Programmiertechnik

C++

Vorlesung

(Autor: Michael Obert)

Redaktionell überarbeitet von Jürgen Wemheuer

Version: 18.09.2010
Changelog:
Verschiedene textliche, sprachliche und Layout-Umstellungen und Korrekturen
To Do: Diverse inhaltliche Erweiterungen, Merksätze. Gliederung logischer

Inhaltsverzeichnis

1 Einführung 5
1.1 Software 5
1.2 Hardware 5
1.3 Programmiersprachen 6
1.3.1 Assemblersprachen 6
1.3.2 Höhere Programmiersprachen 7
1.3.3 Entwicklung von C++ 8
1.3.4 Entwicklung von C++ 9
1.3.5 Compiler und Interpreter 10
1.3.6 Linker 10
1.4 Entwicklung eines C++-Programms 11
2 Phasen der Programmerstellung 12
2.1 Problemanalyse 12
2.2 Programmentwurf 12
2.2.1 Graphische Darstellungsform eines Algorithmus 13
2.3 Erstellen des Quellprogramms 15
2.4 Testen des Programms 15
2.5 Strukturelemente des Stuktogramms (DIN 66261) 16
2.6 Syntax 18
2.7 Zeichensatz 18
2.8 Namen 20
2.9 Erzeugung eines lauffähigen C++ Programms 21
2.10 Kommentare 22
2.11 Präprozessor-Anweisung 23
2.12 Zahlensysteme 28
2.12.1 Darstellung der Zahlen 28
2.13 Fundamentale Datentypen 29
2.13.1 Standarddatentypen 29
2.13.2 Ganzzahlen (`integer ➔ int`) 29
2.13.3 Dezimalzahlen (reelle Zahlen) 30
2.13.4 Zeichen (char-Datentyp) 31
2.13.5 Bool-Datentyp 31
2.14 Operatoren und Ausdrücke 32
2.15 Übersicht der Operatoren von C++ 33
2.16 Arithmetische Operatoren 34
2.16.1 Zuweisungsoperator 36
2.16.2 Increment – Decrement – Operator 37
2.16.3 Mathematische Konstanten und Standardfunktionen 39
2.16.4 Logische Operatoren und Vergleichsoperatoren 40
2.16.5 Zusammengesetzte Operatoren 41
2.16.6 sizeof-Operator 41
2.17 Verbundanweisung 42
2.18 Bedingte Anweisungen 43
2.18.1 If-Anweisung: 44
2.18.2 Switch-Anweisung (Mehrfachauswahl): 48
2.19 Schleifenanweisungen 51
2.19.1 while-Anweisung 51
2.19.2 do-while-Anweisung 52
2.19.3 Gegenüberstellung while und do-while 53
2.19.4 for-Anweisung 54

2.19.4.1 Abbruch der Schleife: break; ... 55
2.19.4.2 nächster Schleifendurchgang: continue; ... 55
2.20 Konstanten ... 56
2.20.1 Allgemeines ... 56
2.20.2 Konstanten aus der Headerdatei math.h ... 57
2.21 Macros ... 58
2.22 Spezielle Probleme bei der Ein- und Ausgabe ... 59
2.23 Steuerzeichen für Ausgabe (Escape-Sequenzen) ... 60
2.24 Manipulatoren ... 61
2.25 IOS-Flags ... 62
2.25.1 Anwenden der Flags ... 63
2.25.2 Setzen der Flags ... 63
2.26 Felder (Arrays) ... 64
2.26.1 Zugriff auf Feld-Elemente ... 66
2.26.2 Initialisierung von Feldern ... 67
2.26.3 Dimension des Feldes ... 69
2.26.4 Zugriff auf Elemente eines zweidimensionalen Feldes: ... 70
2.26.5 Speicherbedarf von mehrdimensionalen Feldern ... 71
2.27 Zeichenketten (Strings) ... 79
2.27.1 Ein- Ausgabe von Zeichenketten ... 80
2.27.2 Standardfunktionen für Zeichenketten ... 82
2.27.3 Felder von Zeichenketten ... 85
2.28 Typenumwandlung ... 86
2.28.1 Implizit: automatische Typenumwandlung ... 86
2.28.2 Explizit: der Cast-Operator ... 87
3 Zeiger-Datentyp ... 89
3.1 Verweisoperator ... 90
3.2 NULL-Zeiger ... 91
3.3 Adressoperator ... 92
3.4 Zeiger auf Felder ... 92
3.4.1 Übungen mit Zeigern ... 94
3.5 Zeigerarithmetik ... 101
3.6 Zeiger und Strings (Zeichenketten) ... 103
3.6.1 Vereinbarung von Zeichenketten: ... 103
3.6.2 Nachteile von char-Zeigern ... 105
4 Funktionen ... 107
4.1 Grundaufbau einer Funktion ... 107
4.2 Funktionen ohne Parameter ... 108
4.3 Prototypenanweisung ... 109
4.4 Funktionen mit Parameterübergabe ... 114
4.5 Prototypanweisung: ... 116
4.6 Funktionsparameter ... 118
4.7 Eindimensionale Felder als Funktionsparameter ... 123
4.8 Standardwerte für Parameter und variable Parameteranzahl ... 127
4.9 Überladen von Funktionen (Polymorphie) ... 128
4.10 Probleme bei Polymorphie mit Standardwerten ... 129
4.11 Zeiger als Funktionsparameter ... 130
4.11.1 Zeiger als Übergabeparameter ... 130
4.11.2 Zeiger als Rückgabeparameter ... 131
4.11.3 Statische Deklaration (static) ... 133
4.12 Rekursion ... 134
4.13 Dynamische Speicherverwaltung ... 139

4.13.1 malloc 139
4.13.2 free 140
4.13.3 memcpy 141
4.14 Strukturen 142
5 Anhang 143
5.1 Literatur 143
5.2 Aufgabensammlung 144

1 Einführung

1937 Zuse Z1: 1. elektromechanischer Rechner, erfunden von Konrad Zuse

1946 ENIAC: 1. elektronische Rechner (Röhrenrechner)

1981 1. Personalcomputer (PC) vorgestellt von der Firma IBM

1.1 Software

Software bezeichnet die Gesamtheit aller Programme

Betriebssysteme	Windows (95/98/ME/XP/Vista/7) DOS UNIX, Linux WIN NT, Windows Server
Systemsoftware	Compiler / Interpreter Assembler Texteditoren Browser, Decoder
Standardsoftware	Textverarbeitung (Word, AmiPro) Tabellenkalkulation (Excel) Datenbankprogramme (dBase, Access, Oracle) CAD-Programme (AutoCAD)
Anwendungssoftware	Spezielle Programme zur Lösung kleiner Aufgaben z.B. Auftragsverwaltung, Notenverwaltung z.B. Mediaplayer, Spiele

1.2 Hardware

Als Hardware bezeichnet man die Menge aller technischen Geräte eines Computers:

- z.B. Hauptplatine, Netzteil, Lüfter
- z.B. Festplattenlaufwerk, CD-, DVD-, BlueRay-Laufwerke
- z.B. Drucker, Monitor, Tastatur, Maus
- z.B. Soundkarte, Netzwerkkarte

1.3 ProgrammiersprachenEine **Programmiersprache** bietet die Möglichkeit, **einem Computer Befehle** zu geben, die dann ausgeführt werden können. Eine **Folge** von solchen **Befehlen** nennt man ein **Programm**.

In den **Anfangszeiten** der Computertechnik blieb den Programmierern nichts anderes übrig, als Programme in Form von **langen Folgen von Nullen und Einsen** einzugeben. Solche Folgen von Nullen und Einsen nennt man **Binärcode.** Diese Art der Programmierung war sehr mühsam und fehleranfällig, dafür aber sehr schnell.

1.3.1 Assemblersprachen

Die erste **Verbesserung** wurde durch den Einsatz von **Assemblersprachen** erzielt. Assemblersprachen sind **maschinenorientierte Programmier-sprachen**, in der die Befehle, die bisher durch Nullen und Einsen beschrieben werden mussten, durch **leichter verständliche Symbole** dargestellt werden. Diese Symbole nennt man **Mnemonics**.

Jeder Computertyp (Prozessor) besitzt seine eigene, spezeille Assemblersprache.

Ein Assemblierer muss die Mnemonics vor der Ausführung des Programms in den **Binärcode der Maschine** übersetzen.

Die maschinennahe Programmierung anhand eines Mnemonic-Codes ist sehr aufwändig und kompliziert.

1.3.2 Höhere Programmiersprachen

Nach den **Assemblersprachen** folgten die sogenannten **höheren Programmiersprachen**, wie **FORTRAN, ALGOL** oder **COBOL**, die die Eingabe von Programmen in einer Form erlauben, die **natürlichen Sprachen**, wie etwa Englisch **nachempfunden** ist.

Die Entwicklung setzte sich mit Sprachen wie **PASCAL** oder **C** fort. Heutzutage werden Sprachen wie **C++** oder **Java** eingesetzt. **Keine** dieser modernen Sprachen ist eine **völlige Neuentwicklung**, denn sie **bauen** entweder direkt auf einem Vorgänger **auf** oder **teilen grundlegende Konzepte** mit zuvor entwickelten Sprachen.

Beispiele für höhere Programmiersprachen

FORTRAN	1954	Formular translator
COBOL	1960	kaufmännischer Bereich
PL/1	1965	von IBM entwickelt
BASIC	1965	nicht für professionelle Anwendungen geeignet
PASCAL	1972	Niklaus Wirth
C	1974	Kernighan/Richie
C++	1980/83	Bjarne Stroustrup
JAVA	1994/95	Sun

Ein Programm oder ein Teil eines Programms wird oft als **Quelle, Quelltext, Quellcode** oder einfach nur als **Code** bezeichnet. Die Entwicklung eines Programms nennt man auch **Implementierung**.

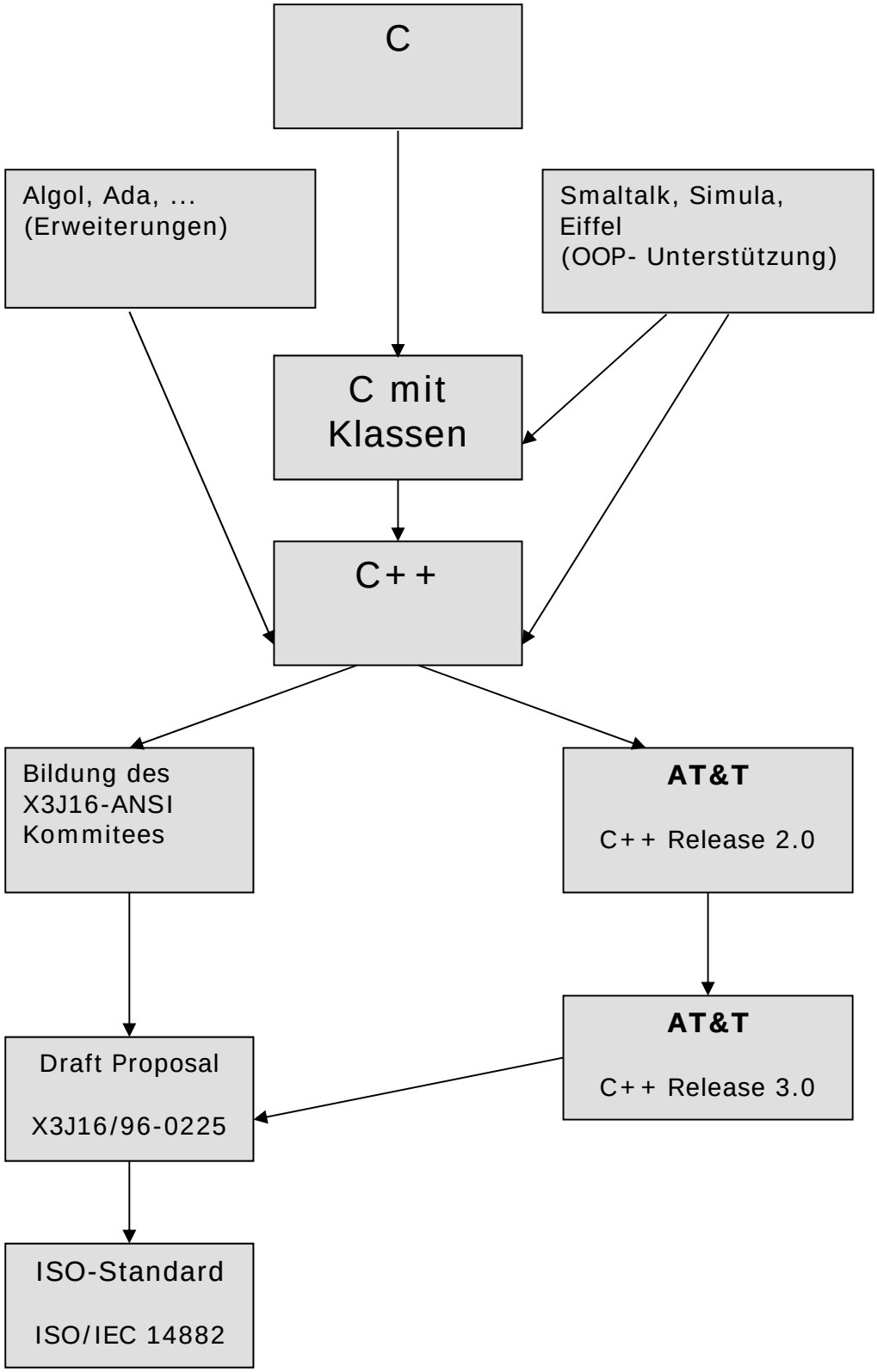
1.3.3 Entwicklung von C++

C++ wurde von **Bjarne Stroustrup** in den Bell-Laboratioen in den USA entwickelt, um Simulationsprojekte mit minimalem Speicherplatz und Zeitbedarf zu realisieren. Frühe Versionen der Sprache, die zunächst als „**C mit Klassen**“ bezeichnet wurde, gibt es seit 1980. Der Name C++ wurde 1983 von Rick Mascitti geprägt. Er weist darauf hin, dass die Programmiersprache C++ evolutionär aus der Programmiersprache C entstanden ist: ++ ist der Inkrementoperator von C.

Die Firma AT&T unterstützte frühzeitig die Etablierung eines C++-Standards. Hierbei ging es darum, dass sich möglichst viele Compiler-Bauer und Software-Entwickler auf eine einheitliche Sprachbeschreibung einigten, um die Bildung von Dialekten und Sprachverwirrungen zu vermeiden.

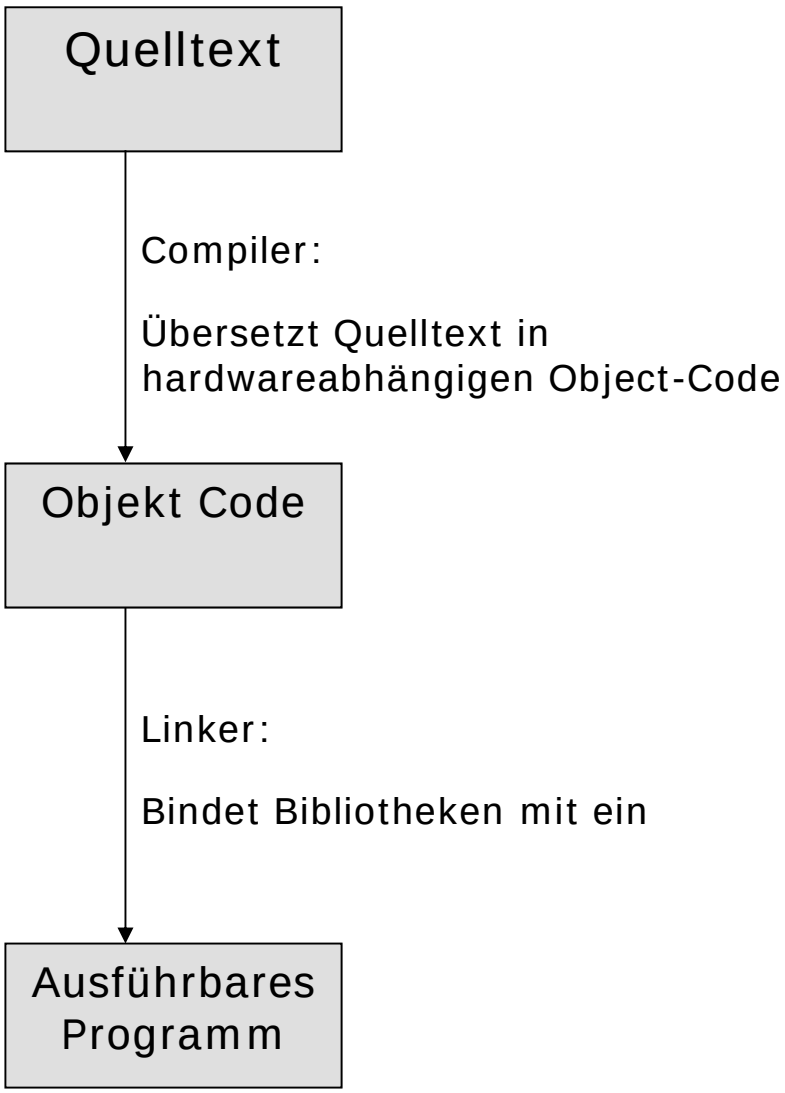
Im Jahr 1989 konnte auf Initiative von Hewlett-Packard ein ANSI-Kommitee (American National Standards Institute) gebildet werden, das verschiedene Entwürfe für einen Standard geschaffen hat. Dieser Standard wurde 1998 als ISO-Standard (International Standards Organization, ISO/IEC 14882) verabschiedet.

1.3.4 Entwicklung von C++



1.3.5 Compiler und Interpreter

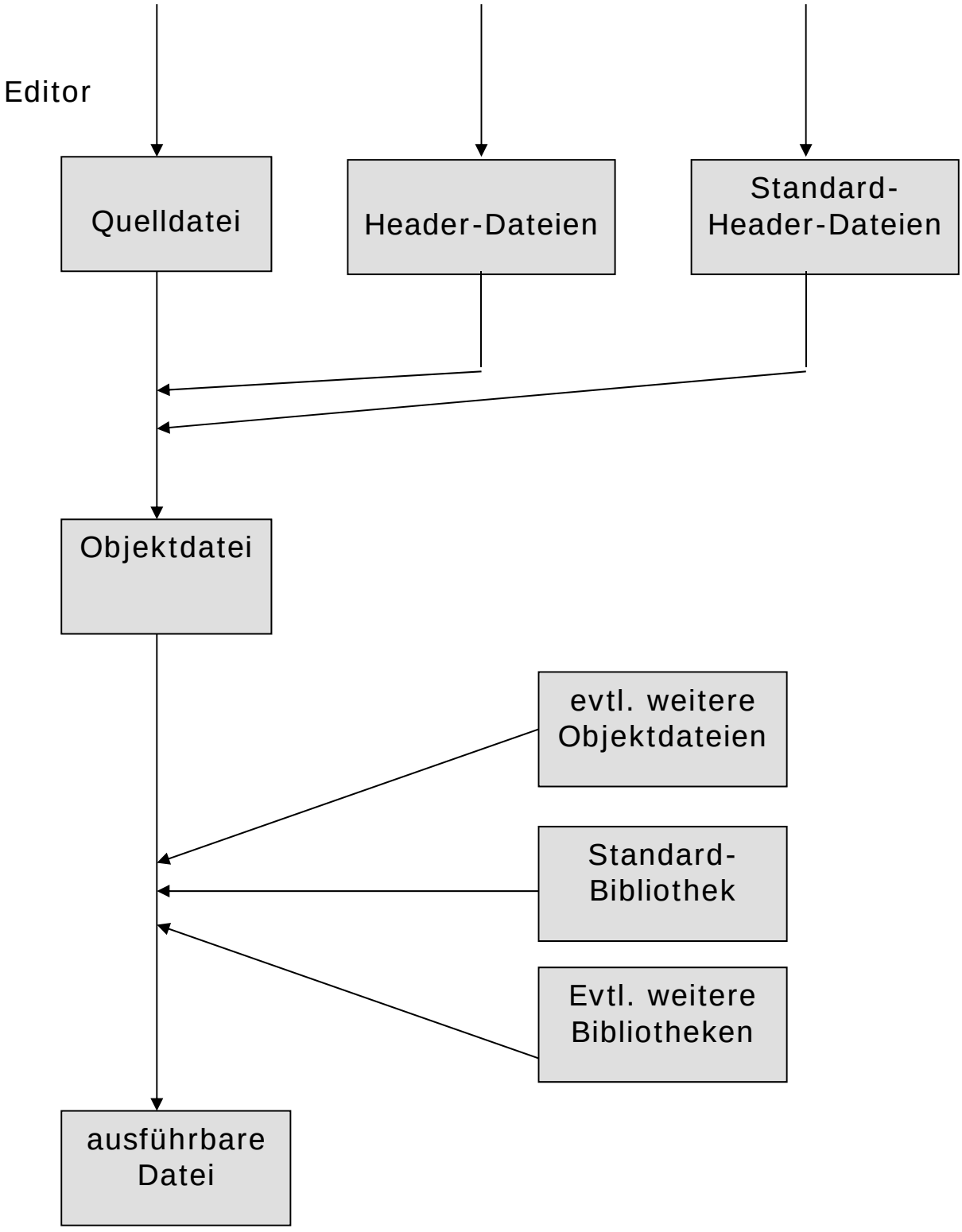
Eine Programmiersprache ermöglicht mit dem Computer zu interagieren. Ein Compiler übernimmt dabei die Übersetzung des Programms,. das in einer Programmiersprache geschrieben wurde, in eine Form, die der Computer verarbeiten kann. Dazu übersetzt der Compiler das Programm in eine Zwischenform, die man Objektcode nennt.



1.3.6 Linker

Der Linker bindet die Objektdateien zu einer ausführbaren Datei. Diese enthält neben den selbsterzeugten Objektdateien auch den Startup-Code und ein Module mit den verwendeten Funktionen und Klassen der Standardbibliothek.

1.4 Entwicklung eines C++-Programms



2 Phasen der Programmerstellung

Die Entwicklung von Computerprogrammen läuft – unabhängig von der verwendeten Sprache – üblicherweise nach dem folgenden Muster ab:

Die vier Phasen der Programmentwicklung

1. Problemanalyse **(am Beispiel Geschwindigkeitsberechnung)**
2. Programmentwurf
3. Erstellen des Quellprogramms
4. Testen des Programms

2.1 Problemanalyse

- Welches Problem ist zu lösen?
- Welche Angaben werden benötigt?
  - zurückgelegter Weg
  - benötigte Zeit
  - (Test, ob eingegebene Werte sinnvoll sind)
- Was soll das Programm tun?
  - Berechnung der Geschwindigkeit aus zurückgelegtem Weg und der dafür benötigten Zeit
    v = s/t => t muss größer 0 sein!
- Ausgabe der berechneten Werte
  mit erläuterndem Text
  z.B. „Die Geschwindigkeit beträgt ... km/h“

2.2 Programmentwurf

Als nächstes wird die Aufgabe als Algorithmus formuliert.

Der Algorithmus ist eine Verarbeitungsvorschrift, die angibt, wie Eingangsdaten **schrittweise** in Ausgangsdaten umgewandelt werden!

Die Folge der Verarbeitungsschritte muss eindeutig festgelegt sein.

Größere Probleme werden dabei in Teilaufgaben und Teilaspekte aufgeteilt.
(Ob der Algorithmus tatsächlich auf dem Papier oder nur im Kopf des Programmierers entwickelt wird, hängt von der Komplexität der Aufgabe und der Genialität des Programmierers ab).

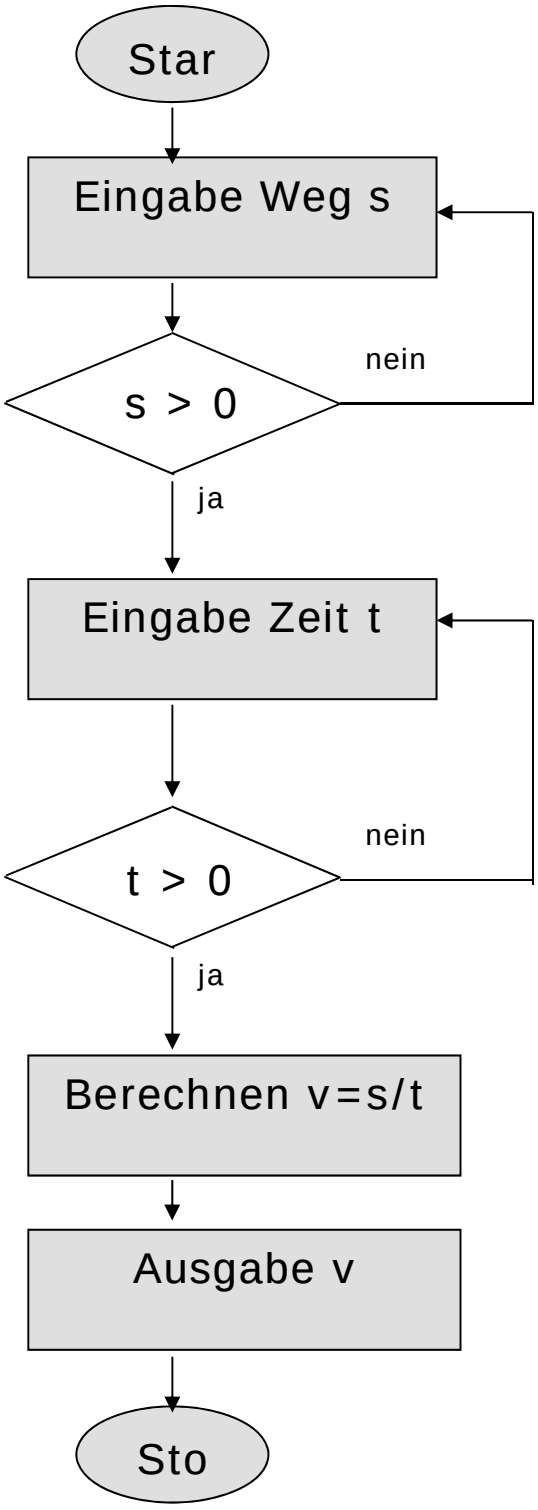
2.2.1 Graphische Darstellungsform eines Algorithmus

Früher wurde ein Algorithmus in Form eines Programmablaufplans (PAP) dargestellt. Heute ist man dazu übergegangen, den Algorithmus als Struktogramm darzustellen.

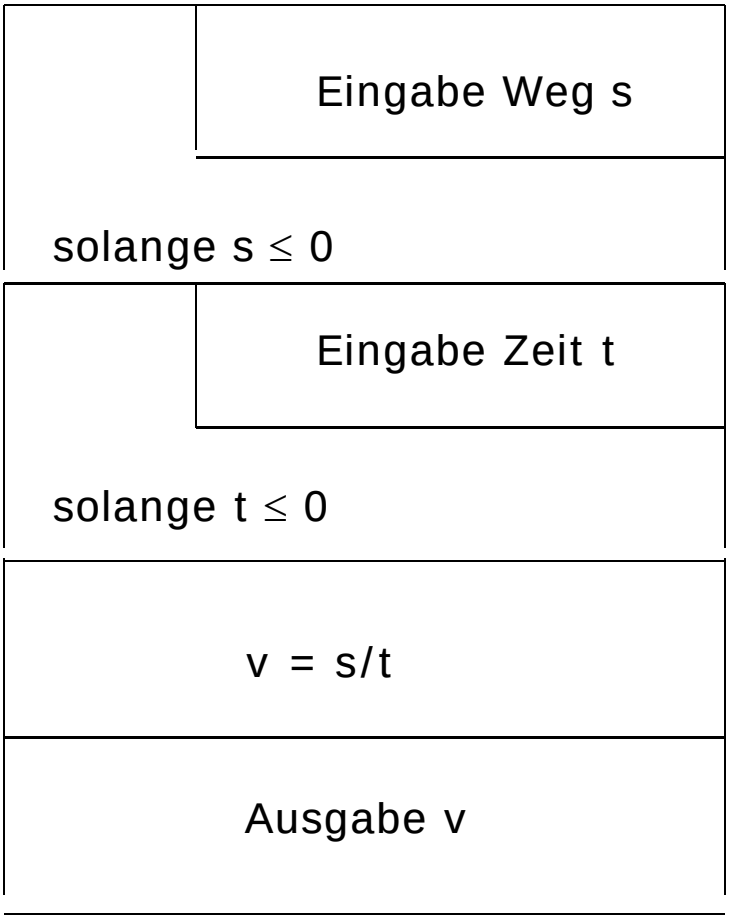
Verbale Beschreibung des Algorithmus
(am Beispiel Geschwindigkeitsberechnung)

- Eingabe von Weg (über Tastatur)
- Prüfung des eingegebenen Weges (> 0), falls nicht > 0 Ausgabe einer entsprechenden Fehlermeldung, neue Eingabe des Weges
- Eingabe der benötigten Zeit (über Tastatur)
- Prüfung der eingegebenen Zeit (> 0), falls nicht > 0 Ausgabe einer entsprechenden Fehlermeldung, neue Eingabe der Zeit
- Berechnung der Geschwindigkeit
- Ausgabe der Geschwindigkeit auf dem Bildschirm

Darstellung des Algorithmus mit PAP



Darstellung des Algorithmus mit Struktogramm



2.3 Erstellen des Quellprogramms

Der Algorithmus wird in für den Computer verständliche Anweisungen einer Programmiersprache umgesetzt. Dies ergibt den sogenannten Quelltext oder Quellcode.

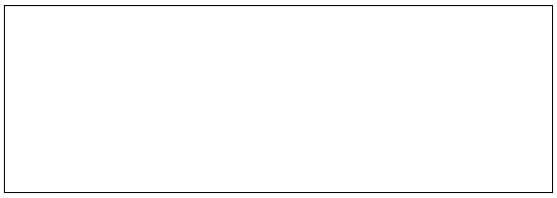
Dieser Quelltext wird dann durch den Compiler in Maschinenanweisungen übersetzt; der Linker „baut“ ein lauffähiges Programm.

2.4 Testen des Programms

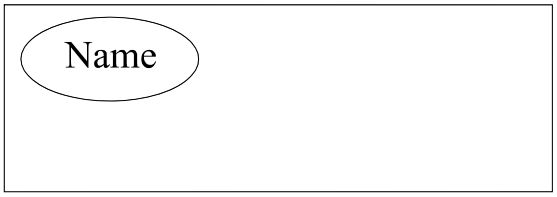
Für den Test des Programms wird es gestartet, d.h. in den Hauptspeicher geladen und vom Prozessor ausgeführt.

Neu entwickelte Programme weisen erfahrungsgemäß mehrere Fehler auf; nachdem die Fehler behoben wurden, muss das Programm ggf. auch mehrfach getestet werden.

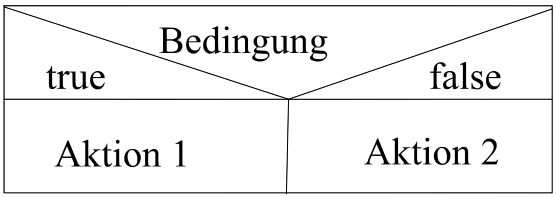
2.5 Strukturelemente des Stuktogramms (DIN 66261)



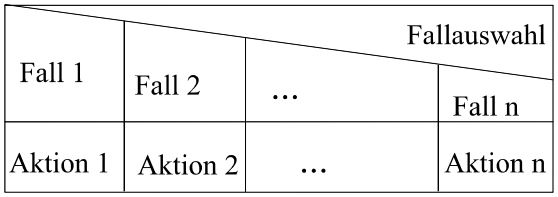
Einfacher Strukturblock



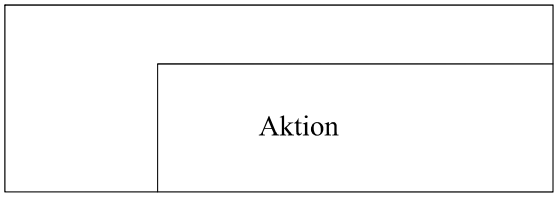
Unterprogrammaufruf
(kein DIN-Element)



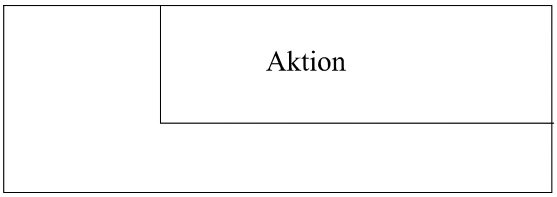
Auswahl



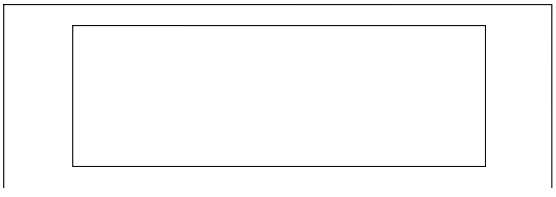
MehrfachAuswahl



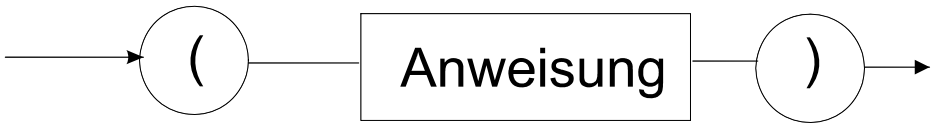
Schleife
(Abbruchbedingung am Beginn)



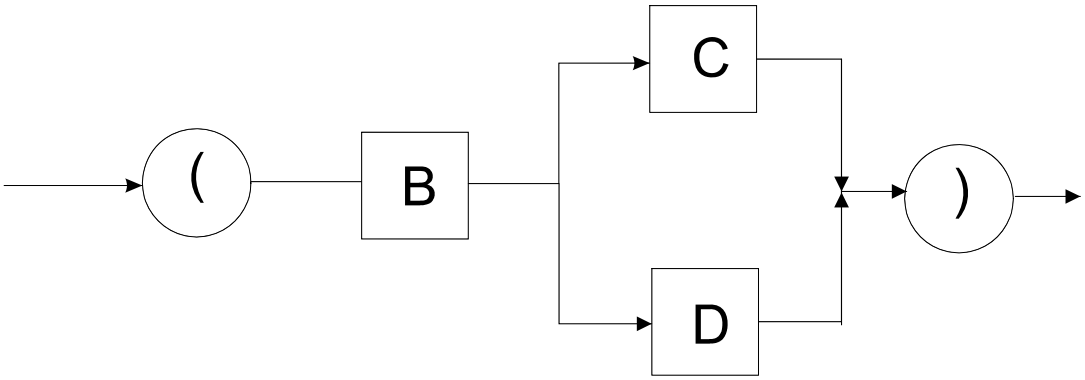
Schleife
(Abbruchbedingung am Ende)



Block
(Programmkennzeichnung)



Da die gesamte Syntax von C++ ein sehr großes Diagramm ergibt, wird es in Teil-diagramme zerlegt.
So bedeutet:



Der Begriff A ist definiert durch (gefolgt von B, gefolgt von C oder D, gefolgt von). Natürlich müssen die Begriffe B, C und D dann anderweitig definiert sein.

2.6 Syntax

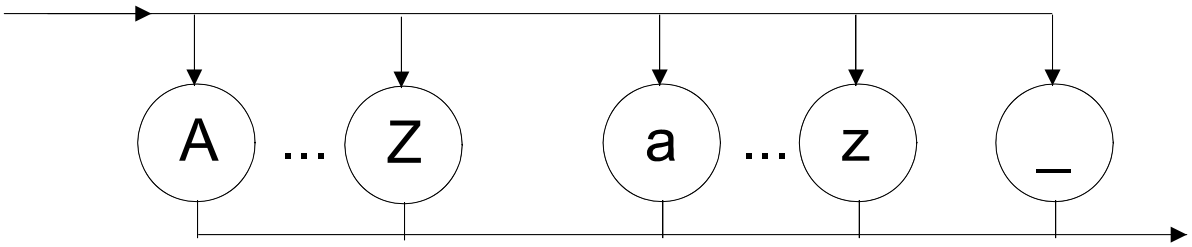
Ein Programm ist eine Folge von Zeichen, die nach bestimmten Regeln aneinandergefügt werden müssen. Diese Regeln nennt man die Grammatik oder Syntax der Sprache. Zur Beschreibung der Syntax sind syntaktische Begriffe notwendig, die definiert werden müssen. Bei Programmiersprachen sind solche Begriffe etwa Zahl, Ausdruck, Prozedur, Anweisung usw. Bei der Definition eines syntaktischen Begriffes können nun sowohl die Zeichen der Sprache als auch andere syntaktische Begriffe vorkommen. Die bei einer Definition verwendeten syntaktischen Begriffe müssen dann ihrerseits wieder definiert werden usw., bis man schließlich nur bei den Zeichen der Sprache angelangt ist.

➔ Die Syntax ist eine formale Beschreibung einer Sprache

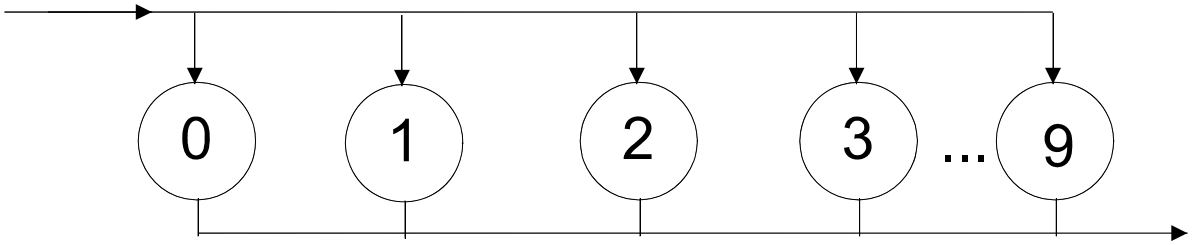
Eine graphische Beschreibung des Syntax hat sich als zweckmäßig erwiesen. Dabei werden die Zeichen der Sprache durch runde oder ovale Symbole, die syntaktischen Begriffe durch rechteckige Kästchen dargestellt. Die Definition besteht dann aus einem Diagramm von runden und eckigen Symbolen, einem gerichteten Graphen, dessen Durchlaufen eine syntaktisch richtige Formulierung ergibt.

2.7 Zeichensatz

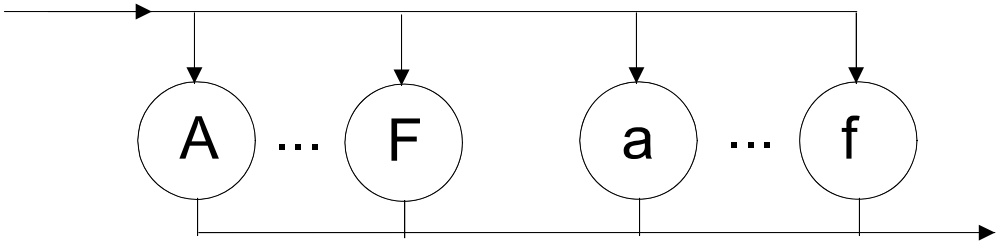
Jedes Programm besteht aus Zeichen, die der Benutzer eingegeben hat. Der zur Verfügung stehende Zeichensatz besteht aus:



Ziffern:

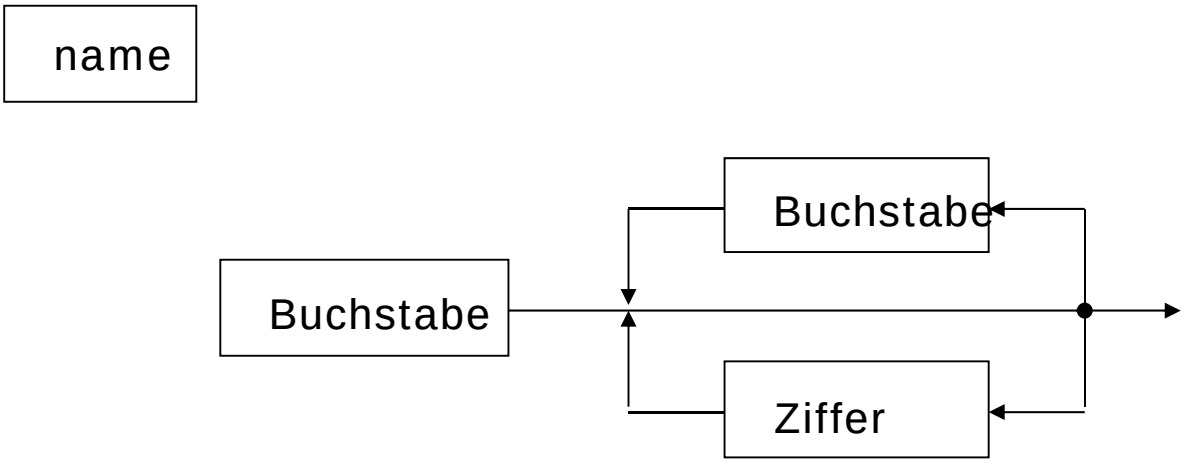


Hexadezimalziffern:



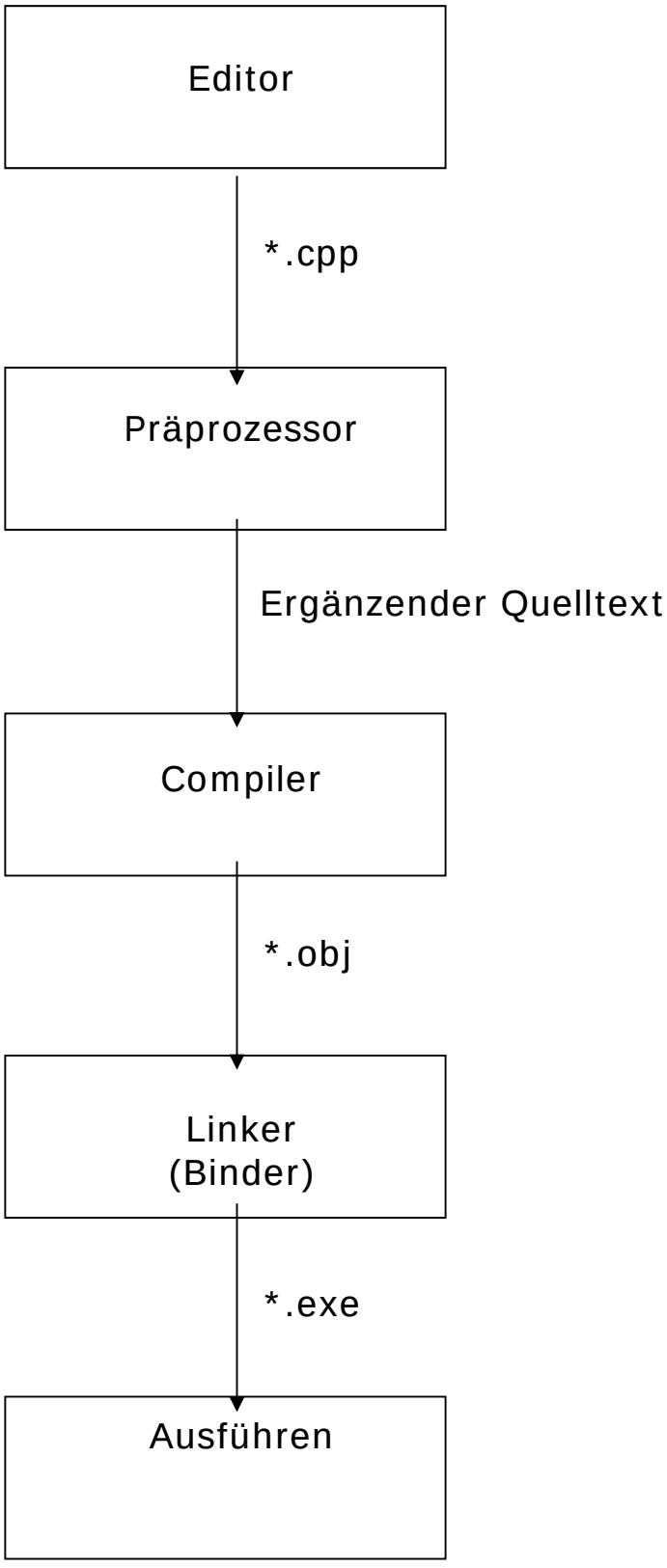
2.8 Namen

- Namen bezeichen Konstanten, Datentypen, Variablen, Funktionen und Methoden
- Namen werden häufig auch Bezeichner genannt (identifier)



- Name beginnt mit einem Buchstaben (oder mit _)
- Buchstaben und Ziffern in beliebiger Reihenfolge
- 32 Stellen sind signifikant
- case sensitive (Groß- Klein-Schreibung wird unterschieden)
- es dürfen keine C++-Schlüsselwörter verwendet werden

2.9 Erzeugung eines lauffähigen C++ Programms



Beispiel: 20 Zahlen von Tastatur einlesen und auf Bildschirm ausgeben

```
/*   Einlesen von 20 Zahlen von der Tastatur
und Ausgababe der Summe dieser Zahlen auf dem Bildschirm
*/

#include <conio.h>
#include <iostream.h>

int main()          // Beginn des Hauptprogramms
{
     double Summe = 0.0;
     double Zahl[20];
     int i;

     clrscr();

     for(i=0; i<20; i++)
     {
          cout << "Bitte geben Sie Zahl("<<i+1<< ") ein: ";
          cin  >> Zahl[i];
     }

     for (i=0; i<20; i++)
          Summe += Zahl[i];

     cout << "Summe: " << Summe;

     getch();
}
```

2.10 Kommentare

C++ kennt zwei Arten von Kommentaren.

// definiert, dass der Rest der Zeile Kommentar ist; dieses Kommentarzeichen kann an jeder Stelle in der Quelle angegeben werden

/* und ***/*** definieren einen evtl. mehrzeiligen Kommentar
Der Kommentar beginnt mit ***/**** und endet mit */. Alles Zeichen zwischen diesen Zeichen, auch mehrere Zeilen, werden ignoriert.

2.11 Präprozessor-Anweisung

```
#include <Dateiname>
```

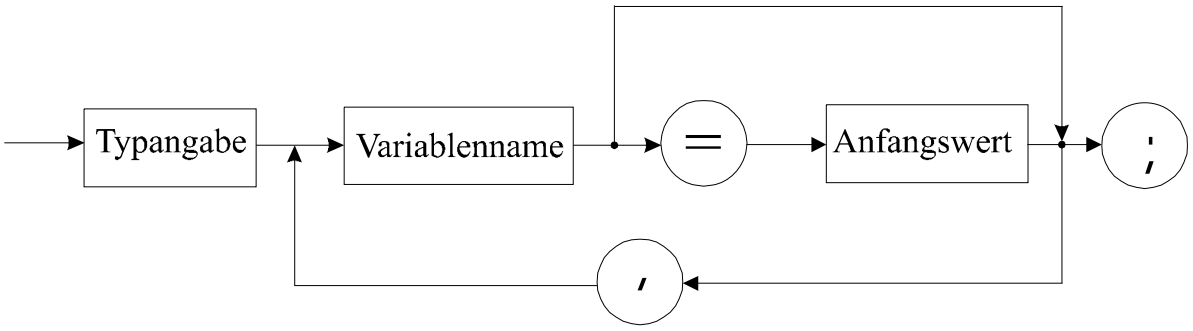
Die Präprozessor-Anweisung `#include` weist den Precompiler an, die angegebene Datei in das Programm einzufügen

Der Dateiname ist der Name einer Prototypendatei (Headerfile *.h)

Wichtige Headerdateien sind:

iostream	Definition für Ein-/Ausgabe in C++
conio.h	Definition für DOS-Funktionen zur Steuerung von Tastatur und Bildschirm
stdio.h	Definition für C-Standard Ein-/Ausgabe-Funktionen

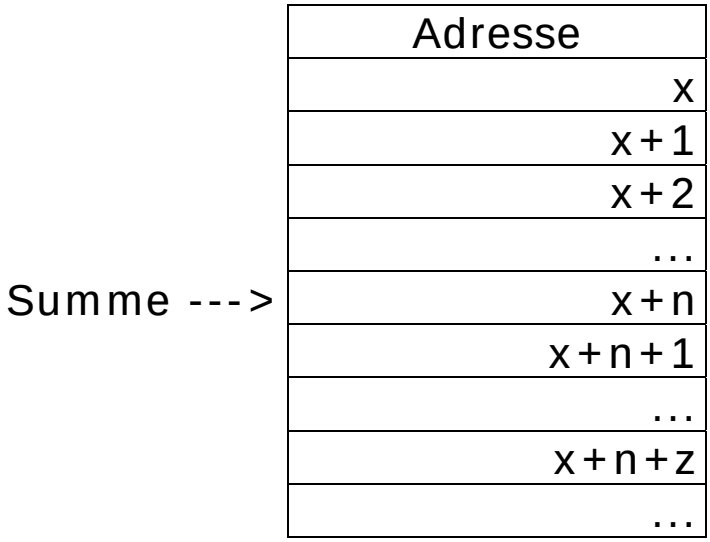
Variablenvereinbarung



Beispiel:
int i;
int i = 1;
int i = 1, j = 2, k = 3;

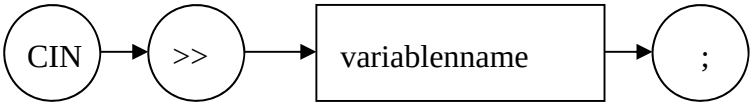
Typenangeabe:
z.B. int (Ganzzahl)
float (Gleitkommazahl)
char (Zeichen)

Arbeitsspeicher

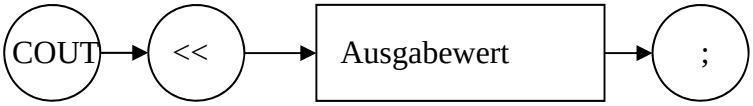


Standardeingabe und –Ausgabe

Eingabe:



Ausgabe:



Ausgabewert kann sein:

- Variable
- Konstante
- Ausdruck

Soll ein Wert in eine Variable eingelesen werden, sollte davor immer ein entsprechender Hinweis ausgegeben werden.

```
cout << „Bitte geben Sie eine ganze Zahl ein:“;
cin >> i;
```

Es können mehrere Texte und Zahlen mit einem Aufruf von cout ausgegeben werden

```
cout << „Die eingegebene Zahl lautet:“ << i << „S Stück!“;
```

Zeilenwechsel bei der Ausgabe

Beispiel:

```
cout << "Wert 1 ist " << i << endl;
cout << "Wert 2 ist " << j << endl;
```

ergibt als Ausgabe:

```
Wert 1 ist 5
Wert 2 ist 6
```

```
cout << "Wert 1 ist " << i << endl << "Wert 2 ist " << j << endl;
```

ergibt ebenfalls als Ausgabe:

```
Wert 1 ist 5
Wert 2 ist 6
```

endl ist eine Konstante, die in der Headerdatei iostream.h definiert ist.
Ebenso sind die Anweisungen `cout` und `cin` in der Headerdatei iostream.h definiert.

Eine weitere Möglichkeit, einen Zeilenwechsel zu erreichen, ist die Ausgabe der Escapesequenz "\n":

```
cout << "Wert 1 ist " << i << "\n" << "Wert 2 ist " << j << "\n";
```

ergibt als Ausgabe:

```
Wert 1 ist 5
Wert 2 ist 6
```

Escapesequenzen:

\n	Zeilenwechsel
\t	Tabulator
\\	Backslash als Zeichen
\”	Anführungszeichen innerhalb eines Textes
\’	Hochkomma innerhalb eines Textes
\0	Nullzeichen

Einlesen eines Zeichens

1. cin
2. getch() Definiert in conio.h
   z.B. c = getch();
   Die Funktion getch() liest ein Zeichen von der Tastatur ein, ohne es auf dem Bildschirm auszugen (kein Echo).
   getch() wartet auf ein Zeichen, z.B. RETURN;

Grundaufbau eines C++ Programms

Ein C++ Programm besteht aus:

1. Anweisungen für den Preprozessor
2. main() Funktion
3. { Funktionskörper}

zu 1) Syntax: `#include <dateiname>`

zu 2) Syntax: `int main()`

Das kleinste C++ Programm lautet demnach: **`int main(){}`**

Es definiert eine Funktion main(), die keine Argumente hat und nichts tut!

Jedes C++ Programm muss eine Funktion mit dem Namen main() enthalten.

Das Programm startet, indem es diese Funktion ausführt. Der int-Wert, den main() gegebenenfalls zurückliefert, ist der Rückgabewert des Programms an das System. Wird kein Wert zurückgegeben, erhält das System einen Wert, der die erfolgreiche Beendigung des Programms anzeigt. Liefert main() einen Rückgabewert ungleich Null, signalisiert das einen Fehler.

zu 3) Syntax: {}

2.12 Zahlensysteme

Dezimalsystem (zur Basis 10): $10^9\ 10^8\ 10^7\ 10^6\ 10^5\ 10^4\ 10^3\ 10^2\ 10^1\ 10^0$

Dualsystem (zur Basis 2): $2^9\ 2^8\ 2^7\ 2^6\ 2^5\ 2^4\ 2^3\ 2^2\ 2^1\ 2^0$

Hexadezimalsystem (zur Basis 16): $16^9\ 16^8\ 16^7\ 16^6\ 16^5\ 16^4\ 16^3\ 16^2\ 16^1\ 16^0$

Darstellung der Zahl 576_{10} ➔ $6*10^0 + 7*10^1 + 5*10^2$

Darstellung der Zahl 576_{16} ➔ $6*16^0 + 7*16^1 + 5*16^2$

$= 6*1 + 7*16 + 5*256$

$= \mathbf{1398_{10}}$

Darstellung der Zahl 576_2 ➔ nicht möglich, da das Dualsystem nur die Werte 0 und 1 kennt!!!

2.12.1 Darstellung der Zahlen

normale natürliche Zahlen: 328, -90

Hexadezimalzahlen: 0x328, 0x0a90

2.13 Fundamentale Datentypen

C++ hat einen Satz von fundamentalen Typen, die mit den verbreiteten grundlegenden Speichereinheiten eines Rechners korrespondieren:

1. ein Boolscher Typ (`bool`)
2. Zeichentypen (`char`)
3. ganzzahlige Typen (`int`)
4. Gleitkommatypen (`float`)

2.13.1 Standarddatentypen

Bei der Deklaration einer Variable muss ihr Typ angegeben werden.

Der Datentyp legt:

- den Wertebereich
- die interne Darstellung
- den internen Speicherplatzbedarf

fest

2.13.2 Ganzzahlen (`integer` ➔ `int`)

Integerzahlen gibt es in drei Größen und zwei Formen:

Die Größen sind: `short int, int, long int`

Die Formen sind: `signed und unsigned`

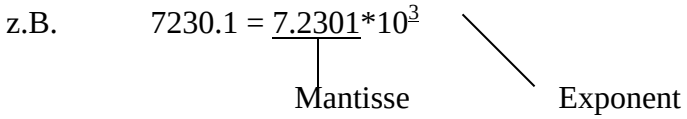
Integerzahlen werden intern als Dualzahlen dargestellt ➔ sie werden exakt dargestellt ➔ es entstehen keine Rundungsfehler.

Datentyp	interne Darstellung		Wertebereich
short int	16 Bit mit Vorzeichen	2 Byte	-32768 - 32767 = -2^{15} - 2^{15}
unsigned short int	16 Bit ohne Voreichen	2 Byte	0 – 65535 = 0 - 2^{16}
int	32 Bit mit Vorzeichen	4 Byte	-2^{31} – 2^{31}-1 = ± 2 Mrd
unsigned int	32 Bit ohne Vorzeichen	4 Byte	0 – 2^{32}-1 = 4 Mrd
long int	32 Bit mit Vorzeichen	4 Byte	-2^{31} – 2^{31}-1 = ± 2 Mrd
unsigned long int	32 Bit ohne Vorzeichen	4 Byte	0 – 2^{32}-1 = 4 Mrd

2.13.3 Dezimalzahlen (reelle Zahlen)

Wie bei den Ganzzahlen, gibt es auch bei den Dezimalzahlen 3 Größen:
float (einfache Genauigkeit), **double** (doppelte Genauigkeit) und **long double** (erweiterte Genauigkeit).

Dezimalzahlen werden intern durch Mantisse und Exponent dargestellt:



Darstellungsformen: -1.0, 12.0, 12.3e-12, 1.5E7, 2.6E+10, .12E1

In einer Gleitkommazahl darf kein Leerzeichen steten.

65.43 e-21 ist keine Gleitkommazahl, sondern vier separate lexikalische Tokens:
65.43 e - 21

Bei der Ausgabe einer Gleitkommazahlen werden 6 Stellen ‚normal' ausgegeben. Hat die auszugebende Zahl mehr als sechs Stellen (vor dem Komma), wird die Ausgabe in Mantisse und Exponent umgewandelt.

Datentyp	intern	Wertebereich	Mantissenstellen
float	4 Byte	± 3.4E-38 – ±3.4E38	≈ 7
double	8 Byte	± 1.7E-308 – ± 1.7E308	15 – 16
long double	10 Byte	± 3.4E-4932 -- ±3.4E4932	≈ 19

Standardmäßig sind Gleitkommazahlen vom Typ `double`.

```
1.23          .23          0.23
```

Wenn man Gleitkommazahlen vom Typ `float` benötigt, kann man es mit dem Suffix **f** oder **F** definieren.

```
1.23f      .23F      0.23f
```

Gleitkommazahlen vom Typ `long` werden mit dem Suffix **l** oder **L** definiert.

```
3.13149265L      2.0L      2.0e-3l
```

2.13.4 Zeichen (char-Datentyp)

Eine Variable vom Typ char kann ein Zeichen aus dem Zeichensatz speichern.
Ein char ist in einfache Hochkommas eingeschlossen!

z.B. `char ch = 'a';`

falsch ist: `char ch = "a";`

Ein char hat acht **Bit** und kann damit einen von 256 verschiedenen Werten speichern.

Üblicherweise ist der Zeichensatz ASCII (American Standard Code for Information Interchange) codiert.

Der ASCII-Zeichensatz besteht aus: Standardwerte (0 – 127)
erweiterter Zeichensatz (128 – 255)

Wobei die Werte	0 – 31	Steuerzeichen
	48 – 57	Ziffern
	65 – 90	Großbuchstaben und
	97 – 122	Kleinbuchstaben

darstellen

Datentyp	intern	Wertebereich
char	8 Bit mit Vorzeichen = 1 Byte	-128 – 127
unsigned char	8 Bit ohne Vorzeichen = 1 Byte	0 – 255

Standardmäßig sind char unsigned!

```
char a = 'ö';
unsigned char b = 'ö';
cout << a << ":" << int(a) << endl;
cout << b << ":" << int(b) << endl;
```

ergibt:

ö:-10
ö:246

2.13.5 Bool-Datentyp

Variablen vom Typ Bool können nur die Werte `true` und `false` annehmen.
Sie dienen zur Darstellung von Wahrheitswerten und werden in logischen Ausdrücken verwendet.

Intern werden die Werte als 1 (`true`) oder 0 (`false`) dargestellt.

Die interne Darstellung ist implementierungsabhängig!

2.14 Operatoren und Ausdrücke

Ein Ausdruck in C++ besteht aus Operanden und Operatoren.

Operanden sind: Konstanten, Variablen, Funktionaufrufe

Operatoren unter liegen einer festgelegten Hirarchie (Rangstufe).

Regeln für die Abarbeitung von Operatoren:

1. Operatoren der kleinsten Rangstufe werden zuerst ausgeführt
2. Operatoren der gleichen Rangstufe werden nach einer festgelegten Abarbeitungs-Reihenfolge ausgeführt (meist von links nach rechts)
3. Das Innere von Klammern wird zuerst ausgeführt

Achtung: Ein Operator kann verschiedene Funktionen haben (z.B. &, +)

2.15 Übersicht der Operatoren von C++

Stufe (Priorität)	Operator		Beispiel	Abarbeitungs-Reihenfolge
0	Klammern	()	(a + b)	links nach rechts
	Funktionsaufruf	(argument)	sin(x)	
	Feldelement	[index]	s[5]	
	Gültigkeitsbereichsoperator (Scope-Operator)	::	ios::replace	
	direkte Komponentenauswahl für Objekte und Strukturen	.	obj.anzeigen()	
	indirekte Komponentenauswahl für Objekte und Strukturen	->	probj->tag	
1	cast-Operator	(Typangabe)	(double)	rechts nach links
	Speicherbedarf	sizeof	sizeof(a)	
	Zeigerverweis	*	*a	
	Adresse	&	&a	
	Inkrement	++	++a oder a++	
	Dekrement	--	--a oder a--	
	negatives Vorzeichen	-	-a	
	positives Vorzeichen	+	+a	
	logische Negation	!	!a	
	Bitkomplement	~	~a	
	dynamische Speicherplatzzuweisung	new	new int	
	Freigabe dynamischer Speicherplatz	delete	delete p	
2	Dereferenzierungszeiger auf ein Klassenelement	.*		links nach rechts
	indirekter Dereferenzierungszeiger auf ein Klassenelement	->*		
3	Multiplikation	*	a * b	links nach rechts
	Division	/	a / b	
	Rest bei ganzzahliger Division (modulo-Operator)	%	a % b	
4	Addition	+	a + b	links nach rechts
	Subtraktion	-	a - b	
5	Shift links	<<	a << n	links nach rechts
	Shift rechts	>>	a >> n	
6	Vergleich	< > <= >=	a < b a > b a <= b a >= b	links nach rechts
7	Gleichheit	==	a == b	links nach rechts
	Ungleichheit	!=	a != b	
8	Bitweises UND	&	a & b	links nach rechts
9	Bitweises exklusives ODER	^	a ^ b	links nach rechts
10	Bitweises ODER	|	a | b	links nach rechts
11	logisches UND	&&	a && b	links nach rechts
12	logisches ODER	||	a || b	links nach rechts
13	bedingt	? :	a > b ? a : b	rechts nach links
14	Zuweisung = += -= /= %= <<= >>= ^= |= &=		a = b a = b = c = 0 s += a	rechts nach links
15	Folge	,	a, b, c	links nach rechts

2.16 Arithmetische Operatoren

Die Operatoren aus Gruppe 3 und 4 sind sogenannte arithmetische Operatoren.

- Multiplikation
- Division
- Rest bei ganzzahliger Division (Modulo-Operator)
- Addition
- Subtraktion

Der **Modulo-Operator** liefert den Rest bei ganzzahliger Division

z.B. liefert 7 % 4 = 3

Bei der Division ist folgendes zu beachten:

- für Dezimaltypen ist das Ergebnis eine Dezimalzahl
- sind die Operanden vom ganzzahligem Typ, so ist das Ergebnis der ganzzahlige Anteil ohne Rest
- Division durch 0 ist nicht erlaubt
- Der %-Operator ist nur auf ganzzahlige Datentypen anwendbar

Beispiele:

```
7 / 4 = 1       // Division von Ganzzahlen -> liefert Ganzzahl
7.0 / 4 = 1.75  // Ist bei der Division ein double-Wert beteiligt,
7 / 4.0 = 1.75  // liefert die Division einen Wert vom Typ double
7.0 / 4.0 = 1.75

int j, i=50;
j = i / 100;    // ergibt 0!
double d;
int i = 50;
d = 2.7 + i / 100;    // ergibt 2.7!
d = 2.7 + (double) i / 100;// ergibt 3.2
```

Ganzzahldivision:

```
 13 /  4 =  3
 13 / -4 = -3
-13 /  4 = -3
-13 / -4 =  3
```

→ bei der Division gelten die mathematischen Regeln:

```
+ * + = +
+ * – = –
– * + = –
– * – = +;
```

Modulodivision:

```
 13 %  4 =  1
 13 % -4 =  1
-13 %  4 = -1
-13 % -4 = -1
```

→ bei der Modulo-Division hat der Rest immer das gleiche Vorzeichen wie der Zähler

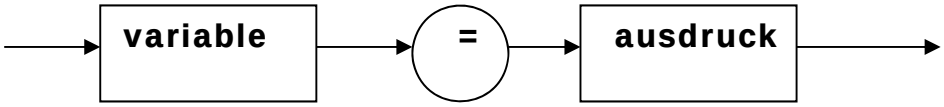
Beispiele zur Umsetzung von Berechnungen in die Programmierung:

$$\frac{x}{y * z} \rightarrow$$

$$\frac{1}{3}x^3 + \frac{7}{5}x^2 + \frac{24}{11} \rightarrow$$

$$\frac{\frac{a}{x} + \frac{2b}{5y}}{1 + \frac{4z}{-2}} \rightarrow$$

2.16.1 Zuweisungsoperator



Die Zuweisung erfolgt immer von rechts nach links!

```
i = i + 1;
```

Im ersten Schritt wird der Ausdruck rechts vom Zuweisungsoperator berechnet (s. Priorität und Abarbeitungs-Reihenfolge) danach wird der berechnete Ausdruck der Variablen links vom Zuweisungsoperator zugewiesen.

Der Ausdruck i = i + 1 ist deshalb zulässig.

Nicht zulässig sind folgende Ausdrücke:

```
i + 1 = i;
7 = a;
a + b = c + d;
```

Der Ausdruck:

```
a = b = c = d = e = 1;
```

ist ebenfalls zulässig, er wird von rechts nach links abgearbeitet (s. Abarbeitungs-Reihenfolge).

Vorsicht bei sehr großen und sehr kleinen Zahlen

Beispiel:

```
void main()
{
   long double a;
   long double b;
   long double c;

   a = 1e20;
   b = 1;

   c = b + a - a;

   cout << c << endl;

   getchar();
}
```

ergibt: 0

2.16.2 Increment – Decrement – Operator

Der Increment-Operator erhöht den Wert der angegebenen Variablen um 1, der Decrement-Operator vermindert den Wert der angegebenen Variablen um 1.

```
x++;  =    x = x + 1;

x--;  =    x = x - 1;
```

Der Decrement- und Increment- Operator sind sogenannte unäre Operatoren, d.h. sie wirken sich nur auf einen Operanden aus.

Es gibt zwei Varianten der Increment- und Decrement- Operatoren:

Prefix-Vaiante: **--x und ++x**

```
void main()
{
   int x = 1;

   cout << ++x << endl;
   cout << ++x << endl;

   getchar();
}
```

ergibt:
2
3

Postfix-Variante: **x-- und x++**

```
void main()
{
   int x = 1;

   cout << x++ << endl;
   cout << x++ << endl;

   getchar();
}
```

ergibt:
1
2

Auch wenn der Inkrement- bzw. Dekrementoperator auf eine double-Variable angewendet wird, ist der Wert des Inkrement bzw. Dekrement immer 1!

```
int i = 1;
double j = 2.4;
char ch = 'a';

cout << i++ << "  " << j++ << "  " << c++ << endl;
cout << i++ << "  " << j++ << "  " << c++ << endl;
cout << i++ << "  " << j++ << "  " << c++ << endl;
```

ergibt:

```
1    2.4  a
2    3.4  b
3    4.4  c
```

Beim Inkrement bzw. Dekrement wird zuerst der Gesamtausdruck ausgewertet, danach wird der Inkrement bzw. Dekrement durchgeführt.

2.16.3 Mathematische Konstanten und Standardfunktionen

definiert in **math.h**

M_PI	π
M_E	Eulersche Zahl e
sin(x)	Sinus
cos(x)	Cosinus
tan(x)	Tangens
atan(x)	Arcus-Tangens
sqrt(x)	Quadratwurzel
exp(x)	e^x
log(x)	natürliche Logarithmus
log10(x)	Logarithmus zur Basis 10
pow(x, y)	x^y

Beispiel:

```
int x = 20;
cout << pow(x, 2) << "\t" << sqrt(x) << endl;
```

ergibt:

```
400   4.47214
```

2.16.4 Logische Operatoren und Vergleichsoperatoren

Logische Operatoren sind auf beliebige Operanden anwendbar, z.B.:

```
'x' > 'a'
5 < 10
6.8 > 1.4
```

Alle Werte die **ungleich 0** sind, sind wahr (true)
Alle Werte, die **gleich 0** sind, sind falsch (false)

```
!     Negation (logisches nicht)
= =   gleich
!=    ungleich
<     kleiner
>     größer
<=    kleiner oder gleich
>=    größer oder gleich
&&    logisches und
||    logisches oder
```

Beispiel:

```
a = 5, b = 10
     a != b     =    true
     a = = b    =    false
     a > b      =    false
     a < b      =    true
     !a         =    0
a = 0
     !a    =    1
```

Wahrheitstabelle und / oder:			
a	b	a && b	a || b
true	true	true	true
true	false	false	true
false	true	false	true
false	false	false	false

`1 < x ≤ 5` → `1 < x && x <= 5`
→ x liegt zwischen 1 (ausschließlich) und 5 (einschließlich)

`7 ≤ x ≤ 20` → `7 <= x && x <= 20`
→ x liegt zwischen 7 (einschließlich) und 20 (einschließlich)

```
antwort = = 'j' || antwort = = 'J'
```
→ Antwort kann 'j' **oder** 'J' sein

```
eingabe > 0 && eingabe < 100
```
→ Eingabe muss größer als 0 **und** kleiner als 100 sein

2.16.5 Zusammengesetzte Operatoren

Grundform:


```
+=    x += 5          →   x= x + 5
-=    x -= 5          →   x= x - 5
*=    x *= 5+3        →   x= x * (5 + 3)
/=    x /= 5          →   x= x / 5
%=    x %= 5          →   x= x % 5
```

2.16.6 sizeof-Operator

Der sizeof-Operator gibt die Größe des Speicherplatzes an, den der angegebene Parameter belegt.

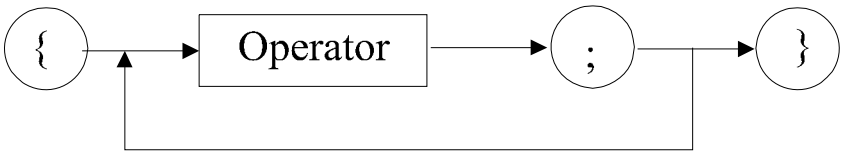
```
int x;
cout << "x           " << sizeof(x) << endl;

cout << "short       " << sizeof(short) << endl;
cout << "int         " << sizeof(int) << endl;
cout << "long        " << sizeof(long) << endl;
cout << "char        " << sizeof(char) << endl;
cout << "float       " << sizeof(float) << endl;
cout << "long double " << sizeof(long double) << endl;
cout << "double      " << sizeof(double) << endl;
cout << "bool        " << sizeof(bool) << endl;
```

ergibt:

```
x           4
short       2
int         4
long        4
char        1
float       4
long double 10
double      8
bool        1
```

2.17 Verbundanweisung

Syntax:



Eine Verbundanweisung fasst mehrere Anweisungen zu einer Anweisung (zu einem Block) zusammen!

Beispiel:

```
{
      int a = 5;
      int b = 6;
      int c = a + b;

      cout << a << "  " << b << "  " << c << endl;
}
```

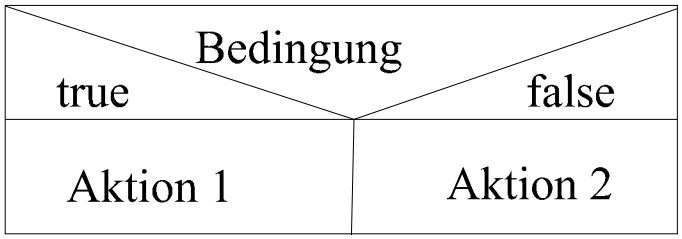
```
cout << a << "  " << b << "  " << c << endl;          →    Fehler
```

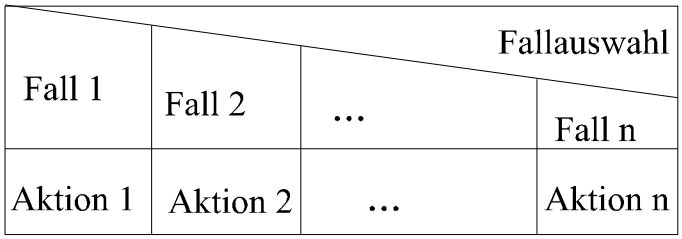
Variablen, die innerhalb eines Blocks definiert sind, sind außerhalb des Blocks nicht bekannt!!!

```
 void main()
{
   int i = 10;
   int j = 20;

   {
      int i;
      int j;

      i = 5;
      j = 6;

      cout << i << "   " << j << endl;          →    5   6
   }

   cout << i << "   " << j << endl;             →    10  20

   getch();
}
```

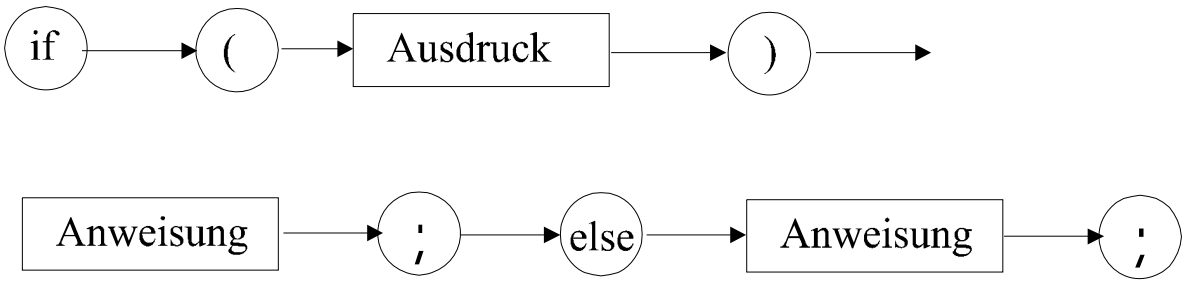
Variablen, die innerhalb einer Verbundanweisung definiert werden, überdecken Variablen, die außerhalb der Verbundanweisung definiert sind!

2.18 Bedingte Anweisungen





2.18.1 If-Anweisung:



Ist der Wert des Ausdrucks ungleich 0 (true) wird die Anweisung direkt nach der runden Klammer ausgeführt; ist der Wert des Ausdrucks = 0 (false) wird die Anweisung nach `else` ausgeführt, fehlt der else-Zweig, wird in diesem Fall keine Anweisung ausgeführt.

Syntax:

```
if (Ausdruck)
      Anweisung;
else
      Anweisung;
```

wobei der `else`-Zweig entfallen kann, er ist optional

Bei dem Ausdruck handelt es sich um einen logischen Ausdruck, der als Ergebnis immer einen boolschen Wert liefert.

Es ist sowohl im `if`- als auch im `else`-Zweig **nur eine Anweisung** erlaubt. Sollen mehrere Anweisungen ausgeführt werden, müssen sie in einer Verbundanweisung zusammengefasst werden.

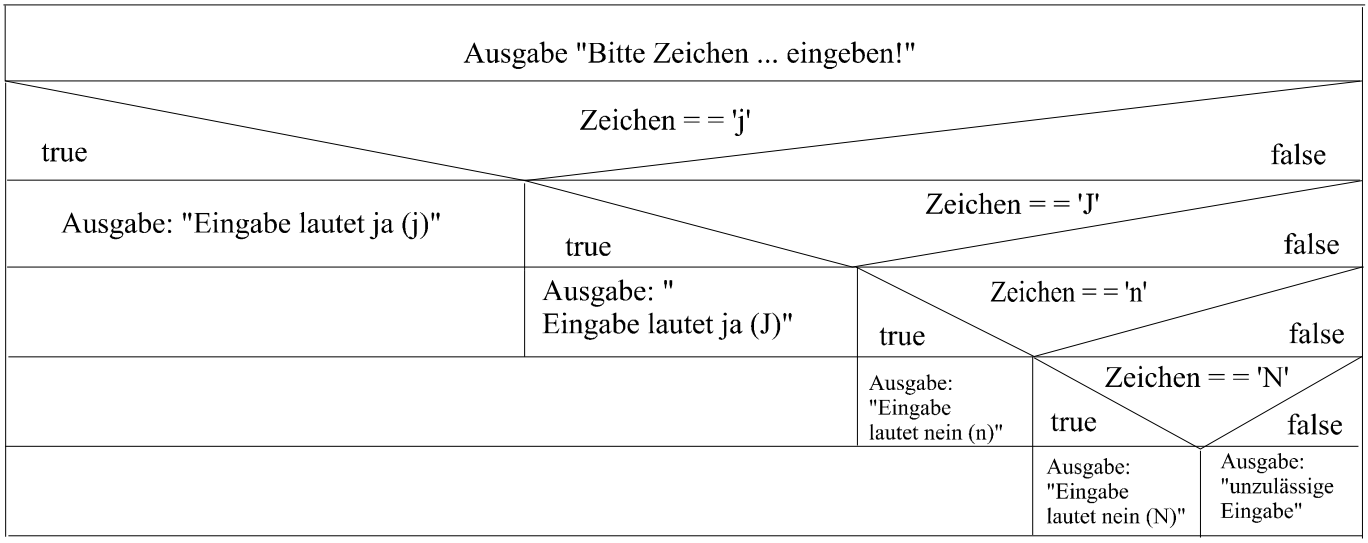
```
#include <conio.h>
#include <iostream.h>

int main()
{
      int i = 9;

  if (i)
     cout << "i ist" << endl;
  else
     cout << "i ist nicht" << endl;

  getch();
}
```

Weitere Beispiele:

```
if (x = = 5)
     j = 3;
else
     j = 10;

if (x = = 5)
{
     j = 3;
     k = 2;
     z = j * k;
}
else
{
     j = 10;
     k = 11;
     z = j / k;
}

/* ------------------------------------------------------------ */
if (x = = 5)
     j = 3;
     k = 2;                →    gehören nicht mehr zur
     z = j * k;                 Anweisung

/* ------------------------------------------------------------ */
if (10 <= i && i <= 20)
   cout << “i liegt zwischen 10 und 20“ << endl;
else
   cout << “i ist kleiner als 10 oder größer als 20“ << endl;

/* ------------------------------------------------------------ */
if (k = = 5 || k = = 10) {
   cout << “k ist 5 oder 10” << endl;
   k = 100;
   cout << ”jetzt ist k = 100“ << endl;
} else {
   cout << “k ist ungleich 5 und ungleich 10” << endl;
   k = 10;
   cout << ”jetzt ist k = 10“ << endl;
}

/* ------------------------------------------------------------ */
if (’a’<= c && c <= ’z’)
{
   cout << “Zeichen ist ein Kleinbuchstabe” << endl;
}
else
{
   if (’A’ <= c && c <= ‘Z’)
      cout << “Zeichen ist ein Grossbuchstabe” << endl;
   else
      cout << “Zeichen ist gar kein Buchstabe” << endl;
}
```

Aufgabe:

Einlesen eines Zeichens von der Tastatur;
Zeichen darf 'j', 'J', 'n ' oder 'N' sein;
Entsprechende Ausgabe auf Bildschirm;
Bei Eingabe eines unzulässigen Zeichens Fehlermeldung

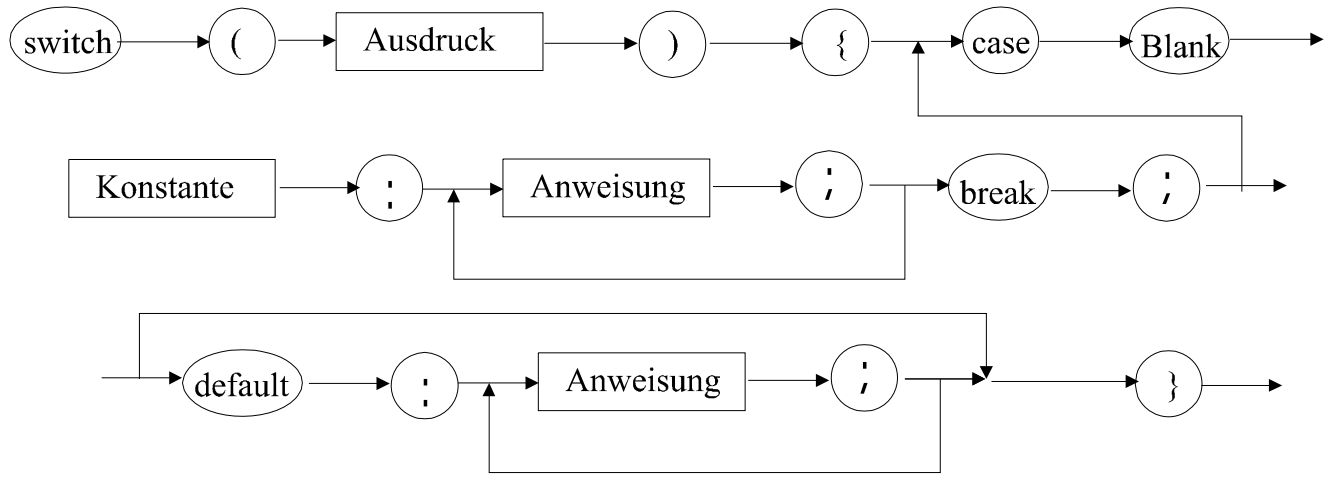


```
char zeichen;

cout << “Bitte Zeichen (’j’, ’J’, ’n ’ oder ’N’) eingeben“ << endl;
cin >> zeichen;

if (zeichen == ’j’)
     cout << “Eingabe lautet ja (j)!“ << endl;
else
     if (zeichen == ’J’)
          cout << “Eingabe lautet ja (J)!“ << endl;
     else
          if (zeichen == ’n’)
          cout << “Eingabe lautet nein (n)!“ << endl;
          else
               if (zeichen == ’N’)
                    cout << “Eingabe lautet nein (N)!“ << endl;
               else
                    cout << “unzulässige Eingabe“ << endl;
getch;
```

oder:

```
char zeichen;

cout << "Bitte Zeichen ('j', 'J', 'n ' oder 'N') eingeben" << endl;
cin >> zeichen;

    if (zeichen = = 'j')
       cout << "Eingabe lautet ja (j)!" << endl;
    else if (zeichen = = 'J')
       cout << "Eingabe lautet ja (J)!" << endl;
    else if (zeichen = = 'n')
       cout << "Eingabe lautet nein (n)!" << endl;
    else if (zeichen = = 'N')
       cout << "Eingabe lautet nein (N)!" << endl;
    else
       cout << "unzulässige Eingabe" << endl;

   getch();
```

2.18.2 Switch-Anweisung (Mehrfachauswahl):



Die switch-Anweisung testet einen Wert gegen eine Menge von Konstanten.

- Die Fallkonstanten müssen verschieden sein.
- Wenn der getestete Wert mit keiner Fallkonstanten übereinstimmt, wird der default-Zweig ausgewählt.
- Der default-Zweig kann auch weggelassen werden.
- nach dem Doppelpunkt können beliebig viele Anweisungen stehen, keine Verbundanweisungen notwendig
- Fallkonstanten dürfen vom Typ int oder char sein
- bei break wird die switch-Anweisung beendet,
  fehlt break, wird die nächste Anweisung ausgeführt

Syntax:

```
switch(Wert)
{
      case Konstante 1:
            Anweisung;
            Anweisung;
            ...
            break;
      case Konstante 2:
            Anweisung;
            Anweisung;
            ...
            break;
      ...
      default:   Anweisung;
            Anweisung;
            ...
}
```

Ausgabe "Bitte Zeichen ... eingeben!"				
Zeichen				
== 'j'	== 'J'	== 'n'	== 'N'	sonst
Ausgabe: "Eingabe lautet ja (j)"	Ausgabe: "Eingabe lautet ja (J)"	Ausgabe: "Eingabe lautet nein (n)"	Ausgabe: "Eingabe lautet nein (N)"	Ausgabe: "unzulässige Eingabe"

Beispiele:

```
char zeichen;

cout << "Bitte Zeichen ('j', 'J', 'n ' oder 'N') eingeben" << endl;
cin >> zeichen;

switch(zeichen)
{
     case 'j':
         cout << "Eingabe lautet ja (j)!" << endl;
         break;
     case 'J':
         cout << "Eingabe lautet ja (J)!" << endl;
         break;
     case 'n':
         cout << "Eingabe lautet nein (n)!" << endl;
         break;
     case 'N':
         cout << "Eingabe lautet nein (N)!" << endl;
         break;
     default:
         cout << "unzulässige Eingabe" << endl;
   }
   getch();
}
```

```
#include <conio.h>
#include <iostream.h>

int main()
{
        int i = 10;

   switch(i == 5)
   {
      case 5:  cout << "i ist 5" << endl;
               break;
      case 10: cout << "i ist nicht 5" << endl;
               break;
      default: cout << "i ist weder 5 noch 10" << endl;
   }
   getch();
}
```

```
#include <conio.h>
#include <iostream.h>

int main()
{
  int i = 10;

   switch(i = 5)
   {
      case 5:  cout << "i ist 5" << endl;
               break;
      case 10: cout << "i ist nicht 5" << endl;
               break;
      default: cout << "i ist weder 5 noch 10" << endl;
   }
   getch();
}
```

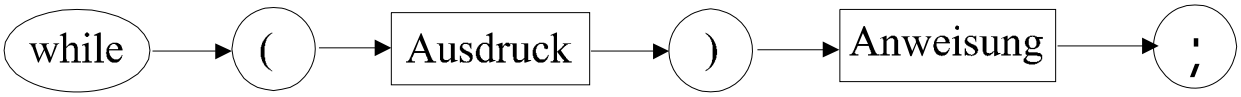
2.19 Schleifenanweisungen

Für wiederholte Ausführungen von Programmcode stehen verschiedene Schleifentypen zur Verfügung:

- die while-Schleife
- die do-Schleife
- die for-Schleife

2.19.1 while-Anweisung

Syntax:



Eine while-Anweisung führt ihre Anweisung so lange aus,
bis ihre Bedingung (ihr Ausdruck) false wird.

1. Berechnung des Ausdrucks
2. Wert des Ausdrucks ungleich 0 (wahr) → Ausführen der Anweisung
   Wert des Ausdrucks gleich 0 (falsch) → Verlassen der Schleife

Sollen mehrere Anweisungen ausgeführt werden → Verbundanweisung

Beispiele:

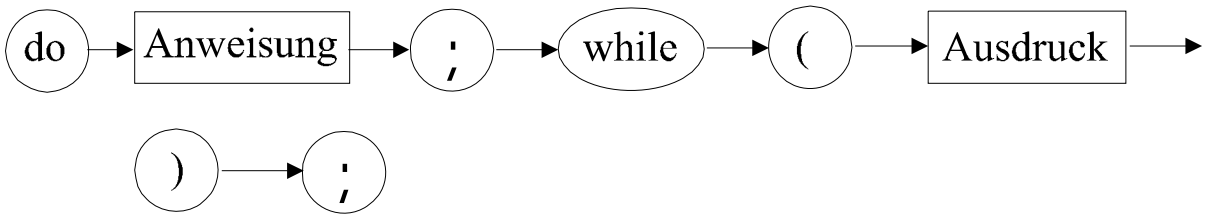
```
int i = 1;
while (i <= 20)
{
      cout << “Wert von i ist: “ << i << endl;
      i++;
}
```

```
int i = 20;
while (i > 0)
{
      cout << “Wert von i ist: “ << i << endl;
      i--;
}
```

```
int i = 20;
while (i > 0)
      cout << “Wert von i ist: “ << i << endl;

→ Endlosschleife, da der Ausdruck nie falsch wird!
```

2.19.2 do-while-Anweisung

Syntax:



1. Ausführen der Anweisung
2. Wert des Ausdrucks ungleich 0 (wahr) → nächster Schleifendurchlauf
   Wert des Ausdrucks gleich 0 (falsch) → Verlassen der Schleife

Beispiele:

```
int i = 1;
do
{
      cout << “Wert von i ist: “ << i << endl;
      i++;
}     while (i <= 20);
```

```
int i = 20;
do
{
      cout << “Wert von i ist: “ << i << endl;
      i--;
}     while (i > 0);
```

```
int i = 20;
do
      cout << “Wert von i ist: “ << i-- << endl;
while (i > 0)
```

```
do
      cout << “Endlosschleife“ << endl;
while (1)

→ Endlosschleife, da der Ausdruck nie falsch wird!
```

2.19.3 Gegenüberstellung while und do-while

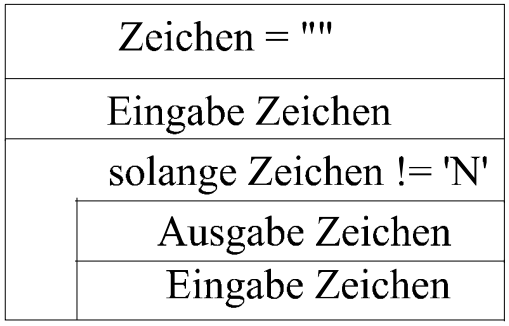
Bei der while-Schleife wird zuerst die Bedingung überprüft, erst dann werden die Anweisungen ausgeführt → die Schleife wird evtl. gar nicht durchlaufen (wenn die Bedingung bereits vor dem ersten Schleifendurchlauf falsch ist)

```
int i=10;
while (i < 10)
{
      cout << “kein Schleifendurchlauf” << endl;
}
```

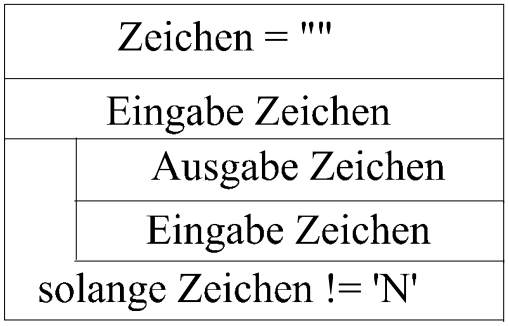
Bei der do-while-Schleife werden zuerst die Anweisungen ausgeführt, erst dann wird die Bedingung überprüft → die Schleife wird mindestens einmal durchlaufen!

```
int i=10;
do
{
      cout << “diese Ausgabe kommt mindestens einmal!” << endl;
}     while (i < 10);
```

while-Schleife



do-while-Schleife

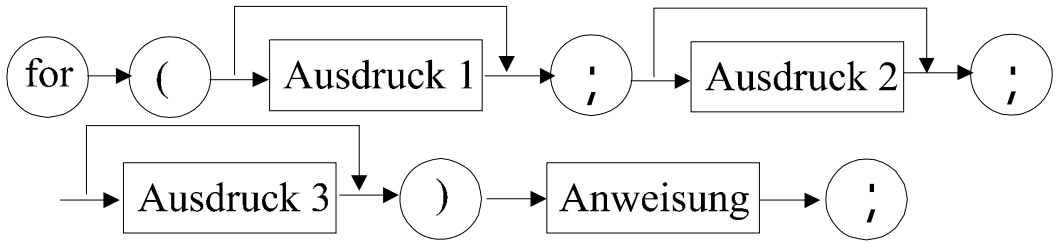


Wenn als erstes Zeichen ein ‚N’ eingegeben wird, wird die erste Schleife nicht durchlaufen, es wird kein Zeichen ausgegeben.

Die zweite Schleife wird genau einmal durchlaufen und das Zeichen ‚N’ wird ausgegeben!

2.19.4 for-Anweisung

Syntax:



Ausdruck 1:
Initialisierungsausdruck; Wird einmal, zu Beginn der Schleife, dausgeführt

Ausdruck 2:
Wiederholungsbedingung; die Schleife wird so lange ausgeführt, bis die Wiederholungsbedingung falsch (false) wird

Ausdruck 3:
Ausführung **nach** jedem Schleifendurchlauf

Abarbeitungsreihenfolge:

1. Ausführung von Ausdruck 1;
2. Prüfen von Ausdruck 2, wenn true → Ausführen von Anweisung, sonst (= false) beenden der Schleife,
3. Ausfürung von Ausdruck 3

Ausdruck 1, 2 oder 3 können jeweils fehlen→ einfachste for-Schleife: for(;;);

Beispiel:

```
for (i=0 ; i < 10 ; i++)
      cout << “Wert von i ist: “<< i << endl;
```

Abarbeitungsreihenfolge:

1. Setzen von i auf 0
2. Prüfen ob i kleine als 10 ist, wenn true → Ausführen von cout, sonst beenden der Schleife
3. Erhöhung von i um 1

```
for (k=1 ; k < 100 ; k++)
      cout << “Wert von k ist: “<< k << endl;
cout << “Wert von k ist: “<< k << endl;

for(i=0, j=10, k=5; i<10, j<20, k>0;i++, j++, k--)
      …
```

2.19.4.1 Abbruch der Schleife: break;

Beispiel:

```
for(;;)
      while(true)
      {
            i++;
            cout << “Wert von i ist: “<< i << endl;
            if (i == 100)
                  break;
      }
```

2.19.4.2 nächster Schleifendurchgang: continue;

Beispiel:

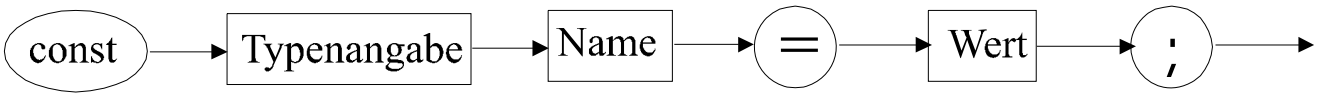
```
for(i=1; i < 100; i++)
{
      if ((i < 10) || (i > 20))
            continue;
      cout << “Wert von i ist: “<< i << endl;
}
```

2.20 Konstanten

C++ bietet das Konzept einer benutzerdefinierten Konstanten, const.

2.20.1 Allgemeines

Schreibweise:



Etwas als const zu definieren stellt sicher, dass sich der Wert nicht mehr ändert

→ Initialisierung zwingend notwendig!

Beispiel:

```
const int ZAHL1 = 90; // zahl1 ist eine Konstante
const int X;    // Fehler, keine Initialisierung
```

Symbolische Konstanten sagen mehr aus als Zahlen.
Ändert sich ein Wert, wird nur die Initialisierung der Konstanten geändert,
kein Durchsuchen des Programms (z.B. MwSt) erforderlich.

Üblich ist, dass **Konstantennamen** mit **Großbuchstaben** angegeben werden

Etwas mit const zu definieren stellt sicher, dass sich der Wert im Gültigkeitsbereich nicht ändert!

```
void fkt()
{
     ZAHL1 = 100;    // Fehler, ZAHL1 ist als const definiert
}
```

Compiler kann konstante Werte direkt in den Programmcode einfügen, zur Laufzeit müssen keine Variablenzuweisungen mehr gemacht werden → schnellere Programmlaufzeiten

2.20.2 Konstanten aus der Headerdatei math.h

```
M_PI       → Pi = 3.14159
M_E        → Eulersche Zahl e = 2.71828
M_SQRT2    → Wurzel aus 2 = 1.41421
M_LN10     → natürliche Logarithmus von 10 = 2.30259
```

M_SQRT2 kann bereits zur Compilierzeit berechnet werden → schnellere Programmlaufzeiten

```
#include <iostream.h>
#include <conio.h>
#include <math.h>

const int X = 100;
const double WURZEL2 = sqrt(2);

void main()
{
   const double ZAHL = 3.9;
   cout << X << endl;
   cout << ZAHL << endl;
   cout << WURZEL2;
   getch();
}
```

Ausgabe:

```
100
3.9
1,41421
```

Noch ein Beispiel:

```
#include <iostream.h>
#include <conio.h>
#include <math.h>

#define SCHUELERZAHL 130

const double WURZEL2 = sqrt(2);
const char NEWLINE = '\n';

void main()
{
   const double zahl = 3.9;
   int dummy;
   cout << zahl << NEWLINE;
   cout << WURZEL2 << NEWLINE;
   getch();
}
```

2.21 Macros

Ein Macro ist eine Präprozessor-Anweisung.

Syntax:

Bei einem Macro wird jedes Auftreten von `Name` in der Quelldatei durch `Zeichenkette` ersetzt!

Macros sind Überbleibsel von C und sollten in C++ so wenig wie möglich verwendet werden!!!

Beispiel:

```
#include <iostream.h>
#include <conio.h>
#include <math.h>

#define SCHUELERZAHL 130
#define BEGIN {
#define END }

// const int x = 100;
const double WURZEL2 = sqrt(2);
const char NEWLINE = '\n';

void main()
BEGIN
   cout << „Anzahl:“ << SCHUELERZAHL << NEWLINE;
   getch();
END
```

→ sehr unsauberer Programmierstil!!!

```
#include <conio.h>
#include <stdlib.h>
#include <iostream.h>

#define AUSGABE cout << "Bitte Buchstaben eingeben!" << endl; cin >> c;

void main()
{
   char c;

   AUSGABE

   cout << "Eingegeben wurde: " << c << endl;
   getch();
}
```

2.22 Spezielle Probleme bei der Ein- und Ausgabe

Standardeingabe:	cin → Tastatur
Standardausgabe:	cout → Bildschirm
Standardfehlerausgabe:	cerr → Bildschirm
Standardprotokollausgabe:	clog → Bildschirm

Beispiel für cerr und clog:

```
#include <conio.h>
#include <iostream.h>

void main()
{
   cerr << "ein Fehler!" << endl;
   clog << "Protokoll" << endl;

   getch();
}
```

Ausgabe:

```
ein Fehler!
Protokoll
```

Und zwar auf dem Bildschirm

Die Standardausgabe und Standardeingabe können bei Aufruf eines Programms umgeleitet werden.

< leitet die Standard**eingabe** um
> leitet die Standard**ausgabe** um

Beispiel:

```
/* prog.exe:
   Programm liest 50 Zeichen von der Standardeingabe ein
   und gibt sie auf der Standardausgabe aus. */
#include <conio.h>
#include <iostream.h>
#include <stdio.h>

void main()
{
   char c;

   for (int i = 0 ; i < 50 ; i++)
   {
      c = getchar();
      cout << c;
   }
}
```

Programmaufruf:

prog	Einlesen von 50 Zeichen von der Tastatur und Ausgabe der Zeichen auf dem Bildschirm
prog < Eingabe	Einlesen von 50 Zeichen aus der Datei Eingabe und Ausgabe der Zeichen auf dem Bildschirm
prog > Ausgabe	Einlesen von 50 Zeichen von der Tastatur und Ausgabe der Zeichen in die Datei Ausgabe
prog < Eingabe > Ausgabe	Einlesen von 50 Zeichen aus der Datei Eingabe und Ausgabe der Zeichen in die Datei Ausgabe

2.23 Steuerzeichen für Ausgabe (Escape-Sequenzen)

Zeichen	Bedeutung
\a	Beep
\b	Löschen Zeichen links (Backspace)
\f	Seitenvorschub (Form Feed)
\n	Zeilenwechsel (\r + \v)
\r	Zeilenrücklauf
\t	Tabulator
\v	Zeilenvorschub
\\	\
\“	“
\’	’

2.24 Manipulatoren

→ Beeinflussung der Ausgabe, z.B. **endl** für Neue Zeile bei Ausgabe

Die Manipulatoren sind in der Headerdatei iomanip.h definiert
(mit Ausnahme von endl).

Weitere Manipulatoren sind:

setw(Weite)	Legt die **Ausgabenbreite des nächsten Ausgabefeldes** fest. Der Wert bezieht sich nur auf das nächste Ausgabefeld! cout << "Zeile 1" << setw(10) << 100 << endl; Die Ausgabe der Zahl 100 wird auf 10 Zeichen Breite vergrößert. die Ausgabe erfolgt rechtsbündig, hat das Ausgabefeld weniger als Weite (hier 10) Zeichen, wird links mit Leerstellen aufgefüllt. Hat das Ausgabefeld mehr als Weite Zeichen, wird die Ausgabebreite angepasst (vergrößert).

Beispiel:

```
cout << "Zeile 1" << setw(10) << 100 << endl;
cout << "Zeile 2" << setw(10) << 3 << endl;
cout << "Zeile 3" << setw(10) << 20 << endl;
```

Ausgabe:

```
Zeile 1       100
Zeile 2         3
Zeile 3        20
```

setfill(Zeichen)	Angabe des Füllzeichens, falls keine Leerzeichen ausgegeben (aufgefüllt) werden sollen. setfill(..) gilt bis zum nächsten Aufruf von setfill(..)

Beispiel:

```
cout << setfill('0') << endl;
cout << "Z1:" << setw(10) << 100 << endl;
cout << "Z2:" << setw(10) << 3 << endl;
cout << "Z3:" << setw(10) << 20 << endl;
```

Ausgabe:

```
Z1:0000000100
Z2:0000000003
Z3:0000000020
```

setprecision(Anzahl)	legt die Anzahl der Dezimalstellen fest. Einstellung gilt bis zum nächsten Aufruf von setprecision!

2.25 IOS-Flags

IOS ist eine Klasse, in der bestimmte IOS-Flags für die Ausgabe definiert sind. Die Flags sind als binärer Wert (Bit) definiert.

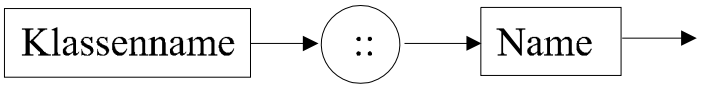
Ist das Flag gesetzt, gilt die entsprechende Eigenschaft;
ist das Flag nicht gesetzt, gilt die entsprechende Eigenschaft nicht!

left	Ausgabe linksbündig
hex	Ausgabe hexadezimal
showpos	Ausgabe von + für positive Zahlen
showpoint	Auffüllen mit Nullen nach dem Dezimalpunkt
uppercase	Ausgabe des Exponentensymbols E und der Hexziffern in Großbuchstaben
fixed	Ausgabe als Dezimalzahl (z.B. 876.123)
scientific	Ausgabe in Exponentialschreibweise (z.B. 8.76123e+2)

Der Aufruf von `setprecision(...)` sollte nur in Zusammenhang mit den IOS-Flags `fixed` oder `scientific` verwendet werden!

2.25.1 Anwenden der Flags

Die Flags sind in der Klasse IOS definiert. Um auf die Flags zugreifen zu können, wird zuerst der Klassenname, dann der Flagname getrennt durch den Gültigkeitsbereichs-Operator (Scope-Operator) angegeben.



Durch bitweises oder können mehrere Flags gesetzt bzw. zurückgesetzt werden!

2.25.2 Setzen der Flags

```
setiosflags(flag-Wert)
resetiosflags(flag-Wert)

cout << setiosflags(ios::showpos | ios::fixed) << setprecision(3);
cout << 10.0;
```

→ +10.000

```
cout << resetiosflags(ios::showpos);
cout << 10.0;
```

→ 10.000

```
cout << setiosflags(ios::scientific);
cout << 10.0
```

→ 1.000e +01

```
cout << setprecision(10);
cout << 10.0;
```

→ 1.0000000000e +01

```
cout << setiosflags(ios::showpos);
cout << 10.0;
```

→ +1.0000000000e +01

2.26 Felder (Arrays)

Beispiel:

10 Zahlen einlesen und in Variablen abspeichern:

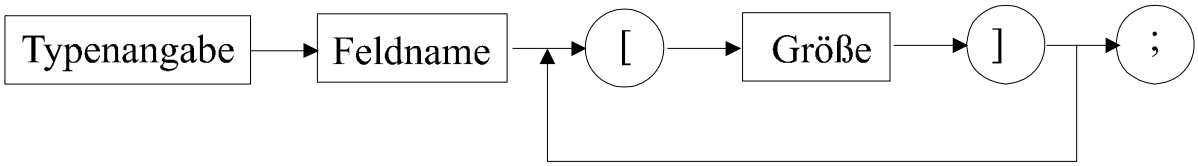
```
int   var0, var1, var2, var3, var4;
int   var5, var6, var7, var8, var9

cout << “Bitte 10 Ganzzahlen eingeben!” << endl;
cin >> var0;
cin >> var1;
cin >> var2;
cin >> var3;
cin >> var4;
cin >> var5;
cin >> var6;
cin >> var7;
cin >> var8;
cin >> var9;
```

→ schlechte Lösung

Ein Feld (Array) ist eine Datenstruktur, die eine Sammlung von Werten des **gleichen Typs** speichert. Auf die einzelnen Werte des Arrays greift man über einen **ganzzahligen** *Index* zu. Handelt es sich zum Beispiel bei `var` um ein Array mit Ganzzahlen, dann ist `var[i]` die *i*te Ganzzahl im Feld `var`.

- die Anzahl der Feld-Elemente ist fest und wird bei der Definition des Feldes festgelegt
- der Zugriff auf die einzelnen Feld-Elemente erfolgt über einen **ganzzahligen** Index
    - der Index des **ersten Elements** des Feldes ist **0**
    - der Index des letzten Elements des Feldes ist (**Größe – 1**)
      → hat ein Feld 10 Elemente, „läuft“ der Index von 0 bis 9
    - Elemente des Feldes lassen sich wie „normale Variablen“ einsetzen

Syntax:



Beispiele:

```
int var[10];
double noten[132];
long int Jahreseinkommen[30];
char Zeichen[300];
bool bestanden[132];
```

var	[0]	Zahl 1
	[1]	Zahl 2
	[2]	Zahl 3
	[3]	Zahl 4
	[4]	Zahl 5
	[5]	Zahl 6
	[6]	Zahl 7
	[7]	Zahl 8
	[8]	Zahl 9
	[9]	Zahl 10

Speicherbedarf von eindimensionalen Feldern:

```
int Zahlen[10]   :  int = 4 Byte      10 Elemente = 4 * 10 = 40 Byte
double dWerte[15]:  double = 8 Byte   15 Elemente = 15 * 8 = 120 Byte
```

Der **Name** des Feldes repräsentiert die **Adresse des ersten Elements**!

```
int Werte[5];   Werte entspricht Adresse von Werte[0]
```

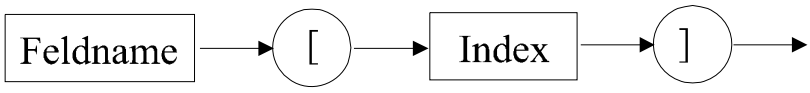
Zuweisung eines Feldes durch Zuweisungsoperator ist nicht möglich!

```
int Werte[5];
int neueWerte[5];
neueWerte = Werte ➔ Fehler!
```

Soll ein Feld in ein anderes Feld kopiert werden, müssen alle Elemente des Feldes einzeln zugewiesen werden!

```
for (int i=0;i < Anzahl Elemente; i++)
     neueWert[i] = Werte[i];
```

2.26.1 Zugriff auf Feld-Elemente



Beispiel:

```
int var[10];

cin >> var[0];
cin >> var[1];
cin >> var[2];
cin >> var[3];
cin >> var[4];
cin >> var[5];
cin >> var[6];
cin >> var[7];
cin >> var[8];
cin >> var[9];

cout  << var[0] << ", " << var[1] << ", " << var[2] << ", "
      << var[3] << ", " << var[4] << ", " << var[5] << ", "
      << var[6] << ", " << var[7] << ", " << var[8] << ", "
      << var[9]
```

→ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

```
int summe;

summe = var[0] + ... + var[9];
```

10 Zahlen einlesen und in Variablen abspeichern:

```
int var[10];

cout << "Bitte 10 Ganzzahlen eingeben! " << endl;

for (index = 0; index < 10; index++)
      cin >> var[index];

for (index = 0; index < 10; index++)
      cout << var[index] << ", ";
```

→ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

2.26.2 Initialisierung von Feldern

Nach der Definition des Feldes (Arrays) sind die einzelnen Elemente nicht initialisiert!!!

```
int var[10];

for (int i=0; i < 10; i++)
     cout << var[i] << “, “;

cout << endl;
```

→ (Ausgabe-Beispiel:
6618600, 4205744, 6618544, 0, 0, 1, -1, 6678640, 6618640,

Die Werte, die ausgegeben werden sind zufällig; abhängig davon, was gerade in dem verwendeten Arbeitsspeicher gespeichert ist!

Wertzuweisung

```
int var[10];

for (int i=0;i < 10; i++)
     var[i] = i;
```

Anfangswertzuweisung

```
int var[i] = {1, 2, 3};

for (int i=0;i < 10; i++)
     cout << var[i] << “, “;
cout << endl;
```

→ 1, 2, 3, 0, 0, 0, 0, 0, 0, 0

Die Felder, für die kein Wert angegeben wurde, werden mit 0 initialisiert!

Es ist nicht möglich, nur bestimmte Elemente zu initialisieren.

```
int zahl[5] = { 1,  ,  5 }
```
→ **nicht zulässig!**

```
int var[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

for (int index = 0; index < 10; index++)
     cout << var[index] << ", ";

cout << endl;
```

→ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

Wird ein Feld ohne Angabe der Größe, jedoch mit einer Initialisierungsliste deklariert, wird vom Compiler die Größe durch Zählen der Elemente in der Initialisierungsliste ermittelt.

```
int var[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
```

→ Größe des Feldes = 10 (Elemente)

```
for (int index = 0; index < 10; index++)
     cout << var[index] << ", ";
```

→ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

A C H T U N G ! ! !
Bereichsüberschreitung 'hinter das Array' wird nicht überwacht!

```
int var[10];

cout << "Bitte 10 Ganzzahlen eingeben!" << endl;

for (int index = 1; index <= 10; index++)
     cin >> var[index];
```

F e h l e r ! ! !

var [0]	Zahl 1
[1]	Zahl 2
[2]	Zahl 3
[3]	Zahl 4
[4]	Zahl 5
[5]	Zahl 6
[6]	Zahl 7
[7]	Zahl 8
[8]	Zahl 9
[9]	Zahl 10
[10]	**Bereich gehört nicht mehr zum Feld!**

Mögliche Folgen der Bereichsüberschreitung:

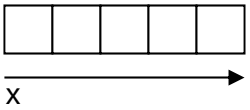
```
int var1[5];
int var2[5];
```

		var1[0]	Zahl 1
		var1[1]	Zahl 2
		var1[2]	Zahl 3
		var1[3]	Zahl 4
		var1[4]	Zahl 5
var2[0]		var1[5]	Zahl 6
var2[1]		var1[6]	Zahl 7
var2[2]		var1[7]	Zahl 8
var2[3]		var1[8]	Zahl 9
var2[4]		var1[9]	Zahl 10
var2[5]		var1[10]	**??**

2.26.3 Dimension des Feldes

Eindimensionales Feld:

```
int zahl[5] ;
```



Mehrdimensionale Felder:

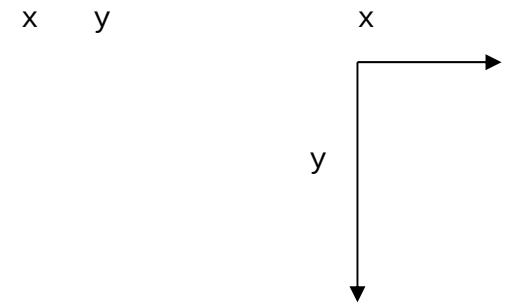
Felder können beliebig viele Dimensionen haben!

```
double Werte[3][5];

bool Zustaende[6][7][9];
```

Von praktischer Bedeutung sind ein- und zweidimensionale Felder!

Zweidimensionales Feld:

```
int zahl[3][5]
```



Zweidimensionale Felder entsprechen einem rechteckigen Schema, das in Zeilen und Spalten organisiert ist (Matrix in der Mathematik).

1. Index = Zeilenindex
2. Index = Spaltenindex

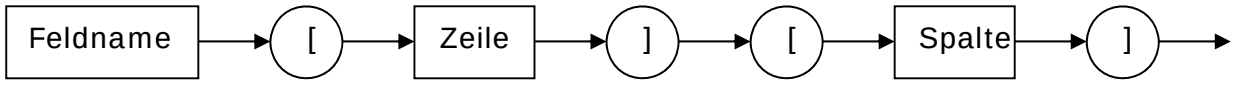
int Matrix[Zeilen][Spalten]

	Spalte 0	Spalte 1	Spalte 2	Spalte 3	Spalte 4
Zeile 0	6	3	1	-8	9
Zeile 1	1	2	3	4	5
Zeile 2	9	1	-2	3	4
Zeile 3	2	6	9	1	-2
Zeile 4	-12	54	9	45	3

Matrix:

$$\begin{Bmatrix} 6 & 3 & 1 & -8 & 9 \\ 1 & 2 & 3 & 4 & 5 \\ 9 & 1 & -2 & 3 & 4 \\ 2 & 6 & 9 & 1 & -2 \\ -12 & 54 & 9 & 45 & 3 \end{Bmatrix}$$

2.26.4 Zugriff auf Elemente eines zweidimensionalen Feldes:



Beim Zugriff auf ein Element eines zweidimensionalen Feldes muss für jede Dimension ein **ganzzahliger** Wert angegeben werden!

```
wert[3][1] = 100;

cout << wert[3][1] << endl;
```

→ Ausgabe: 100

2.26.5 Speicherbedarf von mehrdimensionalen Feldern

Der Speicherbedarf berechnet sich aus der Anzahl der Elemente * Größe des Datentyps.

`int Zahlen[10][80]` int = 4 Byte

10 * 80 = 800 Elemente

4 * 800 = **3200 Byte**

`double dWerte[800][162]` double = 8 Byte

800 * 162 = 129.600 Elemente

8 * 129.600 = **1.036.800 Byte**

Obergrenze für mehrdimensionale Felder ist implementierungsabhängig!

Obergrenze bei Windows 32: **1 MByte (Mega Byte)**

Berechnung:

1 KByte (Kilo Byte) = 1024 Byte

1 Mbyte (Mega Byte) = 1024 KByte = 1024 * 1024 Byte

→ 1.036.800 Byte = 1036800 / (1024 * 1024) = **0,989 MByte**

Initialisierung zweidimensionaler Felder:

```
int Werte[5][5]

Werte[0][0]     = 0;
Werte[0][1]     = 1;
.
.
Werte[0][4]     = 5;
Werte[1][0]     = 2;
Werte[1][1]     = 2;
.
.
Werte[4][0]     = 4;
Werte[4][1]     = 5;
.
.
Werte[4][4]     = 8;

oder:

int Werte[5][5]

for (int i = 0; i < 5; i++)
{
     for (int j = 0; j < 5; j++)
     {
          Werte[i][j] = i + j;
     }
}


for (int i = 0; i < 5; i++)
{
     for (int j = 0; j < 5; j++)
     {
          cout << Werte[i][j] << “, “;
     }
     cout << endl;
}

→    0, 1, 2, 3, 4,
     1, 2, 3, 4, 5,
     2, 3, 4, 5, 6,
     3, 4, 5, 6, 7,
     4, 5, 6, 7, 8,
```

Eine weitere Möglichkeit, zweidimensionale Felder zu initialisieren ist:

```
int Werte[5][5] = { 0, 1, 2, 3, 4,
      1, 2, 3, 4, 5,
      2, 3, 4, 5, 6,
      3, 4, 5, 6, 7,
      4, 5, 6, 7, 8 }


for (int i = 0; i < 5; i++)
{
      for (int j = 0; j < 5; j++)
      {
            cout << Werte[i][j] << “, “;
      }
      cout << endl;
}

→     0, 1, 2, 3, 4,
      1, 2, 3, 4, 5,
      2, 3, 4, 5, 6,
      3, 4, 5, 6, 7,
      4, 5, 6, 7, 8,
```

Bei der Anfangswertzuweisung von zweidimensionalen Feldern werden die Werte zeilenweise angegeben!

Wie bei den eindimensionalen Feldern gilt: Wird nicht für alle Elemente ein Wert angegeben, werden die restlichen Felder mit 0 initialisiert, es ist nicht möglich, nur einzelne Elemente zu initialisieren!

```
int Werte[5][5] = {4};

for (int i = 0; i < 5; i++)
{
      for (int j = 0; j < 5; j++)
      {
            cout << Werte[i][j] << “, “;
      }
      cout << endl;
}

→     4, 0, 0, 0, 0,
      0, 0, 0, 0, 0,
      0, 0, 0, 0, 0,
      0, 0, 0, 0, 0,
      0, 0, 0, 0, 0,
```

Aufgabe:

Entwickeln Sie ein Programm, das zwei ganzen Zahlen von der Tastatur einliest.

Die eingelesenen Zahlen sollen mit sich und den 9 darauffolgenden Zahlen multipliziert werden (ein mal eins), die Produkte werden in einem zweidimensionalen Array abgespeichert.

Danach werden die berechneten Produkte in einer Matrix, die gleichmäßig den Bildschirm füllt, ausgegeben.

Das Programm soll wiederholbar sein.

Verwenden Sie zum Auswerten der Eingabe für Wiederholung eine Mehrfachunterscheidung!

1	2	3	4	5	6	7	8	9	10
2	4	6	8	10	12	14	16	18	20
3	6	9	12	15	18	21	24	27	30
4	8	12	16	20	24	28	32	36	40
5	10	15	20	25	30	35	40	45	50
6	12	18	24	30	36	42	48	54	60
7	14	21	28	35	42	49	56	63	70
8	16	24	32	40	48	56	64	72	80
9	18	27	36	45	54	63	72	81	90
10	20	30	40	50	60	70	80	90	100

Phasen der Programmerstellung (Wiederholung)

Die Entwicklung von Computerprogrammen läuft unabhängig von der verwendeten Sprache üblicherweise nach dem folgenden Muster ab:

Die vier Phasen der Programmentwicklung

1. Problemanalyse
2. Programmentwurf
3. Erstellen des Quellprogramms
4. Testen des Programms

Problemanalyse:

- Welches Problem ist zu lösen
- Welche Angaben werden benötigt?
  - Zahl 1
  - Zahl 2

  (Test, ob eingegebene Werte sinnvoll sind)
- Was soll das Programm tun?
  - Berechnung des ‚ein mal eins' mit den eingegebenen Zahlen
- Ausgabe der berechneten Werte mit erläuterndem Text

Programmentwurf:

Als nächstes wird die Aufgabe als Algorithmus formuliert:

- Der Algorithmus ist eine Verarbeitungsvorschrift die angibt, wie Eingangsdaten **schrittweise** in Ausgangsdaten umgewandelt werden!
- Die Folge der Verarbeitungsschritte muss eindeutig festgelegt sein.
- Größere Probleme werden dabei in Teilaufgaben und Teilaspekte aufgeteilt. (Ob der Algorithmus tatsächlich auf dem Papier oder nur im Kopf des Programmierers entwickelt wird, hängt von der Komplexität der Aufgabe und der Genialität des Programmierers ab).

Graphische Darstellungsform eines Algorithmus

Früher wurde ein Algorithmus in Form eines Programmablaufplans (PAP) dargestellt. Heute ist man dazu übergegangen, den Algorithmus als Struktogramm darzustellen.

<u>Verbale Beschreibung des Algorithmus (am Beispiel ‚ein mal eins')</u>

- Eingabe von Zahl 1 (über Tastatur)
- Prüfung der eingegebenen Zahl
- Eingabe von Zahl 2 (über Tastatur)
- Prüfung der eingegebenen Zahl
- Berechnung der Produkte
- Abspeichern der Ergebnisse in einem zweidimensionalen Array
- Ausgabe der Produkte auf dem Bildschirm

Ausgabe der Programmfunktion, Aufforderung zur Eingabe

Eingabe Zahl 1

Eingabe Zahl 2

i = 0; Solange i < 10

j = 0; Solange j < 10

Ergebnis[i][j] = (Zahl 1 + i) * (Zahl 2 + j)

j um 1 erhöhen

Zahl 1 um 1 erhöhen

i um 1 erhöhen

Bildschirm löschen

i = 0; Solange i < 10

j = 0; Solange j < 10

Positionierung auf Bildschirm

Ausgabe Ergebnis [i][j]

j um 1 erhöhen

i um 1 erhöhen

Frage ob noch ein Durchgang gewünscht ist

Einlesen eines Zeichens von der Tastatur

Fallunterscheidung eingegebenes Zeichen

Zeichen = j oder J	Zeichen = n oder N	sonst
%	Ausgabe Programmende	Fehlermeldung

solange falsches Zeichen eingegeben wurde

Solange noch ein Durchgang gewünscht ist

Erstellen des Quellprogramms

```
#include <conio.h>
#include <stdio.h>
#include <iomanip.h>

void main()
{
  int zahl[10][10];
  int Eingabe_1, Eingabe_2;
  char weiter = 'j';

  do
  {
    clrscr();
    cout << "Multiplikation von 10 * 10 Zahlen" << endl;
    cout << "Bitte erste Zahl eingeben: ";
    cin >> Eingabe_1;
    cout << "Bitte zweite Zahl eingeben: ";
    cin >> Eingabe_2;

    for(int i=0;i < 10;i++)
    {
      for(int j=0;j < 10;j++)
      {
        zahl[i][j] = (Eingabe_1 + i) * (Eingabe_2 + j);
      }
    }

    clrscr();
    for(int i=0; i < 10;i++)
    {
      for(int j=0;j < 10;j++)
      {
        gotoxy((i+1)*7, (j+1)*2);
        cout << setw(5) << zahl[i][j];
      }
    }

    do
    {
      cout << endl << endl << "noch einmal? ";
      cin >> weiter;

      switch(weiter)
      {
        case 'j' :
        case 'J' : break;
        case 'n' :
        case 'N' : cout << "Programm wird beendet" << endl; break;
        default  : cout << "Nur n oder j als Eingabe erlaubt!" << endl;
                   cout << "weiter mit Eingabetaste";
                   getchar();
      }
    } while(weiter != 'n' && weiter != 'N' && weiter != 'j' && weiter !=
'J');
  } while (weiter != 'n' && weiter != 'N');
}
```

Der Algorithmus wird in für den Computer verständliche Anweisungen einer Programmiersprache umgesetzt. Dies ergibt den sogenannten Quelltext oder Quellcode.

Dieser Quelltext wird dann durch den Compiler in Maschinenanweisungen übersetzt; der Linker „baut“ ein lauffähiges Programm.

Testen des Programms!

Für den Test des Programms wird es gestartet, d.h. in den Hauptspeicher geladen und vom Prozessor ausgeführt.

Neu entwickelte Programme weisen erfahrungsgemäß mehrere Fehler auf; nachdem die Fehler behoben wurden, muss das Programm ggf. auch mehrfach getestet werden.

2.27 Zeichenketten (Strings)

Zeichenketten (Strings) sind eindimensionale char-Felder!

```
char Zeichenkette[10];
```

definiert ein char-Feld mit 10 Zeichen ➔ Zeichenkette!

Zeichenkette[x] bezeichnet eine einzelnes Zeichen aus dieser Zeichenkette (ein Buchstaben aus dem String), für x = 0 – 9 .

Anfangszuweisung für einen String

```
char Wort[5] = “hallo“;

char Satz[] = “dies ist ein Beispielsatz“;
```

In C++ wird eine Zeichenkette (ein String) **automatisch** durch ein sog. NULL-Zeichen abgeschlossen

```
NULL = ’\0’ = 0
```

’\0’ definiert das Ende der Zeichenkette!

```
char Wort[5] = “hallo“;
```

➔ Bereichsüberschreitung!!!

richtig ist:

```
char Wort[6] = “hallo“;
```

Wichtig:

char-Felder, die eine Zeichenkette speichern sollen, müssen ein Zeichen länger als benötigt definiert werden, da in C++ automatisch ein ’\0’ zur Ende-Begrenzung angehängt wird! Das Ende einer Zeichenkette wird durch ein NULL-Zeichen ’\0’ markiert!

➔ eine Zeichenkette mit der Länge 0 gibt es nicht!

```
char xyz[] = “”; // Definition eines Leerstrings
```

2.27.1 Ein- Ausgabe von Zeichenketten

Die Ausgabe erfolgt (wie gewohnt) mit cout.

Beispiel:

```
void main()
{
   char c[6] = "world";

   cout << "Hello " << c << endl;
}
```

➔ **Hello world**

Die **Eingabe** von Zeichenketten erfolgt über die Standard-Eingabeklasse `istream`.

`cin` und `cout` sind Standardobjekte dieser Standard-Eingabeklasse.

Mit der Methode `getline` des Objekts `cin` werden Zeichenketten über die Tastatur eingelesen.

Syntax:

```
cin.getline(Variablenname, Anzahl, Endzeichen);
```

→ liest maximal `Anzahl` Zeichen bis zum ersten Auftreten von `Enzeichen` von der Tastatur ein und speichert die eingelesenen Zeichen in `Variablenname` ab.

Standardmäßig ist das Endzeichen `'\n'`.

Das Endzeichen kann beim Methodenaufruf weggelassen werden!

Beispiel:

```
char c[20];
cin.getline(c, 10, 'a');
```

Liest max. 10 Zeichen von der Tastatur ein und speichert sie in dem Feld `x` ein.

Beispiel:

```
cin.getline(c, 10, '\n');
```

=

```
cin.getline(c, 10);
```

Eingabe über Tastatur:

Hello world

➔ in Zeichenkette c:

Hello wor

```
cin.getline(c, 10, 'o');
```

Eingabe über Tastatur:

Hello world

➔ in Zeichenkette c:

Hell

2.27.2 Standardfunktionen für Zeichenketten

Standardfunktionen für Zeichenketten sind in der Headerdatei `string.h` definiert

`strlen(string)`	liefert die Länge der Zeichenkette `String` ohne Endekennzeichen; Return-Wert = `unsigned int`
`strcpy(Ziel, Quelle)`	Kopiert dem String `Quelle` in den String `Ziel` (incl. NULL-Zeichen)
`strcmp(String_1, String_2)`	Vergleicht zwei Zeichenketten Return-Werte: `< 0` `String_1`ist kleiner als `String_2` (`String_1` steht lexikographisch vor `String_2`) `= 0` `String_1` und `String_2` sind lexikographisch gleich `> 0` `String_1` ist größer als `String_2` (`String_1` steht lexikographisch nach `String_2`)
`strncmp(String_1, String_2, Anzahl)`	Vergleicht die ersten `Anzahl` Zeichen von `String_1` mit `String_2` Return-Wert : s.o.
`strcat(String_1, String_2)`	`String_2` wird an `String_1` angehängt (incl. Null-Zeichen)

Bei allen String-Funktionen muss darauf geachtet werden, dass die Speicherbereiche groß genug definiert werden!

Beispiele:

```
cout << strlen("Hello world") << endl;
```

➔ 11

```
void main()
{
   char Ziel[20];
   char Quelle[20] = "Hello World";

   strcpy(Ziel, Quelle);

   cout << Ziel << endl;

   getchar();
}
```

➔ Hello World

```
void main()
{
   char String_1[20] = "Hello";
   char String_2[20] = "World";

   cout << strcmp(String_1, String_2) << endl;

   getchar();
}
```

➔ Ausgabe kleiner 0

```
void main()
{
   char String_1[20] = "Hello";
   char String_2[20] = "World";

   strcat(String_1, String_2);
   cout << String_1 << endl;

   getchar();
}
```

➔ HelloWorld

```
#include <conio.h>
#include <stdio.h>
#include <iomanip.h>

void main()
{
   char x[]= "ABCDEFGHI";
   char y[] = "";

   cout << setiosflags(ios::hex);

   for (int i=0;i < 10;i++)
      cout << (int) x[i] << " ";
   cout  << endl;

   strcpy(x, "Hallo");

   for (int i=0;i < 11;i++)
      cout << (int) x[i] << " ";
   cout  << endl;

   cout << x << endl;
   strcpy(y, "Welt");
   cout << y << endl;
   cout << x << endl;

   getchar();
}
```

Ausgabe:
41 42 43 44 45 46 47 48 49 0

48 61 6c 6c 6f 0 47 48 49 0

Hallo
Welt
elt

```
cout << (long) x << endl;

cout << (long) y << endl;
```

64fdf8

64fdf7

2.27.3 Felder von Zeichenketten

```
#include <conio.h>
#include <stdio.h>
#include <iomanip.h>

void main()
{
   char Monat[12][10] = {"Januar", "Februar", "Maerz", "April",
"Mai", "Juni", "Juli", "August", "September", "Oktober", "November",
"Dezember"};

   for (int i = 0; i < 12; i++)
      cout << Monat[i] << endl;

   cout << endl;


   for (int i = 0; i < 12; i++)
   {
      for(unsigned int j=0;j < strlen(Monat[i]);j++)
         cout << Monat[i][j];
      cout << endl;
   }


   getchar();
}
```

Januar
Februar
Maerz
April
Mai
Juni
Juli
August
September
Oktober
November
Dezember

Januar
Februar
Maerz
April
Mai
Juni
Juli
August
September
Oktober
November
Dezember

2.28 Typenumwandlung

Es ist oft notwendig, dass Daten eines bestimmten Typs in einen anderen Datentyp umgewandelt werden müssen.

Dies geschieht

1. automatisch (implizit)
2. gezielt (explizit)

2.28.1 Implizit: automatische Typenumwandlung

Die automatische Typenumwandlung wird immer dann ausgeführt, wenn in einem Ausdruck verschiedene Datentypen verwendet werden.

Dabei werden alle Ausdrücke (von links nach rechts) in den ‚kompliziertesten' Datentyp umgewandelt.

Beispiel:

```
double dx, dy;
float fx;

dx = 1.234;
fx = 4.3;

dy = dx / fx;

cout << dy << endl;
```

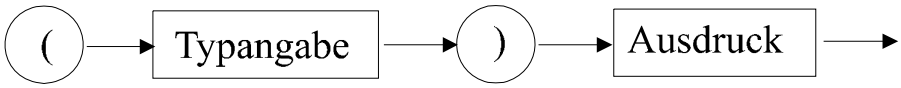
Beispiel:

```
double dx;

dx = 1 / 2;
cout << dx << endl;
```

Ergebnis: 0

2.28.2 Explizit: der Cast-Operator

Syntax:



Der Ausdruck wird in den angegebenen Typ umgewandelt.

Beispiel:

```
double dx;

dx = 1 / (double) 2;
cout << dx << endl;

Ergebnis : 0.5
```

```
double dx;

dx = (double) (1 / 2);
cout << dx << endl;

Ergebnis : 0
```

Sonderformen

```
dx = 1.234567890123456789f;

ix = 3.0;

dx = 1 / (double) 2    ➔    dx = 1 / 2.0
```

A C H T U N G !

Bei der Typenumwandlung können Informationsverluste auftreten!!!

z.B. `fx = (float) dx;`

Beispiel:

```
double dx;
float fx;

// formatieren der Ausgabe

cout << setprecision(20);
cout << setiosflags(ios::fixed) << endl;

dx = (double) 1.234567890123456789;
cout << dx << endl;

fx = (float) dx;
cout << fx << endl;
```

Ergebnis :
1.23456789012345669000
1.23456788063049316400

Informationsverlust, da float nicht so viele Nachkommastellen darstellen kann wie double.

```
int ix;
unsigned uix;

ix = -100;

uix = (unsigned) ix;

cout << uix << endl;
```

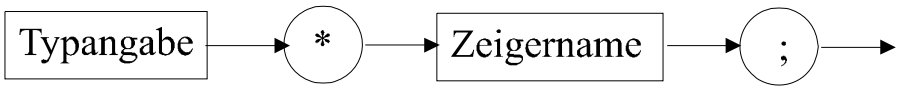
Ergebnis :

4294967196

3 Zeiger-Datentyp

Ein Zeiger (Pointer) ist eine Variable, die auf eine Adresse im Arbeitsspeicher-Bereich **zeigt**. Ein Zeiger wird durch Angabe des Verweisoperators ***** vereinbart.

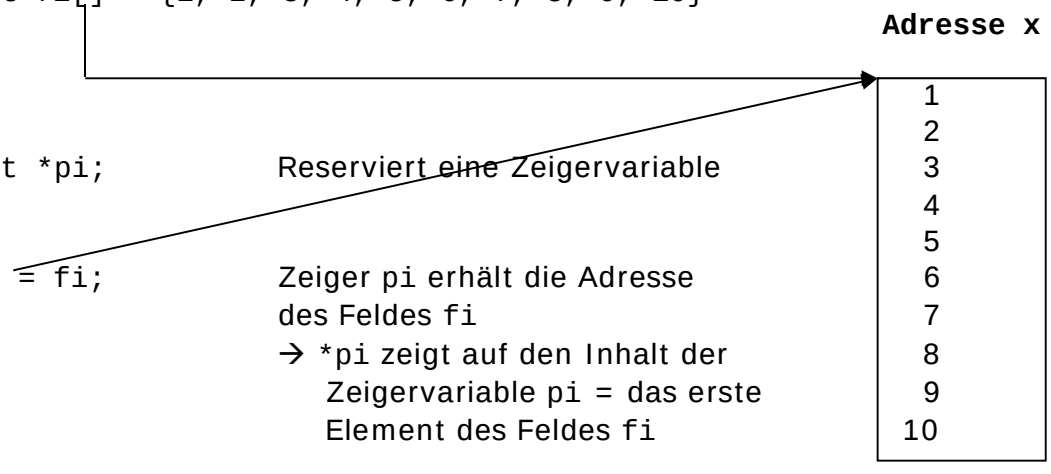
Syntax:



Anfangswertzuweisung ist nicht möglich!

Beispiel:

```
int *pi;         // Zeiger auf int-Variable
double *pd;      // Zeiger auf double-Variable
bool *pb;        // Zeiger auf boolean-Variable
char *pc;        // Zeiger auf character-Variable


int fi[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
// Vereinbarung eines Feldes mit Anfangswertzuweisung

int *pi;   // Vereinbarung eines Zeigers auf eine int-Variable

pi = fi;   // Zuweisung der Adresse des Feldes fi an die
// Zeigervariable pi (der Name eines Feldes ohne
// Angabe einer Indizierung liefert die Adresse des Feldes)

int fi[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
```

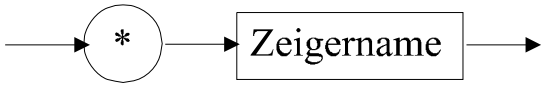



```
int *pi;          Reserviert eine Zeigervariable

pi = fi;          Zeiger pi erhält die Adresse
                  des Feldes fi
                  → *pi zeigt auf den Inhalt der
                     Zeigervariable pi = das erste
                     Element des Feldes fi

cout << fi << endl << pi << endl;      // → Ausgabe der Adresse
cout << fi[0] << endl << *pi << endl; // → Wert des ersten Elements
```

➔ Die Zeigervariable **pi** speichert die **Adresse** des Feldes `fi`, d.h. sie zeigt auf das Feld `fi`
wird die Zeigervariable mit dem Verweisoperator * verwendet, (***pi**) zeigt sie auf den **Inhalt** der Adresse des Zeigers (im Beispiel der Inhalt des 1. Elements des Feldes `fi`)

3.1 Verweisoperator

Syntax:



Liefert den Inhalt der Adresse, auf die `Zeigername` zeigt!

Beispiel:

```
int *pi;         // Definition der Zeigervariablen pi
cout << pi;      // Ausgabe der Adresse der Zeigervariablen pi
cout << *pi;     // Ausgabe des Wertes, auf den die Zeigervariable pi
zeigt
```

Anfangswertzuweisung von Zeigervariablen

```
int *pi = fi;
```

➔ Definition des Zeigers `pi` vom Typ Integer und Zuweisung der Adresse des Feldes `fi`

A U S N A H M E:

Anfangswertzuweisung mit NULL

```
int *pi = NULL;
```

Korrekte Anfangswertzuweisung ➔ NULL-Zeiger (NULL-Pointer)!

3.2 NULL-Zeiger

NULL ist eine Konstante, die in der Headerdatei stddef.h definiert ist.

Es ist empfehlenswert, eine Zeigervariablen bei der Vereinbarung mit NULL zu initialisieren, da ein NULL-Zeiger im Programm abgefragt werden kann, und somit sicher gestellt werden kann, dass ein Zeiger vor seiner Verwendung mit einem sinnvollen Wert initialisiert ist.

Beispiel:

```
int *pi;

cout  << *pi << endl;
```

→ **schlecht,** da pi nicht initialisiert ist. Somit ist nicht festgelegt, auf welche Adresse pi zeigt, evtl. Laufzeitfehler, da unberechtigter Speicherzugriff

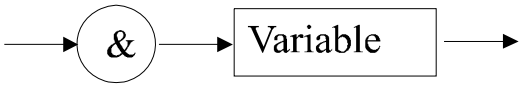
```
int *pi = NULL;

if (pi != NULL)
   cout  << *pi << endl;
else
   cout  << "pi ist nicht initialisiert" << endl;
```

→ **guter Programmierstil,** da sicher gestellt ist, dass auf pi nicht zugegriffen wird, solange er nicht explizit initialisiert wurde.

3.3 Adressoperator

Jede Variable ist an einer Adresse gespeichert!
Auf die Adresse einer Variablen kann man mit dem Adressoperator zugreifen.

Syntax:



liefert die Adresse von `Variable`

Beispiel:

```
int i;
cout << &i;
```

gibt die Adresse der Variablen i aus

```
int fi[10];
cout << &fi[0] << endl << fi << endl;
```

gibt zwei mal die Adresse des Feldes `fi` aus!!!

```
int *pi;
int i = 123;
pi = &i;
cout  << *pi << endl;
```

Ausgabe: 123

3.4 Zeiger auf Felder

Der Name eines Feldes entspricht der Adresse des ersten Elements des Feldes = Adresse des Feldes ➔ **Der Feldname ist ein Zeiger**

```
int *pi=NULL;     // Definition der Zeigervariablen pi
int f[] = { 1, 2, 3, 4, 5};

pi = f;

cout << pi << endl;
cout << f << endl;
```

Ausgabe der Adresse des Feldes im Arbeitspeicher

```
cout << f[0] << endl;
cout << *f << endl;
```

Ausgabe des Wertes des ersten Elements des Feldes

Wiederholungen:

Zeiger-Datentyp:

Pointervereinbarung

```
int *p = NULL;
```

Vereinbarung eines Zeigers auf einen Integer-Wert mit Anfangswertinitialisierung mit NULL

Der Inhalt einer Zeigervariable ist immer eine Adresse (im obigen Beispiel die Adresse einer Integer-Variablen)

→ eine Zeiger belegt **immer** 4 Byte im Arbeitsspeicher (bei PCs unter dem Betriebssystem Windows), egal auf was für einen Datentyp er zeigt!!!

Verweisoperator *

Der Verweisoperator besagt, dass nicht der Inhalt des Zeigers gemeint ist (→ keine Adresse) sondern der Inhalt der Speicherstelle, auf die der Zeiger zeigt!

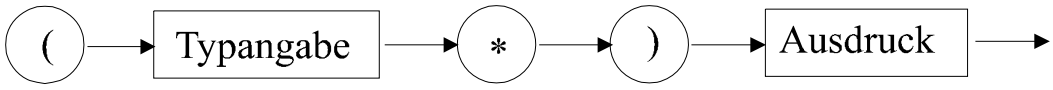
Adressoperator &

Der Adressoperator wird dazu verwendet, die Adresse einer Speichervariablen zu erhalten.

Cast

Der Cast-Operator wird dazu verwendet, eine explizite Typenumwandlung durchzuführen.

Sowie ein Cast-Operator auf einen Standarddatentyp angewendet werden kann, kann er auch auf einen Zeiger-Datentyp angewendet werden!



3.4.1 Übungen mit Zeigern

```
#include <stdio.h>
#include <iostream.h>
#include <math.h>
#include <iomanip.h>

void main()
{
   int vi[] = {1, 2, 3, 4, 5, 6, 7, 8, 9};
   int *p1 = vi;


   cout << " P1: " << p1 << endl;
   cout << "*P1: " << *p1 << endl;
   cout << "&P1: " << &p1 << endl;

   getchar();
}
```

```
 P1: 0x0064fde0
*P1: 1
&P1: 0x0064fddc
```

```
#include <stdio.h>
#include <iostream.h>
#include <math.h>
#include <iomanip.h>

void main()
{
   int i = 256;
   int * p1 = &i;

   p1 = &i;

   cout << " &i: " << &i << endl;          // liefert &i: 0x0064fdfc

   cout << " P1: " << p1 << endl;
   cout << "*P1: " << *p1 << endl;
   cout << "&P1: " << &p1 << endl;

   getchar();
}
```

```
 P1: 0x0064fdfc
*P1: 256
&P1: 0x0064fe00
```

```
#include <stdio.h>
#include <iostream.h>
#include <math.h>
#include <iomanip.h>

void main()
{
   int i = 256;
   int * p1 = &i;
   char *pc;

   pc = &i;

   cout << "*pc:" << *pc << endl;
   cout << "&pc:" << &pc << endl;

   getchar();
}
```

liefert Compilerfehler, da die Adresse einer Integer-Variable nicht direkt einem Character-Pointer zugewiesen werden darf!

```
#include <stdio.h>
#include <iostream.h>
#include <math.h>
#include <iomanip.h>

void main()
{
   int i = 256;
   int * p1 = &i;
   char *pc;

   pc = (char *) &i;

   cout << "*p1:" << *p1 << endl;
   cout << "*pc:" << *pc << endl;

   getchar();
}
```

***p1:256**
***pc:**_

```
#include <stdio.h>
#include <iostream.h>
#include <math.h>
#include <iomanip.h>

void main()
{
   int i = 256;
   int * p1 = &i;
   char *pc;

   pc = (char *) &i;

   cout << "      *p1:" << *p1 << endl;
   cout << "(int) *pc:" << (int) *pc << endl;

   getchar();
}
```

```
      *p1:256
(int) *pc:0
```

```
#include <stdio.h>
#include <iostream.h>
#include <math.h>
#include <iomanip.h>

void main()
{
   int i = 65;
   int * p1 = &i;
   char *pc;

   pc = (char *) &i;

   cout << "      *p1:" << *p1 << endl;
   cout << "(int) *pc:" << (int) *pc << endl;
   cout << "      *pc:" << *pc << endl;

   getchar();
}
```

```
      *p1:65
(int) *pc:65
      *pc:A
```

```
#include <stdio.h>
#include <iostream.h>
#include <math.h>
#include <iomanip.h>

void main()
{
   int i = 65;
   int * p1 = &i;
   char *pc;

   pc = (char *) &i;

   cout << "      *p1:" << (char) *p1 << endl;
   cout << "(int) *pc:" << (int) *pc << endl;
   cout << "      *pc:" << *pc << endl;

   getchar();
}
```

```
     *p1:A
(int) *pc:65
     *pc:A
```

Kleinste adressierbare Einheit

Abhängig von der verwendeten Hardware

bei Intel-Prozessoren:

kleinste adressierbare Einheit:

1 Byte	=	8 Bit
2 Byte	=	1 Wort
4 Byte	=	1 Doppelwort
16 Byte	=	1 Paragraph
1 KByte	=	1024 Byte
1 MByte	=	1024 KByte
1 GByte	=	1024 MByte
1 TByte	=	1024 Gbyte

Größe von Zeigervariablen

Die Größe von Zeigervariablen ist implementierungsabhängig (abhängig vom adressierbaren Speicherbereich)!

Adressierung bei Intel-Prozessoren

Historie

Die ersten Intellprozessoren (8080, 8085) konnten 64 KByte addressieren.

64 KByte = 64 * 1024 Byte = 65535 Byte

1 Bit	2 Adressen	0, 1	2^1 Bit
2 Bit	4 Adressen	0, 1, 2, 3	2^2 Bit
...	...	...	...
8 Bit	256 Adressen	0, 1, 2, ..., 255	2^8 Bit
...	...	...	...
16 Bit	65536 Adressen	0, 1, 2, ..., 65535	2^{16} Bit

➔ für die Adressierung von 64 KByte benötigt man eigentlich 16 Adressleitungen

Bei der Weiterentwicklung der Prozessoren achtete Intel immer darauf, dass die Abwärtskompatibilität gewahrt wurde!

➔ theoretisch funktionieren Programme, die für den Intel 8080-Prozessor entwickelt wurden auch auf den neuen Pentium-Prozessoren

Real-Mode der Intel-Prozessoren

Im **Real-Modus** von Intel Prozessoren (für DOS-Kompatibilität) wird der Arbeitsspeicher in 64 KByte große Segmente unterteilt.

Eine Speicherstelle innerhalb eines Segments wird durch den Offset angegeben.

Daraus ergibt sich, dass eine Speicheradresse durch eine zweigeteilte Adresse angegeben wird: Der **Segmentadresse** und der **Offsetadresse**.

Die Angabe von Speicherbereichen und –Adressen erfolgt üblicherweise in hexadezimaler Darstellung.

Darstellung der hexadezimalen Zahlen in C++ durch Voranstellen von `0x`

Darstellung der Zahl 0x576 ➔ $6*16^0 + 7*16^1 + 5*16^2$

$= 6*1 + 7*16 + 5*256$

$= \mathbf{1398_{10}}$

Zur Darstellung der Wertigkeiten 10, 11, 12, 13, 14 und 15 werden im Hexadezimalsystem die Ziffern a, b, c, d, e und f verwendet.

Beispiel:

3a29:52b5 → Segmentadresse 3a29

Offsetadresse 52b5

Berechnung der absoluten Adresse:

Speicher ist aufgeteilt in 64 KByte große Blöcke. Welcher Speicherblock angesprochen werden soll, wird durch die Segmentadresse angegeben.

→ die Adresse liegt im Speicherblock Nummer `0x3a20`

→ dieser Speicherblock beginnt bei der absoluten Adresse:

$0x3a20 * 16_{10}$ = 0x3a20 * 0x10

Berechnung:

Beispiel Dezimalsystem:

$1000_{10} * 10_{10} = 10000$ → Verschiebung um eine Stelle nach links

0x3a20 * 0x10 = 0x3a200 → Verschiebung um eine Stelle nach links

→ Multiplikation mit 0x10 im Hexadezimalsystem = Verschiebung um eine Wertigkeit nach links

absolute Adresse = Multiplikation der Segmentadresse mit 0x10 + Offsetadresse

```
3a29:52b5 →          3a290
                   + 52b5
                   ------
                     3f545
```

➔ eine Adresse belegt 2 * 2 Byte Speicherplatz (2 Byte für Segmentadresse + 2 Byte für Offsetadresse) = **4 Byte**

Eine physikalische Adresse kann durch mehrere Segment-Offset-Kombinationen angesprochen werden:

Beispiel:

```
1000:1FFF          ➔          10000
                            + 1FFF
                            ------
                             11FFF

1100:0FFF          ➔          11000
                            + 0FFF
                            ------
                             11FFF
```

3.5 Zeigerarithmetik

Auf Zeiger sind die Operatoren + (Addition), – (Subtraktion), ++ (inkrement), -- (dekrement), +=, -= (zusammengesetzte Operatoren) und die Vergleichsoperatoren anwendbar.

Die mathematische Operation beschränkt sich auf ganze Werte!

Da nur bei Feldern die Reihenfolge der Speicherung bekannt ist (die einzelnen Elemente der Felder liegen nacheinander im Arbeitsspeicher) hat die Verwendung von Zeigerarithmetik nur bei Felder einen Sinn!

Beispiel:

```
int fi[] = { 1, 2, 3, 4, 5};
int *pi;
pi = fi;
pi++;
```

Erhöhung des Zeigers um Byte;

```
char fc[] = {‘a’, ‘b’, ‘c’, ‘d’, ‘e’};
char *pc;
pc = fc;
pc++;
```

Erhöhung des Zeigers um Byte;

Die Addition um 1 zu einem Zeiger erhöht den Zeiger um die Größe eines Elements, auf das der Zeigers zeigt!

Bei jeder Rechenoperation auf einen Zeiger verändert sich der Zeiger immer abhängig von dem Datentyp, auf den er zeigt!

A C H T U N G : Bereichsüberschreitung verhindern

Beispiel:

```
double fd[] = {1.0, 2.0, 3.0, 4.0, 5.0};
double *pd;

pd = fd;

cout << *pd << endl;
pd++;
cout << *pd << endl;
pd += 3;
cout << *pd << endl;
```

Aufgabe:

Welche Ausgabe liefert folgendes Programmstück?

```
int fi[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
int *pi;

pi = fi;

cout << "Zeiger: " << *pi << " Feld: " << fi[0] << endl;


*pi += 8;
cout << "Zeiger: " << *pi << " Feld: " << fi[0] << endl;


pi += 5;
cout << "Zeiger: " << *pi << " Feld: " << fi[1] << endl;

&pi += 1;
cout << "Zeiger: " << *pi << " Feld: " << fi[2] << endl;
```

Beispiel:

```
char fc[] = {'a', 'b', 'c', 'd', 'e'};
char *pc;

pc = fc;

cout << *pc << endl;
pc++ ;
cout << *pc << endl;
pc += 3;
cout << *pc << endl;
```

Subtraktion zweier Zeiger

Die Subtraktion zweier Zeiger ist nur innerhalb eines Feldes sinnvoll; hierbei muss darauf geachtet werden, dass der Speicherbereich des Feldes nicht überschritten wird.

Beispiel:

```
int feld[10];
int *pi1;
int *pi2;

pi1 = feld;
pi2 = &feld[5];

pi2 – pi1 = 5
```

Prioritäten

Die Zeigeroperatoren haben eine höhere Priorität als arithmetische Operatoren.

```
int fi[] = { 10, 20, 30, 40, 50};
int *pi = fi;

cout << *pi +3 ;      // Ausgabe : 13
cout << *(pi +3) ;    // Ausgabe : 40
```

Zeigeroperatoren und Inkrement- und Dekrementoperatoren haben die gleiche Priorität. Es sollte deshalb vermieden werden diese Operatoren in einem Ausdruck zu mischen.

```
cout << *pi++;  // ungünstig
```

besser:

```
cout << *pi;
pi++ ;
```

3.6 Zeiger und Strings (Zeichenketten)

Zeiger auf Strings sind ‚ganz normale' Zeiger mit einigen Besonderheiten:

3.6.1 Vereinbarung von Zeichenketten:

```
char str1[100];
char *str2;
```

Bei beiden Vereinbarungen ist eine Anfangswertzuweisung möglich:

```
char str1[100] = ”Teststring”;
char *str2 = ”Teststring”;
```

→ Reservierung von Arbeitsspeicher für die festgelegte Größe (str1 = 100 Byte, str2 = 11 Byte)

Beide Zeichenketten können mit cout ausgegeben werden

```
cout << str1 << endl;
cout << str2 << endl;
```

Es wird nicht die Adresse der Variablen ausgegeben, sondern die Zeichenkette, auf die die Variablen zeigen.

Ein char-Zeiger kann während der Programmlaufzeit neu zugewiesen werden

```
char *str1 = „Teststring”;
.
str1 = ”Stringtest”;
```

char-Zeiger dürfen während der Programmlaufzeit gleichgesetzt werden

```
char *str1 = "Teststring";
char *str2;


cout << str1 << endl << str2 << endl;

str2 = str1;

cout << str1 << endl << str2 << endl;
```

```
void main()
{
   char *str1 = "Text 1";
   char *str2 = "String 2";

   cout << str1 << endl << str2 << endl;
   if (str1 == str2)
      cout << "sind gleich" << endl;
   else
      cout << "sind nicht gleich" << endl;

   str2 = str1;
   cout << str1 << endl << str2 << endl;
   if (str1 == str2)
      cout << " sind gleich" << endl;
   else
      cout << " sind nicht gleich" << endl;

   getchar();
}
```

Ausgabe:

```
Text 1
String 2
sind nicht gleich
Text 1
Text 1
sind gleich
```

3.6.2 Nachteile von char-Zeigern

Sollen Zeichenketten von der Tastatur eingelesen werden, sind char-Zeiger eher ungeeignet!

Wird dennoch eine Zeichenkette in einen Zeiger eingelesen, ist darauf zu achten:

- dass der Zeiger initialisiert werden muss (wurde der Zeiger nicht initialisiert, schreibt `cin` die eingelesene Zeichenkette an die Stelle im Arbeitsspeicher, auf die der Zeiger **zufällig** zeigt
  → ‚fremder' Arbeitsspeicher wird überschrieben!
- dass der bisher reservierte Bereich nicht verlassen wird
  → Bereichsüberschreitung!

Beispiele:

```
char *str2;
cin.getline(str2, 20);
```

➔ fatal, da der Zeiger str2 nicht initialisiert wurde und auf irgend einen zufälligen Bereich im Arbeitsspeicher zeigt

```
char *str1 = "Teststring";
cin.getline(str1, 20);
```

➔ fatal, da für Zeiger str1 nur 11 Zeichen reserviert wurden, und mit dem getline evtl. 19 (+ 1) Zeichen eingelesen werden

```
char *str1 = "Kette 1";
char *str2 = "String 2";

cout << str1 << endl << str2 << endl << "Eingabe: ";

cin.getline(str1, 20);

cout << str1 << endl << str2 << endl;
```

Anzeige am Bildschirm:

```
Kette 1
String 2
Eingabe: abcdefghijk
abcdefghijk
ijk
```

Noch ein Zeiger-Beispiel:

```
#include <iostream.h>
#include <stdio.h>
#include <iomanip.h>

int main()
{

   char *ptr = "Hello World";

   cout << ptr << "\t" << (int) ptr << endl;

   ptr = "weiter gehts";
   cout << ptr << "\t" << (int) ptr << endl;

   ptr = "C++ ist toll";
   cout << ptr << "\t" << (int) ptr << endl;

   getchar();
}
```

Ausgabe:

```
Hello World     40e074
weiter gehts    40e082
C++ ist toll    40e091
```

4 Funktionen

`main` ist die (Haupt-) Funktion eines Programms; nur die main-Funktion eines Programms kann vom Betriebssystem aufgerufen werden ➔ ein lauffähiges C++ Programm **muss** über die Funktion **`main`** verfügen, es **kann** auch über **beliebig viele** weitere Funktionen verfügen!

Weitere Merkmale von Funktionen:

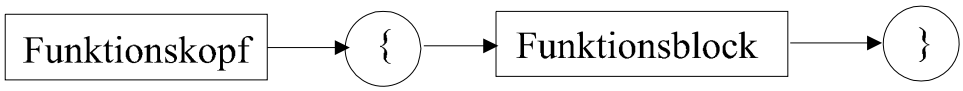
Eine Funktion ist ein abgeschlossener Teil eines C++ Programms

Mit Hilfe von Funktionen kann ein C++ Programm in kleinere, übersichtlichere Teile zerlegt werden (es wird strukturiert) auch wird mit Funktionen erreicht, dass mehrfach auftretende Teilprobleme nur einmal programmiert werden müssen.

Der Datenaustausch (Kommunikation) zwischen den einzelnen Funktionen (Programmteilen) geschieht über eine feste Schnittstelle; Schnittstellen bilden die Eingangs- und Ausgangsparameter.

4.1 Grundaufbau einer Funktion

Syntax:



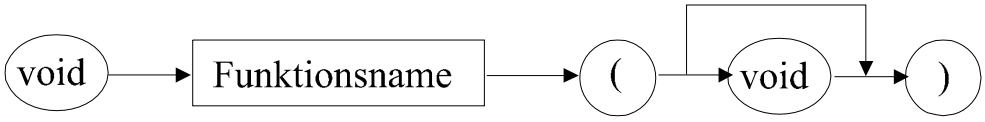
Der Funktionskopf beschreibt die Schnittstelle zur Umgebung:

- Funktionsname
- Datentyp des Rückgabewertes
- Parameterliste für Datenübergabe

Der Funktionsblock stellt die eigentliche Funktion mit Vereinbarung von lokalen Variablen und den Anweisungen dar.

4.2 Funktionen ohne Parameter

Syntax:



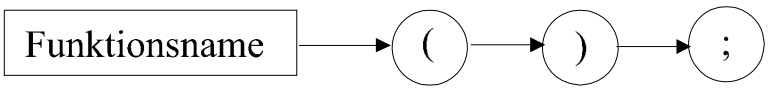
void vor dem Funktionsname bedeutet, dass die Funktion keinen Rückgabewert liefert, void als Parameterliste bedeutet, dass die Funktion keinen Parameter erwartet.

Funktionen können von anderen Funktionen oder von sich selbst aufgerufen werden.

Ruft eine Funktion sich selber auf, spricht man von **Rekursion**.

Funktionsaufruf

Syntax



Eine Funktion kann erst aufgerufen werden, wenn sie im Programm bekannt ist
➔ die Reihenfolge des Quelltextes ist ausschlaggebend.

Bei größeren Programmen kann es sehr schwierig bis unmöglich sein, die Funktionen in die Reihenfolge zu bringen, dass sie obige Anforderungen erfüllen:

Beispiel:

Funktion a() ruft Funktion b() auf

Funktion b() ruft Funktion c() auf

Funktion c() ruft Funktion a() auf

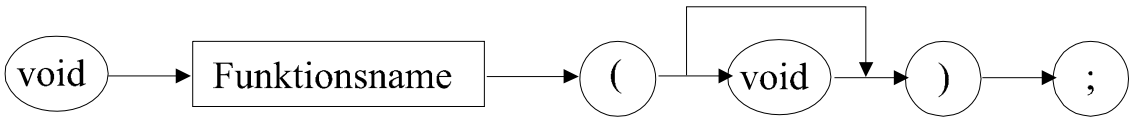
➔ es ist nicht möglich, die Anforderung einzuhalten, dass eine Funktion erst aufgerufen werden darf, wenn sie im Programm bekannt ist.

Abhilfe: Prototypen-Anweisung

4.3 Prototypenanweisung

Bei der Prototypenanweisung wird lediglich der Funktionskopf definiert.
Angegeben wird der Datentyp des Rückgabewertes und die Parameterliste

Syntax:



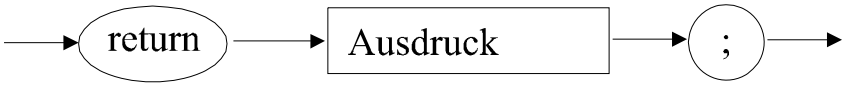
Beispiel:

```
void main()
{
   fkt_1();

   getchar();
}


void fkt_1(void)
{
   fkt_2();
}

void fkt_2(void)
{
   fkt_3();
}

void fkt_3(void)
{
}
```

```
void fkt_1(void);
void fkt_2(void);
void fkt_3(void);


void main()
{
   fkt_1();

   getchar();
}


void fkt_1(void)
{
   fkt_2();
}

void fkt_2(void)
{
   fkt_3();
}

void fkt_3(void)
{
}
```

- Die Rückgabe eines Wertes geschieht durch das **Schlüsselwort** `return`. Nach `return` kann ein beliebiger Ausdruck stehen, dessen Wert von der Funktion zurückgegeben werden soll. Das Ergebnis des Ausdrucks muss nur zum Rückgabetyp passen.
- Stimmen Rückgabetyp und Typ des `return`-Ausdrucks nicht überein, erfolgt eine **implizite Typumwandlung**.

Syntax:



es ist stets nur ein Rückgabewert erlaubt.

Beispiel:

```
int test(void)
{
   return(12);
}



void main()
{
   cout << "Return-Wert von test():"<<test()<<endl;
   getchar();
}
```

Ausgabe: Return-Wert von test():12

```
int test(void)
{
   return(12.7);
}



void main()
{
   cout << "Return-Wert von test():"<<test()<<endl;
   getchar();
}
```

Ausgabe: Return-Wert von test():12

```
int test(void)
{
   return('A');
}



void main()
{
   cout << "Return-Wert von test():"<<test()<<endl;
   getchar();
}
```

Ausgabe: Return-Wert von test():65

```
int test(void)
{
   char ergebnis;

   ergebnis = 'A';
   return(ergebnis);
}



void main()
{
   cout << "Return-Wert von test():"<<test()<<endl;
   getchar();
}
```

Ausgabe: Return-Wert von test():65

In einer Funktion sind mehrere return-Anweisungen erlaubt, es wird jedoch nur eine return-Anweisung ausgeführt, danach ist die Funktion beendet!

Beispiel:

```
bool test(int wert)
{
     if (wert > 100)
          return true;
     else
          return false;
}
```

4.4 Funktionen mit Parameterübergabe

Die Nützlichkeit von Funktionen kommt erst dann voll zum Trage, wenn man Parameter und Rückgabewerte einsetzt.

Ein Parameter kann ein Wert, ein Ausdruck oder eine Variable sein, der der Funktion übergeben wird. Die Funktion kann dann mit dem Wert weiter arbeiten, um ihre Aufgabe zu erledigen.

Eine Funktion bekommt den Parameter immer von einer aufrufenden Funktion, z.B. der main-Funktion, übergeben. Wenn eine Funktion einen Wert zurückgibt, wird dieser ebenfalls an die aufrufende Funktion zurückgegeben.

Die Namen der Parameter sind frei wählbar. Dadurch wird nur festgelegt, unter welchem Namen die Parameter in der Funktion angesprochen werden.
Die Namen müssen nicht mit den Namen der Variabeln übereinstimmen, die der Funktion übergeben werden. Sie dienen lediglich als Platzhalter.

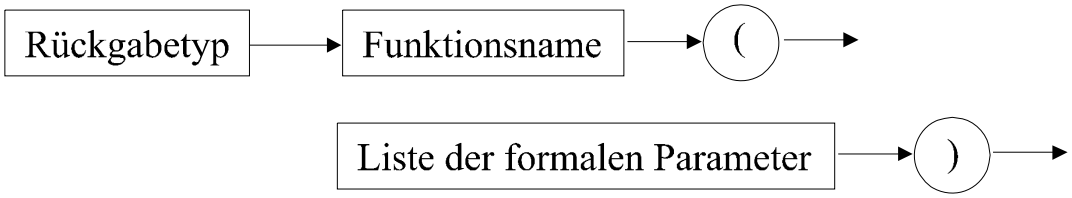
Werden einer Funktion mehrere Parameter übergeben spricht man von einer Parameterliste. Die einzelnen Werte in der Parameterliste werden als Funktionsargument oder Argument bezeichnet.

Beim Aufruf einer Funktion müssen die angegeben aktuellen Parameter in Reihenfolge, Anzahl und Datentyp den bei der Funktion definierten formalen Parametern entsprechen (Ausnahme: Anzahl der Parameter).

Die aktuellen Parameter werden eins zu eins den formalen Parametern zugeordnet; d.h. der 1. aktuelle Parameter wird dem 1. formalen Parameter zugeordnet, der 2. aktuelle Parameter dem 2. formalen Parameter, usw.

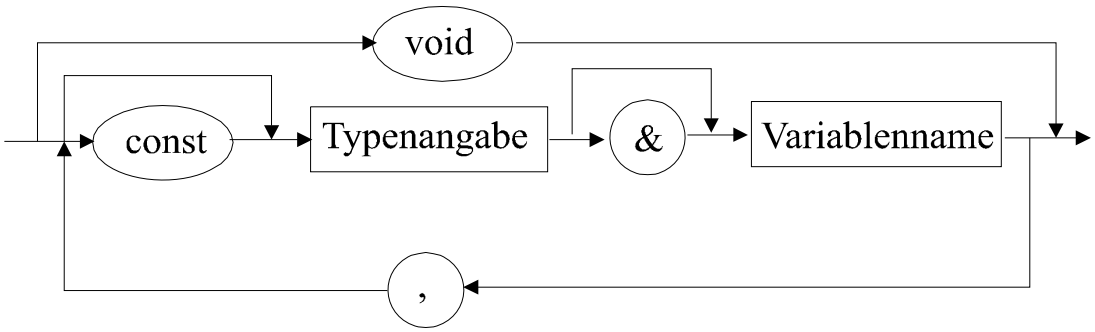
Syntax:

Funktionskopf

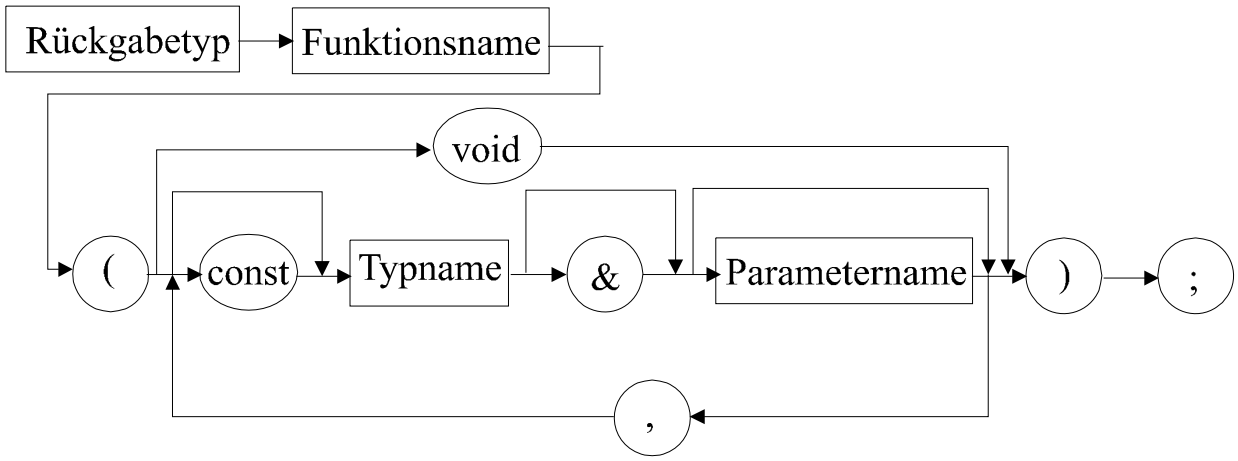


- der Rückgabetyp ist ein beliebiger Datentyp.
  Ausgenommen sind Felder und Funktionen.

Liste der formalen Parameter



4.5 Prototypanweisung:



Der Parametername kann bei der Prototypenanweisung weggelassen werden.

Wird er angegeben, kann er sich von dem Parametername im eigentlichen Funktionskopf unterscheiden.

Beispiel:

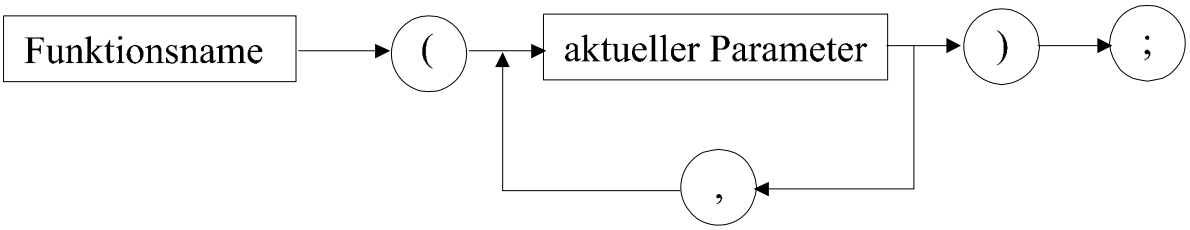
```
#include ...

int fkt_1(int, int var2);

main() ...

int fkt_1(int Anzahl, int Nummer)
{
.
.
.
}
```

Der Aufruf der Funktion erfolgt über den Namen und die Liste der aktuellen Parameter.

Aktuelle Parameter können Variablen, Konstanten oder Ausdrücke sein.

Die formalen Parameter, die bei der Funktion definiert sind werden durch die, beim Aufruf angegebenen Parameter ersetzt (in der angegebenen Reihenfolge).

Syntax:



4.6 Funktionsparameter

In C++ gibt es zwei Möglichkeiten der Parameterübergabe:

1. Wertübergabe (call by value)
2. Referenzübergabe (call by reference)

Bei der Wertübergabe wird der Wert des Parameters an die Funktion übergeben. In der Funktion wird eine Kopie dieses Parameters angelegt, mit dieser Kopie wird gerechnet.

Änderungen an der Kopie des Parameters ändert nichts an dem Parameter der **aufrufenden** Funktion.

Ist die Funktion beendet, verliert die Kopie des Parameters seine Gültigkeit, sie wird ‚zerstört'.

Bei der Referenzübergabe wird die Referenz (die Adresse) des Parameters der aufrufenden Funktion an die aufgerufene Funktion übergeben.

In der aufgerufenen Funktion wird keine Kopie des übergebenen Parameters angelegt.

Die Funktion rechnet und verändert den Parameter der aufrufenden Funktion.

Wird der Parameter in der aufgerufenen Funktion verändert, wird auch der Parameter in der aufrufenden Funktion verändert!

Beispiel:

```
#include <stdio.h>
#include <iomanip.h>

void manip(int & zahl1, int & zahl2)
{
 int hilf;

 zahl1 = 4;
 zahl2 = 9;

 hilf = zahl1 + zahl2;
 cout << "Ergebnis: " << hilf << endl;
}


void main()
{
  int wert1, wert2;


  wert1 = 3;
  wert2 = 7;

  cout << "vor  Aufruf: " << wert1 << ",  " <<
     wert2 << endl;
  manip(wert1, wert2);

  cout << "nach Aufruf: " << wert1 << ",  " <<
     wert2 << endl;
  getchar();
}
```

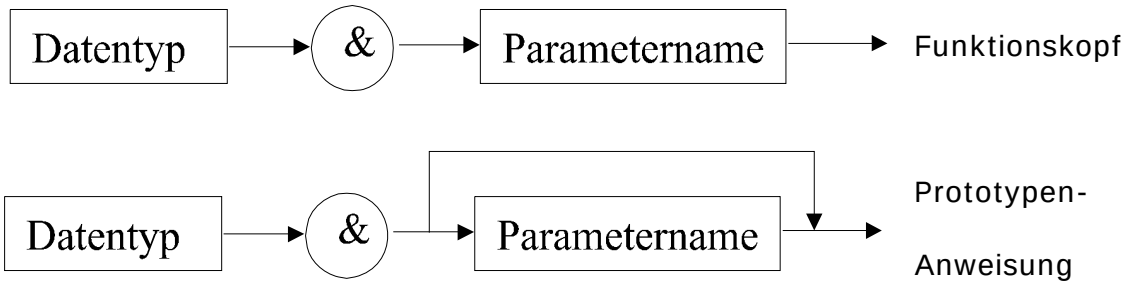
Ausgabe:

vor Aufruf: 3, 7

Ergebnis: 13

nach Aufruf: 4, 9

Syntax:



Da bei Referenzübergabe (call by reference) die Parameter der aufrufenden Funktion verändert werden, ist es möglich, scheinbar mehrere Rückgabewerte einer Funktion zu realisieren. da die aufrufende Funktion evtl. mit, durch die aufgerufene Funktion veränderten Parameterwerten weiter arbeitet.

Vorteile der Referenzübergabe:

- Dadurch, dass in der aufgerufenen Funktion keine Kopie des Parameters angelegt wird, benötigt call by reference weniger Speicherplatz als call by value
- die Parameterübergabe wird schneller abgearbeitet, da die Zeit für das Anlegen der Kopie und das Löschen der Kopie eingespart wird.

Nachteile der Referenzübergabe:

- es besteht die Gefahr, dass Parameter in der aufrufenden Funktion unbeabsichtigt verändert werden, da die aufgerufene Funktion Zugriff auf die ursprünglichen Parameter hat.
- Eine unbeabsichtigte Veränderung der Parameter kann durch Verwendung des Schlüsselworts `const` verhindert werden.
  Hierdurch wird der Parameter als Konstant (unveränderbar) vereinbart!

```
#include <stdio.h>
#include <iostream.h>

void test(int & a, int b)
{
   a = 20;
}



void main()
{
   int var1=10;
   int var2=20;

   cout << "VOR  Test-Aufruf: var1:" << var1 << "
     var2:" << var2 << endl;

   test(var1, var2);

   cout << "NACH Test-Aufruf: var1:" << var1 << "
     var2:" << var2 << endl;


   getchar();
}
```

➔

```
VOR  Test-Aufruf: var1:10   var2:20
NACH Test-Aufruf: var1:20   var2:20
```

```
#include <stdio.h>
#include <iostream.h>

void test(const int & a, int b)
{
   a = 20;
}



void main()
{
   int var1=10;
   int var2=20;

   cout << "VOR  Test-Aufruf: var1:" << var1 << "
     var2:" << var2 << endl;

   test(var1, var2);

   cout << "NACH Test-Aufruf: var1:" << var1 << "
     var2:" << var2 << endl;


   getchar();
}
```

> Führe aus: C:\PROGRAMME\CONTEXT\ConExec.exe "bcc32.exe" test

Borland C++ 5.2 for Win32 Copyright (c) 1993, 1997 Borland International

test.cpp:

Error test.cpp 6: Cannot modify a const object in function test(const int &,int)

***** 1 errors in Compile *****

> Führe aus

4.7 Eindimensionale Felder als Funktionsparameter

Beispiel:

```
void feld_fkt(int feldname[])
{
   feldname[0] = 12;
   feldname[1] = 13;
   feldname[2] = 14;
   feldname[3] = 15;
}


void main()
{
   int feld1[] = {1, 2, 3, 4, 5};

   cout << "vor Aufruf:  " << feld1[0] << "  "
     << feld1[1] << "  "
                           << feld1[2] << "  "
     << feld1[3] << "  "
     << feld1[4] << "  " << endl;
   feld_fkt(feld1);

   cout << "nach Aufruf: " << feld1[0] << "  "
     << feld1[1] << "  "
                           << feld1[2] << "  "
     << feld1[3] << "  "
     << feld1[4] << "  " << endl;
   getchar();
}
```

➔

```
vor  Aufruf:1   2       3   4   5
nach Aufruf:12   13   14   15   5
```

```
void feld_fkt(const int feldname[])
{
   feldname[0] = 12;
   feldname[1] = 13;
   feldname[2] = 14;
   feldname[3] = 15;
}


void main()
{
   int feld1[] = {1, 2, 3, 4, 5};

   cout << "vor Aufruf:  " << feld1[0] << "  "
     << feld1[1] << "  "
                          << feld1[2] << "  "
     << feld1[3] << "  "
     << feld1[4] << "  " << endl;
   feld_fkt(feld1);

   cout << "nach Aufruf: " << feld1[0] << "  "
     << feld1[1] << "  "
                          << feld1[2] << "  "
     << feld1[3] << "  "
     << feld1[4] << "  " << endl;
   getchar();
}
```

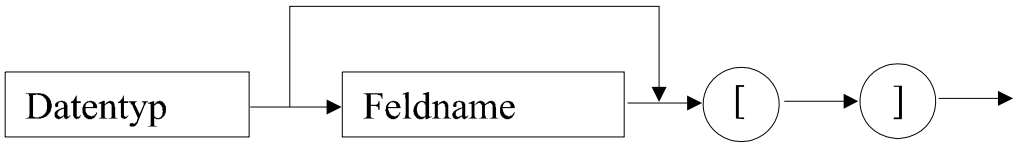
```
> Führe aus: C:\PROGRAMME\CONTEXT\ConExec.exe "bcc32.exe" test

Borland C++ 5.2 for Win32 Copyright (c) 1993, 1997 Borland
International
test.cpp:
Error test.cpp 6: Cannot modify a const object in function
test(const int * const)
Error test.cpp 7: Cannot modify a const object in function
test(const int * const)
Error test.cpp 8: Cannot modify a const object in function
test(const int * const)
Error test.cpp 9: Cannot modify a const object in function
test(const int * const)

*** 4 errors in Compile ***
> Führe aus
```

➔ Werden Felder als Funktionsargumente eingesetzt, wird **immer** die Adresse des Feldes an die Funktion übergeben ➔ **immer** call by reference

Felder können nicht als Rückgabeparameter verwendet werden!

Syntax:

Prototypenanweisung:



formaler Parameter in der Parameterliste:

Mehrdimensionale Felder als Funktionsparameter

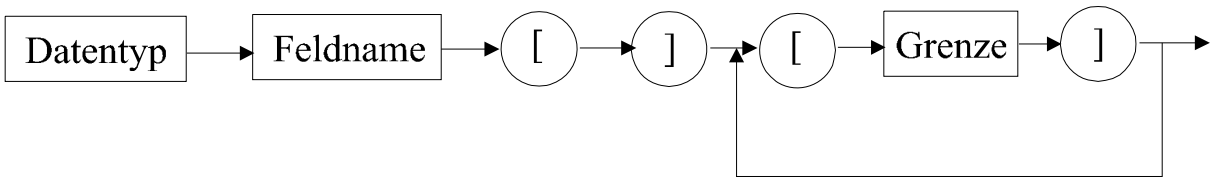
Grundsätzlich entspricht ein mehrdimensionales Feld als Funktionsparameter einem eindimensionalen Feld als Funktionsparameter ➔ call by reference

Ausnahme:

Ab der zweiten Dimension müssen die Indexgrenzen angegeben werden (**im Funktionskopf und in der Prototypenanweisung**).

Die Indexgrenze (ab) der zweiten Dimension muss der Indexgrenze in der aufrufenden Funktion entsprechen!

Syntax:



Beispiel:

```
void feld_fkt(int [][5]);

void main()
{
   int feld1[][5] = {1, 2, 3, 4, 5,
                     6, 7, 8, 9, 10};

   cout << "vor Aufruf:  " << feld1[0][0] << "  "
<< feld1[0][1] << "  " << endl;

   feld_fkt(feld1);

   cout << "nach Aufruf: " << feld1[0][0] << "  "
<< feld1[0][1] << "  " << endl;
   getchar();
}


void feld_fkt(int feldname[][5])
{
   feldname[0][1] = 12;
   feldname[0][2] = 13;
   feldname[0][3] = 14;
   feldname[0][4] = 15;
}
```

4.8 Standardwerte für Parameter und variable Parameteranzahl

Beim Aufruf einer Funktion müssen die angegeben aktuellen Parameter in Reihenfolge, Anzahl und Datentyp den bei der Funktion definierten formalen Parametern entsprechen (**Ausnahme: Anzahl der Parameter**). (s.o.)

In C++ ist es erlaubt, **die letzten** Parameter beim Aufruf wegzulassen!

Voraussetzung:

Bei der Angabe der formalen Parameter werden **in der Prototypenanweisung** Standardwerte angegeben.

(Bei Borland C++ ist die Zuweisung von Standardwerten auch im Funktionskopf möglich, wenn er vor dem Aufruf der Funktion definiert ist!)

Beim Aufruf der Funktion werden die fehlenden aktuellen Parameter durch die Standardwerte ersetzt.

```
void fkt(int, int = 10);

void main()
{
   fkt(29);
   getchar();
}


void fkt(int par_1, int par_2)
{
   cout << par_1 << " " << par_2 << endl;
}
```

Ausgabe: 29 10

```
void fkt_2(int=10, char = ' ', int = 20);

void fkt_2(int par_1, char par_2, int par_3)
{
   cout << par_1 << endl << par_2 << endl << par_3 << endl;
}
```

Erlaubte Aufrufe:

```
fkt_2();
fkt_2(12);
fkt_2(12, 'A', 20);
```

nicht erlaubte Aufrufe:

```
fkt_2(, 'A');
fkt_2(, , 20);
```

4.9 Überladen von Funktionen (Polymorphie)

Bei der objektorientierten Programmierung (OOP) ist es erlaubt, dass die Namen von verschiedenen Funktionen gleich sind, wenn sie sich in der Anzahl bzw. Typ der Übergabeparameter unterscheiden. Der Datentyp des Rückgabeparameters ist nicht relevant.

- es können verschiedene Funktionen mit **gleichem Namen** programmiert werden
- die Funktionen müssen sich in Anzahl und/oder Datentyp der formalen Parameter unterscheiden
- der Rückgabeparameter ist für die Unterscheidung der Funktionen nicht von Bedeutung

Beispiel:

```
void fkt_1(int, int);
void fkt_1(int, char);
void fkt_1(char, char);
```

nicht zulässig:

```
void fkt_1(int, int);
int fkt_1(int, int);
char fkt_1(int, int);
```

Beim Aufruf der Funktion kann der Compiler anhand der verschiedenen Parameter unterscheiden, welche Funktion gemeint ist.

4.10 Probleme bei Polymorphie mit Standardwerten

Beispiel:

```
void fkt_1(int, int, char = 'A');
void fkt_1(int, int);
```

Aufruf:

```
main()
{
.
.
fkt_1(20, 12);
}
```

Beim Aufruf erkennt der Compiler, dass die Funktion mit zwei Integer-Parameter aufgerufen werden soll → bei der ersten Version der Funktion kommt der Standardwert nicht zum tragen → überflüssig, die erste Funktion muss immer mit 3 Parameter aufgerufen werden!

Definition:

Polymorphisierung ist die Fähigkeit von Programmelementen, sich zur Laufzeit auf Objekte verschiedener Klassen beziehen zu können.

Damit wird erst zur Laufzeit die zum Objekt passende Realisierung der Operation ermittelt.

4.11 Zeiger als Funktionsparameter

Auch der Datentyp „Zeiger“ kann als Parameter an eine Funktion übergeben werden oder als Rückgabewert einer Funktion dienen.

4.11.1 Zeiger als Übergabeparameter

Der übergebene Zeiger wird immer **call by value** übergeben → es ist nicht möglich, die Adresse des Zeigers zu ändern.

Es ist natürlich möglich, den Wert, auf den der Zeiger zeigt, zu ändern!

Beispiel:

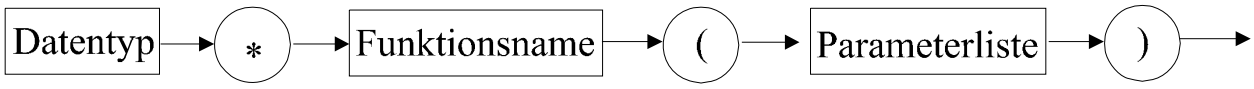
```
void fkt_2(int * wert_1, int * wert_2)
{
   *wert_1 = 12;
   *wert_2 = 13;
}

void main()
{
   int var_1;
   int var_2;

   var_1 = 100;
   var_2 = 200;

   cout << var_1 << endl << var_2 << endl;

   fkt_2(&var_1, &var_2);

   cout << var_1 << endl << var_2 << endl;
   getchar();
}
```

Ausgabe:

```
100
200
12
13
```

4.11.2 Zeiger als Rückgabeparameter

Syntax:



Beispiel:

```
int * fkt_2(int wert)
{
   int var_1 = wert;

   return &var_1;
}

void main()
{
   int *var_1;
   int *var_2;

   var_1 = fkt_2(100);
   var_2 = fkt_2(200);
   cout << *var_1 << endl << *var_2 << endl;
   getchar();
}
```

Negativbeispiel:

```
#include <iostream.h>
#include <stdio.h>

int *funktion(int Anfangswert)
{
   int Endwert = Anfangswert;
   return(&Endwert);
}

int main()
{
   int *Wert = funktion(100);;

   cout << *Wert << endl;
   cout << *Wert << endl;
   getchar();
}
```

Ausgabe:

```
100
1245048
```

Problem: Bei jedem Aufruf einer Funktion wird eine neue Kopie ihrer Argumente und lokalen Variablen angelegt. Der Speicher wird nach dem Ende der Funktion anderweitig benutzt.
Daher sollte nie ein Zeiger auf eine lokale Variable zurückgeliefert werden – der Inhalt der Stelle, auf die gezeigt wird, wird sich unvorhersagbar ändern!

Abhilfe:

```
#include <iostream.h>
#include <stdio.h>

int *funktion(int Anfangswert)
{
   static int Endwert = Anfangswert;
   return(&Endwert);
}

int main()
{
   int *Wert = funktion(100);;

   cout << *Wert << endl;
   cout << *Wert << endl;
   getchar();
}
```

Ausgabe:

```
100
100
```

4.11.3 Statische Deklaration (static)

Normalerweise wird eine lokale Variable einer Funktion genau dann initialisiert, wenn die Funktion aufgerufen wird.

Ist die Funktion beendet, verliert die lokale Variable ihre Gültigkeit, der reservierte Speicherbereich wird wieder freigegeben.

Eine statische (static) deklarierte Variable wird zwar auch dann initialisiert, wenn die entsprechende Funktion zum ersten mal aufgerufen wird, im Gegensatz zu einer nicht statisch deklarierten Variablen wird der reservierte Speicherbereich jedoch nicht freigegeben.

Die Variable verliert zwar ihre Gültigkeit (ist durch den Variablennamen nicht mehr zugreifbar), der Speicherbereich, und damit der Wert bleiben aber weiterhin geschützt.

Beispiel:

```
#include <iostream.h>
#include <stdio.h>

void staticVarTest()
{
   static int Nummer = 1;

   cout << Nummer << ". Aufruf der Funktion" << endl;
   Nummer++;
}


int main()
{
   for (int i=0;i < 5;i++)
      staticVarTest();

   getchar();
}
```

Ausgabe:

```
1. Aufruf der Funktion
2. Aufruf der Funktion
3. Aufruf der Funktion
4. Aufruf der Funktion
5. Aufruf der Funktion
```

4.12 Rekursion

Problem:

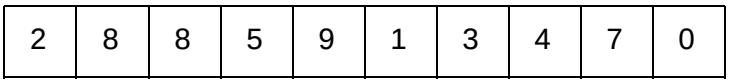
Es soll eine Funktion geschrieben werden, die ein **Feld** von Integer-Werten, das der Funktion als **Parameter** übergeben wird, sortiert werden.

Lösungsansatz:

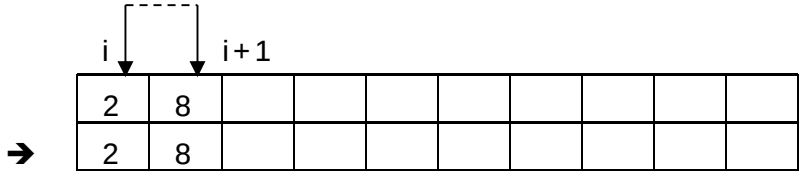
Das Feld wird ‚durchlaufen' und dabei wird überprüft, ob das aktuelle Element größer ist als das nächste Element. Ist dies der Fall, werden die beiden Elemente ausgetauscht.

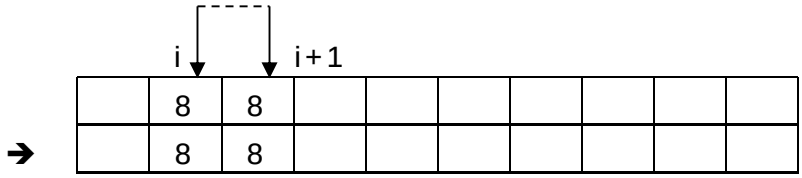
Dieser Vorgang (‚Durchlaufen') wird so oft wiederholt, bis keine Elemente mehr ausgetauscht werden müssen → das Feld ist sortiert!

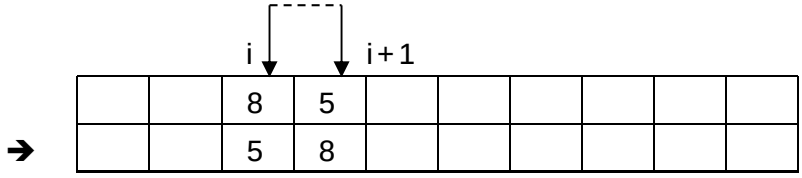
Initialisierungswerte:

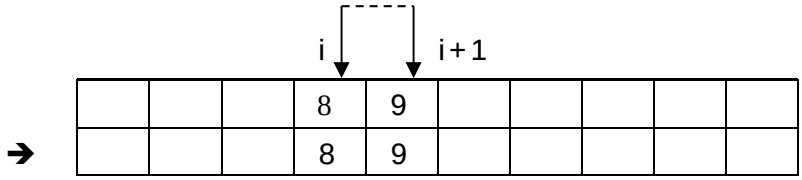


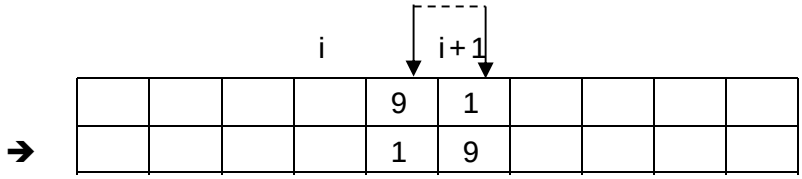
1. Durchlauf:

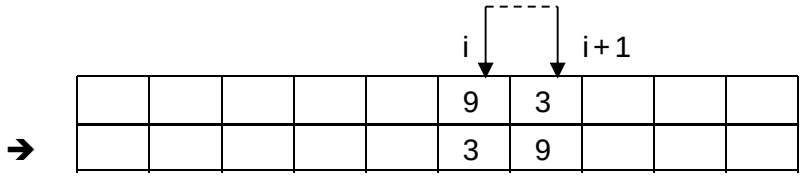


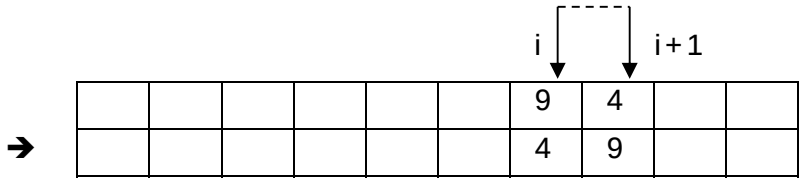


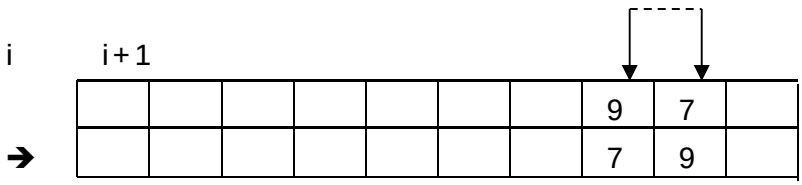


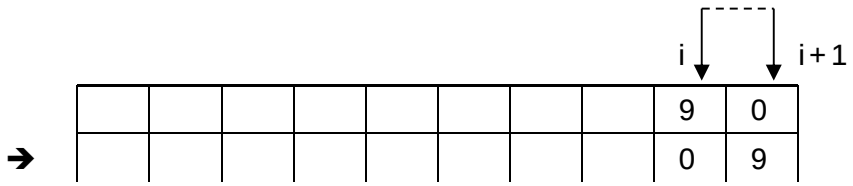


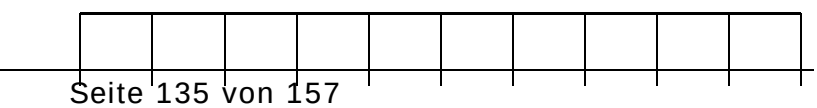












Ergebnis 1. Durchlauf:	2	8	5	8	1	3	4	7	0	9
2. Durchlauf:	2	5	8	1	3	4	7	0	8	9
3. Durchlauf:	2	5	1	3	4	7	0	8	8	9
4. Durchlauf:	2	1	3	4	5	0	7	8	8	9
5. Durchlauf:	1	2	3	4	0	5	7	8	8	9
6. Durchlauf:	1	2	3	0	4	5	7	8	8	9
7. Durchlauf:	1	2	0	3	4	5	7	8	8	9
8. Durchlauf:	1	0	2	3	4	5	7	8	8	9
9. Durchlauf:	0	1	2	3	4	5	7	8	8	9

1. Möglichkeit

```
#include <stdio.h>
#include <iomanip.h>

bool sort(int[], int);
void Ausgabe(int[], int);

void main()
{
  bool fertig;
  // Feld definieren und beliebige Ganzzahlen zuweisen
  int zahlen[] = {2, 8, 8, 5, 9, 1, 3, 4, 7, 0};

  // Kontrollausgabe der Zahlen bei Beginn des Programms
  cout << "\nAnfang!\n";
  Ausgabe(zahlen, sizeof(zahlen)/sizeof(int));
  cout << endl;

  do
  {
      // Sortierfunktion mit Feld und Anzahl Elemente des Feldes aufrufen
      fertig = sort(zahlen, sizeof(zahlen)/sizeof(int));
   }  while(fertig == false);

  // Sortiervorgang abgeschlossen -> Kontrollausgabe der Zahlen
  cout << "\nfertig!\n";
  Ausgabe(zahlen, sizeof(zahlen)/sizeof(int));

  getchar();
}

bool sort(int zahlen[], int anz_Elemente)
{
  int hilf;
  bool fertig = true;

  // für alle Elemente testen, ob das nächste Element größer als das aktuelle ist
  for (int i=0;i < anz_Elemente-1;i++)
  {
    if (zahlen[i] > zahlen[i+1])
    {
      // wenn ja, Elemente tauschen
      hilf = zahlen[i];
      zahlen[i] = zahlen[i+1];
      zahlen[i+1] = hilf;
      // Kontrollausgabe des Feldes
      Ausgabe(zahlen, anz_Elemente);
      fertig = false;
   }
  }
  return fertig;
}

void Ausgabe(int zahlen[], int anz_Elemente)
{
  cout << "Zahlen: ";
  for (int i=0;i < anz_Elemente; i++)
    cout << setw(4) << zahlen[i];
  cout << endl;
}
```

2. Möglichkeit

```
#include <stdio.h>
#include <iomanip.h>

void sort(int[], int);
void Ausgabe(int[], int);

void main()
{
 // Feld definieren und beliebige Ganzzahlen zuweisen
  int zahlen[] = {2, 8, 8, 5, 9, 1, 3, 4, 7, 0};

 // Kontrollausgabe der Zahlen bei Beginn des Programms
  cout << "\nAnfang!\n";
  Ausgabe(zahlen, sizeof(zahlen)/sizeof(int));
  cout << endl;

 // Sortierfunktion mit Feld und Anzahl Elemente des Feldes aufrufen
  sort(zahlen, sizeof(zahlen)/sizeof(int));

 // Sortiervorgang abgeschlossen -> Kontrollausgabe der Zahlen
  cout << "\nfertig!\n";
  Ausgabe(zahlen, sizeof(zahlen)/sizeof(int));

  getchar();
}


void sort(int zahlen[], int anz_Elemente)
{
  int hilf;

 // für alle Elemente testen, ob das nächste Element größer als das aktuelle ist
  for (int i=0;i < anz_Elemente-1;i++)
  {
    if (zahlen[i] > zahlen[i+1])
    {
      // wenn ja, Elemente tauschen
      hilf = zahlen[i];
      zahlen[i] = zahlen[i+1];
      zahlen[i+1] = hilf;
      // Kontrollausgabe des Feldes
      Ausgabe(zahlen, anz_Elemente);
      // der Sortiervorgang noch nicht fertig ist
      // ... (es mussten noch Elemente getauscht werden,
      // ... → rekursiver Aufruf der Funktion
      sort(zahlen, anz_Elemente);
   }
  }
}

void Ausgabe(int zahlen[], int anz_Elemente) {
  cout << "Zahlen: ";
  for (int i=0;i < anz_Elemente; i++)
    cout << setw(4) << zahlen[i];
  cout << endl;
}
```

4.13 Dynamische Speicherverwaltung

Es gibt mehrere Möglichkeiten, den vom Programm belegten Arbeitsspeicher dynamisch zu verwalten. Hier werden nur einige wenige Methoden vorgestellt.

4.13.1 malloc

Memory allocation, Anforderung und Zuweisung von Arbeitsspeicherplatz

Syntax:

```
#include <stdlib.h> or #include<alloc.h>
void *malloc(size_t size);
```

Durch den Aufruf der Funktion malloc wird während der Programmlaufzeit (dynamisch) Speicherplatz reserviert und zwar in der als Parameter angegebenen Größe (in Bytes).

Rückgabewert

malloc liefert einen Zeiger auf den reservierten Speicherbereich. Wenn nicht genügend Speicherplatz zur Verfügung steht, ist der Rückgabewert NULL. Wenn der Parameter size den Wert 0 hat, liefert malloc NULL zurück.

Beispiel:

```
#include <iostream.h>
#include <stdio.h>
#include <stdlib.h>
#include <iomanip.h>

void *malloc(unsigned size);

int main()
{
  char *ptr;

  ptr = (char *) malloc(1000);   // reservieren von 1000 Bytes im
                                   // Arbeitsspeicher und Zuweisung
  if (ptr != NULL)                 // der Adresse an den Pointer ptr
   cout << "Adresse des reservierten Bereichs:" << (int)ptr << endl;
  getchar();
}
```

Natürlich muss der angeforderte Speicherplatz auch wieder freigegeben werden:

4.13.2 free

free gibt einen zuvor mit malloc reservierten Block wieder frei.

Syntax:

```
#include <stdlib.h>
void free(void *block);
```

Beispiel:

```
#include <iostream.h>
#include <stdio.h>
#include <stdlib.h>
#include <iomanip.h>

void *malloc(unsigned size);

int main()
{
  char *ptr;

  ptr = (char *) malloc(1000);   // reservieren von 1000 Bytes im
                                   // Arbeitsspeicher und Zuweisung
  if (ptr != NULL)                 // der Adresse an den Pointer ptr
   cout << "Adresse des reservierten Bereichs:" << (int)ptr << endl;
  getchar();

  free(ptr);
}
```

4.13.3 memcpy

Kopieren eines beliebigen Speicherbereichs: **memcpy** kopiert einen Speicherbereich von n Bytes. Der erste Parameter beinhaltet die Zieladresse, der zweite Parameter zeigt auf die Quelladresse (source). Der dritte Parameter gibt die Anzahl der zu kopierender Bytes dar.

Überlappen sich scr und dest, ist das Verhalten von memcpy undefiniert.

Syntax:

```
#include <mem.h>
void *memcpy(void *dest, const void *src, unsigned n);
```

Rückgabewert

memcpy liefert dest zurück.

Beispiel:

```
#include <iostream.h>
#include <stdio.h>

void *malloc(unsigned size);

int main()
{
   char src[20] = "Hello World";
   char dest[20];

   cout << src << endl;
   cout << dest << endl;

   memcpy(dest, src, 20);
   cout << "\nnach Kopieren\n\n";

   cout << src << endl;
   cout << dest << endl;

   getchar();
}
```

Ausgabe:

```
Hello World
ç§@

nach Kopieren

Hello World
Hello World
```

4.14 Strukturen

Eine Struktur ist wie ein array, eine Zusammenfassung von mehreren Variablen. Im Gegensatz zum array können die zusammengefassten Variablen von verschiedenen Datentypen sein.

Beispiel:

```
struct Personendaten
{
   char Vorname[50];
   char Nachname[50];
   ...
   int Alter;
};
```

Durch diese Definition wird ein neuer Datentyp geschaffen, der (mit Einschränkungen) wie ein Standarddatentyp verwendet werden kann.

Man beachte den Strichpunkt am Ende der Strukturdefinition; dies ist eine der wenigen Ausnahmen, bei denen nach einer schließenden geschweiften Klammer ein Strichpunkt angegeben werden muss.

Variablendeklaration:

```
Personendaten einer;
Personendaten AIK_9[61];

void fkt(Personendaten Person)
```

Beispiel:

```
#include <iostream.h>
#include <conio.h>

struct Personendaten
{
   char Vorname[50];
   char Nachname[50];
   char Strasse[50];
   char plz[6];
   char Ort[50];
   int Alter;
};

void main()
{
   Personendaten Person;
   cout << sizeof(Person) << endl;
   getch();
}
```

5 Anhang

5.1 Literatur

Titel:	**Jetzt lerne ich C++**
Erschienen im:	Markt+Technik-Verlag
ISBN:	3-8272-5663-1
Preis:	ca. 25 €
Titel:	**C++ Lernen und professionell anwenden**
Autor:	Kirch-Prinz / Prinz
Erschienen im:	milp - Verlag
ISBN:	3-8266-0423-7
Preis:	ca. 45 €
Titel:	**Die C++ Programmiersprache**
Autor:	Bjarne Stroutstrup
Erschienen im:	Addison-Wesley - Verlag
ISBN:	3-8273-1660-X
Preis:	ca. 50 €

5.2 Aufgabensammlung

Aufgabe 1: In folgender Konstellation hat das 1. Element des Feldes x die Adresse 64fdf0. An welcher Adresse ist das Element x[2] abgespeichert?

```
#include <conio.h>
#include <stdio.h>
#include <iomanip.h>

void main()
{
   long x[]= {1, 2, 3, 4, 5};
   long y[5] ;
}
```

Lösung: 64fdf8

Aufgabe 2: Wie lautet die Ausgabe von folgendem Programm?

```
#include <conio.h>
#include <stdio.h>
#include <iomanip.h>

void main()
{
   char y[] = "";
   cout << setiosflags(ios::hex);
   strcat(y, "Hello World");
   cout << strlen(y) << endl;
}
```

Lösung: b

Aufgabe 3: Markieren und erläutern Sie die Fehler in folgendem Programm

```
#include <conio.h>
#include <stdio.h>
#include <iomanip.h>

void main()
{
   int x[5]= {1, 2, 3, 4, 5};
   for(int i=1; i <= 5; i++)
   {
      cout << "Bitte Wert " << i << " eingeben!";
      cin >> x[i];
   }
}
```

Lösung: Bereichsüberschreitung im Feld x

Aufgabe 4: Definieren Sie einen String Zeichen und schreiben Sie die Anweisung zum Einlesen eines char-Feldes mit maximal 80 lesbaren Zeichen in diesen String.

Die Eingabe soll durch das Zeichen 'a' begrenzt werden.

Lösung:

```
char Zeichen[81];
cin.getline(Zeichen, 81, ‚a');
```

Aufgabe 5: Wie lautet die Ausgabe von folgendem Programm?

```
#include <conio.h>
#include <stdio.h>
#include <iomanip.h>

void main()
{
   char Woche[][10] = {"Montag", "Dienstag", "Mittwoch",
     "Donnerstag", "Freitag", "Samstag", "Sonntag"};

   cout << Woche[4] << endl;
   cout << Woche[3][2] << endl;
}
```

Lösung:

```
Freitag
n
```

Aufgabe 6: Warum ist die folgende Anweisung syntaktisch fehlerhaft?

```
if (i<5) cout<<"i= "<<i<<endl;
         j=2*i;
else j=i+5;
```

Lösung: Im If-Fall muss ein Block gebildet werden, da zwei Anweisungen ausgeführt werden sollen!

Aufgabe 7: Vereinbaren Sie folgende Felder:

```
FSTR  : Feld von 5 String-Elementen mit jeweils 10 Zeichen Länge,
FDOB  : zweidimensionales Feld von double-Elementen (5 Zeilen, 7 Spalten),
F3INT : dreidimensionales Feld von int-Elementen mit:
        4 Elementen in der 1.Dimension, 7 Elementen in der 2.Dimension und
        3 Elementen in der 3.Dimension
```

Lösung:
```
char FSTR[5][10];

double FDOB[5][7];

int F3INT [4][7][3];
```

Aufgabe 8: Wie groß muss die Länge eines **char** - Feldes mindestens gewählt werden, um den Text **Klausur** abspeichern zu können?

Lösung: 8 Elemente

Aufgabe 9:

a) Programmieren Sie folgendes Programmstück (kein vollständiges Programm!):

Es sollen 2 Zeichenkettenvariablen **str1** und **str2** verglichen werden. Sind sie identisch, soll der Text:

str1 und str2 identisch!

im anderen Fall der Text:

str1 und str2 stimmen nicht überein!

ausgegeben werden!
(3P)

b) Welche Prototypdatei wird für die Vergleichsfunktion benötigt?
(1P)

Lösung:

```
a) if (strcmp(str1, str2) == 0)
     cout << “str1 und str2 identisch!” << endl;
     else
     cout << “str1 und str2 stimmen nicht überein!” << endl;


b)  string.h
```

Aufgabe 10:

Schreiben Sie ein Programmstück (kein vollständiges Programm!!) , das für 20 Elemente eines eindimensionalen Feldes **a** die Summe dieser Elemente berechnet und den Ergebniswert ausgibt.

Lösung:

```
int summe = 0;

for (int i=0;i < 20;i++)
      summe += a[i];

cout << “Die Summe ist: << i << endl;
```

Aufgabe: Vektoraddition / -subtraktion

Schreiben Sie ein C++ - Programm, das zwei eindimensionale Felder (Vektoren) **a** und **b** addiert bzw. subtrahiert!

Es gilt für die Vektoraddition:

$$c_i = a_i + b_i \quad ; \quad i = 1, \ldots, n ,$$

wobei n die aktuelle Anzahl von Elementen der Vektoren A und B ist.

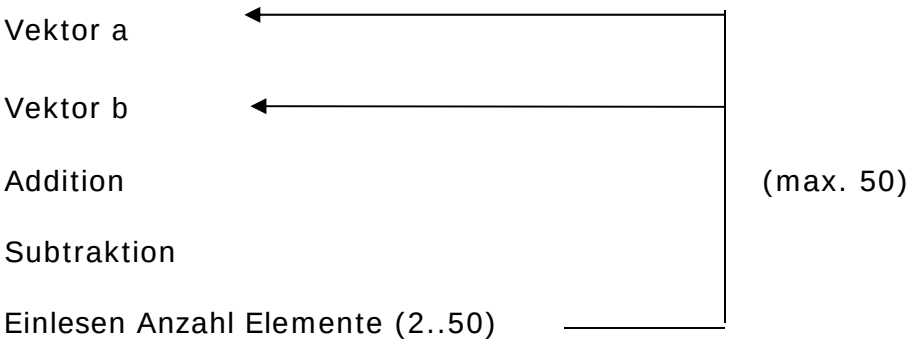
Für die Vektorsubtraktion gilt analog:

$$d_i = a_i - b_i \quad ; \quad i = 1, \ldots, n$$

Die Eingabe soll so gestaltet werden, dass zunächst die aktuelle Größe **n** der Vektoren eingegeben wird. Dabei ist die Eingabe so zu programmieren, dass abgesichert wird, dass **n** größer als 1 und kleiner gleich 50 ist! Bei fehlerhafter Eingabe ist eine Nachricht auszugeben und die Eingabe so lange zu wiederholen, bis ein korrekter Wert eingegeben wird!

Nach der Eingabe von **n** soll abwechselnd ein **a-** und ein **b-** Element eingegeben werden. Danach folgen die Berechnung der Ergebnisvektoren **c** und **d** sowie die Ausgabe dieser Vektoren!

Sowohl bei der Eingabe als auch bei der Ausgabe soll stets erkennbar sein, welches Feldelement ein- bzw. ausgegeben wird.



Bis Eingabe richtig → Fehlermeldung

Einlesen der Elemente a, Einlesen der Elemente b

Durchführen der Addition und der Subtraktion
→ Ergebnisvektor `Addition`
Ergebnisvektor `Subtraktion`

Ausgabe des Ergebnisses

Aktionen, die durchgeführt werden müssen:

- Wiederholbar?
- Definition aller Vektoren
- Definition Variablen für Anzahl Elemente
- Einlesen Anzahl Elemente, solange eingegebener Wert ungültig ist
- (Wenn Eingabe fehlerhaft → Fehlermeldung)
- Einlesen der Werte für Vektor a und b (Anzahl Elemente)
- Durchführen der Addition und Subtraktion (Anzahl Elemente)
- Ausgabe der berechneten Werte (Anzahl Elemente)

- Ausgabe der Programmfunktion
- Aufforderung zur Eingabe der Vektorgröße
- Eingabe der Vektorgröße
- Solange Vektorgröße <= 1 oder > 50
- i = 0; Solange i < Anzahl Elemente
  - Aufforderung zur Eingabe von Vektor a Element i
  - Einlesen Vektor a Element i (a[i])
  - Aufforderung zur Eingabe von Vektor b Element i
  - Einlesen Vektor b Element i (b[i])
  - i um 1 erhöhen
- i = 0; Solange i < Anzahl Elemente
  - Addition[i] = a[i] + b[i]
  - Subtraktion[i] = a[i] - b[i]
  - i um 1 erhöhen
- Bildschirm löschen
- Ausgabe Hinweis Ergebnis
- Ausgabe "Ergebnis der Addition:"
- i = 0; Solange i < Anzahl Elemente
  - Positionierung auf Bildschirm
  - Ausgabe Addition [i]
  - i um 1 erhöhen
- i = 0; Solange i < Anzahl Elemente
  - Positionierung auf Bildschirm
  - Ausgabe Subtraktion [i]
  - i um 1 erhöhen

```
#include <conio.h>
#include <stdio.h>
#include <iomanip.h>

void main()
{
   int a[50];
   int b[50];
   int Addition[50];
   int Subtraktion[50];
   int n;
   int Zeile;           // fuer spaetere Ausgabe

   cout << "Programm addiert und subtrahiert zwei Vektoren (2..50
Elemente)." << endl <<
"Die Ergebnisse werden in zwei weiteren Vektoren gespeichert und auf dem"
<< endl << "Bildschirm ausgegeben!" << endl;

   do
   {
      cout << "Bitte Anzahl der Felder für die Vektoren eingeben!";
      cin >> n;
      if (n <= 1 || n > 50)
         cout << "Anzahl Felder muss zwischen 2 und 50 liegen" << endl <<
endl;
   }  while (n < 1 || n > 50);

   for (int i=0; i < n; i++)
   {
      cout << "Bitte Wert " << i << " für Vektor a eingeben: ";
      cin >> a[i];
      cout << "Bitte Wert " << i << " für Vektor b eingeben: ";
      cin >> b[i];
   }

   for (int i=0; i < n; i++)
   {
      Addition[i] = a[i] + b[i];
      Subtraktion[i] = a[i] - b[i];
   }

   cout << "Die Ergebnisse der Addition bzw. Subtraktion lauten:" << endl;
   clrscr();
   Zeile = 3;
   gotoxy (0, Zeile);
   cout << "Addition:";
   Zeile++;
   for (int i=0; i < n;)
   {
      int j;
      for (j=0; j < 5 && j+i < n; j++)
      {
         gotoxy (j*13, Zeile);
         cout << "[" << j+i << "]: " << Addition[j+i];
      }
      i += j;
      Zeile += 1;
   }

   Zeile += 3;

   cout << "\n\nSubtraktion:\n";
```

```
   for (int i=0; i < n;)
   {
      int j;
      for (j=0; j < 5 && j+i < n; j++)
      {
         gotoxy (j*13, Zeile);
         cout << "[" << j+i << "]: " << Subtraktion[j+i];
      }
      i += j;
      Zeile += 1;
   }

   getchar();
}
```

Aufgabe:

Geben Sie einen Text von max. 127 Zeichen über die Tastatur ein und in umgekehrter Reihenfolge wieder aus!

```
void main()
{
   char text[128];

   cin.getline(text, 127);

cout << "Länge des Textes: " << strlen(text) << endl << endl;

   for(int i = strlen(text)-1; i >= 0; i--)
      cout << text[i];

   getchar();
}
```

Aufgabe:

Geben Sie einen Text von max. 80 Zeichen über die Tastatur ein. Verschlüsseln Sie ihn auf einfache Weise, in dem Sie einen zufällig gewählten Kleinbuchstaben (97 < ASCII-Code < 122) jeweils zwischen die Zeichen einschieben.

Beispiel: Dies ist ein Testtext!
zufällig gewähltes Zeichen: **a**
Daiaeasa aiasata aeaiana aTaeasatataeaxata!

Lösungsvorschlag:

```
#include <stdio.h>
#include <conio.h>
#include <iostream.h>
#include <iomanip.h>
#include <stdlib.h>
#include <time.h>

void main()
{
   char text[81];
   char verschluesselt[161];
   char zeichen;
   int j = 0;

   cin.getline(text, 80);

   cout << "Länge des Textes: " << strlen(text) << endl << endl;

   randomize();
   zeichen = (char) random(122-97+1) + 97;
   cout << endl << "Ersetzungszeichen: " << zeichen << endl;

   for(unsigned int i=0;i < strlen(text);i++)
   {
      verschluesselt[j++] = text[i];
      verschluesselt[j++] = zeichen;
   }
   verschluesselt[j] = '\0';

   cout << verschluesselt << endl;
   getchar();
}
```

Aufgabe: Quizfrage Widerstand

Der Programmbenutzer soll vom Rechner gefragt werden:

Wie lautet die Maßeinheit für den Widerstand?

Wird die Frage mit ***Ohm*** beantwortet, so soll der Rechner mit ***Richtig!*** antworten. Bei jeder anderen Eingabe soll die Rechnerantwort solange ***Falsch! Noch einmal:*** lauten, bis schließlich die richtige Antwort eingegeben wird. Danach soll das Programm enden.

```
#include <stdio.h>
#include <conio.h>
#include <iostream.h>
#include <iomanip.h>
#include <stdlib.h>

void main()
{
   char text[21];

   cout << "Wie lautet die Maßeinheit für den Widerstand?" << endl
<< endl;

   do
   {
      cin.getline(text, 20);

      if (strcmp(text, "Ohm") != 0)
         cout << "Falsch! Noch einmal: ";
      else
         cout << "Richtg";

   }  while(strcmp(text, "Ohm") != 0);

   getchar();
}
```

Aufgabe: Palindrom

Ein Palindrom ist ein Wort oder Satz, das bzw. der von vorn und von hinten gelesen den gleichen Wortlaut besitzt, z.B.:

OTTO
REITTIER
LAGERREGAL
RELIEFPFEILER
EIN NEGER MIT GAZELLE ZAGT IM REGEN NIE
NEUER DIENST MAG AMTSEID REUEN
DIE LIEBE TOTE! BEILEID!
LEG IN EINE SO HELLE HOSE NIE'N IGEL!

Das Programm soll testen, ob eine eingegebene Zeichenkette ein Palindrom ist oder nicht!

Beachten Sie, dass bei Sätzen keine Satzzeichen eingegeben werden dürfen und die Leerzeichen zwischen den Wörtern zu ignorieren sind!

```
#include <stdio.h>
#include <conio.h>
#include <iostream.h>
#include <stdlib.h>

void main()
{
   char text[201];
   unsigned int i, j;
   bool palindrom = true;

   cout << "Das Programm prüft, ob ein eingegebener Text ein Palindrom
ist!" << endl;

   cout << "Bitte Text eingeben!" << endl;
   cin.getline(text, 200);

   for (i=0, j=strlen(text)-1; i<strlen(text); i++, j--)
   {
      while (text[i] == ' ')
         i++;

      while (text[j] == ' ')
         j--;

      if (text[i] != text[j])
      {
         palindrom = false;
         break;
      }
   }

   if (palindrom == true)
      cout << " Text ist ein Palindrom!";
   else
      cout << " Text ist kein Palindrom!";

   getchar();
}
```

Aufgabe: Umlaute ersetzen

Schreiben Sie ein Programm, das einen einzugebenden Text (String) von maximal 80 Zeichen auf Umlaute und ß untersucht. Diese Zeichen sollen entsprechend der folgenden Zuordnung ersetzt werden:

Ä ⇒ Ae Ö ⇒ Oe Ü ⇒ Ue ß ⇒ ss

ä ⇒ ae ö ⇒ oe ü ⇒ ue

Beachten Sie, dass sich die Länge des Ergebnisstrings im Extremfall verdoppeln kann!

```
#include <stdio.h>
#include <iostream.h>

void main()
{
   char Text[81];
   char Ersatz[161];

   unsigned int i, j=0;

   cout << "Programm ersetzt Umlaute!" << endl << endl;
   cout << "Bitte Text eingeben!" << endl;
   cin.getline(Text, 80);

   for (i=0; i < strlen(Text); i++)
   {
      switch(Text[i])
      {
         case 'ä' : Ersatz[j++] = 'a';
                    Ersatz[j++] = 'e';
                    break;
         case 'ö' : Ersatz[j++] = 'o';
                    Ersatz[j++] = 'e';
                    break;
         case 'ü' : Ersatz[j++] = 'u';
                    Ersatz[j++] = 'e';
                    break;
         case 'Ä' : Ersatz[j++] = 'A';
                    Ersatz[j++] = 'e';
                    break;
         case 'Ö' : Ersatz[j++] = 'O';
                    Ersatz[j++] = 'e';
                    break;
         case 'Ü' : Ersatz[j++] = 'U';
                    Ersatz[j++] = 'e';
                    break;
         case 'ß' : Ersatz[j++] = 's';
                    Ersatz[j++] = 's';
                    break;
         default:   Ersatz[j++] = Text[i];
      }

   }
   Ersatz[j] = '\0';
   cout << "Text mit ersetzten Umlauten: " << Ersatz << endl;
   getchar();
}
```