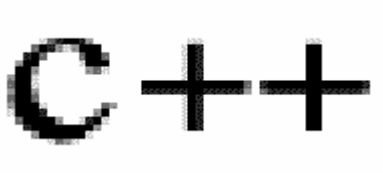
Programmieren in C und C++



Universität Regensburg
Fakultät Physik

Januar 2006

Vorwort

Was will der Kurs?

Dieser 2-wöchige Blockkurs *Programmieren in C und C++* soll alle Teilnehmer(innen) möglichst schnell dazu bringen, eigene kleine Computerprogramme zu schreiben. In der ersten Woche werden die Grundlagen vermittelt, die dann intensiv eingeübt werden. Die zweite Woche behandelt fortgeschrittenere Konzepte wie z.B. Pointer. Am Ende **werden sie sicher kein professionelles Software-Engineering beherrschen.** Der Kurs bietet jedoch alle Voraussetzungen dafür, wenn sie sich in dieser Richtung weiterentwickeln wollen.

Die Veranstaltung ist auch ***kein* Kurs in objektorientierter Programmierung**! Natürlich gehen wir am Ende auf die Grundlagen dieser Technik ein. Der Schwerpunkt liegt aber auf klassischer, sog. prozeduraler Programmierung. Wir haben die Programmiersprache C gewählt, weil sie die größte Verbreitung hat und auf wirklich alle Probleme anwendbar ist. Praktisch alle Elemente der Sprache C werden hier besprochen. Wer diesen Kurs erfolgreich absolviert hat, wird sich sehr leicht in andere Programmiersprachen einarbeiten können, sei es in *Java*, in *Fortran* speziell im naturwissenschaftlich/technischen Bereich, oder auch in Skriptsprachen wie *Perl, Python* oder *php.*

> C++ ergänzt C um viele nützliche Elemente, nicht nur um Objektorientierung. Solche Erweiterungen sind an allen Stellen des Kurses eingearbeitet; sie werden aber immer explizit durch senkrechte Balken am Rand gekennzeichnet.

Wir haben sehr viele Übungsaufgaben zusammengestellt, die zur Vertiefung des Stoffes dienen und in das algorithmische Denken einführen sollen. Das Aufgabenblatt bekommen sie am ersten Kurstag. Diverse Aufgaben stammen aus dem Bereich der **Schulmathematik**, höhere Mathematikkenntnisse werden nicht vorausgesetzt. Aber auch aus anderen Bereichen sind genügend Aufgaben vorhanden. Sie können bei uns also auch gut Programmieren lernen, wenn ihnen Mathematik gar nicht liegt.

Der Kurs ist **für absolute Computer-Laien nicht geeignet**. Grundlegende Kenntnisse über den Umgang mit Rechnern werden vorausgesetzt. Minimale Programmiererfahrung in irgendeiner Sprache sollte vorhanden sein.

Im Übrigen arbeiten wir mit einer möglichst einfachen, heute unprofessionellen **Entwicklungsumgebung**. Sie wollen ja zunächst die Sprache lernen, nicht ein komplexes Entwicklungstool; beides auf einmal ist in einem 2-Wochen-Kurs nicht möglich.

Wozu überhaupt noch programmieren?

Viele Probleme sind heutzutage mit fertiger, integrierter Software zu lösen, z.B. viele mathematische Fragestellungen mit *Maple, Mathematica* oder *Matlab*. Wo immer möglich und sinnvoll (z.B. bei vielen Übungsaufgaben im Studium), sollten diese Pakete benutzt werden. Vieles muss man aber nach wie vor programmieren. Dazu gehören Dinge wie

- Simulationen naturwissenschaftlicher Phänomene, insbesondere, wenn sie die Architektur von Hochleistungsrechnern effizient nutzen sollen
- Microcontroller für Steuerungen, z.B. im Automobil (kein Platz für integrierte Software)

- Echtzeit-Systeme, die sehr schnelle Systemantworten verlangen
- Systemmanagement von Servern
- innovative Lösungen wie z.B. interaktive Webangebote mit Datenbankanbindung

Als Nebeneffekt lernt man außerdem viel über die Arbeitsweise von Computern.

Warum C, warum nicht gleich 'richtig' C++ oder Java?

- C ist *die* universelle Sprache seit über 20 Jahren, zur Entwicklung von Betriebssystemen, für Steuerungen usw. Sie besitzt weltweit **die größte Verbreitung**. **Java** holt inzwischen auf, ist aber bei weitem nicht so breit im Einsatz wie C. Der Java-Code ist oft noch zu langsam und zu ineffizient, die Entwicklungstools sind i.d.R. sehr komplex und träge.

- **C ist sehr effizient**, da es recht maschinennah ist, aber trotzdem unabhängig von der speziellen Rechnerarchitektur. Dies gilt insbesondere dann, wenn man sich – wie wir im Kurs – an die **ANSI-Norm** hält. C-Programme laufen dann auf einem 2€ teuren Waschmaschinen-Controller ebenso wie auf einem 50 Millionen Euro teuren Höchstleistungsrechner. C besitzt viele Operatoren, daher sind sehr kompakte Programme möglich. Modularität, d.h. Aufteilung eines Programms in einzelne Funktionen, wird gut unterstützt.

- Allerdings gibt es **einige Nachteile**: es ist syntaktisch viel erlaubt (z.B. Feldgrenzen-überschreitung, keine strenge Typenprüfung), was zum Totalabsturz des Rechners führen kann; außerdem sind absolut unleserliche und unwartbare Programme durchaus möglich. Im Kurs gibt's diesbezüglich immer wieder eindeutige Warnungen.

- Wir wollen schnell kleine Probleme lösen können. Dazu ist echtes **objektorientiertes Programmieren** viel **zu aufwendig**. Daher fangen wir mit klassischem Programmieren in C mit bequemen C++-Erweiterungen an. Außerdem enthält C++ die Sprache C komplett.

- Selbst fertige Software, z.B. zum interaktiven Erstellen von graphischen Oberflächen oder auch von Messprogrammen erzeugt sehr häufig C-Code, den man per Hand um die eigentlichen Aktionen erweitern muss.

Weitergehende Literatur und Übungsaufgaben

Dazu bitte auf der Homepage dieses Kurses nachschauen:
http://www.physik.uni-regensburg.de/studium/edverg/ckurs/

Dank

Den ursprünglichen Autoren diese Skriptes, **Henrik Schachner, Gerald Schupfner, Burkard Wiesmann, Florian Chmela** danke ich ganz herzlich, ebenso **Johannes Bentner, Anja Ebersbach und besonders Andreas Lassl** für die Mitarbeit an den Neuauflagen. Für Hinweise auf die unvermeidlichen Druckfehler und für jegliche sonstige Resonanz bin ich sehr dankbar! Meine Mailadresse: *fritz.wuensch@physik.uni-regensburg.de*

Regensburg, im Januar 2006 *Fritz Wünsch*

Kap 1: Einführung

1.1 Entstehung von C und C++

ALGOL 60 (1960): erste Programmiersprache mit Blockstruktur, Möglichkeit der Rekursion, höhere Kontrollstrukturen

BCPL ("Basic Combined Programming Language") Cambridge, 1967

B Ken Thompson, Bell Laboratories, 1970; frühe Implementierung von UNIX

C Dennis Ritchie, Bell Laboratories, 1972; Implementierung von UNIX, zugleich Standard-Tool zum Erweitern des Befehlsumfangs; 1978 erscheint *The C Programming Language* (Kernighan, Ritchie) und wird zum *de facto* - Standard.

ANSI C („*American National Standards Institute*")
Festsetzung des Standards X3.159-1989 (1983-1988). ISO Standard ISO/IEC 9899:1990 (1990), Technical Corrigendum 1 (1994), Normative Appendum 1 (1994)

C++ Bjarne Stroustrup, seit 1980 (ursprünglich "C mit Klassen");
Anleihen von SIMULA67, Ada

AT&T-Standard 2.1 für C++ von 1991 mit (u.a.)

- Referenzen, Inline-Funktionen, strenge Typprüfung
- neue dynamische Speicherverwaltung mit *new* und *delete*
- Klassen, einfache und mehrfache Vererbung
- Überladbarkeit von Funktionen und Operatoren
- Stream-Ein-/Ausgabe

AT&T-Standard 3.0 für C++ von 1993 mit zusätzlich *Templates* und *Exceptions*

ANSI C++ : 1998 endlich offizielle Standardisierung

Unser Kurs behandelt ANSI-C (praktisch) komplett, bei C++ werden wir nur Möglichkeiten des AT&T-Standards 2.1 ansprechen.

1.2 Grundlegende Eigenschaften von C und C++

Die Sprache C war ursprünglich ausschließlich gedacht zum Implementieren und Erweitern des Betriebssystems UNIX, also eher weniger als Standard-Programmiersprache mit großer Verbreitung. Ursprünglich, wohlgemerkt! Davon zeugen immer noch deren Eigenschaften wie

- Reichtum an Operatoren,
- relative Maschinennähe,
- möglichst hohe Portabilität auf alle Rechnerarchitekturen,
- kleiner Sprachumfang (nur 32 Schlüsselwörter in C; C++ hat 48 Schlüsselwörter).

Dies sind die 32 C-Schlüsselwörter; wir werden sie alle im Kurs kennenlernen:

```
auto const double float int short struct unsigned
break continue else for long signed switch void
case default enum goto register sizeof typedef volatile
char do extern if return static union while
```

- Dieser sehr kleine Sprachumfang ist fast immer ungenügend (z.B. gibt es keine fest eingebaute Ein-/Ausgabemöglichkeit in C). Deshalb finden fast immer **Routinen** aus definierten **Standard-Bibliotheken** Anwendung. Diese werden bei jedem C(++) - Compiler mitgeliefert. Der Programmierer muss aber explizit angeben, welche Kategorien dieser Routinensammlung er benutzen möchte.

- Es gibt viele syntaktische Möglichkeiten in Kombination mit vereinfachten Schreibweisen.

Speziell der letzte Punkt führt zu einer hohen "Toleranz" des Compilers gegenüber Leichtsinnsfehlern. Diesbezüglich ist C auch ideal, wenn es darum geht, **kryptische Programme** zu schreiben und sich selbst damit ein Bein zu stellen. Spätestens seit Festsetzung des **ANSI-Standards** hat sich die praktische Verwendbarkeit wesentlich verbessert. Durch den obligatorischen **Einsatz von sog. Prototypen** kann der Compiler viele Leichtsinnsfehler des Programmierers abfangen. Dennoch: was die kryptischen Programme angeht, sagen die Entwickler selbst: **"C retains the basic philosophy that programmers know what they are doing."** Als prinzipielles Konzept sollten sie sich daher einprägen:

Keep it Small, Simple, Modular!

1.3 Entstehung eines ausführbaren Programms

Die Übersetzung eines Quelltextes in ein lauffähiges Programm erfolgt in einer Reihe von Teilschritten. Im einfachsten Fall werden diese Schritte durch einen einzigen Compileraufruf automatisch ausgeführt. Der Anwender merkt höchstens, dass danach zusätzlich zur Datei mit dem Programmtext noch gleichnamige sog. Object-Files (mit Extension *.o* oder *.obj*) vorliegen.

Die Bedienung des im Kurs benutzten Compilers wird hier nicht beschrieben, da er nichts mit der Sprache C(++) an sich zu tun hat. Folgende prinzipiellen Schritte gelten ganz allgemein:

1. **Text-Editor** → C-Quelltext (z.B. `myprog.c`) oder in C++: `myprog.cpp`

2. **Präprozessor** → erweiterter Quelltext

3. **C(++)-Compiler** → Maschinencode `myprog.obj, myprog.o` (Object-Code)
   (noch ohne Bibliotheks-Routinen)

4. **Linker** → ausführbares Programm (z.B. `myprog.exe`)

Wenn der Programm-Quelltext auf mehrere Dateien (sog. **Module**) verteilt ist, werden mehrere Objekt-Files erzeugt, die dann schließlich über den Linker in *ein* ausführbares Programm münden.

1.4 „Hallo Welt!“ – Bildschirm-Ausgabe in reinem C

Bevor man richtig in die Programmierung einsteigen kann, ist es wichtig, sich zunächst mit der Ein- und Ausgabe anzufreunden. Dadurch ist es erst möglich, dass das Programm dem Benutzer etwas mitteilt und natürlich auch umgekehrt. Um etwas auf den Bildschirm ausgeben zu lassen, verwendet man die Funktion `printf`. Die C-Funktion `scanf` erlaubt es dem Benutzer, etwas über die Tastatur einzugeben, was vom Programm verwendet werden soll. Beschränken wir uns zunächst aber auf die Ausgabe mit `printf`.

Das folgende Beispiel zeigt das kürzeste vollständige C-Programm; es gibt *Hello world!* auf dem Bildschirm aus:

```
#include <stdio.h>

int main(void)
{
     printf("Hello world!\n");       /* Ein Kommentar... */
}
```

Anmerkungen (sie gelten für C *und* C++, falls nicht explizit anders erwähnt)

- Wie erwähnt, kennt C keine fest eingebauten Routinen zur Ein- und Ausgabe. Sie werden in einer Bibliothek mit dem Namen *stdio* bereitgestellt. Sie müssen aber dem Compiler mitteilen, welche Bibliotheken Sie verwenden wollen, C ist ja modular! Dies geht mit

  `#include <BibName.h>`

  ganz am Anfang des Programmes.

- Das Hauptprogramm muss immer **`main`** heißen. Zunächst schreiben wir

  `int main(void)`

  (ohne Strichpunkt!), um das Hauptprogramm einzuleiten. Formal bedeutet dies, dass wir dem Hauptprogramm keine Parameter übergeben wollen (`void` heißt *leer*) und eine `int`-Variable zurückbekommen wollen. Im Kapitel 4 über Funktionen werden wir diese Schreibweise verstehen und weitere Möglichkeiten kennen lernen.

- Das komplette Hauptprogramm steht zwischen geschweiften Klammern.

- Das eigentliche Programm besteht aus Ausdrücken (sog. statements), die durch einen Strichpunkt terminiert sind. Unser Beispielprogramm hat nur eine einzige Anweisung.

- **Kommentare** (das sind Stellen, die der Compiler ignorieren soll) werden bei C in `/* ... */` eingeschlossen. Diese können sich auch über mehrere Zeilen hinweg erstrecken, aber sie dürfen nicht geschachtelt werden!

- Die Funktion `printf` gibt den Text aus, der zwischen den Anführungszeichen steht. Ziel der Ausgabe ist ein simples Textfenster auf dem Bildschirm (sog. *stdout*-Gerät).

- Es wird unterschieden zwischen Groß- und Kleinbuchstaben (***case sensitive***). Alle Schlüsselwörter werden klein geschrieben.

- Sogenannte *whitespace characters* (Leerzeichen, Tabulator, Zeilenwechsel) werden i.d.R. ignoriert. Sie dürfen z.B. schreiben:

  `printf("ABC");` oder `printf ("ABC") ;`

  Natürlich dürfen keine Schlüsselwörter oder Funktionsnamen ‚zerhackt' werden. **Bitte sorgen sie immer für ein übersichtliches Layout Ihres Programmes!**

- Der Compiler weiß, wo die sog. **Header-Files** `*.h` liegen, die mit `#include` eingebunden werden. Unter Unix liegen sie standardmäßig unter `/usr/include`. Bei einem älteren Borland-Compiler liegen sie in `f:\borland\bcpp.31\include`. Header-Dateien beinhalten Konstanten und Funktionsdefinitionen und sind **Klartext -**C(++)-Anweisungen. Bei der Borland-Entwicklungsumgebung landen Sie manchmal bei Programmabbruch in einer solchen Include-Datei, bitte nicht erschrecken!

- Vergisst man, die nötigen Header-Files einzubinden, gibt's nicht immer eine Fehlermeldung, das Programm macht aber dann i.d.R. Unsinn. ANSI C und erst recht C++ verlangen allerdings i.d.R. das Einbinden der passenden Header-Dateien.

- Beim **Linken** wird dann der **Maschinencode** der entsprechenden **Funktionen** wie z.B. für `printf` dazugebunden. Es sind immer mehrere Funktionen zu einer sog. **Bibliothek** zusammengefaßt. Unter Unix z.B. ist das Standard-Bibliotheks-Directory `/usr/lib`. In der Regel weiß der Compiler selbst, welche Bibliotheken er hinzubinden muss, um ein ablauffähiges Programm zu erzeugen. Manchmal muss man ihn dazu überreden, z.B. unter Unix mit `gcc mathedemo.c -lm`. Die Option `-lm` bindet die Mathmatik-Bibliothek dazu. Dies ist erfoderlich, wenn im `Programm mathdemo.c` z.B. ein Sinus vorkommt.

1.5 Bildschirm-Ausgabe in C++

C++ bietet zusätzlich `cin` und `cout` für die Ein- und Ausgabe an. Diese Funktionen sind manchmal einfacher zu verwenden als `scanf` und `printf`. Vor allem das Einlesen mit `cin` ist weniger fehleranfällig als mit `scanf`. Zunächst aber zu `cout`.

Das folgende Programm zeigt das *"Hello world!"* – Programm in C++:

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

int main(void)
{
  cout << "Hello world!\n";     // Ein Kommentar...
}
```

Was gibt's hier zu beobachten?

- Um `cout` und `cin` verwenden zu können, muss `iostream` eingebunden werden.

- Der Text, der ausgegeben werden soll, steht nicht in Klammern wie bei `printf`, sondern hinter zwei Pfeilen `<<`. Es können mithilfe dieser Pfeile auch mehrere Fragmente hintereinander gehängt werden, wie man in Abschnitt 1.8 sieht.

- C++ bietet eine neue Art von Kommentaren: alles hinter `//` bis zum Ende der Zeile wird ignoriert. Es ist kein Endzeichen nötig.

- Include-Dateien für C++ haben nach dem aktuellen Standard **keine** Endung. (Früher wurde auch `.h` als Endung akzeptiert, wie bei C-Header Dateien).

- In C++ kann man sog. *namespaces* definieren, was hier nicht weiter erklärt werden soll. Für die C++ Ein- und Ausgabe verwendet man den `namespace std`.

1.6 Tastatur-Eingabe in C

Im folgenden Programm wird der Benutzer aufgefordert, zwei Zahlen einzugeben. Diese werden eingelesen und in den Variablen `n` und `m` gespeichert. Dann wird deren Summe ausgegeben.

```
#include <stdio.h>

int main(void)
{
  int n, m;
  printf("Zwei ganze Zahlen eingeben: ");
  scanf("%d %d", &n, &m);
  printf("Die Summe von %d und %d ist %d\n", n, m, n+m);
}
```

Anmerkungen

- Am **Beginn eines Blockes** (ein Block ist spezifiziert durch `{ ... }`) müssen die hierin verwendeten **Variablen deklariert** werden (soweit dies nicht schon außerhalb geschehen ist). Dies geschieht jeweils durch Angabe des **Datentyps**, hier `int` für ganze Zahlen, gefolgt vom Namen der Variablen, hier `n` und `m`. Die Variablendeklaration erscheint oft lästig (in manchen Sprachen wie z.B. *php* ist sie auch nicht notwendig), erleichert aber sehr die Übersicht. Weitere Datentypen sind `char` (für Zeichen) und `float` oder `double` für Fliesskommazahlen.

- C++ erlaubt die Deklaration von Variablen irgendwo innerhalb des Blockes vor der ersten Verwendung. Diese Möglichkeit wird aber i.d.R. nicht empfohlen!

- Beim Einlesen mit `scanf` müssen die Formate der einzulesenden Variablen in Anführungszeichen angegeben werden. Hier ist es `%d` für ganze Zahlen. Weitere Formatangaben finden sie in der Tabelle unten. Dahinter stehen die Variablen, in die die Werte eingelesen werden sollen, in der Regel **mit einem & vor dem Namen!**

- Will man Variablen mit `printf` ausgeben, so schreibt man an die jeweiligen Stellen im Text, wo deren Werte erscheinen sollen, das entsprechende Formatzeichen, hier `%d`. Dahinter erscheinen, mit Kommas getrennt, die Variablen in der richtigen Reihenfolge.

- Um eine neue Zeile zu beginnen, schreibt man einfach das Steuerzeichen `\n` in den Text. Weitere Steuerzeichen finden sie in der Tabelle weiter unten.

1.7 Formatangaben der C-Funktionen `printf` und `scanf`

Sowohl bei der Eingabe als auch bei der Ausgabe ist es absolut wichtig, das richtige Format anzugeben. Man darf keinesfalls Ganzzahlen mit Fließkommazahlen durcheinanderbringen. **Eine falsche Formatangabe führt zu unvorhersehbaren Ergebnissen!**

Format-Spezifikationen bei der Ausgabe mit `printf`

Format	Ausgabe	Typ
`%d`	dezimale Integer-Ausgabe; `%5d` reserviert Platz für fünf Stellen	`int`
`%x`	hexadezimale[1] Integer-Ausgabe	`int`
`%f`	Ausgabe von Fließkommazahlen; mit `%4.2f` wird auf 2 Nachkommastellen gerundet und Platz für 4 Stellen (inkl. Komma) reserviert	`float` oder `double`
`%e`	Fließkomma-Ausgabe im wissenschaftlichen Format, z.B. `2.5e-3` für $2.5 \cdot 10^{-3}$	`float` oder `double`
`%c`	Ausgabe eines einzelnen Zeichens	`int` oder `char`
`%s`	Ausgabe einer Zeichenkette *(String)*	Array aus `char`

Darstellung von Steuerzeichen (z.B. für die Bildschirmausgabe)

`\a`	Piepser
`\n`	neue Zeile
`\r`	Cursor auf Zeilenanfang
`\t`	Tabulator
`\\`	ein `\`
`\'`	ein `'`
`\"`	ein `"`
`\?`	ein `?`

[1] Das Hexadezimalsystem (16er-System) ist ein Zahlensystem mit 16 Ziffern: 0,..,9,A,...,F, im Gegensatz zu unserem bekannten dezimalen System. Im Computer werden alle Zahlen binär abgespeichert, nur mit 0 und 1. Beispiel: Zahl 798 als Binärzahl: `0000 0011 0001 1110`. Damit die Zahlen nicht so lang sind, fasst man gerne 4 Binärziffern zu einer Hexadezimal-Ziffer zusammen, in diesem Beispiel also zu `031E`.

Ausgabe von Tabellen

Ausgaben mit `printf` lassen sich sehr einfach mit Hilfe der Formatangaben aus obiger Tabelle formatieren. Will man z.B. mehrere Fließkommazahlen untereinander ausgeben, so dass sie am Komma ausgerichtet sind, so kann man dies erreichen, indem man den entsprechenden Platz für die Ausgabe reserviert. Mit `%6.2f` wird Platz für insgesamt sechs Stellen (inkl. dem Komma) freigehalten und das Ergebnis wird auf zwei Nachkommastellen gerundet:

```
float x=123.456, y=1.2;
printf("%6.2f \n%6.2f \n", x, y);
```

liefert:
```
123.46
  1.20
```

Analog läßt sich bei der Ausgabe von Ganzzahlen mit `%3d` Platz für insgesamt 3 Stellen reservieren.

Format-Spezifikationen bei der Eingabe mit `scanf`

Beim **Einlesen** von der Tastatur mit `scanf` sind die gleichen Formatangaben gültig wie bei `printf` mit **einer fehlerträchtigen Ausnahme:**

`%f`	Einlesen von `float` (Fließkommazahlen mit kurzem Wertebereich)
`%lf`	Einlesen von `double` (Fließkommazahlen mit langem Wertebereich)

Ausserdem muss den Variablennamen bei `scanf` ein **`&`** vorangestellt werden. Bei **Zeichenketten** allerdings, die als Felder von Zeichen realisiert werden, **darf kein &** stehen! Der Grund dafür wird im Kapitel 8 über Pointer klar.

Folgendes Beispiel zeigt, wie ein einzelnes Zeichen (Typ `char`) in die Variable `c` eingelesen wird und wie eine Zeichenkette, deklariert als `char wort[50]` (maximale Länge ist 50 Zeichen), der Variablen `wort` zugewiesen wird:

```
int main(void)
 {
   char c;
   char wort[50];
   ...
   scanf("%c %s",&c, wort);
   ...
 }
```

Merke:	Bei allen Variablentypen, ausser bei *Zeichenketten*, steht beim Einlesen mit `scanf` ein **`&`** vor dem Variablennamen.

1.8 Eingabe in C++

Das folgende Programm leistet dasselbe wie das vorher gezeigte: zwei Zahlen werden eingelesen und deren Summe wird ausgegeben. Nun wird aber die Ein- und Ausgabe von `cin` und `cout` erledigt.

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

int main(void)
{
  int n, m;
  cout << "Zwei ganze Zahlen eingeben: ";
  cin >> n >> m;
  cout <<"Die Summe von "<< n <<" und "<< m <<" ist "<< n+m <<"\n";
}
```

Anmerkungen

- Beim Einlesen mit `cin` ist **weder eine Formatangabe noch ein &-Zeichen** nötig. Dies macht die Eingabe weniger fehleranfällig als es mit `scanf` der Fall ist.

- Es ist möglich, mehrere Werte einzulesen, indem man die entsprechenden Variablen mit `>>` aneinanderhängt. Um mehrere Werte einzugeben, kann man als Trennzeichen ein Leerzeichen oder einen Zeilenvorschub (Return) verwenden. Ein Komma oder sonstige ‚lesbare' Zeichen sind nicht brauchbar.

- Will man bei `cout` zwischen Textfragmenten Variablenwerte ausgeben, so kann man diese mit `<<` verknüpfen.

1.9 Das Konzept der Stream-IO von C++ [2]

Die Ein- und Ausgabe in C wird also durch die Funktionen `printf` und `scanf` erledigt, die sehr fehleranfällig sein können. **C++** implementiert zusätzlich `cin` und `cout`, wie wir gesehen haben. Mit diesem sog. **Streams-Konzept** sind äußerst einfach Ein- und Ausgaben von beliebigen Objekten auf beliebige Geräte möglich sind. Es ist in Form von Klassenbibliotheken implementiert, kann also an dieser Stelle sicher noch nicht richtig verstanden werden. Hier sollen nur **Kochrezepte** für die Benutzung gegeben werden. Im Kapitel 7 über Dateibearbeitung kommen wir noch einmal auf die Stream-IO zurück, dann werden sie ein paar weitere Möglichkeiten kennenlernen.

Zunächst einmal muss natürlich mit `#include <iostream>` die entsprechende Bibliothek eingebunden werden. **Datenströme** laufen von einem Sender zu einem Empfänger, also:

[2] Dieser Abschnitt ist ein kurzer Auszug aus dem C++ - Skript von U. Werling. Vielen Dank!

- **von** der Tastatur **zu** einer Variablen: `cin >> variable`
- **von** einer Variablen **zum** Bildschirm: `cout << variable`

Der entsprechende Name des C++ Objektes `cin` entspricht in C `stdin`, `cout` entspricht `stdout` in C. Außerdem gibt's noch eine Möglichkeit zur Fehlerausgabe: `cerr` in C++, in C `stderr` genannt; dies ist i.A. gleichbedeutend mit `cout`.

Die Pfeilrichtung der Operatoren `<<` und `>>` gibt – ganz intuitiv – die Richtung wieder, in der der Datenstrom fließt. **Vorsicht:** Nicht verwechseln mit den Bit-Operatoren für ganze Zahlen, wie sie in Kapitel 2 eingeführt werden.

Anmerkungen zur Ausgabe

- Das Ausgabeformat wird **automatisch** aus dem Typ der Variablen bestimmt. Das ist meist vorteilhaft, manchmal sind hierfür aber auch explizite Typumwandlungen nötig:

```
char a='A';            (belege char-Variable a mit Zeichen A)
int  b='B';            (belege int-Variable b mit dem Zeichen B)
cout << a;             (es wird A ausgegeben)
cout << b;             (es wird 66 ausgegeben (ASCII-Wert von B))
cout << char(b);       (jetzt wird B ausgegeben)
```

- Man kann beliebige Typen mischen, alle C-Kontrollzeichen sind erlaubt:

```
cout <<  "Text1"  << intzahl << "Text2" << realzahl << "\n";
```

  Es werden **keine** Leerzeichen automatisch eingefügt, auch ein Zeilenvorschub muss explizit angegeben werden.

- Ausdrücke sind erlaubt, Klammern sind nicht immer notwendig: `cout << a/b;`

Anmerkungen zur Eingabe

- Das Eingabeformat wird **automatisch** aus dem Typ der Variablen bestimmt.

- Man kann, wie bei der Ausgabe, Variablentypen beliebig mischen:

```
int a,b; float x; char text[10];
cin >> a >> x >> b >> text;
```

  Die erste Eingabe wird `a` zugeordnet, die zweite `x`, die dritte `b` und die vierte `text.`

- Führende Leerzeichen werden überlesen (auch bei Strings, also Texteingaben).

- Das Lesen eines Wertes endet beim ersten Zeichen, das nicht mehr zum Typ der entsprechenden Variablen passt. Das Zeichen bleibt aber im Eingabestream, wird also von der nächsten Eingabe gelesen. Bitte ausprobieren: Sie wollen eine Integer- und eine Floatzahl einlesen, tippen aber `2.3 17`. Dann wird der Floatzahl `0.3` zugeordnet!

- Vorsicht: Integerwerte, die mit `0` beginnen, werden als oktal betrachtet. Solche, die mit `0x` beginnen, werden hexadezimal interpretiert.

- Bei Stringeingaben wird bis zum ersten Whitespace-Zeichen gelesen (Leerzeichen, Tabulator, Return).

1.10 Formatieren der Stream-Ausgabe von C++

Die gleichen Formatierungen wie bei `printf` kann man natürlich auch mit `cout` erreichen, allerdings erweist sich hier sie Stream-IO als viel komplizierter in der Anwendung. Um die Ausgabe mit `cout` zu formatieren gibt es sog. **Manipulatoren**. Sie ändern die Eigenschaften von Stream-Objekten und bleiben i.d.R. **solange gültig, bis sie zurückgesetzt werden** (auch über mehrere Befehle hinweg!). Um einen Manipulator zu verwenden fügt man ihn einfach zwischen dem auszugebenden Objekt und `cout` ein:

```
cout << var1 << manipulator << var2 << var3;
cin  >> var1 >> manipulator >> var2 >> var3;
```

In diesem Beispiel werden `var2` und `var3` in einer anderen Form ausgegeben als `var1`. Es gibt Manipulatoren mit und ohne zusätzlichen Parameter; in ersterem Fall muss `iomanip` mit eingebunden werden.

- `setw(int n)`

legt die gesamte Ausgabebreite (inkl. dem Komma) auf `n` Stellen fest. Dieser Manipulator wirkt nur auf die *direkt folgende* Variable, danach wird die Breite wieder automatisch gewählt. Der Manipulator wird ignoriert, falls die Breite zu klein ist.

- `setprecision(int n)`

definiert die maximale Anzahl der Stellen, die ausgegeben werden, wobei sowohl die Stellen *vor* als auch *nach* dem Komma gezählt werden. Defaultmäßig sind 6 Stellen eingestellt. Der Default wird eingeschaltet mit `setprecision(0)`.

Weitere Manipulation können mit `setiosflags` gesetzt werden; das Rücksetzen erfolgt mit `resetiosflags`. So kann z.B. mit `setiosflags(ios::scientific)`[3] die Ausgabe auf wissenschaftliches Format gesetzt werden. Mit `setiosflags(ios::fixed)` wird auf Festkommadarstellung umgeschaltet. Dann gibt `setprecision` die Anzahl der **Nachkommastellen** an.

Dieses Beispiel erzeugt die gleiche Ausgabe wie das vorherige Beispiel mit `printf`.

```
#include <iomanip>
...
float x=123.456, y=1.2;
cout << setiosflags(ios::fixed) << setprecision(2) <<
        setw(6) << x << "\n" << setw(6) << y << "\n";
```

[3] Diese Notation können wir an der Stelle noch nicht verstehen. Im Kapitel 10 über objektorientierte Programmierung wird dies näher erläutert. Wer's wissen will: `ios` ist die Klasse, `scientific` die Methode und `::` ist der Bezugsoperator.

Wir sehen, dass diese Art der Ausgabe um einiges komplizierter und länger ist, als die obige Variante mit `printf`. Ganz allgemein kann man empfehlen, bei formatierten Ausgaben `printf` statt `cout` zu verwenden.

Festlegung des Zahlensystems

Bei ganzen Zahlen kann man mit den Manipulatoren `dec` und `hex` zwischen dezimaler und hexadezimaler Ausgabe hin- und herschalten:

```
cout << 192  << "\n";                    → 192
cout << hex << 192  << "\n";             → c0
cout << 192 << dec << 192 << "\n";       → c0 192
```

Der erste Befehl gibt `192` aus. In der zweiten Anweisung wird die Zahl `192` zuerst in Hexadezimaldarstellung gewandelt und dann ausgegeben, es erscheint `c0` am Bildschirm. Bei der dritten Zeile erscheint `c0 192`, weil zunächst die hexadezimale Ausgabe noch aktiv ist, dann wird sie mit `dec` wieder auf dezimale Ausgabe zurückgestellt.

Leeren des Ausgabepuffers

Gibt man etwas auf dem Bildschirm aus, so kann es sein, dass die Ausgabe nicht sofort auf dem Bildschirm erscheint. Sie wird gepuffert und zu irgendeinem Zeitpunkt ausgegeben. Dies fällt vor allem dann auf, wenn zwischen verschiedenen Ausgaben Pausen durch Rechenarbeit sind. Will man Inhalte sofort ausgeben, so kann man dies manuell mit den folgenden beiden Manipulatoren erreichen:

`flush`	leert den Ausgabepuffer.
`endl`	fängt eine neue Zeile an (`\n`) und leert den Ausgabepuffer.

Zum Beispiel:

```
cout << "Dieser Text wird sofort ausgegeben" << endl;
```

1.11 Abfangen von Fehlern bei der Stream-Eingabe von C++

Das Einlesen von Werten in eine Variable ist immer **typspezifisch**. Deshalb können hier häufig Fehler auftreten, wenn der Benutzer den falschen Datentypen eingibt. Am unproblematischsten sind Eingaben von Zeichenketten, weil diese alle Arten von Zeichen speichern können: Ziffern, Buchstaben und Sonderzeichen. Will man aber eine Zahl einlesen und der Benutzer gibt z.B. das Wort *Hallo* ein, dann stürzt das Programm i.A. ab. Man möchte gerne solche Eingabefehler abfangen und den Benutzer bei falscher Eingabe seine Eingabe wiederholen lassen (siehe Beipiel 3). Um dies zu realisieren, müssen wir auf **`while`- Schleifen** vorgreifen, die in Kapitel 3 über Kontrollstrukturen eingeführt werden. Nur soviel vorweg: eine `while`- Schleife wiederholt die Anweisungen zwischen den geschweiften Klammern solange der Ausdruck nach `while` wahr (>0) ist.

Die Anweisung `cin >> x` liest einen Wert von der Tastatur in die Variable x ein. Gleichzeitig erhält der **Ausdruck `(cin >> x)`**

- den Wert 1 = *true*, wenn der eingegebene Wert dem geforderten Typ entspricht
- den Wert 0 = *false*, wenn ein Fehler auftrat.

Beispiel 1: Lies eine ganze Zahl ein, fange Eingabefehler ab:

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

int main(void)
{
  int n;
  cout << "Ganzzahl eingeben: ";
  if ((cin >> n) == 0)
  {
     cout << "Fehler" << endl;
  }
}
```

Anmerkung: wird z.B. `2.3` eingegeben, so gilt das nicht als Fehler; `n` wird der Wert `2` zugewiesen und `0.3` bleibt im Eingabestrom.

Beispiel 2: Lies solange Zahlen ein, bis ein Fehler auftritt

```
int main(void)
{
  float x;
  while ((cin >> x)>0)
  {
     cout << "Eingegeben wurde " << x << endl;
  }
}
```

Beispiel 3: Lies solange ein, bis die **Eingabe korrekt** ist

```
int main(void)
{
  int n;
  char str[100];
  while ((cin >> n)==0)
  {
     cin.clear();          (unbedingt Fehler löschen)
     cin >> str;           (beseitige Inputstream)
  }
  cout << "Korrekte Eingabe war: " << n << endl;
}
```

Wenn beim Einlesen ein Fehler auftritt, geht das Programm in die `while`- Schleife. Dort wird mit `clear` zunächst die Fehlersituation gelöscht. Dann wird der Inhalt, der noch im Eingabestrom steht, als Zeichenkette in die Variable `str` eingelesen (als Zeichenketten kann man alles einlesen!). Danach beginnt die Schleife wieder von vorne, und es wird erneut versucht, eine Zahl in `n` einzulesen.

1.12 Abfangen von Fehlern bei der Eingabe unter Standard-C

Auch `scanf` liefert 1 oder 0 zurück, je nachdem ob die Eingabe funktioniert hat oder nicht. Die Anweisung **`fflush(stdin)`** beseitigt das, was noch im Eingabestrom steht.

```
#include <stdio.h>
int main(void)
{
        int a;
        printf("\nEingabe: ");
        while(scanf("%d",&a)==0)
        {
                fflush(stdin);
                printf("Neue Eingabe: ");
        }
        printf("Korrekte Eingabe war %d\n",a);
}
```

Kap 2: Variable, Konstanten, Operatoren

In diesem Kapitel werden wesentliche Grundlagen von C(++) bereitgestellt, auf die dann anschließend immer wieder zurückgegriffen wird. Für sich allein gesehen ist dieses Kapitel leider nicht besonders spannend...

Konkret geht es in diesem Abschnitt
- um die **Deklaration** und die interne **Darstellung** von Zahlen und Text
- um **Operatoren**: `a+b` ist klar, aber `n%m` kennt noch nicht jeder, und allgemein unbekannt dürfte ein Ausdruck wie dieser sein:
  ```
  c=(c>='a' && c<='z') ? c-('z'-'Z'):c;
  ```
- um **Datentypen** und Typumwandlungen, z.B. von Fließkomma nach Integer
- um **Felder** und Strings

2.1 Variable, Datentypen, Speicherklassen

Variable repräsentieren die Objekte, mit denen ein Programm arbeitet; sie haben einen konkreten Inhalt, der i.A. gelesen und geschrieben werden kann. Wie schon bemerkt müssen Variable vor ihrer Verwendung **deklariert** worden sein, d.h. der Compiler muss wissen, welcher Art diese Variablen sind. Dies geschieht **spätestens am Beginn des betreffenden Blocks** und besteht (im vollständigen Format) in der Angabe von
- Speicherklasse
- Qualifizierer
- elementaren Datentyp

Beispiel:
```
auto   unsigned long   int   semesterzahl;
```
Speicherklasse *zwei Qualifizierer* *elem. Typ* deklarierte Variable

> Wie schon erwähnt, sind in C++ Variablendeklaration an beliebiger Stelle (vor der ersten Benutzung der Variablen) erlaubt. Dies wird aber i.d.R. wegen der Unübersichtlichkeit nicht empfohlen, außer z.B. bei Schleifen.

Ganz exakt betrachtet, ist eine **Variablenvereinbarung** nötig. Dies kann eine **Definition** sein, die gleichzeitig auch das Objekt erzeugt, oder eine **Deklaration**, die nur die Eigenschaften des Objektes festlegt.

2.2 Elementare Datentypen in C(++)

Allgemeine Anmerkung:
Eine **Vorbelegung** bei der Deklaration ist möglich, z.B. `int i=5;` dies geht für alle Variablentypen.

In C gibt es nur die **elementaren Datentypen** **`char int float double void`**.
Es gibt keinen Datentyp *boolean* o.ä!

int

Integer, also ganze Zahl, typischerweise von der Länge eines Maschinenwortes (>= 2 Bytes) (`int` ist Default, wird z.T. auch weggelassen, z.B. bei der Deklaration `long a;`)
Die reale Länge von `int` in C(++) ist nicht festgelegt, sie beträgt z.B. bei älteren Borland-Compilern 2 Bytes, unter Unix meist 4 Bytes.

Es gibt vorzeichenlose Integerzahlen und vorzeichenbehaftete, festgelegt durch den *Qualifier*. Gibt man keinen an, z.B. bei der Deklaration `int n;` dann handelt es sich um **vorzeichenbehaftete** Integerzahlen, mit denen man auch bei negativen Zahlen vernünftig rechnen kann. Beispiel mit 16 Bits:

```
 1 dez. = 0001 hexadezimal  oder 0000 0000 0000 0001 binär
-1 dez. = FFFF hexadezimal  oder 1111 1111 1111 1111 binär
```

Addiert man nun Bit für Bit, ergibt sich eine binäre `1` als Übertrag ganz links (Carry), was aber nicht weiter berücksichtigt wird, und als Ergebnis korrekt 0. Bei 2 Bytes haben vorzeichenbehaftete Zahlen einen Wertebereich `-32768 (=8000 hex) ... +32767 (=7FFF hex)`. Bei **vorzeichenlosen** 2-Byte-Integers beträgt der Wertebereich `0 ... 65535 (= FFFF hex)`, `8000` hexadezimal bedeutet hier `32768` dezimal.

char

In der Regel ist dieser Typ **ein Byte** lang und hat Platz für ein Zeichen *(character).* Hier ein paar Beispiele aus dem allgemein üblichen ASCII-Standard (hexadezimal):

```
0A = CR (carrage return)                    0D=LF (line feed)
20 = Blank
30 = 0          31 = 1      .........      39 = 9
40 = @          41 = A      .........      5A = Z
                61 = a      .........      7A = z
```

Der ASCII-Standard definiert 128 Zeichen. Nach ISO sind auch die Zeichen 128 bis 255 genormt; an diese Norm halten sich Unix und Windows, jedoch nicht DOS.
`char` - Konstanten werden in *single quotes* eingeschlossen, z.B:

```
char zeichen =  'c';
```

Die Typen `char` und `int` sind in C(++) kompatibel. Eine `char`-Variable kann also normale ganze Zahlen beinhalten (allerdings nur kleine....), eine `int`-Variable auch ein Zeichen (wobei Platz vergeudet wird). Ob `char` vorzeichenbehaftet oder vorzeichenlos behandelt wird, ist implementationsabhängig. Man sollte also *chars* immer explizit als `(un)signed` deklarieren, wenn man mit ihnen rechnen will.

Strings sind keine elementaren Datentypen, sondern Felder von `char` und werden weiter unten behandelt.

float

Fließkommazahl (einfach genau, meist 4 Byte). Die Darstellung ist genormt nach IEEE-754, Wertebereich ca. `+-1.4E-45 ... +-3.4E38`, Genauigkeit ca. 8 dezimale Stellen.

In diesem Kurs wollen wir auch etwas über Computer-Interna lernen, daher hier die sog. normalisierte Darstellung einer 4-Byte-Float-Zahl nach IEEE:

```
VEEE EEEE   EMMM MMMM   MMMM MMMM   MMMM MMMM
```

Dies bedeutet:
1. Zahl = `+- (1 + M1/2 + M2/4 + ... + Mn/(2^n)) * 2^Ex`
2. V = Vorzeichen der Mantisse (`0`: positiv, `1`: negativ)
3. Exponent binär, 8 Exponenten-Bits mit Exponent-Bias 127; dies bedeutet:
   `Ex=0` entspricht `EEE EEEE E = 0111 1111 = 127`

Beispiel: `2.0 = (1 + 0*M1/2 + 0*M2/2 + ... ) * 2^1`

also: `EEE..E = 128` (dezimal) oder `100 0000 0` (binär)
also: `2.0 = 0100 0000 0000 0000 0000 0000 0000 0000` (binär)
oder `40 00 00 00` (hexadezimal)

Viele Zahlen können auf diese Methode nicht exakt dargestellt werden, z.B. `1/7` !

Noch eine Anmerkung: Manche Rechner speichern die einzelnen Bytes einer Realzahl in umgekehrter Reihenfolge ab.

double

Fliesskommazahl (doppelt genau, meist 8 Byte)
Höhere Genauigkeit durch mehr Mantissenbits, 16 Stellen dezimale Genauigkeit,
Bereich ca. `+- 10 ^(+-307)`

void

leer (vor allem bei Zeigern und Funktionen wichtig)

2.3 Qualifizierer

`signed` : vorzeichenbehaftet (nur für ganzzahlige Datentypen); ist Default für `int`

`unsigned` : nicht vorzeichenbehaftet (nur für ganzzahlige Datentypen)

`short` (bei `int`) : `int` mit evtl. kleinerem Wertebereich (>= 2 Byte)

`long` (bei `int`) : `int` mit evtl. größerem Wertebereich (>=4 Byte)

`long` (bei `double`): double mit evtl. größerem Wertebereich

`const` : Variable kann nur gelesen werden
(wird u.U. compilerabhängig ignoriert, allerdings liefert Verstoß Compiler-Warnungen.)

nur zur Vollständigkeit:
`volatile` : verhindert "Wegoptimieren" (kann ebenfalls compilerabhängig ignoriert werden).

Wie ersichtlich, sind – außer evtl. bei `char` – diese **Größen** nicht festgelegt, sondern **abhängig vom jeweiligen Compiler**. Definitiv festgelegt sind die angegebenen Mindestgrößen und die folgenden Relationen:

```
short <= int <= long ;     float <= double <= long double
```

Die tatsächliche Größe kann man den Header-Files `<limits.h>` und `<float.h>` entnehmen. Auch über den `sizeof` -Operator läßt sich die Länge bestimmen, siehe weiter unten.

2.4 Speicherklassen

Verstehen und anwenden werden wir diesen Abschnitt erst bei Funktionen.

`auto`: Default; "normale" lokale Variable, zufällig initialisiert; nur innerhalb einer Blocks definiert.

`static`: "statische" Variable, initialisiert mit 0;
Inhalt bleibt innerhalb eines Unterprogrammes erhalten, wenn das Unterprogramm verlassen und wieder aufgerufen wird.

`extern`: lokale Deklaration von globalen Variablen, initialisiert mit 0.

nur zur Vollständigkeit:
`register`: "Wunsch", `char`- oder `int`-Variable im Maschinenregister zu speichern (zufällig initialisiert).

2.5 Konstanten

Ganzzahlig

- dezimal: `-1, 0, 2001, ...`
- oktal: (beginnt mit `'0'`) `020, 01, 0777, ...`
- hexadezimal: (beginnt mit `'0x'`) `0x5F`, `0xF7A`, ...

Diese Konstanten gelten als `signed int`; sollen sie dagegen vom Typ `unsigned`, `long` oder `unsigned long` sein, müssen sie durch Anhängen von `u` oder `U`, bzw. `l` oder `L` als solche spezifiziert werden (Bsp.: `34000UL`).

Fließkomma

mit Dezimalpunkt oder Exponent:

`3.141592,  2.9979e8,  6e-31,` ... (`'e'` oder `'E'`)

Solche Konstanten sind automatisch `double`, wenn nicht `F` oder `f` (für `float`) bzw. `L` oder `l` (für `long double`) folgt.

Zeichenkonstanten

zwischen ' ' eingeschlossen: `'a', 'B', '\n', '\0',` ...

*Konstante Zeichenketten (**Strings**)*

zwischen " " eingeschlossen:

```
"ich bin ein String\n", "hier piept\'s!\a", ...
```

2.6 Operatoren

Die paar wenigen neuen Operatoren von C++ sind in der folgenden Übersicht nicht enthalten, sondern werden jeweils im Zusammenhang erläutert.

Die Tabelle auf der nächsten Seite soll zum Überblick dienen und zum Nachschlagen. Wie schon angemerkt, ist C(++) sehr reich an Operatoren. Zu interpretieren ist die Tabelle z.B. so:

1. `*` steht in einem Rahmen weiter oben als `+`; also wird ausgewertet
   `3*4+5 = (3*4)+5`

2. 'Assoziativität': `+` wird von links nach rechts ausgewertet; bei `4+5+7` wird also gerechnet: 1. Schritt: `4+5=9` 2.Schritt: `9+7=16`

arithmetisch

`+ - * / %` modulo-Operator (nur bei `int`): liefert den Rest bei einer Ganzzahldivision.
z.B. `n=7%5` → `n=2` `n=7/5` → `n=1`

Achtung: Jeder Ausdruck hat einen Wert!
z.B. hat der Ausdruck `y=3` den Wert 3 (Integer). Daher geht dann auch: `x=y=3`
(Auswertung nach Tabelle von rechts nach links)

z.B.
```
float x,y,erg; ...printf("%f %f %f",x,y,erg=x*y);
```
`x*y` wird berechnet, der Variablen *`erg`* zugewiesen, dann ausgegeben.

Vorzeichenoperatoren

`- (+)` Beisp.: `b = -a` (sog. unitärer Operator, + ist erlaubt!)

Vergleichsoperatoren

`< > <= >= ==` (Identität) `!=` (Ungleichheit)
also z.B.: `if` **`(a==b)`** `if (a!=b)` `if (a>=b)`

Vergleichsoperatoren haben geringere Priorität als arithmetische Operatoren!
Also: `a+b>=c+1` entspricht `(a+b) >= (c+1)`
Also sinnvoll: immer klammern, wenn irgendwie unklar.

Übersicht aller C-Operatoren mit Vorrang und Assoziativität

`()`	Funktionsaufruf	links nach rechts
`[]`	Arrayelement	
`.`	Strukturelement	
`->`	Zeiger auf Strukturelement	
`!`	logisch NOT	**rechts nach links**
`~`	bitweises Komplement	
`-`	unitäres Minus	
`++`	Inkrement	
`--`	Dekrement	
`&`	Addresse	
`*`	Inhalt	
`( type )`	Cast	
`sizeof`	Größe in Byte	
`*`	Multiplikation	links nach rechts
`/`	Division	
`%`	Ganzzahl-Modulo	
`+`	Addition	links nach rechts
`-`	Subtraktion	
`<<`	bitweise Linksverschiebung	links nach rechts
`>>`	bitweise Rechtsverschiebung	
`<`	arithmetisch kleiner als	links nach rechts
`>`	arithmetisch größer als	
`<=`	arithmetisch kleiner gleich	
`>=`	arithmetisch größer gleich	
`==`	arithmetisch gleich	links nach rechts
`!=`	arithmetisch ungleich	
`&`	bitweise AND	links nach rechts
`^`	bitweise XOR	links nach rechts
`|`	bitweise OR	links nach rechts
`&&`	logisch AND	links nach rechts
`||`	logisch OR	links nach rechts
`? :`	Fragezeichenoperator	links nach rechts
`= *=` usw.	Zuweisungsoperator(en)	**rechts nach links**
`,`	Kommaoperator	links nach rechts

logische Operatoren

`!` (NOT) `&&` (AND) `||` (OR) (die Worte *not, and, or* dienen nur zur Erläuterung, im Programm dürfen sie – im Gegensatz z.B. zu PHP – nicht stehen!)

Es gibt in C keine logischen Variablen; es werden hierfür Integers benutzt:
falsch: `0`, wahr: `!0`

Die logischen Operatoren liefern 0 oder 1 zuück. Also entsprechen sich z.B
`if (!(A==B))` und `if (A!=B)`

`if (a<b<c)` ist zwar syntaktisch korrekt, aber unsinnig! Beispiel: `a=1, b=5, c=3;`
Auswertung nach Tabelle von links nach rechts:

`(a<b)` ist wahr, Rückgabe `1`; `(1<3)` ist auch wahr, also insgesamt Rückgabe *wahr*.

Bit-Operatoren für *char* und *int*

`&` (AND) | (OR) << *(left shift)* >> *(right shift)*
^ (XOR, kein Exponent!) ~ (sog. 1er-Komplement, alle Bits umdrehen)

z.B. `x = y & 0xFF00;` Setze rechtes Byte eines 16-Bit-Wortes auf 0
z.B. `m = n >> 2;` Dividiere Integer-Zahl durch 4

Bitte auf keinen Fall mit den logischen Operatoren verwechseln!

"Cast"-Operator zur expliziten Typumwandlung

explizite Typenumwandlung (implizite Regeln siehe weiter unten); Beispiel:

```
float q=3.89222, r;
r = (int)q;            // r wird 3.0, q ist unveraendert
```

In C++ ist auch folgende Schreibweise erlaubt: `r = int(q);`

sizeof-Operator

`sizeof(int);` liefert den benötigten Speicherplatz des entsprechenden Variablentyps, in diesem Fall `int`, oder einer Variablen in Bytes zurück. Dies funktioniert auch bei (statischen) Arrays.

Fragezeichenoperator

Kurzschreibweise für ein "*wenn-dann-ansonsten*"-Konstrukt; Beispiel:

```
max = (a>b) ? a : b
```

entspricht: "wenn `a>b` dann `max=a` ansonsten `max=b`".

Weitere Beispiele:

```
i=(j==2)?1:3;              wenn j==2 dann i=1, sonst i=3
i=(i==1)?i=a+b:j=k+l;      also auch Statements erlaubt
c=(c>='a' && c<='z') ? c-('z'-'Z'):c;
```

macht aus Kleinbuchstaben Großbuchstaben, tricky! Zur Erinnerung: Rechnen mit `char`*s* ist erlaubt.

Kommaoperator

`,` zur Trennung von mehreren Anweisungen an derselben Stelle des Programms;
abschreckenendes Beispiel:

```
a = (b=8, c=3, b-c) + 1;
```
*(*a *bekommt den Wert 6 zugewiesen)*

Hier wird von **links nach rechts** berechnet, der Ausdruck ganz rechts in der Klammer hat den Wert `5`, also: `a=6`. Bitte so undurchschaubare Anweisungen nicht verwenden!
Aber sehr brauchbar in Schleifen: `for (i=0,k=0; ....)`

Zuweisungsoperator

Kurzschreibweisen: `a op= b` entspricht `a = a op b;`
(`op` steht für die binären Operatoren `+ - * / % | & ^ >> <<`)

Beispiele:
`a += 5` entspricht `a = a+5`
`a %= b` entspricht `a = a%b`
`i = 3; j = (i += 2);` → `i=5, j=5`

Beispiel zur Abschreckung: `a %= b = d = 1 + e/2;`

Inkrement/Dekrement (Addition/Subtraktion um 1)

`a = a+1` entspricht `a++` (postfix) oder `++a` (präfix) (`--` analog);
die Reihenfolge bestimmt, ob die Variable zuerst gelesen und dann geändert wird oder umgekehrt. Merke: `++` und `--` haben **Vorrang** vor `/ * %`

Beispiele:

```
a = 1; printf("%d", a++); → 1      printf("%d", a);  → 2
a = 1; printf("%d", ++a); → 2      printf("%d", a);  → 2
```

Anmerkungen zur Auswertung von Ausdrücken

- Wichtig: bei binären Operatoren ist die **Reihenfolge der Entwicklung** i.d.R. **unbestimmt**, d.h. z.B. bei `(expr1)+(expr2)` ist nicht klar, welcher der beiden *Expressions* zuerst entwickelt wird. Also ist z.B. *nicht* sinnvoll: `(a++ -1) + (a-b)`

  Bitte dies nicht verwechseln mit der Abarbeitung z.B. einer Summe. Sind die *Expressions* erst mal berechnet, so erfolgt natürlich die Addition von
  `(expr1) + (expr2) + (expr3)`
  gemäß Tabelle von links nach rechts.

- Das gleiche gilt für Funktionsaufrufe; z.B. ist `printf("%d %d",n++,n*5);` Blödsinn.

- Vergleiche dazu C++: auch hier gibt's Probleme:

```
a=2;
cout << a++ << a*5;        -->  2 10  (Compiler-abhängig!)
cout << a++ << a*5;        -->  3 15
```

- Ausnahmen: bei den Operatoren
  `&&` (logisches Und)
  `||` (logisches Oder)
  `?:` (Fragezeichen-Operator)
  `,` (Kommaoperator)
  ist die Reihenfolge der Entwicklung (und auch die Abarbeitung gemäß Tabelle) **von links nach rechts**! Siehe obiges Beispiel mit dem Komma-Operator.

- Bei den **logischen Operatoren** `&&` und `||` kommt noch hinzu, dass die zugehörigen Argumentausdrücke **nur soweit entwickelt** werden, bis das Ergebnis feststeht, also bei
  `(!0) || expr` bzw. `0 && expr`
  der Ausdruck *expr* erst gar nicht berechnet wird.

  Beispiel: `int n; c = (n && ((1.0/n)<b));` (also kein Unglück, falls `n==0`)

2.7 Implizite Typ-Umwandlung

Bei der Verknüpfung von Variablen unterschiedlichen Datentyps ist das Ergebnis vom höherwertigeren der beiden Datentypen gemäß folgender Hierarchie:

```
char < int < unsigned int < long < unsigned long <
float < double < long double
```

Beispiel: `result = ch * inum + ch * lnum – fnum/dnum`

betrachte die einzelnen Summanden:

```
ch: char;      inum: int;     → int
ch: char;      lnum: long     → long
fnum: float;   dnum: double   → double
```

jetzt Addieren von links nach rechts laut Tabelle:

`int + long → long + double → double`

Es wird nie der Typ einer Variablen automatisch geändert!
Wenn z.B.`result` als `int` deklariert wurde, erfolgt bei der Zuweisung Abschneiden der Nachkommastellen des `double`-Ergebnisses. Man sagt auch **der Links-Typ ist maßgebend.**

Konkret passiert folgendes z.B. bei der Zuweisung eines `float`-Wertes an eine `int`-Zahl: `float` wird nach `long` konvertiert, das niederwertige Wort wird genommen. Ist die Zahl zu groß für `long`, ist Ergebnis undefiniert, eine Fehlermeldung gibt's i.d.R. nicht.

Typumwandlung bei **Funktionsaufrufen:** z.B. erwartet die Wurzelfunktion `sqrt` eine `double` Zahl. Aber `sqrt('@')` ist erlaubt und korrekt und liefert `8.`

Anmerkungen

- Division zweier `int` ergibt wieder `int`, also den **Ganzzahlanteil** des exakten Wertes.
- Bei Umwandlung von höherwertigem zu niederwertigem Datentyp ist Informationsverlust möglich!
- Vorsicht bei Vergleich von `signed` und `unsigned` Typen! Beispielsweise könnte `-1 > 1U` wahr sein. Bei Unklarheiten immer explizites Casting verwenden!
- Zu `char` und `int`:
  `char c='A'; printf("%c %d %c",c,c,c+1);` → A 65 B
- Bei C++ geht's nicht so einfach; da muss man schreiben
  `cout << c << int(c) << char(c+1);`

2.8 Eindimensionale statische Arrays

Arrays (Vektoren) sind Felder von Variablen gleichen Typs. Diese Variablen werden durch Indizes unterschieden. Im Kapitel zu **Pointern** werden wir nochmals auf Arrays zu sprechen kommen.

Beispiel: Deklaration eines Integer-Feldes mit 5 Elementen: `int n[5];`
Diese 5 Variablen liegen dann im Speicher hintereinander:

```
n[0] n[1] n[2] n[3] n[4]
```

Wichtig: Die Angabe der **Feldgröße** muss in diesem Fall durch eine **Konstante** erfolgen, deshalb die Bezeichnung "statisch"! Die so deklarierten Felder **beginnen immer mit Index `[0]`** und **enden mit Index `[Feldgröße-1]`.** Mit `sizeof()` kann die Größe später auch (z.B. in einer Funktion) abgefragt werden.

Vorsicht! Der Umgang mit Feldelementen ist direkter Speicherzugriff! **Es erfolgt keine Überprüfung auf gültigen Speicherbereich von Seiten des Compilers!**

Ansonsten gilt für Feld**elemente** das gleiche wie für sonstige Variable dieses Typs:

```
int n[4],m[4],i=3; n[1] = 1; m[2] = i;      (dies ist ok)
n[4] = 5*i;                                 (schwerer Fehler)
n = m   oder  if (n == m) ...               (geht nicht!)
```

Bei der Deklaration kann eine elegante Initialisierung erfolgen; Beispiel:

```
int n[] = {-8,7,9,-13};
```

Die (Mindest-)Feldgröße wird dabei automatisch festgelegt. Ohne explizite Initialisierung gilt: der **Inhalt eines definierten Feldes ist zunächst beliebig** (wie bei jeder Variablen).

2.9 Strings

Strings in C stellen einen Sonderfall von Arrays dar:
`"........"` sind bereits eindimensionale Felder, die mit `'\0'` terminiert sind. Beispiel:

```
char ch[] = "String"; ist völlig äquivalent mit
char ch[] = {'S','t','r','i','n','g','\0'};
```

mögliche Verwendung bei `printf()` bzw. `scanf()`:

```
char msg[] = "Hello world!", format[] = "%s\n";
printf(format, msg);
```

oder:

```
char msg[100];      (sicherheitshalber überdimensionieren...)
scanf("%s", msg);   (hier steht vor msg KEIN '&', weil msg Adresse ist)
```

Vorsicht: Außer bei der Deklaration kann man Strings nicht einfach mit = zuweisen. Hierzu gibt es die Funktion `strcpy` aus der Standard-C-Bibliothek **`<string.h>`**.

```
#include <string.h>
...
char wort[30];                (Zeichenkette mit maximal 30 Zeichen)
wort = "Hallo";               (Diese Zuweisung geht nicht!)
strcpy (wort, "Hallo");       (So wird's gemacht)
```

Die Bibliothek **`<string.h>`** bietet viele Möglichkeiten, mit Zeichenketten zu arbeiten, wie z.B. Strings zu vergleichen oder aneinander zu hängen. Wenn man intensiver mit Zeichenketten arbeiten will, empfiehlt es sich, in der **Online-Hilfe** nachzulesen. Lassen Sie sich nicht abschrecken durch die Syntax; die Routinen sind ganz leicht anzuwenden. Die am häufigsten benötigten Routinen sind:

`strcat`	*Append string*
`strchr`	*Find character in string*
`strcmp`	*Compare two strings*
`strcpy`	*Copy string*
`strlen`	*Return string length*
`strstr`	*Find substring*

C++ bietet im übrigen noch viel mehr nützliche String-Tools...

Um zwei Zeichenketten `s1` und `s2` miteinander zu vergleichen, verwendet man die Funktion `strcmp(s1, s2)`. Sie liefert `0` zurück, wenn die beiden Strings identisch sind, eine Zahl kleiner `0`, wenn `s1<s2` (alphabetisch) und eine Zahl größer `0`, wenn `s1>s2`.

```
char wort[30];
...
if (strcmp(wort, "Hallo")==0)...   (wenn wort=="Hallo" ist, dann...)
```

2.10 Mehrdimensionale Felder (statisch)

Bei mehrdimensionalen Arrays werden die Variablen durch ein Tupel von Indizes unterschieden. Beispiel:

```
float x[8][30];
```

erzeugt eine Matrix mit **8 Zeilen** und **30 Spalten.** Hier gilt dann analog: erstes Element ist `x[0][0]`, bzw. letztes ist `x[7][29]`.

Auch hier gibt es bei der Deklaration wieder die bequeme Möglichkeit der Initialisierung:

```
int y[][3] = {{8,7},{-3,19,18},{1},{12,-83}};
```

Nicht explizit belegte Felder werden mit 0 belegt. Der Compiler kann aber selbstständig nur die Zahl der Zeilen feststellen; die notwendige (Mindest-)Zahl der Spalten (hier 3) muss ihm mitgeteilt werden. Die vier Zeilen der Matrix schauen also so aus:

```
(8 7 0)  (-3 19 18)  (1 0 0)  (12 -83 0)
```

Intern werden mehrdimensionale Felder eindimensional angelegt; **die Zeilen liegen hintereinander im Speicher**. Wenn also z.B. ein Array deklariert ist mit `int a[4][3];`, dann sind die Aufrufe

`a[i][j]`, `a[0][i*3+j]` und `a[k][(i-k)*3+j]`

völlig äquivalent (und erlaubt!). Im Speicher liegen alle Feldelemente hintereinander. Überschreitet ein Index die (eigentliche) Grenze, so geht der Compiler die entsprechende Anzahl an Speicherzellen weiter und liest diesen Inhalt. Folgende Skizze zeigt, dass sich zum Beispiel `a[1][1]` und `a[0][4]` auf den gleichen Speicherbereich beziehen.

a[0][0]	a[0][1]	a[0][2]	a[1][0]	a[1][1]	a[1][2]	a[2][0]	...
a[0][0]	a[0][1]	a[0][2]	a[0][3]	a[0][4]	a[0][5]	a[0][6]	...

Kap 3: Kontrollstrukturen

3.1 Blöcke

Ein Ausdruck (z.B. `a=5`) gefolgt von einem `;` bezeichnet man als **Statement** (`a=5;`).
Als **Block** wird bezeichnet, was in einem Klammerpaar `{ }` eingeschlossen ist. Ein solcher Block fasst die in ihm enthaltenen Statements zu einem sog. **Verbundstatement** zusammen; dies ist syntaktisch äquivalent zu einem einzelnen Statement.

Blöcke verfügen über bemerkenswerte Eigenständigkeit: **innerhalb jedes Blockes können lokale Variablen** deklariert werden. Beispiel:

```
int i=3;
{
  int i=4;
  printf("%d", i);            (es wird 4 ausgegeben)
}
printf("%d", i);              (es wird 3 ausgegeben)
```

Es ist üblich, **Klammerebenen durch Einrücken** zu verdeutlichen. Vor allem wenn mehrere verschachtelte Blöcke auftreten erhöht dies sehr die Übersicht!

3.2 while-Schleife

Schleifen sind Kontrollstrukturen, die es ermöglichen, ein bestimmtes Statement so oft auszuführen, bis eine Abbruchbedingung erfüllt wird. Eine Möglichkeit bietet die `while`-Schleife, es folgen `do-while` und `for`.

```
while (expr) statement;
```

Hier wird `statement` wiederholt, solange `expr` wahr ist, d.h. einen Wert > 0 hat.
Wichtig: es wird **zuerst die Abbruchbedingung geprüft**, dann die Schleife durchlaufen!
Wenn `expr` von Anfang an falsch ist, wird `statement` **nie** ausgeführt.

Beispiele:
`while (x<100) x++;` *(erhöht* `x` *solange, bis es 100 wird)*

Will man mehr als nur eine Anweisung öfter durchlaufen, so muss man diese zu einem **Block** zusammenfassen. Wir deklarieren eine `int` Variable `zeile` und setzen den Wert auf 1. Bei jedem Schleifendurchlauf wird der Wert von `zeile` ausgeben und anschließend um eins erhöht. Dies wird wiederholt, solange `zeile<=5` ist.

```
int zeile=1;
while (zeile<=5)
  {
    cout << "Zeile Nr. " << zeile << '\n';
    zeile++;
  }
```

Die Ausgabe dieses Programms sieht folgendermaßen aus:

```
Zeile Nr. 1
Zeile Nr. 2
Zeile Nr. 3
Zeile Nr. 4
Zeile Nr. 5
```

Achtung:

```
while (i=3) ... ;          (ist Unsinn, Endlosschleife!)
while (i==3) ... ;         (so ist's korrekt)
```

Die Anweisung `while(i=3)` betrachtet den Wert des Ausdrucks `(i=3)`, also 3, als Wahrheitswert. Die Zahl 3 ist ungleich 0, also immer wahr – wir haben eine nette Endlosschleife.

3.3 for-Schleife

Die allgemeine Form einer `for`-Schleife ist:

```
for(expr1; expr2; expr3) statement
```

was vollkommen äquivalent ist zu folgender `while`-Schleife:

```
expr1;
while(expr2)
  {
    statement
    expr3;
  }
```

Die Anweisung `expr1` wird nur **einmal vor Beginn** der Schleife ausgeführt. Der Ausdruck `expr2` beinhaltet das **Ausführungskriterium** und `expr3` wird **am Ende** jedes Schleifendurchlaufes ausgeführt, was man typischerweise dazu verwendet, einen Zähler zu erhöhen.

Das klassische Beispiel einer `for`-Schleife:

```
int i;
for (i=0; i<10; i++)
  cout << "Hallo!\n";
```

Dieses Programm gibt zehnmal untereinander *Hallo!* auf dem Bildschirm aus. Das gleiche wie die vorher gezeigte `while`-Schleife leistet folgende `for`-Schleife:

```
int zeile;
for (zeile=1; zeile<=5; zeile++)
  cout << "Zeile Nr. " << zeile << '\n';
```

Die erste Anweisung des `for`-Statements, `zeile=1`, wird *einmal* ausgeführt. Dann wird die Bedingung `zeile<=5` geprüft. Ist sie wahr, so wird die `cout`-Anweisung ausgeführt und anschließend die letzte Anweisung des `for`-Statements `zeile++`. Dann wird erneut die Bedingung geprüft und ggf. die Schleife wiederholt.

Jetzt kombinieren wir `for` und `while`:

```
int main(void)
{
  int zeile=1, block;
  while (zeile<=5)
    {
      for (block=1; block<=4; block++)
        {
          cout << "*******";
          cout << "       ";
        }
      cout << '\n';
      zeile=zeile+1;
    }
}
```

Hier sieht man schon deutlich, wie nützlich ein **sinnvolles Layout** ist. Mithilfe der `while`-Schleife werden also 5 Zeilen ausgegeben. Für jede Zeile läuft die `for`–Schleife von 1 bis 4, es werden also viermal einige Sterne und einige Leerzeichen ausgedruckt. Danach wird ein Zeilenvorschub erzeugt und der Zeilenzähler erhöht. Wenn die Zeilennummer 6 erreicht hat, ist die Bedingung für `while` nicht mehr erfüllt und das Programm wird beendet. Wir sehen also am Bildschirm:

```
*******       *******       *******       *******
*******       *******       *******       *******
*******       *******       *******       *******
*******       *******       *******       *******
*******       *******       *******       *******
```

Anmerkungen

- Ganz allgemein gilt, dass `for`- und `while`-Schleifen absolut kompatibel sind und ineinander überführt werden können. Allerdings ist `for` ideal geeignet, wenn man Befehle eine bestimmte Anzahl oft ausführen will. Sollen komplexere Bedingungen überprüft werden, ist eher `while` vorzuziehen.

- Man muss vorsichtig sein mit `;` direkt hinter der `for`- oder `while`-Anweisung.

  ```
  for (i=0; i<10; i++);
    cout <<"Hallo";
  ```

  Dieses Beispiel gibt nur **einmal** *Hallo* aus, und nicht zehnmal, wie man vielleicht gerne hätte. Der Strichpunkt hinter `for` legt fest, dass die Schleife zehnmal **leer** durchlaufen wird. Nachdem die Schleife beendet ist wird einmal *Hallo* ausgegeben.

- **Weitere Beispiele:**

```
int n, i, fak=1;
...
for (i=1; i<=n; i++) fak*=i;
```

berechnet die Fakultät der Zahl `n`.

Die beiden Strichpunkte in der `for`-Klammer sind essentiell. Es können allerdings auch leere Anweisungen enthalten sein. Mehrere Anweisungen können mit dem Komma-Operator getrennt werden:

```
int i=0;
for (; i<10; putchar('a'+i), i++);
```
gibt aus: `abcdefghij`

Die erste Anweisung in der `for`-Klammer ist leer, da `i` schon bei der Deklaration auf 0 gesetzt wird. Die Funktion `putchar` aus `stdio.h` gibt ein einzelnes Zeichen aus. Bei jedem Schleifendurchlauf werden zwei Anweisungen (`putchar` und `i++`) ausgeführt, die mit dem Komma-Operator getrennt werden.

Die `for`-Klammer kann auch nur leere Anweisungen enthalten. Zum Beispiel ist
```
for (;;);
```
die kürzeste Endlosschleife. Man kann aber eine sinnvolle Schleife daraus machen, indem man mit `break` (siehe 3.5) aus der Schleife springt:
```
for(;;)
  {
    ...
    if (...) break;
  }
```

3.4 do-while-Schleife

```
do
     statement
while(expr);
```

Die `do-while`-Schleife unterscheidet sich gegenüber der `while`-Schleife dadurch, dass hier **zuerst das Statement** ausgeführt wird und **dann** die Abbruchbedingung `expr` auf 0 = *false* getestet wird. Die Anweisungen in der `do-while`-Schleife werden also in jedem Falle einmal ausgeführt. Bei `expr=0` wird die Schleife beendet.

Beispiele:

- `Do-while`-Schleifen eignen sich hervorragend, um Eingabefehler abzufangen. Im folgenden Beispiel wird der Benutzer aufgefordert, eine Zahl zwischen 1 und 10 einzugeben. Die Schleife kontrolliert, ob die eingegebene Zahl tatsächlich im richtigen

Bereich liegt. Wenn nicht wird der Benutzer erneut aufgefordert, eine Zahl einzugeben, bis die Eingabe korrekt war.

```
do
  {
    cout << Zahl zwischen 1 und 10 eingeben: ";
    cin >> n;
  }
while (n<1 || n>10);
```

- Hier wird ein Programmteil immer wieder ausgeführt, solange der Benutzer mit `j` *(ja)* antwortet. Die Standard-C Funktion `getchar` aus `stdio.h` liest ein einzelnes Zeichen von der Tastatur ein.

```
do
  {
    ...
    printf("Nochmal (j/n)?");
  }
while (getchar() == 'j');
```

3.5 break und continue

- `break` führt zum sofortigen **Verlassen der jeweils innersten Schleife**. (Beispiel bei 3.6)
- `continue` erzwingt den sofortigen nächsten Schleifendurchlauf (Beispiel bei 3.6)

3.6 if - else

```
if (expr)
  statement1;
else
  statement2;
```

Wenn `expr` wahr ist, d.h. nicht 0, dann wird `statement1` ausgeführt, ansonsten `statement2`. Das `else` ist optional; wenn in diesem Fall nichts geschehen soll, reicht ein reines `if`-Konstrukt, das nach `statement1` endet. Natürlich können auch hier mehrere Anweisungen in einem Block zusammengefasst werden.

Beispiele:

- `if (x<1) y=2;` *(Strichpunkt beachten!)*
  `else y=3;`
  oder:
  `if (x<1) { .... }` *(Hier steht **kein** ; nach dem Ende des Blocks)*
  `else { .... }`

- ```
  int i;
  if (i==0) machwas;          (beachte ==)
  if (!i)   machwas;          (identisch zu obigem; eleganter?)
  ```

- ```
  float x;
  if (x == 0) ....            (muss bei reellen Zahlen vermieden werden!)
  if (fabs(x)<1E-7) ....      (so ist's sauber. Die Funktioin fabs liefert den
                               Betrag einer float-Zahl)
  ```

- Diese beiden Beipiele lesen 10 Zeichen ein und geben den ASCII-Wert des 'größten' Zeichens aus:

  Möglichkeit 1:

  ```
  int c=0, maxi=0, aux;
  while (c++ <10)
  if ((aux = getchar()) > maxi)
    maxi=aux;
  printf("Groesster ASCII-Wert: %d\n", maxi);
  ```

  Möglichkeit 2:

  ```
  while(1)                          (eigentlich eine Endlosschleife!)
    {
      if ((aux = getchar()) < 32)   (Bei nicht-druckaren Zeichen zum
        continue;                    nächsten Schleifendurchlauf)
      if (aux > maxi) maxi=aux;
      if (c++ == 9) break;          (aufhören nach 9 Durchläufen)
    }
  printf("Groesster ASCII-Wert: %d\n", maxi);
  ```

Geschachtelte `if-else-`Konstrukte

Ein `else` bezieht sich immer auf das nächstliegende `if`, das noch nicht durch ein `else` abgeschlossen ist. Bei mehreren verschachtelten `if-else-`Anweisungen muss man unbedingt auf eine korrekte Klammerung achten!

```
if (t<80)
if (t>60) mache1;
else mache2;              (bezieht sich auf t>60)


if (t<80)
{
  if (t>60) mache1;
}
else mache2;              (jetzt bezieht es sich auf t<80)
```

3.7 switch-case

```
switch(expr)                    (hier steht ein beliebiger Ausdruck)
  {
     case (const)expr1 :        (hier nur ein konstanter Ausdruck)
          statement11
          statement12   ....
          break;
     case (const)expr2 :
          statement21
          statement22   ....
          break;
     default :                  (default ist optional und wird immer
          statements            ausgeführt)
  }
```

Anmerkungen

- `switch`-Anweisungen dürfen verschachtelt werden.

- es sind keine Listen erlaubt, also nicht z.B.
  ```
  switch (c=getchar()) {
       case 'a','b': dosomething
  ```
  stattdessen kann man schreiben:
  ```
  switch (c=getchar()) {
       case 'a':
       case 'b': dosomething
  ```

- **Wichtig:** der *drop through* - Mechanismus: ab der ersten erfüllten `case`-Bedingung werden **alle nachfolgenden Statements** (einschließlich der nach `default`) **ausgeführt!** Zur Vermeidung dessen verwendet man die `break`-Anweisung. Zum Beispiel:

  ```
  char antwort;
  antwort = getchar();
  switch(antwort)
    {
      case 'y': printf("Zustimmung!");
                break;
      case 'n': printf("Ablehnung!");
                break;
      case 'u': printf("Unentschlossenheit!");
                break;
      default : printf("Falsche Eingabe!");
                break;              (Dieses break ist nicht nötig)
    }
  ```

3.8 goto

Mit `goto` kann man zu einer mit einem Label markierten Stelle innerhalb der gleichen Funktion springen. Die Verwendung gilt als **schlechter Stil** und kann bzw. sollte immer vermieden werden. Eine der wenigen sinnvollen Anwendungen könnte z.B. sein, tief verschachtelte Schleifen mit einer einzigen Anweisung zu verlassen.

Beispiel:

```
for(...)
  {
    for(...)
      {
        ... ... if(desaster) goto myerror;
      }
  }
...
myerror :            (markiert das Label myerror)
...
```

Kap 4: Funktionen, globale und lokale Variable

4.1 Wozu braucht man Funktionen?

Funktionen stellen die essentiellen Grundlagen der strukturierten Programmierung dar. Die Idee ist, das Programm in möglichst eigenständige Untereinheiten zu zerlegen, die genauso aussehen, wie das Hauptprogramm `main`, aber mit einem anderen Namen versehen werden. Damit Funktionen nicht komplett isolierte Programmeinheiten sind, kann man ihnen beim Aufruf Parameter mitgeben und sie können der aufrufenden Einheit (z.B. dem Hauptprogramm) einen Wert zurückgeben. Die allgemeine Form einer Funktion ist:

```
Rückgabetyp Funktionsname(Parameterliste)
{
  ...
}
```

Die wichtigsten Gründe für die Verwendung von Funktionen sind:

- Wenn in einem Programm ein Teil öfter auftaucht, so braucht man diesen Teil nur **einmal** als Funktion zu programmieren, die man dann immer wieder aufrufen kann.

- Sobald eine Funktion programmiert (und getestet) ist, kann man sie immer wieder verwenden. Auch in anderen Programmen kann man sie einbinden und als ***black box*** behandeln, d.h. man muss sich nicht um Interna der Funktion kümmern.

- Durch die Verwendung von Funktionen lassen sich Fehler vermeiden, weil die Programmeinheiten kürzer werden. Es ist nachgewiesen, dass man pro 1000 Zeilen Programmcode im Schnitt 25 Fehler einbaut, die zwar syntaktisch richtig sind (also vom Compiler nicht als Fehler erkannt werden), aber zu fehlerhaften Ergebnissen führen können.

- Programme werden übersichtlicher und somit besser lesbar und leichter wartbar.

Dafür gleich mal ein einleitendes Beipiel. Sie benötigen in ihrem (Haupt-)programm an mehrern Stellen den gleichen Algorithmus, z.B. ein schnelles Berechnen von n^m für Integerzahlen. Anstatt an fünf Stellen im Programm immer wieder die gleichen Zeilen einzufügen, lagern sie die Angelegenheit einfach aus in ein extra Modul namens `intpow` und testen es, bis es korrekt funktioniert:

```
long int intpow(int n, int m)
 {
   int i; long int erg=1;
   for (i=1; i<=m; i++)
       erg*=n;
   return erg;
 }
```

In diesem Beispiel erwartet die Funktion `intpow` beim Aufruf zwei Integerzahlen `n` und `m`, berechnet n^m, und gibt das Ergebnis an das Hauptprogramm zurück. Der Rückgabetyp ist `long int` und der Wert wird mit `return` zurückgeliefert. Das Hauptprogramm könnte dann so ausschauen:

```
int main(void)
  {
    cout << "2 hoch 3 ist " << intpow(2,3) << endl;
    . . .
    cout << "3 hoch 5 ist " << intpow(3,5) << endl;
    . . .
    cout << "7 hoch 4 ist " << intpow(7,4) << endl;
  }
```

4.2 Unterprogramme, Funktionen, Prozeduren

Unterprogramme sind also möglichst eigenständige Programm-Untereinheiten; **ihnen wird ein definierter Satz von Werten übergeben**.

Es gibt sog. **Funktionen** als Unterprogramme, die auch immer **einen Wert zurückliefern** (default: `int`). Eine **Prozedur** ist ein Unterprogramm, das **keinen** Wert zurückliefert. In C(++) ist das ziemlich egal, Prozeduren sind einfach Funktionen vom Typ `void`. Also:

`y=demofunc(17.5,3);` (*Aufruf einer **Funktion**)*
Im Funktionskopf steht `float demofunc(float f, int n)` *(ohne **;**)*

Die Funktion `demofunc` **hat selbst einen Wert**, nämlich den, der mit `return` zurückgeliefert wird. Dieser Wert kann entweder einer Variablen zugewiesen werden, wie hier, oder direkt ausgegeben werden, wie in obigem Beispiel.

`demoproc(756);` (*Aufruf einer **Prozedur**)*
Im Funktionskopf steht `void demoproc(int m)`

Formal ist ein Aufruf von `demofunc` auch als Prozedur möglich, dann ist aber ein expliziter *Cast* von `float` nach `void` nötig, die beiden Arten von Unterprogrammen sind ansonsten inkompatibel:

`(void)demofunc(753.3,17);` Aufruf einer Funktion als Prozedur

Wenn ausnahmsweise nichts übergeben werden soll **an** eine Funktion:
`n=dummy();` Aufruf einer Funktion ohne Übergabeparameter
Im Funktionskopf steht `int dummy(void)`

Der Compiler erkennt Funktionen an dem **Klammerpaar**, das dem Funktionsnamen folgt; in diesem Klammerpaar stehen die übergebenen Werte, d.h. die so initialisierten **lokalen Variablen** der Funktion, durch Kommata getrennt (hier kein Kommaoperator). Es erfolgt immer ***Call by Value***, Änderungen der lokalen Variablen im Unterprogramm sind im Hauptprogramm nicht wirksam. Im Pointer-Kapitel werden wir sehen, wie man einen Workaround konstruieren kann, der diese Restriktion umgeht und ein *Call by Reference* nachbildet.

Anmerkungen

- Zur Erinnerung:
  keine feste Reihenfolge bei der Entwicklung der übergebenen Argumente!

- Notwendige **Typumwandlungen** werden in ANSI-C **automatisch** durchgeführt (z.B. auch Abschneiden von Nachkommastellen).

- In diesem Kapitel werden noch **keine Arrays oder Funktionsnamen als Parameter** behandelt. Dies funktioniert erst mit Pointern.

- Was es in C auch noch gibt, hier aber nicht näher behandelt wird:
  Funktionen mit variablen Argumentlisten
  Dies kennen wir schon: bei `printf` z.B. bestimmt der erste übergebene Parameter, der Formatstring, wie viele Argumente dann noch kommen:

```
printf("Hallo Welt \n");        (kein weiterer Parameter)
printf("Ergebnis: %d\n", m);    (ein weiterer Parameter)
```

Um Funktionen zu **verlassen**, wird das Schlüsselwort `return` verwendet; der Wert des evtl. darauffolgenden Ausdrucks stellt den **Rückgabewert** der Funktion dar. Dort kann auch ein Ausdruck stehen:

```
return(a>b?a:b)
```

Ein `return` darf beliebig oft in der Funktion vorkommen, allerdings muss man bedenken, dass die Funktion beim **ersten** `return` verlassen wird.

Ähnlich den Variablen **müssen** Funktionen **vor ihrem ersten Aufruf deklariert** werden, d.h., der Compiler muss Typ und Anzahl der zu übergebenden Argumente sowie den Typ des Rückgabewertes kennen. ANSI-C und erst recht C++ ist hier sehr streng. Programmtechnisch geschieht dies durch Verwendung eines **Funktions-Prototyps**. Dieser Funktions-Prototyp hat folgendes Format:

```
Rückgabetyp Funktionsname(Parameterliste);
```

("Kopfzeile + ***Strichpunkt****")*

Die übliche Reihenfolge in einer **Quelldatei** ist:

1. zuerst die Funktionsprototypen
2. dann das Hauptprogramm `main`
3. zuletzt die Funktionsdefinitionen, also die eigentliche Funktionsroutinen.

Funktionen sind i.d.R.global gültig. Sie können nur **außerhalb jeder anderen Funktion** (einschließlich `main()`) **definiert** werden und sind dann aus jeder Funktion (einschließlich derselbigen – das heißt dann *Rekursion*) aufrufbar.

Das folgende Beispiel zeigt ein vollständiges Programm mit einer Funktion, die näherungsweise die Exponentialfunktion $\exp(x) \approx \sum_{n=0}^{k} x^n / n!$ berechnet.

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

double myexp(double, int);        (Funktions-Prototyp mit ; genauso ginge
                                  double myexp(double xx, int kk);
                                  wobei die Variablennamen xx und kk beliebig sind)
int main(void)
{
  int i=10; double d=5.0, w;
  w = myexp(d, i);                (Funktions-Aufruf)
  cout << "Ergebnis: " << w;
}

double myexp(double x, int k)     (Funktions-Definition, kein ;)
{
  int n;
  double summand=1.0, sum=1.0;

  for(n=1; n<=k; n++)
    {
      summand *= x/n; sum += summand;
    }
  return sum;                     (Rueckgabewert der Funktion)
}
```

4.3 Lokale und statische Variable

Wir erinnern uns an Kap.2: Die Eigenschaften von Variablen (lokal, global usw.) werden als **Speicherklassen** bezeichnet. Gibt man bei der Deklaration nicht explizit eine andere Speicherklasse an, sind alle 'normal' deklarierten Variablen **lokal**. Sie besitzen nach der Deklaration zunächst einen zufälligen Wert.

Lokale Variable sind innerhalb eines Blocks bzw. einer Funktion deklariert und existieren nur dort. Von „außen" gesehen sind sie **völlig verborgen**. **Nach Verlassen der Funktion oder des Blockes verlieren sie ihren Inhalt.** Um dies zu verhindern, kann man sie explizit der Speicherklasse `static` zuordnen. Die Initialisierung für den ersten Aufruf der zugehörigen Funktion erfolgt sinnvollerweise bei der Deklaration. Initialisiert man nicht, sind statische Variabale automatisch mit 0 belegt. Beispiel:

```
int sum(int n)
{
     static int a=10;
     a += n;
     return a;
}
```

Bei den ersten vier Aufrufen könnte sich dann folgende Sequenz ergeben:

1. `sum(2) = 12`
2. `sum(3) = 15`
3. `sum(1) = 16`
4. `sum(4) = 20`

4.4 Globale Variable

Globale Variable werden **außerhalb jeder Funktion** ohne Schlüsselwort deklariert; ihre Werte können an jeder Stelle des Programms, d.h. auch in jeder Funktion, gelesen und verändert werden. Globale Variable sollte man deshalb möglichst **vermeiden**!

Die Deklaration muss vor dem ersten Aufruf erfolgen (klar!); zusätzlich sollten sie **innerhalb einer Funktion**, in der auf sie zugegriffen wird, **als `extern` redeklariert** werden. Beispiel:

```
double Xdim, Ydim;                 (Deklaration global Variabler Xdim und Ydim)
void myplot(....)
{
  extern double Xdim, Ydim;        (optional, aber guter Stil!)
  double d;
  d = Xdim; Ydim = ....
}
```

Wenn in einem Block / in einer Funktion eine Variable mit dem selben Namen deklariert wird, so **überdeckt** sie hierin die globale Variable.

Nur als Randanmerkung: In C++ gibt es eine Methode, auch auf diese verdeckten globalen Variablen zuzugreifen.

Variable vom Typ `extern` sind auch mit 0 initialisiert. Bei externen Variablen garantiert der Standard nur, dass die ersten **6 Zeichen** des Variablennamens signifikant sind, wobei außerdem Klein-/Großschreibung ignoriert werden kann. Funktionen sollten immer so geschrieben werden, dass man sich bei deren Verwendung nicht um Internas kümmern muss, sie also als *black box* ansehen kann. Dies ist ein weiteres Argument gegen die Verwendung von globalen Variablen. Wenn schon dann sollte man sie klar kennzeichnen z.B. durch einen Namen nur mit GROSSBUCHSTABEN.

Globale Variable werden typischerweise benützt, um Größen, die in vielen Funktionen in einem Programm(-Projekt) bedeutsam sind, mitzuteilen, ohne sie ständig als Variablen übergeben zu müssen. Um sicherzustellen, dass diese nicht irgendwo versehentlich geändert werden, kann man sie vorteilhaft als `const` deklarieren.

4.5 Der Exit-Status eines (Haupt-)Programms

Wie schon erwähnt, kann ein Programm, wenn es seinen Ablauf beendet hat, dem System mitteilen, auf welche Weise („alles OK", „Fehler aufgetreten", etc.) diese Beendigung eingetreten ist. Dazu dient der **Exit-Status**. Der Exit-Status ist eine **ganze Zahl**, die im System unmittelbar nach Programmbeendigung in irgendeiner Form zurückbleibt. Zum Beispiel ist bei der C-Shell unter UNIX `$status` diejenige Shell-Variable, die diesen Wert enthält. Die Art dieser Statusabfrage ist aber stark von einer Reihe von Umständen abhängig (Betriebssystem, Shell etc.). Die Beachtung des Exit-Status wird spätestens dann wichtig, wenn Programme sich gegenseitig aufrufen und abhängig vom Ausgang Entscheidungen treffen müssen.

Der Exit-Status eines Programms ist der **Rückgabewert der Funktion `main`**, also i.a der Wert, der in `main` mit `return` zurückgeliefert wird. Deshalb wird das Hauptprogramm deklariert mit

```
int main(void)
```

Zur sofortigen Beendigung eines Programms bei einem kritischen Fehler - wenn etwa eine Eingabedatei nicht geöffnet werden konnte - und zum definierten Setzen des Exit-Status verwendet man die Bibliotheksfunktion

```
void exit(int status);
```

Wie schon erwähnt, bedeutet nach Konvention der Wert 0, dass alles in Ordnung ist, während eine von 0 verschiedene Zahl einen Fehlerindikator darstellen kann. Bitte nicht mit der Tatsache verwechseln, dass in C *false* durch 0 dargestellt wird! Will man sich nicht um die konkreten Werte des Exit-Status kümmern müssen, bietet sich die Verwendung der dafür bereits vordefinierten Konstanten `EXIT_SUCCESS` und `EXIT_FAILURE` an. Diese Definitionen und der Prototyp von `exit` sind enthalten in `<stdlib.h>`. Ein Beispiel

```
#include <stdlib.h>
int main(void)
{    ...
     if(fehler)
     exit(EXIT_FAILURE);
     ...
     return EXIT_SUCCESS
}
```

4.6 Getrennte Compilierung von Modulen

Bei großen Projekten ist es sinnvoll, die Funktionen auf mehrere Dateien aufzuteilen. Der Linux-Kernel z.B. besteht aus tausenden Modulen. Hier das Prinzip:

Datei `file1.c`:

```
int x;                    (Definition von x als globale Variable)
extern double f;          (Redeklaration als extern, Definition in file2.c)

int fun(void)             (Die Funktion fun wird hier definiert)
{
     return(x + (int)f);
}
```

Datei `file2.c`:

```
extern int x;             (Redeklaration von x als extern)
extern int fun(void);     (Deklaration der externen Funktion fun)
double f;                 (globale Definition von f)
```

```
int main(void)
{
     x=3; f=3.5;
     printf("%d\n",fun());
}
```

Die globalen Variablen `x` und `f` sind in beiden Dateien `file1` **und** `file2` gültig! Unter Unix z.B. ist das Projekt dann zu kompilieren mit

```
gcc file1.c file2.c
```

4.7 Rekursion

Eine Rekursion liegt vor, **wenn eine Funktion sich selbst aufruft**. Durch rekursive Algorithmen sind oft **überraschend kurze und elegante Lösungen** möglich: für eine genauere Besprechung sei auf Kapitel 4.7 verwiesen. ☺ Allerdings: wenn es darum geht, sehr schnelle Programme zu schreiben, sollte man Rekursionen vermeiden.

Der Grundgedanke dabei besteht darin, durch den rekursiven Aufruf das gegebene Problem solange zu verkleinern bzw. in identische kleinere Teilprobleme aufzuteilen, bis ein **Trivialfall** (Abbruchbedingung) erreicht wird. Im Prinzip funktioniert's so, dass Kopie 1 der Funktion Kopie 2 aufruft, die wiederum Kopie 3 usw. In jeder Kopie gibt es einen eigenen lokalen Variablensatz. Hier ein banales Beispiel zur Berechnung der Fakultät:

```
long fak(int wert)
{    if (wert==1)
          return 1;
     else
          return(wert*fak(wert-1));
}
```

Ein weiteres Beispiel: Die Fibonacci-Zahlen f_n sind definiert als Zahlenfolge mit $f_0 = f_1 = 1$ und $f_n = f_{n-1} + f_{n-2}$. Dies lässt sich direkt so programmieren:

```
long fib(int n)                  (Funktion fib liefert die n-te Fibonacci-Zahl)
{    if(n<2) return 1L;          (Trivialfall: gib 1 als long int zurück)
     return fib(n-1)+fib(n-2);
}
```

Ein weiteres Beispiel: Beim Spiel "**Türme von Hanoi**" hat man n unterschiedlich große Lochscheiben, die sich auf 3 möglichen Steckplätzen befinden können. In der Ausgangssituation liegen alle Scheiben als Turm der Größe nach geordnet (die kleinste oben) auf einem Steckplatz. Das Problem besteht darin, den Turm auf einen bestimmten anderen Steckplatz zu bringen, wobei immer nur eine Scheibe bewegt werden und niemals eine größere auf einer kleineren liegen darf.

Zur Lösung geht man wie folgt vor: zuerst werden n-1 Scheiben irgendwie auf den übrigen dritten Platz gebracht (das verkleinerte Problem), dann bringt man die unterste Scheibe auf den Endplatz (Trivialfall), danach werden auch die n-1 Scheiben auf den Endplatz gebracht (ebenfalls verkleinertes Problem).

Kaum ein anderes Beispiel demonstriert so eindrucksvoll die Eleganz von rekursiven Algorithmen. Das Programm findet 'von selbst' die richtige Lösung! Es bewegt `n` Scheiben von Steckplatz `p1` nach `p2`, `p3` ist leer.

```
void hanoi(int n, int p1, int p2)      (es gibt Steckplätze mit Nummern 1,2,3)
{
  int parkplatz;
  if(n>1)
  {
     parkplatz = 6-p1-p2;              (parkplatz ist der dritte unbenutzte Platz)
     hanoi(n-1, p1, parkplatz);        (schiebe n-1 Scheiben auf den Parkplatz)
  }

  printf("1 Scheibe von %d nach %d\n", p1, p2);
                                       (Unterste Scheibe auf den Endplatz)
  if (n>1)
    hanoi(n-1, parkplatz, p2);         (n-1 Scheiben vom Parkplatz auf Endplatz)
}
```

Probieren sie es unbedingt aus! `hanoi(5,1,2)` z.B. bewegt 5 Scheiben von Position 1 nach Position 2. Der integrierte Druckbefehl zeigt Ihnen, wie der Computer automatisch die richtige Lösung des Problems findet.

4.8 Was bringt C++ Neues für Funktionen?

(Dank an U.Werling für diesen Abschnitt)

Wichtige Neuerungen, die hier bereits behandelt werden können, sind
1. Es gibt Default-Parameter für Funktionen.
2. Funktionen können **überladen** werden.

Default-Parameter

Parameter mit Defaultwerten können beim Funktionsaufruf weggelassen werden.
Beispiel für Prototyp (meist auch bei Definition als Kommentar angebbar):

```
void setzeZeit(int hour, int min, int sec=0, char dp='a');
                                  ----------------------
```

Dies kann man aufrufen mit

```
setzeZeit(12,5);          (nur Stunden und Minuten)
setzeZeit(12,5,10);       (Stunden, Minuten, Sekunden)
setzeZeit(12,5,10,'p');   (Stunden, Minuten, Sekunden, Zeichen)
```

Überladen von Funktionen

Problem: ein und dieselbe Funktion soll mit verschiedenen Datentypen arbeiten. C ermöglicht dies nur mit viel Tricks (siehe z.B. die Bibliotheksroutine `qsort`). I.d.R. braucht man für jede Funktion einen eigenen Namen (an den sog. Präprozessor sei jetzt mal nicht gedacht...):

```
int max(const int a, const int b)
char maxchar(const char a, const char b)
float maxfloat(const float a, const float b)
```

obwohl in allen 3 Funktionen das gleiche stehen könnte, z.B. `return(a>b? a:b);`

C++ erlaubt es, Funktionen mit gleichem Namen mehrmals zu definieren, nur die Parameterliste (**Signatur)** muss sich unterscheiden. Der Compiler entscheidet dann selbst, welche Funktion nun zu nehmen ist.

Beispiel:
Vorschlag für Funktion `power` zur Berechnung von a^b . Wenn möglich, sollte dies durch Multiplikation gelöst werden, da dies um Größenordnungen schneller geht als über den Logarithmus.

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

using namespace std;

unsigned long power(unsigned long a, unsigned long b)
{
     long result=1;
     for (int i=1; i<=b; i++)         (einfach multiplizieren)
          result*=a;
     return(result);
}

double power(double a, double b)
{
     return(exp(b*log(a)));
}

int main(void)
{
     unsigned long int a=10, b=5;
     double x=625.0, y=0.25;
     cout << "10^5 = " << power(a,b) << endl;
     cout << "625.0^0.25 = " << power(x,y) << endl;
}
```

Falls keine exakte Übereinstimmung vorhanden ist, z.B. bei $(625.3)^2$, versucht der Compiler, nach ANSI C - Regeln zu wandeln und die 'richtige' Funktion einzusetzen. Wenn er nicht durchblickt, sagt er es.

Kap 5: Abgeleitete Datentypen

In diesem Kapitel geht es um die **Definition von abgeleiteten Datentypen,** bestehend aus einer 'Ansammlung' von elementaren `char`, `float`, `double`, `int` – Variablen. Dabei ist zu beachten, dass nicht nur die zugehörigen Variablen, sondern auch diese Definitionen selbst kontextabhängig global oder lokal sind.

Zur Syntax: bisher ging die Definition/Deklaration von Variablen so:

Typ	tatsächliche Instanz der Variable
`int`	`varx;`

jetzt:

`struct`	`neuerTyp`	`vara;`	*(Strukturen sind essentiell in C(++)!)*
`union`	`neuerTyp`	`varb;`	*(nur zur Vollständigkeit)*
`enum`	`neuerTyp`	`varc;`	*(nur zur Vollständigkeit)*

5.1 Strukturen

Mehrere i.a. verschiedene Datentypen können zu einem einzigen neuen Datentyp, einer sog. Struktur, mittels `struct` zusammengebunden werden. Man denke an ein Array aus unterschiedlichen Datentypen. Der Aufruf der einzelnen Elemente erfolgt dann nicht über Indizes, sondern über deren Namen.

Für das folgende Beispiel denke man an Lampen im Theater. Jede Lampe habe eine x- und y-Koordinate, eine Farbe und eine Kennung aus einem Buchstaben. Ein dem Problem angepasster Datentyp könnte so ausschauen:

```
struct point                (point ist der Strukturname)
{
  double spx, spy;          (spx, spy, etc. sind die Elementnamen der Struktur)
  int farbe;
  char label;
};                          (Der Strichpunkt ist hier wichtig!)
```

Will man Variablen vom neu definierten Typ `point` deklarieren, so schreibt man:

```
struct point punkt1, punkt2;
```

Der Variablentyp heißt hier `struct point`, die definierten Variablen sind `punkt1` und `punkt2`.

Die geschlossene Initialisierung erfolgt (analog zu den Arrays) bei der Deklaration, z.B. mit

```
struct point punkt3 = {2.8, -33.7, 15, 'A'};
```

Um auf die enzelnen Elemente der Struktur zuzugreifen verwendet man einen Punkt:
`Strukturvariable.Elementname`

Zum Beispiel: `punkt1.farbe = 11; punkt2.spy = punkt3.spy;`

Man kann direkt bei der Definition der Struktur Variablen dieses Typs deklarieren, z.B:

```
struct complex
{ double re;
  double im;
} var1, var2;        (var1 und var2 sind Variablen vom Typ complex)
```

Anmerkungen:

- **Arrays** aus Strukturen sind natürlich möglich: `struct complex cfeld[100];` `cfeld` ist dann ein Array aus Strukturen vom Typ `complex`.

- Komplette **Zuweisungen** sind erlaubt: `punkt1 = punkt2;`
  Alle einfachen Strukturelemente werden hier kopiert. Allerdings: ist ein Struktur**element** ein ganzes Array, wird nur die Anfangsadresse kopiert (der Pointer), nicht der Inhalt!

- Komplette Vergleiche sind **nicht** erlaubt: `if (punkt1 == punkt2)` geht nicht!

- Eine ganze Struktur einfach mit dem eingebauten `cout` ausgeben geht nicht.

- **Strukturvariable können an Funktionen übergeben werden**. Das funktioniert mittels ***Call by Value*** wie bei einfachen Variablen. Der Inhalt jedes Strukturelements wird der Funktion übergeben (außer wieder Arrays).

- **Funktionen können Strukturen als Rückgabetyp haben**, so kann eine Funktion "mehrere Werte" zurückliefern. Das Beispiel belegt eine Strukturvariable mit Werten. Der Rückgabetyp der Funktion `createpoint` ist `struct point`.

```
struct point createpoint(double x, double y, int farbe, char l)
{                                (die Übergabeparameter sind x, y, farbe und l)
  struct point dummy;
  dummy.spx = x; dummy.spy = y;
  dummy.farbe = farbe;           (gleiche Namen interferieren NICHT!)
  dummy.label = l;
  return dummy;
}
```

- Strukturen können geschachtelt werden. Dann hat man so etwas wie

```
neueintritt.geburtstag.tag = 17;
```

- In C++ können Instanzen von Strukturen ohne das zusätzliche Schlüsselwort `struct` (und ohne dass der neue Typ mit `typedef`, s.u. deklariert wurde) erzeugt werden. C++ erzeugt bei der Deklaration einer Struktur intern einen neuen Datentyp, dessen Namen identisch zum Namen der Struktur ist. Statt wie oben

```
   struct point punkt1, punkt2;
```

  reicht es also aus zu schreiben

```
   point punkt1, punkt2;
```

  Das gilt auch für `union` und `enum`.

- **Objekte / Klassen** der objektorientierten Programmierung sind Weiterentwicklungen von Strukturen!

Hier noch ein vollständiges, aber ziemlich sinnloses Beispiel – Strukturen erweisen sich erst bei komplexen Programmieraufgaben als sehr nützlich. Einige Beispiel werden uns in den nächsten Tagen noch über den Weg laufen.

```
#include <iostream>
using namespace std;
int main(void)
{
    struct mensch                          (Struktur mensch)
    {
       char name[20], first[9];
      int age;
    } ich;                                 (ich ist eine Variable vom Typ mensch)
    struct mensch du={"Maier","Sepp",50}, er;

    ich=du;                                (pauschale Kopie)
    er=du; er.age=22;                      (Zuweisung einzelner Elemente)

    cout<<ich.first<<' '<<ich.name<<' '<<ich.age<<endl;
    cout<< du.first<<' '<< du.name<<' '<< du.age<<endl;
    cout<< er.first<<' '<< er.name<<' '<< er.age<<endl;
}
```

liefert:
```
Sepp Maier 50
Sepp Maier 50
Sepp Maier 22
```

5.2 typedef

Mittels `typedef` lassen sich bereits bestehende, u.U. komplizierte Datentypen mit einem neuen Namen bezeichnen, z.B. auch wüste Pointer-Typen. Zur Syntax: Der neue Typname steht an genau der Stelle, an der ohne `typedef` der Variablenname stünde. Beispiele:

```
typedef unsigned long int bigint;          (bigint ist der neue Typ)

typedef struct
{    int i;
     float f;
     double df;
} collect;
```

`collect` ist hier also keine Strukturvariable, sondern der so neu definierte Typname. Oder:

```
typedef struct
{
     double re; double im;
} complex;
```

Variablendeklaration mit:
```
complex var1, var2;
```

5.3 `union` und `enum`

Syntaktisch sind **`union`** und `struct` analog (bis auf die Initialisierung). Der wesentliche Unterschied ist, dass eine Variable vom Typ `union` **zu einer Zeit immer nur eines der angegebener Elemente enthalten kann**. Interne Realisierung: die Elemente liegen im Speicher übereinander. Es ist Buchführung notwendig, um zu wissen, welcher Datentyp in welcher `union`-Variablen zuletzt abgespeichert wurde. Von der Verwendung dieses Konzepts wird daher abgeraten. Beispiel:

```
union int_or_double
{
     int n;
     double d;
} a_number;

a_number.n = 3;
a_number.d = 11.7;          (Jetzt ist die Zahl 3 wieder vergessen)
```

C(++) stellt genügend Speicherplatz für das größte Element der Variante bereit. Im Gegensatz zu Strukturen werden die `int`-Variable `n` und die `double`-Variable `d` am gleichen Speicherplatz abgelegt. Es lässt sich also **entweder** eine `int`-Zahl **oder** eine `double`-Zahl speichern – niemals beides zur selben Zeit.

Variable vom Typ **`enum`** (sog. Aufzählungstyp) sind bestimmte **mit Namen versehene Integer-Konstanten**; diese können bei der Typendefinition explizit angegeben werden:

```
enum wochentag {MO, DI, MI, DO, FR, SA, SO} termin;
enum wochentag fasttag;
```

`termin` und `fasttag` sind so dekl. Variable

`termin = DI; fasttag = FR;` *(*`termin` *hat den Wert* `1`, `fasttag` *den Wert* `4`*)*

Die Integer-Zuweisung erfolgt durch Weiterzählen vom letzten zugewiesenen Wert (Default ist 0), also: `MO = 0, DI = 1, ...` Schreibt man

```
enum monat {JAN=1, FEB, MAERZ, ...}
```

dann ist `JAN = 1, FEB = 2, MAERZ = 3, ...`

Kap 6: Der Präprozessor

Bevor ein C(++) - Quelltext compiliert wird, können noch **formale Ersetzungen** vorgenommen werden. Dies macht der sog. **Präprozessor**. Vor dem Aufruf des Compilers wird ihr Quelltext wie mit einem ganz normalen Editor überarbeitet, der überhaupt nichts von C(++) versteht.

Präprozessoranweisungen beginnen mit einem `#`. Da sie rein zeilenorientiert sind, stehen sie immer am **Zeilenanfang**, bzw. als erstes und einziges in einer Zeile und werden **nicht** durch einen `;` terminiert.

1. `#include` zum Einbinden von Files

- `#include <StdBibFilename>` setzt an diese Stelle das genannte File aus der Standardbibliothek (d.h. der Suchpfad ist implementationsabhängig).
- `#include "MyFilename"` dient zum Einbinden eigener Dateien.
- `#include`-Anweisungen können geschachtelt werden, d.h. es können Files eingebunden werden, die ihrerseits `#include`-Anweisungen enthalten.

2. `#define` zum (Um-)Definieren eines Tokens

- **Einfache Ersetzung**; Beispiel:
  `#define MAX 1000` bewirkt, dass ab dieser Anweisung alle `MAX` im Quellcode durch `1000` ersetzt werden. Konstanten sollte man niemals explizit in den Quellcode schreiben, sondern entweder über den Präprozessor definieren oder als C-Befehl codieren mit z.B.
  `const int MAX=1000;`
- Es ist Konvention (wenn auch syntaktisch nicht zwingend), dass so definierte Konstanten **großgeschrieben** werden. Weitere Beispiele:
  ```
  #define PLUS +
  #define ERROR printf("\nFehler!"); →  if (n<0) ERROR
  ```
- **Makroersetzung**; Beispiele:
  ```
  #define CMULT(x) (3.8*(x))
  #define SQ(x) ((x)*(x))
  #define MAX(a,b) ((a) > (b) ? (a) : (b))
  ```

  Vorteil: Makroersetzungen funktionieren – im Gegensatz zu Funktionen – für alle Typen.

  Nachteil: Die Wahrscheinlichkeit von Seiteneffekten ist hoch. So ist z.B. die exzessive Klammerung in den obigen Beispielen absolut notwendig:

```
#define SQR(x) x*x
SQR(z+1)  →   z+1*z+1
```

- C++ bietet **Inline-Funktionen**, die nach Möglichkeit verwendet werden sollten:

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

Jedes Mal, wenn `max(x,y)` im Programm vorkommt, kopiert der Compiler den Funktionstext an diese Stelle, es findet kein langsamer Funktions**aufruf** statt. Nachteil: dies funktioniert nur mit einem bestimmten Datentyp (aber dies wird mit Hilfe von sog. *Templates* doch noch ermöglicht).

- `#define` wirkt nicht innerhalb von `" "`.

- Die `#define`-Anweisung lässt sich rückgängig machen mit dem `#undef`-Befehl; Beispiel: `#undef MAX`

3. #if - Verzweigungen

Nochmals zur Erinnerung: dies sind *keine* C-Befehle, sondern Anweisungen an den Präprozessor.

```
#if MAX < 1000
     #define MIN 100
#else
     #define MIN 200
#endif
```

Allgemein wertet `#if` einen konstanten Integerausdruck aus; statt `<` kann irgendein Vergleichsoperator stehen. Weiterhin lässt sich mit `#ifdef name` verzweigen, wenn dieses Token definiert, bzw. mit `#ifndef name`, wenn es nicht definiert ist, Beispiel:

```
#ifdef DEBUG
     printf("a hat hier den Wert: %f\n", a);
#endif
```

4. Eigene Präprozessor-Fehlermeldungen

lassen sich mit `#error` ausgeben, was außerdem den Compilierungsvorgang abbricht. Beispiel:

```
#ifndef AFLAG
     #error "FLAG" nicht definiert!
#endif
```

Kap 7: Dateibearbeitung

Analog zur Eingabe von der Tastatur und Ausgabe auf dem Bildschirm gibt es natürlich auch in Standard-C Routinen zur Dateibearbeitung, z.B. `fprintf` oder `fscanf`. Da diese aber ebenso unpraktisch und fehleranfällig sind, wollen wir sie hier nicht näher behandeln, sondern gehen nur auf die File-IO von C++ ein.

In diesem Abschnitt geht es *nicht* darum, Dateien auf der Festplatte zu suchen, Verzeichnisse anzulegen oder zu löschen. Dafür werden fertige Standardfunktionen angeboten, z.B. in `dir.h` unter DOS. Es geht vielmehr darum, Inhalte in Dateien zu schreiben oder aus Dateien zu lesen. Hierbei ist zu beachten, dass **Dateien in C(++) nur unstrukturierte Sequenzen von Zeichen** sind. Eine Struktur – falls nötig – muss ihr Programm den Dateien geben.

Da die Klassenkonzepte von C++ erst später behandelt werden, fällt hier einiges vom Himmel; dieser Abschnitt ist einfach als **Kochrezept** aufzufassen. Außerdem wird nur ein Bruchteil der Möglichkeiten präsentiert. Die File-IO wird aber trotzdem schon jetzt behandelt, da sie für viele Programmieraufgaben unentbehrlich ist.

7.1 Basisfunktionalität

Um die Dateiein- und -ausgabe nutzen zu können, muss am Anfang `fstream` eingebunden werden. Man deklariert dann eine Variable vom Typ `ofstream` (Output File Stream) oder `ifstream` (Input File Stream), je nachdem ob in eine Datei geschrieben oder aus einer Datei gelesen werden soll. Die so definierte Variable wird beim Öffnen der Datei zugewiesen und symbolisiert im späteren Programmverlauf die Datei. Der Variablentyp `fstream` erlaubt lesen *und* schreiben gleichzeitig; dies sollte aber am Anfang lieber vermieden werden.

Folgendes Programm liest zwei Zahlen ein und schreibt diese, durch einen Tabulator getrennt, in die Datei *numbers.dat*.

```
#include <fstream>
#include <iostream>

using namespace std;

int main(void)
{
     double x, y;
     ofstream datei;

     cout << "2 Zahlen eingeben: ";
     cin >> x >> y;

     datei.open("numbers.dat", ios::out);
     datei << x << "\t" << y;
     datei.close();
}
```

Anmerkungen

- Das Schreiben in eine Datei erfolgt in drei Schritten:
  Datei öffnen – schreiben – Datei schließen

- Das Öffnen einer Datei geschieht mit dem Befehl `open`. In obigem Beispiel wird die Datei *numbers.dat* (im aktuellen Verzeichnis) geöffnet; der Schalter `ios::out` aktiviert den Schreibmodus. Einen Überblick über weitere Modi (z.B. Lesen) gibt die Tabelle in Abschnitt 7.2.

- In eine Datei kann genauso geschrieben werden wie mit `cout` auf den Bildschirm, man muss lediglich `cout` durch die Dateivariable ersetzen.

- Nach dem Schreiben sollte die Datei mit `close` wieder geschlossen werden. Dies geschieht zwar am Ende des Programms automatisch, wenn aber mehrmals in eine Datei geschrieben bzw. von einer Datei gelesen werden soll, dann ist das Schließen unbedingt erforderlich.

Das Lesen aus einer Datei geht analog, wie das folgende Beispiel zeigt. Hier wird das erste Wort aus der Datei *testdatei.txt* in die Variable `wort` eingelesen:

```
int main(void)
{
  char wort[30];
  ifstream demodat;

  demodat.open("testdatei.txt", ios::in);
  if (!demodat)
    {
      cout << "Fehler beim öffnen!";
      exit(1);
    }

  demodat >> wort;
  demodat.close();
}
```

Anmerkungen

- Der Variablentyp beim Einlesen ist `ifstream`.

- Nach dem Öffnen zum Lesen sollte man unbedingt prüfen, ob alles geklappt hat, da der Programmierer nicht sicher sein kann, ob die Datei überhaupt existiert. Falls ein Fehler auftritt, hat die Dateivariable den Wert 0 (falsch), was man abfangen kann und so ggf. das Programm beenden kann. Das Überprüfen nach dem Öffnen ist auch Schreiben in eine Datei ratsam.

- Beim Lesen mit `>>` analog zu `cin` wird bis zum ersten Whitespace (Leerzeichen, Tabulator, Zeilenwechsel) gelesen, oder bis zum ersten Zeichen, das nicht mit dem Variablentyp verträglich ist. Dies kann z.B. ein `'.'` sein, wenn man eine ganze Zahl einlesen will.

7.2 File-Modi

`ios::in`	Einlesen aus Datei; verhindert bei Ausgabedateien das Löschen.
`ios::out`	Schreiben in Datei; ohne weitere Angaben werden vorhandene Dateien überschrieben, nicht vorhandene Dateien neu angelegt.
`ios::app`	Ausgabe wird an Datei angehängt, der Rest bleibt erhalten.
`ios::noreplace`	Vorhandene Datei wird nicht überschreiben
`ios::nocreate`	Nicht vorhandene Datei wird nicht angelegt.
`ios::trunc`	Vorhandener Dateiinhalt wird gelöscht (Standardeinstellung)
`ios::ate`	Ausgabe wird an Datei angehängt. Nach erster Schreibanweisung kann der Dateizeiger auch auf alten Inhalt verschoben werden.

Mehrere Modi können mit *bitweisem Oder* verknüpft werden. Will man an eine bestehende Datei etwas anhängen und, falls die Datei noch nicht existiert, eine neue erzeugen, so verwendet man

```
demodat.open("C:\\verzeichnis\\datei.dat", ios::out|ios::app);
```

In diesem Beispiel ist `demodat` eine Variable vom Typ `ofstream`. Will man einen kompletten Pfad der Datei angeben, so ist zu beachten, dass man einen backslash \ durch durch \\ darstellen muss.

7.3 Der Dateizeiger

Der Dateizeiger spiegelt die aktuelle Lese- bzw. Schreibposition in einer Datei wider. Bei jedem Lese- bzw. Schreibvorgang wird der Dateizeiger automatisch weitergerückt. Es gibt aber auch die Möglichkeit, ihn manuell zu positionieren.

`pos=demodat.tellg()`	liefert eine Leseposition
`demodat.seekg(pos)`	setzt eine absolute Leseposition
`demodat.seekg(pos, bezug)`	setzt eine relative Leseposition
`pos=demodat.tellp()`	liefert eine Schreibposition
`demodat.seekp(pos)`	setzt eine absolute Schreibposition
`demodat.seekp(pos, bezug)`	setzt eine relative Schreibposition

`pos` ist eine ganze Zahl, die die Position in Bytes angibt; `bezug` kann folgende Werte haben:

`ios::beg`	relativ zum Dateianfang
`ios::cur`	relativ zum aktuellen Stand des Zeigers
`ios::end`	relativ zum Dateiende

Was häufig benötigt wird ist den Dateizeiger auf den Anfang der Datei zu positionieren. Dies geschieht mit `demodat.seekg(0)`.

7.4 Weitere Ein- und Ausgabemöglichkeiten

Neben den oben aufgeführten Möglichkeiten der Dateiein- und -ausgabe analog zu `cin` und `cout` gibt es noch weitere Optionen um z.B. eine ganze Zeile oder ein einzelnes Zeichen einzulesen. Die wichtigsten sind in der folgenden Tabelle aufgeführt.

`demodat.get(zeichen)`	liest genau ein Zeichen aus der Datei in die Variable `zeichen` (*char*) ein; liefert Null zurück bei einem Fehler (z.B. Dateiende).
`demodat.get(feld,anzahl)`	liest `anzahl` viele Zeichen in das char-Array feld ein. Das Einlesen bricht ***vor*** dem ersten Stopzeichen `'\n'` ab
`demodat.getline(feld,anzahl)`	liest eine Zeile aus der Datei in das char-Array `feld` ein, maximal aber `anzahl` Zeichen. Ist die Zeile länger, so wird der Rest abgeschnitten.
`demodat.read(feld,anzahl)`	liest `anzahl` viele Bytes in das Array `feld` ein ohne jegliche Inhaltsüberprüfung
`demodat.put(zeichen)`	schreibt genau ein Zeichen in die Datei
`demodat.write(feld,anzahl)`	schreibt `anzahl` viele Zeichen aus dem Array `feld` in die Datei ohne jegliche Inhaltsprüfung
`int demodat.good()`	gibt 0 zurück, wenn ein Fehler bei einem Stream-IO-Befehl aufgetreten ist (z.B. Dateiende bei Lesen)
`demodat.clear()`	setzt nach einer Fehlersituation den Stream wieder zurück, sonst kann weder gelesen noch geschrieben werden

Dies sind nur einige Möglichkeiten, die aber für die meisten Situationen ausreichen. Im Allgemeinen funktionieren die oben aufgeführten Möglichkeiten auch für `cin` und `cout`. So kann man z.B. mit

- `cin.get(zeichen)`

ein einzelnes Zeichen von der Tastatur in die Variable `zeichen` einlesen;

- `cin.getline(feld, 50)`

liest mehrere Wörter inklusive aller Leerzeichen bis zum Return, maximal aber 50 Zeichen.

Ein komplettes Beispiel

Das folgende Programm zeigt drei Varianten, um die Datei *demo.txt* einzulesen und deren Inhalt auf dem Bildschirm auszugeben. Zuerst wird wortweise gelesen, dann zeilenweise und schließlich zeichenweise, wobei zusätzlich Klein- in Großbuchstaben konvertiert werden.

In der Datei *demo.txt* soll z.B. folgender Inhalt stehen:

Dies soll eine kleine Demodatei sein.
2 Zahlen: 17.5 13
Letzte Zeile

```
#include <iostream>
#include <fstream>

include namespace std;

int main(void)
{
  char feld[80], c;
  ifstream lesdat;

  lesdat.open("C:\\ckurs\\demo.txt", ios::in);
  if (!lesdat)
  {
    cout << "\n Finde File nicht\n";
    return 1;
  }

  cout << "****** Lesen analog cin ******\n";
  while (lesdat.good())
  {
    lesdat >> feld;
    cout << feld << "\t";
  }

  cout << "\n****** Lesen zeilenweise ******\n";
  lesdat.clear();
  lesdat.seekg(0);
  while (lesdat.good())
  {
    lesdat.getline(feld,80);
    cout << feld << endl;
  }

  cout << "****** Lesen zeichenweise ******\n";
  lesdat.clear();
  lesdat.seekg(0);
  while(lesdat.get(c))
  {
    cout << char ( (c>='a' && c<='z')?(c+'A'-'a'):c );
  }

  lesdat.close();
  return 0;
}
```

Anmerkungen

- Nach dem Öffnen der Datei wird zuerst überprüft, ob die Datei existiert.
- Es wird jeweils solange gelesen, bis man an das Dateiende stößt. Dies wird mit `demodat.good()` festgestellt.
- Nachdem das Dateiende erreicht wurde, muss der Stream mit `demodat.clear()` entsperrt werden und dann der Dateizeiger auf den Anfang positioniert werden.
- Das Konvertieren von Klein- nach Großbuchstaben erreicht man durch eine Verschiebung im ASCII-Code, hier mit dem Fragezeichenoperator realisiert.

- Die Ausgabe dieses Programms sieht folgendermaßen aus:

```
****** Lesen analog cin ******
Dies     soll     eine     kleine     Demodatei     sein.     2
Zahlen:     17.5     13     Letzte     Zeile

****** Lesen zeilenweise ******
Dies soll eine kleine Demodatei sein.
2 Zahlen: 17.5 13
Letzte Zeile

****** Lesen zeichenweise ******
DIES SOLL EINE KLEINE DEMODATEI SEIN.
2 ZAHLEN: 17.5 13
LETZTE ZEILE
```

Kap 8: Pointer

Im nun folgenden wird abwechselnd von Adressen, Zeigern und Pointern die Rede sein; gemeint ist damit aber immer dasselbe: **Zeiger sind (Anfangs-)Adressen von Objekten (Variablen, Funktionen, . . .) eines Programms im Speicher**. Diese Objekte können auch über ihre Adressen angesprochen werden.

Ein Beispiel zur Verdeutlichung:

Speicheradresse	Inhalt	*Bedeutung*
20750	.....	
20752	17	*Variable i*
20754	.....	
20756	20752	*Pointer-Varibale ip*
20758	…..	

- Irgendwo im Speicher bei der Adresse `20752` hat das System also die Integer-Variable `i` abgelegt, der Wert sei `17`.

- Man kann in C eine Variable `ip` deklarieren vom Typ *Zeiger auf Integer*. Diese Variable möge das System bei der Speicheradresse `20756` plazieren.

- Weist man durch einen geeigneten C-Befehl der Variable `ip` die Adresse von `i` zu, so steht dann bei `20756` im Memory einfach die Zahl `20752`.

In diesem Kapitel wird alles primär Wichtige über Pointer gesagt. Nach Durcharbeitung sollten die Konzepte erst mal richtig eingeübt werden. Ein paar fortgeschrittene Pointer-Techniken sind in das folgende Kapitel ausgelagert.

8.1 Warum eigentlich Pointer?

Bevor wir in diese doch recht schwierige Materie einsteigen, zunächst einmal ein paar Motivationen. Die Arbeit mit Pointern – so häßlich sie nun einmal sein mag – ermöglicht u.a. folgendes:

- Rückgabe von mehreren Werten bei Funktionen
- Austausch von Feldern und Strings zwischen Funktionen (Nachbildung von *Call by Reference*), Manipulation von Feldern und Strings
- dynamische Speicherverwaltung, d.h. Anforderung und Freigabe von Speicherplatz zur Laufzeit
- Funktionsnamen können als Parameter übergeben werden. Sie schreiben z.B. in den Übungen eine Funktion `integriere(a,b,fu)`, die eine beliebige Funktion `fu` im Intervall [a,b] integriert, sei sie selbstdefiniert oder eine Standardfunktion wie `sin`.
- Wenn sie die mächtigen Standardbibliotheken von C nutzen wollen, müssen sie sich wohl oder übel mit Pointern beschäftigen.
- Behandlung komplexer Datenstrukturen wie z.B. verkettete Listen

8.2 Zeigeroperatoren

Adressoperator: **`&`** liefert die Adresse einer (bereits definierten) Variablen zurück.
`int a;` => Adresse von `a` ist **`&a`**

Wie die Adresse konkret ausschaut (meist eine lange `unsigned int` - Zahl), ist i.d.R. nicht von Interesse. Wer's unbedingt wissen will: `%p` bei `printf` gibt eine Adresse aus (implementierungsabhängig).

Inhaltsoperator: **`*`** gibt den Inhalt einer Speicherstelle mit einer bestimmten Adresse an.
`int a;` => `a = *(&a)`

Anderer Name für * : **Dereferenzierungsoperator** (ein Pointer enthält die Adresse, also die Referenz; * dagegen bezieht sich auf den Inhalt).

Beispiel:

```
float x = 3.8;
printf("Beh.: die Adresse von x ist %p, mit dem Inhalt %f\n", &x,x);
printf("Bew.: der Inhalt ist tatsaechlich %f\n", *(&x));
```

8.3 Zeigervariable

Zeigervariable sind Variable vom Typ *"Zeiger auf `type`"*, d.h. solche Variable, die die **Adressen** von Objekten eines Datentyps *`type`* enthalten. Ihre Deklaration geschieht auf **indirekte** Weise:

`type *name;`

Beispiel: `double *d;` definiert die Variable `d` vom Typ *Zeiger auf* `double`

`d` ist die Variable, die eine Adresse zu einer `double`-Zahl enthält. Auf den Inhalt der Adresse `d` wird mit `*d` zugegriffen, was eine `double`-Zahl ist. Deshalb die umständliche Definition mit `double *d`. Die Pointervariable `d` ist nicht mit Pointern auf eine Integerzahl kompatibel, obwohl intern beide Variablen lange Integers mit einer Speicheradresse darstellen).

Natürlich gilt wieder der Zusammenhang zwischen Adresse `d` und Inhalt `*d`:

`d = &(*d)`

Sie können sich diese Art der Deklaration auch anders merken und hinschreiben: Fasse einfach **`double*`** als Variablentyp "**Zeiger auf `double`**" auf.

`double* d;` definiert die Variable `d` vom Typ *Zeiger auf* `double`

8.4 Ein erstes Beispiel

Solche **Zeigervariable** sind auch oft **Funktionsargumente**, damit die Funktion die Inhalte an diesen Adressen nicht nur lesen, sondern auch verändern kann; dies ist die typische Methode, **wenn Funktionen mehrere** bzw. mehrdimensionale **Rückgabewerte** haben sollen.

Beispiel: eine Funktion, die die Inhalte zweier Variablen vertauscht:

```
int main(void)
{    int a, b;
     ...
     swap(&a, &b);                    (Funktionsaufruf)
     ...
}

void swap(int *n, int *m)             (Parameter: Pointer auf int)
{
     int park; park = *n;
     *n = *m;  *m = park;
}
```

Die Funktion muss wissen, **wo** die beiden Variablen im Speicher stehen, daher erfolgt der **Aufruf vom Hauptprogramm mit den Adressen der Variablen**. **In der Funktion** wird der **Inhalt dieser Speicherzellen modifiziert**. In der **Funktionsdeklaration steht `*n`**; `n` ist also eine lokale Pointervariable der Funktion, in die beim Aufruf die Adresse von `a` des Hauptprogrammes kopiert wird.

Beachte die ‚klassische' Falle:

- `*n` bei `swap(int *n,...)` signalisiert, dass vom Hauptprogramm die **Adresse** einer Variablen übergeben wird.
- `*n` bei `*n=*m` bedeutet, dass **Inhalte** zugewiesen werden.

8.5 Ein paar Hinweise

Hinweis 1: *Call by Reference*

Inhalte von lokalen Variablen (hier `a,b` im aufrufenden Programm) werden hier von einer anderen Funktion geändert. Das ist eigentlich typisch für *call by reference,* formal ist die Aussage aber richtig, dass es in C prinzipiell nur *call by value* gibt – die Adresse, die übergeben wird, wird ja nicht verändert.

Will man, dass der Inhalt einer Adresse nur gelesen werden soll, verwendet man den Qualifizierer *const*. Beispiel:

```
int show(const int *n)
{
return ((*n) + 1);        ( (*n)++ sollte Fehler/Warnung ergeben!)
}
```

Anmerkung: in C++ gibt's sog. Referenzen, z.B. `int i; int& iref=i;`
Damit kann man das Vertauschungsprogramm so schreiben:

```
int main(void)
{     int a, b;
      ...
      swap_r(a, b);                 (Funktionsaufruf ohne &)
      ...
}

void swap_r(int& n, int& m)       (Parameter: Referenzen)
{
      int park;  park = n;
      n = m;     m = park;
}
```

Das ganze ist auch noch effizienter als die obige Version in reinem C, bringt aber z.Z. zu viel Verwirrung. Daher sollte dieser Punkt zunächst weggelassen werden.

Hinweis 2: Ein typischer Fehler ist: `int *a; *a = 5;`

Wie jede andere Variable **hat eine Zeigervariable nicht von vorneherein einen** "vernünftigen" Inhalt, sondern es muss ihr erst ein solcher in Form eines **gültigen Speicherbereichs** zugewiesen werden! Wenn man das nicht macht, zeigt die Zeigervariable irgendwohin in den Speicher; eine Zuweisung der Form `*a = 5;` zerstört dort den (vermutlich wichtigen) Inhalt und **bringt u.U. den Rechner zum Absturz**.

Was aber jederzeit geht: `int b, *pb = &b;`

Falls dies unklar sein sollte: diese Zeile ist eine abgekürzte Schreibweise von

```
int b; int *pb;  pb=&b;
```

Hinweis 3: Nullpointer

Zeigervariablen dürfen auch konstante Werte zugewiesen werden. Bemerkenswert in diesem Zusammenhang ist der **Nullpointer**. Der Nullpointer ist keine gültige Speicheraddresse in dem Sinne, dass dorthin etwas geschrieben werden darf, sondern er findet Verwendung als **Fehlerindikator**, Markierung etc.. Der Nullpointer entspricht (meistens) `0`, ist aber maschinenabhängig definiert als `0` oder `0L` o.ä.; er ist deshalb standardmäßig bereits unter dem Namen `NULL` in `<stdio.h>`, etc. vordefiniert.

Beispiel: `float *ptr; ptr = NULL;`

Gemeint ist *nicht*: `*ptr=0;` dies würde bedeuten: *ptr* zeigt irgendwohin ins Nirwana; an diese Stelle schreiben wir dann 0 hinein...

Hinweis 4: Der strenge Compiler....

Schon unter ANSI-C und erst recht unter C++ ist der Compiler sehr streng bei der Typenprüfung. Ein Zeiger auf `int` ist nicht verträglich mit einem auf `float`. Will man ihn trotzdem zuweisen, so braucht man einen expliziten *Cast*. Will man offen lassen, auf welchen Datentyp ein Zeiger zeigt, verwendet man einen Zeiger vom Typ `void` (mit dem aber

natürlich keine Pointerarithmetik wie weiter unten geht.) Bei einer Zuweisung ist dann auch ein expliziter *Cast* nötig:

```
void *ptr; int *number; float result;

ptr=number;              (erlaubt)
ptr=&result;             (erlaubt)
number=ptr;              (Fehler)
number=&result;          (Fehler)
number=(int*)ptr;        (korrekt)
```

8.6 Zeiger und eindimensionale Arrays

Es gibt hier einen ganz **einfachen Zusammenhang** zwischen Pointern und Arrays.

Fall 1: Betrachte ein bereits definiertes Feld *type* `a[N];`

Dann ist automatisch auch eine **Zeigerkonstante** **`a`** (nicht Variable! Also keine Zuordnung möglich oder so etwas wie `a++` !) definiert mit

`a = &a[0]`

Der Zeiger `a` zeigt also auf das 0-te Element des Vektors.
Betrachten wir weitere Elemente. Für die Adressen gilt:

`a + n = &a[n]`

Dies gilt **unabhängig vom jeweiligen Datentyp**, der Compiler berücksichtigt ihn und seine Länge automatisch. Das nennt man Pointerarithmetik. Analoges gilt natürlich auch für die Inhalte:

`*(a+n)=a[n]`

Beispiel: `for (i=0; i<5; i++)  *(a+i) = 0;`

Fall 2: Betrachte eine bereits definierte Pointervariable *type* `*b;`

Im Gegensatz zur Array-Deklaration ist hiermit noch **kein Speicher reserviert**. Hat man dies aber selbst besorgt, so existiert dann auch ein Array `b[..]`, man kann statt `*(b+5)` z.B. auch `b[5]` schreiben. Auch geht in diesem Fall z.B. `b++` , da `b` ja eine Variable ist.

Eigentlich **gibt es** in C(++) **nur eindimensionale Arrays** oder Vektoren. Eine Matrix ist einfach ein Vektor aus dem zusammengesetzten Datentyp *Zeile*. Dazu mehr im nächsten Kapitel. In diesem Kapitel reichen uns echt **ein**dimensionalen Arrays aus elementaren Datentypen.

Beispiel:
```
int a[8]; *p;
p=a+3;
for (i=-3; i<5; i++)  p[i]=i*i; (p[i] entspricht *(a+3+i))
```

Übergabe von eindimensionalen Arrays an Funktionen

Die **Übergabe** erfolgt ganz einfach durch Angabe der **Anfangsadresse**. Daher steht im Hauptprogramm so etwas wie:

```
double vect[N], laenge;
     ...
laenge = norm(vect, N);        (Funktionsaufruf mit Adresse von vect[0])
```

Die Funktions-Deklaration und -Definition mit Pointer auf `double` schaut so aus:

```
double norm(const double *v, int dim)
alternative Schreibweise:
double norm(const double v[], int dim)

{
     int i;
     double sqsum;
     for(i=0, sqsum=0.0; i<dim; i++)
          sqsum += v[i]*v[i];
     return sqrt(sqsum);
}
```

Im Unterprogramm darf `v[i]` verwendet werden, da ja im Funktionskopf `*v` oder `v[]` aufgeführt ist.

8.7 Pointerarithmetik

Obige Äquivalenz zwischen `a[n]` und `*(a+n)` ist möglich, weil – wie erwähnt – intern die Konvertierung von `n` in einen Zeiger vom jeweiligen Typ erfolgt, also zur Adresse in `a` tatsächlich **`n*sizeof(*a)`**-Byte addiert werden. Dies wird als **Pointerarithmetik** bezeichnet. **Deshalb macht für Zeiger auf Elemente desselben Arrays auch ein numerischer Vergleich oder Addition bzw. Subtraktion Sinn.** Beispiele:

```
float matr[10], *p1, *p2;
long a, b;
p1 = matr + 1;           (p1 zeigt auf matr[1])
p2 = matr + 9;           (p2 auf matr[9])
a = (long)(p2-p1);       (=> a=8 (8 Elemente Differenz))
b = (long)p2-(long)p1;   (=> b=32, oder jedenfalls 8*sizeof(float))
```

Anmerkung am Rande: Die Pointerarithmetik wird intern sehr konsequent angewandt; mit z.B. `float a[10]; int n;` gilt die Äquivalenz:

```
a[n] = *(a+n) = *(n+a) = n[a],
```

da `n` in einen `(float*)` konvertiert wird, und bei der Addition die Reihenfolge keine Rolle spielt. Das ist wirklich so! Das folgende Prögrämmchen ist syntaktisch korrekt und gibt auch zweimal eine 3 aus:

```
int main(void)
{
      int a[5]={1,2,3,4,5};
      int n=2;
      printf("%d %d",a[n],n[a]);
}
```

Was macht der Compiler mit `n[a]`? Der Ausdruck `n[a]` wird vom Compiler zu `*(n+a)` gewandelt, die beiden Zahlen `n` und `a` werden addiert (ohne Rücksicht darauf, dass in `a` eine Adresse steht, in `n` einfach 2). Das Ergebnis wird als Adresse interpretiert, deren Inhalt ausgegeben wird.

Was lernen wir daraus? **C-Compiler sind sehr einfach gestrickt und gehen nach ganz formalen Regeln vor, sind allerdings sehr effizient und erzeugen einen schnellen Maschinencode**.

Weiteres Beispiel: ***for*-Schleife mit Pointern** (Kopieren von Feldern)

Die Schleife über alle Feldelemente kann man natürlich ganz 'normal' machen mit `p[i]=q[i]; (i=0..N-1)`. Mit Pointerarithmetik geht's aber auch anders:

```
int p[10],q[10],i, *aux1, *aux2;

for (i=0;           i<10;        i++)            p[i]=q[i];
for (aux1=q,aux2=p; aux2<(p+10); *aux2++=*aux1++)          ;
```

Die Hilfsvariablen sind notwendig, da ja `p` und `q` Pointer-Konstanten sind und nicht verändert werden dürfen.

8.8 Zeiger auf Strukturen

Strukturpointer finden so häufig Verwendung, dass es dafür eine eigene Syntax gibt; bei gegebener Adresse auf eine Strukturvariable werden deren Elemente mit `'->'` angesprochen, in der Form: *`Zeiger auf Strukturvariable`*`->`*`Elementname`*.

Beispiel: Strukturdefinition
```
struct point
{
  double px, py;
  int farbe;
};
```
Ist ein Zeiger auf die Struktur `point` deklariert mit
```
     struct point *p;
```
so gibt es zwei Arten, um auf die Strukturvariable `px` zuzugreifen:
```
     (*p).px=17.3;        oder        p->px = 17.3;
```

Bei einem Feld `f[100]` gibt's die Versionen
```
     f[20].px=29.5;       oder        (f+20)->px = 29.5
```
da ja bei Arrays der Name schon die Adresse darstellt.

8.9 Zeiger auf Funktionen

Auch Funktionen haben Anfangsadressen im Speicher. Funktionen werden als solche am **Klammerpaar** hinter ihrem Namen erkannt, dabei hat **der Name allein** (ohne Klammern), analog zu den Arrays, bereits **deren Adresse als Wert**; dies gilt für Bibliotheksroutinen ebenso, wie für eigene Funktionen.

Betrachte eine ganz normale Funktion

```
int myfunc(double);        (das ist der Prototyp)
```

Die Funktion kann aufgerufen werden mit

```
myfunc(3.8);    oder   (*myfunc)(3.8);        analog:
sin(PI/3.0);    oder   (*sin)(PI/3.0);
```

Plausibel machen kann man sich's so: `sin` zeigt ja auf einen Memory-Block, wo die Sinusroutine steht. `(*sin)` ist so etwas wie der Beginn der Sinusroutine.

Sie wollen nun z.B. eine Routine schreiben, die eine beliebige Funktion integriert. **Die** zu integrierende **Funktion soll der Routine als Parameter übergeben werden**. Das ist kein Problem:

```
#include <iostream>
#include <math.h>
using namespace std;
double dumint(double (*f)(double x))
  {
    return f(0);          (statt integrieren ...)
  }
int main(void)
{
cout << dumint(sin) << dumint(cos)  << dumint(exp);
}
```

Was ist zu beobachten: Der einzige Parameter der Routine `dumint` ist die Adresse der Funktion, die integriert werden soll. Es gibt aber in C(++) nicht einfach einen *Pointer auf eine Funktion.* Aufgrund der strengen Typenprüfung muss man angeben:

1. welche Art von Rückgabewert die Funktion hat (hier `double`*)*
2. wie die Parameterliste ausschaut (hier: nur ein Parameter vom Typ `double`*)*

`double *f(double x)` oder `double* f(double x)` (ohne die Klammern um `*f`) wäre falsch, weil dies aussagt, dass `f` einen *Zeiger* auf `double` zurückliefert, siehe gleich anschließend.

`sin`, `cos`, `exp` usw. passen zu dieser Deklaration. Deswegen kann man einfach hinschreiben

```
cout << dumint(sin);
```

`sin` ist ja die Anfangsadresse des Sinus-Routine.

Was **nicht** gehen kann: `dumint(sin*cos);` `sin*cos` ist ja keine Routine, hat also auch keine Anfangsadresse. Abhilfe: definiere Hilfsfunktion `g(x) = sin(x)*cos(x);` deren Adresse können sie natürlich wieder übergeben.

Deklaration von Funktionspointern

Bisher haben wir Funktionspointer nur implizit benutzt. Jetzt deklarieren wir einfach mal einen. Wie oben gilt: **Bei der Deklaration einer Zeigervariablen auf eine Funktion muss deren Variablenliste und Rückgabetyp angegeben werden.**

Beispiel

```
int myfunc(double, int);          (Prototyp einer Funktion)
int main(void)
{
int n;
int (*demofptr)(double,int);      (Rueckgabe: int, Parameter: double, int)
demofptr = myfunc;
n = demofptr(2.9, 3);             (neuer Name fuer myfunc)
          ...
}
```

Anders gesprochen: man definiert die Variable `demofptr` so, dass `(*demofptr)` eine Funktion mit einem `double` und einem `int`-Parameter ist.

Randanmerkung: Natürlich sind auch **Arrays von Funktionspointern** möglich, Beispiel:

```
double (*trig[3])(double), x;
     trig[0] = exp;               (exp ist die Adresse der exp-Funktion)
     trig[1] = sin;
     trig[2] = cos;
```

Die Aufrufe z.B. `sin(x)` und `trig[1](x)` sind dann äquivalent; Funktionen werden so also mit einem Index versehen, was manchmal ganz elegant sein kann.

Noch 'ne Anmerkung: Natürlich gibt's andersherum auch **Funktionen, die Pointer zurückliefern**, z.B.

```
char *strchr(char *s, int c);
```

Hier wird der String `s` durchsucht und ein Zeiger auf die erste Fundstelle des Zeichens `c` zurückgeliefert (oder NULL, wenn `c` nicht enthalten ist).

8.10 Dynamische Speicherallozііerung

Unter dynamischer Speicherallozііerung versteht man die **Belegung von Speicherplatz variabler Grösse zur Laufzeit**. Arrays müssen also nicht sicherheitshalber weit überdimensioniert, die Programme nicht bei jeder Größenänderung neu compiliert werden. Man kann jetzt z.B. die Arraygröße einlesen und dann genau so viel Speicher beschaffen wie notwendig ist. Allerdings ist ein solcher Code vom Compiler **schlechter zu optimieren**.

Speicher-Alloziierung in Standard-C

Die klassischen Funktionen aus der C-**Standardbibliothek** sind (u.a.):

- `size_t` ist ein maschinenabhängig definierter Datentyp (enspricht meist `unsigned int`, definiert in `<stdlib.h>`, `<stdio.h>`, `<stddef.h>`, etc.).

- `void *malloc(size_t size)` **belegt** `size` Bytes großen **zusammen hängenden Speicherbereich** und liefert die Anfangsadresse davon zurück, benutzbar für jeglichen Variablentyp.

- `malloc` liefert bei Fehlern (z.B. **wenn der angeforderte Speicherplatz nicht zur Verfügung steh**t) den Nullpointer NULL zurück. → Nach jedem Aufruf sollte deshalb deren Rückgabewert **getestet** werden!

- `void free(void *)` wird schließlich verwendet, um nicht mehr benötigten dynamisch belegten Speicherplatz wieder freizugeben.

Für `malloc` und `free` wird das Headerfile <`stdlib.h`> oder <`alloc.h`> benötigt. Wie oben ersichtlich, ist der **Rückgabewert** von `malloc` vom Typ **`(void*)`**, d.h. **ein generischer Pointer**, also gewissermaßen eine „Adresse als solche und schlechthin". Wie erwähnt: Variablen vom Typ `(void*)` können Pointer auf beliebige Datentypen aufnehmen. Umgekehrt muss man allerdings 'casten'. Die klassischen Alloziier-Funktionen sind daher etwas umständlich, z.B.:

```
matrix = (double*)malloc(dimx*sizeof(double));
```

C++ - Funktionen zur dynamischen Speicher-Allozierung

`new` und `delete` erlauben ein viel einfacheres Handling, außerdem sind sie auch bei der Objektorientierten Programmierung zu verwenden. Man braucht keine Variablenlänge angeben, sie wird aus dem Variablentyp automatisch bestimmt. Der zurückgegebene Pointer hat schon den korrekten Typ, ein *Casting* ist nicht nötig. Das folgende Beispiel erläutert eigentlich alles. `new` und `delete` sind integraler Bestandteil von C++, man benötigt dafür also keine Header-Dateien.

```
#include <iostream>                 (für cout)
#include <stdlib.h>                 (für exit)
using namespace std;


int main(void)
{
int *a, *b, (*c)[10];               (c: 'Matrix' mit 10 Spalten)

a = new int;                        (Platz für einfache Variable)
*a = 5;                             (und gleich belegen)
```

```
if (!(b = new int[100]))          (Platz beschaffen; Nullpointer?)
   {
   cout << "Insufficient memory\n"; exit (1);
   }

if (!(c = new int[100][10]))      (100 Zeilen zu je 10 Spalten)
   {
   cout << "Insufficient memory\n"; exit (1);
   }

for (int i=0; i<10; i++)          (als Demo einige Elemente belegen)
  {  c[i][i]=i; cout << c[i][i] << " "; }

delete a;                         (alles wieder freigeben)
delete b; delete c;
}
```

Beachte folgende Besonderheit im obigen Programm:

`int (*c)[10];` ist *ein* Pointer auf einen Vektor aus 10 Integers. Nur über so eine Deklaration ist die Speicherallozierung mit `c=new int[100][10]` möglich. Kontrolle: `sizeof` liefert 2 oder 4.

`int *c[10];` wäre ein Vektor aus *10 Pointern* auf Integerzahlen. Kontrolle: `sizeof` liefert 20 oder 40.

Kap 9: Fortgeschrittenere Programmiertechniken in C

9.1 Zeiger auf mehrdimensionale Arrays

Zur Vorbemerkung und Erinnerung: **mehrdimensionale Arrays sind intern eindimensional angelegt**, die **Arrayzeilen liegen alle hintereinander**.
Beispielsweise ist

`float f[N] [M];`
Zeilen Spalten
abgespeichert als `f[0][0] ... f[0][M-1] f[1][0] ... f[1][M-1] ...`

n=0	m=0	...	m=M-1
n=1			
...			
n=N-1			

Die Schreibweise `float f[N][M]` soll schon nahe legen, dass es sich um ein (**eindimensionales) Feld aus `N` Elementen handelt. Diese Elemente wiederum sind ganze Zeilen, bestehend aus M Einzelvariablen.** Wird die 2. Dimension nicht angegeben, so meint man die Anfangsadresse dieser Elemente, also der Zeilen:

Zeigerkonstante `f[0]` zeigt auf 0. Element, also auf Beginn der 0. Zeile
Zeigerkonstante `f[1]` zeigt auf 1. Element, also auf Beginn der 1. Zeile

Diese Zeigerkonstanten sind also schon automatisch mitdeklariert.

Wie bei 1D-Arrays gilt: Pointerkonstante
`f = &f[0]` und `*f = f[0]` = **Zeiger** auf Beginn der 0.Zeile

Der Name `f` zeigt also auf den Beginn eines Feldes aus N Elementen, wobei jedes Element wieder aus M `float`s besteht. **`f[0]` ist noch keine Float-Zahl, sondern ein weiterer Zeiger**. Dieser letztendlich zeigt auf das `float`-Element `f[0][0]`. Also ist

`f[0] = &f[0][0]`.

Intern gilt wieder die übliche Indexarithmetik in Kombination mit der Zeigerarithmetik:

```
f[i][j]   = *(f[i]+j)
              ------
               ^Adresse der Zeile i  + j floats dazu

          = *((f[0]+i) + j)
               ------
                    ^Adresse der Zeile 0 + i Zeilen dazu

          = *((f[0]) +i*M +j)        M = Zeilenlänge

          = * (*f    +i*M + j)      speziell:   f[0][0]=**f
```

Nützliche Hilfs-Konstruktion: Vektor mit Adressen der Zeilenanfänge

Betrachte folgendes Beispiel mit einem Feld `matrix[N][M]`:

```
int matrix[N][M], *mvec[N], i;
for(i=0; i<N; i++)
     mvec[i] = matrix[i];
```

Das Hilfsfeld `mvec` enthält nun alle Adressen der *N* Zeilenanfänge. Der Vorteil ist u.a., dass diese Adressen nicht jedesmal neu berechnet werden müssen. Man spricht in diesem Fall **von echt-zweidimensionalen** Feldern. Nutzt man kein Hilfsfeld, so spricht man von **pseudo-zweidimensionalen Feldern**.

Aufgrund der im Pointer-Kapitel beschriebenen Äquivalenz zwischen Arrays und Pointern gilt auch Folgendes :

> **Sowohl `matrix[3][4]` als auch `mvec[3][4]` sind gültige Verweise auf den gleichen `int`-Wert.**

9.2 Übergabe von mehrdimensionalen Arrays an Funktionen

Wie bei eindimensionalen Arrays wird die Anfangsadresse übergeben. Der Zwittercharakter zwischen Ein- und Mehrdimensionalität führt aber zu großen Problemen. Es hat wenig Sinn, darüber zu philosophieren; wir zeigen im folgenden Demoprogramm ganz pragmatisch, wie man statisch und dynamisch alloziierte Arrays an eine Funktion übergibt.

Beispiel für statische Arrays:

```
int main(void)
{
  int stat[3][5];               (statisches Array)
  int *hilf[3]; int i;          (Hilfsvariable)

  for (i=0; i<3; i++)
    hilf[i] = stat[i];          (Konstruktion eines echt-2-dimensionalen Arrays)

  Ausgabe(hilf, 3, 5);          (Aufruf der Funktion)
}
```

Beispiel für dynamisch alloziierte Arrays:

```
int main(void)
{
  int **dyn;                    (Dies wird das Array)
  int N=4, M=7, i;

  dyn = new int*[N];            (Konstruktion eines echt-2-dimensionalen Arrays)
  for (i=0; i<N; i++) dyn[i] = new int[M];

  Ausgabe(dyn, N, M);           (Aufruf der Funktion)
}
```

Die Funktion *Ausgabe* kann in beiden Fällen so aussehen:

```
void Ausgabe(int **a, int dimX, int dimY)
{
  int i, j;
  for (i=0; i<dimX; i++)
    for (j=0; i<dimY; j++)
      printf("%d ", a[i][j]);     (a wird wie ein gewöhnliches Array angesprochen)
}
```

Die beiden oben gezeigten Möglichkeiten der Variablenübergabe sind universell einsetzbar. Für die Übergabe von statischen Arrays gibt es noch eine einfachere Möglichkeit ohne Hilfsarray, die allerdings nicht von allen Compilern akzeptiert wird:

```
int main(void)
{
  int stat[3][5];                 (statisches Array)
  Ausgabe(stat);
}

void Ausgabe(int a[][5])          (Die Zeilenlänge muß angegeben werden!)
{
  ...
}
```

9.3 Kommandozeilenparameter

Unter Verwendung von Kommandozeilen-Parametern versteht man die Übergabe von Argumenten an das Hauptprogramm, also an die Funktion `main`.
Damit dies funktioniert, muss man `main` so deklarieren:

```
int main(int argc, char **argv)
```

Dabei gibt `argc` die Zahl der Argumente an; `argv` ist ein Array, in dem diese Argumente in Form von Strings vorliegen. Die Bezeichnungen `argc` und `argv` sind Konvention, aber syntaktisch nicht vorgeschrieben.

Beispiel: ein Programm heißt *prog.exe* und wird aufgerufen mit
`prog myfile 1 3.8`
Dann sind die Argumente von `main` folgendermaßen belegt:

`argc`	4 (Anzahl der Argumente)
`argv[0]`	`... <Pfad>... \prog.exe` (DOS-Version)
`argv[1]`	`myfile`
`argv[2]`	`1`
`argv[3]`	`3.8`
`argv[4]`	`NULL`

Um diese Strings in Zahlen zu konvertieren, stehen z.B. die Funktionen
`int atoi(char*)` und `atof(char*)` (deklariert in `<stdlib.h>`) zur Verfügung.

9.4 Generische Funktionen

Funktionspointer und Funktionsvariable erlauben es, **generische Funktionen** zu schreiben. Das sind Funktionen, **die mit beliebigen Datentypen funktionieren** (also sind die Übergabeparameter vom Typ `void` bzw. `*void`. Als Beispiel sei hier die Bibliotheksfunktion `qsort()` aus `<stdlib.h>` angeführt:

```
void qsort(void *base, size_t nel, size_t width,
     int (*compar) (const void *, const void *));
```

`qsort()` sortiert ein Array `base[0]` bis `base[nel-1]` von Objekten der Größe `width` in aufsteigender Reihenfolge. Die Vergleichsfunktion `compar()` , die man selbst schreiben muss, und die `qsort` zur Verfügung gestellt wird, muss einen negativen Wert zurückgeben, wenn ihr erstes Argument kleiner ist als das zweite, Null wenn die Argumente gleich sind und einen positiven Wert, wenn das erste größer als das zweite ist. In `qsort()` werden Vergleiche immer mit der Funktion `compar()` vorgenommen:

```
void qsort(void *base, size_t nel, size_t width,
     int (*compar) (const void *, const void *))
{
     ...
     if ((*compar)(base[i], base[j]) < 0)
          swap(base, i, j);
     ...
}
```

(Type `size_t` = meist `unsigned int`)

Mit einer **geeigneten Vergleichsfunktion** können also **Arrays beliebigen Typs** sortiert werden. Beispiel:

```
#include <stdlib.h>
#include <string.h>

int agecompare(const void *i, const void *j);          (Alter vergleichen ...)
int namecompare(const void *i, const void *j);         (Namen vergleichen ...)

typedef struct
     {     char *name;
           int age; } person;

int main(void)
{     person a[ARRAYSIZE];

     ...  Initialisierungen ...
     (a nach Alter sortieren:)
     qsort(a, ARRAYSIZE, sizeof(person), agecompare);
                              (agecompare ist die Adresse der Funktion OHNE Klammern)
     ...
     (a nach Namen sortieren:)
     qsort(a, ARRAYSIZE, sizeof(person), namecompare);
     ...
}
```

```
int agecompare(const void *i, const void *j)
```
(Uebergabe des `i`*-ten und* `j`*-ten Elements der Struktur, aber als* `void*`*)*
```
{     int ii,jj;

      ii = ((person*)i)->age;       (cast von void* nach person*)
      jj = ((person*)j)->age;       (nur so ist -> ueberhaupt definiert)

      if ( ii > jj) return 1;
      if ( jj > ii) return -1;
      return 0;
}

int namecompare(const void *i, const void *j)
{     char *ii, *jj;
      ii = ((person*)i)->name;
      jj = ((person*)j)->name;
      return strcmp(ii, jj);        (Vergleich von Zeichenketten aus <string.h>)
}
```

9.5 Verkettete Listen

Dies ist eine klassische Struktur der Informatik. Sie findet z.B. Anwendung in der Festplattenorganisation unter Unix. Sehr nützlich ist so etwas auch bei einer Datenbank. Worum geht's:

- ein Kettenglied besteht aus einem Nutzinhalt und einem Zeiger auf ein weiteres Kettenglied.
- das letzte Kettenelement enthält den Null-Pointer.
- die einzelnen Kettenglieder können irgendwo im Speicher liegen, i.a. *nicht* benachbart.
- es ist sehr einfach, an einer beliebigen Stelle *N* der Kette ein neues Element einzufügen: alloziere Speicherplatz für ein neues Element, Zeiger von Element *N* zeigt auf neues Element, Zeiger des neuen auf Element *N+1*.

Das folgende Demoprogramm ist so gegliedert:
1. erzeuge Kettenanfang
2. erzeuge 10 neue Kettenglieder und füge sie an
3. füge nach dem 7. Element ein neues ein
4. lösche das dritte Element
5. drucke Kette aus zur Kontrolle
6. lösche Kette wieder (geht nur sequentiell!)

Für ein Kettenelement wird eine Struktur `self` und ein zusammengesetzter Datentyp `link` definiert:

```
typedef struct self           (Struktur mit Zeiger auf Struktur gleicher Art)
{ int zahl;
  struct self *next; } link;
```

`next` ist also ein Pointer, er zeigt wieder auf eine Struktur vom Typ `self`.

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

using namespace std;

typedef struct self                  (Struktur mit Zeiger auf Struktur gleicher Art)
{ int zahl;
  struct self *next; } link;         (next ist ein Pointer, der INHALT von next ist
                                      wieder eine Struktur vom Typ link)

int  main(void)
{
   int  n=10, i;
   link  *anfang, *kette, *hilf;

   if (!(anfang = kette = new link)) exit(1);   (Den Anfang der Kette herstellen)
   anfang->next = NULL;                         (... und auf NULL zeigen lassen)

   for(i=1; i<=n; i++)                    (n neue Kettenglieder anfuegen)
   {
     if (!(hilf=new link)) exit(1);       (Erzeugen irgendwo im Speicher)
                                (hilf ist Adresse des Kettenelements, daher Syntax hilf->zahl)
     hilf->zahl = i;            (irgendwie mit Inhalt füllen)
     hilf->next = NULL;         (Ende markieren)
     kette->next = hilf;        (anhängen)
     kette = hilf;              (weiterrücken)
                  (kette zeigt immer auf den ANFANG des gerade aktuellen Elements)
   }

                  (nach dem 7. Element ein neues Kettenglied einfuegen)
   for(i=0, kette=anfang; i<7; i++) kette=kette->next;      (zum 7. Element gehen)
   if (!(hilf = new link)) exit(1);
   hilf->zahl = 100;            (Den Wert 100 einsetzen)
   hilf->next = kette->next;    (einfügen)
   kette->next = hilf;

                  (das 3. Element loeschen)
   for(i=0, kette=anfang; i<2; i++, kette=kette->next);
   hilf = kette->next;                (Adresse merken)
   kette->next = kette->next->next; (TRICKY! Zeiger auf das übernächste Element setzen)
   delete hilf;                       (Speicher freigeben)

                  (Kette ausgeben)
   kette = anfang->next; cout << "Das ist der Inhalt der Kette: ";
   while(kette != NULL)
     {
       cout << kette->zahl <<' ';
       kette = kette->next;
     }
                  (Output: 1 2 4 5 6 7 100 8 9 10)

                  (die ganze Liste wieder löschen)
   while(anfang->next!=NULL)
    {
      hilf = anfang->next; anfang->next = anfang->next->next;
      delete hilf;
    }
   delete anfang; cout << endl;;
   return EXIT_SUCCESS;
}
```

Kap 10: Erste Schritte der *OOP* mit C++

Wie wir bisher gesehen haben, kann man mit C++ ganz 'klassisch' prozedural programmieren. Wir formulieren Anweisungen, um Daten zu manipulieren. Die Reihenfolge ist durch das Programm vorgegeben.

Bei der objektorientierten Programmierung werden **Daten und Anweisungen** immer zu Einheiten verschmolzen, genannt **Objekte**. Objekte können nur durch ihre jeweils spezifischen Anweisungen manipuliert werden, auf sie kann nur über definierte Schnittstellen zugegriffen werden. Besonders praktisch ist dies bei ereignisgesteuerten Programmen wie einem Fenstersystem, wo der Programmierer noch gar nicht weiß, in welcher Reihenfolge die Aktionen ablaufen sollen. Wenn der **Bediener** des Programmes z.B. ein Fenster verschiebt oder verkleinert, muss der Inhalt neu gezeichnet werden. Das Fenster ist dann das Objekt, es selbst kennt die Funktion, die in diesem Fall aufgerufen werden müssen.

10.1 Objekte und Klassen in C++

Statt von Objekten spricht man in C++ von **Klassen**. Klassen sind ein abstraktes Abbild einer bestimmten Realität, Programmiertechnisch ist eine Klasse eine **Struktur**, die enthält:

1. **private Daten**, die nur innerhalb der Klasse verfügbar sind
2. **private Methoden oder Implementations-Funktionen**, die nur innerhalb der Klasse verfügbar sind
3. **öffentliche Daten** für die Kommunikation mit anderen Objekten
4. **öffentliche Methoden oder Interfacefunktionen,** über die von außen her auf das Objekt zugegriffen werden kann

Definition einer Klasse **(**ähnlich zu `struct`, aber mit *public/private* - Bereichen)

```
class winclass
{
     public:
          ... oeffentliche Daten ...
          ... oeffentliche Funktionen
     private:
          ... private Daten ...
          ... private Funktionen ...
};
```

Öffentliche Daten könnten z.B. die Koordinaten des Fensters und die Farbe sein. Interfacefunktionen könnten z.B. vorhanden sein zum Verkleinern und Ikonisieren von Fenstern. *Wie* das alles gemacht wird, ist Sache des privaten Bereiches und interessiert außerhalb des Objekts überhaupt nicht. Man spricht von **Kapselung.**

Eine Klasse ist eine abstrakte Struktur, z.B. für Bildschirmfenster. Jedes reale Fenster ist dann eine **Instanz** der Klasse. Jede Instanz muss – wieder analog zu `structs` – explizit erzeugt werden:

```
int int main(void)
{    winclass fenster1, fenster2;
     .....
```

Äußerst praktisch ist das Konzept der **Vererbung.** Die **abgeleitete Klasse** aller verschiebbaren Fenster `movwins` z.B. kennt alle (öffentlichen) Methoden der **Basisklasse** `winclass`, zusätzlich gibt es Methoden zum Verschieben:

```
class movwins: public winclass
{
     ... zusaetzliche Daten und Funktionen ...
};
```

10.2 Erste Programmfragmente

Ein Fenstersystem ist zu komplex, um in diesem Kurs behandelt zu werden. Daher wird das elementare Klassen-Handling in C++ an folgendem Beispiel erläutert: Die Klasse `fraction` soll die Menge aller Brüche mit ganzen Zahlen darstellen:

```
long           num;          Zähler
unsigned long  denom;        Nenner
```

Welche Methoden sind nun sinnvoll? Hier reicht:

- Wertzuweisung zu einem Objekt der Klasse
- Ausgabe des Bruches am Bildschirm
- arithmetische Operationen

Erster Schritt:

```
class fraction
{
public:
     long demo;                              (öffentliche Testvariable)
     long getNum();                          (Zähler abfragen)
     unsigned long getDenom();               (Nenner abfragen)
     void setNumDenom(long n, long d);       (Zähler und Nenner setzen)
     void print();                           (Ausgabe)

private:
     long num;                               (Zähler)
     unsigned long denom;                    (Nenner)
     void reduce();                          (kürzen)
}

int main(void)
{
     fraction var1,var2;                     (Instanzen erzeugen)
     var1.num=100;                           (geht NICHT! privat)
     var1.demo=299;                          (GEHT!)
}
```

Was beobachten wir? Die privaten Variablen `num` und `denom` sind von außen her **nicht** ansprechbar, auf sie kann nur über die öffentlichen Interfacefunktionen zugegriffen werden.
Es geht aber im Hauptprogramm: `var1=var2` (komplette Zuweisung von Instanzen)

Natürlich können Instanzen auch über **Zeiger** erzeugt werden**:**

```
fraction *var;          (erzeugt einen Zeiger auf die Instanz)
var = new fraction      (jetzt ist auch der Speicherplatz da)
var->demo=0;            (erlaubter Zugriff auf public)
var->num=1;             (geht NICHT! privat)
...
delete var;
```

Definition und Deklaration der sog. Member- oder Element-Funktionen
(d.h. der Methoden einer Klasse)

Die **Deklaration** erfolgt wie üblich über Prototypen innerhalb der Klassendeklaration.
Die **Definition** kann entweder zusammen mit der Deklaration erfolgen (für kurze (Inline-) Funktionen sinnvoll) oder außerhalb der Klassendeklaration. Dann muss allerdings dabeistehen, für welche Klasse sie gilt. Dies geschieht mit dem neuen **Bezugsoperator `::`**

Fall 1:
```
class fraction
{
public:
     long getNum()                  (Deklaration UND Definition)
          {return(num);}
.....
}
```

Fall 2:
```
class  fraction
{
public:
     void print(void)               (nur Deklaration, Prototyp)
.....
}
void fraction::print()              (für fraction-Members)
{ cout << num << '/' << denom; }
```

10.3 Ein erstes vollständiges objektorientiertes Programm

```
#include <iostream>
#include <process.h>            (für exit)
#include <math.h>               (für labs)
using namespace std;

unsigned ggt(unsigned long, unsigned long);     (konventioneller Prototyp)
```

```
class fraction
{
public:
       long getNum() { return(num); }          (Zähler abfragen)
       unsigned long getDenom()                (Nenner abfragen)
         { return (denom); }
       void setNumDenom(long n, long d);       (Prototyp für Zähler, Nenner setzen)
       void print(void);                       (Prototyp für Ausgabe)

private:
        long num;                              (interne Variable
        unsigned long denom;                   für Zähler und Nenner)
        void reduce();
};

void fraction::print()                         (Definition der Interfacefunktion)
{ cout << num <<'/' << denom; }


void fraction::setNumDenom(long n, long d)     (Definition der Interfacefunktion)
{    if (d==0)                                 (Fehler abfangen!)
     { cerr << "Fehler: Nenner ist 0!" << endl; exit(1); }
     if (d<0)     { num=-n; denom=-d; }        (private Variable setzen)
     else         { num=n;  denom=d; }         (kürzen)
     reduce();
}


void fraction::reduce()                        (private Kürz-Funktion)
{    long div;
     if (num==0) return;                       (Zaehler=0: nichts zu tun)
     div=ggt(labs(num),denom);                 (ggT von |num| und denom
     num/=div; denom/=div;                     (und kürzen...)
}


unsigned ggt(unsigned long a, unsigned long b)
{    (Definition einer konventionellen Funktion; das geht immer! Algorithmus bitte einfach akzeptieren!)
  if (b==0) return(a); else return(ggt(b,a%b));
}


int main(void)
{
fraction x,*y;                                 (Erzeuge 2 Instanzen)
y=new fraction;

x.setNumDenom(12,24);          (Belege sie, dabei wird automatisch gekürzt)
y->setNumDenom(-1,3);          (Verwende öffentliche Methode, um auf eine Instanz zuzugreifen)
x.print(); y->print();         (Gib Inhalte der Instanzen aus)

                               (Anderer Weg zum Abfragen und Ausgeben der Instanzen:)
cout << "x ist: " << x.getNum() << '/' << x.getDenom() << endl;
cout << "y ist: " << y->getNum() << '/' << y->getDenom() << endl;

delete y;                      (nötig, da über new initialisiert)

}
```

Anmerkungen

1. Zugriff auf die *Public* - Komponenten der Instanzen erfolgt über Mitgliedsoperator `.` (bei *x*, das direkt erzeugt wurde) bzw. über den Zeigeroperator `->` bei *y*, das über Pointer und `new` erzeugt wurde.

2. Auf die Datenkomponenten `num, denom` kann **nur** über die Interfacefunktion `setNumDenom` geschrieben werden: `x.setNumDenom(12,24);`
   Aufgrund dieser Kapselung können sofort Fehler abgefangen (und auch gleich gekürzt) werden. Auch Lesen geht nur über Interfacefunktionen.

3. Aufrufe wie `x.print(), x.setNumDenom(12,24)` sind **Messages** explizit an eine bestimmte Instanz einer Klasse. Natürlich können nur öffentliche Methoden von außen angesprochen werden, `x.reduce()` z.B. ist vom Hauptprogramm aus nicht möglich.

10.4 Konstrukturen

I.d.R. ist es sinnvoll, wenn bei der Erzeugung einer Instanz **automatisch** den Objekten hierin definierte Werte zugewiesen (und andere Aktionen durchgeführt) werden. Im obigen Beispiel waren `num` und `denom` zunächst undefiniert und werden erst durch das Hauptprogramm belegt.

Besser geschieht dies über (öffentliche) **Konstruktor**-Funktionen:

Deklaration:
```
class fraction
{
public:
     fraction();                    (Defaultkonstruktor)
     fraction(long n, long d=1);    (weiterer Konstruktor)
....
```

Definition:
```
fraction::fraction()                (Name wie der Klassenname!)
     { num=0; denom=1;}
fraction::fraction(long n, long d=1)
     { ... igendwas anderes ....}
```

Instanz-Erzeugung mit Konstruktor-Aufrufen
```
int main(void)
{
     fraction x,y(2),z(2,3);
     ...
```

Anmerkungen

1. Konstruktor-Elementfunktionen heißen immer wie die Klasse selbst. Es darf kein Funktionstyp angegeben sein.
2. Parameterliste darf beliebig sein, auch leer.

3. Mehrere Konstrukturen mit unterschiedlicher Parameterliste (Signatur) sind erlaubt.
4. Bei der Instanzierung können Parameterlisten angegeben werden, die einen bestimmten Konstruktor aufrufen.
5. Bei Erzeugung von *x* wird der Default-Konstruktor aufgerufen, da die Liste leer ist; `num/denom` wird auf 0/1 gesetzt. Das gilt genauso, wenn eine Instanz über einen Pointer und `new` erzeugt wird.
6. Die Parameterliste bei der Erzeugung von `z` entspricht der des zweiten Konstruktors, daher wird dieser aufgerufen.
7. Da der zweite Konstruktor einen Defaultparameter enthält, wird er auch der Erzeugung von `y(2)` aufgerufen.

10.5 Überladen von Operatoren

Innerhalb von Klassen lassen sich sehr viele Operatoren wie z.B. `+ - / < =` **überladen**, d.h., die Wirkung der Operatoren ist abhängig von der Art der Objekte, auf die sie wirken. Zu beachten ist:
1. Man kann keine neuen Operatoren definieren.
2. Operandenzahl, Priorität und Assoziativität können nicht geändert werden.
3. Wenn z.B. `*` neu definiert wird, ist `*=` nicht automatisch definiert.

Zur Neudefinition der Operatoren werden diese auf Funktionen abgebildet.

Beispiel: `*` für die Klasse `fraction` -- Was muss die zugehörige Funktion enthalten?

1. Ein "aktuelles Bruchobjekt" ist ja innerhalb der Klasse bzw. einer Instanz der Klasse bereits definiert: `num / denom`
2. Erzeuge ein neues, temporäres Bruchobjekt für das Ergebnis
3. das zweite Bruchobjekt, mit dem multipliziert werden soll, muss dieser Funktion von außen übergeben werden
4. Berechne jetzt mit dem 'klassischen' Multiplizieroperator

```
        num (aktuell)          num (übergeben)
        -------------    *   ----------------
       denom (aktuell)        denom (übergeben)
```

Implementierung mit dem neuen Schlüsselwort `operator`, gefolgt von dem Operator, der überladen werden soll:

```
Rückgabetyp     Klasse                    übergebener Wert
fraction fraction::operator*(fraction y)
{
     fraction ergebnis;
     ergebnis=fraction(num*y.num,denom*y.denom);
     return ergebnis;
}
int main(void)
{
     fraction a(2,3),b(6,11),c;
     c=a*b;
     c.print();
}
```

Die Funktion ist in diesem Fall vom Typ `fraction` (sie liefert ja einen Bruch zurück!), übergeben wird ihr ein Bruchobjekt. `ergebnis` ist ein temporär erzeugtes Bruchobjekt, für deren Erzeugung auch der (zweite) Konstruktor aufgerufen wird. Der übergebene Parameter `y` stellt den rechten Operanden dar, das aktuelle Objekt den linken. Mit einem Pointer auf `y` geht das ganze im übrigen effizienter.

Statt `c=a*b` könnte man auch schreiben: `c=a.operator*(b)`, "auf `a` wird die Memberfunktion `operator*(b)` angewandt".

10.6 Was gibt's sonst noch alles?

Wir haben nur die allerersten Anfänge der objektorientierten C++ - Programmierung angesprochen. Im folgenden soll kurz aufgezählt werden, was es da sonst noch alles gibt:

- **Destruktoren** geben den Speicherplatz von Klasseninstanzen wieder frei.

- ***friends*** - Funktionen sind Funktionen außerhalb einer Klasse, denen man aber trotzdem den Zugriff auf private Komponenten eines Objektes erlaubt.

- Die **Überladung von Operatoren** ganz allgemein ist ganz schön kompliziert (es können immerhin 36 unterschiedliche Operatoren überladen werden, z.B. auch `()` oder `[]`. Auch Mehrfachüberladungen analog zu Funktionen gibt es (Stichwort unterschiedliche Signaturen).

- Es gibt 2 Arten von **abgeleiteten Klassen**. Die **Vererbung** hatten wir schon angesprochen. Eine neue Klasse erhält alle Eigenschaften und Methoden der Basisklasse. Daneben gibt es die **Komposition,** wo eine neue Klasse sich aus Objekten mehrerer vorhandener Klassen zusammensetzt.

- **Polymorphie** („Vielgestaltigkeit"), **virtuelle Funktionen**, **dynamische Bindung:** Betrachten wir nochmal unser Beispiel mit den einzelnen Fenstern einer graphischen Oberfläche. Bei einem *Refresh* - Befehl müssen alle Fenster neu gezeichnet werden. Das lässt sich einfach realisieren durch eine Schleife über alle Fenster der Basisklasse `winclass`. Einige Fenster sind aber z.B. Mitglieder der abgeleiteten Klasse `movwins`, andere Mitglieder einer Klasse `sonstwins`. Die Refresh-Methode ist sicher jeweils eine andere. Der Polymorphismus garantiert, dass für jedes Fenster die richtige Methode aufgerufen wird --- man bezeichnet sie als **virtuelle Funktionen**. Dies wird erst zur Laufzeit entschieden: sog. **dynamische** oder **späte Bindung.**

- **Templates** (Schablonen) erlauben es, Funktionen und Klassen ohne Typfestlegung zu definieren. Ein bestimmter Algorithmus wird nur einmal programmiert. Der Compiler setzt dann beim Aufruf die für die aktuellen Typen adapierte Version ein. Die ***Standard Template Library*** (STL) bietet eine Menge von Algorithmen an.