Programmieren 1
C++

Inhalt

- Infos und Vorbemerkungen
- Grundlegende Begriffe
- Algorithmus: Begriff und Darstellung
- Einführung in die imperative Programmierung mit C++

Infos und Vorbemerkungen

Literatur zum Thema C++:

- Deitel, H. M., Deitel, P. J.: "C++ How to Program"; Upper Saddle River, NJ (Prentice Hall) 2001[3].
- Eckel, Bruce: "Thinking in C++"; Upper Saddle River, NJ (Prentice Hall) 2000.
- Stroustrup, B.: "The C++ Programming Language"; Reading, Massachusetts (Addison-Wesley [3]) 1997.
- Schader, M., Kuhlins, S.: "Programmieren in C++"; Berlin (Springer [4]) 1998.
- Niemann, A., Heitsiek, S.: "Das Einsteigerseminar C++ - Objektorientierte Programmierung"; Bonn (moderne industrie) 2002.
- Krienke, R.: "C++ kurzgefaßt"; (Spektrum)

Literaturempfehlungen

- Sehr schönes, sehr ausführliches Lehrbuch, ausreichend für die Vorlesung:
  Stanley B. Lippman et al., "C++ Primer", 4th ed., Addison-Wesley (2005)
- Gut für Anfänger, scheint aber vergriffen zu sein:
  A. Willms, "GoTo C++ Programmierung", Addison-Wesley (1999)
- Was Schönes zum Schmökern, teilweise subjektive Sichtweisen bzgl. Programmierstil:
  M. Cline et al., "C++ FAQs", 2nd ed., Addison-Wesley (2003)

- Kurzreferenz zum Thema STL (kein Lehrbuch!):
  R. Lischner, "STL Pocket Reference", O'Reilly (2003)
- Für die, die alles wissen wollen und auch vor "Amtsenglisch" nicht zurückschrecken (kein Lehrbuch!):
  "The C++ Standard, Incorporating Technical Corrigendum 1". John Wiley (2003)

Literaturempfehlungen

- Ulrich Breymann
  C++ Einführung und professionelle Programmierung
  HANSER 2003

Programmierumgebungen

- MS Visual C++ / Visual Studio (FH – RZ, Informatik R-Lab)
- Borland C++ (FH – RZ, Informatik R-Lab)

- Studentenversionen (max. 150,- €)
  - Im Fall der MS-Produkte: kostenlos für Studenten und Angestellte akademischer Institutionen

- Freeware –Compiler, Kommandozeilen-Versionen (z.B. GNU C++ Compiler)
  - Grafische Entwicklungsumgebungen sind heute ebenfalls frei verfügbar
  - Vorteile: kostet nichts, weit verbreitet, bei Linux immer dabei
  - Neueste Versionen sind auch sehr standardkonform

Bemerkungen zum Nachbereiten - Erwartungen

- Wichtiges mitschreiben, im Skript ergänzen
- Die Ziele und Inhalte der Veranstaltung reflektieren
  - was wollte der/die Dozent/in ? Was haben wir gemacht ?
  - was ist neu ?
  - wie funktioniert das?
  - Beispiele nacharbeiten
- Eigene Übung anfertigen
- Stoff in einer Lerngruppe nacharbeiten
  - eröffnet neue Fragen / Lösungen
  - hilft bei Problemen
  - trainiert das Sprechen über den Stoff / das aktive Bearbeiten → Verständnis
- Querbezüge zu anderen Fächern herstellen – Vernetzung im Kopf!
- Lernumgebung und -Bereitschaft schaffen (!!!)

Grundlegende Begriffe

Vorausgehende Überlegungen

- **Was ist ein Computer ?**
- **Wie zeichnet sich Problemlösungsverhalten mit dem Computer aus ?**
- **Was muss ich wissen, um mit dem Computer Probleme lösen zu können ?**

Aufgabenteilung Mensch / Maschine

- **Der Mensch muss ...**
  - die allgemeine Lösungsmethode einmal erarbeiten
  - die Lösungsmethode in maschinengerechter Art formulieren
- **Der Automat / Computer ...**
  - "braucht sich um Methode nicht zu kümmern"
  - Liegt ein Programm für ein Problem vor und existiert ein entsprechender Automat zur automatischen Ausführung, so ist das Problem prinzipiell und für alle Zeiten gelöst
  - Der Automat ersetzt menschliche Arbeitskraft (Rationalisierung), genauer geistige Arbeitskraft

Computer - Problem - Programmiersprache

- **Computer sind elektronische Automaten**
- **Lösung eines Problems**
  - Viel Wissen über das Problem erforderlich
  - Relativ geringes Wissen über den Computer als Maschine notwendig
- **Programmiersprache**
  - Ausdrucksmittel des Menschen gegenüber der Maschine
  - Eigenschaften der Programmiersprache gegenüber dem
    - Problem und dem
    - formulierenden Menschen

- Hardware / Software
- Algorithmus (→ eigenes Kapitel)
- Programm
- Programmierstile
- Programmiersprachen
  - C / C++
  - aber auch:
    Pascal, Modula, Ada, Cobol, Fortran, BASIC, Oberon, Lisp, Prolog, ML, Forth, Assembler, SQL, Pearl, ...)
  - Vielzahl von Scriptsprachen: Perl, Shell, JScript, …

- Hardware - Komponenten
  - Ein-/Ausgabegeräte (Bildschirm, Drucker, Scanner, Tastatur, ...)
  - Prozessoren (Pentium,..., interne Rechen- und Verarbeitungseinheiten, Arithmetik, Logik,..., Operationen auf Daten)
  - Speicher (RAM, Diskette, Festplatte, ...)
  - Verbindungsnetzwerke (LAN, Internet, ...)
- Bezug zur Programmierung
  - allzwecktauglich (~ Allzweck-Rechner - General Purpose Computer)
  - in der Regel nicht selbstständig arbeitsfähig
  - wird durch Software zur Anwendungsmaschine, SW definiert die Funktionalität des Rechners

Software: Programmausstattung einer Rechenanlage

- Bearbeitungsvorschrift für den Rechner
- steuert Hardware-Abläufe entsprechend Anwenderwunsch / Aufgabenstellung
- organisiert die Rechenanlage
- macht sie als Anwendungsmaschine betriebsbereit

↗ Algorithmen

Algorithmus

Begriff und Darstellung

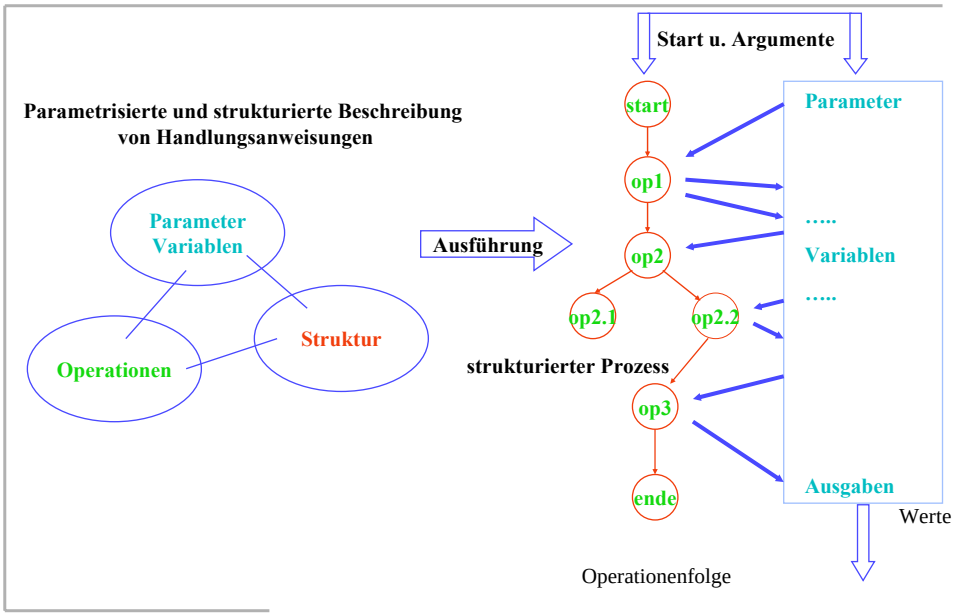
Algorithmus: formale Verarbeitungsvorschrift ...

- endliche Vorschrift (finit, endl. Länge der Formulierung)
- löst *Klasse* von Problemen (Klasse ~ Sorte, Typ ; spezielles Problem ~ Problemauswahl durch Parameter, Eingabewerte)
- mit Eingabewerten (veränderliche, das Problem charakterisierende Werte ≡ Parameter)
- vorherbestimmter Ablauf (deterministisch, Zustände in jeder Phase nachvollziehbar und voraussehbar)
- ist wiederholbar (determiniert)
- Ergebnis nach endlich vielen Schritten (Halten, Terminieren)

ein Programm ist ein in einer Programmiersprache formulierter Algorithmus ...

mit den unter “Algorithmus” beschriebenen Eigenschaften:

- finit
- parametrisiert
- determiniert
- deterministisch
- terminierend



- **Parameter, Werte, Größen, Variablen**
  - **Eingaben – Argumente/Werte, die die Parameter-Variablen des Algorithmus ersetzen und mit denen der Algorithmus arbeitet**
  - **interne Werte, Zwischenergebnisse und Zustände, die der Algorithmus verwendet**
    vgl. Nebenrechnungen beim Rechnen mit Bleistift und Papier
  - **Ausgaben, die der Algorithmus erzeugt**
    Berechnungsergebnisse, Ziel der Bearbeitung des Algorithmus
- **Operationen bzw. Funktionen, die der Algorithmus beschreibt, die der Bearbeiter ausführt**
- **Struktur, die die auszuführenden Operationen und Funktionen verknüpft**

Elemente von Algorithmen

- **Anweisungen** an den Bearbeiter (Mensch / Maschine) - **Operationen**

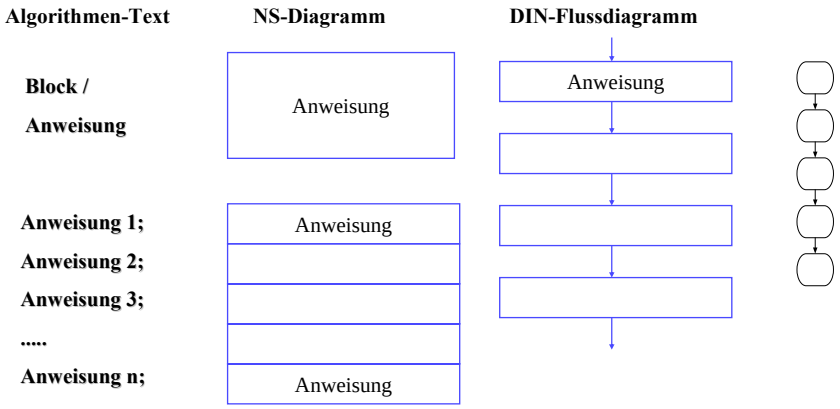
  Imperative: lies ein, berechne, transportiere, empfange, sende, wandle um, suche, …, überprüfe, ob … und …, gib aus

- **Struktur** bestimmt den Ablauf, die Ausführungsreihenfolge der Operationen
  - Sequenzielle Ausführung der Anweisungen (Aufschreibreihenfolge)
  - Bedingte Ausführung einer Anweisung (wenn ... dann ...)
  - Alternative Ausführung zweier Anweisungen (wenn ... dann ... sonst ...)
  - Alternative Ausführung mehrerer Anweisungen (... falls ... dann ..., falls ... dann ..., falls...dann ..., falls...dann ...)
  - Wiederholte Ausführung von Anweisungen
    - feste Anzahl Wiederholungen
      führe ... mal aus: .....
    - unbestimmte Anzahl mit Abbruchbedingung
      solange ... führe ... aus
      führe … aus , bis …

(Pseudocode-Notation: halbformal, noch nahe an Umgangssprache, meist englisch)

Struktur - Sequentielle Ausführung

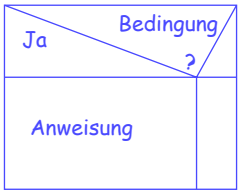
- **Ausführen der Anweisungen in der Reihenfolge der Aufschreibung**

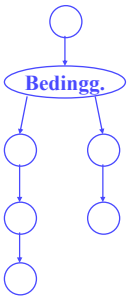


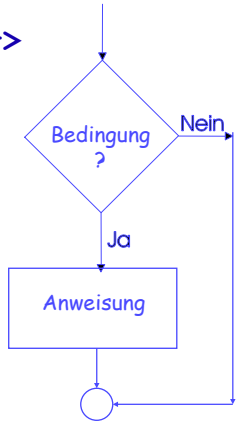
metasprachlicher Ausdruck

Wenn <bedingung > gilt, dann <anweisung> ausführen

```
IF <bedingung> THEN <anweisung>
```



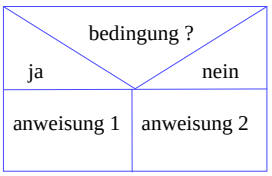




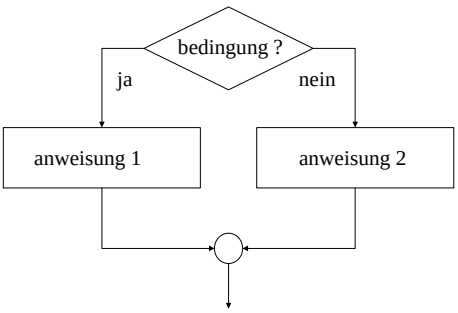
weiter mit Folien „Strukturierte Programmierung"

Vollständige Fallunterscheidungen

- wenn <bedingung> gilt, dann <anweisung 1>, sonst <anweisung 2>
- `IF <bedingung> THEN <anweisung 1> ELSE <anweisung 2>`



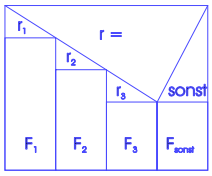
- Entspricht der Alternative
- Mit leerem Nein-Zweig: Bedingte Anweisung



Mehrfachverzweigung / Abfragekaskade

Mehrfachverzweigung / Switch-Konstrukt

Vergleiche r mit r_i
bei Gleichheit $r = r_i$ führe F_i aus
wenn r ungleich allen r_i : führe "sonst" – Alternative aus




```
SWITCH  <vergleichswert>
    CASE <alternativwert 1>:    <anweisung 1>;
    ….
    CASE <alternativwert n>:   <anweisung n>;
    ELSE <sonst-anweisung>;
```


solange <bedingung > gilt, wiederhole <anweisung> — **Kopfschleife**

```
while (<bedingung>)   <anweisung>
```

wiederhole <anweisung>, solange <bedingung> gilt — **Fußschleife**

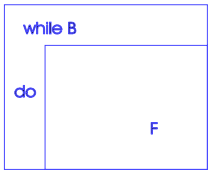
```
do {<anweisung>}  while (<bedingung>);
```

wiederhole für <anzahl> mal die <anweisung> — **Zählschleife**

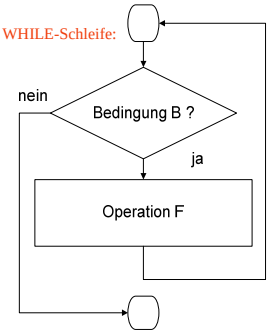
```
for ( <zähleranfang>  ; <zählbedingung>
      ; <zählschritt>  ) {<anweisung>}
```

- **Schleifen von vorangehender Bedingungsabfrage**
  - **solange Bedingung B erfüllt, führe Operation F aus**

```
WHILE B DO F
```



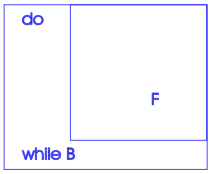
Flussdiagramm für WHILE-Schleife:



- **DO-WHILE-Schleife (Bedingungsprüfung am Ende)**
  - **führe Operation F aus, solange Bedingung B erfüllt ist**

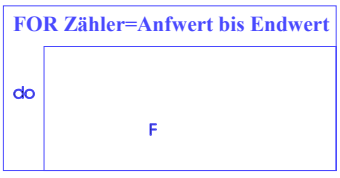
```
DO F WHILE B
```

Flussdiagramm für DO-WHILE-Schleife:

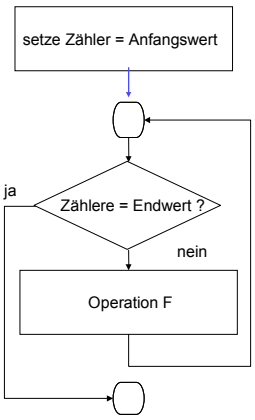


- **Schleifenkonstrukt mit fester Wiederholungszahl**

Nassi-Shneiderman-Diagramm für Zähl-Schleife:



Flussdiagramm für Zähl-Schleife:



Alg 1: Entwerfen Sie einen Algorithmus für das Einlassen einer Badewanne mit geeigneter Wassertemperatur zwischen 38 und 45 ºC und einem Wasserstand von 35-40 cm. Wählen Sie die Parameter / Größen, mit denen der Algorithmus arbeitet.

Zur Verfügung stehende Operationen:

- **Starten;**
- **Kaltwasser öffnen; … schließen**
- **Warmwasser öffnen; … schließen**
- **10 sec warten;**
- **Wassertemperatur messen;**
- **Wasserstand messen;**
- **beenden;**

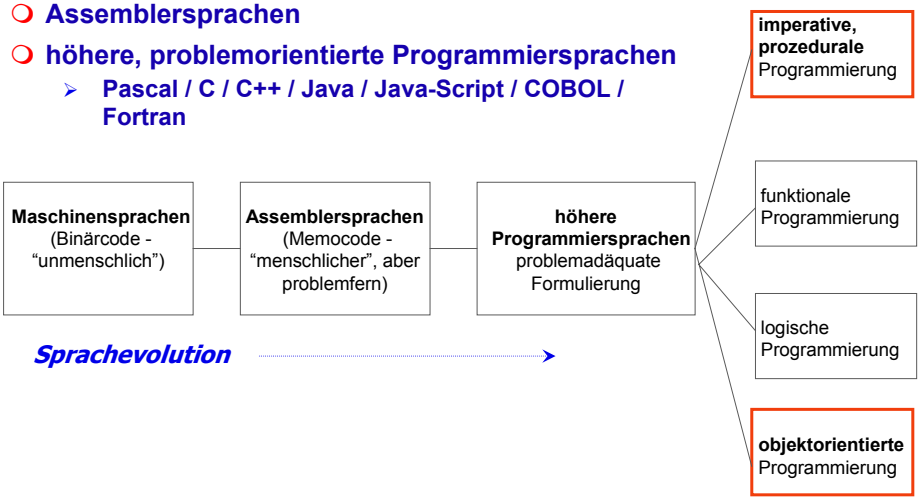
Alg 2: Entwerfen Sie einen Algorithmus für die schriftliche Addition zweier ganzer Zahlen. Wählen Sie die Parameter / Größen und die Operationen, mit denen der Algorithmus arbeiten soll.

Alg 3: Beschreiben Sie einen Algorithmus für den Vergleich (=, <) zweier Zahlen l und m. Operationen für den Zugriff auf Ziffern (Stellen) und Stellenwert seien vorhanden.

Alg 4: Beschreiben Sie einen Algorithmus, der überprüft, ob zwei gegebene Texte zeichenweise übereinstimmen. Welche Operationen und Größen werden benötigt?

- **Was ist ein Programm?**
  - ein in einer Programmiersprache niedergeschriebener Algorithmus!
  - enger Blickwinkel auf "Programmierung"
- **Erweiterte Sicht:**
  - viele Kriterien zusätzlich zur reinen Umsetzung von Algorithmen in Programme → Programmiersprachen und ihre Merkmale und Leistungsfähigkeit
- **Zentrale Frage: Wie lässt sich der Algorithmus am besten zur praktischen Anwendung umsetzen?**

- **Maschinensprachen**
- **Assemblersprachen**
- **höhere, problemorientierte Programmiersprachen**
  - Pascal / C / C++ / Java / Java-Script / COBOL / Fortran



- **Entscheidung für eine Programmiersprache**
  - syntaktische u. semantische Klarheit
  - Ausdrucksfähigkeit, Problem-Angemessenheit
  - Erlernbarkeit, Handhabbarkeit
  - Methodenunterstützung
  - Verifizierbarkeit der Programme, Korrektheit der Implementierung
  - Modularisierungskonzepte, Wiederverwendbarkeit
  - Zertifizierbarkeit der Compiler, der Programme
  - Standardisierung, Normierung
  - Offenheit
- **Erfüllt C++ alle Kriterien ?**

Intermezzo

Zahlendarstellung im Computer

Intermezzo: Zahlendarstellung im Computer

- Das Stellenwertprinzip
- Bits, Bytes & mehr: das duale Zahlensystem (Binärsystem)
- Andere Stellenwertsysteme
  - Oktalsystem
  - Hexadezimalsystem
- Umrechnen zwischen verschiedenen Zahlendarstellungen
- Rechnen im Binärsystem
- Gebrochene Zahlen
  - Festkommadarstellung
  - Fließkommadarstellung
  - Rechnen mit gebrochenen Zahlen
- Spezielle Ausprägungen der Ganzzahl- und Fließkommadarstellung in der Datenverarbeitung
  - Byte, Word, Doppelwort, Quadwort, einfache/doppelte Genauigkeit

Das Stellenwertprinzip: Wie zählen wir?

- Die „gewohnte“ Zahlendarstellung (Dezimalsystem):
  - 10 Ziffern (0…9)
  - Eine Zahl hat beliebig viele Stellen, die mit je einer Ziffer besetzt sind
  - Jede Stelle einer Zahl hat die Wertigkeit

    $10^{\text{(Nummer der Stelle, von rechts gezählt, mit 0 beginnend)}}$

  - Die Ziffer auf einer Stelle gibt an, wie stark die Stelle gewichtet wird
  - Zentrale Rolle der Ziffer „0“!
- Beispiel:

$$107_d = 1 \bullet 10^2 + 0 \bullet 10^1 + 7 \bullet 10^0 = 100 + 0 + 7$$

- Zählen
  - Erhöhen der Einerziffer, bis „Überlauf“ stattfindet
  - Jede Stelle zählt die Überläufe der nächst niedrigeren Stelle
  - Bei Überlauf einer Stelle wird wieder bei 0 begonnen

- **Warum die Einschränkung auf 10 Ziffern?**
  - Im Prinzip ist eine beliebige Ziffernzahl denkbar
- **Z. B. unendlich viele Ziffern!**
  - Stellenwertsystem wird sinnlos
  - Jede Zahl wird durch eine eigene Ziffer (ein Symbol) dargestellt
  - Hier haben wir ein Problem mit dem Kopfrechnen…
- **Anderes Extrem: nur zwei Ziffern, z.B 0 und 1**
  - „Dualsystem“, „Binärsystem“, „Basis 2“
  - Stellenwerte einer Zahl sind Potenzen von 2

7	6	5	4	3	2	1	0	Nummer der Stelle
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	Wertigkeit
128	64	32	16	8	4	2	1	

- **Beispiel:**
  $107_d = 1 \bullet 64 + 1 \bullet 32 + 0 \bullet 16 + 1 \bullet 8 + 0 \bullet 4 + 1 \bullet 2 + 1 \bullet 1 = 1101011_b$
- **Eine binäre Stelle wird als ein „Bit“ bezeichnet**
- **Acht binäre Stellen heißen „Byte“**
  - Welche Zahlen kann man mit einem Byte darstellen?

    00000000_b … $11111111_b \rightarrow 0_d$ … 255_d

  - Ein Byte ist die „Basis-Dateneinheit“ in der EDV, jeder Speicher ist in Bytes organisiert
  - Sehr geeignet zur Repräsentation von alphanumerischen Zeichen
    - Groß- und Kleinbuchstaben, Ziffern und übliche Sonderzeichen passen locker in den Bereich, der mit einem Byte abgedeckt werden kann (es ist sogar noch jede Menge Platz)
    - Der „nackte Text“ eines Dokumentes mit 2000 Zeichen belegt im Computer 2000 Byte
- **Rechnen im Binärsystem ist extrem einfach (s.u.)**

Andere Stellenwertsysteme

- **Basis 8 (Oktalsystem) und Basis 16 (Hexadezimalsystem) sind von größerer Bedeutung**
- **Oktalsystem: Tritt gelegentlich in der Sprache C auf**
- **Hexadezimalsystem: Extrem praktisch für den Umgang mit Binärzahlen**
  - **16 Ziffern: 0…9, A…F**
  - **Eine Stelle kann genauso viele Werte annehmen wie 4 Bits**
  - **Ein Byte lässt sich mit 2 Hexadezimalstellen exakt repräsentieren**
  - **Beispiel:**

$$107_d = 0110\ 1011_b = 6 \bullet 16 + 11 \bullet 1 = 6B_h$$

  - **Zahl der benötigten Stellen im Vergleich zum Binärsystem um Faktor 4 geringer**
  - **Ein Byte kann also die Werte 0_h … FF_h annehmen**
  - **Umrechnung hexadezimal – binär ist einfach**
- **Wie rechnet man auf einfache Weise von dezimal nach binär und zurück?**

Umrechnen von/nach Binärdarstellung

- **Binär → dezimal**
  - **Einfach die (dezimalen) Werte der Binärstellen addieren, die nicht 0 sind**
  - **Beispiel:**

$$1001011_b = 1 \bullet 64 + 0 \bullet 32 + 0 \bullet 16 + 1 \bullet 8 + 0 \bullet 4 + 1 \bullet 2 + 1 \bullet 1 = 75_d$$

- **Dezimal → binär: Rechenvorschrift (Algorithmus) zur Konvertierung**
  - **Beispiel:**

75_d / 2 = 37	Rest 1
37 / 2 = 18	Rest 1
18 / 2 = 9	Rest 0
9 / 2 = 4	Rest 1
4 / 2 = 2	Rest 0
2 / 2 = 1	Rest 0
1 / 2 = 0	Rest 1

1001011_b

Rechnen im Binärsystem

- Einfachste Operation: Multiplikation mit / Division durch 2
  - Division durch 2: schieben um eine Stelle nach rechts
  - Multiplikation mit 2: schieben um eine Stelle nach links
  - Analog Dezimalsystem
- Addition/Subtraktion: Analog Dezimalsystem, Übertrag ist maximal 1
- Multiplikation ist simpel, denn es wird im Gegensatz zum Dezimalsystem nur mit einer Zweierpotenz multipliziert
  - Beispiel:

```
010 • 101
  010
   000
    010
  01010
```

- Wie stellt man negative Zahlen dar?
  - Zahl negieren = jedes Bit umdrehen und zum Ergebnis 1 addieren („Zweierkomplement“)
  - Beispiel: $-1_d = -(00\ldots0001_b) = 11\ldots1110_b + 1 = 11\ldots1111_b$

Negative Zahlen im Binärsystem

- Warum funktioniert das mit dem Zweierkomplement?
  - Wir suchen nach einer Zahl, die addiert zu einer positiven Zahl Null ergibt
  - 1. Versuch: Einfach alle Bits umdrehen („invertieren“)

```
 01001011
+10110100
 11111111
```

nicht ganz 0, aber…

  - Addition einer weiteren 1 ergibt 0 (Überlauf ignoriert):

```
 01001011
+10110100
+00000001
100000000
```

- Das funktioniert, weil endlich viele Stellen zur Darstellung verwendet werden und der Überlauf einfach unter den Tisch fällt
- Kriterium für größer oder kleiner Null: oberstes Bit
  - Oberstes Bit = 1 → negative Zahl
- Damit funktioniert die Addition von Ganzzahlen auch im negativen Bereich ohne weitere Sonderbehandlung des Vorzeichens

Gebrochene Zahlen - Festkommadarstellung

- **Einfachste Möglichkeit für gebrochene Zahlen: Festkommadarstellung**
  - Komma sitzt an einer festen Stelle
  - Konstante Anzahl von Vor- und Nachkommastellen, konstante Wertigkeit jeder Stelle
  - Nachteil: Sehr kleine und große Zahlen können nicht dargestellt werden
  - Beispiel im Dezimalsystem: Sei die Zahl der Nachkommastellen gleich 4

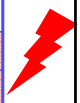
```
253     →   253,0000
2,435   →   002,4350
0.00024     →   000,0002
0.00002     →   000,0000
```

  - Dies ist für die meisten praktischen Anwendungen unbrauchbar!
  - Weiteres Problem: Stellenzahl im Computer ist begrenzt, d.h. auch nach oben wird es schnell eng
- **Wie nutzt man die zur Verfügung stehenden Stellen besser aus?**
  - Fließkommadarstellung! (Gleitpunkt, „floating point“)

Gebrochene Zahlen - Fließkommadarstellung

- **Warum sollte das Komma überhaupt an einer festen Stelle stehen?**
  - Wenn die Position des Kommas irgendwo gespeichert wäre, hätte man ebenfalls die volle Information über die Zahl!
- **Strategie: Gespeichert wird**
  - die Mantisse der Zahl (eine Anzahl der höchstwertigen Stellen, ohne führende Nullen)
  - der Exponent (die Stelle, an der das Komma steht – von *links* gezählt)
- **Beispiel im Dezimalsystem**
  - Zur Verfügung stehen 4 Mantissenstellen und ein Exponent mit einer Stelle plus Vorzeichen

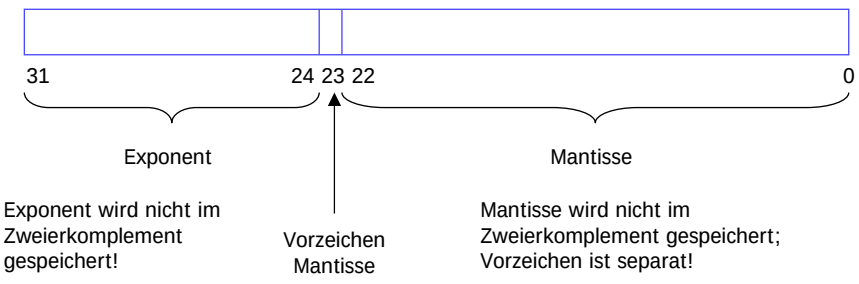
Zahl	normiert	Mantisse	Exponent	Festkomma (5 Stellen)
122,5	$0{,}1225 \bullet 10^{3}$	1225	3	122,50
0,004298	$0{,}4298 \bullet 10^{-2}$	4298	-2	000,00
6679,4	$0{,}66794 \bullet 10^{4}$	6679	4	679,40



  - Konvention: Mantisse soll keine führenden Nullen haben („Normierung“)!
  - Darstellbarer Zahlenbereich: $1{,}000 \bullet 10^{-9}$ … $9{,}999 \bullet 10^{9}$ (19 Größenordn.!)

- **Vorteile der Fließkommadarstellung**
  - Es werden immer die höchstwertigen Stellen gespeichert
  - Genauigkeitsverlust tritt somit immer in den niedrigstwertigen Stellen auf
  - Erweiterung des Zahlenbereiches und der Rechengenauigkeit sind getrennt möglich
    - Mehr Stellen in der Mantisse → mehr Genauigkeit
    - Mehr Stellen im Exponenten → größerer Zahlenbereich
- **Nachteile**
  - Etwas kompliziertere Operationen beim Rechnen (s.u.)
  - Fallstricke (Genauigkeitsprobleme) bei der Addition (s.u.)
  - Speichern von vier Werten notwendig
    - Mantisse
    - Exponent
    - Vorzeichen Mantisse
    - Vorzeichen Exponent

- **Fließkommazahlen im Computer**
  - Binärsystem!
  - Packe komplette Zahl (Mantisse, Exponent etc.) in eine Anzahl Bytes, z.B. 4
  - Mantisse und Exponent „teilen“ sich die Bits
  - Führende Eins der Mantisse wird üblicherweise nicht gespeichert
  - Beispiel: 32 Bit Fließkommazahl (4 Bytes)



Multiplikation

- Addition der Exponenten der Faktoren
- Multiplikation der Mantissen der Faktoren
- Normierung des Ergebnisses (keine führenden Nullen in der Mantisse)
- Beispiel (5 Stellen Mantisse):

$0{,}30422 \bullet 10^{3} \bullet 0{,}20024 \bullet 10^{-4} = (0{,}30422 \bullet 0{,}20024) \bullet 10^{(3-4)}$
$= 0{,}0609170128 \bullet 10^{-1}$
$= 0{,}60917 \bullet 10^{-2}$

- Genauigkeitsverlust in den hintersten Stellen ist meist unvermeidlich, aber oft akzeptabel
- Vorsicht: Nach vielen Rechenoperationen können sich Fehler in den letzten Stellen trotzdem zu deutlichen Abweichungen vom exakten Ergebnis akkumulieren!

Addition

- Verschiebung der Kommas in den Summanden, damit Exponenten übereinstimmen
- Addition der Mantissen
- Normierung des Ergebnisses
- Beispiel (5 Stellen Mantisse):

$0{,}25922 \bullet 10^{1} + 0{,}20024 \bullet 10^{-2} = (259{,}22 + 0{,}20024) \bullet 10^{-2}$
$= 259{,}42024 \bullet 10^{-2}$
$= 0{,}25942 \bullet 10^{1}$

- Genauigkeitsverlust kann gravierend sein, je nachdem wie weit die Exponenten auseinander liegen!
- Im schlimmsten Fall wird ein Summand zu Null renormiert:

$0{,}25922 \bullet 10^{1} + 0{,}20024 \bullet 10^{-5} = (25922 + 0{,}020024) \bullet 10^{-4}$
$= 25922{,}020024 \bullet 10^{-4}$
$= 0{,}25922 \bullet 10^{1}$

- D.h. die Addition von Zahlen mit weit auseinanderliegenden Exponenten ist gefährlich!

- **Addition (Forsetzung)**
  - **Weitreichende Folgen: Addition von Fließkommazahlen im Computer ist** nicht assoziativ
  - **Beispiel:**

$$\underbrace{((\ldots(0{,}25922 \bullet 10^{1} + 0{,}20024 \bullet 10^{-5}) + 0{,}20024 \bullet 10^{-5}) + \ldots)}_{\text{20000 mal}} = 0{,}25922 \bullet 10^{1}$$

  - **Aber:**

$$0{,}25922 \bullet 10^{1} + (0{,}20024 \bullet 10^{-5} + 0{,}20024 \bullet 10^{-5} + \ldots)$$
$$= 0{,}25922 \bullet 10^{1} + 0{,}40048 \bullet 10^{-1}$$
$$= 0{,}2632248 \bullet 10^{1}$$

  - **D.h. das Ergebnis hängt stark von der Klammerung der Summanden ab**

- **Blindes Vertrauen in numerische Ergebnisse des Computers kann katastrophal sein!**

- **Konversion vom Dezimal- ins Binärfließkommaformat**
  - **Problem: Endliche Dezimalbrüche können zu unendlichen Binärbrüchen werden**
  - **Beispiel:** $1{,}2_d = 1{,}001100110011\ldots_b = 1{,}\overline{0011}_b$
  - **Wird üblicherweise in Kauf genommen, kann aber zum ernsten Problem werden, wenn sich Rundungsfehler akkumulieren**
  - **Bereits bei der Zahleneingabe entstehen durch das Zahlenformat bedingte Fehler**
  - **Tip: Fließkommazahlen nie auf exakte Gleichheit testen!**
    - **Besser: kleinen Fehler beim Vergleich zulassen (z.B. 10^{-15})**
- **Typische Fließkomma-Zahlenformate in der EDV**
  - **Einfache Genauigkeit: 32 Bit (23 Bit Mantisse)**
  - **Doppelte Genauigkeit: 64 Bit (53 Bit Mantisse)**
  - **Erweiterte Genauigkeit: 80 Bit (63 Bit Mantisse)**
- **Typische Ganzzahlformate**
  - **Byte (mit Vorzeichen): 8 Bit**
  - **2 Byte = 1 Wort (16 Bit)**
  - **4 Byte = 1 Doppelwort (32 Bit)**
  - **8 Byte = 1 Vierfachwort („Quadword“) (64 Bit)**

s. später!

Einführung in die imperative Programmierung mit C++

Objektorientierung als Vertiefung

- **Analogie: Rezept (→ Algorithmus)**
  - **man nehme zuerst..., dann ... Tue ... Solange bis ... Falls ...**
- **Programm besteht aus Anweisungen an den Computer, die beschreiben, was er zu tun hat bzw. wie er vorzugehen hat, eine Aufgabenstellung zu lösen**
  - `a = b + c*c + 2*(b - 1);` **arithmetischer Ausdruck**
  - `cout << "Hallo Du!";` **Textausgabe**
  - `einfuege(datei,zeile,spalte,zeichen);` **Funktionsaufruf**
  - `c = sqrt (a*a + b*b);` **arithmetischer Ausdruck mit Funktion**
  - `while (Temperatur < 100)`
    `warte(1);` **bedingte wiederholte Ausf.**

Programm zur Ausgabe einer Zeile auf dem Bildschirm

```
#include <iostream> /* eine Anweisung an den Präprozessor */
using namespace std;// Namensraum

/* jetzt die Definition des Hauptprogramms */
int main (void)      // Hauptprogramm/Funktionsdeklaration
{
  cout << "Meine erste Zeile auf dem Bildschirm !\n";
  cout << "und, weil es so schön ist, gleich noch eine
  hinterher";
  cout << endl;
  return 0;
}
```

Elemente dieses Programms???

- Präprozessoranweisung
- Kommentar
- Namensbereich
- Funktionsaufrufe
- Typvereinbarungen
- Wertrückgabe
- Bildschirmausgabe

Besonderheit bei älteren Compilern:
`#include <iostream.h>`
statt der obersten beiden Zeilen!

Programmkomponenten, im einzelnen

`/* ... */`	Kommentar über mehrere Zeilen
`//`	Kommentar bis zum Zeilenende (in C++ / in C ab C99)
`using namespace ...`	nicht vereinbarte Namen in angegebenem Namensbereich suchen
`main`	Name der Funktion für das Hauptprogramm - genau einmal je Programm
`int`	Datentyp integer (ganze Zahl) für Rückgabewert der Funktion
`void`	Parametertyp, falls keine Übergabe- oder Rückgabe-Parameter gewünscht
`{ ... }`	Block-Klammerung (Begin ... End)
`cout`	"Operation" zur (formatierten) Ausgabe auf Console
`\n`	Pseudozeichenfolge (Escape-Sequenz) zur Steuerung der Ausgabe
`endl`	end of line / Zeilenende explizit
`return`	Bestimmen des Rückgabewerts einer Funktion

Vom Text zum ausführbaren Programm (Unix-Variante)

- **Programmerstellung (Eingabe) mit beliebigem Texteditor**
  - `vi prog.c`
    - Programmeingabe
    - Abspeichern & Verlassen des Editors
    - Programm steht in der Datei `prog.c`
- **Compilieren (Übersetzen & Binden)**
  - `g++ prog.cc`
    - g++: Aufruf eines C++-Compilers (z.B. GNU-Compiler unter Unix, Linux oder Win95)
    - Programm wird übersetzt, ggf. Fehlermeldungen auf Bildschirm ausgegeben
    - Bei fehlerfreier Übersetzung steht in einer neuen Datei (z.B. a.out unter Unix) das ausführbare Programm
- **Ausführen / Programmaufruf**
  - `a.out` (oder anderer Name der erzeugten ausführbaren Datei)

```
Meine erste Zeile auf dem Bildschirm !
und, weil es so schön ist, gleich noch eine
hinterher
```

  - Nächste Eingabe, z.B. Weitereditieren mit vi

Vom Text zum ausführbaren Programm
(Variante mit integrierter Entwicklungsumgebung, v.a. Windows)

- **Start der Entwicklungsumgebung**
  - bcc, turbo, vc, klicki…
- **Erzeugen einer C/C++-Datei**
  - Befehl im Menue Datei/File: Neu bzw.
  - Öffen einer bereits vorhandenen Datei
    > Ein entsprechendes (ggf. leeres) Fenster wird geöffnet
- **Programmerstellung (Eingabe) in diesem Fenster**
- **Compilieren (Übersetzen & Binden)**
  - Auswahl des entsprechenden Compile/Make-Befehls im Menue 'Compile'
  - Evtl. Fehlermeldungen werden in einem separaten Fenster angezeigt
  - Durch Klicken auf eine Fehlermeldung wird an die Stelle im Programm gesprungen
- **Ausführen / Programmaufruf**
  - Auswahl des Run-Befehls im Menue 'Compile'

```
Meine erste Zeile auf dem Bildschirm !
und, weil es so schön ist, gleich noch eine
hinterher
```

  - Nächste Eingabe, z.B. Weitereditieren im Programmtextfenster

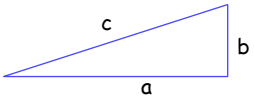
Übungsaufgabe

- **Schreiben Sie ein Programm zur Ausgabe des Textes**

```
Bumerang
War einmal ein Bumerang;
War ein weniges zu lang.
Bumerang flog ein Stueck,
aber kam nicht mehr zurueck.
Publikum - noch stundenlang -
wartete auf Bumerang.
            Ringelnatz
```


Ein etwas aufwendigeres Programm

- **Hypotenusenberechnung**
  - **Welche Komponenten werden für ein Programm benötigt, das die Berechnung der Hypotenuse aus den Katheten eines rechtwinkligen Dreiecks durchführt ?**



$$c = \sqrt{a^2 + b^2}$$

Berechnung der Hypothenuse - Welche Programmkomponenten werden benötigt?

- **Variable** — zum Speichern der Werte der Katheten und der Hypotenuse
- **Eingabefunktion** — zum Einlesen der Werte der Katheten
- **Math. Funktionen** — +, *, insbesondere Quadratwurzel
- **Wertzuweisung** — Zuweisung des Werts eines arithmetischen Ausdrucks an eine Variable
- **Ausgabefunktion** — Ausgeben der eingelesenen Werte und des berechneten Werts der Hypotenuse

Programm Pythagoras

```
#include <iostream>  // Headerfile für Ein-/Ausgaben
#include <cmath>     // Headerfile für math. Funktionen
using namespace std;

int main (void)
{
  double kathete1, kathete2, hypotenuse;

  cout << "Bitte gib die Katheten ein : \n";
  cin >> kathete1 >> kathete2;

  hypotenuse = kathete1*kathete1 + kathete2*kathete2;
  hypotenuse = sqrt(hypotenuse);

  cout << "Kathete 1:   " << kathete1 << endl;
  cout << "Kathete 2:   " << kathete2 << endl;
  cout << "Hypotenuse: " << hypotenuse << endl;

  return 0;
}
```

Variablenvereinbarung

Abfrage der Werte

Berechnung d. Hypot.

Ausgabe der Werte

Überblick über wesentliche Programmelemente

Wesentliche Programmelemente (1)

- **Hauptprogramm - Beginn des Programms**
  - **Schema:**

```
void main()
{
   <Anweisungen> ;
}
```

metasprachlicher Ausdruck in <…>

- **Variable vereinbaren mit Typ und Name (Speicher für Werte)**
  - **Deklaration (Vereinbarung):**

```
int      <name1>, <name2> ;   für ganze Zahlen
double   <name1>, <name2> ;   für reelle Zahlen
```

  - **Beispiele:**

```
int      i;
int      j, k, l;
double   a, b;
```

- **Wertzuweisung - Zuweisung des Werts eines (arithmetischen) Ausdrucks an eine Variable**
  - **Beispiel:**

```
i = 5;
a = 5.0;
```

  komplexere Ausdrücke:

```
j = 5*i + k*k*(i-1);
```

- **Eingabefunktion: Einlesen von Werten für Variable**
  - Notwendig am Dateianfang:
    ```
    #include <iostream>
    using nemaspace std;
    ```
  - Anweisungsschema:
    ```
    cin >> <Integer-Variable>;    oder
    cin >> <Double-Variable>;     auch
    cin >> <Int-Var> >> <Double-V.>;
    ```

- **Ausgabefunktion: Ausgeben von berechneten Werten**
  - Notwendig am Dateianfang:
    ```
    #include <iostream>
    using namespace std;
    ```
  - Ausgeben einer int-Var: `cout << <Integer-Var>;`
  - Ausgeben einer double-Var: `cout << <double-Var>;`
  - Ausgeben eines Zeilenendes: `cout << endl;  oder cout << "\n";`
  - Gemischt:
    ```
    cout << "Text: " << i << "Text: " << a;
    ```

- **Math. Funktionen - z.B. Quadratwurzel**
  - Notwendig am Dateianfang: `#include <cmath>`
  - Verwendung:
    ```
    a = sqrt(9.0);
    a = 3.0 * sqrt(9.0 * b + a);
    ```


- **Fallunterscheidung mit `if`**
  - Schema:
    ```
    if ( <Bedingungsausdruck> ) {
        < Anweisungsfolge >
    }
    else {
        < Anweisungsfolge >
    }
    ```
  - Relationale Operatoren & Verknüpfungen:
    `<, <=, >, >=, ==, !=     &&, ||, !`
  - Beispiele:
    ```
    if (i < 5) ...
    if (j >= 0 && a < 2.0 || b <= 5.0)...
    ```
  - Anmerkung: `else`-Teil kann entfallen, falls nicht benötigt

- **Wiederholung von Anweisungen mit `while`**
  - Schema:
    ```
    while ( <Bedingungsausdruck> )
    {
        < Anweisungsfolge >
    }
    ```
  - Beispiel: `while ( i < 5) { ... }`
  - Bedingungsausdruck wie oben; Anweisungsfolge in { ... } wird solange wiederholt, wie der Bedingungsausdruck den Wert "*wahr*" besitzt.

Aufgaben

- Verändern Sie das Programm zur Berechnung der Hypotenuse so, dass es nach der Berechnung bestimmt, ob das Dreieck gleichschenklig ist und dies gegebenenfalls am Bildschirm anzeigt.
- Schreiben Sie Programme zur Berechnung von Oberflächen und Volumina unterschiedlicher Körper. Gestalten Sie dazu geeignete Eingabedialoge mit dem Bediener und entsprechende Ausgaben.
- Schreiben Sie ein Programm, das als Eingaben die Längen der Seiten eines Dreiecks erwartet und daraus bestimmt, um welchen Typ Dreieck es sich handelt. Es soll auch geprüft werden, ob das Dreieck zu einer Strecke zusammenfällt.

C++-Programmierkurs

Gliederung C++-Programmierkurs, 1. Semester

1. Alphabet und Schlüsselwörter
2. Variable und Datentypen
3. Ein-/Ausgabe Teil I
4. Kontrollstrukturen
5. Funktionen
6. Datenstrukturen: Felder, Zeiger, Referenzen
7. Datenstrukturen: Strukturen, Klassen und Objekte
8. Ein-/Ausgabe Teil II: Arbeiten mit Dateien
9. Ausdrücke
10. Vertiefung einzelner Themen

(1) Alphabet und Schlüsselwörter

Alphabet – zugelassene Zeichen / Zeichensatz

- **Welche Zeichen sind in der Programmiersprache C++ überhaupt zur Eingabe des Codes zugelassen?**

- **alle Zeichen des englischen Alphabets**
  - Kleinschreibung (ohne ä, ö, ü)
  - Großschreibung (ohne Ä, Ö, Ü)
  - Ziffernzeichen

- **White – Space – Zeichen**
  - Leerzeichen, Zeilenende, horizontaler u. vertikaler Tabulator, Seitenvorschub

- **Sonderzeichen**

  { } [ ] ( ) < > + - * / % ^ ~ & | _ = ! ? # \ , . : ; , " '

Schlüsselwörter – reservierte Wörter

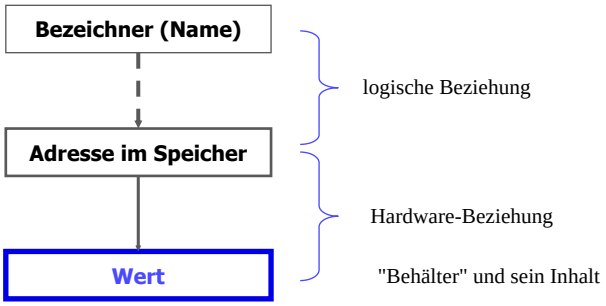
- **Welche Schlüsselwörter versteht C bzw. C++?**

C:	auto	double	int	struct
	break	else	long	switch
	case	enum	register	typedef
	char	extern	return	union
	const	float	short	unsigned
	continue	for	signed	void
	default	goto	sizeof	volatile
	do	if	static	while
C++:	class	inline	private	template
	delete	new	protected	this
	friend	operator	public	virtual

- **Bitte Schlüsselwörter nicht mit Anweisungen verwechseln!**

(2) Variable & Datentypen

Variable - Variablenmodell der imperativen Programmiersprachen



- Zugriff auf Wert (Behälterinhalt) über Bezeichner oder Adresse möglich
- Lebenszeit und -dauer abhängig von Gültigkeitsbereich
- Werte können zusammengesetzt sein → strukturierte Variablen

Programmvariablen ...

- **müssen deklariert sein , d.h. ...**
  - ihre Namen müssen vor Benutzung eingeführt werden
  - besitzen einen Datentyp, der ebenfalls dem Compiler bekanntgemacht werden muß
- **Variablen, die miteinander verknüpft werden, müssen verträgliche Typen besitzen**
  - siehe später
- **Beispiel :**

```
double  kathete1, kathete2, hypotenuse;
```

<Datentyp> <Liste der Variablen dieses Typs> ;

Konvertierbarkeit / Verträglichkeit

```
void main()
{
  int    zaehler, nenner;
  double ergebnis, faktor, erg2;

  zaehler = 8;
  nenner  = 16;
  faktor  = 1.5;

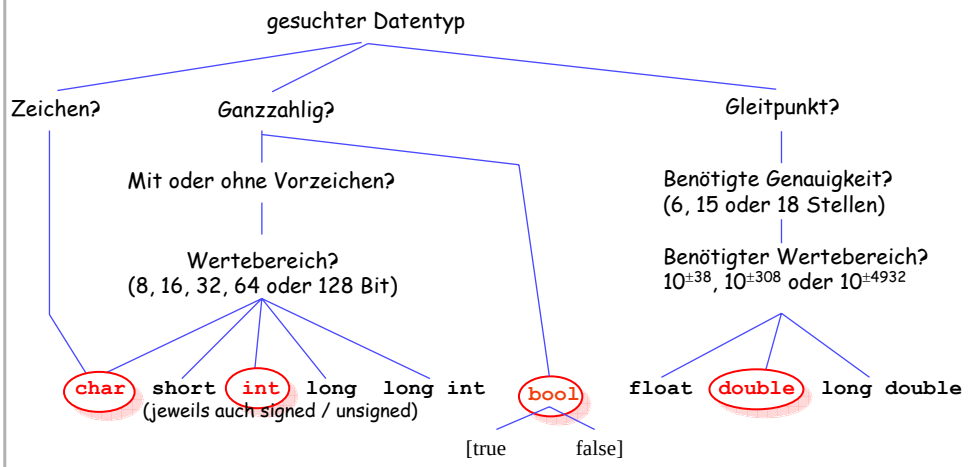
  // Beispiel für Anweisung mit Typanpassung:
  ergebnis = zaehler / nenner * faktor ;
  erg2     = faktor * zaehler / nenner ;
   // dabei Konvertierung von int-Darstellung in double
}
```

Wert von 'ergebnis' und 'erg2' ???

Wertzuweisung

- **<linke Seite>** = **<rechte Seite>**
  - veränderlicher Wert — wertliefernder Ausdruck
  - L-Wert — R-Wert
- **Zuweisungsoperator =**
  rechte Seite z.B. (arithmetischer) Ausdruck
- **Anweisung: <ausdruck> ;**
  Semikolon erzeugt aus Ausdruck eine Anweisung!
- **Beispiele:**
  - `x = 2;` Zuweisung + Ausdruck
    (C/C++-spezifisch! Nicht in anderen Prog.sprachen!)
  - `x = 2;` Anweisung
  - `a + 3;` Anweisung (aber in dieser Form sinnlos!)

Basisdatentypen



ANSI - Character-Datentypen

Datentyp	Mind.Anz. Bytes	Wertebereich (typisch)	Bemerkung
signed char char	1	[-128 , 127]	Ganzzahl mit Vorzeichen, Zeichen
unsigned char	1	[0 , 255]	Zeichen vorzeichenlose Ganzzahl

Achtung: Ob ein Zeichen (`char`) ein Vorzeichen trägt oder nicht, hängt u.U. vom Compiler ab!

ANSI - Integer-Datentypen

Datentyp	Mind.Anz. Bytes	Wertebereich (typisch)	Bemerkung
short, signed short signed short int **short int**	2	[-32768, 32767]	mind. so groß wie char
unsigned short **unsigned short int**	2	[0 , 65535]	so groß wie short
signed int signed, **int**	4	$[-2^{31} , 2^{31}-1]$	mind. so groß wie short
unsigned **unsigned int**	4	$[-2^{31} , 2^{31}-1]$	so groß wie int
signed long int signed long long int, **long**	4	$[-2^{31} , 2^{31}-1]$	mind. so groß wie int
unsigned long int **unsigned long**	4	$[0 , 2^{32}-1]$	so groß wie long

ANSI - Gleitkommadatentypen

Datentyp	Bytes	Mantisse	Bemerkung
float	4	23 Bits + Vz	Gen. 6 Stellen
double	8	52 Bits + Vz	Gen. 15 St.
long double	10 (8)	63 Bits + Vz	Mind. wie double

Typumwandlungen, cast

```
int     zaehler = 8 ;
int     nenner  = 16;
double  ergebnis;
double  faktor  = 1.5 ;
```

- `ergebnis = zaehler / nenner * faktor;` → Resultat = 0.0
- **"cast" – erzwungene, explizite Typumwandlung**
  - **(<neuer Datentyp>) <Ausdruck>, z.B.**
    `double_var = (double) int_var;`
  - `ergebnis = (double) zaehler;` → **Resultat = 8.0**
  - `ergebnis = (double)zaehler/(double)nenner * faktor;` → **Resultat = 0.75**
  - `ergebnis = (double) zaehler / nenner * faktor;` → **Resultat = 0.75**
  - `ergebnis = (double) (zaehler / nenner) * faktor;` → **Resultat = 0.0**
  - `ergebnis = faktor * zaehler / nenner;` → **Resultat = 0.75**

Direkte Konstanten, unmittelbar definiert

- **Numerische Konstanten sind in C/C++ typbehaftet!**

- `12345` → int
  `1.2345` → double (! nicht float !)
  `1.2345f` → float
  `0x1A , 0X1A` → int, hexadezimal (Wert: 26)
  `012345` → int, oktal (Wert: 5349)
  `7926U` → unsigned
  `7926L` → long
  `'A','\101','\x41'` → char, Wert abhängig vom Zeichensatz

- **Typinterpretation durch U(unsigned) / L(long) / f(float)**
- **Was ist `12/5` ?**

Deklarierte Konstanten

- **Deklarierte Konstanten sind an einen Bezeichner gebunden**
  `#define` <name> <zeichenfolge>
  - **textuelle Ersetzung durch den Präprozessor (vor Compiler)**
  - **jedes nachfolgende Auftreten von 'name' wird durch die in `#define` angegebene 'zeichenfolge' ersetzt!**
- **Konstanten (ANSI C & C++)**
  - **`const` <typ> <name> = <wert>; Beispiel:**
    `const double e = 2.71828182845905;`
  - **"name" muss gleich mit einem Wert initialisiert sein, sonst Fehler!**
  - **"name" bezeichnet nur einen typisierten Wert, es muss nicht unbedingt tatsächlich eine konstante Variable im Speicher abgelegt sein!**
  - **Empfehlung "`const`" gegenüber `#define` aufgrund der hier möglichen Typ- und Syntaxprüfungen unbedingt bevorzugen**

(3) Ein-/Ausgabe, Teil I

Vorgriff auf Objektorientierung:

- `cin` und `cout` sind globale Objekte (s. Objektorientierung)
- Vielzahl von Einstellmöglichkeiten und Funktionen
- zugehörige Header-Datei: `iostream`
  - "Ein- /Ausgabe-Streams"
  - `#include <iostream>`
    `using namespace std;` // aus Bequemlichkeit
  - Bei älteren C++-Dialekten: `iostream.h`

- **Einige wichtige Einstellmöglichkeiten ("Manipulatoren") bei der Ausgabe**

```
#include <iostream>
#include <iomanip>
using namespace std;

cout << setprecision(4) << x;      // Zahl der Mantissenstellen
cout << setw(10) << x;             // Feldbreite
cout << i << hex << j << dec << k;
                                   // Basis 16 bzw. 10
cout.fill('.');                    // Füllzeichen
```

- **Viele weitere Manipulationen möglich**
  - s. späteres Kapitel über Stream-I/O

- **Beispiele:**

```
cout.fill('.');
cout << setprecision(3) << setw(10) << 42.56;
```

ergibt als Ausgabe

```
......42.6
```

- **Achtung: `setw()` wirkt nur auf die jeweils direkt folgende Ausgabe**
  - Feldbreite wird normalerweise dynamisch dem Inhalt angepasst

- Eingabe von Daten mittels des Schiebeoperators >>

```
cin >> <variable> ;
```

- ">>" weiß für jeden Datentyp, was zu tun ist
- Eingabe überspringt rigoros sämtliche Leerräume (space, Tab etc.), bis ein zusammenhängender Eingabetext erscheint
- Trick: Warten auf Eingabe eines Zeichens

```
cin.get();
```


(4) Kontrollstrukturen

- Fallunterscheidungen
- Schleifen

- **bisher:**
  - lineare Programme
  - sequentielle Ausführung der Anweisungen
- **Probleme:**
  - keine Reaktion auf äußere und innere Bedingungen möglich
  - keine Ausnahmebehandlung / Fehlerbehandlung möglich
  - starrer Ablauf, keine Wiederholungen
  - Parametrisierung eingeschränkt
  - arme Algorithmen-Schemata für Programmentwurf
- **Abhilfe**
  - Anweisungen für die bedingte Ausführung von Anweisungsblöcken
  - Verzweigung und Mehrfachverzweigung, Kaskadierung von Bedingungen
  - Wiederholungsanweisungen für die wiederholte Ausführung von Anweisungsblöcken

- **allgemeine Formen bedingter Anweisungen**

```
if   ( <Bedingungsausdruck> )
  <Anweisung> ;                  // Einzelanweisung

if   ( <Bedingungsausdruck> )
  <Anweisung> ;                  // Einzelanweisung
else
  <Anweisung> ;                  // Einzelanweisung
```

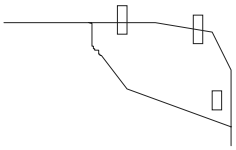
- **Behandlung von Anweisungsfolgen**

```
if ( <Bedingungsausdruck> ) {
  <Anweisungsfolge>              // Anweisungsfolge in { ... }
}

if ( <Bedingungsausdruck> ) {
  <Anweisungsfolge>              // Anweisungsfolge in { ... }
}
else {
  <Anweisungsfolge>              // Anweisungsfolge in { ... }
}
```

Beispiele für Fallunterscheidungen

- Schleichwegfahrt: Pirckheimerstr. Fahrt Richtung Osten, wenn Tageszeit zwischen 7 und 19 Uhr und Fußgängerampel auf rot, dann fahre Schleichweg, sonst fahre geradeaus.



- Abbruchbedingungen für Eingaben: noch eine Diskette formatieren ?
- Entscheidungen über Variablenwerte
  - Steuerung technischer Vorgänge: Wenn Vorlauftemp. unter Soll, dann öffne Mischer
  - wenn Vorlauf-Temp. nicht höher als Zimmer-Temp. + 5° und Rücklauf zwischen Vorlauf und Vorlauf - 2°, dann Umwälzpumpe aus
- Übungsaufgabe
  - Schreiben Sie einen Programmabschnitt, der eine eingegebene Variable auf Erreichen eines Grenzwertes überprüft und abhängig vom Prüfergebnis unterschiedliche Meldungen ausgibt

Beispiel: Zahlenklassifizieren

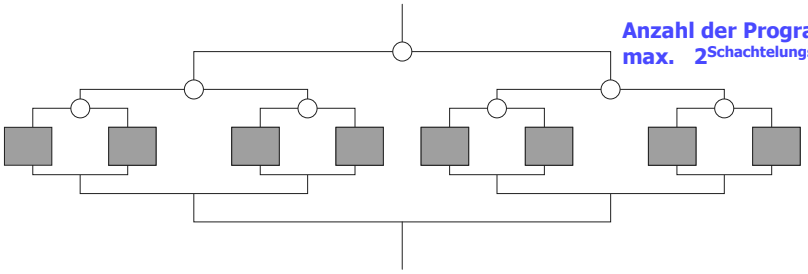
```
void main ()
{
   const int zehn = 10;
         int wert = 0;

   cout << "Gib eine Zahl ein : ";
   cin >> wert;

   if ( wert == zehn ) {          // falls Wert gleich (Vorsicht !!)
      cout << "Der Wert ist zehn" << endl;
   }
   else {                         // andernfalls überprüfe, ob
      if ( wert > zehn ){
         cout << "Größer als zehn" << endl;
      }
      else {                      // falls das auch nicht gilt
         cout << "Kleiner als zehn" << endl;
      }
   }
}
```

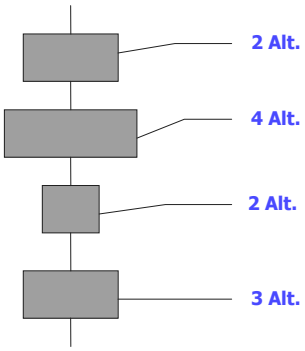
Beobachtungen (bei großer Schachtelungstiefe)

- Große Schachtelung führt schnell zur Unübersichtlichkeit
- Einrückung im Programmtext ermöglicht Überblick
- Strukturierung durch Verteilen auf mehrere Funktionen fördert Übersichtlichkeit
- Falls möglich auf switch-Anweisung (*später*) ausweichen
- Achtung beim Test: im Extremfall Verdoppelung der Programmpfade mit jeder neuen Entscheidungsebene (Schachtelungsebene)



Anzahl der Programmpfade bei aufeinanderfolgenden Verzweigungen

- Hintereinanderausführung von Verzweigungen
  - Anzahl möglicher Pfade =
    Produkt der Alternativen der sequenziell ausgeführten Verzweigungen



- gesamt : 2 * 4 * 2 * 3 = 48 Pfade ,
  in größeren Programmen leicht einige 1000...

Vergleiche und logische Operatoren

- **Vergleichsoperatoren und logische Operatoren liefern als Ergebnis ...**
  - in C++ Wahrheitswerte (Datentyp: bool),
  - in C jedoch einfach einen Zahlenwert, mit entsprechender Interpretation:
    wahr / falsch
    true / false
    ≠0 / 0 (interne Realisierung)
- **Vergleichsoperatoren**

gleich	ungleich	kleiner als	größer als	kleiner o. gleich	größer o. gleich
==	!=	<	>	<=	>=

  - Vergleiche haben geringere Auswertepriorität als arithmetische Operatoren
    `if (i < lim-1)` → `if (i < (lim-1))` // wie erwartet!
- **Besonderheit in C/C++**
  - Jeder Ausdruck kann als Wahrheitswert interpretiert werden
  - Daher: `if (3*x/12)` zulässig; In Abhängigkeit von x wird verzweigt
  - Insbesondere: `if ( x = 12 )` ebenfalls zulässig; Konsequenzen???
  - Wertzuweisung hat niedrigere Priorität als Vergleichsoperatoren!
    `if ((c = x) != 'n')` // Klammerung nötig!

Verwendung von Bedingungen

- **Beispiele**
  - `if (zahl > 10)`
  - `if (zahl)`
  - `if (zahl != 0)`
  - `if (3*x < 0)`
  - `if (x*x > 3*y*z)`
  - `if (fp = fopen("Datei", "r"))`
  - `if (test(x))`
  - `while (*d++ = *s++) ;` (*später*)
  - `for(i=0; i<27; i++)`
- **Empfehlung: auf Lesbarkeit achten!**
  - Bei Vergleichen den "konstanteren" Teil auf die rechte Seite des Vergleichsoperators stellen
  - Aber: Bei Prüfung auf Gleichheit besser die Konstante zuerst nennen:
    `if(0 == x) {…}`
  - Kein unnötiger Gebrauch von Wertzuweisungen im Bedingungsteil
  - Lieber (aufgrund der Auswerteprioritäten) zuviel Klammern setzen als eine zuwenig!

logische Operatoren

- **... verknüpfen Wahrheitswerte zu einem neuen Wahrheitswert**
- **... sind definiert über Ausdrücken, die zu Wahrheitswerten ausgewertet werden können**

Operator		**Schreibweise**	Schema	wahr, wenn	Beispiel
NICHT	/ NOT	`!`	`! a`	a ist falsch	`! (a < b)`
UND	/ AND	`&&`	`a && b`	beide sind wahr	`(a==b)&&(b < c)`
ODER	/ OR	`||`	`a || b`	ein Operand wahr	`(a<b) || (a>b)`

- **Prioritäten**
  - **`!` hat höhere Priorität als die Vergleichsoperatoren**
  - **die Priorität von `&&` ist größer als die von `||`**
  - **beide haben geringere Priorität als die Vergleichsoperatoren!**
- **Wichtige Besonderheit bei C/C++**
  - **`&&` oder `||` - Ausdrücke werden strikt von links nach rechts bewertet**
  - **... und zwar nur solange (!), bis das Resultat der logischen Verknüpfung feststeht!**

Verwendungsbeispiele

- **Bedingungsausdrücke (if-Anweisung , bedingte Schleifen) Verknüpfen von Wahrheitswerten, Variablen und Vergleiche**
  - `if (!( a < 1.0) )`
  - `if (!a)`
  - `if (!test(a))`
  - `if ( (a < 1.0) && (b > 4.7))` **// ausreichend wäre gewesen:**
    `if ( a < 1.0 && b > 4.7 )`
  - `do {...} while ((c=lese()) != 'n' && c != 'y')`
  - **in Wertzuweisungen für Wahrheitswerte:**

```
a = a && (b < c) ;
a = (a == b) || (b == c);
```

  - `for(i=0; i<lim-1 && (c=lese()) != '\n' && c != EOF; ++ i)`
    `s[i] = c;`
  - `if(test() && tue_manches())` **// Achtung: `tue_manches()` wird nur aufgerufen, wenn `test()` true war!**

Fallunterscheidungskaskade

- **Typische Anwendungen**
  - Fallunterscheidung
  - Auswahlentscheidungen - 1 aus n
  - Anweisungskaskade mit berechnetem Einstieg
  - vollständige Mehrfachverzweigung
- **Syntax**
  - `switch ( <Ganzzahlausdruck> )`
    `{`
    `case <Ganzzahlkonstante>: <Anweisung>` // Fall 1
    `....`
    `case <Ganzzahlkonstante>: <Anweisung>` // Fall n
    `default : <Default-Anweisung>` // sonst-Fall
    `}`
  - <Ganzzahlausdruck>: Ausdruck, der zu einem ganzzahligen Wert ausgewertet wird
  - <Ganzzahlkonstante>: ganze Zahl oder Zeichen , keine Ziechenketten o.ä.!
  - <Anweisung>: alle in C/C++ erlaubten, formulierbaren Anweisungen, besonders: break; (Verlassen der switch-Anweisung)
  - <Default-Anweisung>: Sonst-Fall, kann fehlen

Abarbeitung der switch-Anweisung

- **Auswerten des `<Ganzzahlausdruck>`**
- **Vergleich mit `<Ganzzahlkonstante>` der case-Konstante**
- **Bei Gleichheit Ausführen der `<Anweisung>` hinter ":"**
- **Anweisungsausführung ohne / mit `break;`**
  - sequenzielle Bearbeitung aller folgenden `<Anweisungen>` bis zum ersten `break;`
  - oder bis Ende der `switch {...}` Anweisung
- **Besonderheit: `<Anweisung>` bzw. `break` kann auch fehlen:**
  - Zusammenfassen mehrerer Fälle zu gleicher Bearbeitung

```
switch (ausdruck) {
case 1:
case 2:   cout << "Fälle 1 & 2 ...";
          break;
}
```

- **keine Übereinstimmung mit Fall-Konstante**
  - Ausführen der `<Default-Anweisung>`
  - fehlt `<Default-Anweisung>`, Fortsetzung hinter `switch {…}`

Beispiel für Verwendung von switch: Realisierung von Auswahl-Menüs

```
char Cmd;
cin.get(Cmd);

switch (Cmd) {
case 'h':
     cout << "Hilfe:\n";
     cout << "p - PKW eingeben" << endl;
     cout << "m - Motorrad eingeben" << endl;
     cout << "s - Daten zu Kennzeichen abrufen" << endl;
     cout << "d - alle Daten abrufen" << endl;
     cout << "x - Programm verlassen" << endl << endl;
     break;
case 'p':
     // Abfrage der PKW-Daten und Ablegen der Daten …
     break;
case 'm':
     // Abfrage der Motorrad-Daten und Ablegen der Daten …
     break;
   /* ... */
default:
     cout << "Falsche Eingabe!" << endl;
}
```


Schleifen

Zählschleife mit for

- Wiederholung von Anweisungen abhängig von Zählvariable
- typische Anwendungen:
  - Indexlauf, Indizierung von Feldelementen
  - Berechnungen, abhängig von Zählvariable

```
for (<Initialisierungsausdruck> ;   // vor dem Anfang auszuführen
     <Durchlassausdruck> ;          // Bedingung f. einen Durchlauf
     <Schleifennachlaufausdruck> )  // nach jedem Durchlauf
     <Anweisung>; oder {< Anweisungsfolge>}
```

- Besonderheiten / zu beachten
  - Durchlaßausdruck wird vor dem (ersten) Betreten des Schleifenrumpfs überprüft und kann somit zum "Überspringen" des Schleifenrumpfs führen (abweisende Schleife, Kopfschleife)
  - Wert einer globalen Zählvariable ist nach der for-Schleife definiert
  - Die Berechnung innerhalb des Schleifenrumpfs muss der Abbruchbedingung zustreben (Durchlassausdruck = `false`)

Besonderheiten der C for-Schleife (2)

- Komma-Operator ermöglicht Initialisierung mehrerer Variablen bzw. Fortschaltung mehrerer Variablen im Schleifennachlaufausdruck

```
for (j=0,i=1; j<max; i++, j++) ...
```

- Jeder der drei for-Schleifenabschnitte (Init, Durchlassen, Fortschalten) kann leer sein. Ein fehlender Durchlassausdruck wird als 'wahr' angenommen!

```
for ( ; ; )  // Formulierung einer unendlichen Schleife
```

Die Abbruchbedingung muss dann im Schleifenrumpf stehen (`break`, `return`, o.ä.)

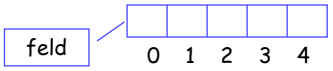
- Der Anweisungsteil kann leer sein (weil schon alles im for-Kopf passiert). Empfehlung: Dies optisch durch eine Zeile nur mit Semikolon verdeutlichen!!!

```
for (i=0; z[i]=q[i]; i++) // z[] und q[] Zeichenketten
    ;
```

Beispiele

```
for (x=10 ; x<=15 ; x=x+1)    // Schleifenkopf
{
   cout << x << " " << x*x << endl;     // Schleifenrumpf
}                                       // { } Hier eigentlich unnötig
```

```
max = feld[0];                // Was wird berechnet ?
for (x=1; x<maxanz ; x=x+1)   //
{
   if (feld[x] > max)         //
      max = feld[x];          //
}
```



(Vorgriff auf Behandlung von Feldern)

Äquivalente Formulierung in Form einer while-Schleife

```
<Initialisierungsausdruck>

while (<Durchlassausdruck>)
{
  <Anweisungen....>
  <Schleifennachlaufausdruck>
}
```

Bedingte Schleifen mit while und do ... while

- **Anwendungen:**
  - bedingte Wiederholung einer Anweisung / Anweisungsfolge
  - `do ... while` besonders für Tastaturabfragen ("... bis Eingabe gültig")

- **Syntax der `while`-Schleife:**

```
while ( <Bedingungsausdruck> )
   <Anweisung>   bzw.   { <Anweisungsfolge> }
```

  - **Semantik**
    - Solange `<Bedingungsausdruck>` wahr, führe `<Anweisung...>` aus.
    - Prüfe vor erstem Durchlauf

- **Syntax der `do ... while` - Schleife:**

```
do
   <Anweisung> bzw. { <Anweisungsfolge> }
while (<Bedingungsausdruck>)
```

  - **Semantik:**
    - Führe den Schleifenrumpf aus, solange der <Bedingungsausdruck> wahr ist
    - Prüfe nach dem (ersten) Durchlauf → mindestens ein Durchlauf!

Bedingungen für alle Schleifen

- **Der <Bedingungsausdruck> muss durch die Operationen im Schleifenrumpf dem Wahrheitswert `false` zustreben**
  - dann: Schleife terminiert
  - sonst: Endlosschleife

- **Als <Bedingungsausdruck> zulässig:**
  - alle arithmetischen Ausdrücke,
  - alle Zeiger-Ausdrücke und
  - alle unzweideutig in solche konvertierbaren Ausdrücke

Unbedingte Fortsetzungen mit goto, continue, break, return

- `goto marke ;`
  - Herkunft: Assembler, Fortran, Basic,...
  - Wirkung: Unbedingte Fortsetzung des Programmlaufs bei der durch Marke angezeigten Stelle innerhalb derselben Funktion
- Anwendung
  - möglicher, aber nicht nötiger Anwendungsfall kann das Verlassen einer tiefen Schachtelung sein, wenn Fehler, z.B. bei Funktionsaufrufen, eingetreten sind.
  - ```
    for (...)
    {  ...
       while (...)
       { ...
            if (schwerer_fehler) goto fehlerbehandlung;
       }
    }
    fehlerbehandlung: fehler_behandlungs_anweisung ;
    ```
- Vereinbarung
  - im Rahmen der LV und Prüfungen Programmieren I u. II verboten!

continue

- Bedeutung
  - Beende aktuellen Durchlauf durch den Schleifenrumpf
  - setze Schleife mit neuem Durchlauf (d. h. ggf. Fortschaltung, Durchlaßprüfung) fort
- Wirkung
  - Abbrechen des aktuellen Schleifendurchlaufs
  - nicht jedoch der ganzen Schleifenanweisung!!!
- Anwendung
  - eher selten
  - Einsparung von if-Schachtelungen innerhalb des Schleifenrumpfs
- Vereinbarung
  - Anwendung in LV und Prüfungen Prog. I u. II bei Abwertung verboten
  - Grund: `continue` verletzt die Blockbildung der strukturierten Programmierung – ein Block besitzt nur einen Eingang und einen Ausgang!
    ```

Beispielprogramm für continue

- // programm **anz_ohne,** Beispiel für die Anwendung von **continue**
  // das Programm zählt die Zeichenvorkommen != dem Zeichen in c (variable)

```
void main (void)
{
    const int maxstr = 20;
    char str [maxstr] = "Maximilian";
    char c = 'i';                       // Vergleichszeichen
    int anzahl = 0, anz_ohne_c = 0;     // index bzw. zählervariable

    while (str[anzahl])                 // solange string-Ende n. erreicht
    {
       if (str[anzahl++] ==c)
          continue;                     // beende Durchlauf falls==c
       anz_ohne_c = anz_ohne_c + 1;     // sonst: erhöhe zähler
    }                                   // enthält Zeichenanzahl ohne c
}
```

- Wie kann man auf dieses `continue` verzichten bzw.es ohne Verlust ersetzen?

```
    if (str[anzahl++] != c)          // erhöhe zähler bei zeichen != c
       anz_ohne_c = anz_ohne_c + 1;  // erhöhe feldindex
```


break

- **Bedeutung / Wirkung:**
  - **Bricht die Bearbeitung des aktuellen umgebenden `switch`-Blocks oder der umgebenden Schleife ab.**
  - **Fortsetzung bei der Anweisung, die dem `switch` bzw. der Schleife folgt.**
- **Anwendung**
  - **durchaus geläufige Verwendung**
  - **Realisierung einer echten Mehrfachverzweigung in `switch`**
  - **Abbruch einer Schleife beim Eintreten von Ablaufbesonderheiten, evtl. von Fehlern**
  - **Einsparung von 'Hilfsvariablen', die im Durchlassausdruck immer wieder abgetestet werden müssen sowie das Verlassen einer möglicherweise tiefen `if`-Schachtelung**
- **Vereinbarung**
  - Verwendung in LV und Prüfungen Prog. I u. II nur in switch-Anweisungen
  - **Grund: analog zu `continue`**

Was wirklich wichtig ist
Teil I

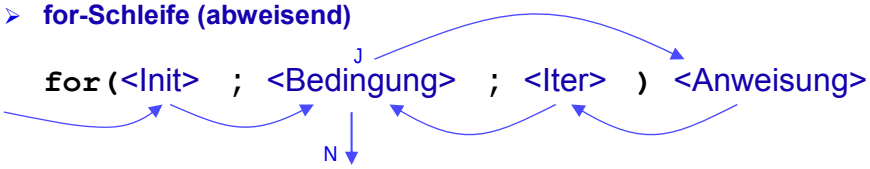
- **Basis-Datentypen:**
  - `[unsigned] char`: 8 Bit (-128…127 bzw. 0…255)
  - `[unsigned] int`: 32 Bit (Umfang ca. 2 Mrd.)
  - `[unsigned] long`: 32 oder 64 Bit (dann Umfang ca. 10^{19})
  - `float`: 32 Bit Fließkomma (Umfang ca. 10^{38})
  - `double`: 64 Bit Fließkomma (Umfang ca. 10^{308})

Expliziter Typecast:
(<typname>)<Variable>
Vorsicht bei impliziten Typecasts!

- **Deklaration von Variablen**
  - Deklaration immer notwendig!
  - <Datentyp> <Name>[=<Initalwert>][,<Name>[=…]][,…];
- **Kontrollstrukturen**
  - `if (<Bedingung>) <Anweisung> [else <Anweisung>]`
  - `switch (<Ganzzahlausdruck>) {`
    `case <Ganzzahlkonstante>: <Anweisungen>`
    `[case…]`
    `[default: <Anweisungen>]`
    `}`
  - `break`: Verlassen der umgebenden Switch-Struktur oder Schleife

Schleifen

- for-Schleife (abweisend)



- while-Schleife (abweisend)

```
while( <Bedingung> ) <Anweisung>
```

- Do-Schleife (zulassend)

```
do <Anweisung> while ( <Bedingung> )
```


Bedingungen

- Jeder (auch jeder arithmetische) Ausdruck kann als Wahrheitswert interpretiert werden; C++ hat eigenen Datentyp `bool`
  - 0: FALSE, ≠0: TRUE
- Vorsicht bei Seiteneffeken (Ausführung von Anweisungen im Bedingungsausdruck)

Logische Operatoren

- In der Reihenfolge der Bindungsstärke: !, &&, || (NICHT, UND, ODER)

Vergleichsoperatoren

- ==, <, >, <=, >=, !=
- Bindungsstärke zwischen ! und &&

Zuweisung

- Bindungsstärke geringer als Vergleichsoperatoren
- Falls Reihenfolge unklar: klammern!

Arithmetik

- Vorsicht bei Fließkomma-Additionen
- Vorsicht bei Integer-Operationen

Standard-Ein-/Ausgabe

- `cout << <Daten> [[ << <Daten>] ... ] ;`
- Alle Standard-Datentypen (auch Zeichenketten) erlaubt
- Zeilenende: `<< endl`
- Formatierung durch Manipulatoren: `setprecision()`, `setw()`, `hex`, `cout.fill()`, ...

- `cin >> <Variable> [[ >> <Variable>] ... ] ;`
- Alle Standard-Datentypen erlaubt
- Entfernung von Space (Leerzeichen, Tabs, Enter) durch `cin >> ws`

Bisherige Struktur eines Programms

```
/* Kommentar */        // oder so
#include <iostream>    // für Ein/Ausgabe
#include <cmath>       // für mathematische Funktionen
using namespace std;

int main(void)
{
  <Variablendefinitionen>

  <Anweisungen>

  return <Rückgabewert>;
}
```

Verbundanweisung: `{ <Anweisung> <Anweisung> ... }`

(5) Funktionen

- Funktionen sind Unterprogramme
  - gängige Sorten : "Funktionen", "Prozeduren", "Subroutines"
  - zusätzlich : "Monitore", "Koroutinen"
- Unterprogramm löst Teilproblem eines Algorithmus
  - Parameter + Algorithmenschema
- Funktionen sind Ergebnisse von Abstraktionsschritten in der Programmentwicklung
  - Erkennen häufig benötigter Anweisungsfolgen,
  - Erkennen logisch eng zusammengehöriger Anweisungen
  - Parameter identifizieren, Algorithmenschema entwickeln
  - Bezeichner zuordnen
- Aufruf mit Namen und Parametern
  - als Teil (Operand) eines Ausdrucks (funktional)
    z.B.: `a=b+c*f(x);`
  - als Ausdruck einer Ausdrucksanweisung (prozedural)
    `f(x);`

Syntax

```
<Fkt-Rückgabetyp> <Fkt-Name> (<typ> <Formaler Par.-Name> [, ...])
{
    // Funktionsrumpf
    // ggf. return Wert vom Typ <Fkts-Rückgabetyp>
}
```

- <Fkt-Rückgabetyp> bezeichnet Typ der Wertrückgabe, des Funktionswerts
- <Fkt-Name> benennt die Funktion
  - In C : <funktionsname> identifiziert Funktion
  - In C++ : Funktionsname+Parameter-Anzahl/Typen identifizieren Funktion
- <Formaler Par.-Name>
  - Platzhalter für Wert, der vom Aufrufer an die Funktion übergeben wird
- Funktionsrumpf
  - Anweisungen zur Berechnung eines Ergebnisses der Funktion, welches dann über eine `return`-Anweisung an den Aufrufer zurückgeliefert wird

return - Anweisung

- Syntax:
  - `return <ausdruck> ;`
  - <ausdruck> vom Typ <Fkt.-Rückgabetyp> oder in diesen konvertierbar
- Bedeutung / Wirkung / Anwendung:
  - Verlassen der aktuell geöffneten "Funktionsschachtel" mit der Möglichkeit zur Rückgabe eines Wertes.
  - Zwingend vorgeschrieben für Wertübergabe an aufrufende Funktion, sonst kann Angabe unterbleiben;
  - Evtl. sinnvoll zur Markierung des Funktionsendes.
- Anmerkungen
  - Typ von <ausdruck> muß gleich sein zu oder konvertierbar sein in Typ der Funktion (s.o.)
  - `return` ohne <ausdruck> nur erlaubt bei Funktionen ohne Rückgabewerte, d.h. vom Typ void.
  - Wichtig bei Wertrückgabe ist tatsächlicher Durchlauf durch return-Anweisung (Unterschied z.B. zu Pascal)

Beispiel: Funktion max zur Berechnung des Maximums zweier int-Werte

```
int max(int a,int b)
{
    int max;
    if (a > b)
       max = a;
    else
       max = b;
    return max;
}
```

```
int max(int a,int b)
{
    if (a > b)
       return a;
    else
       return b;
}
```

```
int max(int a,int b)
{
    return (a > b)? a : b;
}
```


Aufruf-(Aktual)-Parameter:

- **skalare Parameter ➔ Wertübergabe (Parameter "call-by-value")**

```
int  max_2 (int a, int b)        // a, b - formale Parameter
{
   if (a >= b)
      return a;
   else
      return b;
}

int  max_2 (int x, int y)
{
   if (x >= y)
      return x;
   else
      return y;
}
```

Aufruf :

```
u = v + max_2(x,  y) * w;
// x, y aktuelle Parameter
```

Jedes Vorkommen von a, b in der Definition wird bei Aufruf durch Wert von x, y ersetzt

Ausdruck als Aktualparameter

- **Call-by-value ➔ Ausdrücke als Aktualparameter möglich**
  - Ausdruck darf wiederum Funktionsaufrufe enthalten, … , …
  - Mechanismus: Ausdruck auswerten - Wert bestimmen - Formalparameter ersetzen

```
void f(int  a)
{
  cout << "a = " << a << endl;
  // ...
}

void main()
{
  int x = 1;

  f(2*x+1);
}
```

- **Ausgabe**
  - a = 3

Parameterübergabe

- In C standardmäßig Wertparameter (call by value), Ausnahme: Felder
- C++: zusätzlich Adreßparameter (call by reference) → später!

```
void f(int  a)
{
  cout << "a (Orginal)         = " << a << endl;
  a = 5;
  cout << "a (nach Zuweisung) = " << a << endl;
}

void main()
{
  int x = 1;

  f(x);
  cout << "x (nach f())        = " << x << endl;
}
```

- **Ausgabe**
  - a (Orginal) = 1
  - a (nach Zuweisung) = 5
  - x (nach f()) = 1

Der Wert einer Variablen kann durch einen Funktonsaufruf nicht (nach oben bleibend) verändert werden!!!

Wertrückgabe über Parameter

- Über Wertparameter keine Veränderung einer Variablen der rufenden Funktion möglich (außer bei Feldern – später !), da nur Wert übergeben und Adressbezug zur Variablen verloren

- Wertrückgabe über Parameter ➔ Notwendigkeit der Adressübergabe
  - Wohin soll ein Wert gespeichert werden?
  - Speicheradressen von Formal- und Aktualparameter sind unterschiedlich

- Da nur Wertparameter (in C) ➔ Adreßübergabe der Zielvariable als Wertparameter vom Typ

Zeiger auf...
(später)

Kommentare zu den Übungen

- **Aufgabe 4: Schaltjahre**

```
#include <iostream>
using namespace std;

int main(void) {
   int jahr;
   cout << "Jahr eingeben: " ;
   cin >> jahr;
   if((jahr % 4)!=0 ||          // nicht durch 4 teilbar ODER
       (jahr % 100)==0 &&
       (jahr % 400)!=0 )        // durch 100 UND nicht durch 400
        cout << jahr << " ist kein Schaltjahr!" << endl;
   else
        cout << jahr << " ist ein Schaltjahr!" << endl;
   return 0;
}
```

Aufgabe 6: Stunden

Lösung 1: mit if

```
if(stunde==23 || stunde >=0 && stunde <=5)
   cout << "Gute Nacht" << endl;
else if(stunde>=6 && stunde<=10)
        cout << "Guten Morgen" << endl;
     else if(...)...
...
                else
                   cout << "Falsche Stundenangabe!" << endl;
```

Lösung 2: mit switch/case

```
switch(stunde) {
   case 23: case 0: case 1: case 2: case 3: case 4: case 5:
      cout << "Gute Nacht" << endl;
      break;
   case 6: case 7: case 8: case 9: case 10:
      cout << "Guten Morgen" << endl;
      break;
   ...
   default:
      cout << "Falsche Stundenangabe!" << endl;
}
```


Aufgabe 7: Sinustabelle drucken

```
int main() {
   double x;

   for( x=0 ; x <= 1.6 ; x=x+0.1 )
       cout << "sin(" << x << ") = " << sin(x) << endl;

   return 0;
}
```

Ergebnis:

```
sin(0) = 0
sin(0.1) = 0.0998334
sin(0.2) = 0.198669
...
sin(1.4) = 0.98545
sin(1.5) = 0.997495
```

sin(1.6) fehlt!

Aufgabe 7 (Forts.)

- Wo liegt das Problem?
- Test: Ausgabe der Daten auf 17 Stellen Mantissengenauigkeit mittels
  `cout << setprecision(17);`
- Ergebnis:

```
sin(0) = 0
sin(0.10000000000000001) = 0.099833416646828155
sin(0.20000000000000001) = 0.19866933079506122
...
sin(0.69999999999999996) = 0.64421768723769102
...
sin(1.4000000000000001) = 0.98544972998846025
sin(1.5000000000000002) = 0.99749498660405445
```

Rundungsfehler bei der Darstellung von Dezimalbrüchen!

Aufgabe 7 (Forts.)

- Was ist also die Lösung für das Problem?
- Lösung 1: `for(x=0; x<=16; x=x+1) ... sin(x/10) ...`
  - Vorteil: bombensicher
  - Nachteil: nicht universell
- Lösung 2: Zulassen einer kleinen Ungenauigkeit beim Durchlasstest

```
for(x=0 ; x <= 1.6+1.0e-15 ; x=x+0.1) ...
```

  - Vorteil: trägt den Fließkomma-Ungenauigkeiten Rechnung
  - Nachteil: unelegant, Unschärfeschranke schwer festzulegen

```
int f(int n)
{
   int x;
   n=n+1;
   x = 5;
   /* ... */
}

main()
{
   int  n=1, x=5;

   f(n);
}
```

```
int  x; // x - global
int f()
{
   int x, y;    // x lokal zu f
                // globales x verdeckt!
   x = 1; y = 1;
   if (x == y)
   {
      int x;    // x lokal zu {}
      x = 2;
   }
}
```

Frage: Wo ist welches n bzw. x gemeint ???

- Jede Definition einer Variablen (in einer Funktion / in einem Block) erzeugt neue Speicheradresse für Wertablage
- Im gleichen Sichtbarkeitsbereich Variable nur einmal definieren!
- auch in rekursiven Aufrufen (alle 'n' bei der Fakultätsberechnung unterschiedlich!)

- Wo wird dieser Platz angelegt?
  - auf dem Programmstack!

- **Beispiel: Formatierte Ein- bzw. Ausgabe**

```
void formatierte_ausgabe(double x, bool nz) {
   cout << setprecision(10);

   cout << x << " " << x*x << " " << x*x*x ;
   if(nz)
      cout << endl;
}

double eingabe(void) {
   double in;
   cout << "Bitte Wert eingeben: ";
   cin >> in;
   return in;
}

int main(void) {
   double x;
   x = eingabe();
   formatierte_ausgabe(x,true);
   ...
}
```

leere Parameterliste

kein Rückgabewert

- **Formalparameter in der Funktion haben nichts mit den Variablen im "aufrufenden" Teil des Programmes zu tun**
  - **… übedecken aber u.U. globale Variablen und machen sie damit zeitweilig unsichtbar**
  - **Namensgebung ist völlig unabhängig vom Hauptprogramm**
- **Eine Funktion sollte immer eine definierte Schnittstelle zur "Außenwelt" haben, d.h.**
  - **Eingabewerte in Form von Formalparametern (kann auch entfallen)**
  - **einen Rückgabewert (kann auch entfallen)**
  - **Alle anderen Möglichkeiten zur "Kommunikation" mit einer Funktion sollten unterbleiben!**
  - **Ein eventueller Rückgabewert darf beim Aufruf ignoriert werden**
- **In der Funktion deklarierte Variablen ("lokale Variablen") leben nur, solange die Funktion durchlaufen wird und werden beim Verlassen zuerstört**
  - **auch lokale Variablen haben nichts mit den Variablen im Hauptprogramm zu tun**

Algorithmus zur Wurzelberechnung (Newton-Verfahren)

- Berechne iterativ Wurzel einer Zahl z, indem nach einer Nullstelle der Funktion

$$f(x) = x^2 - z$$

gesucht wird

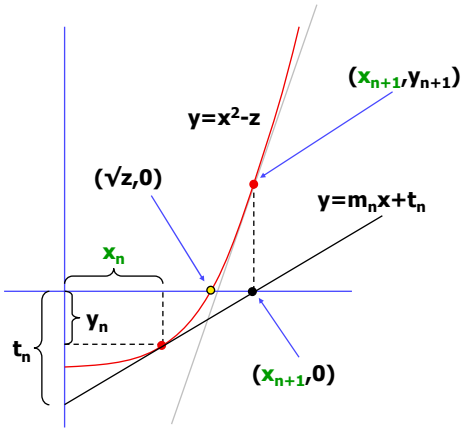
- Iterationsschritt: Sei x_n eine Näherung für das Ergebnis. Dann ergibt sich eine bessere Näherung x_{n+1} durch

$$x_{n+1} = (x_n + z/x_n)/2$$

- Abbruch der Iteration nach Schritt n, wenn $|x_n - z/x_n|$ einen gewissen Schwellenwert k unterschreitet

Aufgabe: Entwickeln eines Programmes, das nach Eingabe des Radikanden und k die Wurzel ausgibt

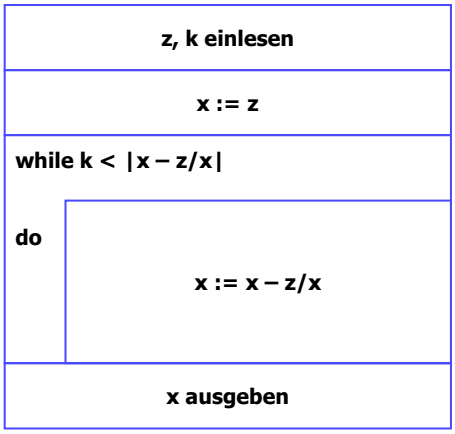
Warum funktioniert das?



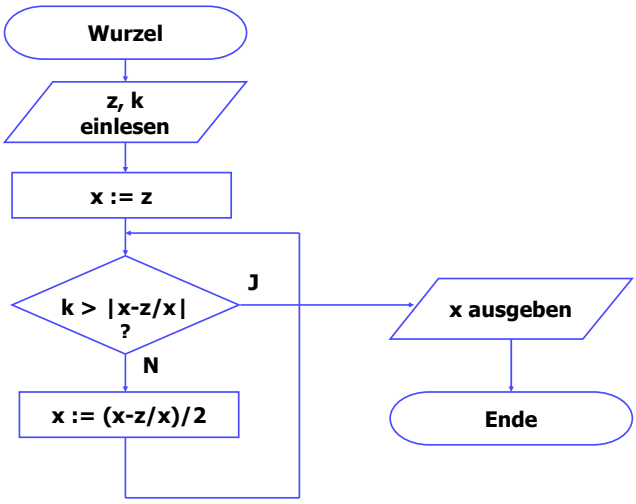
- Schätzwert für Wurzel von z: x_n
- Tangente im Punkt (x_n,y_n) an Parabel legen
- Nullstelle der Tangente ist neuer Schätzwert x_{n+1}

$$x_{n+1} = (x_n + z/x_n)/2$$

NS-Diagramm



Flussdiagramm



Pseudocode

```
z, k einlesen

Setze Startwert: x = z

wiederhole solange |x-z/x| > k

      x = (x - z/x)/2

x ausgeben
```


C++-Code

```
#include <iostream>
#include <cmath>
using namespace std;

int main(void) {
   double z,x,k;

   cout << "Radikand, k: ";
   cin >> z >> k;

   x = z;

   while( k < fabs(x - z/x) ) {
      x = (x + z/x)/2;
   }

   cout << "sqrt(" << z << ") = " << x << endl;
   return 0;
}
```

- **Wurzelfunktion**

```
double wurzel(double z) {
   double x,k;

   x = z;
   k = 0.000000000000001; // k = 1e-15

   while( k < fabs(x - z/x) )
   {
      x = (x + z/x)/2;
   }

   return x;
}
```


- **Endziel: Einpacken des Codes in eine universell verwendbare Wurzelfunktion**
  - **Problem: wie soll k bestimmt werden?**
- **Wichtig: Robustheit!**
  - **Gibt es numerische Fallstricke (Zahlenbereich, Rundungsfehler)?**
  - **Ist der Startwert immer gut?**
  - **Konvergiert das Verfahren immer?**
- **Diese Fragen sind auch heute noch zentral bei der Programmierung von Funktionsbibliotheken!**

- **Funktionen sind unverzichtbare Hilfe beim Strukturieren eines Programmes:**

Version ohne Funktion:

```
int main(void) {
   int a,b,c,d,m1,m2;
   cin >> a,b;

   if(a<b) m1=a; else m1=b;
   …
   if(c>d) m2=d; else m2=c;
   …

}
```

Größter Nachteil der Version ohne Funktion: Bei Änderungen an der Funktionalität muss das Programm an vielen Stellen geändert werden!

Version mit Funktion:

```
int minimum(int x, int y) {
   int res;
   if(x<y) res=x; else res=y;
   return res;
}

int main(void) {
   int a,b,c,d,m1,m2;
   cin >> a,b;

   m1=minimum(a,b);
   …
   m2=minimum(d,c);
   …
}
```

Vorteile: Übersichtlichkeit, Wartbarkeit

- **Funktionen eines Namens dürfen in C++ innerhalb eines Programmes mehrfach vorkommen**
  - **Unterscheidung durch Anzahl und Typ der Formalparameter**
  - **… aber nicht anhand des Rückgabetyps!**

```
void formatierte_ausgabe(double x, bool nz) {
   cout << setprecision(10);

   cout << x << " " << x*x << " " << x*x*x ;
   if(nz)
      cout << endl;
}

void formatierte_ausgabe(int x){
   long r2,r3;
   r2 = x*x; r3 = r2*x;

   cout << x << " " << x*x << " " << x*x*x << endl;
}
```

Polymorphismus bei Funktionen

```
int main(void) {
   double x;
   int i;
   …
   formatierte_ausgabe(x,true);  // Compiler weiß, was zu
   formatierte_ausgabe(i);       // tun ist
   …
}
```

- Compiler kann zur Übersetzungszeit anhand der übergebenen Typen unterscheiden, welche Funktion gemeint ist
- Vor- und Nachteile für Programmierer
  - Nachteil: man weiß u.U. bei einem Funktionsaufruf nicht immer gleich, um welche Funktion es sich handelt
  - Vorteil: ähnliche Funktionalität kann unter gleichem Namen abgerufen werden; keine "Verschmutzung" der Funktionsnamen durch Datentyp-Anhängsel (`minimum_int`, `minimum_double` etc.)

Default-Parameter bei Funktionen

- Nicht immer benötigte Parameter können in C++ Defaultwerte (Standardwerte) bekommen:

```
void formatierte_ausgabe(double x, bool nz=true) {
   cout << setprecision(10);

   cout << x << " " << x*x << " " << x*x*x ;
   if(nz)
      cout << endl;
}

int main(void) {
   double d;
   formatierte_ausgabe(d);
   …
}
```

Implizite Übergabe von `true` als 2. Aktualparameter

Default-Parameter bei Funktionen

- Default-Parameter stehen immer geschlossen am Ende der Parameterliste
- Werden Parameter angegeben, die auch Default-Werte haben können, so stehen diese alle am Anfang der Liste der Default-Parameter

  Beispiel: `void f(double x=0.0, double y=0.0, double z=0.0) {…}`

  `f(1.2,3.0)` ruft `f(1.2,3.0,0.0)`

- Praxis: Default-Parameter werden oft für sehr selten veränderte Argumente benutzt
  - Tipp: keinen übermäßigen und unnötigen Gebrauch davon machen

Verwendung des Debuggers

- Debugger sind unverzichtbare Werkzeuge bei der Fehlersuche in Programmen
  - Programm kann Anweisung für Anweisung abgearbeitet werden
  - Variableninhalte können überprüft und ggf. verändert werden
- Debugger erlaubt das Setzen von Haltepunkten (Breakpoints), an denen die Programmausführung anhält
  - damit muss nicht das gesamte Programm im Einzelschritt durchlaufen werden, um an die interessante Stelle zu kommen
- Debugger erlaubt das Auswerten von Ausdrücken, in denen Programmvariablen vorkommen
- Demo: Untersuchung des Wurzelprogrammes

Rekursion

- Funktionen dürfen wieder weitere Funktionen aufrufen (Aufrufhierarchie)
- Funktionen dürfen auch sich selbst wieder aufrufen; Rekursion!
- Bekanntes Beispiel: Fakultätsfunktion
  - Fakultät(1) := 1
  - Fakultät(n) := n * Fakultät(n-1)
- in C / C++:

```
int fakultaet(int n)
{
    if (n == 1)
        return 1;
    else
        return n * fakultaet(n-1);
}

main()
{
    cout << "Fakultaet(5) = " << fakultaet(5) << endl;
}
```

Warum so kompliziert? Iterative Implementierung:

```
int fakultaet(int n)
{
    int i,p=1;
    for(i=1; i<=n; i++)
       p=p*i;
    return p;
}
```


Rekursive Implementierung

```
int fakultaet(int n)
   {
      return n==1 ? 1 : n * fakultaet(n-1);
   }
```

- Deutlich knappere Formulierung möglich!
- Man kommt mit weniger Hilfsvariablen aus
- Manchmal lässt sich das Problem "verständlicher" programmieren
- ABER:

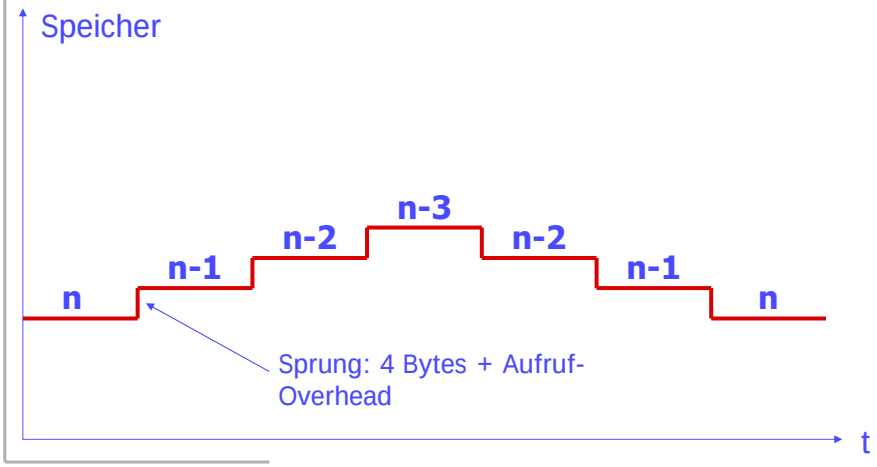
Wissen wir wirklich, was bei einer Rekursion genau abläuft?

- **Wichtigstes Element bei rekursiven Funktionen:**

  Lokalität von Variablen!

- **Durch "Call by Value" werden alle Argumente beim Aufruf kopiert und dadurch zu lokalen Variablen**
- **Auch "direkt" deklarierte lokale Variablen werden bei jedem Aufruf neu angelegt**
- **Folge: Rekursive Algorithmen gehen u.U. nicht besonders sparsam mit dem Speicherplatz um!**
  - **Kann ein großes Problem sein**
  - **Eventuell ist Reformulierung als nicht-rekursiver Algorithmus angebracht**

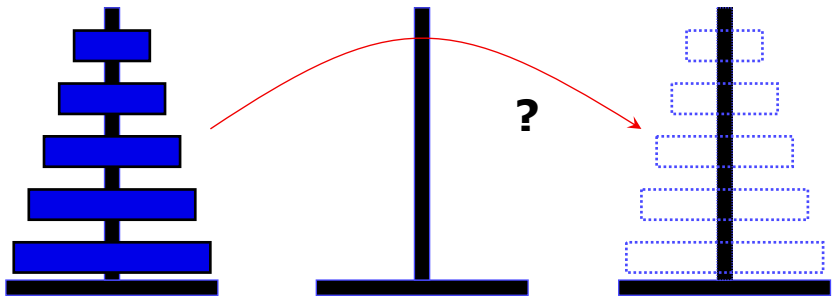
- **Speicherverbrauch über der Zeit bei der rekursiven Fakultätsberechnung (Beispiel n=4)**



Beispiel für Rekursion:

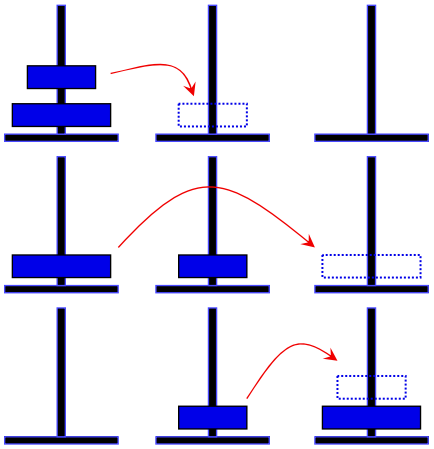
Die Türme von Hanoi

- **Spielregeln:**
  - Gegeben sind drei Stäbe, auf die Scheiben gesteckt werden können
  - Auf einem Stab sitzen N Scheiben, die kleinste zuoberst, nach der Größe geordnet
  - Ziel ist es, den Turm auf einen anderen Stab zu transportieren
  - Es darf pro Zug die oberste Scheibe eines Stapels umgesteckt werden
  - Niemals darf eine größere Scheibe über einer kleineren zu liegen kommen



Wie löst man dieses Problem?

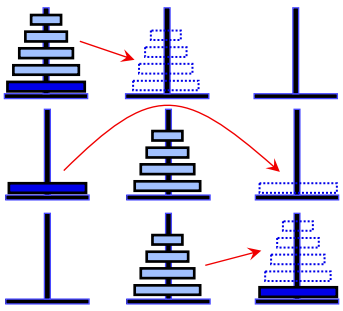
- Wichtigste Beobachtung: Mit 2 Scheiben ist es ganz einfach!



Wenn man das 2-Scheiben-Problem gelöst hat, dann ist das Gesamtproblem ebenfalls gelöst. Warum?

Lösung des Gesamtproblems (N Scheiben):

1. Versetze die N-1 kleinsten Scheiben auf die mittlere Stange
2. Versetze größte der N Scheiben auf die rechte Stange
3. Versetze die N-1 kleinsten Scheiben auf die rechte Stange



Folge:

Problem der Größe N ist rückführbar auf Problem der Größe N-1 (Rekursion)

... und daraus folgt:

Gesamtproblem ist rückführbar auf Problem der Größe N=2

- **Pseudocode:**

```
funktion löse(N, Quellstab, Zielstab)
{
  falls N=2:
      verschiebe(Quellstab, Zwischenstab)
      verschiebe(Quellstab, Zielstab)
      verschiebe(Zwischenstab, Zielstab)
  sonst
      löse(N-1, Quellstab, Zwischenstab)
      verschiebe(Quellstab, Zielstab)
      löse(N-1, Zwischenstab, Zielstab)
}
```

- **Hausaufgabe: Wie funktioniert die iterative Lösung?**

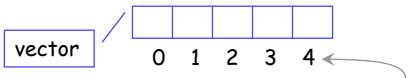
(6) Datenstrukturen I: Felder, Zeiger, Referenzen

- **Beispiele aus der Mathematik**
  - Vektoren (eindimensionale Felder):
    a = (1, 2, 3, 4, 5, 6, 7) u.ä.
  - Matrizen (2-dim. Felder):

$$\begin{pmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{pmatrix}$$

    (Reihen- und Spaltenlauf!)
- **Sonstige Beispiele**
  - Ordner Nr. 10
  - Telefonkurzwahlnummer 5
  - Kontonummer #0815
  - Wartenummer 12
  - Personalstammdatensatz #4711
  - Student mit Matrikelnummer
- **Gemeinsamkeit: Basisdatentyp + Zugriff über (numerischen) Index**
  - Basisdatentyp, z.B. `int`, `double`, `unsigned`, etc., aber auch (*später*) `Konto`, `Ordner`, etc.

- **Beispiel für Vektor:** `int vector [5];`
  - beschreibt homogene Datenstruktur mit 5 Komponenten
    `vector[0]`, `vector[1]`, ... `vector[4]`
  - Zählbeginn immer bei Komponenten-Nr. 0
  - Bezeichner der gesamten "Struktur" ist `vector`
  - Datentyp der Komponenten ist `int`
  - sequenzielle Speicherung



- **mit Vorbesetzung**
  - `int vector[5] = {1, 2, 3, 4, 5} ;`
  - `int vec2[5]  = {1, 2, 3};` // Fehlende Einträge // werden mit 0 initialisiert!
- **mit impliziter Dimensionierung: Dimension abhängig von Vorbesetzung**
  - `int vector[] = {1, 2, 3, 4, 5} ;`
- **Zugriff durch Angabe des Index:** `a = vector[3];` // 4. Element
  - Vorsicht! Es erfolgt keine Überprüfung, ob ein Index im erlaubten Bereich liegt!

"Index"

Beispiel

```
int main(void) {
   int i, anzahl;
   double feld[1000],sum;

   cout << "Anzahl eingeben: ";
   cin >> anzahl;
   for(i=0; i<anzahl; i++) {
      cout << i+1 << ". Element: ";
      cin >> feld[i];
   }

   for(sum=0.0,i=0; i<anzahl; i++)
      sum = sum+feld[i];

   cout << "Summe: " << sum << endl;
   return 0;
}
```

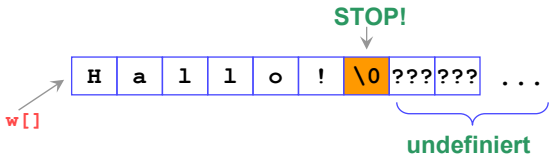
Index = 0…anzahl-1

Zeichenketten sind Vektoren vom Typ `char`

- in C existiert kein dedizierter "string"-Datentyp
- in C++ sehr flexible Klasse '`string`', die vielfältige String-Objekte ermöglicht → später!

Zeichenketten in C / C++ sind null-terminiert!

```
#include <iostream>
using namespace std;
int main ()
{
  char w [20];
  w [0] = 'H' ;
  w [1] = 'a' ;
  w [2] = 'l' ;
  w [3] = 'l' ;
  w [4] = 'o' ;
  w [5] = '!' ;
  w [6] = '\0';
  cout << w;
}
```



Alternativen:

```
char w[20] = {'H','a','l','l','o','!','\0'};
char w[20] = "Hallo!"; // kein '\0' nötig!
```

```
// Endekriterium für Zeichenketten
// Ausgabe des Wortes 'Hallo!'
```

Zeichenketten in C/C++

- **Wichtiges zu Zeichenketten in C/C++**
  - Eine Zeichenkette ist ein Vektor mit Daten vom Typ `char`
  - Bei der Initialisierung gibt es einige Erleichterungen gegenüber Vektoren anderer Typen (`char w[]="Hello!";`)
  - Um das Ende einer Zeichenkette erkennen zu können, wird stets ein `'\0'` an den Text angehängt. Konsequenzen:
    - Eine Zeichenkette muss den im Vektor vorhandenen Platz nicht voll ausfüllen, sondern kann auch kürzer sein
    - Bei der Größe des Vektors muss das Ende-Zeichen berücksichtigt werden!
- **Wichtig (für Felder allgemein): Felder können nicht einfach mit "=" einander zugewiesen werden:**

FALSCH:
```
char w[]="Hallo"; char v[20];
v = w;
```

- **Die Standard-Bibliothek definiert viele Funktionen, die den Umgang mit Zeichenketten erleichtern**
  - Längenmessung, Zusammenfügen, Vergleich, Kopie…

Zeichenkettenfunktionen sind beispielsweise ...

- `#include <cstring>  // früher (C): string.h`
- `strlen()` **- ermittelt die Länge einer Zeichenkette**
  - `a = strlen(w);` // Anzahl der Zeichen vor dem abschließenden Null-Zeichen, jedoch ohne dieses selbst!
- `strcpy()` **- kopiert eine Zeichenkette**
  - `strcpy(ziel_zk,quell_zk);` // "ziel_zk = quell_zk"
- `strcat()` **- hängt eine Zeichenkette an eine bestehende an (Konkatenation)**
  - `strcat(ziel_zk,quell_zk);` // "ziel_zk = ziel_zk + quell_zk"
- `strcmp()` **- vergleicht zwei Zeichenketten lexikografisch**
  - `if (strcmp(zk1,zk2) > 0) ...`
  - `strcmp()` liefert ...

    -1, wenn "zk1 < zk2"

    0, wenn "zk1 == zk2"

    +1, wenn "zk1 > zk2"

Beispielprogramm 'Zeichenkettenoperationen'

```
#include <cstring>
#include <iostream>
using namespace std;

int main(void)
{
  char Nachricht[80];
  strcpy( Nachricht, "Hello world from " );
  strcat( Nachricht, "strcpy " );
  strcat( Nachricht, "and " );
  strcat( Nachricht, "strcat!" );
  cout << "Nachricht = " << Nachricht << endl;
  return 0;
}
```

Output:

```
Nachricht = Hello world from strcpy and strcat!
```


Matrizen in C/C++

- **Beispiel für Matrix (3x4) , d.h. 3 Reihen (Zeilen), 4 Spalten:**

```
double matrix [3][4];
double matrix [3][4] = {1,  2,  3,  4,
                        5,  6,  7,  8,
                        9, 10, 11, 12 };
double matrix [ ][4] = {{1,  2,  3,  4},
                        {5,  6,  7,  8},
                        {9, 10, 11, 12}};
```

- **Gruppierung wesentlich bei automatischer Dimensionierung!**
- **beliebig viele Dimensionen möglich (je nach Ressourcen)**
- **Zugriff auf Feldelemente durch vollständige Indizierung**
  - **erstes Element :** `matrix [0][0]`
  - **letztes Element:** `matrix [2][3]` **// Achtung: Feldindizierung beginnt // bei [0][0] !**
  - **mit Hilfe von Zeigern** . . . . . **(*etwas später*)**

- **Beispiel: Belegen einer Matrix mit Werten (obere Dreiecksmatrix)**

```
double m[20][20]; // 20 Zeilen, 20 Spalten
int zeile,spalte;

for(zeile=0; zeile<20; zeile++)
   for(spalte=0; spalte<20; spalte++)
      if(spalte>=zeile)
         m[zeile][spalte] = zeile*spalte;
      else
         m[zeile][spalte] = 0;
```


- **Felder können als Aktualparameter an Funktionen übergeben werden:**

```
void drucke_feld( int daten[10] ) {
   int i;
   for( i = 0; i < 10; i = i+1 )
    cout << "Feldelement " << i << " = " << daten[i] << endl;
}
```

  - **Bei eindimensionalen Feldern (Vektoren) kann die Größenangabe dabei entfallen:**

```
void drucke_feld( int daten[] ) {…}
```

  - **Bei Matrizen und höherdimensionalen Feldern kann die Größenangabe im ersten Index entfallen:**

```
void drucke_matrix( int daten[][100][200] ) {…}
```

Felder als Aktualparameter

- Für Felder scheint die Regel "Call by Value" nicht zu gelten:

```
void incr_elems(int feld [10])
{
   int  i;
   for ( i = 0; i < 10; i = i+1 )
        feld[i] = feld[i] + 1;
}
```

  - Aufruf mit Feldvariable `a_feld`:

```
int a_feld[10] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
...
for (i = 0; i< 10; i++)
   cout << a_feld[i] << " "; // ---> 0 1 2 3 4 5 6 7 8 9
incr_elems(a_feld);
for (i = 0; i< 10; i++)
   cout << a_feld[i] << " "; // ---> 1 2 3 4 5 6 7 8 9 10
```

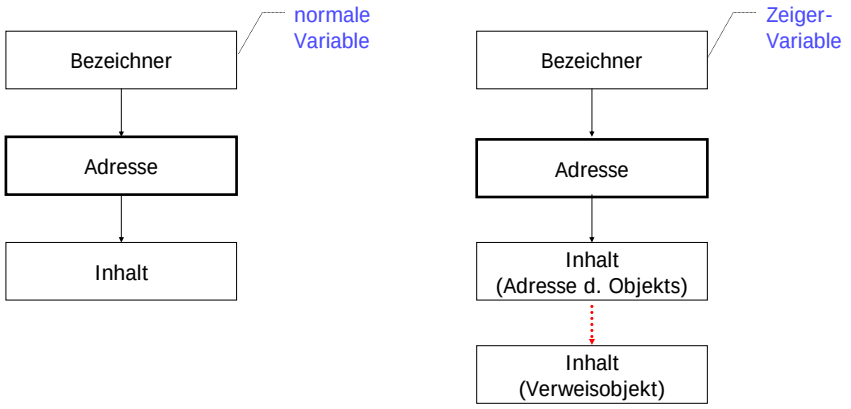
- Grund: Der Name des Feldes (`a_feld`) steht nicht für den Inhalt, sondern die Speicheradresse des ersten Feldelements!

Aufgaben

- Deklarieren Sie einen Vektor mit 20 Gleitpunkt-Komponenten
  - initialisieren Sie die ersten 5 Komponenten bei der Definition
  - weisen Sie den nächsten 5 Komponenten im Programm-Rumpf je einen Wert zu
- Deklarieren Sie eine String-/Zeichenkettenvariable, die 25 Zeichen aufnehmen kann
  - Initialisieren Sie die Variable mit Ihrem Rufnamen
  - konkatenieren Sie Ihren Familiennamen
  - Löschen Sie den ganzen String auf adäquate Weise
- Schreiben Sie ein Programm, das eine 10x10-Matrix mit einem Vektor aus 10 Elementen multipliziert (alle Einträge sind vom Typ `double`)

Besonderheiten der Zeigervariablen:

- **Inhalt (der Wert) der Zeigervariablen ist die Adresse** einer anderen Variablen / eines Objektes



Konventionell mittels Feld und Indexvariable

```
void strcpy(char zk[], const char qk[])
{
   int Ind = 0;        // Die Vararіable 'Ind' hat die Aufgabe eines (Lese-)
                       // Zeigers auf die gerade betrachtete Stelle
   while (qk[Ind] != '\0') {
      zk[Ind] = qk[Ind];// Ein Zeichen von der Quelle zum Ziel kopieren
      Ind ++;           // Lesezeiger eine Position weitersetzen
   }
   zk[Ind] = '\0';     // Zielzeichenkette abschließen
}
```

Bei Verwendung von Zeigern

```
void strcpy(char *zk, const char *qk)
{
   while (*qk != '\0') {  // *-Op = Dereferenzieren ~ Wertbestimmung
      *zk = *qk;          // Ein Zeichen von der Quelle zum Ziel kopieren
      zk++;               // Lesezeiger eine Position weitersetzen
      qk++;               // Lesezeiger eine Position weitersetzen
   }
   *zk = '\0';            // Zielzeichenkette abschließen
}
```

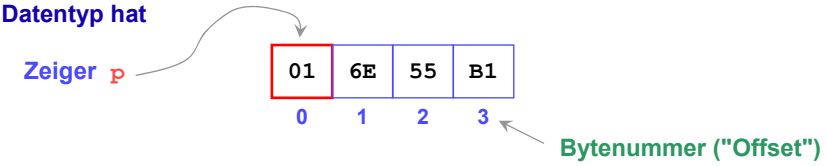
- **Kompaktere Darstellung (Vorgriff!):**

```
void strcpy(char *zk, const char *qk)
{
   while (*zk++ = *qk++)  // Zunächst Wertzuweisung wie oben, dann
       ;                  // beide Zeiger weitersetzen (Post-Inkrement!)
}                         // und zwar solange das Ergebnis der Wertzu-
                          // weisung <> 0 ist
                          // Also wo???
```


- **Einsatzbereiche**
  - Sequenzieller Durchlauf durch Felder (Analogie ~ Lesezeichen)
  - Wertrückgabe bei Funktionen (außer Funktionswert) trotz call-by-value
  - Aufbau und Verwaltung von dynamischen Variablen und Datenstrukturen
  - Hantieren mit Funktionen und Objekten (C++)
- **Deklaration einer Zeigervariablen:**
  - <typangabe> *<bezeichner>;
    Sprechweise: bezeichner ist Zeiger auf Objekt vom Typ <typangabe>
    Bsp.: "`bezeichner` ist Zeiger auf `int`"
- **Beispiel**
  - `int *intpointer;` // einzelner Zeiger auf int
  - `int *ip;` // C-typische Formulierung i-int, p-pointer
  - `char *cp;` // Zeiger auf ein Zeichen
  - `float *floatzeiger;`
  - `int *zei_vec[10];` // Feld von 10 Zeigern auf Integer
  - `double *x, *y, u, v, *w;` // gemischte Deklaration: x,y und w sind
    // double Zeiger,
    // u und v "nur" double Variable

Deklaration von Zeigern

- Ein Zeiger ist nichts weiter als eine Variable, in der eine Speicheradresse abgelegt werden kann
- Für jeden Zeiger muss der Datentyp angegeben werden, auf den er zeigt. Warum?
  - Es muss immer klar sein, wie die Daten, auf die er zeigt, zu interpretieren sind
  - Beim Rechnen mit Zeigern (s.u.) muss bekannt sein, wieviele Bytes der Datentyp hat



Ist p ein…	char*	int*	...
dann zeigt p auf den Wert…	1	$B1556E01_h$	

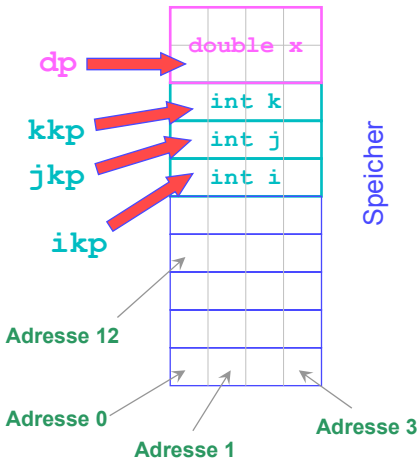
Definition von Zeigern

- Wie bekommt eine Zeigervariable einen sinnvollen Wert?
  - Als Rückgabewert von Funktionen (später!)
  - Durch "Errechnen" der Speicheradresse einer anderen Variablen und Zuweisung an die Zeigervariable. Beispiel:

```
double *dp;
int *ikp, *jkp, *kkp;
```

Nach entsprechender Zuweisung der Adressen (nächste Folie) gilt:

- dp "zeigt auf" x
- kkp "zeigt auf" k
- jkp "zeigt auf" j
- ikp "zeigt auf" i



Operatoren für Zeigervariablen – der Adressoperator

- **&-Operator: "errechnet" die Adresse eine Variablen**
  - Adressoperator - ermittelt die Adresse der Variablen, auf die er angewendet wird

```
int  m = 5;    // Integervariable m mit Wert 5 initialisiert

int  *ip;      // ip ist Pointer auf int, Wert von ip: noch undefiniert

ip = &m;       // Die Adresse der Variable m wird ermitelt und der
               // Variablen ip zugewiesen - ip ist jetzt ein Zeiger auf m
```

  - Es kann beliebig viele Zeiger auf dieselbe Variable geben!

- **Mit Zeigertyp verträglich: Variablenname eines Feldes / einer Funktion**
  -
```
int  val[10]; // Feld mit 10 Integern
int  *ip;     // Zeigervariable
ip = val;     // ip zeigt auf das Feld, d.h. den ersten Eintrag: &val[0]
```
  - Kurz: `int *ip = val;`

Operatoren für Zeigervariablen – der Dereferenzierungsoperator

- *** - Operator:**
  -
```
int  m = 5; // Integervariable m mit Wert 5 initialisiert
int  *ip;
ip = &m;  // ip ist Pointer auf integer, zeigt auf die Variable m
*ip = 10; // Wert an der Adresse - auf die ip zeigt - wird verändert!
          // Was ist nun der Wert von m ???
```

- **Der Wert einer Variablen kann von beliebig vielen Zeigernamen aus geändert werden!**
  - Wenn der Wert von einer Stelle aus geändert wurde, ist die Änderung an allen Stellen sichtbar! Achtung: Sehr große Fehlergefahr!

- **"Dereferenzieren": Verfolgen des Adressverweises**
  - Ermitteln/Verändern des Wertes in der Variablen, auf die der Zeiger verweist.

```
int  i;
i = *ip;  // i bekommt den Inhalt der Zelle, auf die ip verweist, d.h.
          // lies Inhalt von ip, verwende das als Adresse im Speicher,
          // suche die Zelle zu dieser Adresse auf, lies den Inhalt
          // und weise ihn i zu.
```

Zeigerarithmetik: Basis mit Offset

- Zeiger können mehr als nur "auf etwas zeigen" oder das, worauf gezeigt wird, lesen bzw. verändern

- Zugriff auf das n-te Datenelement nach dem, auf das der Zeiger zeigt: "Basiszeiger[Offset]"

```
int k[100];
int *ip = &k[50];
q = ip[4];          // äquivalent: q = k[54];
ip[-10] = 4;        // äquivalent: k[40] = 4;
```

  - also wirkt ein Zeiger wie ein Feldname
  - auch negative Offsets sind erlaubt

  - Mischung der Zeigersyntax mit der Feldsyntax ist möglich:

```
cout << *ip << " " << ip[1];
```


Zeigerarithmetik: Zeiger weiterschalten

- Veränderung eines Zeigers erfolgt durch
  1. Zuweisung einer neuen Adresse: `ip = &i;`
  2. oder durch Inkrement, Dekrement bzw. Addition eines Offset
- Beispiel für die 2. Möglichkeit:

```
int feld[100];
int *zeiger;
...
zeiger = feld;        // Zeiger auf Feldanfang
zeiger++;             // zeigt auf 2. Element
zeiger = zeiger+2;    // zeigt jetzt auf 4. Element
...
int k = *(zeiger+5)   // Zugriff auf 9. Element von feld[]
```

  - d.h. die Addition eines Wertes zu einem Zeiger wird auf besondere Weise interpretiert

  "Zeiger + k" bedeutet eigentlich "Zeiger + k*<Länge Verweisobjekt>"

  Für `double` ist <Länge Verwiesobjekt> z.B. gleich 8, für `int` ist sie 4 etc.

Zeigerarithmetik: Zeiger weiterschalten

- "Zeiger + k"

  bedeutet eigentlich

  "die Adresse, die sich ergibt, wenn man zur Adresse im Zeiger

  <Länge Verweisobjekt> * k

  hinzuaddiert"

  Für `double` ist <Länge Verweisobjekt> z.B. gleich 8, für `int` ist sie 4 etc.

- Zeigerarithmetik bedeutet also
  - Es wird beim Rechnen mit Zeigern (Adressen) immer elementweise, nicht byteweise weitergeschaltet
- Das ist ein weiterer Grund, weshalb Zeiger einen Typ haben müssen!

Zeigerarithmetik: Ausdrücke und void-Zeiger

- Zulässige Ausdrücke und Operationen mit Zeigern
  - `zeiger + <int-Ausdruck>` // Weiter um entsprechend viele Elemente
  - `zeiger - <int-Ausdruck>` // Zurück um entsprechend viele Elemente
  - `zeiger1 = zeiger2;` // Zuweisung von Zeigern deselben Typs
  - `zeiger2 - zeiger1` // Anzahl der dazwischenliegenden Elemente
  - `zeiger1 == zeiger2` // Zeigen beide auf dieselbe Adresse?
  - `zeiger = NULL;` // Synonym zu: "zeigt auf keine gültige // Adresse"
  - Sonst keine weiteren!
- Feldzugriff über Zeiger
  - `zeiger[i]` // entspricht `*(zeiger+i)`, d.h. zeiger wird als // Anfangsadresse des Feldes interpretiert
- Typloser Zeiger: `void *`
  - Wird verwendet, um anzuzeigen, dass der Typ des Elements hier nicht wichtig ist
  - Konsequenz: Nur noch Zuweisungs- und Vergleichsoperationen zulässig, auch nicht mehr Dereferenzierung!
  - `void *` wird häufig auch als Zwischenspeicher für Zeiger verwendet

Beispiele

```
double feld[10], *feld_zeiger, zahl;
...
feld_zeiger = feld;     // Zeiger auf Feldanfang setzen

zahl = *feld_zeiger;    // Zahl aus 1. Feldelement lesen
zahl = zahl + 1;        // Zahl erhöhen
*feld_zeiger = zahl;    // Zahl zurück in Feldelement

feld_zeiger = feld_zeiger +1;   // Feldzeiger auf nächstes
                                // Element
*feld_zeiger = *feld_zeiger +1;// Element um 1 erhöhen
```

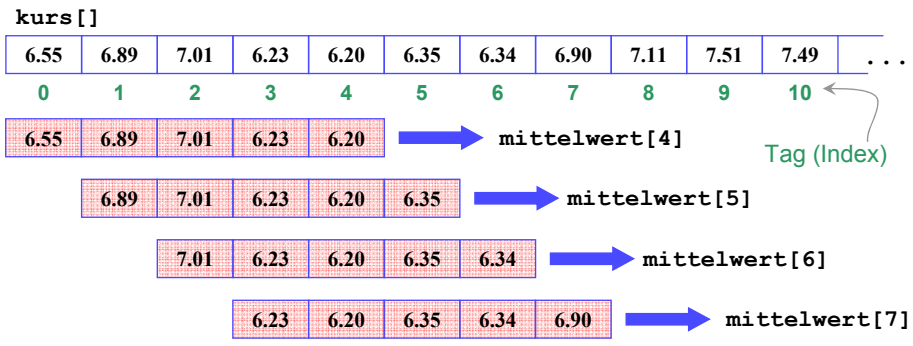
- Achtung:

  `*feld_zeiger++` ungleich `(*feld_zeiger)++`

`++` und `*` haben gleiche Priorität, sind aber rechts-assoziativ!

Beispiele zu Feldern und Zeigern

- **Beispiel: laufendes Mittel über einen Teil eines Vektors (z.B. die letzten 5 Börsenschlusskurse)**



`mittelwert[N]` hält das laufende Mittel über Tag N, N-1, N-2, N-3, N-4

Beispiel: laufendes Mittel (Forts.)

- **Lösung mit Feldindizes:**

```
double kurs[1000], mittelwert[1000], m;
...       // Belegung des Feldes
for(i=4 ; i<1000 ; i++) {
   for(m=0.0, j=i-4 ; j<=i ; j++)
      m = m + kurs[j]/5;
   mittelwert[i] = m;
}
```

- **(eine) Lösung mit Zeiger:**

```
double kurs[1000], mittelwert[1000], *kp, *mp, m;
...       // Belegung des Feldes
for(kp=kurs,mp=mittelwert+4;kp<&kurs[1000];kp++,mp++){
   for(m=0.0,j=0 ; j<5 ; j++)
      m = m + *(kp+j)/5;
   *mp = m;
}
```

Wo soll also der Vorteil von Zeigern sein?

In C gilt grundsätzlich "Call by value":

```
void f(int  a)
{
   cout << "a (Orginal)         = " << a << endl;
   a = 5;
   cout << "a (nach Zuweisung) = " << a << endl;
}
void main()
{
   int x = 1;

   f(x);
   cout << "x (nach f())        = " << x << endl;
}
```

Ausgabe

- `a (Orginal)` = 1
- `a (nach Zuweisung)` = 5
- `x (nach f())` = 1

Der Wert einer Variablen kann durch einen Funktionsaufruf nicht (nach außen bleibend) verändert werden!

Zeiger in Funktionsaufrufen

- **Jetzt: Nicht Übergabe des Wertes, sondern Übergabe seiner Adresse**

```
void f(int *a)  // erwartet Zeiger auf int
{
   cout << "a (Orginal)         = " << *a << endl;
   *a = 5;
   cout << "a (nach Zuweisung) = " << *a << endl;
}

void main()
{
   int x = 1;

   f(&x);  // übergibt Adresse von x
   cout << "x (nach f())        = " << x << endl;
}
```

Immer noch Call by value: Die Adresse wird kopiert!

- **Ausgabe**
  - a (Orginal) = 1
  - a (nach Zuweisung) = 5
  - x (nach f()) = 5

Die übergebene Adresse ist unveränderlich, jedoch kann der Wert unter dieser Adresse (nach außen bleibend) verändert werden!

Beispiel: Maximum-Funktion mit Rückgabeparameter

- **Deklaration**

```
void max(int a, int b, int *ergeb)  // Zeiger auf Ergebnisvar.
{
   if (a > b)
      *ergeb = a;                  // Dereferenzieren
   else                            // und
      *ergeb = b;                  // Speichern
}
```

- **Aufruf u. Verwendung**

```
void main()
{
    int a, b, c, u, v, w;
    ....
    a = 2;
    b = 5;
    max(a, b, &c);                 // Übergabe der Adresse
    ...                            // und
    u = v + c * w;                 // Benutzung des Wertes
}
```

Wirklich nützlich wird das erst, wenn mehr als ein Rückgabewert existiert!

Beispiel: Rekursive Maximum-Funktion

- **Maximum über ein Feld (Länge N) ist gleich**
  - max(feld[0], feld[1]), falls N==2, ansonsten
  - das Maximum vom 1. Element und dem Rest des Feldes (N-1 Elemente)

```
int feld_max(int *feld, int n) {
   int m;
   if( n == 2 )
      if( feld[0] > feld[1] )
         m = feld[0];
      else
         m = feld[1];
   else {
      int restm = feld_max(feld+1, n-1);
      if( feld[0] > restm )
         m = feld[0];
      else
         m = restm;
   }
   return m;
}
```

Selbstaufruf!

Zeigerarithmetik!

Beispiel: Vertauschungsfunktion

- **Vertauschen der Werte zweier double-Variablen**

```
void swap_d(double *a, double *b)
{
        double temp;  /* Hilfsvariable */
        temp=*a;      /* Zugriff über Dereferenzierung */
        *a=*b;        /* … auch schreibend */
        *b=temp;
}

int main()
{
        double x,y;
        …
        swap_d(&x, &y);
        …
}
```

Kopieren der Adressen der Aktualparameter x,y in die Formalparameter a,b

Beispiel: Vertauschungsfunktion

- Problem: Verschiedene Vertauschungsfunktionen für verschiedene Datentypen (double, float, int,…)
- Lösung in C: Übergabe eines Zeigers auf `void` zusammen mit der Länge des Datentyps (s.u.)
- Lösung in C++: Überladen der Funktionen (Übung!)

```
void swap_universal(void *a, void *b, int len)
{
   char temp,*p1,*p2;
   int i;
   p1=(char*)a; p2=(char*)b;
   for(i=0; i<len; i++)
   {
         temp = p1[i];
         p1[i] = p2[i];
         p2[i] = temp;
   }
}
```

Byteweises Kopieren der Daten

Verwendung:

```
swap_universal(&a,&b,sizeof(a));
```

Vorteil: Auch ganze Felder können getauscht werden!

Zeiger auf Zeiger

- Bislang:
  - Verändern von "normalen" Werte-Variablen in Funktionen
  - mit Hilfe von übergebenen Zeigern
- Nun: Übergabeparameter vom Typ "Zeiger auf ..."
  - Zeigervariable bewahrt Wert über Funktionaufruf;
  - Übergabemechanismus auch bei Zeigervariablen call-by-value!
- Verändern von Zeiger-Parametern in Funktionen
  - Gleicher Mechanismus wie bei numerischen Variablen: Übergeben eines Zeigers auf die Variable (d.h. auf die Zeigervariable)
  - Resultat: Der Zugriff auf einen Wert erfolgt doppelt indirekt!
  - z.B. `char **cp;`
- Dereferenzierung im Einzelnen; Beispiel: `int **cp;`
  - `cp` - Zeiger auf einen Zeiger auf int
  - `*cp` - Zeiger auf einen int
  - `**cp` - int

Beispiel: Zeiger auf Zeiger

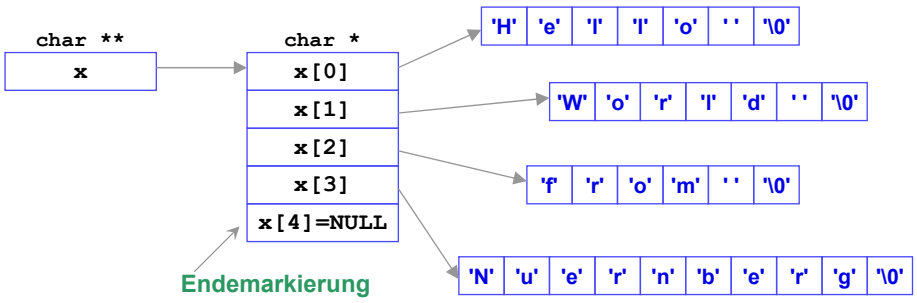
- **Funktion `satz()` setzt einen Satz aus Einzelwörtern zusammen**

```
void satz(char **worte, char *s) {
   *s = '\0'; // definierter Anfang
   while(*worte != NULL) {
      strcat(s, *worte);
      worte++;
   }
}

int main(void) {
   char *x[] = {"Hello ", "World ", "from ",
                "Nuernberg", NULL},erg[1000];
   satz(x,erg);
   cout << erg << endl;
   return 0;
}
```


Beispiel: Zeiger auf Zeiger

- **Funktion `satz()` (Forts.):**



Beispiel für das Verändern von Zeiger-Parametern in Funktionen: Destruktive Variante der Funktion 'Zerlege in Worte'

```
char *ZerlegeInWorte(char *Zeile, char **Wort)
{
   while (*Zeile && !isalnum(*Zeile)) // Nicht-Wortzeichen überspringen
      Zeile++;
   *Wort = Zeile;                     // Wortanfang merken
   if (*Zeile) {
      while (isalnum(*Zeile))         // Wortende suchen
         Zeile++;
      if (*Zeile != '\0')             // Falls Wort-, aber nicht Zeilen-
         *Zeile++ = '\0';             // ende, dann 0-Zeichen setzen
   }
   return Zeile;
}

void main()
{
    char Zeilenpuffer[150], *Wortanfang, *cp;
    cin.getline(Zeilenpuffer,150); cp = Zeilenpuffer;

    while (*cp != '\0') {
       cp = ZerlegeInWorte(cp, &Wortanfang);
          cout << "Wort: " << Wortanfang << endl;
   }
}
```


Zeiger und mehrdimensionale Vektoren - der kleine Unterschied

- **Variablendefinitionen:**
  - `int a[10][10];`
  - `int *b[10];`
- **Zugriff: Möglichkeit der gleichen/gleichartigen Verwendung**
  - `a[5][5] = 5;`
  - `b[5][5] = 5;` // möglich, da `zeiger[i]` ~ `*(zeiger+i)`
- **Jedoch ...**
  - `a` ist wirklich ein 2D-Feld; 100 Elemente sind bereitgestellt worden
  - `b` reserviert nur 10 Zeiger!
  - Falls jeder dieser 10 Zeiger wieder auf Vektoren mit 10 Elementen zeigt, dann wird in diesem Fall Speicherplatz für 100 Elemente plus 10 Zeiger benötigt
  - Der Speicher, auf den die Zeiger bei `b` zeigen, muss noch separat angefordert werden (s. später)!

- **Vorteile des Zeigerfelds (`b`)**
  - **`b` ist Zeiger auf [ein Feld von] Zeiger[n] auf `int`**
  - **Ein Element wird indirekt mit Hilfe eines Zeigers adressiert und nicht mittels Multiplikation und Addition**
    - **ist das im Einzelfall wirklich ein Vorteil?**
  - **Die Zeilen bei `b` können unterschiedlich lang sein; d.h. nicht jedes Element von `b` muss auf einen Vektor von 10 Elementen zeigen; sie dürfen auch kürzer, länger oder leer sein!**

- **Manchmal ist es wünschenswert, nicht nur Daten, sondern auch Funktionen an andere Funktionen zu übergeben. Beispiele:**
  - **Ein Unterprogramm, das eine "schöne" Tabelle von x und Funktionswerten erstellt. Das Unterprogramm erhält die Funktion, die auf x angewendet werden soll, "als Parameter", d.h. Zeiger auf die Funktion**
  - **Ein generisches Sortierunterprogramm. Die varianten Teile sind lediglich das Vergleichen und das evtl. Vertauschen zweier Elemente. Diese Parameter können als Zeiger auf die entsprechenden Funktionen übergeben werden:**

```
void qsort( void *base, int num, int width,
   int ( *compare )( const void *elem1,
   const void *elem2 ) );
```

C-Bibliotheksfunktion!

- **Wie macht man das?**
  - **C/C++ interpretiert den Namen einer Funktion als Zeiger auf dieselbe**
  - **Aufruf kann dann über den Zeiger einfach mittels () erfolgen**

- **Vereinbarung von Zeigern auf Funktionen**
  - **Beispiel:**

```
int ( * comp ) ( void *, void * )
```

comp ist ein Zeiger auf eine Funktion, die zwei `void*` - Argumente hat und ein `int` als Resultat liefert:

"pointer to function (void*,void*) returning int"

- **Achtung: Die Klammern um `*comp` sind zwingend erforderlich!**

```
int *comp (void*, void*)
```

ist eine Funktion , die zwei `void*` - Argumente hat und einen `int`-Zeiger als Resultat liefert

- **Aufruf der Funktion über den Zeiger erfolgt "wie gewohnt": `comp(a,b)`**


```
#include <cmath>
#include <iostream>
using namespace std;

void Tab(double a, double b, double step, double (* fn) (double))
{  // fn ist Funktion mit double als Argument und einem double als Ergebnis
   double x;

   for (x = a; x <= b; x = x + step)
     cout << x << " " << fn(x) << endl;// Verwendung wie gewöhnlich!
}


void main()
{
   Tab(0.0, 6.28, 0.5, sin);              // Aufruf durch einfache Angabe des
   cout << "----------------------\n";    // Funktionsnamens. Verwendung
   Tab(0.0, 6.28, 0.5, cos);              // von '&' nicht erforderlich!
}
```

Typspezifikation

- C ermöglicht sehr vielfältige und weitreichende Typspezifikationen, die präzise zu unterscheiden sind! Beispiele:

```
int  a;                    // Integer-Variable
int  *b;                   // Zeiger auf Integer
int  *c[3];                // Feld von 3 Zeigern auf Integer
int  (*d)[3];              // Zeiger auf ein Feld mit 3 Integer-Elementen
int  *e();                 // Funktion, die einen int-Zeiger als Resultat hat
int  (*d)( );              // Zeiger auf eine Funktion mit int-Resultat
int (*comp) (int [], int [], int);// comp: Zeiger auf Funktion
                           // mit int-Resultat, die als Parameter zwei int-
                           // Felder unbestimmter Größe
                           // und einen weiteren int-Parameter erwartet
char (*(*x())[])();        // x: Funktion mit Resultat Zeiger auf Feld[] mit
                           // Zeiger auf Funktion mit Resultat char
char (*(*y[3])())[5];// y: Feld[3] mit Zeiger auf Funktion mit Resultat
                           // Zeiger auf Feld[5] mit char
```


(6a) Beispiele und Anwendungen

Minimumssuche (Klausur 16.01.1998, Nr. 2)

- Schreiben Sie eine Funktion, die ein gegebenes Feld der Länge 1<n<=100 von Gleitpunktzahlen x_n nach folgender Vorschrift bearbeitet:

  Drucke Tabelle für k=1,2,3,…,n mit folgendem Tabellenelement:

  k x_k y_k

  mit y_k definiert durch: $y_1 = x_1$; $y_2 = \min(x_1,x_2)$; $y_n = \min(x_1,\ldots,x_n)$

  Als Funktionswert soll y_n zurückgegeben werden. Geben Sie alle im Hauptprogramm nötigen Deklarationen an.

- Was wird benötigt?
  - Schleife über alle Elemente des Feldes x
  - Speicher für "aktuell kleinstes" Element

Minimumssuche

```
double tabelle(double x[], int n)
{
   int k;
   double y;

   for(k=0; k<n; k++)
   {
      if(0==k) y=x[k];  // Spezialfall: erstes Element
      else
      {
         if(y > x[k])
            y = x[k];
      }
      cout << k+1 << " " << x[k] << " " << y << endl;
   }
   return y;
}
```

- **Hauptprogramm**

```
#include <iostream>
using namespace std;

int main(void)
{
   double r[20];

   ... // Feld r[] belegen

   cout << "Kleinster Wert: " << tabelle(r, 20) << endl;

   return 0;
}
```


- **Zu definieren ist ein zweidimensionales Feld ganzer Zahlen**

```
int zahlen[20][80];
```

Belegen Sie zur Initialisierung die Feldelemente mit ganzzahligen Zufallswerten aus dem Bereich [0..9]. Verwenden Sie hierzu die Funktion `rand()`, die Zufallszahlen im Bereich [0..`RAND_MAX`] liefert. `RAND_MAX` ist als Konstante in `cstdlib` definiert, genau wie der Prototyp `int rand(void);`

Auszudrucken ist, wie häufig die jeweiligen Zufallswerte – also 0 bis 9 – "gewürfelt" wurden.

Zufallswerte berechnen

```
#include <iostream>
#include <cstdlib>
using namespace std;

int main()
{
   int zahlen[20][80], hist[10], i,j;

   for(i=0; i<10; i++)
      hist[i]=0;

   for(i=0; i<20; i++)
      for(j=0; j<80; j++)
      {
         while((zahlen[i][j]=(double)rand()/RAND_MAX*10.0)==10);
         hist[zahlen[i][j]]++;
      }

   for(i=0; i<10; i++)
      cout << i << hist[i] << endl;

   return 0;
}
```


Doubletten suchen (Klausur 16.01.1998, Nr. 3)

- **Schreiben Sie eine Funktion, die zu einem gegebenen Feld von integer-Zahlen die Anzahl der voneinander verschiedenen Zahlen liefert. Das Feld habe maximal 100 Einträge.**
- **Lösungsmöglichkeiten?**
  1. **"Scannen" des Feldes nach Einträgen, die ab der akt. Position bis zum Ende nicht mehr vorkommen. Zählen dieser Einträge.**
  2. **Sortieren des Feldes in aufsteigender Reihenfolge. Doubletten sind dann "blockweise" angeordnet und können ohne Probleme identifiziert werden.**
- **Welche Elemente werden benötigt?**
  1. **2 verschachtelte Schleifen zum Durchsuchen des Feldes**
  2. **Sortierfunktion qsort mit nachfolgender Elimination der Doubletten**

Lösung 1: Komplexität n^2

```
int unique1(int *f, int num)
{
   int i,j,flag,count=0;

   for(i=0; i<num; i++)
   {
      flag=0;            // Anzeiger für 'kommt nochmal vor'
      for(j=i+1; j<num; j++)
      {
         if(f[i]==f[j]) // falls f[i] nochmal vorkommt...
            flag=1;     // unten (*) nicht zählen
      }
      if(!flag)
         count++;       // (*)
   }
   return count;
}
```


Lösung 2: Komplexität n log n (wegen Quicksort-Algorithmus)

```
int compare(const void *a, const void *b)
{
   int x,y;
   x=*(int*)a;
   y=*(int*)b;
   return x==y ? 0 : x>y ? 1 : -1 ;
}

int unique2(int *f, int num)
{
   int i;

   qsort(f, num, sizeof(int), compare); // sortieren

   for(i=0,count=1; i<num-1; i++)
   {
      if(f[i]!=f[i+1])// 1. Element vom neuen 'Block' zählen
         count++;
   }
   return count;
}
```

Beispiel & Aufgabe: Bubblesort-Algorithmus

- Ziel: Sortieren einer Zahlenreihe
- "Bubblesort": sehr langsamer, aber einfacher Sortieralgorithmus
- Pseudocode:

```
wiederhole für I=N..2
   wiederhole für J=1..I-1
      falls feld(J)>feld(J+1) vertausche(feld(J),feld(J+1))
```

- Beispiel

  3241 2341 2341 2314 2314 2134 1234

- Komplexität des Algorithmus für N Elemente: $O(N^2)$
  - Bessere Verfahren sind deutlich weniger aufwendig
- Aufgabe 1: Implementierung für Integers
- Aufgabe 2: Implementierung für beliebige Datentypen

Anwendung: Kommandozeilenparameter

- Übergabe von Daten an ein Programm konnte bisher immer nur durch Eingaben mit Hilfe von C-Funktionen erfolgen
- C bietet die Möglichkeit, Kommandozeilenparameter an das Programm zu übergeben. Aufruf eines Programmes:

```
programm.exe <arg1> <arg2> <arg3> …
```

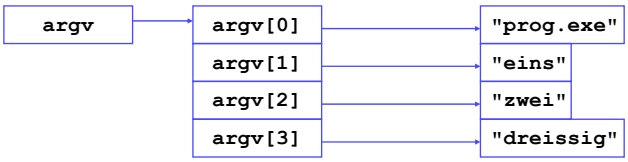
- Parameter werden an das Programm als Zeichenketten übergeben
- Feld von Zeigern auf `char` enthält Zeiger auf die einzelnen Argumente
- Wie funktioniert das?

- **main-Funktion bekommt Parameter:**

```
int main(int argc, char *argv[]);
```

- **argc: Zahl der Parameter**
- **argv: Zeiger auf Feld der Länge argc, das Zeiger auf die Parameter-Zeichenketten enthält**
- **Beispiel: Programmaufruf** `prog.exe eins zwei dreissig`

argc = 4



- **Beispielprogramm**

```
#include <iostream>
using namespace std;

int main(int argc, char *argv[])
{
   int i;

   cout << argc << endl;

   for(i = 0; i < argc; i++)
      cout << argv[i] << endl;

   return 0;
}
```

Ausgabe:

```
4
prog.exe
eins
zwei
dreissig
```

- **Problem: Alle Eingaben liegen als Zeichenketten vor; Konvertierung muss "per Hand" erfolgen**
  - **siehe** `atof()`, `atoi()`

(6b) Referenzen

Referenzen in C++

- Aus C bekannt ist der Pointer: Eine Adresse, die den Ort einer Variablen angibt
  - Variable wird damit "hinten herum" manipulierbar
  - Nachteil: komplizierte Akrobatik mit "*" und "&" Operatoren, spezielle Zeigerarithmetik
  - Pointer funktionieren in C++ wie gewohnt weiter
- C++-Referenz: Zunächst nur ein neuer Name für eine existierende Variable
  - Deklaration mit vorangestelltem "&"
  - Verwendung als Synonym zu referenzierter Variable, Dereferenzierung nicht notwendig
  - Wichtig: Referenz muss gleich bei der Deklaration initialisiert werden!

```
int i=1;
int &k=i;       // k ist Referenz auf i
int &l=k;       // l ist ebenfalls Referenz auf i

++i; ++k; ++l; cout << "i = " << i << endl; // liefert 4
```

- Damit ist die Funktionalität der Referenz auch nicht besser als die eines Zeigers
  - bis auf die einfachere Schreibweise! C-Code:

```
int i=1,*k,**l;
k=&i;
l=&k;

i++; (*k)++; (**l)++; cout << "i = " << i << endl;
```

- Eigentlicher Vorteil von Referenzen: Ermöglichen "Call by Reference" auf komfortable Art und Weise
  - In C musste Call by Reference "explizit" mit Hilfe von Pointern realisiert werden:

```
void swap( double *x, double *y )
  { double t; t=*x; *x=*y; *y=t; }
```


- C++ erlaubt Referenzen als Formal- und Rückgabeparameter:

```
void swap( double &x, double &y )
{
     double t;
     t=x; x=y; y=t;
}

int main(void)
{
     double x,y;
     …
     swap(x,y);      // Compiler weiß, dass swap() Referenzen
                     // erwartet
     …
}
```

- Der Compiler erzeugt automatisch Call by Reference, wenn die Funktion eine Referenz erwartet
- Explizite Übergabe von Zeigern ist damit überflüssig!
- Referenz als Rückgabe erspart die sonst übliche Kopieroperation (siehe später)

- **Vorsicht bei der Rückgabe von Referenzen und Zeigern!**

```
double &skalarprodukt(double x[3], double y[3]) {
   int i;
   double s=0.0;

   for(i=0; i<3; i++)
      s = s + x[i]*y[i];

   return s;
}
```

Referenzen und Zeiger auf *temporäre* Variablen (lokal bzgl. Funktion) sind nicht sinnvoll!

...oder auch:

```
double *skalarprodukt(double x[3], double y[3]) {
   ... // wie oben
   return &s;
}
```


Kommentare zu den Übungen

Effiziente Vertauschung von Feldern (Aufgabe 17)

keine Größenangabe notwendig!

```
void swap(double* &x, double* &y) {
   double *t;
   t=x;
   x=y;
   y=t;
}

int main() {
   double a[1000],b[1000], *ap, *bp;
   ... // Felder belegen
   ap=a; bp=b;

   swap(ap,bp); // Felder ap[], bp[] vertauschen
   ...
}
```

Referenz auf Zeiger auf `double`

nun ist `ap[i]==b[i]` bzw. `a[i]==bp[i]` für alle `i=0…999`

Effiziente Vertauschung von Feldern (Forts.)

- Warum kann man nicht gleich `a` bzw. `b` an `swap()` übergeben?
- `swap(a,b)` erzeugt Warnmeldung des Compilers (hier Borland C++):

```
"Warning: Temporary used for parameter 'x'
        in call to 'swap(double *&, double *&)'"
```

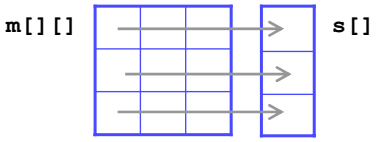
- Grund: Namen von Feldern sind zwar Zeiger, aber insbesondere sind es konstante (unveränderliche) Zeiger. `a` ist also aus der Sicht des Compilers nicht vom Typ `double*`, sondern

```
double * const
```

  - Compiler kann keine Referenzen auf Konstanten übergeben und erzeugt stattdessen eine Kopie
- Folge: Das Programm läuft zwar, es wird aber nichts vertauscht (nur die temporäre Kopie)
- Später mehr zu konstanten Zeigern und Zeigern auf Konstanten…

Aufgabe 18: Matrizenzeilen aufaddieren

```
double *mzadd( double m[10][10], double s[10]) {
   int zeile,spalte;
   double *smin;
   for(zeile=0; zeile<10; zeile++) {
      s[zeile]=0;
      for(spalte=0; spalte<10; spalte++)
         s[zeile] = s[zeile] + m[zeile][spalte];
      if(0==zeile)
         smin=s;
      else
         if(*smin > s[zeile])
            smin=s+zeile; // oder &s[zeile]
   }
   return smin;
}
```



Aufgabe 18: Matrizenzeilen aufaddieren (Forts.)

- andere Möglichkeit der Minimumbildung: eigene Funktion, z.B.

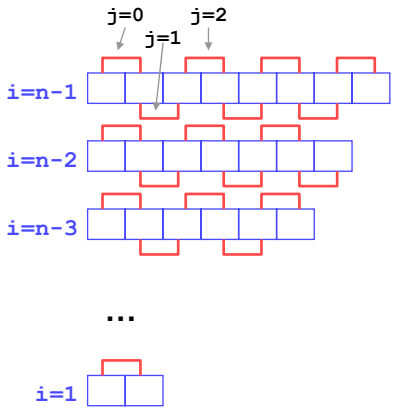
```
double* feld_min(double *feld, int n) {
   double *m;
   if( n == 2 )
      if( feld[0] < feld[1] )
         m = &feld[0];
      else
         m = &feld[1];
   else {
      double *restm = feld_min(feld+1, n-1);
      if( feld[0] < *restm )
         m = &feld[0];
      else
         m = restm;
   }
   return m;
}
```

Bubblesort für `int` (Aufg. 20)

```
void swap(int &a, int &b) {
   int t=a;
   a=b;
   b=t;
}

int bubble(int *f, int n) {
   int i,j,count=0;

   for(i=n-1; i>0; i--)
      for(j=0; j<i; j++)
         if(f[j]>f[j+1]) {
            swap(f[j],f[j+1]);
            count++;
         }
   return count;
}
```



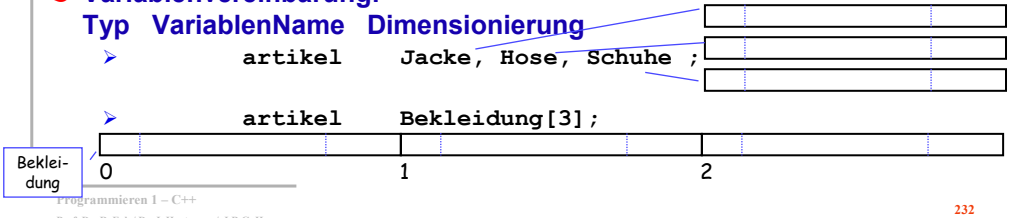
(7) Datenstrukturen II: Strukturen, Klassen, Objekte

- Häufig reicht eine Variable zur vollständigen Beschreibung eines Gegenstands / eines Objekts nicht aus; Beispiele:
  - Rationale Zahl (Zähler, Nenner)
  - Datum (Tag, Monat, Jahr)
  - Anschrift (Straße, Hausnummer, PLZ, Wohnort)
  - Person (Geburtsdatum, Name, Vorname, Geschlecht, Anschrift, Tel., ...)
  - KFZ (Typ {PKW, LKW, Motorrad}, Hubraum, Erstzulassung, ...)
- Problem der Strukturierung von Informationen (Beispiel: Datum)
  - Jeweils eine Variable für jeden Teil des Datums (int Tag, int Monat, int Jahr)?
  - Was, wenn mehrere Daten benötigt werden (Geburtsdatum, Hochzeitsdatum, ...)?
- Sinnvoller:
  - Zusammenhängende Informationen eines Objekts werden auch 'strukturiert' abgelegt
  -
```
struct Datum { int Tag, Monat, Jahr; };  // Typvereinbarung
Datum Geburtsdatum, Hochzeitstag;        // Definition
```
  - Zugriff:
```
Geburtsdatum.Tag = 7;
Hochzeitstag.Jahr = 1992;
```


- Strukturen bezeichnen zusammengesetzte Datenstrukturen
- Die `struct`-Deklaration beschreibt einen abgeleiteten Datentyp, d.h. ein neuer Datentyp wird aus bestehenden Datentypen zusammengesetzt
- Vorteil: Strukturierung und Zusammenfassung

```
struct artikel
{
  int    ArtikelNummer;
  char   ArtikelName [20];
  double Umsatz ;
};
```

- Variablenvereinbarung:
  Typ VariablenName Dimensionierung
  -
```
artikel   Jacke, Hose, Schuhe ;
```
  -
```
artikel   Bekleidung[3];
```



- **Zugriff auf die Struktur-Komponenten : Strukturoperator "."**

```
struct artikel {
   int    ArtikelNummer;
   char   ArtikelName [20];
   double Umsatz ;
} ;
```

```
Hose.ArtikelNummer  = 0815 ;
Hose.Umsatz= 119.95 ;
Hose.ArtikelName[0] = 'b' ;
strcpy(Hose.ArtikelName, "Boss-Jeans");
```

```
artikel Hut = {4711, "Louis Trenker", 19.95};
```

```
Bekleidung[0].ArtikelNummer  = 1234;
strcpy(Bekleidung[0].ArtikelName, "Levis-Jeans");
Bekleidung[0].Umsatz     = 30000.0;
Bekleidung[1].Umsatz     = 10000.0;
Bekleidung[2].Umsatz     = 20000.0;
```

```
Bekleidung[0].ArtikelName[0] = 'l';
```

```
GesamtUmsatz = Bekleidung[0].Umsatz +
Bekleidung[1].Umsatz +
                Bekleidung[2].Umsatz;
```


- **Operationen auf Strukturen**
  - Zuweisung: `Bekleidung[0] = Jacke;`
  - Bestimmung der Adresse: `&Jacke, auch &Jacke.Umsatz`
  - Zugriff auf die Komponenten `Jacke.Umsatz`
  - Initialisierung `artikel Hut = {4711, usw. };`
  - Übergeben einer Struktur an eine Funktion (Call-by-value)
  - Zurückliefern einer Struktur als Returnwert einer Funktion (s.u.)
- **Nicht erlaubt**
  - Vergleichen von zwei Strukturvariablen, auch nicht auf Gleichheit/Ungleichheit
- **Verwendung in Funktionen**

```
struct point { int x,y; };          // Deklaration Struktur point

point makepoint(int x, int y)       // Erzeuge einen 'Point' aus x,y
{
   point temp;                      // Def. lokale Variable
   temp.x = x;                      // Zuweisung an die lokale Variable
   temp.y = y;
   return temp;                     // Rückgabe "by value"!
}
```

- **Beispiel: Rationale Zahl als Strukturdatentyp**

```
struct bruch {
   int z,n;
};

// Produkt zweier Brüche
bruch mult(bruch a, bruch b) {
   bruch t;
   t.z = a.z*b.z;
   t.n = a.n*b.n;
   return t;
}

// Summe zweier Brüche
bruch sum(bruch a, bruch b) {
   bruch t;
   t.n = a.n*b.n;
   t.z = a.z*b.n+a.n*b.z;
   return t;
}
```

- **Anwendung:**

```
int main() {
   bruch p={3,4},q={7,3},r;

   r=mult(p,q);
   r=sum(p,r);
   ...
}
```

- **Nachteil: Bei jedem Funktionsaufruf werden zwei komplette Brüche kopiert**
  - **besser Referenzen übergeben:**

```
bruch mult(bruch &a, bruch &b) {
   bruch t;
   t.z = a.z*b.z;
   t.n = a.n*b.n;
   return t;
}
```


- **Zugriff auf Strukturkomponenten wird durch Zeigerzugriff schnell unübersichtlich; Häufiges Auftreten in Funktionen!**

```
struct typ {
   int     a;
   double  f;
};
void f(typ *abp)        // Übergabe als eigener Datentyp
{
   (*abp).a = 1;        // Klammern sind notwendig , da Priorität von
   (*abp).f = 1.0;      // '.' höher als von '*' !!!
}
```

- **Alternativ, aber kürzer und leichter lesbar:**

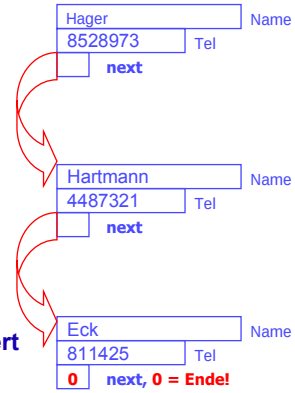
```
void f(typ *abp)
{
   abp->a = 1;
   abp->f = 1.0f;
}
```

- **Achtung: '->' hat höchste Priorität!**
  `++p->len` erhöht z.B. `len`, nicht `p`! Auch möglich: `p++->len`

Rekursive Strukturen

- **Anwendungsgebiete: lineare Liste, binärer Baum, ...**
- **Aufbau: Ein/mehrere Komponenten einer Struktur verweisen (Zeiger!) auf etwas, was den gleichen Aufbau wie die Struktur selbst hat**

```
struct Adresse {
    char Name[20];
    char Tel[20];
    Adresse *next;
};
```



- **Regeln ...**
  - **Eine Struktur darf sich nicht selbst enthalten**
  - **Aber eine Struktur darf einen Zeiger auf ein Objekt vom selben Typ besitzen!**
  - **Üblich: Ende der Kette durch NULL-Zeiger markiert**
- **Anwendungsbeispiel:**
  - **etwas später, Abschnitt dynamische Speicherverwaltung**

Sich gegenseitig referenzierende Strukturen

- **Anwendungsbeispiel**
  - **Beispiel: Liste von Mitarbeitern, jeder: Verweis auf Vorgesetzten**
  - **Vorgesetzter: Liste seiner Mitarbeiter durch Verweis auf Mitarbeiter**
- **Realisierung**

```
struct s;                 // unvollständige Strukturdeklaration

struct t {
   int i;
   s *p;                  // Zeiger auf unvollständige Strukturdeklaration ist zulässig!
};

struct s {
   int j;
   t *q;                  // hier ist struct t bereits vollständig bekannt
};
```

- **Beachte ...**
  - **Dies heißt nicht, dass zwei Variablen `sv` bzw. `tv` vom Typ `s` bzw. `t` zwangsläufig gegenseitig verkettet sind, sondern nur, dass `sv.q` auf *ein* Objekt vom Typ `t` zeigt bzw. `tv.p` auf *ein* Objekt vom Typ `s` zeigt**

- **Prototyp-Strukturen für kommende Beispiele:**

```
struct person {
      char name[20]
      int alter;
};
```


- **Feld von Strukturen kann mittels allgemeiner Sortierfunktion sortiert werden**

```
person m[100];

...       // Daten einlesen

bbsort(m, 100, sizeof(person), comp);
```

- **Was genau tut bbsort?**
  - **Argumente:**

```
int bbsort(void *f, int n, int sz,
           bool (*compare)(void* a,void* b));
```

  - **Funktion compare() vergleicht zwei Feldelemente a,b und gibt zurück:**

`true` falls a>b
`false` sonst.

- Vergleichsfunktion: benutzt `stricmp(char *s1, char *s2)`
- `stricmp()` liefert ohne Berücksichtigung der Groß-/Kleinschreibung

0	falls	beide Strings gleich sind
>0	falls	s1 lexikalisch hinter s2 steht
<0	falls	s2 lexikalisch hinter s1 steht

```
#include <string.h>

bool comp(person *a, person *b)
{
  int r = stricmp(a->name, b->name);
  if(r > 0) return true; else return false;
}
```

- Analog lässt sich nach jedem beliebigen Kriterium sortieren

- Universeller Bubblesort (kein Bezug zu speziellem Datantyp):

```
int bbsort(void *f, int n, int sz, bool (*compare)(void*, void*))
{
   int i,j,count=0;
   char *p,*q;

   for(i=n-1; i>0; i--)
      for(j=0; j<i; j++) {
         p=(char*)f+sz*j;
         q=p+sz;
         if(compare(p,q)) {
            swap_universal(p,q,sz);
            count++;
         }
      }
   return count;
}
```

Berechnung der Adressen der gerade zu vergleichenden Elemente

- **Anwendung des universellen Bubblesort:**

```
bool comp(person *a, person *b) {
   int r = stricmp(a->name, b->name);
   if(r > 0) return true; else return false;
}

void main() {
   person leute[3]= {{"Mueller",23},
                     {"Arends",45},
                     {"Albert",34}};

   bbsort(leute,
          3,
          sizeof(person),
          (bool (*)(void*,void*))comp
          );
   ...
}
```

Einziger Bezug auf eigentlichen Datentyp!

explizite Typkonversion

- **In C sind Strukturen nur "abgeleitete Datentypen" und dienen der besseren Datenorganisation**
- **In C++ können Strukturen benutzt werden, um Daten zusammen mit den Funktionen, die auf sie wirken, zu kapseln**
  - **Eine Struktur kann auch Funktionen (sog. Methoden) enthalten!**
- **Beispiel:**

```
struct bruch {
   int z,n;
   bruch mult(bruch&);
};

bruch bruch::mult(bruch &a) {
   bruch t;
   t.z = a.z * z;
   t.n = a.n * n;
   return t;
}
```

Deklaration der Methode `mult()` der Struktur `bruch`

Definition (Implementierung) der Methode

Welches z und n ist das?

- Innerhalb von Methoden kann auf "eigene" Daten einfach mittels des Namens zugegriffen werden
- Zugriff auf Methoden einer Struktur erfolgt ganz analog wie bei Datenelementen mit "." bzw. "->"
- Beispiel (Forts.):

```
int main() {
   bruch a,b,c;
   ...
   c = a.mult(b);
   ...
}
```

`z` und `n` in `a.mult()` sind identisch mit `a.z` bzw. `a.n`

- Nachteil der Struktur: Immer noch kann "jeder" auf alle Daten und Methoden de Struktur zugreifen
  - C++ erlaubt hier eine Differenzierung

Was wirklich wichtig ist, Teil 2

- Wichtigstes Element der Programmstrukturierung: Funktionen

```
<Rückgabetyp> Funktionsname ( <typ> Formaler Par.-Name [, ...] )
{
  // Funktionsrumpf
  // ggf. return-Wert vom Typ <Rückgabetyp>
}
```

- Funktionsaufruf

```
[ variable = ] Funktionsname( Aktualparameter [, …] )
```

- Rückgabewert muss nicht notwendigerweise ausgewertet werden
- In C wird "Call by Value" verwendet, d.h. es werden Kopien der Aktualparameter an die Funktion übergeben!
- Auch der Rückgabewert ist eine Kopie

- Ausdrücke als Aktualparameter sind erlaubt
- Vorsicht bei Seiteneffekten!
  - `if( f() && g() ) …`
- In der Funktion deklarierte Variablen sind dort lokal, d.h. nur in der Funktion sichtbar
  - Dazu zählen auch die Formalparameter
  - Besonderheit: "static"-Variablen behalten ihre Werte von Aufruf zu Aufruf (s. später)
- Lokale Variablen können globale Variablen gleichen Namens (außerhalb aller Funktionen deklariert) temporär überdecken

Was wirklich wichtig ist, Teil 2

- Zeiger sind Variablen, die Speicheradressen von Objekten eines bestimmten Typs enthalten
- Zeigerdeklaration:

```
<Typ> * zeigername;
```

- Zeigerdeklaration beinhaltet nicht die Reservierung von Speicherplatz für ein Objekt des referenzierten Typs!
- Adresse eines Objekts errechnen: Adress-Operator ("Adresse von")

```
zeiger = &<Objekt>;
```

- Objekt über einen Zeiger ansprechen, der auf das Objekt zeigt: Dereferenzierungsoperator

```
*zeiger = <Wert>;  // oder auch
<variable> = *zeiger;
```


Was wirklich wichtig ist, Teil 2

- Zeiger sind typbezogen, d.h. sie zeigen immer auf Daten eines bestimmten Typs (außer void-Zeiger)
- Zeigerarithmetik
  - Beim Rechnen mit Zeigern ist die Basiseinheit nicht ein Byte, sondern die Länge des Typs, auf den der Zeiger zeigt: `sizeof(<typ>)`
  - Beispiele

```
int *zeiger = &a;
zeiger=zeiger+1;
x = *(zeiger+5);
```

- Alternative Form von `*(zeiger+5)` ist `zeiger[5]`
- Zeiger sind die einzige Möglichkeit, in C "Call by Reference" zu "simulieren" (in C++ gibt es noch Referenzen, s.u.)
  - An Funktionen können Zeiger auf Objekte übergeben werden
  - Manipulation von Daten der aufrufenden Funktion ist in C nur so möglich

Was wirklich wichtig ist, Teil 2

- **Referenzen in C++**
  - sind eine Möglichkeit, einen weiteren Namen (eine "Referenz") für eine Variable zu etablieren
  - Referenzen müssen gleich bei der Deklaration initialisiert werden:

```
int i;
…
int &k = i;// k ist Referenz auf i
```

  - Für alle praktischen Belange ist die Referenz mit der Variable identisch
  - Referenzieren kann man alles, auch Zeiger, Strukturen, Klassen etc.
- **Wichtigste Anwendung: Call by Reference!**
  - Übergabe einer Referenz an eine Funktion macht die referenzierte Variable in der Funktion manipulierbar

```
void f(int &r) {
   r=5;      // es wird tatsächlich die übergebene Variable manipuliert!
}
```

- **Referenzen sind ein wichtiges Mittel effizienter Programmierung in C++ (Vermeidung unnötiger Kopien bei der Datenübergabe)**

Was wirklich wichtig ist, Teil 2

- **Felder sind n-dimensionale Aneinanderreihungen eines Datentyps**
- **Felddeklaration**

  1D: `<Typ> Feldname[<Länge>]`
  2D: `<Typ> Feldname[<Länge1>][<Länge2>]`
  …
- **Zugriff auf Feldelement: `Feldname[<Index>]...`**
- **Wichtig: Felder sind in C von 0 beginnend indiziert!**
  - `int feld[100]` hat genau 100 Elemente, von `feld[0]` bis `feld[99]`
- **Kein Check auf Bereichsüberschreitung!**
- **Wichtiger Spezialfall: `char`-Felder**
  - Konvention: Ein char-String wird mit einem `'\0'` abgeschlossen
  - Initialisierung von char-Feldern: `char s[]="Hello"; // s hat Länge 6`
  - Alle sonstigen Operationen auf Strings werden über geeignete Funktionen erledigt (`strcmp`, `strcpy` etc.)
- **Feldname ist synonym zu Zeiger auf Feldanfang: `feld == &feld[0]`**
  - einziger Unterschied: Feldname ist konstanter Zeiger!

Was wirklich wichtig ist, Teil 2

- **Strukturen** sind abgeleitete Datentypen, die aus einfacheren Typen zusammengesetzt sind
- Strukturdeklaration

```
struct name {
  <Typ> element1; ...
};
```

- Deklaration eines Objektes von Typ `name`

```
name Objekt;
```

- Erlaubt sind Felder von Strukturen, Zeiger auf Strukturen, etc.
- Strukturen werden "by value" an Funktionen übergeben (und auch zurückgegeben)

Was wirklich wichtig ist, Teil 2

- Zugriff auf Strukturelemente mit dem "."-Operator

```
struktur.element
```

bei Strukturen als Elemente:

```
struktur.elementstruktur.element
```

- Nur wenige erlaubte Operationen auf ganze Strukturen möglich (z.B. Zuweisung, Adresse), z.B. kein Vergleich!
- Zugriff auf Strukturen über Zeiger:

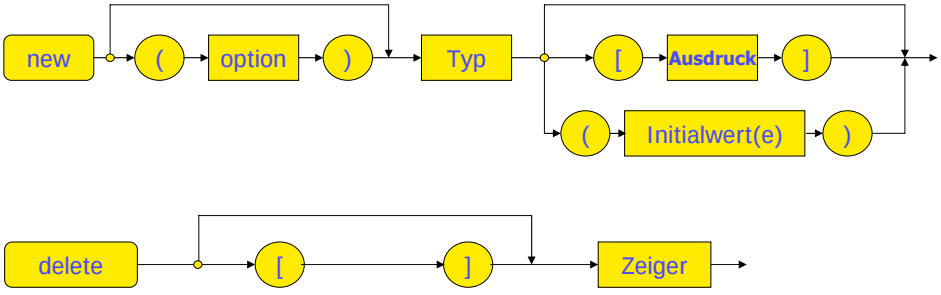
```
(*struktur).element   identisch mit
struktur->element
```

- In C++ können Strukturen auch Funktionen enthalten ("Memberfunktionen", "Methoden")
  - "Kapselung" von Daten mit den Funktionen, die mit ihnen arbeiten
  - Einstieg in die Objektorientierung

(7a) Dynamische Speicherverwaltung

Dynamische Speicherverwaltung

- In C kann mittels `malloc()` und `free()` zur Programmlaufzeit Speicher vom Betriebssystem angefordert und freigegeben werden (wird hier nicht behandelt)
- In C++ gibt es für diesen Zweck die neuen Operatoren `new` und `delete`



- Der Operator `new` kann
  - einzelne Objekte allokieren (d.h. Speicher für sie reservieren)
  - einzelne Objekte allokieren und mit Startwert versehen
  - Felder von Objekten allokieren
- `new` gibt einen Zeiger auf das/die allokierte/n Objekt/e zurück
- `delete` gibt den durch `new` allokierten Speicher wieder frei
- Beispiele:

```
int *feld = new int[20]; // 20 integers
int *ctr  = new int;     // 1 integer, nicht vorbelegt
int *ctr  = new int(1);  // 1 integer, vorbelegt
...
delete ctr;              // Einzelobjekt freigeben
delete [] feld;          // Feld freigeben
```

- Wichtig: Unterscheidung zwischen Feld- und Objektfreigabe!

- Details zu `new` und `delete`
  - `delete []` gibt Felder frei, `delete` nur Einzelobjekte!
  - Standardverhalten von `new` bei Speichermangel: "Auswerfen" einer Ausnahme
    - Wird die Ausnahme nicht abgefangen, so bricht das Programm ab (siehe später)
    - Option `(nothrow)` bei `new` bewirkt anderes Verhalten (Rückgabe von `NULL` bei Speichermangel)

```
int *feld = new(nothrow) int[20];
if(!feld)
     cerr << "Speichermangel!" << endl;
.....
```

  - `delete` kann ohne Gefahr einen `NULL`-Zeiger übergeben bekommen
  - `delete` auf ein bereits freigegebenes Objekt bzw. auf eine "wilde" Adresse führt zu undefinierten Ergebnissen!

Beispiel: Zeichenkette in Großbuchstaben wandeln

```
char *uppercase(char *s)
{
    char *p,*uc = new char[strlen(s)+1];
    p=uc;
    while(*(p++) = toupper(*(s++)));
    return uc;
}

int main(void)
{
    char *u, *str = "Hello World";
    u = uppercase(str);
    cout << u << endl;
    delete[] u;
    return 0;
}
```


Beispiel: Flexible Allokierung eines Feldes zur Aufnahme von Daten

```
int main(void) {
   double *p;
   int anzahl,i;

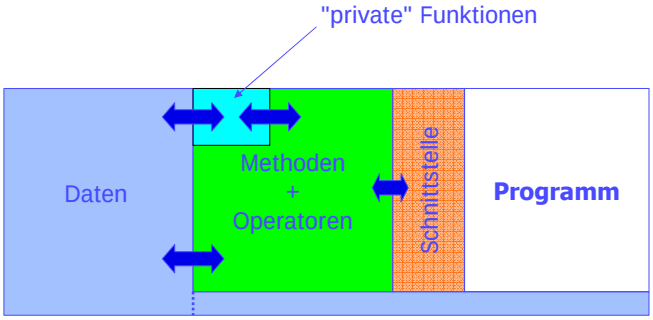
   cout << "Wie viele Werte? ";
   cin >> anzahl;

   p = new double[anzahl];

   cout << "Bitte " << anzahl << " Werte eingeben: ";
   for(i=0; i<anzahl; i++)
      cin >> p[i];
   ...
}
```

Einführung in die Objektorientierung mit C++

- C++ ermöglicht die Definition von "abstrakten Datentypen"
- Daten und Funktionen ("Methoden"), die auf die Daten wirken, werden gemeinsam definiert und zusammengefasst
- Damit definiert sich ein neuer Datentyp nicht mehr (nur) über eine Datenstruktur, sondern über eine Schnittstelle zur Außenwelt
  - Details der Implementierung der Datenstruktur ist vor dem Anwender (idealerweise) verborgen



- **Beispiel: Datentyp "bruch", nicht objektorientiert**
  - **Früher hatten wir definiert:**

```
struct bruch {
   int z,n;
};

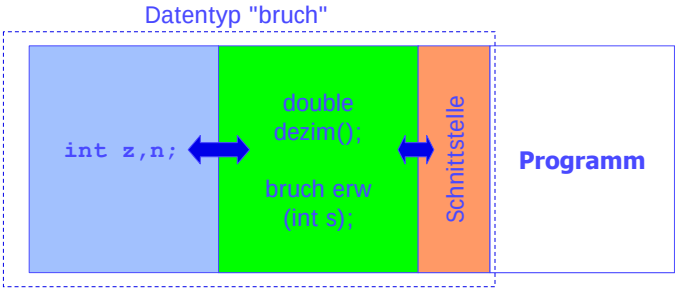
double dezim(bruch &x);
bruch erw(bruch &x, int s);
… // weitere Funktionen
```

Datenstruktur

Funktionen

  - **Funktionen und Daten, auf die sie operieren, sind hier nur "lose" verbunden (semantische Verbindung)**
  - **Daten können auch ohne die zugehörigen Funktionen manipuliert werden**

- **Datentyp "bruch", objektorientiert**
  - **die Daten selbst sind nur durch eine funktionale Schnittstelle zugänglich**
  - **Daten und zugehörige Methoden werden "zusammengeklammert" und sind nicht trennbar (syntaktische Verbindung)**



  - **Wie genau die Daten gespeichert sind, ist nicht wichtig und idealerweise vor dem Benutzer verborgen**

- Wie wird das in C++ realisiert?
- Ausgangspunkt: Struktur (`struct`)
  - in C dienen Strukturen dem Aufbau zusammengesetzer Datentypen
  - in C++ können Strukturen auch Methoden enthalten
  - bevorzugtes Sprachelement in C++: Klasse (`class`)
    - genauer Unterschied zwischen `struct` und `class`: s.u.
  - die Methoden einer Klasse werden auch "member-Funktionen" genannt
  - es gibt spezielle Methoden, die bestimmte Aufgaben wie z.B. die Initialisierung der Daten übernehmen
- Beispiel: Komplexe Zahlen in C++

```
class complex {
private:
    double re,im;
public:
    complex(double r, double i);
    double betrag();
};
```

Datenstruktur und Schnittstelle

- Beispiel: Komplexe Zahlen in C++ (Fortsetzung)

```
complex::complex(double r, double i)
{ re = r; im = i; }

double complex::betrag()
{ return sqrt(re*re + im*im); }

int main(void)
{
    complex z1, z2(1.0,0.0);
    double b;
    ...
    b = z2.betrag();
    ...
}
```

Implementierung

Instanziierung

Zugriff auf Member

Funktioniert das so? Ausprobieren!

- **Was bedeutet das alles?**
  - `private` und `public` geben an, ob die nachfolgenden Deklarationen von nicht-Klassenmitgliedern aus sichtbar sind oder nicht
    - jeder kann `betrag()` und `complex()` verwenden, nicht aber auf die eigentlichen Daten zugreifen
  - `complex()` ist eine spezielle Memberfunktion: ein Konstruktor! Er wird genau dann aufgerufen, wenn eine Variable (ein Objekt) des Klassentyps erschaffen werden muss (Instanziierung)
    - mehrere Konstruktoren sind möglich und sogar üblich (Polymorphismus!)
  - Deklaration der Datenstrukturen und der Schnittstelle ist von der eigentlichen Implementierung üblicherweise getrennt (meist sogar in einem anderen File)
  - die Implementierung der Memberfunktionen erfolgt mit Nennung des Klassennamens gefolgt von "`::`" und dem Funktionsnamen
  - Punktoperator macht die Komponentenauswahl (Daten + Methoden)

- **Details zur Instanziierung**
  - Es wird der Konstruktor aufgerufen, der zum Initialisierungsausdruck bei der Instanziierung passt
  - Es gibt einen Default-Konstruktor, der nichts tut als die Datenelemente des Objektes zu erschaffen (im Beispiel `re` und `im`)
- **Details zur Sichtbarkeit der Klassenmember**
  - Alle Datenstrukturen oder Methoden, die in einer `public`-Sektion stehen, können von nicht-Klassenmembern erreicht werden
    - "erreichen" bedeutet "Zugriff" bei Daten, "Aufruf" bei Methoden
  - Alles unter `private` ist nur für Klassenmember erreichbar
    - Ausnahme: befeundete Funktionen oder Klassen (siehe später)
  - Zugriffsspezifizierer `protected`: Member sind für Tochterklassen `public`, für alle anderen `private` (siehe später)
- `class` und `struct`: der "kleine Unterschied"
  - `class`: Alle Member sind per Default `private`
  - `struct`: Alle Member sind per Default `public`

- Etwas allgemeiner: Das Prinzip der Objektorientierung in C++
  - Eine Klasse wird deklariert (Datenstrukturen + Schnittstellen)
  - Die Klasse wird implementiert (hier wird erstmals Code geschrieben)
    - Deklaration und Implementierung haben üblicherweise keine Reservierung von Speicher zur Folge (Ausnahme: Klassenvariablen, s. später)
  - Die Klasse als Datentyp wird verwendet, indem man sie instanziiert, d.h. ein konkretes Objekt vom Typ der Klasse erzeugt
    - Reservierung von Speicher für die Datenstrukturen
    - Aufruf des Konstruktors, abhängig von der Art und Weise der Instanziierung
  - Das erzeugte Objekt trägt die Methoden der Klasse "in sich"; diese arbeiten dann üblicherweise mit den konkreten Daten dieses Objekts
  - Zugriff auf Methoden bzw. ggf. Daten eines Objektes erfolgt mit
    - dem "`.`"-Operator, falls der Name des Objektes bekannt ist oder eine C++-Referenz darauf existiert
    - dem "`->`"-Operator, falls das Objekt über einen Zeiger referenziert wird

- Was ist übliche Praxis?
  - Möglichst viele Member-Datenstrukturen einer Klasse sind `private`
    - Folge: `struct` wird praktisch nie zur Klassendeklaration genutzt
  - Deklaration enthält praktisch nie auch die Implementierung von Member-Funktionen
    - Ausnahme: "Einzeiler" stehen oft komplett in der Deklaration
    - Nebeneffekt: Solche Methoden sind automatisch `inline`-Funktionen

```
class complex {
      private:
         double re,im;
      public:
         complex(double r, double i);
         double betrag() {
            return sqrt(re*re+im*im);
         }
};
```

automatisch `inline`

  - Deklarationen stehen meist in `.h`-Files, Implementierungen und sonstige Funktionen in `.cc` (`.cpp`, `.c++`, `.cxx`)-Files
    - folgend der üblichen Praxis in C

Inline-Funktionen

- In C konnten Präprozessor-Makros verwendet werden, um oft gebrauchte kleine Codestücke zu definieren:

```
#define max(A,B) ( (A) > (B) ? (A) : (B) )
```

- Vorteil: Ein Hauch von generischer Programmierung (obiges geht für alle Datentypen, die der ">" Operator verarbeiten kann)
- Nachteil: Einfaches suchen-ersetzen-Schema, keine Typprüfung, Seiteneffekte
- C++ kennt das Schlüsselwort `inline` bei einer Funktions- oder Methodendefinition

```
inline int max(int a, int b)
{ return a>b ? a:b; }
```

- Compiler versucht dann (hoffentlich), die Funktion bei Benutzung direkt in den Code zu schreiben
- `inline` ist nur ein "Hint", der vom Compiler ignoriert werden darf

Eine etwas vollständigere `complex`-Klasse: Deklaration

```
class complex {
   private:
      double re,im;
   public:
      void init(double,double);
      complex();                // Default-Konstruktor
      complex(double r,double i)
         { init(r,i); }
      double betrag();
      double real() {return re;}
      double imag() {return im;}
      void print();             // huebsch drucken
      void add(complex &);      // Arithmetik
};
```

Eine etwas vollständigere `complex`-Klasse: Implementierung

```
void complex::init(double r, double i)
{
   re=r; im=i;
}

complex::complex()
{
   init(0.,0.);
}

void complex::print()
{
   cout << "(" << re << "+" << im << "i)";
}

double complex::betrag()
{
   return sqrt(re*re + im*im);
}

void complex::add(complex &s)
{
   re += s.real(); im += s.imag();
}
```


Eine etwas vollständigere `complex`-Klasse: Verwendung

```
int main()
{
   complex z1;
   complex z2(1.0,2.0);

   z1.init(2.0,2.0);

   cout << "z1="; z1.print(); cout << "\nz2="; z2.print();

   cout << "\nBetraege: |z1|=" << z1.betrag();
   cout << ", |z2|=" << z2.betrag() << endl;

   cout << "z1+z2=";
   z1.add(z2); z1.print();

   return 0;
}
```

- Kommentare zum Programm
  - Konstruktoren greifen auf "gemeinsame" Methode `init(double, double)` zurück
  - Konstruktor `complex(double, double)` wird bereits bei der Deklaration implementiert, genauso wie `real()` und `imag()`
  - Referenz im Argument von `add(complex &)`
    - verhindert Kopie des Objektes nur zum Zwecke der Addition
  - Das Ganze ist noch etwas holprig
    - Addition sollte mit "+" geschehen, Ausdrucken mit "<<"
  - Realteil und Imaginärteil könnte man einfacher auch durch Freigabe der Daten mittels `public` gewinnen
    - das ist aber spezifisch für diesen speziellen Datentyp

- Konstruktoren sind spezielle Member-Funktionen
- Jede Klasse hat mindestens einen Konstruktor (Default-K.)
  - mehrere sind möglich (Überladung!)
  - wenn mindestens ein Konstruktor deklariert ist, gibt es keinen Default-Konstruktor
- Jeder Konstruktor hat den gleichen Namen wie die Klasse
- Ein Konstruktor gibt nie etwas zurück (auch nicht `void`!)
- Bei der Instanziierung eines Objekts wird der Konstruktor aufgerufen, dessen Signatur zum Initialisierungsausdruck passt
  - oder der Default-Konstruktor, wenn es keine Initialisierung gibt

```
complex z1;      // Default-Konstruktor
complex z2(1,0); // complex::complex(double,double)
```

  - es gibt keinen direkten Aufruf eines Konstruktors in einem Objekt

```
z1.complex(1,0); // FALSCH!
```

- Konstruktoren können `public`, `private` oder `protected` sein
- "Ansonsten" ist ein Konstruktor eine "normale" Member-Funktion

- In der `complex`-Klasse hat der Konstruktor nur Werte zugewiesen; er kann aber mehr:

```
class iarray {
       private:
          int *start;
       public:
          iarray(int laenge);
          int element(int);
};

iarray::iarray(int laenge) {
       start = new int[laenge];
}

int iarray::element(int i) {
       return start[i];
}
```

Verwendung:

```
iarray x(100);
…
int i = x.element(59);
```

- Frage: Wer gibt in diesem Fall den Speicher wieder frei?
  - Antwort: der Destruktor!

- Destruktor wird automatisch aufgerufen, wenn das Objekt aufhört zu existieren
  - z.B. lokale Objekte in Funktionen oder mit `new` allokierte Objekte, wenn sie mit `delete` zerstört werden
- Es gibt keinen Default-Destruktor
- Name des Destruktors: `~<Klassenname>`
- Ein Destruktor gibt nie etwas zurück (auch nicht `void`!) und hat keine Parameter
  - keine Überladung möglich
- Häufige Anwendung: Freigabe von dynamisch allokiertem Speicher

```
class iarray {
      private:
         int *start;
      public:
         iarray(int laenge);
         ~iarray();
         int element(int);
};

iarray::~iarray() {
      delete[] start;
}

...
```


- **Verwendung**

```
void f()
{
      iarray x(100);
…
      d = x.element(45);
…
      return;
}
```

Instanziierung, Erschaffung des Feldes `start[]` im Konstruktor

Verwendung

Zerstörung des Feldes `start[]` durch Destruktor

- **Wie werden Felder von Objekten konstruiert/vernichtet?**
  - **Default-Konstruktor wird für jedes Feldelement aufgerufen, beginnend mit dem niedrigsten Index**
    - **Default-Konstruktor muss vorhanden sein!**
  - **Destruktor wird für jedes Feldelement aufgerufen, beginnend mit dem höchsten Index**

```
class iarray {
   private:
      int *start;
      const int deflen;
   public:
      iarray();
      iarray(int laenge);
      int element(int);
      ~iarray();
};
```

Standard-Feldlänge

verpflichtend bei Feldern von `iarray`

- **Verwendung:**

```
void f() {
    iarray x[2];
    ...
    t = x[1].element(9) + x[0].element(4);
    ...
    return;
}
```

Aufruf von `x[0].iarray()` gefolgt von `x[1].iarray()`

Zugriff wie bei Strukturen

Aufruf von `x[1].~iarray()` gefolgt von `x[0].~iarray()`

- **Für die Feldelemente ist keine explizite Initialisierung bei der Instanziierung vorgesehen**
  - **deswegen muss es den Default-Konstruktor geben**
  - **wie wird die Standard-Feldlänge gesetzt?**

- **Aktuelle Implementierung**
  - Initialisierung von konstanten Memberdaten erfolgt mit " : " im Kopf des Default-Konstruktors
  - Jeder Konstruktor kann die Daten anders initialisieren

```
iarray::iarray() : deflen(20) {
   start = new int[deflen];
   // cout << "Konstruktion " << start << endl;
}

iarray::iarray(int laenge) : deflen(laenge) {
   start = new int[laenge];
}

iarray::~iarray() {
   // cout << "Destruktion " << start << endl;
   delete[] start;
}
```

hier eigentlich unnnötig,
siehe jedoch später...

- **Diese Art der Initialisierung funktioniert auch für mehrere und nicht konstante Daten (Beiordnung mit " , ")**
  - eigentlich werden hier die passenden Konstruktoren der Member gerufen

```
class iarray {
   private:
      int *start;
      const int deflen;
      bool used;
   public:
      iarray();
      iarray(int laenge);
      int element(int);
      ~iarray();
};
iarray::iarray() :
deflen(20),used(false) {
   start = new int[deflen];
}
```

```
iarray::iarray(int laenge) :
deflen(laenge),used(false) {
   start = new int[laenge];
}

iarray::~iarray() {
   delete[] start;
}

int iarray::element(int i) {
   used = true;
   return start[i];
}
```

- Oft müssen Memberfunktionen die Daten des Objekts nur lesen, aber nicht schreiben können
  - der Versuch zu schreiben sollte dann mit einem Fehler quittiert werden
- Lösung: konstante Memberfunktionen
  - müssen in Deklaration und Implemetierung als `const` gekennzeichnet sein
  - Beispiel: `iarray::element(int)`

```
class iarray {
  …
  int element(int) const;
};

iarray::element(int i) const {
  // used = true;    // FEHLER!
  return start[i];
}
```

- Compiler wacht über die Einhaltung der `const`-Deklaration

- Wie bewerkstelligt man, dass eine konstante Memberfunktion bestimmte Daten trotzdem schreiben kann?
  - Deklaration der Daten als `mutable`

```
class iarray {
   private:
      int *start;
      const int deflen;
      mutable bool used;
   public:
      iarray();
      iarray(int laenge);
      int element(int) const;
      ~iarray();
};
```

```
int iarray::element(int i) const {
   used = true;
   return start[i];
}
```

Veränderung von `used` erlaubt trotz `element(int) const`

- Daten, die allen Instanzen einer Klasse gemeinsam gehören, heißen statische Datenkomponenten oder Klassenvariablen
  - Deklaration mittels `static`
  - Klassenvariablen können alle Sichtbarkeitsregeln haben
  - Definition und Initialisierung erfolgt stets auf globaler Ebene
  - Klassenvariablen können auch `const` sein

```
class iarray {
   private:
     …
   public:
      static int memory;
      iarray();
      iarray(int laenge);
      int element(int) const;
      ~iarray();
};

int iarray::memory = 0;
```

```
iarray::iarray(int laenge) :
deflen(laenge),used(false) {
 memory += laenge*sizeof(int);
 start = new int[laenge];
}

iarray::~iarray() {
 memory -= deflen*sizeof(int);
 delete[] start;
}
…
```


- Zugriff auf statische Memberdaten
  - innerhalb von Objekten der Klasse einfach mit dem Namen (s.o.)
  - aus anderen Objekten oder dem globalen Sichtbarkeitsbereich mittels

    `<Klassenname>::<stat. Member>`

    oder

    `<Objekt>.<stat. Member>`

    oder

    `<Zeiger auf Objekt>-><stat. Member>`

  - Beispiel:

```
iarray f;
cout << "Speicherverbrauch: " << f.memory << "=" <<
  iarray::memory << "=" << (&f)->memory << endl;
```

- Jede Instanz einer Klasse enthält automatisch einen Zeiger vom Typ `(<Klassenname> * const)`, der auf die Instanz zeigt
  - dieser Zeiger heißt `this`
  - alle Member eines Objektes können über `this` angesprochen werden:

```
double z = this->re;
int i = this->element(12);
```

- Hauptanwendungen
  - Listen (Objekt kann sich selbst mittels `this` in die Liste einklinken)
  - Rückgabe des Objektes aus sich selbst (v.a., aber nicht nur bei überladenen Operatoren → später):

```
complex& complex::conj() {
 im *= -1;
 return *this;
}
```



```
#include <iostream.h>

class iarray {
   private:
      int *start;
      const int deflen;
      mutable bool used;
   public:
      static int memory;
      iarray();
      iarray(int laenge);
      int element(int) const;
      void set(int,int);
      ~iarray();
};
int iarray::memory=0;
```

```
iarray::iarray() :
deflen(20),used(false) {
   start = new int[deflen];
   memory += deflen*sizeof(int);
}

iarray::iarray(int laenge) :
deflen(laenge),used(false) {
   start = new int[laenge];
   memory += laenge*sizeof(int);
}
```

Komplettes iarray-Beispiel (aktuelle Version)

```
iarray::~iarray() {
   cout << "Destruktion " << start << endl;
   memory -= deflen*sizeof(int);
   delete[] start;
}

int iarray::element(int i) const {
   used = true;
   if(i<deflen)
      return start[i];
   else {
      cerr << "iarray::element : " << i
           << " > " << deflen-1 << endl;
      return 0;
   }
}
```

Fehlerausgabe bei Bereichsüberschreitung

```
void iarray::set(int i, int val) {
   used = true;
   start[i] = val; // check?
}
```


Komplettes iarray-Beispiel (aktuelle Version)

- **Benutzung**

```
int main()
{
   iarray f(100),u;
   cout << "Bytes: " << iarray::memory << endl;
   f.set(10,56);
   cout << f.element(10) << endl;
   return 0;
}
```

Objektorientierung in C++: Zwischenstop

- **Klassen** sind eine starke Erweiterung des Strukturbegriffes von C
  - sie bestehen aus Daten und Funktionen (Member)
- **Objekte** sind konkrete Instanzen von Klassen
- **Konstruktoren**: spezielle Memberfunktionen, bei Instanziierung gerufen
  - Überladung möglich
  - Felder von Objekten: Default-Konstruktor!
  - Spezielle Behandlung konstanter Memberdaten
- **Destruktor**: spezielle Memberfunktion, bei Zerstörung gerufen
- **Konstante Memberfunktionen**: kein schreibender Zugriff auf Objektdaten
- **Klassenvariablen**, statische Member: gehören allen Instanzen der Klasse ("klassenglobal"?)
  - globale Definition erforderlich
- **this**: Zeiger auf "dieses" Objekt

Anwendungen und Beispiele zu Klassen und Objekten

- **Beispiel 1: Bruchklasse (Aufgabe 28)**
  - **Deklaration:**

```
class bruch {
 private:
   int z,n;
 public:
   bruch();                    // Default-Konstruktor
   bruch(const int, const int=1);// 2 Konstruktoren
   double dezim() const;
   int zaehler() const;
   int nenner() const;
   bruch sum(const bruch &) const;
   bruch mult(const bruch &) const;
   bruch sub(const bruch &) const;
   bruch div(const bruch &) const;
   void print(bool=false) const;
 private:
   void kuerzen();
};
```

Default-Argument

soviel `const` wie möglich!

nur intern benötigte Funktion

Bruchklasse (Forts.)

- Implementierung:

```
bruch::bruch() {
   z=1; n=1;
}

bruch::bruch(const int a, const int b) {
   z=a;
   if(0==b)
      cerr << "Fehler in bruch::bruch(int,int) - Nenner=0"
           << endl;
   else
      n=b;
   kuerzen();
}

double bruch::dezim() const {
   return (double)z / n;
}
```

Nenner=0 nicht erlaubt!

Gleitkomma-Division erzwingen!

Bruchklasse (Forts.)

```
int bruch::zaehler() const { return z; }
int bruch::nenner() const { return n; }

bruch bruch::sum(const bruch &s) const {
   bruch t;
   t.z = z*s.n + n*s.z;
   t.n = n*s.n;
   t.kuerzen();
   return t;
}

bruch bruch::mult(const bruch &s) const {
   bruch t;
   t.z = z*s.z;
   t.n = n*s.n;
   t.kuerzen();
   return t;
}
```

direkter Zugriff trotz `private` innerhalb der Klasse erlaubt!

Bruchklasse (Forts.)

```
bruch bruch::sub(const bruch &s) const {
   bruch t(z*s.n - n*s.z, n*s.n);
   t.kuerzen();
   return t;
}

bruch bruch::div(const bruch &s) const {
   bruch t(s.n, s.z);
   t = mult(t);
   t.kuerzen();
   return t;
}

void bruch::print(bool nl) const {
   cout << z ;
   if(1!=n)
      cout << "/" << n;
   if(nl) cout << endl;
}
```


Bruchklasse (Forts.)

```
void bruch::kuerzen() {
   int dd = (z<n?z:n);
   bool done = false;

   for(; dd >= 2 && !done ; dd--)
      if(z%dd == 0 && n%dd == 0) {
         z = z / dd;
         n = n / dd;
         done = true;
      }
}
```

Bruchklasse (Verwendung):

```
int main() {
   bruch a,b,c(1,2),feld[10];

   a=c.mult(c);
   a.print(true);
   b=c.div(a);
   b.print(true);
   a=b.sum(c);
   a.print(); cout << " = " << a.dezim() << endl;
   feld[0]=c.sub(c);
   feld[0].print(true);
...
}
```


`iarray` soll (irgendwann) eine verbesserte Version des C++-Feldes implementieren

Aktuelle Version

- dynamisches Anlegen eines `int`-Feldes
- Zugriff auf Feldelemente mit `iarray::element()` zum Lesen bzw. `iarray::set()` zum Schreiben
- Überprüfung auf Bereichsüberschreitung beim Feldindex
- Protokollierung des Speicherverbrauchs aller Objekte der Klasse
  - mittels Klassenvariable

Wünschenswerte Verbesserungen

- schreibender Zugriff mittels `iarray::element()`, d.h. `iarray::set()` sollte überflüssig sein
- automatische Vergrößerung des Feldes bei Zugriff auf einen Index außerhalb der Feldgröße
- Zugriff auf Feldelemente mittels Indexklammer `[]`
- Anwendung der entwickelten Methoden auch auf andere Datentypen als `int`

Verbesserte `iarray`-Klasse, Version 1 (ohne `iarray::set()`)

```
class iarray {
 private:
   int *start;
   const int deflen;
   mutable bool used;
 public:
   static int memory;
   iarray();
   iarray(const int);
   int& element(const int)
       const;
   ~iarray();
};

int iarray::memory=0;
```

Referenzrückgabe macht `iarray::element()` zum L-Wert

```
int& iarray::element(const int i) const
{
   used = true;
   if(i<deflen)
      return start[i];
   else {
      cerr << "iarray::element : "
           << i << " > "
           << (deflen-1) << endl;
      return start[0];
   }
}
```


Verbesserte `iarray`-Klasse, Version 1 (ohne `iarray::set()`)

- **Anwendung:**

```
int main() {
   iarray f(100);
…
   f.element(42)=5;
   f.element(42)++;

   f.element(67)=f.element(42);
…
}
```

- `iarray::element()` ist nun die zentrale Zugriffsfunktion
  - automatische Vergrößerung sollte hier ansetzen
- Strategie: Falls Feldindex größer als erlaubt,
  - allokiere neues Feld
  - kopiere alte Daten ins neue Feld
  - gib altes Feld frei

Verbesserte `iarray`-Klasse, Version 2 (autom. Vergrößerung)

```
class iarray {
 private:
   int *start;
   int deflen;
   mutable bool used;
 public:
   static int memory;
   iarray();
   iarray(const int);
   int& element(const int);
   ~iarray();
 private:
   void resize(const int);
};

int iarray::memory=0;
```

`const` fehlt

```
void iarray::resize(const int i) {
   // neuen Platz reservieren
   int *t = new int[i];
   // alte Daten kopieren
   for(int j=0; j<deflen; j++)
      t[j] = start[j];
   // alten Bereich freigeben
   delete[] start;
   // Speicherverbrauch Update
   memory = memory+sizeof(int)*
            (i-deflen);
   start=t; deflen=i;
}
int& iarray::element(const int i)
{
   used = true;
   if(i>=deflen)
      resize(i+10);
   return start[i];
}
```


Verbesserte `iarray`-Klasse, Version 2 (autom. Vergrößerung)

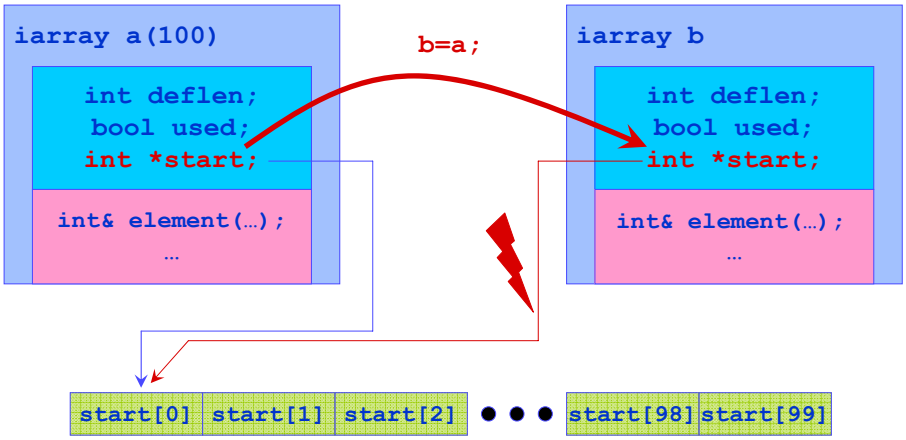
Anwendung

```
int main() {
   iarray f(100);
…
   f.element(150)=5;              // 1. Vergrößerung
   f.element(150)++;

   f.element(67)=f.element(200);  // 2. Vergrößerung
…
}
```

Probleme bei Objektzuweisungen

- **Vorsicht bei der Zuweisung von `iarray`-Objekten mittels "="!**
  - Jedes Objekt enthält einen Zeiger `start`, der auf die eigentl. Daten zeigt
  - bei einer Zuweisung werden die Daten des Objekts 1:1 byteweise kopiert
  - das schließt `start` mit ein, nicht jedoch die Daten, auf die `start` zeigt!



Verbesserte Personenklasse

- **`struct Person` ist ziemlich einfach gestrickt**
  - keine Methoden, keine dynamische Allokierung
- **Verbesserung: `class Person` soll**
  - einen Konstruktor haben, der eine Person mit Namen, Alter und Tel.-Nr. erzeugt und dabei keinen Speicher verschwendet
  - einen Destruktor haben, der diesen Speicher wieder freigibt

```
class Person {
 private:
   char *name;
   int alter;
   char *telefon;
 public:
   Person(const char*, const int, const char*);
   ~Person();
   char *getname() const;
   int getalter() const;
   char *getfon() const;
};
```

Implementierung

```
Person::Person(const char* n, const int a, const char* t)
   : alter(a){
   name = new char[strlen(n)+1];
   strcpy(name, n);
   telefon = new char[strlen(t)+1];
   strcpy(telefon,t);
}

Person::~Person() {
   delete[] name;
   delete[] telefon;
}

char* Person::getname() const {
   return name;
}

...
```

Einkopieren der neuen Daten in das Objekt

Verwendung

```
int main() {
   Person y("Hans Mueller", 45, "09131/3456788");

   cout << "Person y heisst " << y.getname() << " und ist "
        << y.getalter() << " Jahre alt." << endl;

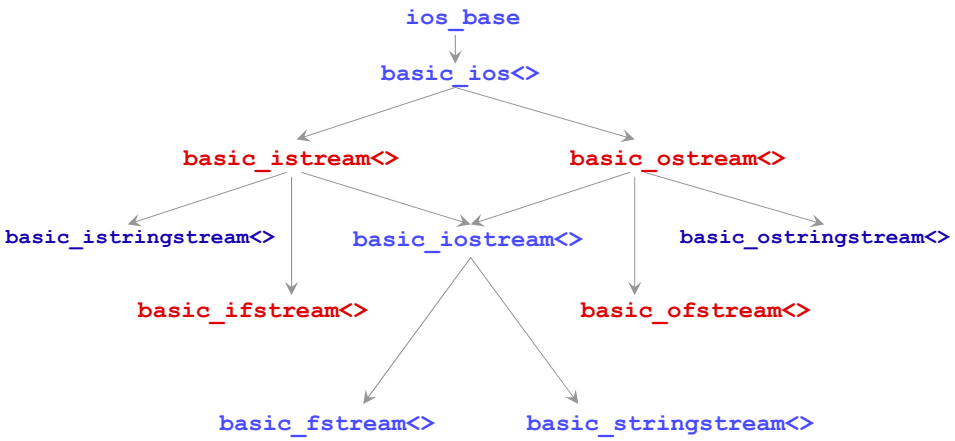
...
}
```

Erweiterungen?

(8) Ein-/Ausgabe Teil II: Arbeiten mit Dateien

- C++ definiert eine von C völlig getrennte I/O-Struktur durch die `iostream`-Bibliothek
- Ein- und Ausgaben erfolgen durch sog. Streams
  - "Stream" = Datenstrom von einer Quelle zu einem Ziel
  - Umgangssprache: Quelle bzw. Ziel wird als "Stream" bezeichnet
- Bekannt: Objekte `cin`, `cout`, `cerr`, (`clog`)
  - Instanzen der Klassen `istream` bzw. `ostream`, beim Programmstart präsent
  - `cerr`: Standard-Fehlerausgabe, ungepuffert
  - `clog`: Logausgabe (ebenfalls auf Konsole), gepuffert
  - Schiebeoperatoren <<,>> : erwarten Referenz und geben Referenz auf Stream-Objekt zurück (Verkettung!)

- Vorgriff: Klassenhierarchie des C++-I/O-Systems (ANSI)



- Diese Hierarchie ist normalerweise für die Arbeit mit der Library nicht wichtig
  - Fehlermeldungen werden bei Kenntnis jedoch klarer
- Leider folgen nicht alle Compiler(versionen) diesen ANSI-Konventionen
  - Quelle von Portabilitätsproblemen!

- Kurze Erklärung der Funktionen einiger Klassen
  - `ios_base` kontrolliert die Art, wie Ein-/Ausgaben durchgeführt werden (auch das Format)
  - `ostream`, `istream`, `iostream`: Operationen zum Lesen und Schreiben der Standard-Datentypen (z.B. <<)
  - `fstream` enthält Interfaces für die Datei-Ein-/Ausgabe
  - `stringstream` erlaubt die Behandlung eines Textfeldes als Quelle oder Senke eines Streams

- Eingabe-Schiebeoperator >> liest rigoros über Leerzeichen, Tabs und Newline hinweg
  - ähnlich `scanf()` in C
  - Spezielle Memberfunktionen von `istream` erlauben flexiblere Eingabemöglichkeiten…

- Einige nützliche Memberfunktionen von `istream` und `ostream`:

```
ostream& ostream::put(char);
ostream& ostream::write(const char*, int);
ostream& ostream::flush();
istream& istream::get(char&);
int istream::peek();
istream& istream::putback(char);
istream& istream::read(char*, int);
istream& istream::get(char*, int, char='\n');
istream& istream::getline(char*, int, char = '\n')
```

- Datei-Ein-/Ausgabe wird durch die Klassen `fstream`, `ifstream` bzw. `ofstream` implementiert (durch `#include <fstream>` deklariert)
- Konstruktoren (X = 'f', 'if' oder 'of') stellen ein Dateistream-Objekt zur Verfügung:

```
Xstream::Xstream();
Xstream::Xstream(const char* name, int mode);
```

`name`: Name der zu öffnenden Datei
`mode`: Dateimodus beim Öffnen

- Nach dem Öffnen eines Datei-Streams sind alle von `iostream` gewohnten formatierten und unformatierten Ein-/Ausgabemöglichkeiten vorhanden
- Dateistream-Objekt kann auch mit Default-Konstruktor ohne zugehörige Datei erzeugt werden
  - Zuordnung kann später durch Memberfunktion erfolgen

- Dateimodi in `ios_base` (implementierungsabhängig):
  - Modus gibt an, wie die Datei geöffnet werden soll

```
enum open_mode  {
        in    = 0x01,     // open for reading
        out   = 0x02,     // open for writing
        ate   = 0x04,     // seek to eof upon original open
        app   = 0x08,     // append mode: all additions at eof
        trunc     = 0x10, // truncate file if already exists
        nocreate = 0x20,  // open fails if file doesn't exist
        noreplace= 0x40   // open fails if file already exists
        binary    = 0x80  // open in binary mode
};
```

- ODER-Verknüpfung (Überlagerung) der Modi ist möglich (ähnlich Formatflags):

```
ofstream datei("ausgabe.txt",
  ios_base::out | ios_base::app | ios_base::nocreate);
```

Ergänzung: Aufzählungsdatentypen mit enum

- `enum` dient zur Deklaration eines speziellen Aufzählungs-Datentyps

```
enum bla { a, b, c, d };
```

- `bla` ist ein neuer Datentyp; Variablen dieses Typs können nur Symbole aus der angegebenen Menge zugewiesen werden

```
bla x;
x = b;      // OK
x = 2;      // FALSCH
```

- Die Symbole des Aufzählungstyps (Aufzählungskonstanten) werden ohne weitere Angabe von 0 beginnend aufsteigend mit Ganzzahlwerten belegt
  - also: a=0, b=1, c=2, d=3
- Explizite Angabe von Konstanten int-Ausdrücken ist erlaubt:

```
enum blub { a=1, b=2, c, d, e=b-1 }; // 1,2,3,4,1
```


Aufzählungsdatentypen mit enum

- Aufzählungstypen sind kompatibel zu `int` – überall wo `int` stehen kann, kann auch ein `enum`-Typ verwendet werden

```
blub f = a;
int x = 4;

x += f;          // Ergebnis 5
```

- Aufzählungstyp innerhalb einer C++-Klasse: Aufzählungskonstanten sind im Namensraum der Klasse enthalten

```
class schueler {
 public:
   enum noten { sehr_gut=1, gut, befriedigend,
                ausreichend, mangelhaft, ungenuegend };
…
};

int main() {
   schueler::noten Note = schueler::gut;
   …
}
```

- **Default-Dateimodi**

  `fstream:` kein Default
  `ifstream:` `ios_base::in`
  `ofstream:` `ios_base::out`

- **Memberfunktionen der Dateistream-Klassen**
  - zum Öffnen von Dateien (falls Default-Konstruktor verwendet wurde)

```
void fstream::open(const char*, int);
void ifstream::open(const char*, int=ios_base::in);
void ofstream::open(const char*, int=ios_base::out);
```

  - zum Schließen (erfolgt auch automatisch durch Destruktor)

```
void Xstream::close(); // X ist 'f', 'if' oder 'of'
```

  - weitere Memberfunktionen (Positionierung) siehe später

- **Beispiele:**
  - Formatierte Datenausgabe in eine Datei:

```
ofstream g("bla.txt");
for(int i=0; i<10; i++) {
   g << setw(10) << i << " " << setw(10) << i*i << endl;
}
g.close();
```

  - Unformatierte Ausgabe in eine Datei:

```
int feld[] = {1,2,3,4,5,6,7};
ofstream h("bin.out");
h.write((char*)feld,sizeof(int) * 7);
h.close();
```

Beispiele (Forts.)

- Programm zum Ausdrucken einer Datei

```
int main(int argc, char** argv) {
   char d;
   if (argc!=2) return 1;
   ifstream f(argv[1]);
   if(!f) {
      cerr << argv[1] << " nicht lesbar" << endl;
      return 1;
   }
   while(f.get(d)) {
      cout << d;
   }
   f.close();
   return 0;
}
```

Konvertierung erlaubt Prüfen auf "Fehler" (hier "kein File")

Konvertierung erlaubt Prüfen auf "Fehler" (hier End-Of-File [EOF])

Beispiele (Forts.):

- Feld von Brüchen speichern und wieder lesen
- globale Funktionen übernehmen die eigentlichen Dateioperationen
  - Ein-/Ausgabe-Streams werden als Referenzen übergeben, d.h. Funktionen nehmen an, dass der Stream bereits mit einem File verknüpft ist
  - Rückgabe der Referenz zur einfachen Fehleranalyse im aufrufenden Programmteil(s.u.)

Übergabe des Streams!

```
ostream& sichern(ostream& os, bruch* f, int n) {
   int i;
   for(i=0; i<n; i++)
      os.write((char*)(f+i),sizeof(bruch));
   return os;
}

istream& lesen(istream& is, bruch* f, int n) {
   is.read((char*)f, n*sizeof(bruch));
   return is;
}
```

Speichern Bruch für Bruch

Lesen als Block von n Brüchen

Bruchfeld speichern und lesen (Forts.)

```
#include <iostream>
#include <fstream>
using namespace std;

int main(void) {
   bruch f[10], g[10];
…
   fstream file("brueche.dat", ios_base::out);
   if(!sichern(file, f, 10))
      cerr << "Fehler beim Schreiben!" << endl;
   file.close();
   file.open("brueche.dat", ios_base::in);
   if(!lesen(file, g, 10))
      cerr << "Fehler beim Lesen!" << endl;
   file.close();
   for(int i=0; i<10; i++)
      g[i].print(true);
…
}
```

nach dem Aufruf auf Stream-Fehler prüfen

Beispiele (Forts.):

- Möglichkeit zum Speichern auf File innerhalb der Bruchklasse
  - neue Methoden `bruch::sichern()` bzw. `bruch::lesen()`

```
ostream& bruch::sichern(ostream& os) {
   os.write((char*)&z,sizeof(int));
   os.write((char*)&n,sizeof(int));
   return os;
}

istream& bruch::lesen(istream& is) {
   is.read((char*)&z, sizeof(int));
   is.read((char*)&n, sizeof(int));
   return is;
}
```

Anwendung z.B.:

```
ofstream file;
...
for(i=0; i<n; i++)
   f[i].sichern(file);
```

Beispiele (Forts.):

- Speichern von Personendaten aus Objekten mit dynamischer Speicherverwaltung (verbesserte `Person`-Klasse)
- Problem: Woher weiß `Person::lesen()`, welche Länge die Felder für Namen und Tel.-Nr. haben?
  - Längen werden mit abgespeichert!

```
ostream& Person::sichern(ostream& os) {
   int t;
   t=strlen(name);
   os.write((char*)&t, sizeof(int));
   os.write(name, t);
   os.write((char*)&alter, sizeof(int));
   t=strlen(telefon);
   os.write((char*)&t, sizeof(int));
   os.write(telefon, t);
   return os;
}
```

Feldlängen von `name` und `telefon` speichern

	Länge name	name	alter	Länge tel.	telefon
Bytes	4	???	4	4	???

Speichern und lesen von Personendaten (Forts.)

```
istream& Person::laden(istream& is) {
   delete[] name;
   delete[] telefon;
   int t;
   is.read((char*)&t, sizeof(int));
   name = new char[t+1];
   is.read(name, t);
   name[t]='\0';
   is.read((char*)&alter, sizeof(int));
   is.read((char*)&t, sizeof(int));
   telefon = new char[t+1];
   is.read(telefon, t);
   telefon[t]='\0';
   return is;
}
```

alte Daten wegwerfen

Längen der Zeichenketten lesen und passend Speicher anfordern

- Speichern und lesen von Personendaten (Forts.)
  - Anwendung: Abspeichern und laden eines Feldes von Personendaten

```
void save_personen(Person* p, int n, char* filename) {
   ofstream f(filename);
   int i;
   for(i=0; i<n; i++)
      p[i].sichern(f);
   f.close();
}

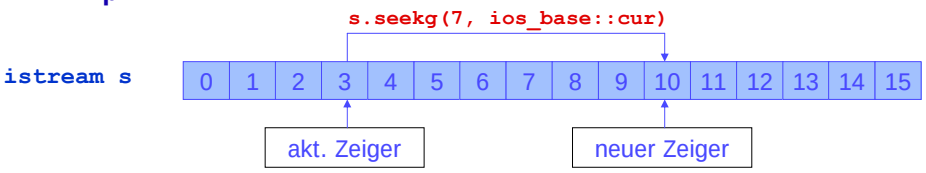
void load_personen(Person* p, int n, char* filename) {
   ifstream f(filename);
   int i;
   for(i=0; i<n; i++)
      p[i].laden(f);
   f.close();
}
```

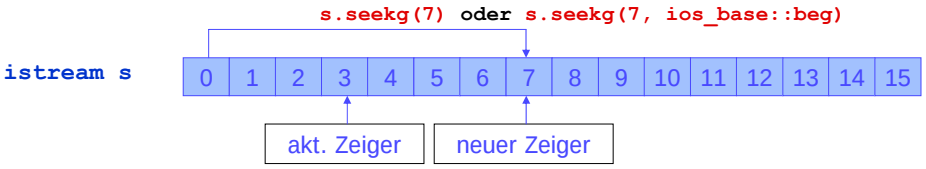

- Zu jedem Stream exisitiert ein sog. Dateizeiger, der angibt, an welcher Stelle die nächste Ein-/Ausgabeoperation stattfindet
- In C++ ist das Positionieren dieses Zeigers innerhalb eines Streams möglich
- Positionierung für Lesen und Schreiben kann unabhängig erfolgen:
  - `istream& istream::seekg(pos_type);`
    Positionieren des Lesezeigers absolut
  - `istream& istream::seekg(off_type, seekdir);`
    Positionieren des Lesezeigers relativ zu bestimmter Position
  - `ostream& ostream::seekp(pos_type);`
    Positionieren des Schreibzeigers absolut
  - `ostream& ostream::seekp(off_type, seekdir);`
    Positionieren des Schreibzeigers relativ
  - `pos_type istream::tellg();`
    `pos_type ostream::tellp();`
    Ermittlung der Position des Schreib- bzw. Lesezeigers (in `char`s von 0 an gezählt)

- `pos_type` ist ein Ganzzahltyp, `seekdir` ein Aufzählungstyp:

```
class ios_base {
…
   enum seekdir {
      beg,      // Suche vom Anfang aus
      cur,      // Suche von akt. Position
      end       // Suche vom Ende aus
   };
};
```

- **Prinzip:**





- **Beispiel**

```
ofstream g("bla.txt");
for(int i=0; i<10; i++) {
   g << "*";
}
g.seekp(-4, ios_base::cur);
g << "XXX";
cout << "Dateizeiger bei " <<
   g.tellp() << endl;
g.close();
```

Ausgabe:
`Dateizeiger bei 9`

Dateiinhalt:
`******XXX*`

- Jede Dateioperation kann den Stream in einem von mehreren Fehlerzuständen hinterlassen
- Abfrage des Zustandes mittels

```
int ios_base::rdstate();
```

mögliche Rückgabewerte:

```
class ios_base {
…
   enum io_state   {
        goodbit  = 0x00,    // no bit set: all is ok
        eofbit   = 0x01,    // at end of file
        failbit  = 0x02,    // last I/O operation failed
        badbit   = 0x04,    // invalid operation attempted
    };
};
```


- Interpretation des Fehlerstatus
  1. Falls `goodbit` gesetzt ist, lief die letzte Operation ok und die nächste wird voraussichtlich auch erfolgreich sein
  2. `eofbit` zeigt einen Fileende-Status an
  3. `failbit` zeigt Fehler bei der letzten Operation an, der Stream kann jedoch weiter genutzt werden
  4. `badbit` signalisiert, dass eine unerlaubte Operation auf dem Stream versucht wurde; Wiederaufsetzen nur nach einer Korrektur der Fehlerbedingung
- Abfrage des Status ist auch mit Memberfunktionen möglich:

```
bool ios_base::good(); // goodbit
bool ios_base::eof();  // eofbit
bool ios_base::fail(); // failbit | badbit
bool ios_base::bad();  // badbit
bool operator!();
```

`true`, falls `failbit` oder `badbit` gesetzt

- (Rück)setzen des Fehlerstatus (alle Flags) geschieht über die Funktion

```
void ios_base::clear(int = 0);
```

  - damit ist auch das explizite Setzen des Fehlerstatus möglich (z.B. in einer Funktion)

- Direkte Abfrage des Streams als Bedingung liefert `true`, falls `fail()` den Wert `false` liefern würde (also auch bei EOF!):

```
char buffer[100];
ifstream in("input.txt");
if(in.getline(buffer, 100)) {
   // Daten verarbeiten
} else {
   if(in.fail()) cerr << "Fehler beim Lesen!" << endl;
}
```


- Beispiel: Programm zum Zählen von Wörtern in einem File

```
#include <iostream>
#include <fstream>
#include <ctype>
using namespace std;

int naechstes(ifstream& f) {
   char c;
   int wortlaenge=0;

   f >> c;
   while(!f.eof() && !isspace(c))
   {
      ++wortlaenge;
      f.get(c);
   }
   return wortlaenge;
}
```

`true`, falls `c` Leerraum ist

```
int main() {
   int woerter=0;
   ifstream
       file("words.cpp");
   while(naechstes(file))
      ++woerter;
   cout << woerter <<
     " Woerter im File\n";
   return 0;
}
```

Hier sind am Ende `eofbit` und `failbit` gleichzeitig gesetzt (kein Setzen von `eofbit` nach Lesen des letzten Zeichens!)

- C++ erlaubt das Binden von Streams an dynamisch verwaltete Zeichenketten (stringstreams)
  - Klassen: `istringstream`, `ostringstream`, `stringstream`
- Konstruktoren:

```
#include <sstream>
using namespace std;

ostringstream(int = ios_base::out);
ostringstream(char*, int = ios_base::out);

istringstream(int = ios_base::in);
istringstream(char*, int = ios_base::in);

stringstream(int = ios_base::out | ios_base::in);
stringstream(char*, int = ios_base::out | ios_base::in);
```

auf String schreiben

aus String lesen

lesen & schreiben

- Stringstreams können behandelt werden wie normale Files, d.h. alle Memberfunktionen aus den `iostream` sind vorhanden
- `ostringstream` und `stringstream` besitzen noch die Methode

```
char* Xstringstream::str();
```

die einen Zeiger auf den String zurückliefert

- `char*`-Initialisierer belegen den Stream mit einem "Startstring" vor
  - `ostringstream`: Null-terminierter Startstring und append-Modus führen zum Anhängen der folgenden Daten, ansonsten wird von vorne anfangend überschrieben
- Speicherverwaltung ist vollkommen automatisch!

- **Beispiel:**

```
#include <sstream>
#include <iostream>
using namespace std;

int main() {
   char *a="Starttext";
   ostringstream s(a,ios_base::app);
   ostringstream t(a);
   s << " Anhang" ;
   t << "XXXX";
   cout << s.str() << endl;
   cout << t.str() << endl;
   return 0;
}
```

anhängen an den Initialisierungs-String

überschreiben des Initialisierungs-Strings

Ausgabe:

```
Starttext Anhang
XXXXttext
```


- **Vorsicht: (String)streams sind Quelle von Kompatibilitätsproblemen**
- **Nicht-aktuelle Compiler verwenden oft andere Bezeichungen bzw. andere Semantik bei den Konstruktoren**
  - `strstream.h` **statt** `sstream`
  - **Klasse** `Xstrstream` **statt** `Xstringstream`
  - **zwingende Längenangaben beim Konstruktor (Pufferlänge!)**
  - `ios::` **statt** `ios_base::`
  - `seek_dir statt seekdir`
  - **...**
- **Problematisch: BC++ 5.0, GNU C++ 2.95.X**
- **O.k.: Intel Compiler V7, GNU C++ 3.2, aktuelle Microsoft-Compiler**

(9) Vertiefung ausgewählter Themen

(9a) Ausdrücke

- **Beispiel :**

$$h = a * b + \frac{c}{1 + \frac{m}{k*s}} * \ln(m)$$

- **Komponenten:** **Operanden und Operatoren**

Operanden: gemischt

- Summenausdrücke
- Produktausdrücke
- Quotientenausdrücke
- einfache Variablen
- Konstanten

- **Operatorvorrang:** "Punktrechnung vor Strichrechnung"
- **Stelligkeit von Operatoren:** Anzahl der nötigen Operanden
  - Vorzeichen: *einstellig*
  - Addition, Subtraktion, Multiplikation, Division: *zweistellig*
  - Potenzierung: *zweistellig* (a hoch b)
- **Assoziativität der Operatoren:** implizite Klammerung bei mehrgliedrigen Ausdrücken
  `a*b*c` → `(a*b)*c`
- **nichttriviale Ausdrücke:**
  - funktionale Ausdrücke: ln (m)
  - Indizierte Ausdrücke in Matrix- oder Vektor-Operationen ($a_{i,j}$)
  - Index-Ausdrücke $a_{i*k-1,j}$

Ausdrücke in Programmiersprachen

- **Ausdrücke sind Teile von Anweisungen, d.h. von ausführbaren programmiersprachlichen Gebilden**

- **Ausdrücke dienen**
  - **der Berechnung von Werten und**
  - **der Steuerung des Programmablaufs**

- **in C++ : Ausdrücke sind Folgen von Operanden und Operatoren, die**
  - **ein Objekt bezeichnen**
  - **einen Wert berechnen**
  - **einen Objekt-Typ festlegen oder ändern**
  - **einen Seiteneffekt auslösen.**

  - **diese Wirkungen können auch kombiniert auftreten!**

Operatoren

Operator	Bedeutung	Prio.	Stelligkeit	Assoz.
`( )`	Klammern in Ausdruck u. Funktion	1		l
`[ ]`	Klammernpaar für Feldindex-Ausdruck			
`.`	Punktop.: Auswahl einer Strukturkomp.		2	
`->`	Pfeilop.: Dereferenzieren u. Strukturkomp.		2	
`!`	logische Negation	2	1	**r**
`~`	bitweise Negation			
`++`	Inkrement um 1			
`--`	Dekrement um 1			
`-`	neg. Vorzeichen			
`(<typ>)`	explizite Typumwandlung (casting)			
`*`	Inhaltsoperator (Dereferenzieren)			
`&`	Adreßoperator			
`sizeof`	Speichergröße eines Objekts in Byte			
`*`	Multiplikation	3	2	l
`/`	Division			
`%`	Modulo-Operator (Divisionsrest bei ganzzahl.Div.)			
`+`	Addition	4	2	l
`-`	Subtraktion			
`<<`	bit-weises Schieben nach links (left shift)	5	2	l
`>>`	bit-weises Schieben nach rechts (right shift)			

Operatoren (2)

Operator	Bedeutung	Prio.	Stelligkeit	Assoz.
`<`	Vergleichsoperator - kleiner als	6	2	l
`<=`	Vergleichsoperator - kleiner als oder gleich			
`>`	Vergleichsoperator - größer als			
`>=`	Vergleichsoperator - größer als oder gleich			
`==`	Vergleichsoperator - gleich	7	2	l
`!=`	Vergleichsoperator - ungleich			
`&`	bit-weise UND (AND)	8	2	l
`^`	bit-weise XODER (eXklusives ODER, XOR)	9	2	l
`|`	bit-weise ODER (OR)	10	2	l
`&&`	logisches UND (AND)	11	2	l
`||`	logisches ODER (OR)	12	2	l
`? :`	Bedingter Ausdruck	13	3	**r**
`=`	Zuweisungsoperatoren	14	2	**r**
`+= -=`				
`*= /=`				
`%=`				
`>>= <<=`				
`&= |= ^=`				
`,`	Komma-Operator: Verkettung von Anweis.	15	2	l

Reihenfolge der Operandenauswertung

- **Reihenfolge festgelegt nur für:** `&&` `||` `? :` `,`
  - `e1 && e2` Auswertung von e2 nur wenn e1 wahr
  - `e1 || e2` Auswertung von e2 nur wenn e1 falsch
  - `e1 ? e2 : e3` erster Operand e1 zuerst;
  - `e1 , e2` e1 zuerst, dann e2; Wert des Kommaausdrucks ist Wert von e2.
- **Alle anderen Operatoren: nicht festgelegt!**
- **Beispiele :**
  - `(a + b) + (c + d)` unbekannt, aber keine Probleme
  - `f (x, y) + g (y)` undefiniert und Problem der Ausführungsfolge der Funktionen, wenn y Rückgabeparameter: evtl. Wert von `y` in `g(y)` undefiniert !
  - `x/x++` implementierungsabhängig (x/x oder x/(x+1))?
  - `f(i, a[i++]);` ???
- **Abhilfe: Vollständig klammern oder Ausdrücke aufspalten!**

- `sizeof` wird angewendet auf Objekt oder Typ
- Zwei verschiedene Verwendungsweisen für `sizeof`
  - `sizeof <ausdruck>` Byte-Anzahl für Speicherung des Ausdrucks, bei Feldnamen: Speicherbedarf des Feldes Achtung: keine Auswertung von <ausdruck>
  - `sizeof ( <typname> )` Byte-Anzahl für ein Objekt des Typs <typname>, z.B. bei `char`: 1 (ANSI C)
- Explizite Typumwandlung mit Cast-Operator
  - `( <typname> ) x` Wert von x in Typ <typname> umgewandelt

von \ nach	void	Integer	Gleit-komma	Zeiger	Feld	Struktur	Funktion
void	#						
Integer	#	#	#	#			
Gleitkomma	#	#	#				
Zeiger	#	#		#	#		#
Feld	#			#			
Struktur	#					#	
Funktion	#						

- Integer: alle Ganzzahltypen
- Gleitkomma: alle Gleitkommatypen

Typumwandlungen

- **Umwandlung nach Integer**
  - von Integer: wenn Wert in Zieltyp darstellbar, dann umwandeln, sonst undefiniert
  - Kompromisse: vorzeichenloser Zieltyp - durch Abschneiden der MSB~Modulo-Bildung
  - von Gleitkomma: wenn ganzzahliger Anteil in Zieltyp darstellbar, dann umwandeln, sonst undefiniert
  - Zeiger (nur in C): Interpretation als gleichgroße Integerzahl
- **Umwandlung nach Gleitkomma**
  - `float` nach `double` möglich
  - `double` nach `float` durch Runden oder Abschneiden
  - von Integer entsprechende Näherung
- **Umwandlung nach Zeigertyp**
  - von Zeigertypen: jeder Zeigerwert kann zu jedem bel. Zeigertyp gewandelt werden
  - von Integer-Typen: nur NULL-Wert als NULL-Zeiger interpretierbar
  - von Feld-Typen: Ausdruck des Typs Feld-von-Typ T stets umgewandelt zu Zeiger auf T ; Feldname wird identifiziert mit Zeiger auf erstes Element (außer in sizeof)
  - von Funktions-Typen: Ausdruck des Typs Funktion mit Wert vom Typ T umgewandelt in Zeiger auf Funktion mit .... (außer im Funktionsaufruf)

Spezielle arithmetische Ausdrücke

- **Inkrement- und Dekrement mit den unären Operatoren `++` , `--`**
  - `++x` **Präinkrement**
    - Erst x um 1 erhöhen, dann im Ausdruck verwenden
  - `x++` **Postinkrement**
    - Erst x im Ausdruck verwenden, dann um 1 erhöhen
  - `--x` und `x--` analog (Prä- und Postdekrement)
- **Beispiele:**
  - `int i, j;`

```
i = 1;
j = ++i + 1;        //  j = 3, i = 2
i = 1;
j = i++ + 1;        //  j = 2, i = 2
i = 1;
i = ++i + 1;        //         i = 3
```

Beispiele

- Probleme aufgrund der Auswertungsreihenfolge von Ausdrücken bei der Programmierung mit Seiteneffekten:
  - `i = 0;`
    `a [i] = i++;` // Auswertungsreihenfolge ungeklärt !
    // erst Indexausdruck a[0] oder erst Zuweisungsausdruck,
    // dann a[1] (Implementierungsabhängig !)
  - Zuweisungsausdruck in der Wertzuweisung an das Feldelement bewirkt Seiteneffekt

Vergleiche und logische Ausdrücke

- Vergleichsoperatoren: > >= < <= == !=
- Vergleiche mit arithmetischen Operanden
  - gewohnte math. Interpretation
- Vergleiche mit Zeigern:
  - Vergleichbarkeit nur innerhalb derselben Struktur bzw. desselben Feldes,
  - Relation in Bezug auf Indexordnung und Abspeicherreihenfolge der Elemente
  - Vergleichsausdrücke erzeugen einen der Wahrheitswerte `true` oder `false` und sind damit kompatibel mit dem Typ `int`
  - Vergleichsausdrücke sind also logische Ausdrücke
- Achtung: keine Kettenbildung gemäß der math. Abkürzung: a < b < c
  - weil `a < b < c` → `(a < b) < c`, Jedoch `(a < b)` liefert true oder false
  - somit: `(true/false) < c`
  - Lösung: Auflösen der Kette durch Benutzung des UND Operators : `(a < b) && (b < c)`

logische Ausdrücke

- Logische Ausdrücke sind
  - alle `int`-Ausdrücke
  - alle Vergleichsausdrücke
  - alle durch Anwendung der Operatoren `&&` (UND), `||` (ODER), `!` (NICHT) auf logische Ausdrücke entstehenden Ausdrücke
  - Zeiger (implizite Typwandlung nach Integer)
    - Überprüfung auf NULL-Zeiger
  - Auswertung von links – Abbruch, wenn Ergebnis vorliegt
    - Bsp.: `a && b` : b nicht ausgewertet, falls a falsch
    - Bsp.: `a || b` : b nicht ausgewertet, falls a wahr
- Vergleichsoperatoren binden stärker als logische Operatoren!

Bedingte Ausdrücke u. Klammerausdrücke

- Bedingte Ausdrücke mit dem 3-stelligen Operator __ ? __ : ___
- Komma-Ausdrücke mit dem 2-stelligen Operator __ , __
- Ausdruck — Wert / Wirkung
  - `e0 ? e1 : e2` — `e0` auswerten, falls wahr → `e1` auswerten, falls falsch → `e2` auswerten
  - `e1 , e2` — 1. `e1` auswerten; 2. `e2` auswerten; Typ und Wert des Gesamtausdrucks werden durch Typ und Wert von `e2` bestimmt!
- Anwendung des Kommaausdrucks:
  - Verwendung zweier Ausdrücke, wo nur ein Ausdruck erlaubt ist
    `for (i=0, j=0; i<100; i++, j+=2)`
  - Achtung: Bei Verwendung bei einem Funktionsaufruf muss geklammert werden, um eine Verwechslung mit der Parametertrennung durch Kommata zu vermeiden!

Konstante Ausdrücke

- **Konstante Ausdrücke sind Ausdrücke, die der Compiler zur Übersetzungszeit auswerten kann**
  - Beispiel: `int i = 2*3+4/2;  --> int i = 8;`
    (von konstanten Ausdrücken wird nur das Ergebnis verwendet, jedoch kein Auswertungscode erzeugt)
- **Anwendungsbeispiele**
  - Argument der `#if`-Präprozessordirektive (s. später)
  - Dimensionierung von Feldern
  - `case`-Ausdrücke in `switch`-Anweisungen
  - explizite Definition von Aufzählungswerten
  - Initialisierungswerte von `static`- und `extern`-Variablen
- **Konstante Ausdrücke können aus `int`- und `char`-Konstanten durch Anwendung der folgenden Operatoren gebildet werden:**
  - unär : `+ - ~ . !`
  - binär: `+ - * / % << >> == != < <= > >= & ^ | && ||`
  - ternär: `_ ? _ : _`

Zuweisungen

- **Abkürzungsoperatoren: ursprünglicher Zuweisungs-Operator ist "="**
- **zusätzliche Zuweisungsoperatoren zur Abkürzung der**
  - binären arithmetischen und der
  - binären Bit-Ausdrücke
  - Schema:
    Operand_1 = Operand_1 ⊗ Operand_2 ⇒ Operand_1 ⊗= Operand_2
- **Vorteile**
  - manchmal bessere Lesbarkeit des Ausdrucks, insbesondere bei langen Operanden
  - Evtl. Geschwindigkeit, wenn der linke Operand aufwendig berechnet werden muss
    Beispiel:
    ```
    a[3*fkt(x)][x*z*fkt2(x,y,z)] =
                    a[3*fkt(x)][x*z*fkt2(x,y,z)] + 5;
    ```
  - `-->` `a[3*fkt(x)][x*z*fkt2(x,y,z)] += 5;`
    Bei dieser += Lösung muß die aufwendige Berechnung der Indices nur einmal erfolgen. Hierdurch kann sich evtl. ein Geschwindigkeitsvorteil ergeben

Zuweisung als Ausdruck – Ausdruck als Anweisung

- Unterschied in C/C++ zu allen anderen geläufigen Programmiersprachen: Zuweisungen sind Ausdrücke!
- Folgen:
  - Zuweisungen dürfen überall da verwendet werden, wo Ausdrücke des betreffenden Datentyps erlaubt sind !
    Beispiel: `if (y = f(x))` // Hier ist wirklich die Zuweisung gemeint!
- Aber auch
  - jeder Ausdruck wird durch Anhängen von ';' zu einer Anweisung:

    ```
    ausdruck ;
    ```

  - Also z.B. auch `2*3*x;` // Hier sinnlos, da der Wert nicht verwendet

    aber auch: `cout << "Text";`

    (<< ist eigentlich ein binärer Operator, der den Ausgabestream (hier `cout`) als Referenz zurückliefert)

(9b) Der Präprozessor

- **Prä - vor ; Prozessor - Verarbeitungsphase**
  - Bearbeitung des Quelltextes vor der eigentlichen Übersetzung und Codeerzeugung
  - Präprozessoranweisungen beginnen mit '#' und gelten ab dieser Zeile
- `#include <datei.h>` Einbinden von datei.h aus dem include-Verz.
- `#include "datei.h"` Einbinden von datei.h aus dem aktuellen Verz.
- `#define konstnam wert`
  - Konstantendefinition, z.B.: `#define MAXWERT 100`
  - Achtung! Kein abschließendes Semikolon! 'Wert' von `MAXWERT` ist der Rest der aktuellen Zeile; ein Semikolon würde daher mit zum Wert dazugehören
  - Achtung! Nur textuelle Definition der Konstante; Hier erfolgt weder Syntax- noch Typprüfung. Hier gemachte Fehler werden erst später an den verwendeten Stellen erkannt
- **Verwendung:**
  ```
  if (ergebnis < MAXWERT)      wird ersetzt zu ...
  if (ergebnis < 100)          (nur textuelle Ersetzung)
  ```
- **Durch Typisierung u. Syntaxprüfung unbedingt `const` bevorzugen!**

- **`#define`-Makros können auch parametrisiert werden**
  - `#define max(a,b) (((a)<(b)) ? (b) : (a))` // a und b sind textuell verwendete Parameter
- **Starke Klammerung sehr empfehlenswert, z.T. unbedingt notwendig!**
  - `#define sq(a) a*a` // `sq` soll das Quadrat einer übergebenen Zahl // bezeichnen
  - `#define incr(x) x+1` // `incr` soll die um 1 erhöhte Zahl liefern
  - Anwendungen:
  - `x = sq(d);` // noch problemlos
  - `x = sq(d+1);` // textuelle Ersetzung: `x = d+1 * d+1;`
  - `x = incr(d);` // wieder problemlos
  - `x = incr(d) * 2;` // textuelle Ersetzung: `x = d+1 * 2;`
- **Richtig wäre:**
  - `#define sq(a) ((a)*(a))`
  - `#define incr(a) ((a) + 1)`
- **Makros werden in C gern eingesetzt, um damit kleine Funktionen wie max zu erzeugen, ohne jedoch den Overhead zu haben, der mit jedem Funktionsaufruf einhergeht; in C++ besser: `inline`-Funktionen**

- **Beispiel für bedingte Compilierung: Bedingtes Setzen von Konstanten**

```
#define TEST 12
#if TEST == 10
     #define MAXWERT 99
#elif TEST == 12
     #define MAXWERT 101
#else
     #define MAXWERT 50
#endif
```

- **Anwendung: Bedingtes Compilieren**

```
#if TEST == 10
     cout << "Wert von 'ergebnis':" << ergebnis << endl;
#else
     cout << "Wert von 'ergebnis':" << ergebnis << endl;
     cout << "Wert von 'zaehler': " << zaehler << endl;
#endif
```

- **Auswirkung**
  - `if (ergebnis < MAXWERT)` **wird ersetzt durch**
  - `if (ergebnis < 101)`

- **Anwendungsfall: Bedingte Compilierung mit nur zwei Alternativen**

```
#define DEMO
  :
#ifdef DEMO
  #define VERSION 0.9
#else
  #define VERSION 1.0
#endif
```

```
......
void main()
{
   cout << "Programm SuperDuper, Vers. " <<
           VERSION << endl;
}
```

- **Häufige Anwendung: Einfügen von Codestücken für 'besondere' Fälle**

```
#ifdef DEBUG
    cerr << "Wert von 'ergebnis':" << ergebnis << endl;
#endif// Nachricht wird (nur) im 'DEBUG'-Modus ausgegeben!
```

- **`#pragma` - Compilerdirektiven, - Einstellungen**
  - **nicht festgelegt, compilerabhängig >>> Compiler-Handbücher**
- **`#error` Fehlertext**
  - **Ausgabe des Fehlertexts (während des Präprozessorlaufs!) und Abbruch des Compiliervorgangs (z.B. als Hinweis auf nicht mehr unterstützte Programmversionen)**

- Es ist der Übersicht sehr dienlich, wenn der Code auf mehrere Files verteilt wird
  - z.B. können die Klassendeklarationen in andere Files wandern wie die Implementierungen
- Nebeneffekt: Beschleunigung des Compiler-Vorganges
  - nur das, was sich geändert hat, muss neu compiliert werden
- Konventionen:
  - In Header-Files (.h) landet alles, was nicht direkt Code erzeugt
    - Deklarationen, Prototypen, Typnamen
    - Implementierungen von `inline`-Funktionen
    - globale Konstanten
    - Templates (s. später)
  - In Codefiles (.cpp, .cc etc.) landet der Rest
    - Klassen- und Funktions-Implementierungen
    - globale Objekte
- Code muss stets die Header `#include`n
  - Vorsicht bei mehrfachem `#include` auf das gleiche File!

- Beispiel Notenprogramm: Aufteilung auf 3 Dateien!
  - `schueler.h`:

```
#ifndef _SCHUELER_H
#define _SCHUELER_H

#include <iostream>
#include <string.h>

using namespace std;

class Schueler {
...
};

#endif
```

Verhinderung mehrfachen `#include`s

Deklaration der Klasse (eventuell mit inline-Implementierungen)

- **schueler.cpp:**

```
#include <iostream>
#include <string.h>
#include "schueler.h"

bool Schueler::init(char *n, int nt)
{
   if(nt<1 || nt>6)
      return false;   // Unsinn
…

}
…
```

Deklarationen werden aus Header-File gelesen

Implementierung der Klasse

- **Hauptprogramm (noten.cpp)**

```
#include <iostream>
#include <string.h>
#include "schueler.h"

int main() {
…
   Schueler *Klasse = new Schueler[anzahl];
…
   for(i=0; i<anzahl; i++)
      Klasse[i].init(name,note);
…

}
```

Für die Compilierung genügt das Header-File!

- **Ablauf beim Compilieren: unterschiedliche Möglichkeiten**

- **Modell "make":**
  - jedes Quellcodefile (.cc, .cpp) wird getrennt übersetzt
  - danach werden die erhaltenen Objektfiles "gelinkt" und ein `.exe` erzeugt
  - Vorteil: es wird nur das compiliert, was sich wirklich geändert hat
  - Nachteil: gegenseitige Abhängigkeiten müssen genau spezifiziert werden
  - UNIX-Tool "`make`" ist das Standardwerkzeug, viele Windows-Compiler-IDEs können etwas Ähnliches
- **Simples Modell:**
  - ein einziger Compiler-Aufruf compiliert und linkt alle Quellfiles
  - jedes Mal wieder!
  - Vorteil: einfach
  - Nachteil: bei großen Projekten kann die Compilierzeit explodieren, auch wenn man nur kleine Änderungen gemacht hat

- **In Header-Files werden nicht nur Klassen, sondern auch Funktionen deklariert! Beispiel:**

`math.h`:

```
...
double sin(double);
...
```

  - Damit ist dem Compiler der Name und die Signatur der Funktion bekannt
  - Bestimmte Fehler (falsche Zahl von Argumenten, falsche Typen) können so bereits vom Compiler bemerkt werden
  - Die eigentliche Implementierung der Funktion ist bereits vorcompiliert in einem Binärfile oder einer Bibliothek versteckt
    - Linker setzt nur die compilierten Stücke zusammen

- **Deklarationen ("Prototypen") sind auch bei Vorwärtsreferenzen im gleichen Quellfile notwendig:**

```
void sortiere_int(int *, int); // Funktionsprototyp

int main() {
…
    int *feld = new int[n];
…
    sortiere_int(feld, n);
…
}

void sortiere_int(int *f, int l) {
…
}
```

Vorwärtsreferenz wird durch den Prototypen möglich

- **Notenprogramm mit Dateien**
  - **Schueler-Klasse, Deklaration**

```
class Schueler {
 private:
   char *name;
   int note;
 public:
   Schueler();
   Schueler(const char*, const int);
   ~Schueler();
   bool init(const char*, const int);// Setzen der Daten
                                     // von außen, false bei
                                     // Fehler
   int getnote() const;              // Note zurückgeben
   char *getname() const;            // Namen zurückgeben
   ostream& speichern(ostream&);
   istream& laden(istream&);
};
```

Notenprogramm mit Dateien (Forts.)

- Schueler-Klasse, Implementierung

```
bool Schueler::init(const char* nm, const int nt) {
   if(nt<1 || nt>6)
      return false;
   note=nt;
   delete[] name;
   name = new char[strlen(nm)+1];
   strcpy(name,nm);
   return true;
}

Schueler::Schueler() {
   note=1;
   name=NULL;
}
```


Notenprogramm mit Dateien (Forts.)

- Schueler-Klasse, Implementierung

```
Schueler::Schueler(const char*nm, const int nt) {
       if(!init(nm, nt))
       cerr << "Ungueltige Daten!" << endl;
}

Schueler::~Schueler() {
       delete[] name;
}

int Schueler::getnote() const { return note; }

char * Schueler::getname() const { return name; }

ostream& Schueler::speichern(ostream& os) {
   os << name << " " << note << endl;
   return os;
}
```

Notenprogramm mit Dateien (Forts.)

- Schueler-Klasse, Implementierung

```
istream& Schueler::laden(istream& is) {
   char vn[100], nn[50];
   int nt;
   is >> vn >> nn >> nt;
   strcat(vn, " ");
   strcat(vn,nn);
   if(!init(vn, nt));
       cerr << "Ungueltige Daten!" << endl;
   return is;
}
```


Notenprogramm mit Dateien (Forts.)

- Hauptprogramm (speichern)

```
int main() {
   ofstream out;
   int i,n;
   char name[100];
   int note;

   cout << "Dateiname? "; cin >> name;
   out.open(name,ios_base::app);
   if(!out) {
      cerr << "Fehler beim Oeffnen!" << endl;
      return 1;
   }

   cout << "Anzahl Schueler? "; cin >> n;
   Schueler* Klasse = new Schueler[n];
```

Notenprogramm mit Dateien (Forts.)

- Hauptprogramm (speichern)

```
   for(i=0; i<n; i++) {
      cin.ignore(1000,'\n');
      cout << "Name Schueler " << (i+1) << " ? ";
      cin.getline(name, 99);
      cout << "Note Schueler " << (i+1) << " ? ";
      cin >> note;
      Klasse[i].init(name, note);
      Klasse[i].speichern(out);
   }

   out.close();

   return 0;
}
```


Notenprogramm mit Dateien (Forts.)

- Hauptprogramm (laden)

```
int main() {
   ifstream in;
   char name[100];

   cout << "Dateiname? "; cin >> name;
   in.open(name);
   if(!in) {
      cerr << "Fehler beim Oeffnen!" << endl;
      return 1;
   }
   Schueler s;
   while(s.laden(in)) {
      cout << s.getname() << " hat Note " <<
              s.getnote() << endl;
   }
   in.close();
   return 0;
}
```

Verchlüsselungsprogramm

- **Ziesetzung**
  - Es soll eine gegebene Datei eingelesen und nach einem bestimmten Algorithmus verschlüsselt unter anderem Namen wieder gespeichert werden.
  - Die Verschlüsselungsroutine soll als C++-Funktion realisiert sein, damit das Verfahren leicht geändert werden kann.
  - Die Verschlüsselungsfunktion soll immer eine gesamte Textzeile verschlüsseln
  - Algorithmus (zunächst): "ROT13"
- **ROT13: Jeder Buchstabe wird durch den Buchstaben ersetzt, der im Alphabet 13 Stellen weiter hinten (oder vorne) steht. Dabei wird das Alphabet als geschlossene Buchstabenkette gesehen [ABC..YZABC…]. Nicht-Buchstaben werden nicht verschlüsselt.**

  A ⇒ N B ⇒ O C ⇒ P …. Y ⇒ L Z ⇒ M 1 ⇒ 1 etc.

- **Beispiel:**

  "Geheime Nachricht" ⇒ "Trurvzr Anpuevpug"

Verschlüsselungsprogramm

- **Welche Komponenten werden benötigt?**
  - Übergabe des Klar- und Geheimtextes an die Verschlüsselungsfunktion mittels Zeigern
  - Überprüfung, ob ein Zeichen ein Buchstabe ist

```
#include <ctype.h>

int isupper(int c);
int islower(int c);
int isalpha(int c);
```

Diese Funktionen geben einen Wert ≠0 zurück, falls das übergebene Zeichen ein großer bzw. kleiner bzw. überhaupt ein Buchstabe ist.

  - Textdatei zeilenweise lesen: `istream.getline();`
  - Zeilenweise schreiben: `<<`

Verschlüsselungsprogramm: Verschlüsselungsfunktion

```
void encrypt_rot13(char *input, char *output)
{
   unsigned char c;

   do
   {
      c=*input;
      if(islower(c))
      {
         c = c+13;
         if(c > 'z')
            c = c-26;
      }
      else if(isupper(c))
      {
         c = c+13;
         if(c > 'Z')
            c = c-26;
      }
      *output=c;
      output++;
      input++;
   } while(*(input-1));
}
```


Verschlüsselungsprogramm: main()

```
int main()
{
   char text[200],vtext[200],infile[20],outfile[20];
   ifstream fdi; ofstream fdo;

   cout << "Klartextfile: "; cin >> infile;
   cout << "Geheimtextfile: "; cin >> outfile;
   fdi.open(infile); fdo.open(outfile);
   if(!fdi || !fdo)
   {
      cerr << "Fehler!" << endl;
      return 1;
   }
```

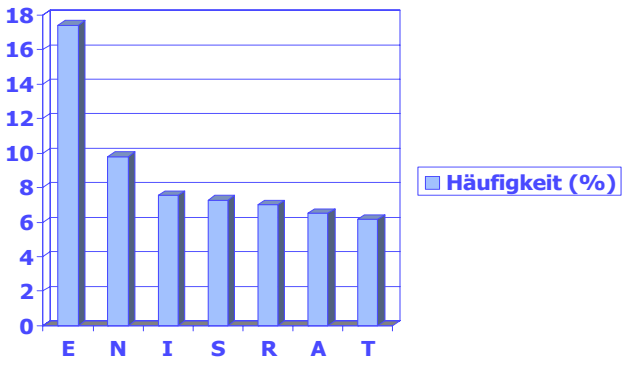
Verschlüsselungsprogramm main(), Forts.

```
   while(!fdi.getline(text, 200))
   {
      encrypt_rot13(text,vtext);
      fdo << vtext << endl;
   }

   fdi.close();
   fdo.close();
   return 0;
}
```


Verschlüsselungsprogramm

- **ROTn-Verschlüsselung ist leicht zu knacken**
  - **Häufigkeitsverteilung der Buchstaben verrät die verwendete Abbildung**
  - **Auch zufälliges Verwürfeln statt einfachem Rotieren ist nicht besser**
  - **Können Sie ein C-Programm schreiben, das beim Knacken eines solchen Codes hilft?**



Was wirklich wichtig ist, Teil 3

- **Dynamische Speicherverwaltung**
  - Operator `new` erzeugt ("allokiert") Objekt oder Feld von Objekten, Rückgabe eines Zeigers
  - Operator `delete` zerstört Objekt (Speicherfreigabe)
  - Operator `delete[]` zerstört Feld von Objekten

```
int *a = new int;             // einzelner int
bruch *b = new bruch(3,4);    // explizite Initialisierung
char *p = new char[x];        // Feld mit x Elementen

delete a; delete b;           // Objektfreigabe
delete[] p;                   // Feldfreigabe
```

  - Bei der Allokation von Objektfeldern wird der Default-Konstruktor für jedes Objekt aufgerufen
    - Fehler, falls kein solcher existiert!

Was wirklich wichtig ist, Teil 3

- **Klassen** sind eine starke Erweiterung des Strukturbegriffes von C
  - sie bestehen aus Daten und Funktionen (Member)
- **Objekte** sind konkrete Instanzen von Klassen
- **Konstruktoren**: spezielle Memberfunktionen, bei Instanziierung gerufen
  - Überladung möglich
  - Felder von Objekten: Default-Konstruktor!
  - Spezielle Behandlung konstanter Memberdaten
- **Destruktor**: spezielle Memberfunktion, bei Zerstörung gerufen
- **Konstante Memberfunktionen**: kein schreibender Zugriff auf Objektdaten
- **Klassenvariablen**, statische Member: gehören allen Instanzen der Klasse ("klassenglobal"?)
  - globale Definition erforderlich
- **this**: Zeiger auf "dieses" Objekt

Was wirklich wichtig ist, Teil 3

- Eine Klasse ist zunächst eine Erweiterung des Strukturbegriffes
  - Klassen können neben Daten auch Funktionen enthalten, die auf die Daten wirken:

```
class bruch {
 private:
   int z,n;
 public:
   double dezim();
};
```

  - Sichtbarkeit der Klassenmember von außen wird durch `private` bzw. `public` (bzw. `protected`) geregelt
  - Implementierung von Memberfunktionen erfolgt üblicherweise außerhalb der Klassendeklaration

```
double bruch::dezim() { return (double)z/n; }
```

  - Ein Objekt (Klasseninstanz) wird wie eine Variable deklariert:

```
bruch b;
```

- **Spezielle Memberfunktionen**
  - Konstruktoren: zur Initialisierung eines Objekts bei der Instanziierung
    - Name = Klassenname
    - Überladung erlaubt
    - Keine Rückgaben
    - Initialisierung von (auch konstanten) Memberdaten im Kopf der Konstruktor-Implementierung möglich
    - Default-Konstruktor: Konstruktor, der ohne Argumente gerufen werden kann

```
bruch::bruch() : z(1) , n(1) { }

bruch::bruch(int zhl) : z(zhl) , n(1) { }

bruch::bruch(int zhl, int nenn) { z = zhl; n = nenn; }
```

  - Destruktor: zur Zerstörung eines Objektes am Ende seiner Lebensdauer
    - Name = ~Klassenname
    - Keine Argumente, keine Rückgaben
    - Nur wirklich notwendig, wenn "aufgeräumt" werden muss (Speicherfreigabe etc.)

```
iarray::~iarray() { delete[] start; }
```


- **Konstante Memberfunktionen**
  - Deklaration und Implemetierung mit nachgestelltem `const`

```
class bruch {
…
   double zaehler() const;
};

double bruch::zaehler() const { return z; }
```

  - Keine Veränderung der Memberdaten durch diese Funktionen erlaubt!
  - Zweck
    - Verhinderung von Programmierfehlern
    - Generierung von schnellerem Code durch den Compiler

Ein-/Ausgabe mit Dateien

- Objekte der Klassen

```
ofstream        (schreiben)
ifstream        (lesen)
fstream         (lesen und/oder schreiben)
```

werden durch Konstruktor bzw. Memberfunktion open() mit Dateien assoziiert:

```
ofstream d("datei1.dat");
ofstream e("datei2.dat", ios_base::app);
ifstream f;
f.open("datei3.dat");
```

- Schließen einer Datei mit Memberfunktion close()

```
f.close();
```


Ein-/Ausgabe mit Dateien

- Formatierte Ein- und Ausgabe auf geöffneten Stream mit den gewohnten Schiebeoperatoren nebst Zubehör

```
ofstream g("out.dat");
g << i << " " << hex << j << endl;
```

- Unformatierte Ein-/Ausgabe durch Memberfunktionen read() bzw. write()

```
int feld[100];
…
g.write((char*)feld, sizeof(int)*100);
```

- Überprüfung des Fehlerstatus mit Memberfunktionen `good()` bzw. `fail()` bzw. `eof()`

```
while(g.good()) { … }
```

- Test auf FAIL-Zustand auch einfach durch !-Operator: `if(!g) { … }`

Weitere Aufgaben und Beispiele

Spielereien mit Zeichenketten (Klausur 06.07.2000, Nr. 1.2)

- Gegeben sei ein char-Feld `zk[]`, das eine Zeichenkette unbekannter Länge enthält. Schreiben Sie eine Funktion, die die Zeichenkette rückwärts auf dem Bildschirm ausgibt. Dabei werden alle Vokale (egal ob Groß- oder Kleinschreibung) ausgelassen.

- Welche Elemente weden benötigt?
  - Bestimmung des Endes einer Zeichenkette:
    `int strlen(const char*);`
  - Durchlaufen der Zeichenkette von Ende bis Anfang: for-Schleife
  - Prüfung eines Buchstabens auf Vokal: switch-Konstrukt
  - Klein- in Großschreibung ändern: `int toupper(char);`
- Prototyp:

```
void rueckwaerts(char zk[]);
```

Spielereien mit Zeichenketten

```
#include <string.h> // fuer strlen()
#include <ctype.h>  // fuer toupper()
#include <iostream>
using namespace std;
void rueckwaerts(char zk[])
{
   int last,i;
   last = strlen(zk)-1;    // letztes Zeichen
   for(i=last; i>=0; i--)
   {
      switch(toupper(zk[i]))      // in Großbuchstaben wandeln
      {
      case 'A':
      case 'E':
      case 'I':
      case 'O':
      case 'U': break;
      default: cout.put(zk[i]);
      }
   }
}
```


Reihenbildung (Klausur 06.07.2000, Nr. 1.3)

- **Schreiben Sie eine Funktion f(x), die für eine als Parameter gegebene double-Zahl x folgende Reihe berechnet und als Returnwert zurückgibt. Die Funktion darf die pow()-Funktion nicht verwenden!**

  $f(x) = 1.0/x - 2.0/x^3 + 1.0/x^5 - 2.0/x^7 + \ldots.$

  Die Berechnung soll angebrochen werden, sobald der Betrag eines Summanden den Wert 10^{-9} unterschreitet.
- **Welche Elemente weden benötigt?**
  - **Schleife über die Glieder der Reihe**
  - **jedes 2. Mal ändert sich der Vorfaktor**
  - **jedes Mal wächst der Exponent um 2**
  - **Absolutwert mit `fabs()`**
- **Prototyp:**

```
double f(double x);
```

Reihenbildung

```
#include <math.h>   // fuer fabs()
#define EPS 1.0e-9  // Schwelle
double f(double x)
{
   double invpot,summand,summe=0.0;
   int i=1,ende=0;
   invpot = x;
   do{
      invpot /= x*x;        // jedes Mal ein x*x mehr im Nenner
      summand = invpot*( i&1 ? 1:-2); // alternierender Faktor
      if(fabs(summand) < EPS)          // Abbruchbedingung
         ende=1;
      else
      {
         summe += summand;
         i++;
      }
   } while(!ende);
   return(summe);
}
```


Messwerte auswerten (Klausur 06.07.2002, Nr. 3)

- Ein Wissenschaftler hat seine Messergebnisse, die pos. und neg. double-Zahlen darstellen, in eine Textdatei geschrieben. Dabei sind die Zahlen durch mind. eine Leerstelle getrennt; Leerzeilen können vorkommen. Die Anzahl der Zahlen ist so groß, dass sie nicht im Hauptspeicher Platz haben. Schreiben Sie eine Funktion, die als einen der Parameter den Namen der Datei hat und die mit einem einmaligen Durchlesen der Datei folgende Größen bestimmt und als weitere Parameter an die Funktion zurück gibt:
  - G: größter neg. Messwert; 0 falls kein neg. Wert existiert
  - K: kleinster pos. Messwert; 0 falls kein pos. Wert existiert
  - M: Mittelwert aller Messwerte
  - Q: Summe der Quadrate der Messwerte
- Returnwert der Funktion
  - 1: alles lief Ok
  - 0: Datei nicht gefunden
  - -1:Datei vorhanden aber leer

Messwerte auswerten

- **Welche Elemente werden benötigt?**
  - einlesen jeder Zahl mit >> ...
  - Überprüfung auf EOF
  - Übergabe der Werte G,K,M,Q mittels Referenzen an die Funktion
  - Update des Mittelwertes M?

```
int einlesen(char *filename, double &G, double &K, double &M,
                                                    double &Q)
{
   ifstream fd(filename);
   double neu;
   int anzahl=0, ret;
   G=K=Q=M=0;

   if(!fd)
   {
      cerr << "Fehler! Datei " << filename << " nicht gefunden";
      return 0;
   }
```


Masswerte auswerten

```
fd >> neu;      // Vorauslesen!
while(!fd.eof())
   {
      anzahl++;
      Q += neu*neu;
      // Update Mittelwert
      M = (1.0/anzahl) * (M*(anzahl-1) + neu);
      if(neu < 0 && neu > G || neu < 0 && G==0.0)// Achtung bei G==0
         G = neu;
      if(neu > 0 && neu < K || neu > 0 && K==0.0)// Achtung bei K==0
         K = neu;
      fd >> neu;
   }
   if(!anzahl)
   {
      cerr << "Datei " << filename << " ist leer" << endl;
      ret = -1;
   }
   else ret=1;
   fd.close();
   return ret;
}
```

Beispiel: Dodon, der Märchenkönig (zu Aufgabe 20)

- Dodon, der Märchenkönig, nahm bei einem Feldzug 100 Feinde gefangen, die er in 100 Einzelzellen steckte. An seinem Geburtstag sollten einige freigelassen werden, und zwar nach einem ganz eigenartigen Verfahren (vom Hofmathematiker ausgedacht).
  Dieses Verfahren arbeitet mit mehreren Durchgängen, wobei in jedem Durchgang für jede betroffene Zellentür folgender Zustandswechsel durchgeführt wird:
  - Ist entsprechende Zellentür zu diesem Zeitpunkt offen, wird sie geschlossen
  - Ist entsprechende Zellentür zu diesem Zeitpunkt geschlossen, wird sie geöffnet.
  - Zunächst sind alle Zellentüren geschlossen.
    1) Im ersten Durchgang ist dann jede Tür vom Zustandswechsel betroffen.
    2) Im zweiten Durchgang ist nur jede zweite Tür vom Zustandswechsel betroffen.
    3) Im dritten Durchgang ist nur jede dritte Tür vom Zustandswechsel betroffen.
    ... und so geht es bis zum 100. Durchgang weiter.
- Die Frage ist nun: welche Türen waren am Geburtstag des Königs offen?

Dodon, der Märchenkönig

- "Luxusversion" des endgültigen Programmes soll wahlweise auch die einzelnen Schritte ausgeben
- Verwende bedingte Compilierung, um nach jedem Durchgang – falls gewünscht – alle Türen zu "visualisieren"

```
#include <iostream>
using namespace std;
#define TUEREN 100
#define DEBUG
#define WAIT
int main()
{
   char t[TUEREN];
   int i,j,offen;
   for(i=0; i<TUEREN; i++)
      t[i]=0;
   offen=0;
   for(i=0; i<TUEREN; i++)
   {
      for(j=i; j<TUEREN; j+=(i+1))
         t[j]=~t[j];
#ifdef DEBUG
      for(j=0; j<TUEREN; j++)
         cout << t[j]?'O':'Z';
      cout << endl;
#ifdef WAIT
      cin.get();
#endif
#endif
      if(t[i]) offen++;
   }
   cout << "Es sind " << offen
    << " Tueren offen." << endl;
   return 0;
}
```

Beispiel: Matrizen einlesen

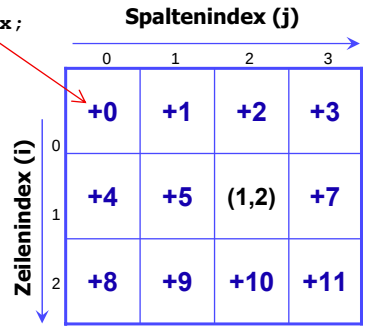
- Wie liest man die Einträge einer Matrix ein, deren Größe nicht vorher bekannt ist?
  - Deklaration einer "Maximalmatrix" ist unschön
  - Also: Speicher dynamisch allokieren!
- Annahme: Zeilen- und Spaltenzahl wurden bereits eingelesen

```
int zeilen, spalten;
double *matrix;
…
matrix = new double[zeilen*spalten];
```

- Wie erfolgt nun der Zugriff auf die Matrixelemente?
  - `matrix[i][j]` geht nicht, da `matrix[]` jetzt einfach ein Feld von `double`s ist
  - Also: Zeilen- und Spaltenindex müssen in 1D-Index umgerechnet werden!
  - Umrechnung kann in z.B. in parametrisierter Konstante (Makro) erfolgen

Matrizen einlesen

`double *matrix;`

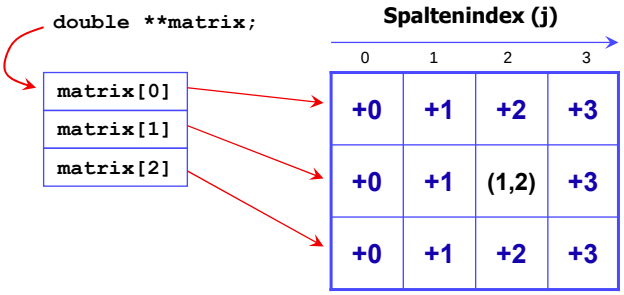


Matrixeinträge sind mit aufsteigendem Spaltenindex nacheinander zeilenweise im Speicher abgelegt

Eintrag (i,j) hat also den Offset

(i*spalten+j)

- Zugriff auf Matrixelement (i,j) erfolgt also mit `matrix[i*spalten+j]`
- Alternativ: `*(matrix + i*spalten+j)`
- Beachte: Alle Indizes sind 0-basiert!
- Dies ist etwas unübersichtlich – geht es nicht doch mit zwei Indizes?
  - `matrix[i]` müsste ein Zeiger sein, zu dem dann `j` addiert wird
  - Die Zeiger im Feld `matrix[]` zeigen dann auf die Zeilen der Matrix



Zugriff auf Element (i,j) mittels

`matrix[i][j]`

- Nachteile der zweiten Methode
  - Es wird zusätzlicher Platz für das Zeigerfeld benötigt
  - Das Allokieren des Speichers gestaltet sich etwas aufwendiger – jede Zeile muss getrennt allokiert werden:

```
int zeilen,spalten,i;
double **matrix;
...
matrix = new double*[zeilen];
for(i=0; i<zeilen; i++)
   matrix[i] = new double[spalten];
```

Matrizen einlesen

- Wie übergibt man eine Matrix an eine Funktion?

1. Bei statischer Allokation (feste Dimensionen)

```
funktion(double matrix[][<spalten>]);
```

2. Bei dynamischer Allokation eines eindimensionalen Feldes

```
funktion(double *matrix, int spalten);
```

3. Bei dynamsicher Allokation jeder Zeile getrennt

```
funktion(double *matrix[]);     // oder
funktion(double **matrix);
```


Aufgabe 36

- **Aufgabe 36 (ROTn-Verfahren):**
  Schreiben Sie ein Programm, das einen Text ("Klartext"), der in einer Datei gespeichert ist, mittels des ROTn-Verfahrens verschlüsselt und das Resultat ("Geheimtext") in einer zweiten Datei abspeichert. Es sollen ausschließlich Buchstaben, also keine Ziffern und Sonderzeichen, verschlüsselt werden. Der Wert n wird vom Benutzer erfragt. Das Programm soll auch den umgekehrten Weg, d.h. das Entschlüsseln eines mit ROTn behandelten Textes ermöglichen.

- Was wird benötigt?
  - Lesen aus der Datei: zeichenweise (istream::get(char&))
  - Schreiben dito (ostream::put(char))
  - Funktion, die ein Zeichen verschlüsselt (Key aus 0…25)
  - Entschlüsselung = Verschlüsselung mit (26-Key)%26

Aufgabe 36 (ROTn)

```
#include <iostream>
#include <fstream>
#include <string.h>
#include <ctype.h>  /* für isupper() und islower() etc. */
using namespace std;

void encrypt_rotn(char &x, unsigned int key) {
   unsigned char c = x;
   if(isupper(c)) {
      c=c+key;
      if(c > 'Z')
         c=c-26;
   }
   else if(islower(c)) {
      c=c+key;
      if(c > 'z')
         c=c-26;
   }
   x=c;
}
```

warum `unsigned`?

Konversion nur bei Buchstaben

ringförmiges Schließen des Alphabets

Aufgabe 36 (ROTn)

```
int main()
{
   char file[100],c;
   int i;
   unsigned int key;

   cout << "Filename input? ";
   cin >> file;

   ifstream f1(file);

   if(!f1) {
      cerr << "Fehler beim Oeffnen von " << file << endl;
      return 1;
   }

   cout << "Filename output? ";
   cin >> file;

   ofstream f2(file);
```

Aufgabe 36 (ROTn)

```
   if(!f2) {
      cerr << "Error opening " << file << endl;
      return 1;
   }

   cout << "Key? ";
   cin >> key;
   if(key>25)
      key %= 26;

   cin.ignore(1000,'\n');
   c=0;
   cout << "[V]er- oder [E]ntschluesseln? ";
   while(c!='V' && c!='E')
      cin.get(c);

   if(c=='E')
       key=(26-key)%26;
```

Sicherheitsmaßnahme

Wiederholung, bis korrekte Eingabe erfolgt

Entschlüsseln = Verschlüsseln mit komplementärem Key

Aufgabe 36 (ROTn)

Vorauslesen

```
   f1.get(c);
   while(f1.good() && f2.good()) {
      encrypt_rotn(c,key);
      f2.put(c);
      f1.get(c);
   }

   f1.close();
   f2.close();
}
```

- **Aufgabe 37 (ROTn brechen):**
  Schreiben Sie nun ein Programm, das den in Aufgabe 36 geschriebenen Geheimtext ohne Kenntnis von n entschlüsselt, indem es ihn einer Häufigkeitsanalyse unterzieht. Zählen Sie dazu, wie oft jeder einzelne Buchstabe (Groß- und Kleinschreibung ignorieren*) im Geheimtext vorkommt. Der häufigste Buchstabe entspricht dann höchstwahrscheinlich im Klartext dem "E", falls es sich um einen deutschen Text handelt. Errechnen Sie daraus die Zahl n und entschlüsseln Sie den Geheimtext!

  *** Hinweis: Verwenden Sie die Bibliotheksfunktion `toupper()`, um einen Buchstaben in einen Großbuchstaben zu wandeln.**

- **Was wird benötigt?**
  - **Datei muss zweimal gelesen werden, einmal zum Ermitteln der Häufigkeiten und einmal zum Entschlüsseln**
  - **Histogramm mit Häufigkeiten in int-Feld hist[26] ablegen!**


```
#include <iostream>
#include <fstream>
#include <string.h>
#include <ctype.h>
using namespace std;

void decrypt_rotn(char &x, unsigned int key) {
   unsigned char c=x;
   key=(26-key)%26;
   if(isupper(c)) {
      c=c+key;
      if(c > 'Z')
      c=c-26;
   }
   else if(islower(c)) {
      c=c+key;
      if(c > 'z')
      c=c-26;
   }
   x=c;
}
```

Entschlüsseln = Verschlüsseln mit komplementärem Key

Aufgabe 37 (ROTn brechen)

```
int main()
{
   char file[100];

   int i, key;
   char c;

   cout << "Filename input? ";
   cin >> file;

   ifstream f1(file);

   if(!f1) {
      cerr << "Fehler beim Oeffnen von " << file << endl;
      return 1;
   }
```


Aufgabe 37 (ROTn brechen)

```
   int hist[26];
   for(i=0; i<26; i++)
      hist[i]=0;

   f1.get(c);
   while(f1.good()) {
      if(isalpha(c)) {
         c=toupper(c);
         hist[c-'A']++;
      }
      f1.get(c);
   }

   char maxi=0;
   int h=0;
   for(char i=0; i<26; i++)
      if(h<hist[i]) {
         h=hist[i];
         maxi=(char)i;
      }
```

Vorbelegung mit Nullen – nicht vergessen!

nur Buchstaben zählen

Histogramm mit Häufigkeiten der Buchstaben 'A'…'Z'

Position `maxi` des häufigsten Buchstabens bestimmen (0…25)

Aufgabe 37 (ROTn brechen)

```
    key=maxi-4;
    if(key<0)
       key=key+26;

    cout << "Haeufigster Buchstabe: "
         << (char)('A'+maxi) << " -> Schluessel ist "
         << key << endl;

    f1.clear();
    f1.seekg(0);
    f1.get(c);
    while(f1.good()) {
       decrypt_rotn(c,key);
       cout.put(c);
       f1.get(c);
    }

    f1.close();
}
```

Key = Position des häufigsten Buchstabens relativ zum 'E' (positiv)

Fehlerbits (`ios_base::failbit` und `ios_base::eofbit`) zurücksetzen

Lesen wieder beim Anfang beginnen

Klausurrichtlinien

- **Richtlinien für die Klausur**
  - **ZUERST DIE KOMPLETTE AUFGABE VOLLSTÄNDIG LESEN!**
  - **Keine globalen Variablen!**
  - **Alle Klassen mit getrennter Deklaration und Definition schreiben**
  - **Speicherplatzreservierungen für Felder und Zeichenketten (außer Zeichenkettenkonstanten) sind innerhalb der Klasse und dynamisch zu gestalten (wenn nicht anders angegeben)**
  - **Der Test auf unzureichenden Platz auf dem Heap (d.h. für dynamisch allokierte Objekte) kann entfallen**
  - **Jede sinnvolle Möglichkeit, Elemente als `const` zu deklarieren, ist zu nutzen**
  - **Alle Datenelemente sind so geschützt wie möglich zu formulieren, also `private` oder `protected`**