval ( 1, 2, 3 );
  showval ( 4, 5 );
  showval ( 6 );
}
void showval (int i, int j, int k)
{
  cout << "i = " << i << ", j = " << j << ", k = " <<
  k << "\n";
}
```


Funktionen mit variabler Argumente-Zahl

- Für voreingestellte Argumente gelten die folgenden Regeln:
  - Sie müssen immer die letzten Elemente der Argumentliste sein.
  - Sie dürfen nur genau einmal in einer Übersetzungs-einheit definiert werden. Jede nochmalige Definition (auch mit dem selben Wert) ist ein Fehler.
- Voreingestellte Argumente sind auch bei der Deklaration von Zeigern auf Funktionen zulässig:

```
int (*fpFunc) (int i = 0);
```


Ausdrücke und Befehle

- Funktionen
  - Formale und aktuelle Argumente
  - Deklaration und Definition
  - ***Function Overloading***
  - Funktionen, Makros und Nebenwirkungen

Function Overloading

- *Function Overloading* (*Überladen* von Funktionen): Definition von Funktionen mit *gleichen* Namen, aber *unterschiedlichen* Argumentsätzen — *ein einziger* Name für mehrere Funktionen unabhängig von der Zahl und Type der Argumente.
- Interne Unterscheidung zwischen den für die unterschiedlichen Argumentsätze benötigten Funktionen durch *Dekorieren* des Funktionsnamens im compilergenerierten *Object*-Code.
- Beispiel: Ausgabe von *Strings*, ganzen Zahlen und Gleitkommazahlen mit einer generischen Funktion `print`:

Function Overloading

```
#include <stdio.h>          // C-I/O-Funktionen

// Separate Deklarationen
void print (char *);        // String-Ausgabe
void print (int);           // int-Ausgabe
void print (double);        // double-Ausgabe

int main ()
{
  print ("Hello, world!");
  print (4711);
  print (3.141592);
}
```


Function Overloading

```
// print-Funktion für Strings:
void print (char *buffer) { puts (buffer); }
                                // String-Ausgabe
// print-Funktion für ganze Zahlen
void print (int i) { printf ("%i\n", i); }
                                // formatierte Ausgabe
// print-Funktion für Gleitkomma-Zahlen
void print (double x) { printf ("%lf\n", x); }
                                // formatierte Ausgabe
```


Function Overloading

- Der C++-Compiler erzeugt z.B. folgende *Object*-Namen für die drei `print`-Funktionen:
  - `_print__FPc` für `void print (char *);`
  - `_print__Fi` für `void print (int);`
  - `_print__Fd` für `void print (double);`

Function Overloading

- Funktionen, die *Function Overloading* verwenden, müssen sich in mindestens einer der folgenden Charakteristiken voneinander unterscheiden:
  - Anzahl der Argumente;
  - Typen der Argumente;
  - Vorhandensein oder Fehlen von Auslassungspunkten ("...");

Function Overloading

- Die folgenden Charakteristika können *nicht* für die Unterscheidung von Funktionen bei *Function Overloading* verwendet werden:
  - Typ des Resultats der Funktion;
  - Verwendung von mit `typedef` definierten Namen, die gleiche Basistypen ergeben;
  - Unspezifizierte Feldgrößen.

Ausdrücke und Befehle

- Funktionen
  - Formale und aktuelle Argumente
  - Deklaration und Definition
  - *Function Overloading*
  - **Funktionen, Makros und Nebenwirkungen**

Funktionen, Makros und Nebenwirkungen

- *Nebenwirkungen* bei der Ermittlung der aktuellen Argumente einer Funktion durch Veränderung des Wertes von Variablen der aufrufenden Funktion oder Einlesen neuer Daten:

Funktionen, Makros und Nebenwirkungen

```
#include <conio.h>
int func1 (int);
char func2 (char);
int main ()
{
  int i = 1, j;
  char ch;
  j = func1 (i++);
  ch = func2 (getch());
}
```


Funktionen, Makros und Nebenwirkungen

- Beim Aufruf von *Funktionen*: jedes aktuelle Argument wird *genau einmal* ausgewertet, d.h., `i` wird *genau einmal* inkrementiert, und `getch()` *genau einmal* aufgerufen.
- Dies gilt auch für `inline`-Funktionen.
- Beim Aufruf von *Makros*: Argumente des Makros können auch öfter als einmal oder überhaupt nicht ausgewertet werden.

Funktionen, Makros und Nebenwirkungen

- Beispiel:

```
// Makro für die Konversion von Klein- in
// Großbuchstaben; Original-Definition:
#define toupper(c) ((islower(c)) ? \
  ((c)-'a'+'A') : (c))
```


Funktionen, Makros und Nebenwirkungen

- Beim Aufruf eines *Makros*: Alle Stellen, wo eines seiner *formalen* Argumente vorkommt (hier "`(c)`"), werden durch das *aktuelle* Argument ersetzt. Der Aufruf

  ```
  char ch = toupper (getch());
  ```

  wird daher umgesetzt in

  ```
  ch = ((islower(getch())) ? ((getch())-
    'a'+'A') : (getch()))
  ```

- `getch()` wird *zweimal* aufgerufen. Statt *eines* Zeichens fordert das Programm also *zwei* an.

Funktionen, Makros und Nebenwirkungen

- Modernere Definitionen von `toupper(c)` vermeiden dieses Problem:

```
#define toupper(c)  ({int _c=(c); \
  islower(_c) ? (_c-'a'+'A') : _c; })
```

- Die beschriebenen Nebenwirkungen unterbleiben, wenn `inline`-Funktionen an Stelle von Makros verwendet werden.

Ausdrücke und Befehle

- Initialisierungen
- *L-Values* und *R-Values*
- Funktionen
- **Operatoren**
- Befehle (*Statements*)

Ausdrücke und Befehle

- Operatoren
  - **Arithmetische Operatoren**
  - Vergleichs-Operatoren
  - Bitweise und logische Operatoren
  - Zuweisungs-Operatoren
  - Der Operator für bedingte Ausführung
  - Die Operatoren `new` und `delete`
  - Präzedenz von Operatoren

Ausdrücke und Befehle

- Operatoren
  - Arithmetische Operatoren
    - **Inkrement- und Dekrement-Operatoren**
    - Verschiebungs-Operatoren

Inkrement- und Dekrement-Operatoren

- *Inkrement*-Operatoren bewirken die *Erhöhung* des Wertes der Variablen, auf die sie angewendet werden, um eine Einheit der entsprechenden Type.
- *Dekrement*-Operatoren bewirken eine *Verringerung* um eine Einheit.
- Eine Einheit muss nicht immer der Wert 1 sein; der Wert eines Zeigers auf `long` ändert sich beispielsweise um 4, weil `sizeof(long) == 4`.

Inkrement- und Dekrement-Operatoren

- In C/C++ existieren zwei Typen von Inkrement- und Dekrement-Operatoren mit unterschiedlicher Funktion und Präzedenz (*Operator Precedence*):
  - Postfix-Inkrement und -Dekrement;
  - Präfix-Inkrement und -Dekrement.

Inkrement- und Dekrement-Operatoren

- Postfix-Inkrement und -Dekrement:
  - Verändern den Wert einer Variablen, *nachdem* deren *ursprünglicher* Wert in einer allfälligen anderen Operation verwendet wurde:

```
int i = 1, j;
j = i++;   // j wird auf 1 gesetzt
           // i ist jetzt 2
```

- Die Postfix-Inkrement- und Dekrement-Operatoren haben die höchste Präzedenz von allen arithmetischen Operatoren.

Inkrement- und Dekrement-Operatoren

- Präfix-Inkrement und -Dekrement:
  - Verändern den Wert einer Variablen, *bevor* deren Wert in einer allfälligen anderen Operation verwendet wird:

```
int i = 1, j;
j = ++i;   // i ist jetzt 2
           // j wird auf 2 gesetzt
```


Ausdrücke und Befehle

- Operatoren
  - Arithmetische Operatoren
    - Inkrement- und Dekrement-Operatoren
    - **Verschiebungs-Operatoren**

Verschiebungs-Operatoren

- Die Operatoren "<<" und ">>" bewirken eine bitweise Verschiebung des linken Operanden um die im rechten Operanden angegebene Anzahl von Bits nach links bzw. rechts.
- Beide Operanden müssen ganzzahlig sein.
- Negative rechte Operanden oder rechte Operanden größer als die Bitzahl des linken Operanden haben undefinierte Ergebnisse zur Folge.

Verschiebungs-Operatoren

- Bei der *Links*verschiebung mit "<<" werden die freiwerdenden niederwertigen Bits mit "0" aufgefüllt. Eine Linksverschiebung um *n* Bits entspricht einer Multiplikation mit 2^n.
- Bei der *Rechts*verschiebung mit ">>" werden die freiwerdenden Bits je nach dem Typ des linken Operanden aufgefüllt:
  - Bei `unsigned` Operanden werden sie immer mit "0" aufgefüllt (*logische Verschiebung*);
  - Bei `signed` linken Operanden wird der Wert des höchstwertigen (äußerst linken) Bits dupliziert (*arithmetische Verschiebung*).

Verschiebungs-Operatoren

- Damit entspricht eine Rechtsverschiebung um *n* Bits bei CPUs, die mit Zweier-Komplement-Darstellung negativer Zahlen arbeiten, in allen Fällen einer Division durch 2^n.
- Bits, die aus dem Bereich des linken Operanden herausgeschoben werden, sind verloren; ihr Wert hat keinen Einfluss auf nachfolgende Operationen.

Ausdrücke und Befehle

- Operatoren
  - Arithmetische Operatoren
  - **Vergleichs-Operatoren**
  - Bitweise und logische Operatoren
  - Zuweisungs-Operatoren
  - Der Operator für bedingte Ausführung
  - Die Operatoren new und delete
  - Präzedenz von Operatoren

Vergleichs-Operatoren

- Die Operatoren "<", "<=", ">", ">=", "==" und "!=" dienen zum Vergleich ihrer linken und rechten Operanden.
- Beide Operanden werden nach den in C++ üblichen Regeln in kompatible Typen umgewandelt, bevor der Vergleich durchgeführt wird.
- Das Ergebnis einer Vergleichs-Operation ist `1`, wenn der Vergleichs-Ausdruck eine wahre Aussage ist; ansonsten ist es `0`.
- Das Ergebnis der Vergleichs-Operation ist immer vom Typ `int`. Häufig wird für die Manipulation logischer Aussagen eine Type `BOOL` definiert:

```
typedef int BOOL;
```


Ausdrücke und Befehle

- Operatoren
  - Arithmetische Operatoren
  - Vergleichs-Operatoren
  - **Bitweise und logische Operatoren**
  - Zuweisungs-Operatoren
  - Der Operator für bedingte Ausführung
  - Die Operatoren new und delete
  - Präzedenz von Operatoren

Bitweise und logische Operatoren

- Die *bitweisen* Operatoren ("~" — NOT, "&" — AND, "^" — EX-OR und "|" — OR) beeinflussen *alle* Bits ihrer Operanden.
- Die *binären* Operatoren "&", "^" und "|" verknüpfen jedes Bit ihres ersten Operanden mit dem korrespondierenden Bit des zweiten Operanden.
- Die bitweise Negation "~" invertiert *alle* Bits.

Bitweise und logische Operatoren

- Es gilt daher:

```
~0x5a5a == 0xa5a5;
0x5555 & 0x00ff == 0x0055;
0x5555 ^ 0x00ff == 0x55aa;
0x5555 | 0x00ff == 0x55ff;
```


Bitweise und logische Operatoren

- Die *logischen* Operatoren ("!" — NOT, "&&" — AND und "||" — OR) interpretieren ihre Operanden als Boole'sche Variable
- Dabei werden alle Werte ungleich `0` als TRUE und der Wert `0` als FALSE interpretiert.
- Es gilt daher:

```
!0x5a5a == 0;
0x5555 && 0x00ff == 1;
0x5555 || 0x00ff == 1;
```

Ausdrücke und Befehle

- Operatoren
  - Arithmetische Operatoren
  - Vergleichs-Operatoren
  - Bitweise und logische Operatoren
  - **Zuweisungs-Operatoren**
  - Der Operator für bedingte Ausführung
  - Die Operatoren `new` und `delete`
  - Präzedenz von Operatoren

Zuweisungs-Operatoren

- Zuweisungs-Operatoren ("=" sowie "*op*=") weisen ihrem linken Operanden das Ergebnis eines Ausdrucks zu.
- Der Wert eines Ausdrucks mit dem Zuweisungs-Operator ist identisch mit dem Wert des linken Operanden *nach der Zuweisung*.
- Der linke Operand einer Zuweisungs-operation muss immer ein *L-Value* sein. Der folgende Ausdruck ist daher illegal:

```
(a += b) += c;      // Fehler: (a += b)
                    // ist kein L-Value
a += (b += c);      // Ok
```


Zuweisungs-Operatoren

- Die Verbund-Zuweisungsoperatoren "*op*=" weisen dem linken Operanden das Ergebnis der mit dem Operator *op* und dem rechten Operanden durchgeführten Operation zu.
- *op* kann jeder binäre arithmetische und bitweise Operator sein.
- Ein Ausdruck der Form

  `a` *op*`= b;`

  ist äquivalent zu

  `a = a` *op* `b;`

Ausdrücke und Befehle

- Operatoren
  - Arithmetische Operatoren
  - Vergleichs-Operatoren
  - Bitweise und logische Operatoren
  - Zuweisungs-Operatoren
  - **Der Operator für bedingte Ausführung**
  - Die Operatoren `new` und `delete`
  - Präzedenz von Operatoren

Der Operator für bedingte Ausführung

- Der Operator "`? :`" ist der einzige *ternäre* Operator in C/C++.
- Der erste Operand (vor dem "?") wird ausgewertet, und alle Nebenwirkungen dieses Ausdrucks werden abgeschlossen.
- Wenn das Ergebnis des ersten Operanden ein von Null verschiedener Wert war, wird der zweite Operand ausgewertet.
- Wenn das Ergebnis des ersten Operanden Null war, wird der dritte Operand ausgewertet.
- Das *Ergebnis* eines bedingten Ausdrucks ist das Ergebnis jenes Operanden, der ausgewertet wurde.

Der Operator für bedingte Ausführung

- Die folgenden zwei Ausdrücke sind daher äquivalent:

```
x = (i > 0) ? 5 : 3;
i > 0 ? (x = 5) : (x = 3);
   // Klammern um "(x = ...)" wegen
   // Operator Precedence!
```

- Der erste Operand muss von einer ganzzahligen oder Zeiger-Type sein; der zweite und dritte Operand müssen kompatible Typen haben.
- Alle in der nachfolgenden Tabelle *nicht* angeführten Kombinationen sind illegal:

Der Operator für bedingte Ausführung

Type 1:	Type 2:	Resultat:
gleiche Type		gleiche Type
arithmetische Type		Type aus Standard-Konversion
Zeiger	Zeiger, `NULL`	Type aus Zeigerkonversion
Referenz-Type		Type aus Referenz-Konversion
void	void	void
gleiche Klasse		gleiche Klasse

Der Operator für bedingte Ausführung

- Im Gegensatz zu ANSI-C sind Ausdrücke mit dem Operator für die bedingte Ausführung in C++ *L-Values*.
- Der folgende Ausdruck wird in C++, nicht aber in ANSI-C akzeptiert:

```
((a < 5)? b : c) = 9;
  // äquivalent zu:
  // a < 5 ? (b = 9) : (c = 9);
```


Ausdrücke und Befehle

- Operatoren
  - Arithmetische Operatoren
  - Vergleichs-Operatoren
  - Bitweise und logische Operatoren
  - Zuweisungs-Operatoren
  - Der Operator für bedingte Ausführung
  - **Die Operatoren new und `delete`**
  - Präzedenz von Operatoren

Ausdrücke und Befehle

- Operatoren
  - Die Operatoren new und `delete`
    - **Allgemeines**
    - Der Operator new
    - Der Operator `delete`
    - Vergleich mit `malloc()` und `free()`

Die Operatoren new und `delete`

- Die Operatoren new und `delete` wurden in C++ neu eingeführt.
- Sie dienen zur dynamischen Verwaltung von Speicherblöcken aus dem "*Free Store*" (in C : "*Heap*").
- Sie ersetzen damit die in C verwendeten Bibliotheksfunktionen `malloc()` bzw. `free()`.

Ausdrücke und Befehle

- Operatoren
  - Die Operatoren new und `delete`
    - Allgemeines
    - **Der Operator new**
    - Der Operator `delete`
    - Vergleich mit `malloc()` und `free()`

Der Operator new

- new bezieht einen Speicherblock aus dem *Free Store*, dessen Größe durch die Größe der Operand-Type bestimmt ist, und gibt als Ergebnis einen Zeiger auf diesen Block zurück, dessen Type durch die Operand-Type festgelegt ist.
- Wenn new verwendet wird, um Speicherplatz für eine Nicht-Feld-Type zu erhalten, ist das Ergebnis von new ein Zeiger auf diese Type:

```
double *pdVariable = new double;
class A;
A *pA = new A;
```


Der Operator new

- Wenn mit new Speicherplatz für ein eindimensionales Feld bezogen wird, ist das Ergebnis ein Zeiger auf das erste Element des Feldes:

```
char *sBuffer = new char[80];
```

- Wenn unzureichender Speicherplatz vorhanden ist, ist das Resultat von new NULL:

Der Operator new

```
#include <iostream.h>
int main ()
{
  int *pi = new int[1000000];
  if (! pi)      // oder: if (pi == 0)
  {
    cout << "Zu wenig Speicher\n"
    return 1;    // Programm beenden mit Fehlercode
  1
  }
  ...
}
```


Der Operator new

- Mit new erstellte Objekte bleiben bis zum Aufruf von delete für das betreffende Objekt bzw. bis zum Programmende bestehen.
- Das Ergebnis von new muss daher einer Zeigervariablen von ausreichender Lebensdauer zugewiesen werden.
- Mit new erstellte Objekte können gleichzeitig initialisiert werden.
- Für die Initialisierung können beliebige Ausdrücke (nicht nur Konstanten) verwendet werden.

Der Operator new

- Für Klassen-Objekte wird ein für die Initialisierung geeigneter Konstruktor aufgerufen.
- Die Initialisierung von Klassen-Objekten mit Konstruktor ist mit new nur möglich, wenn eine der Bedingungen erfüllt ist:
  - Die Type und Anzahl der Argumente der Initialisierung stimmt mit denen eines Konstruktors überein; oder
  - Die Klasse besitzt einen Default-Konstruktor (ohne Argumente).

Der Operator new

```
#include <iostream.h>
class POINT
{
public:
  POINT (void) { _x = 0; _y = 0; }
    // Default-Konstruktor ohne Argumente
  POINT (short x, short y) { _x = x; _y = y; }
    // Overloaded Konstruktor mit Argumenten
  Zeige (void)
  { cout << "x = " << _x << ", y = " << _y << "\n"; }
private:
  short _x, _y;
};
```

Der Operator new

```
int main ()
{
  int *i = new int (13);  // Initialisierung auf 13
  cout << "i = " << *i << "\n";
  POINT *pp1 = new POINT; // Zeiger auf POINT
  pp1->Zeige();           // Initialisierung auf (0,0)
  POINT p1;               // Objekt der Klasse POINT
  p1.Zeige();             // Initialisierung auf (0,0)
  POINT *pp2 = new POINT (11, 13);  // Zeiger
  pp2->Zeige();        // Initialisierung auf (11, 13)
  POINT p2(17, 19);                 // Objekt
  p2.Zeige();          // Initialisierung auf (17, 19)
}
```


Der Operator new

- Die individuelle Initialisierung von Elementen eines Feldes ist mit new nicht möglich; es wird nur (sofern vorhanden) der Default-Konstruktor aufgerufen.
- Die Reihenfolge ist nicht definiert, in der Initialisierungs-Ausdrücke ausgewertet werden.
- Wenn eine Zuordnung von Speicher aus dem *Free Store*-Bereich nicht möglich ist, kann auch keine Initialisierung erfolgen.
- Die Ausdrücke, die für die Initialisierung verwendet wurden, werden in diesem Fall nicht ausgeführt.

Der Operator new

- Wenn eine Klassen-Type eine `operator new`-Funktion definiert hat, wird für diese Type

  type`::operator new (sizeof (`*type*`))`

  aufgerufen.
- Die Standard-new-Funktion kann explizit mit

  `::operator new (sizeof (`*type*`))`

  aufgerufen werden.

Der Operator new

- Optional können benutzerdefinierte Klassentypen eine beliebige Anzahl weiterer Argumente (zusätzlich zu `sizeof(`*type*`)`) definieren.
- Der Aufruf einer solchen beispielsweise als

  type`::operator new (sizeof (`*type*`),`
  par1`, `*par2*`, `*par3*`)`

  definierten `operator new`-Funktion kann dann etwa so erfolgen:

  type `*p`*type*` = new (`*par1*`, `*par2*`, `*par3*`) `*type*`;`

Ausdrücke und Befehle

- Operatoren
  - Die Operatoren new und `delete`
    - Allgemeines
    - Der Operator new
    - **Der Operator `delete`**
    - Vergleich mit `malloc()` und `free()`

Der Operator `delete`

- `delete` stellt den Speicherbereich, der mit new für die Erzeugung von Objekten allokiert wurde, dem Programm wieder zur Verfügung.
- Für Felder, die mit new allokiert wurden, muss delete mit dem Operator "`[]`" (als "`delete[]`") aufgerufen werden.
- `delete` hat die Resultat-Type `void`; als Operand wird ein mit new generierter Zeiger benötigt.
- Das Ergebnis ist unbestimmt, wenn `delete` *nicht* auf einen derartigen Zeiger angewendet wird.
- `delete` darf aber auf einen Zeiger mit dem Wert `NULL` angewendet werden.

Der Operator `delete`

- Wenn `delete` mit einem Zeiger auf ein Objekt einer Klasse als Operand verwendet wird, für die ein Destruktor definiert wurde, wird dieser Destruktor aufgerufen, bevor der Speicher zurückgegeben wird.
- Eine eigene Definition von `operator delete` in einer Klasse ist möglich.
- Nachdem ein Zeiger als Operand für `delete` verwendet wurde, darf er nicht mehr anderweitig verwendet werden.

Der Operator `delete`

```
#include <iostream.h>
int main ()
{
  int *pi1 = new int (3);         // einfaches Objekt
  double *pd1 = new double [5];  // Feld von 5 doubles
  *pd1 = 1.1; *(pd1 + 1) = 2.2; *(pd1 + 2) = 3.3;
  *(pd1 + 3) = 4.4; *(pd1 + 4) = 5.5;
  cout << "pi1 = " << *pi1 << "\n";
  for (int i = 0; i < 5; i++)
    cout << "pd1[" << i << "] = " << pd1[i] << "\n";
  delete pi1;                      // *pi1 zerstören
  delete [] pd1;                   // Feld zerstören
```


Der Operator `delete`

```
  // Free Store-Speicher wird "recycled"
  int *pi2 = new int (17);
  double *pd2 = new double [5];
  *pd2 = 11.1; *(pd2 + 1) = 22.2; *(pd2 + 2) = 33.3;
  *(pd2 + 3) = 44.4; *(pd2 + 4) = 55.5;
  // das Folgende ist unzulässig:
  cout << "pi1 = " << *pi1 << "\n";
  for (i = 0; i < 5; i++)
    cout << "pd1[" << i << "] = " << pd1[i] << "\n";
}
```


Der Operator `delete`

- Das Programm erzeugt die Ausgabe:

```
pi1 = 3
pd1[0] = 1.1
pd1[1] = 2.2
pd1[2] = 3.3
pd1[3] = 4.4
pd1[4] = 5.5
pi1 = 17
pd1[0] = 11.1
pd1[1] = 22.2
pd1[2] = 33.3
pd1[3] = 44.4
pd1[4] = 55.5
```


Ausdrücke und Befehle

- Operatoren
  - Die Operatoren `new` und `delete`
    - Allgemeines
    - Der Operator `new`
    - Der Operator `delete`
    - **Vergleich mit `malloc()` und `free()`**

Vergleich mit `malloc()`und `free()`

- `new` und `delete` übernehmen in C++ die Aufgaben der C-Funktionen `malloc()` und `free()`.
- `malloc()` benötigt ein Argument vom Typ `size_t` (`typedef` als Synonym für `unsigned`) und gibt einen Zeiger vom Typ `void *` zurück.
- `free()` benötigt einen Zeiger auf einen mit `malloc()` allokierten Speicherblock oder einen Null-Zeiger.

Vergleich mit `malloc()` und `free()`

- Die wesentlichen Unterschiede zwischen den C-Funktionen und den C++-Operatoren sind:
  - `new` und `delete` sind integraler Bestandteil der Sprache.
  - Die Type des Ergebnisses von `new` steht fest, während `malloc()` immer einen *Type Cast* auf einen korrekt typisierten Zeiger erfordert.
  - `new` und `delete` rufen automatisch den Konstruktor bzw. Destruktor einer Klasse auf.

Vergleich mit `malloc()` und `free()`

  - Eine Initialisierung des allokierten Speichers ist mit `new` möglich, während `malloc()` nur uninitialisierten Speicher verwaltet.
- Eine gemischte Verwendung von `new` und `malloc()` bzw. `delete` und `free()` ist möglich.

Ausdrücke und Befehle

- Operatoren
  - Arithmetische Operatoren
  - Vergleichs-Operatoren
  - Bitweise und logische Operatoren
  - Zuweisungs-Operatoren
  - Der Operator für bedingte Ausführung
  - Die Operatoren `new` und `delete`
  - **Präzedenz von Operatoren**

Präzedenz von Operatoren

- Die Präzedenz der C++-Operatoren (*Operator Precedence*) erfolgt nach ihrer fixen Reihung (siehe Tabelle im Skriptum, Abschnitt 2.3.5).
- Operatoren, die in einem Ausdruck "gleichberechtigt" nebeneinander stehen, werden in der Reihenfolge ihrer Präzedenzen abgearbeitet.

Präzedenz von Operatoren

- Durch Verwendung von Klammern kann diese Reihenfolge modifiziert werden; Klammerausdrücke werden immer zuerst ausgewertet:

```
int i = 2 + 3*4;      // ergibt i == 14;
int j = (2 + 3)*4;    // ergibt i == 20;
```


Präzedenz von Operatoren

- Der Postfix-Inkrement-Operator hat eine *höhere* Präzedenz als der Indirektions-Operator; der Präfix-Inkrement-Operator hat die *gleiche* Präzedenz wie der Indirektions-Operator.
- Die folgenden Ausdrücke haben daher — je nach Vorhandensein und Position von Klammern — die folgenden Bedeutungen:

Präzedenz von Operatoren

```
char *bufptr;
char ch;
```

Ausdruck:	Bedeutung:
`ch = *bufptr;`	`ch` = Zeichen, auf das `bufptr` zeigt.
`bufptr++;`	Zeige auf das nächste Zeichen im Puffer.
`++bufptr;`	wie oben
`ch = *bufptr++;`	`ch` = Zeichen, auf das `bufptr` zeigt; `bufptr` wird inkrementiert und zeigt auf das nächste Zeichen.
`ch = *(bufptr + 1);`	`ch` = Zeichen hinter jenem, auf das `bufptr` zeigt.

Präzedenz von Operatoren

Ausdruck:	Bedeutung:
`ch = *bufptr + 1;`	`ch` = Zeichen, auf das `bufptr` zeigt, um 1 inkrementiert.
`ch = ++(*bufptr);`	`ch` = Zeichen, auf das `bufptr` zeigt, wird im Puffer um 1 inkrementiert und anschließend an `ch` zugewiesen.
`ch = ++*bufptr;`	wie oben
`ch = (*bufptr)++;`	`ch` = Zeichen, auf das `bufptr` zeigt; wird nach der Zuweisung im Puffer um 1 inkrementiert.
`ch = *(++bufptr);`	`bufptr` zeigt auf das nächste Zeichen; dieses wird `ch` zugewiesen.
`ch = *++bufptr;`	wie oben

Präzedenz von Operatoren

- Im Zweifelsfall ist es günstiger, zu viele Klammern zu verwenden als zu wenige.
- Insbesondere dann, wenn nicht-arithmetische Operatoren mit arithmetischen gemischt werden, können unerwartete Effekte auftreten:

Präzedenz von Operatoren

- Beispiel:

  ```
  cout << 3 < 2 << "\n";
  ```

  gibt nicht den Wert `0` aus, sondern bewirkt eine Compiler-Fehlermeldung. Wegen der höheren Präzedenz von "<<" gegenüber "<" bedeutet der obige Ausdruck:

  ```
  (cout << 3) < (2 << "\n");
  ```

- Korrekt wäre gewesen:

  ```
  cout << (3 < 2) << "\n";
  ```


Ausdrücke und Befehle

- Initialisierungen
- *L-Values* und *R-Values*
- Funktionen
- Operatoren
- **Befehle (*Statements*)**

Ausdrücke und Befehle

- Befehle (*Statements*)
  - **Allgemeines**
  - Der Sprungbefehl `goto`
  - Programm-Rückkehr mit `return`
  - Programmverzweigungen
  - Schleifen
  - `break` und `continue`

Ausdrücke und Befehle

- Befehle (*Statements*)
  - Allgemeines
    - **Befehle mit Ausdrücken**
    - Der leere Befehl (*Null Statement*)
    - Befehlsblöcke

Befehle mit Ausdrücken

- Der einfachste Typ von Befehlen in C/C++ besteht aus einem Ausdruck, der mit einem Strichpunkt (";") abgeschlossen ist.
- Die Ausdrücke werden der Reihe nach abgearbeitet.
- Mit der Bearbeitung des nächsten Ausdrucks wird erst begonnen, wenn der aktuelle Ausdruck und alle seine Nebenwirkungen zur Gänze abgeschlossen sind.

Ausdrücke und Befehle

- Befehle (*Statements*)
  - Allgemeines
    - Befehle mit Ausdrücken
    - **Der leere Befehl (*Null Statement*)**
    - Befehlsblöcke

Der leere Befehl

- Überall dort, wo C/C++ einen Befehl erwartet, wird auch ein leerer Befehl (*Null Statement*) akzeptiert.
- Ein leerer Befehl besteht nur aus dem (abschließenden) Strichpunkt.

Der leere Befehl

```
// String Source auf Dest kopieren
char *strcpy (char *Dest, const char *Source)
{
  char *DestStart = Dest; // Hilfs-Zeiger
  while (*Dest++ = *Source++)
    ; // Leerer Befehl; alles ist schon in der obigen
      // Zeile passiert.
  return DestStart;
      // Die Funktion soll einen Zeiger auf die Kopie
      // zurückgeben.
}
```


Ausdrücke und Befehle

- Befehle (*Statements*)
  - Allgemeines
    - Befehle mit Ausdrücken
    - Der leere Befehl (*Null Statement*)
    - **Befehlsblöcke**

Befehlsblöcke

- Eine beliebige Anzahl von Befehlen (auch null) kann mit geschwungenen Klammern zu einem (Befehls-) Block (*Compound Statement*) zusammengefasst werden.
- Überall dort, wo C/C++ einen Einzelbefehl erwartet, wird auch ein Befehlsblock akzeptiert:

```
if (a > b)
{
  int temp = a;
  a = b;
  b = temp;
}
```


Ausdrücke und Befehle

- Befehle (*Statements*)
  - Allgemeines
  - **Der Sprungbefehl `goto`**
  - Programm-Rückkehr mit `return`
  - Programmverzweigungen
  - Schleifen
  - `break` und `continue`

Der Sprungbefehl `goto`

- Der Befehl `goto` bewirkt einen unbedingten Sprung auf eine Sprungmarke.
- Die Ziel-Sprungmarke muss innerhalb der gleichen Funktion liegen wie der Befehl `goto`.
- Sie darf aber in einem beliebigen Block liegen.
- `goto` ist z.B. sinnvoll, wenn ein vorzeitiger Abbruch mehrerer ineinander geschachtelter Schleifen erforderlich ist:

Der Sprungbefehl `goto`

```
int main ()
{
  for (int i = 0; i < 1000000; i++)
  {
    for (int j = 0; j < 1000000; j++)
    {
      if (Abbruch ())  // irgendeine Abbruchbedingung
        goto ende;     // "ende" ist eine Sprungmarke
// Code, der in der Schleife bearbeitet werden soll
    }
  }
// Code für volle Abarbeitung der Schleifen
ende:                  // Sprung mit goto landet hier
// Weiterer Programmcode
}
```


Ausdrücke und Befehle

- Befehle (*Statements*)
  - Allgemeines
  - Der Sprungbefehl `goto`
  - **Programm-Rückkehr mit `return`**
  - Programmverzweigungen
  - Schleifen
  - `break` und `continue`

Programm-Rückkehr mit `return`

- Der Befehl `return` bewirkt eine sofortige Rückkehr einer Funktion zur aufrufenden Funktion.
- Innerhalb einer Funktion können beliebig viele `return`-Befehle vorkommen.
- `return` in der Funktion `main()` beendet das Programm.
- Das "Ergebnis" der Funktion `main()` wird an das Betriebssystem weitergegeben und kann dort von *Scripts* abgefragt werden.

Programm-Rückkehr mit `return`

- Bei Funktionen, die *nicht* die Type `void` haben, erwartet C++ einen Ausdruck unmittelbar nach `return`, dessen Ergebnis an die aufrufende Funktion zurückgegeben wird.
- `return` *ohne* Ausdruck stellt in diesem Fall einen Fehler dar.

Programm-Rückkehr mit `return`

- Der mit dem `return`-Befehl angegebene Ausdruck kann beliebig kompliziert sein; sein Ergebnis wird allenfalls in den Typ der Funktion konvertiert:

```
#include <math.h>
double ZehnHochX (double x)
{
  return exp(x*log(10));
}
```


Programm-Rückkehr mit `return`

- Bei Funktionen der Type `void` darf *kein* Ausdruck mit `return` angegeben werden.
- Das Ende des Funktions-Blocks entspricht einem `return` ohne nachfolgenden Ausdruck.
- Es ist nur bei Funktionen vom Typ `void` zulässig, die Funktion am Ende ihres Blocks *ohne* `return` zu verlassen.

Ausdrücke und Befehle

- Befehle (*Statements*)
  - Allgemeines
  - Der Sprungbefehl `goto`
  - Programm-Rückkehr mit `return`
  - **Programmverzweigungen**
  - Schleifen
  - `break` und `continue`

Ausdrücke und Befehle

- Befehle (*Statements*)
  - Programmverzweigungen
    - **`if…else`**
    - `switch`

Programmverzweigungen mit `if…else`

- Auf den Befehl `if` muss ein Ausdruck in Klammern ("`(  )`") folgen, der von einer arithmetischen oder Zeiger-Type sein muss oder in eine solche umgewandelt werden kann.
- Nur wenn das Ergebnis dieses Ausdrucks von Null verschieden ist, wird der darauf folgende Ausdruck oder Block ausgeführt.

Programmverzweigungen mit `if…else`

- Optional kann danach der Befehl `else` mit einem weiteren Ausdruck oder Block folgen, der nur dann ausgeführt wird, wenn der mit `if` angegebene Ausdruck den Wert Null ergeben hat:

```
if (x > 0)
  cout << "x ist größer als 0\n";
else
  cout << "x ist kleiner oder gleich 0\n";
```


Programmverzweigungen mit `if…else`

- Bei ineinander geschachtelten `if…else`-Befehlen bezieht sich ein `else` immer auf das *letzte* ihm vorausgehende `if`, das noch nicht mit einem `else` gepaart ist.
- Geschwungene Klammern ("`{ }`") und Einrückungen können diesen Zusammenhang verdeutlichen:

Programmverzweigungen mit `if…else`

```
if (B1)
  if (B2)
    cout << "B1 == TRUE, B2 == TRUE\n";
  else
    cout << "B1 == TRUE, B2 == FALSE\n";
else
  if (B2)
    cout << "B1 == FALSE, B2 == TRUE\n";
  else
    cout << "B1 == FALSE, B2 == FALSE\n";
```


Ausdrücke und Befehle

- Befehle (*Statements*)
  - Programmverzweigungen
    - `if…else`
    - **`switch`**

Programmverzweigungen mit `switch`

- Der Befehl `switch` erlaubt eine Verzweigung zwischen *beliebig vielen* Wegen durch ein Programm.
- Der in Klammern ("`( )`") auf `switch` folgende Ausdruck muss von einer ganzzahligen Datentype sein oder in eine solche eindeutig konvertiert werden können.
- Dieser Ausdruck wird ausgewertet und bestimmt die weitere Vorgangsweise.
- Auf `switch` folgt ein Befehlsblock, innerhalb dessen sich beliebig viele (auch keine) `case`-Sprungmarken und maximal eine `default`-Sprungmarke befinden können.

Programmverzweigungen mit `switch`

- Jede `case`-Sprungmarke muss von einem Ausdruck gefolgt sein, dessen Ergebnis eine konstante ganze Zahl sein muss.
- Diese Werte müssen für alle `case`-Sprungmarken individuell verschieden sein.
- Wenn das Ergebnis des auf `switch` folgenden Ausdrucks mit dem Wert einer `case`-Sprungmarke übereinstimmt, wird zu dieser Sprungmarke verzweigt.
- Existiert keine `case`-Sprungmarke mit passendem Wert, dann wird zur `default`-Sprungmarke verzweigt, sofern eine solche vorhanden ist.

Programmverzweigungen mit `switch`

- Existiert keine `default`-Sprungmarke, so verzweigt das Programm hinter das Ende des auf `switch` folgenden Befehlsblocks.
- Mehrere Sprungmarken dürfen unmittelbar hintereinander stehen.
- Ein `break`-Befehl bewirkt einen Sprung auf den ersten Befehl nach dem `switch`-Block.
- Typische Anwendungen für `switch`: Befehlsprozessoren.

Programmverzweigungen mit `switch`

```
#include <iostream.h>
int main ()
{
  int x = 0;
  do
  {
    cout << "Bitte Wert
      eingeben: ";
    cin >> x;
    switch (x)
    {
      case 0:
        cout << "Null\n";
        break;
      case 1:
        cout << "Eins\n";
        break;
      case 2:
        cout << "Zwei\n";
      case 3:
      case 5:
      case 7:
        cout<<"Primzahl!\n";
        break;
      default:
        cout<<"Irgendwas\n";
    }
  } while (x != 0);
}
```


Ausdrücke und Befehle

- Befehle (*Statements*)
  - Allgemeines
  - Der Sprungbefehl `goto`
  - Programm-Rückkehr mit `return`
  - Programmverzweigungen
  - **Schleifen**
  - `break` und `continue`

Ausdrücke und Befehle

- Befehle (*Statements*)
  - Schleifen
    - **`while`-Schleifen**
    - do…`while`-Schleifen
    - `for`-Schleifen

`while`-Schleifen

- Eine `while`-Schleife beginnt mit dem Schlüsselwort `while`, gefolgt von einem Ausdruck in Klammern (`"( )"`), der von einer ganzzahligen Type, einer Zeigertype oder einer Klassentype sein muss, die eindeutig in einen Ausdruck mit einer ganzzahligen Type umgewandelt werden kann.
- Der darauf folgende Befehl oder Befehlsblock wird ausgeführt, solange der Ausdruck in der Klammer hinter `while` ein von Null verschiedenes Ergebnis hat.
- Der Ausdruck wird jeweils *vor* der Ausführung des Befehls oder Blocks getestet; die Ausführung einer `while`-Schleife kann daher auch völlig unterbleiben.

`while`-Schleifen

- Endlosschleifen können mit einem Befehl der Form `while(1)` realisiert werden.
- Das folgende Programm entfernt führenden *White Space* von einem *String*:

while-Schleifen

```
#include <stdio.h>
const char *strip (const char *buffer);
void write (const char *buffer);
int main ()
{
  char buffer[256];            // Eingabe-Puffer
  printf ("Bitte String eingeben: ");
  gets (buffer);
  write (buffer);              // Original
  write (strip (buffer));      // gestrippt
}
```


while-Schleifen

```
const char *strip (const char *buffer)
{
  const char *ptr = buffer;
  while (ptr && *ptr && *ptr <= ' ')
    ptr++;
  return ptr;
}
void write (const char *buffer)
{
  printf ("\n***%s***\n", buffer);
               // String mit "***" "eingeklammert"
}
```


Ausdrücke und Befehle

- Befehle (*Statements*)
  - Schleifen
    - `while`-Schleifen
    - **`do…while`-Schleifen**
    - `for`-Schleifen

do…while-Schleifen

- Im Gegensatz zu `while`-Schleifen wird hier die Schleifenbedingung erst geprüft, *nachdem* die Schleife durchlaufen wurde.
- do…`while`-Schleifen werden daher *mindestens einmal* ausgeführt.
- Sie bestehen aus dem Schlüsselwort `do`, das von einem Befehl oder Befehlsblock gefolgt ist, auf den `while` mit der Schleifenbedingung in Klammern ("`(  )`"), gefolgt von einem Strichpunkt, folgt.
- Das folgende Programm wartet, bis der Buchstabe "x" auf der Tastatur eingegeben wurde:

do…while-Schleifen

```
#include <conio.h>
#include <iostream.h>
int main ()
{
  char ch;
  do
    ch = getche (); // warte auf Zeichen von der
                    // Tastatur, und gib sein Echo aus
  while (ch != 'x');
  cout << "\nEndlich ein \"x\"!";
}
```


Ausdrücke und Befehle

- Befehle (*Statements*)
  - Schleifen
    - `while`-Schleifen
    - do…`while`-Schleifen
    - **`for`-Schleifen**

for-Schleifen

- for-Schleifen haben gegenüber while-Schleifen eine erweiterte Funktionalität.
- Dem for-Befehl folgt eine Klammer ("( )"), in der, durch Strichpunkte (";") getrennt, drei Befehle bzw. Ausdrücke stehen können, die die folgenden Funktionen haben:
  - Ein Befehl, der ausgeführt wird, *bevor* das Programm in die Schleife eintritt, und der Ausdrücke und/oder Deklarationen enthalten kann (*Initialisierung* des Schleifenzählers).

for-Schleifen

  - Ein Ausdruck, der funktionell und syntaktisch dem auf while folgenden Klammerausdruck äquivalent ist. Dieser Ausdruck wird *vor* jedem Schleifendurchlauf ausgewertet; die Schleife wird so lange durchlaufen, so lange das Ergebnis des Ausdrucks von Null verschieden ist (*Abbruchbedingung*).
  - Ein Befehl, der auf jeden Fall am *Ende* jedes Schleifendurchlaufs ausgeführt wird, unmittelbar, *bevor* der mittlere Ausdruck vor dem nächsten Durchlauf getestet wird (z.B. *Inkrementieren* des Schleifenzählers).

for-Schleifen

- Die beiden Befehle können auch aus mehreren durch Kommas (",") voneinander getrennten Befehlen oder Ausdrücken zusammengesetzt sein.
- Alle drei Ausdrücke in einer for-Schleife sind optional; die beiden Strichpunkte zwischen ihnen müssen jedoch auf jeden Fall vorhanden sein.
- Mit for(;;) kann eine Endlosschleife eingeleitet werden (äquivalent zu while(1)).

for-Schleifen

- Die Schleife

```
for (Initialisierung; Testen; Weiter)
{
  Schleifenkörper;
}
```

ist äquivalent zu:

```
Initialisierung;
while (Testen)
{
   Schleifenkörper;
   Weiter;
}
```


for-Schleifen

- Das folgende Programm nimmt bis zu 20 ganze Zahlen über die Konsole entgegen, sortiert sie in aufsteigender Reihenfolge und gibt sie wieder aus. Die Eingabe wird als beendet betrachtet, wenn die Zahl 0 eingegeben wurde:

for-Schleifen

```
#define FELDGROESSE 20
#include <iostream.h>

int HoleZahlen (int *Feld, int Groesse);
void Schreibe (int *Feld, int Aktiv);
void Sortiere (int *Feld, int Aktiv);

int main ()
{
  int IntFeld [FELDGROESSE];

  int AktGroesse = HoleZahlen (IntFeld,
    sizeof IntFeld/sizeof IntFeld[0]);
  Schreibe (IntFeld, AktGroesse);
  Sortiere (IntFeld, AktGroesse);
  Schreibe (IntFeld, AktGroesse);
}
```

for-Schleifen

```
int HoleZahlen (int *Feld, int Groesse)
{
  for (int Zaehler = 0, temp = 1; temp && (Zaehler <
    Groesse); Zaehler++)
  {
    cout << "Zahl: ";
    cin >> temp;
    if (temp)         // != 0
      *(Feld + Zaehler) = temp;
    else
      Zaehler--;   // zähle 0 nicht
  }
  return Zaehler;
}
```


for-Schleifen

```
void Schreibe (int *Feld, int Aktiv)
{
  for (int i = 0; i < Aktiv; i++)
    cout << Feld[i] << ((i < Aktiv - 1) ? ", " :
  "\n");
}
```


for-Schleifen

```
void Sortiere (int *Feld, int Aktiv)
{
  for (int i = 0; i < Aktiv - 1; i++)
  {
    for (int j = i + 1; j < Aktiv; j++)
    {
      if (Feld[i] > Feld[j])
      {
        int temp = Feld[i];
        Feld[i] = Feld[j];
        Feld[j] = temp;
      }
    }
  }
}
```


Ausdrücke und Befehle

- Befehle (*Statements*)
  - Allgemeines
  - Der Sprungbefehl `goto`
  - Programm-Rückkehr mit `return`
  - Programmverzweigungen
  - Schleifen
  - **`break` und `continue`**

Ausdrücke und Befehle

- Befehle (*Statements*)
  - `break` und `continue`
    - **Der Befehl `break`**
    - Der Befehl `continue`

Der Befehl `break`

- `break` bewirkt einen Sprung auf den ersten Befehl, der auf die aktuelle Programm-Umgebung folgt, in folgenden Fällen:
  - Innerhalb des Befehlsblocks, der auf einen `switch`-Befehl folgt; und
  - im "Körper" einer `while`-, `do…while`- oder `for`-Schleife.
- Das folgende Beispiel ist eine elegantere Variante der Funktion `HoleZahlen()` aus dem vorigen Abschnitt:

Der Befehl `break`

```
int HoleZahlen (int *Feld, int Groesse)
{
  for (int Zaehler = 0; Zaehler < Groesse; Zaehler++)
  {
    int temp;
    cout << "Zahl: ";
    cin >> temp;
    if (! temp)     // == 0
      break;
    *(Feld + Zaehler) = temp;
  }
  return Zaehler;
}
```


Der Befehl `break`

- Wenn mehrere `switch`-Befehle und/oder Schleifen ineinander geschachtelt sind, wird mit `break` jeweils der diesen Befehl unmittelbar umschließende `switch`-Befehl bzw. die ihn umschließende Schleife verlassen.
- Wenn *alle* ineinander geschachtelten `switch`-Befehle oder Schleifen verlassen werden sollen, sind weitere `break`-Befehle in den äußeren `switch`-Befehlen oder Schleifen erforderlich.
- In diesem Fall kann ein `goto` wesentlich einfacher und auch übersichtlicher sein.

Ausdrücke und Befehle

- Befehle (*Statements*)
  - `break` und `continue`
    - Der Befehl `break`
    - **Der Befehl `continue`**

Der Befehl `continue`

- Dieser Befehl ist in C/C++ nur im Kontext einer Schleife gültig.
- `continue` bewirkt einen Sprung ans Ende der den Befehl unmittelbar umgebenden Schleife.
- In `while`- und `do…while`-Schleifen erfolgt der Sprung unmittelbar vor die Prüfung der Fortsetzungsbedingung der Schleife; in `for`-Schleifen erfolgt er unmittelbar vor den dritten Ausdruck im `for`-Befehl (also im Allgemeinen vor die Inkrementierung des Schleifenzählers).

Der Befehl `continue`

```
#include <iostream.h>
int main ()
{
  for (int i = 0, j = 0; i < 10; i++)
  {
    int x;
    cout << "Wert: ";
    cin >> x;
    if (x%2)            // durch 2 nicht teilbar
      continue;
    cout << "ist teilbar durch 2\n";
    j++;                // zähle Treffer
  }
  cout << j << " Treffer!\n";
}
```