Programmieren in C/C++

Software-Schule Schweiz

Roland Aeschbacher

Frühjahr 2004 Version 5.2

Vorwort

Dieser C-Kurs wurde speziell für die Bedürfnisse der Software-Schule Schweiz aus verschiedenen Quellen zusammengestellt. Einerseits hielt ich mich streng an das ANSI Standard Buch von Brian Kernighan und Dennis Ritchie "The C Programming Language, second edition", andererseits übernahm ich Teile aus dem C-Kurs von A. Rüegg und Chr. Hedinger und aus dem Kurs "Programmieren in C" von E. Menet. An dieser Stelle möchte ich vorgenannten Herren herzlich danken. Ein besonderer Dank gilt auch K. Imhof, die den Text korrigierte und grösstenteils eintippte.

Diese Kursunterlagen wurden nach dreimaligen Anpassungen in den Jahren 1993, 1995 und 1998, 1999 nochmals völlig überarbeitet und in C++ Syntax umgeschrieben.

Es wird grundsätzlich die erweiterte C Schreibweise benutzt, also mit den ANSI C++ Erweiterungen auf **prozeduraler** Ebene.

Biel, Januar 2001 Roland Aeschbacher

1 Einführung 6
1.1 Systemimplementierungssprachen 6
1.2 Geschichte von C und C++ 7
1.3 Übersicht zu C und C++ 8
1.4 Bemerkungen für die Praxis 9

2 Der Aufbau eines C-Programms 10
2.1 Zerlegung einer C-Funktion 10
2.2 Zerlegung eines C-Programms 11
2.3 Schlüsselwörter (Keywords) 14

3 Daten 16
3.1 Grundlegende Datentypen 16
3.2 Konstante 18
3.2.1 Integer-Konstante 18
3.2.2 Zeichenkonstante 18
3.2.3 Gleitpunktkonstante 19
3.2.4 String-Konstante 19
3.3 Deklaration und Definition 19
3.4 Speicherklassen 21
3.5 Zusammengesetzte Datentypen 23
3.5.1 Array 23
3.5.2 Mehrdimensionaler Array 25
3.5.3 Struktur 26
3.5.4 Array von Strukturen 28
3.5.5 Union 29
3.5.6 enum 30
3.6 typedef 31
3.7 Referenztyp bei Datendefinition 31

4 Ausdrücke und Operatoren 32
4.1 Einfache Ausdrücke 32
4.2 Operatoren 32
4.2.1 Arithmetische Operatoren 32
4.2.2 Vergleichs- und logische Operatoren 34
4.2.3 Inkrement- und Dekrementoperatoren 36
4.3 Typensichere Ein-/Ausgabe 36
4.4 Listen Ausdrücke 37
4.5 sizeof Operator 37
4.6 Präzedenz von Operatoren 38
4.7 Konversionen 40
4.8 Bitweise Operatoren 45
4.9 Bit-Felder 47
4.10 Arithmetische und logische Zuweisungsoperatoren 48
4.11 Bedingte Ausdrücke 49
4.12 Überladene Operatoren 50
4.13 Scope Operator :: 50

5 Kontrollfluss 51
5.1 Einführung 51
5.2 Blockanweisung 52
5.3 if Anweisung 53
5.4 do while Anweisung 54
5.5 while Anweisung 54
5.6 for Anweisung 55
5.7 Zusammenfassendes Beispiel 56
5.8 break Anweisung 57
5.9 switch Anweisung 57
5.10 goto Anweisung 58
5.11 Die leere Anweisung 59

6 Funktionen 61
6.1 Einführung 61
6.2 Funktionsdefinitionen 61
6.3 return Anweisung 63
6.4 Argumente und Parameter 64
6.4.1 Wertübergabe 64
6.4.2 Referenzübergabe 64
6.4.3 Übergabe eines Arrays 65
6.5 Beispiel zu Funktionen und Parameter 66
6.6 Rekursion 68
6.7 Default Parameter 68
6.8 Inline Funktionen 69
6.9 Überladen von Funktionen 70

7 Pointer 71
7.1 Einführung 71
7.2 Definition von Pointern 71
7.3 Operationen mit Pointern 74
7.4 Pointer und Arrays 76
7.5 Pointer auf Strukturen 78
7.6 Anlegen von dynamischen Objekten 81
7.7 Argumente beim Programmaufruf 82
7.8 Pointer auf Funktionen 85
7.9 Typenqualifikation (type Qualifier) 86

8 Präprozessor 87
8.1 Einführung 87
8.2 #define, #undef 87
8.3 #include 88
8.4 #if, #ifdef, #ifndef, #else, #elif, #endif 88
8.5 #error 89

9 Erstellen modularer C/C++ Programme 90
9.1 Einleitung 90
9.2 Vorteile einer Aufteilung in Module 90
9.3 Aufteilung von Programmen in Headerfile und Implementationsfile 91
9.3.1 Das Hauptprogramm 92
9.3.2 Die Module 92
9.4 Programmentwicklung mit Modulen 94
9.4.1 Compilation 94
9.4.2 Linker 94
9.4.3 Make Utility 94

10 Ein-/Ausgabe in C++: Streams 95
10.1 Streams als Abstraktion der Ein-/Ausgabe 95
10.2 Ausgabe 95
10.3 Eingabe 96
10.4 Formatierte Ein-/Ausgabe 97
10.5 Streams und Dateien 98
10.5.1 Öffnen von Dateien 99
10.5.2 Lesen und Setzen von Positionen 100

11 C-Bibliotheken 102
11.1 Übersicht der ANSI-C Standard Libraries 102
11.2 Ein- und Ausgabe: <stdio.h> 103
11.2.1 Formatierte Ausgabe 104
11.2.2 Formatierte Eingabe 105
11.2.3 Ein- und Ausgabe von Zeichen 106
11.2.4 Fileoperationen 107
11.2.5 Direkte Ein- und Ausgabe 110
11.2.6 Positionieren in Files 110
11.2.7 Fehlerbehandlung 112
11.3 Tests für Zeichenklassen: <ctype.h> 113
11.4 Funktionen für Zeichenketten: <string.h> 114
11.5 Mathematische Funktionen: <math.h> 115
11.6 Hilfsfunktionen: <stdlib.h> 116
11.7 Fehlersuche: <assert.h> 119
11.8 Funktionen für Datum und Uhrzeit: <time.h> 119
11.9 Grenzwerte einer Implementierung: <limits.h> 122
11.10 Variable Argumentliste: <stdarg.h> 122

Anhang A : Traditionen in C 124

Anhang B : Häufig gemachte Fehler in C 126

Anhang C : Regeln zur Erstellung von portablen Codes 127

Anhang D : Methode zur Zerlegung der Deklarations- und Definitionssyntax 128

Übungen 131

Stichwortverzeichnis 140

1 Einführung

1.1 Systemimplementierungssprachen

Im Bereich der Programmierung von Computern unterscheidet man seit jeher zwischen den sogenannten Systemprogrammen und Anwendungsprogrammen. Wo die genaue Grenze zwischen beiden liegt, lässt sich nicht exakt definieren; anhand von Beispielen lassen sich aber die beiden Gruppen charakterisieren. Zu den typischen Systemprogrammen gehören etwa:

- Betriebssysteme
- Compiler, Interpreter
- Editoren
- Treiberprogramme für Peripheriegeräte

Zu den Anwendungsprogrammen gehören sicherlich

- Lohnabrechnung
- Finanzbuchhaltung
- Maschinensteuerungen
- Statistikprogramme etc.

Entsprechend dieser Klassifizierung bestehen in bezug auf die Programmierung gewisse Unterschiede; die Anforderungen an Systemprogramme sind in etwa:

- Effizienz
- Hardwarenähe
- Zuverlässigkeit

während bei den Anwenderprogrammen folgende Gesichtspunkte mehr im Vordergrund stehen:

- gute Anpassung an das Problem
- leichte Erlernbarkeit
- Wartbarkeit

Wie man sofort sieht, überdecken sich die Anforderungen in mehreren Punkten. Dennoch haben sich für die genannten Bereiche spezielle Sprachen etabliert, im Anwendungsbereich etwa:

COBOL	für den Bereich der kommerziellen Anwendung
Fortran	für technisch - wissenschaftliche Problemstellungen
Pascal+Modula	Lehr- und 'general-purpose' -Sprache
Java	als Plattform unabhängige Sprache

Auf der anderen Seite stehen

BCPL, C/C++, Portal, Chill, Ada etc...

Vielfach wird gerade für Aufgaben der Systemprogrammierung die jeweilige Maschinensprache verwendet; obwohl die reine "Lehre" die Verwendung von Assemblern mit Bann belegt hat, haben sichin der Praxis die höheren Sprachen noch nicht überall durchgesetzt. Wenn es auch einige Anwendungen geben mag, die extrem zeitkritisch sind, so ist in den meisten dieser Fälle der Einsatz höherer Sprachen aus Gründen der Zuverlässigkeit und Wartbarkeit angebracht; die Verwendung strukturierter Sprachen ist bei der Entwicklung grosser Systeme ein wesentliches Hilfsmittel zur Bewältigung der exponentiell wachsenden Komplexität solcher Programme.

1.2 Geschichte von C und C++

Die Ursprünge der Sprache C liegen, wie auch bei Ada, interessanterweise in Europa. Ausgehend von der Sprache BCPL, die von Richards im Jahre 1968 für Systemaufgaben entwickelt wurde, entstand in den Forschungslaboratorien von Bell Telephone in USA erst eine Sprache B, aus der dann schliesslich C resultierte.

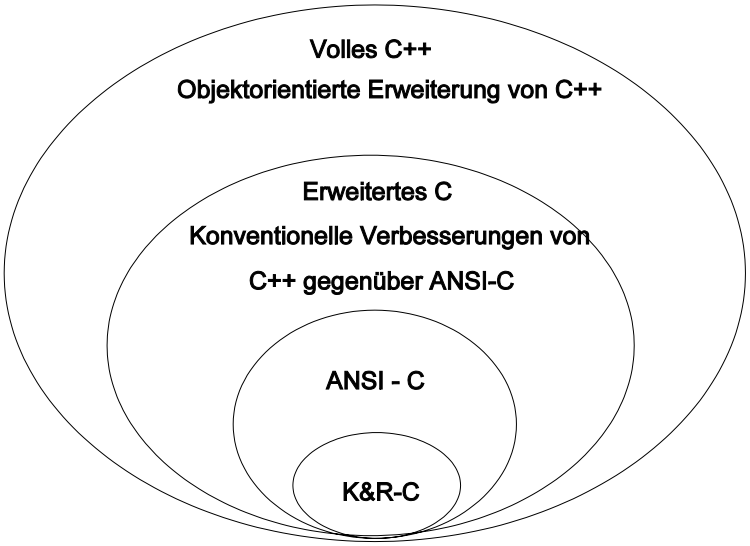
Der Kontext, in dem C entwickelt wurde, ist sehr aufschlussreich. In demselben Labor war damals, also Anfang der siebziger Jahre, ein Betriebssystem entstanden, das im wesentlichen für die Bedürfnisse der dortigen Entwickler gemacht war. Dessen erste Fassung war noch in Assembler geschrieben; für die Neu Implementierung entschied man sich, eine höhere Sprache zu verwenden - eben C. Diese Entscheidung hat wohl wesentlich zu der weiten Verbreitung beigetragen, die dieses Betriebssystem heute erlebt. Sein Name ist UNIX, und es sieht derzeit so aus, als würde UNIX das wichtigste Standardbetriebssystem neben der Mikrosoft Welt bleiben. Die Verwendung von C als Implementierungssprache hat zur Folge, dass das System relativ leicht zu portieren ist.

Die Väter von C sind Kernighan und Ritchie, die auch das Standardwerk über C verfasst haben: "The C Programming Language" bei Prentice Hall. Dieses Buch hat der Sprache zum weltweiten Durchbruch verholfen. Die Zeit geht aber auch an einer Programmiersprache nicht spurlos vorbei. Man gewinnt neue Erkenntnisse über guten Sprachentwurf, die Praxis zeigt Schwächen der Sprache auf etc. Dies hat zu zwei wichtigen Weiterentwicklungen geführt: Zum einen hat sich **ANSI** (**A**merican **N**ational **S**tandards **I**nstitute) um eine rigorose Definition der im Original doch manchmal zu unpräzis beschriebenen Sprache bemüht, und zum zweiten hat Bjarne Stroustrup (ebenfalls Bell Labs) die Sprache um objektorientierte Mechanismen und Konstruktionen erweitert.
Als Resultat der zuerst erwähnten Anstrengungen liegt heute ein Draft-Standard für ANSI-C vor, und die Weiterentwicklung durch Bjarne Stroustrup (Bell Labs, 1983) ist unter dem Name C++ kommerziell verfügbar. C++ basiert auf dem ANSI-C Standard. Es stellt also eine Erweiterung von C/C++ dar und ist seit Juli 1998 auch standardisiert.

Zusammen mit dem C/C++ Compiler werden jeweils auch immer der **Präprozessor** sowie die **Library-Funktionen** geliefert und dokumentiert. Diese sind **nicht** Bestandteil der Sprache C/C++. ANSI hat aber gut daran getan, auch diese compilerexternen Komponenten zu standardisieren.

1.3 Übersicht zu C und C++

Dieser Kurs ist grundsätzlich in ANSI-C abgefasst. Wo es aber sinnvoll erscheint werden auch C++ Elemente (erweitertes C) verwendet. Diese C++ Elemente sind *kursiv* gedruckt und benötigen somit einen C++ Compiler. Die Objektorientierte Programmierung ist aber **nicht** Inhalt dieses Kurses.



Beispiel:

```
#include <iostream>

int main()
{
      std::cout << "Hello World\n";
      return 0;

}
```

1.4 Bemerkungen für die Praxis

C/C++hat, wie jede Programmiersprache, gewisse Mängel und ist deshalb, auch wie jede andere Programmiersprache, unter Dauerbeschuss durch Puristen, Theoretiker und Fanatiker. Bis zu einem gewissen Grade muss man diesen auch recht geben, und man muss vor allem etwas gegen diese Mängel (nicht die Puristen etc.!) tun. Nachfolgend sind einige der 'glitschigen' Stellen erwähnt, zusammen mit rudimentären Massnahmen. Auch im weiteren Verlauf dieser Einführung in das Programmieren in C/C++ wird der Leser und die Leserin weitere Hinweise finden, welche über den reinen Sprachgebrauch hinausgehen und die Absicht haben, Praktiker zum guten Gebrauch von C/C++ anzuleiten.

1. C/C++ ist nicht restriktiv. Im Gegensatz zu Sprachen wie Pascal, Modula-2 etc. setzt C/C++ kein sehr restriktives Type-Checking durch. Durch konsequentes Deklarieren und Definieren kann jedoch der Programmierer dem Compiler das Type-Checking ermöglichen. Der Lohn dafür besteht in eindeutigen Fehlermeldungen und Warnungen des Compilers anstelle von unverständlichen Linkermeldungen und z.T. kaum feststellbaren Unstimmigkeiten und Fehlern beim Ausführen des Programms.

2. C/C++ lebt von Nebeneffekten. Grundsätzlich ist die Ausnützung sprachlicher Schwächen zu vermeiden (Portabilität!). Werden Nebeneffekte dennoch ausgenutzt, dann sind diese unbedingt klar zu dokumentieren (Kommentare).

3. C/C++ hat Goto's. Hier ist's wie mit anderen Programmiersprachen, Waffen, Autos und anderen gefährlichen Spielzeugen : nicht das Ding an sich ist gefährlich, sondern möglicherweise derjenige, der damit umgeht und wie er damit umgeht.

4. C/C++ hat Pointer. Die Puristen behaupten, die Pointer seien für die Daten etwa dasselbe Gift wie die Goto's für die Steuerstrukturen. Wir verweisen auf 3. und empfehlen einen gezielten und konsistenten Umgang mit diesem unerlässlichen Werkzeug.

5. C/C++ ist kurz und bündig (1). Böse Zungen behaupten, C/C++ sei write-only. In der Tat lassen sich C-Programme in sehr kompakter Weise schreiben, was natürlich die Lesbarkeit in Mitleidenschaft ziehen kann. Durch die üblichen Massnahmen (sinnvolle Variablennamen, eingerückte Darstellung, klare Kommentare etc.) können aber auch Resultate erzielt werden, die sich neben Modula-2 oder Pascal-Programmen durchaus sehen lassen.

6. C/C++ ist kurz und bündig (2). In C/C++ werden viele Symbole nicht mit Schlüsselworten dargestellt, sondern mit einzelnen Sonderzeichen. Bestes Beispiel ist wohl die Blockklammer {...} anstelle von BEGIN...END. An diese Schreibweise muss man sich etwas gewöhnen, aber die Routine wird diese anfängliche Schwierigkeit rasch überwinden.

7. Brian Kernighan und Dennis Ritchie schreiben zu Beginn ihres Buches
'The only way to learn a new programming language is by writing programs in it.'
Dem ist nichts weiter hinzuzufügen.

2 Der Aufbau eines C-Programms

2.1 Zerlegung einer C-Funktion

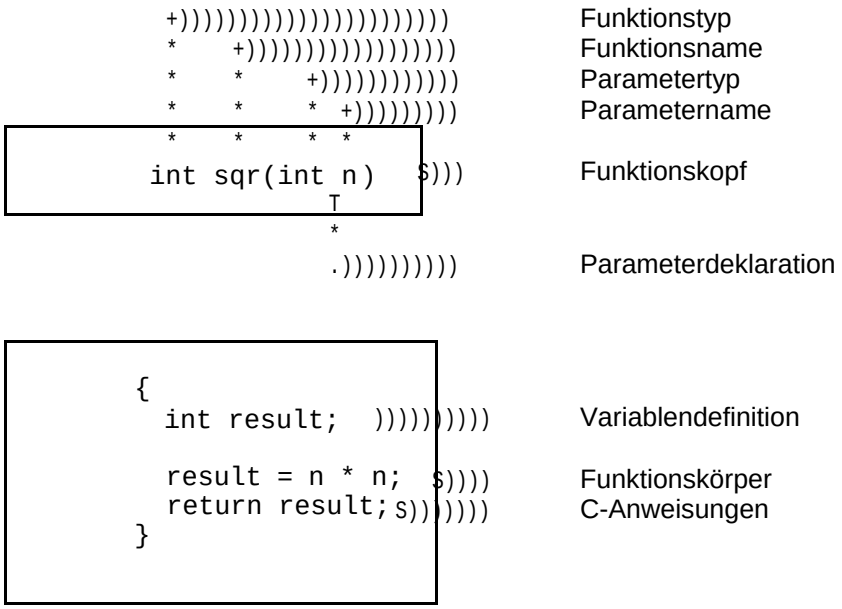
Der wichtigste Baustein eines C-Programms ist die Funktion.
Beispiel :

```
int sqr(int n)
{
      int result;

      result = n * n;
      return result;
}
```

`sqr` ist eine ganz einfache Funktion, welche aber nicht Bestandteil von C/C++ ist. Sie berechnet das Quadrat der ganzen Zahl `n` und gibt dieses als Funktionswert an den Aufrufer zurück.

In ihre Einzelteile zerlegt, präsentiert sich `sqr` wie folgt :



Der Funktionskopf besteht also aus dem Funktionstyp, d.h. dem Datentyp des gelieferten Resultats (in unserem Beispiel wird das Quadrat einer ganzen Zahl geliefert, dies ist wieder eine ganze Zahl, repräsentiert durch die Variable `result` und damit `int`), dem Funktionsnamen, welcher, wie wir später sehen werden, den Regeln der Namensgebung in C/C++ für Variablen, Funktionsnamen etc. (allg. Identifier) unterliegt, sowie der Parameterliste eingefasst von Klammern `(..)`. Die Parameterliste kann leer sein, dann steht zwischen den Klammern (diese sind obligatorisch!) die Bezeichnung für den leeren Datentyp `(void)`, oder sie kann einen oder mehrere Parameter umfassen. Sind Parameter vorhanden, so müssen diese mit ihrem Datentyp und Namen aufgeführt sein.

Der Funktionskörper ist von geschweiften Klammern `{..}` umgeben (syntaktisch ist alles, was mit `{..}` eingefasst ist, ein Block). Zu Beginn des Blockes stehen die Definitionen und Deklarationen von Variablen, danach folgen die Anweisungen.

Damit sind die allerwichtigsten konstituierenden Elemente einer Funktion aufgezählt. Wir werden später noch sehen, für welche Teile sog. Defaults bestehen, wie die Funktionsköpfe in Prä-ANSI-Zeiten ausgesehen haben, wie man bei einer nicht spezifizierten Anzahl von Parametern verfahren muss etc. Im wesentlichen setzt sich eine C-Funktion aus einer Anzahl von Anweisungen zusammen. Ein C-Programm seinerseits besteht aus einer Reihe von C-Funktionen.

2.2 Zerlegung eines C-Programms

```
 1    #include <iostream>
 2    using namespace std;
 3    #define EINS 1
 4    short int fak(short int);
 5    /*************************************************************************
 6    * Funktionsname       : main
 7    * Beschreibung        : fragt nach einer Zahl, liest sie ein
 8    *                       und druckt die Fakultaet davon aus
 9    * Parameter           : keiner
10    * Rueckgabewert       : 0
11    *************************************************************************/
12    int main( )
13    {
14        short int zahl;      // Definition lokaler Variablen
15        short int resultat;
16
17        cout << endl << "Zahl eingeben (Prg-Abbruch : Zahl < 0) : ";
18        cin >> zahl;
19        while(zahl >= 0)
20        {
21           resultat = fak(zahl);
22           cout << zahl << "! = " << resultat << endl;
23           cout << endl << "Zahl eingeben(Prg-Abbruch : Zahl < 0) : ";
24           cin >> zahl;
25        }
26        return 0;
27    }
28
29    /*************************************************************************
30    * Funktionsname       : fak
31    * Beschreibung        : berechnet die Fakultaet (rekursiv)
32    * Parameter           : die zu berechnende Zahl
33    * Rueckgabewert       : die Fakultaet
34    *************************************************************************/
35    short int fak(short int n)
36    {
37        short int resu;
38        if(n == 0)
39           resu = EINS;
40        else
41           resu = n * fak(n - EINS);
42        return resu;
43    }
```

Programm Fakultäten

Ohne den Stoff des Kurses vorwegzunehmen, sollen einige Stilmerkmale der Sprache C/C++ erklärt werden. Zur Erinnerung: alle *kursiv* gedruckten Programmzeilen benötigen einen C++ Compiler.

Wie bereits angedeutet, sind C-Programme Sammlungen von Funktionen (die Reihenfolge spielt dabei keine Rolle).

Das Programm Fakultäten besteht aus den zwei Funktionen main und fak. Bekanntlich ist es die Aufgabe einer Funktion, einen Wert zu berechnen und diesen seinem Aufrufer zu liefern (siehe Funktion fak Zeile 35).

Zu Beginn des Programmtextes steht auf Zeile 1

```
1     #include <iostream>
```

Dies ist eine Anweisung an den Präprozessor, an dieser Stelle den Inhalt des Textfiles iostream einzufügen. Es enthält Deklarationen aller I/O Funktionen.

Mit

```
2     using namespace std;
```

wird der Zugriff auf die I/O Funktionen erleichtert. Man könnte auch
`std::cout << std::endl << "Zahl...";` schreiben (anstelle Zeile 17)

```
3     #define a b
```

ist eine Anweisung an den Präprozessor, im Rest des Files jeweils das Token a durch den String b zu ersetzen. In den Zeilen 39 und 41 geschieht demnach eine derartige Substitution. Obige `#define` Anweisung kann später im Programmtext durch `#undef` a widerrufen, d.h. ausgeschaltet werden.

```
4     short int fak(short int);
```

zeigt eine Funktionsdeklaration (engl. function prototype).

Der nachfolgende Text zwischen den Zeichen

```
5/11  /* ... */
```

ist offensichtlich ein Kommentar. Das Zeilenkommentarsymbol `//` ermöglicht Kommentare bis ans Zeilenende einzufügen (siehe Zeile 14).

```
12    main
```

ist eine Funktion mit der besonderen Bedeutung, dass jedes Programm genau eine Funktion mit diesem Namen haben muss. main ist denn auch der Anfangspunkt für die Ausführung eines Programms.

Die geschwungenen Klammern

```
13/27 { ... }
```

umschliessen den Funktionstext (Block). Sie entsprechen in ihrer Funktion der Pascal'schen BEGIN ... END-Klammer. Auf Zeile 14 wird die `short int` Variable `zahl` definiert, denn C/C++ verlangt die Definition oder Deklaration aller verwendeten Objekte (Variablen, Funktionen etc.) **vor** deren Gebrauch. Eine Definition oder Deklaration wird wie die Anweisungen mit ; abgeschlossen.

```
17/23 cout  << "TEXT" << zahl << endl ;
```

ist ein Funktionsaufruf zum Ausdrucken von Texten und Variablen. In den Zeilen 17 und 24 wird ein Text ausgedruckt. Er steht zwischen den Anführungszeichen ". Dagegen werden in der Zeile 22 die Variablen `zahl` und `resultat` und dazwischen der Text `"! = "` ausgedruckt. Mit `<< endl` wird ein "newline" erzeugt.

```
18/24 cin >> zahl;
```

ist ein Funktionsaufruf zum Einlesen von Variablen. Hier wird eine Dezimalzahl eingelesen.

Anmerkung: man sollte ein zweimaliges Einlesen der gleichen Variablen möglichst vermeiden!

```
19      while(zahl >= 0)
```

ist eine Schleifenanweisung. Die Schleife wird wiederholt, solange `zahl >= 0` wahr ist, also bis `zahl` einen negativen Wert hat. Der Schleifenkörper besteht aus mehreren Einzelanweisungen, deshalb werden diese mit `{...}` zu einer Blockanweisung zusammengefasst.
Mit **Argument** wird ein Ausdruck im Funktionsaufruf bezeichnet, mit **Parameter** die entsprechende Variable im Funktionskopf. Alternativ werden oft auch die Bezeichnungen **aktueller** bzw. **formaler Parameter** verwendet.

Zeile 35 besagt, dass die Funktion `fak` als Resultat einen Wert vom Typ `short int` liefert und ein einziges Argument, ebenfalls vom Typ `short int`, entgegennimmt. Die Deklaration der Funktion `fak` auf Zeile 3 ist zwingend, weil die Funktion selber textlich **nach** ihrer Verwendung (d.h. dem Aufruf) folgt. Würde allerdings die Funktion `fak` dem Text von `main` vorangehen, dann wäre die Deklaration nicht obligatorisch.

```
38      if(n == 0)
```

testet die Variable `n` auf Gleichheit mit 0. Man beachte den Unterschied zwischen dem Zuweisungsoperator `=` und dem logischen Operator für Gleichheit `==`. Dies ist darum besonders wichtig, weil `if(n = 0)` **syntaktisch** ebenfalls korrekt wäre, aber natürlich ein anderes (hier falsches) Resultat erzielen würde. In C gibt es, im Gegensatz zu C++, keinen eigentlichen Boole'schen Datentyp, sondern der Wert 0 entspricht `false`, alle anderen Werte sind `true`. Eine wahre Bedingung liefert den Wert 1. In Tabellenform kann dieser Sachverhalt wie folgt dargestellt werden :

Ausdruck	liefert	Wert
wahr	->	1
falsch	->	0

umgekehrt betrachtet:

Wert	liefert	Ausdruck
<>0	->	wahr
0	->	falsch

Also hat z nach der Ausführung von

```
...
int x = 5, y = 6, z;

z = (x == y);
...
```

den Wert `0`, weil `(x == y)` falsch `(= 0)` ist, was z zugewiesen wird. Demgegenüber hat z nach

```
z = (x == (y - 1));
```

den Wert `1`, weil x gleich `y - 1` ist.

Würde nun im Programm Fakultäten `if(n = 0)` stehen, so würde `n` der Wert `0` zugewiesen. Die Bedingung wäre damit invariant falsch, was aber semantisch nicht mehr der ursprünglichen Absicht entspricht. Demgegenüber wäre die Bedingung `if(!n)` sowohl syntaktisch als auch semantisch korrekt, da `!` den logischen Operator 'not' darstellt.

Die Zeile 40 enthält einen **rekursiven** Aufruf von `fak`.

Mit `return resu` in Zeile 42 wird der Funktionswert zurückgegeben.

N.B. Die **Syntax** einer Sprache befasst sich mit grammatikalischen Aspekten, also mit dem Format, dem Aufbau von Sätzen etc. Die **Semantik** hingegen befasst sich mit der Aussage, der Bedeutung von Sätzen und sprachlichen Konstruktionen. Es ist unter anderem die Aufgabe des Compilers, syntaktische Fehler in einem Programm zu erkennen, hingegen liegt die Kontrolle der Semantik vollständig in der

Verantwortung des Programmierers. Ein syntaktisch korrektes, aber semantisch falsches Programm pflegt seinem Autor oft den Spruch zu entlocken : "das Programm läuft zwar, aber die Resultate sind falsch !".

2.3 Schlüsselwörter (Keywords)

Die folgenden Namen sind in C/C++ reservierte Schlüsselwörter und dürfen nicht anderweitig verwendet werden.

Kategorie	**Schlüsselwörter**
Basisdatentypen	*bool*,char,*wchar_t*,int,float,double,void
Ergänzungen zu Basisdatentypen	short,long,signed,unsigned,
Datentypen	enum,*class*,struct,union,typedef
Speicherklassen	auto,extern,register,static,*mutable*
Qualifizierer	const,volatile
Kontrollstrukturen	if,while,else,case,switch,default,do,for
Sprunganweisungen	break,continue,return,goto
Typumwandlungen	*const_cast,dynamic_cast,reinterpret_cast*, *static_cast*
Typsystem	*typeid,typename*
Namensräume	*namespace,using*
Operatoren	sizeof,*new,delete,operator,this*
Zugriffsschutz	*friend,private,public,protected*
Funktionsattribute	*inline,virtual,explicit*
Generizität	*template*
Ausnahmebehandlung	*catch,throw,try*
Assemblercode	*asm*

Schlüsselwörter (Keywords) in ANSI-C und *ANSI-C++*

Manche Compiler kennen zum Teil weitere Schlüsselwörter wie Fortran; ANSI-C/C++ erkennt diese als übliche Erweiterungen. Man beachte, dass durchwegs Kleinschreibung zur Anwendung gelangt.

********** **Aufgabe** **********

1) Identifizieren Sie in nachstehender Funktion die in diesem Kapitel auf Seite 9 vorgestellten Funktionsbausteine (Kopf, Block, Funktion, Parameter, Rückgabewert) und versuchen Sie den Ablauf (Reihenfolge) dieses Programs zu verstehen :

```
/*****************************************************************************************************
 * HEADER FILES
 *****************************************************************************************************/
#include <iostream>
using namespace std;

/*****************************************************************************************************
 * DEFINITIONEN UND DEKLARATIONEN
 *****************************************************************************************************/
int suche(char);

/*****************************************************************************************************
 * FUNKTIONSNAME          : main
 * BESCHREIBUNG           : fragt nach dem Buchstaben, liest ihn ein
 *                          und druckt die Position aus
 * PARAMETER              : keiner
 * RÜCKGABEWERT           : keiner
 *****************************************************************************************************/
int main( )
{
    char buchstabe;
    int position;

    cout << endl << "Buchstabe eingeben (A...Z) : ";
    cin.get(buchstabe);
    position  = suche(buchstabe);
    cout << endl;
    cout << "Buchstabe " << buchstabe << " ist an der " << position;
    cout << ". Position" << endl;
    return 0;
}

/*****************************************************************************************************
 * FUNKTIONSNAME          : suche
 * BESCHREIBUNG           : sucht Position eines Buchstaben im Alphabet
 * PARAMETER              : der Buchstabe
 * RÜCKGABEWERT           : die Position des Buchstaben
 *****************************************************************************************************/
int suche(char zeichen)
{
    int zaehler;
    char alpha;

    zaehler = 1;
    alpha = 'A';
    while(zeichen != alpha)
    {
      zaehler = zaehler + 1;
      alpha = alpha + 1;
    }
    return zaehler;
}
```

3 Daten

3.1 Grundlegende Datentypen

Die zwei wesentlichen Konzepte in allen Programmiersprachen sind das Datenkonzept und die Kontrollstruktur. Daten sind interne Repräsentationen der Aussenwelt, die in bestimmter Weise verarbeitet werden. Welche Daten in welcher Weise dargestellt werden, ist sehr unterschiedlich. Wichtig ist, dass man unterscheidet zwischen der internen Repräsentation und dem sog. Datentyp, der eine Interpretation der Repräsentation darstellt.

Beispiel:

ein 16-Bit Wort enthält die Bitfolge

1111'1111'1111'1111 oder hexadezimal FFFF

Wird dieses Wort als vorzeichenbehaftete Integer-Zahl interpretiert, dann stellt sie bei 2erKomplement-Darstellung den Wert -1 dar. Ohne Vorzeichen ist ihr Wert aber 65535.

Man muss zwischen den Begriffen Daten**definition** und Daten**deklaration** unterscheiden. Eine Deklaration gibt nur die Eigenschaften (z.B. Datentyp) einer Variablen an, während eine Definition sowohl eine Deklaration ist als auch den benötigten Speicherplatz für die Variable bereitstellt. Jede Variable hat einen gewissen **Geltungsbereich** (Scope), d.h. sie wird von gewissen Programmabschnitten gesehen (is visible) und von anderen nicht (is invisible). Jede Variable hat auch eine bestimmte **Lebensdauer** (Lifetime), sie hat einen **Namen** (Identifier) und einen **Wert** (Value). Jedes **definierte** Datenobjekt benötigt eine ganz bestimmte Anzahl Speichereinheiten. Die Grösse der Speichereinheit ist maschinenabhängig (in der Regel 1 Byte zu 8 Bits). Der sizeof Operator liefert die Anzahl Speichereinheiten, die ein Objekt belegt.

C/C++ kennt nur sechs Grundtypen:

`bool`	*boolescher Wert, der wahr oder falsch ist (`true` und `false` sind vordefiniert)*
`char`	einzelnes Byte zur Speicherung eines Zeichens
`int`	Integer, Länge entspricht im allgemeinen einem Maschinenwort bzw. einem Register
`float`	Gleitpunktzahl einfacher Genauigkeit
`double`	Gleitpunktzahl doppelter Genauigkeit
`void`	Pseudotyp, der auf einen beliebigen Typ oder eine Funktion ohne Rückgabewert zeigt

Der Grundtyp `int` und `char` sind vorzeichenbehaftet. Folgende Qualifizierer sind als Ergänzung möglich:

`short`	zum Datentyp `int`	z.B.	`short int`
`long`	zu den Datentypen `int` und `double`	z.B.	`long int` oder `long double`
`unsigned`	zu den Datentypen `int` und `char`	z.B.	`unsigned int` oder `unsigned char`
`signed`	zu den Datentypen `int` und `char` (default)		

Welche Grösse eine `short int` bzw. `long` besitzt, ist vom System und vom Compiler abhängig. Als Regeln gelten nur :

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

Ganzzahlige Datentypen in einer 32-Bit Umgebung

Datentyp	Grösse	Bezeichnung	möglicher Wertebereich
bool	8	boolescher Wert	*true* oder *false*
char	8	Zeichen	-128 ... 127
signed char	8	Zeichen	-128 ... 127
unsigned char	8	Zeichen	0 ... 255
short	16	Integerwert	-32'768 ... 32'767
signed short	16	Integerwert	-32'768 ... 32'767
unsigned short	16	Integerwert	0 ... 65'535
int	32	Integerwert	-2'147'483'648 ... 2'147'483'647
signed int	32	Integerwert	-2'147'483'648 ... 2'147'483'647
unsigned int	32	Integerwert	0 ... 4294967295
long int	32	Integerwert	-2'147'483'648 ... 2'147'483'647
signed long int	32	Integerwert	-2'147'483'648 ... 2'147'483'647
unsigned long int	32	Integerwert	0 ... 4'294'967'295

Gleitkomma Datentypen in einer 32-Bit Umgebung:

Datentyp	Grösse (Bit)	Vorzeichen (Bit)	Exponent (Bit)	Mantisse (Bit)	Wertebereich	Präzision (Stellen)
float	32	1	8	23	+/-3,4 * 10E 38	7
double	64	1	11	52	+/-1,7 * 10E 308	15
long double	80	1	15	64	+/-1,2 * 10E 4932	19

3.2 Konstante

3.2.1 Integer-Konstante

Integer-Konstanten können in drei Formen dargestellt werden:

dezimal: Folge von Dezimalzeichen, die **nicht mit 0** beginnt

oktal: Folge von Oktalzeichen {0..7}, beginnend **mit einer 0**
z.B.: 012 -> 001 010 (binär) -> 10 (dezimal)

hexadezimal: Folge von Zeichenmenge {0..9, a..f, A..F}, beginnend **mit 0x** oder **0X**
z.B.: 0x89ab, 0X9aBc

Ein nachgestelltes **l** oder **L** signalisiert eine `long` Konstante, ein nachgestelltes **u** oder **U** bezeichnet eine unsigned. Man beachte, dass C/C++ kein positives Vorzeichen kennt. Der ANSI-Standard hat aber neu den unären Operator + eingeführt, was in der Schreibweise dem positiven Vorzeichen gleichkommt.

3.2.2 Zeichenkonstante

Einzelne Zeichenkonstanten werden in einfache Hochkommas eingeschlossen, wie z.B.:

```
'a'   'A'   '0'   '*'
```

Bestimmte Sonderzeichen können mit Hilfe des "Fluchtsymbols" '****' (Backslash) dargestellt werden:

Zeichen	Kennung	Kodierung
alert	BELL	`\a`
newline	LF	`\n`
horizontal tab	HT	`\t`
vertical tab	VT	`\v`
backspace	BS	`\b`
question mark	?	`\?`
form feed	FF	`\f`
carriage return	CR	`\r`
backslash	\	`\\`
single quote	‘	`\’`
double quote	“	`\”`
oktal number	000	`\000`
hexadezimal number	xhh	`\x00`

\ooo wird benutzt, um Byte-Werte darzustellen (ASCII), ooo ist dabei eine oktale Zahl. So sind **'0' und '\x30'** das Gleiche. **'\0'** stellt das ASCII-Zeichen **NUL** dar, und es hat den **numerischen Wert 0**.

3.2.3 Gleitpunktkonstante

Gleitpunktkonstanten haben, wie aus anderen Sprachen auch bekannt, die folgende Form:

sign integer_part . fraction_part e/E sign(opt) exponent

Vom integer_part bzw. fraction_part kann einer weggelassen werden. Der Exponententeil kann ganz wegfallen. Dezimalpunkt oder Exponententeil müssen vorhanden sein.

Beispiel:

```
1.2         1.2e+10     4.5E-5      3E7
1.          .2          .3E6        .9e-27
```

3.2.4 String-Konstante

Zeichenketten-Konstanten kommen sehr häufig vor. In C/C++ werden sie in doppelte Hochkomma's eingeschlossen.

```
"Das ist eine Zeichenkette"
```

Eine Zeichenkette repräsentiert einen Array von Zeichen, wobei das letzte Zeichen eine binäre 0 ist. Damit ist es sehr einfach, das Ende eines Strings zu finden. Eine Zeichenkette kann Sonderzeichen enthalten, die mit Hilfe des Fluchtsymbols '****' definiert sind.

3.3 Deklaration und Definition

An dieser Stelle sollen einfache Definitionen bzw. Deklarationen dargestellt werden. Die Feinheiten werden im weiteren Verlauf noch im Detail erklärt.

Die Definition bezeichnet die Vereinbarung, in der für eine Variable Speicherplatz reserviert wird. Die Deklaration bezeichnet die Vereinbarung, die zwar die Variable beschreibt, aber eben keinen Speicherplatz reserviert.

Jede Variable muss definiert oder deklariert werden, bevor sie benutzt wird. Die einfachste Definition bzw. Deklaration besteht aus einem Typ und dem Namen der Variablen:

type_name identifier

Namen (identifier) bestehen aus der Menge der Alpha-Zeichen (**a..z, A..Z**), den numerischen Zeichen (**0..9**) und dem Unterstreichungszeichen '_'. Erstes Zeichen muss ein Alpha- oder das '_' Zeichen sein. Mindestens die ersten 31 Zeichen eines internen Namens werden unterschieden. Bei externen Namen, wie den Namen von Funktionen und globalen Variablen, werden aber nur die ersten 6 Zeichen garantiert. Meist können aber auch hier längere Namen verwendet werden.

Gültige Namen sind:

```
i     name  GROSS GeMiScHt    k2    abc_txt     _small
```

Falsche Namen sind:

```
1b    ABC.TXT     -sonderzeichen
```

Die Definition für eine Integer-Variable mit Namen `i` lautet also

```
int i;
```

Sollen mehrere Variablen vom gleichen Typ definiert werden, so kann man das durch eine Liste von Namen erreichen:

```
int i, j, k;
```

Das entspricht den Definitionen

```
int i;
int j;
int k;
```

Eine Variable kann bei ihrer Definition auch initialisiert werden. Das bedeutet, dass ihr sofort ein Wert zugewiesen wird. Das wird erreicht, indem hinter den Namen ein = Zeichen und ein konstanter Wert geschrieben wird.

Beispiel:

```
double gewicht = 83.304;
char zeichen = 'e';
```

C/C++ kennen die Typen Qualifier `const` und `volatile` (siehe auch Kapitel 7.9). Sie können nur auf Ausdrücke wirken, welche lvalues ergeben (ein left value ist etwas, was links vom Zuweisungsoperator = stehen kann).

const verhindert das Verändern eines gekennzeichneten Identifier.

z.B. `const double pi = 3.141;`

der Wert der Konstanten `pi` darf nicht verändert werden, er muss also bei der Definition zugewiesen werden. Dies entspricht (mit Einschränkungen) `#define PI 3.141`

Wichtig ist die Angabe des Datentyps. Wird er weggelassen wie in `const pi = 3.14;` , nimmt der Compiler per Default `int` an und konvertiert 3.14 zu 3 !!! Es wird keine Fehlermeldung erzeugt.

Objekte können zusätzlich als `volatile` (flüchtig) definiert werden. `volatile` ist ein Hinweis an den Compiler, eine Optimierung an den Stellen zu vermeiden, an denen solche Objekte auftreten. Dies ist zum Beispiel erforderlich, wenn der Wert des Objekts nicht nur durch das Programm geändert, sondern auch durch andere laufende Programme oder durch eine programmunab-hängige Instanz (z.B. DMA) verändert wird. Ein `volatile` Objekt wird jedesmal direkt aus dem Hauptspeicher geladen (forced refetch).

z.B `volatile int port[0];`

3.4 Speicherklassen

Wir haben noch nichts gesagt über den Gültigkeitsbereich und die Lebensdauer von Variablen. Dazu müssen wir das Datenmodell der Sprache C/C++ etwas näher anschauen. Wir unterscheiden zwischen statischen (`static, extern`) und dynamischen (`auto, register`) Daten.

`auto` : Die auto Variablen werden **innerhalb** von Funktionen definiert; ihre Lebensdauer ist an die der Funktion gekoppelt. Beim Aufruf der Funktion wird für diese Variablen Speicherplatz reserviert. Beim Verlassen der Funktion, also nach einem `return`, wird dieser Speicherplatz wieder freigegeben. Bei einem erneuten Aufruf derselben Funktion kann man **nicht** (!!!) davon ausgehen, dass die Variablen noch die alten Werte besitzen.

`register` : Wenn eine `auto` Variable nicht im Speicher definiert werden soll, sondern in einem internen Register des Prozessors (CPU), so ist die Speicherklasse register zu verwenden. Es sind so viel Registervarianten möglich, wie der Prozessor zur Verfügung stellt; die eventuell Überzähligen werden dann als auto Variablen festgelegt.

`extern` : Die Lebensdauer der globalen Variablen reicht im Gegensatz zu den dynamischen Variablen von deren Definition bis zum Verlassen des Programms. Eine globale Variable kann ausserhalb oder innerhalb von Funktionen deklariert werden (mit der Vor Anstellung von `extern`). Hingegen darf eine globale Variable nur dort initialisiert werden, wo sie auch definiert wurde, und zwar ohne Vor anstellung von `extern` und immer ausserhalb einer Funktion. Globale Variablen sind in **allen** Funktionen und auch dem Linker bekannt.

`static` : Wird eine Variable innerhalb einer Funktion als `static` erklärt, funktioniert sie als ein Art "Gedächtnis". Sie behält ihren Wert von einem Funktionsaufruf zum nächsten. Mit den dynamischen Variablen lässt sich dieser Effekt ja nicht erreichen.
Wird eine `static` Variable im Top Level definiert, ist sie im ganzen File bekannt, aber nicht dem Linker.

Initialisierung von statische Variablen:
Statische Variablen dürfen nur mit Konstanten vorbelegt werden. Nicht vorbelegte statische Variablen werden mit 0 initialisiert. Statische Variablen sind globale Variablen (`static` oder `extern`) und `static` Variablen innerhalb von Funktionen.

Initialisierung von auto Variablen:
auto Variablen dürfen mit Konstanten **und** Variablen vorbelegt werden. Nicht vorbelegte Variablen sind undefiniert. `auto` Variablen sind lokale Variablen innerhalb von Funktionen.

Tabellarische Darstellung

	Def. ausserhalb eines Blocks	Def. ausserhalb eines Blockes **mit** Spezifizierer **static**	Def. innerhalb eines Blockes **mit** Spezifizierer **static**	Def. innerhalb eines Blockes **ohne** Spezifizierer **static**
Geltungsbereich	global	modulglobal	lokal	lokal
Lebensdauer	gesamte Laufzeit des Programms	gesamte Laufzeit des Programms	gesamte Laufzeit des Programms	Laufzeit des Blockes
Speicherklasse	-	static	static	auto
Initialisierung	implizit zu 0	implizit zu 0	implizit zu 0	keine
Speicherung	im Datenteil	im Datenteil	im Datenteil	auf dem Stack

Beispiel:

```
/*****************************************************************************************************************
 * 04-10-03          Modul mit Funktion main()              File : extern_m.cpp
*****************************************************************************************************************/

int g1 = 2;               // Definition (global bekannt, somit auch dem Linker, statisch)
extern int g2;            // Deklaration (global bekannt, statisch, in anderem Modul definiert)
extern void fun(void);    // Deklaration einer globalen Funktion

int main(void)
{
      fun();              // 1. Aufruf der Funktion fun (keine Argumente, kein Returnwert)
      g1 = g1 + g2;       // global statisch, zu g1 wird g2 addiert
      fun();              // 2. Aufruf der Funktion fun
      g1 = g1 + g2;       // global statisch, zu g1 wird g2 addiert
      fun();              // 3. Aufruf der Funktion fun

      return 0;
}
```

```
/*****************************************************************************************************************
 * 04-10-03          Modul mit Funktion fun()               File :  extern_f.cpp
*****************************************************************************************************************/
#include <iostream>
using namespace std;

int g2 = 1;                     // Definition (global bekannt, somit auch dem Linker, statisch)
static int mg = 5;              // Definition (nur im File bekannt,also modulglobal, statisch)
extern int g1;                  // Deklaration (global bekannt, statisch, in anderem Modul
                                definiert)
void fun(void)                  // Funktionsdefinition (global bekannt, somit auch dem Linker)
{
      int la=0;                 // Definition (lokal bekannt, somit nur in fun(), dynamisch)
      static int ls = 35;       // Definition (lokal bekannt, somit nur in fun(), statisch)

      la = la + 3;              // la wird um 3 erhöht (lokal)
      g1 = g1 + la;             // zu g1 wird la addiert (global, statisch)
      ls = ls + mg;             // zu ls (lokal, statisch) wird mg addiert (modulglobal, statisch)

      cout << "\tla: " << la <<"\tls: " << ls << "\tg1: " << g1 <<endl;
}
```

Ausgabe des Programms:

```
      la: 3    ls: 40    g1: 5
      la: 3    ls: 45    g1: 9
      la: 3    ls: 50    g1: 13
```

3.5 Zusammengesetzte Datentypen

3.5.1 Array

Arrays enthalten Elemente gleichen Typs. Bei der Definition von Arrays wird neben dem **Typ** der Arrayelemente auch die **Anzahl** der Elemente mit angegeben:

```
int     intarr[100];     // Array mit 100 Integer-Elementen
double  douarr[10];      // Array mit 10 double-Elementen
```

Der erste Arrayindex ist in C/C++ immer 0, so dass mögliche Arrayelemente wie folgt bezeichnet werden können:

```
intarr[0]                // erstes Element
intarr[99]               // letztes Element
intarr[i + j * k]
```

Wie man sieht, kann als Arrayindex auch ein Integerausdruck verwendet werden. Es liegt allerdings in der Verantwortung des Programmierers, dass der Wert des Ausdrucks auch einen gültigen Indexwert ergibt. **Eine automatische Indexprüfung, wie in Pascal, findet in C/C++ nicht statt.**

Die einzige Operation, die auf einen Array als Ganzes angewendet werden kann, ist die `sizeof` Operation. Alle anderen Manipulationen müssen elementweise vorgenommen werden.

Arrays mit Elementen vom Typ `char` werden in C/C++ als Strings bezeichnet. Sie müssen mit einem Null Byte (`'\0'`) abgeschlossen werden.

Initialisierung von Arrays

Das geht so vor sich, dass nach dem Initialisierungszeichen = die Werte in `{...}` eingeschlossen aufgeführt werden.

Beispiel: `int numbers[5] = {0, 1, 4, 9, 16};`

'numbers' ist ein Array von 5 Integern, die entsprechend den Werten in den geschweiften Klammern initialisiert sind.
Wird ein Array ohne Grössenangabe initialisiert, dann wird dies implizit aus der Zahl der Werte im Initialisierungsfeld bestimmt:

```
int value[ ] = {2, 4, 7};
```

ist ein Array mit 3 Elementen vom Typ integer.

```
char text[ ] = "Hello world";
```

ist ein Array (String) mit 12 Elementen vom Typ `char`. Die abschliessende `'\0'` wird bei der Initialisierung automatisch angehängt. Die geschweifte Klammer wird bei Strings weggelassen.

Sollen die Werte nach der Variabelndefinition zugewiesen oder geändert werden, muss dies einzeln geschehen:

```
value[0] = 34;     bzw.        text[0] = 'O';
value[1] = 51;                 text[1] = 'h';
value[2] = 3;                  text[2] = 'j';
                               text[3] = 'e';
                               text[4] = '\0';
```

********** **Aufgaben** **********

2) Geben Sie eine Möglichkeit an, die Zahl der Aufrufe einer Funktion zu zählen.

3) Erklären Sie den Unterschied zwischen

```
0     '0'   "0"
```

4) Was ist der Unterschied zwischen den Variablen h, k und t?

```
int h;
static int k;
void fct(void)
{
      static int t;
      .....
}
```

5) Gegeben sei folgendes Programm. Was wird nun in den 3 Fällen ausgedruckt?

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

int main( )
{
      char ca[] = "abcdef";      // Definition und Initialisierung eines Strings
      int ia[5] = {10, 4, 35};   // Definition und Init. eines integer Array's

      cout << "ca = " << ca << endl;       //1


      cout << "ca[3] = " << ca[3] << endl;      //2


      cout << "&ca[3] = " << &ca[3] << endl;    //3


      cout << "ia[2] = " << ia[2] << endl;      //4

}
```

3.5.2 Mehrdimensionaler Array

In C/C++ lässt sich auch ein mehrdimensionaler Array darstellen:

```
int DIM2 [3] [4];
```

definiert einen Array, der aus 3 Elementen besteht - jedes Element ist ein eindimensionaler Array von 4 Integern, also 3 Zeilen und 4 Kolonnen.

Anschaulich sieht `DIM2` so aus:

```
0  1  2  3
4  5  6  7
8  9  10 11
```

Ein Element aus `DIM2` wird mit Hilfe von zwei Indizes benannt, z.B.

```
DIM2[0][0]   // 0
DIM2[0][1]   // 1
DIM2[1][2]   // 6
```

Eine abgekürzte Schreibweise der Art `DIM2[i, j]` gibt es in C/C++ nicht. Die einzige Operation, die auf einen Array als Ganzes angewendet werden kann, ist die sizeof-Operation. Alle anderen Manipulationen müssen elementweise vorgenommen werden.

Initialisierung eines mehrdimensionalen Array

Mehrdimensionale Arrays müssen so initialisiert werden, dass jede Dimension eine Klammer der Art {...} enthält.

Beispiel:

```
int y[4][3] =  {
                  {1, 2, 3},   // y[0][0] - y[0][2]
                  {9, 8, 7}    // y[1][0] - y[1][2]
               };
```

`y[2][0] - y[3][2]` sind bei lokaler Definition uninitialisiert bzw. bei globaler Definition auf 0 initialisiert

```
int z[4][3] =  {{1}, {2}, {3}, {4} };
```

initialisiert nur das erste Element einer Zeile.

3.5.3 Struktur

Da die Elemente eines Arrays alle vom gleichen Typ sind, lassen sich damit nicht Objekte beschreiben, deren Daten bezüglich des Typs inhomogen sind. So sind die zu einem Menschen zugehörigen persönlichen Daten vielleicht:

Name:	Zeichen
Geburtsdatum:	gemischt (Jahr, Monat, Tag)
Eltern:	strukturiert
Wohnort:	Zeichen
etc.	

Die Sprache C/C++ ermöglicht die Definition solcher Daten mittels des 'struct'-Konstruktes.

Die Deklaration einer Datums-Struktur könnte folgendermassen aussehen:

```
struct Datum      {
            int tag;
            char monat[12];
            int jahr;
            };

struct Person     {
            char name[20];
            char vorname[20];
            Datum gebdatum;
            char strasse[30];
            int plz;
            char wohnort[15];
            };
```

Man beachte, dass im obigen Beispiel die `struct Person` die Komponente `Datum` enthält. Ausser `void` und die betreffende `struct` selber sind alle Datentypen als Komponenten verwendbar.

Diese Deklarationen liefern quasi eine Schablone für die Struktur. Bevor jene jedoch gebraucht werden können, muss mittels Variablendefinition auch noch der nötige Speicherplatz reserviert werden. Es gibt hier zwei Formen der Definition.

Entweder mit obiger Deklaration und

```
Datum heute;
Person ich, du;
```

oder dann direkt

```
struct Datum {
                  int tag;
                  char monat[12];
                  int jahr;
             } heute;
```

Einzelne Komponenten der Struktur werden mit Hilfe des Strukturoperators **.** referenziert. Der Operator assoziiert von links nach rechts.

In C muss immer das Schlüsselwort `struct` vor den selbstdefinierten Datentyp gesetzt werden.

z.B. `struct Datum heute;`

Beispiel:

```
heute.jahr  = 2001;
ich.plz     = 3000;
ich.name[0] = 'A';
ich.gebdatum.tag = 11;
du.gebdatum.jahr = heute.jahr;
```

Initialisierung einer Struktur

Strukturen werden so initialisiert, dass nach dem Initialisierungszeichen = die Werte der einzelnen Komponenten durch Kommas getrennt und in `{...}` eingeschlossen werden.

Beispiel:

```
Datum montag = { 5, "Mai", 2001 };
```

Strukturen können als Ganzes zugewiesen, als Argumente an Funktionen übergeben und als Resultat von Funktionen zurückgegeben werden (call by value). Allerdings ist die Referenz-übergabe (siehe Kapitel 7.5) meist effizienter. Andere Manipulationen müssen explizit komponentenweise erfolgen.

Als weitere Operation kann `sizeof` angewendet werden. So liefert beispielsweise

```
sizeof heute;     oder   sizeof (Datum) ;
```

die Grösse der Struktur `Datum` in Bytes. Innerhalb von Struktur werden Komponenten (mit Ausnahme von Bit-Feldern) auf Wort-Grenzen gelegt.

********** **Aufgabe** **********

6) Wie viele Bytes benötigt die obige Struktur `Datum` auf einer 32-Bit Umgebung?

7) Warum darf eine Struktur `Soso` kein Element enthalten, das vom Typ `Soso` ist, wie in

```
struct Soso  {
            int i;
            Soso s;
            char c;
        };
```

3.5.4 Array von Strukturen

So wie man Arrays aus Zeichen oder Integern bilden kann, lassen sich Arrays konstruieren, deren Elemente Strukturen sind.

Beispiel:

```
#define MAXNAM    30
#define MAXSTUD   25

struct Student {
                  char name[MAXNAM];
                  long int telefon;
                  Datum geburt;
               };

Student klasse[MAXSTUD];
```

Der Array mit Namen `klasse` besteht aus Arrayelemente, welche die Form der Struktur `Student` haben. Das erste Element von `klasse` ist `klasse[0]`; dieser Ausdruck `klasse[0]` entspricht jetzt einer Struktur-Variablen, so dass die Strukturelemente in der üblichen Weise bezeichnet werden können:

```
klasse[0].telefon
klasse[0].name[3]
klasse[0].geburt.jahr
```

Initialisierung eines Array von Strukturen

Strukturen werden analog zu mehrdimensionalen Arrays initialisiert. Es werden also nach dem Initialisierungszeichen = die Werte, in `{...}` eingeschlossen, angefügt.

Beispiel:

```
Student klasse5a[] =
                  {
                        {"Koebi Schaub",6912266, {5,"Mai",1962}},
                        {"Heini Frick",6351872, {20,"Juni",1959}},
                        {"Hans Matter",6330924,{4,"Maerz",1954}}
                  };
```

3.5.5 Union

Unions stellen bei nahezu gleicher Syntax von Strukturen eine Möglichkeit dar, Variantenrecords von Pascal oder EQUIVALENCE von Fortran nachzubilden. Der Sinn dieser Konstrukte ist, denselben Speicherbereich in unterschiedlicher Weise interpretieren zu können, d.h. mit Objekten verschiedenen Typs zu belegen.

Beispiel:
```
union Art   {
            int   ival;
            double dval;
            char  cval;
            };
Art uval;
```

Die Variable `uval` stellt ein Objekt dar, das in dreifacher Weise interpretiert werden kann: als Integerwert, Gleitpunktwert oder ein Zeichen. Der Compiler sorgt dafür, dass `uval` gross genug ist, um den grössten der drei Typen zu beinhalten. Dabei ist es wichtig zu sehen, dass die Maschinenabhängigkeit nicht im Programm sichtbar ist, sondern vom Compiler automatisch abgefangen wird. Es liegt aber in der Verantwortung des Programmierers oder der Programmiererin, den jeweils gültigen Inhalt der `union` in der richtigen Weise zu verwenden. Die Elemente einer `union` werden wie die Elemente einer Struktur verwendet:

```
uval.ival   oder     uval.dval
```

Die Union-Konstruktion kommt häufig als Element einer Struktur vor.

Beispiel :
```
struct Sym  {
               char name[20];
               int flags;
               union {
                       int ival;
                       double dval;
                       char cval;
                     } uval;
            };

Sym symtab[NSYM];
```

Diese Datenstruktur stellt eine Symboltabelle dar, auf deren Elemente man in der gewohnten Weise zugreifen kann:
```
symtab[i].uval.dval
```

Pointer auf Unions werden in der genau gleichen Weise wie bei Strukturen verwendet.

In C muss immer das Schlüsselwort `union` vor den selbstdefinierten Datentyp gesetzt werden.
z.B. `union Art uval;`

************ Aufgabe ************

8) Gegeben seien die Definitionen

```
struct Stru {                   union Uni {
            int si;                         int ui;
            char sc;                        char uc;
            double sd;                      double ud;
          } svar;                         } uvar;
```

Welchen Speicherplatz belegen die Variablen, wenn Zeichen 1, Integer 4 und double 8 Byte benötigen?

3.5.6 enum

`enum` bezeichnet einen abzählbaren Datentyp (Enumeration Type).

Beispiel:

```
enum Sieben_tage
{
      montag, dienstag, mittwoch,
      donnerstag, freitag, samstag,
      sonntag
};
```

Im obigen Beispiel ist Sieben_tage der sog. tag (engl.) des abzählbaren Typs, und `montag`, `dienstag ... sonntag` sind die Enumerators. Diese sind konstante, ganzzahlige Werte und können wie solche verwendet werden.

Variablen eines abzählbaren Typs können wie folgt definiert werden:

Mit obiger Deklaration sowie der Definition

```
Sieben_tage wochentag;
```

oder in der Art

```
enum
{
      montag, dienstag, mittwoch,
      donnerstag, freitag, samstag,
      sonntag
}  wochentag;
```

Am letzten Beispiel ist zu erkennen, dass der 'tag' in einer Definition einer `enum` nicht obligatorisch ist, er könnte aber vor `{` stehen.

Intern werden den Enumerators die Werte 0, 1, 2 ... (diese sind immer vom Typ `int`) zugeordnet. Diese Default-Zuordnung kann mit einer Zuweisung eines Konstanten-Ausdruckes durchbrochen werden.

Beispiel:

```
enum Farben
{
      gelb,
      blau = 4,
      schwarz,
      violett,
      rot = 8
};
```

ergibt die Werte 0 für `gelb`, 4 für `blau` (entsprechend Zuweisung), 5 für `schwarz` (weil auf `blau` folgend), 6 für `violett` (weil auf `schwarz` folgend) und 8 für `rot`. Es ist sogar möglich, Dinge wie `{ gelb = 1, schwarz, rot = 1 }` zu konstruieren. Ob dies allerdings sinnvoll ist, steht auf einem anderen Blatt. Mit Sicherheit werden solche Exotika in der Praxis öfter zu Schwierigkeiten führen als einem lieb ist.

In C muss immer das Schlüsselwort `enum` vor den selbstdefinierten Datentyp gesetzt werden.

z.B. `enum Sieben_tage wochentag;`

3.6 typedef

C/C++ bietet eine einfache Möglichkeit, aus vorhandenen Typen einen neuen Typ mit eigenem Namen zu definieren. Indem man einer Deklaration des Schlüsselwort `typedef` voranstellt, macht man aus dem Namen dieser Deklaration einen Typnamen.

```
typedef int GANZEZAHL;
```

GANZEZAHL zu Synonym für `int`. Oder

```
typedef char ZEI, *PZEI, STRING[10], FZEI();
```

macht `ZEI` zu einem Synonym für `char, PZEI` zu einem Pointer auf `char, STRING` steht für einen Array aus 10 Zeichen und `FZEI` für eine Funktion, welche ein Resultat vom Typ `char` liefert. Diese Synonyme können dann z.B. wie folgt gebraucht werden :

```
ZEI alpha;        // = char alpha
PZEI pter;        // = char *pter
STRING name;      // = char name[10]
FZEI func;        // = char func()
```

`typedef`‘s dienen in erster Linie zwei Zwecken: Erstens können sie in gewissen Fällen die Lesbarkeit von Programmen verbessern, und zweitens sind sie ein geeignetes Mittel, allenfalls maschinen-abhängige Typen einheitlich zu benennen. Ein Anpassen eines Programms für eine andere Maschine beschränkt sich dann auf eine Modifikation der `typedef`‘s.

3.7 Referenztyp bei Datendefinition

Der Referenztyp kann in C++ nicht nur in der Parameterliste von Funktionen verwendet werden, sondern auch bei der Definition von Daten.

Beispiel:

```
int i = 12, j = 5;
int &k = i;        // k ist ein Stellvertreter für i; i und k sind identische Variablen

k = 30;            // da k und i identisch sind, wird auch i = 30
&k = j;            // falsch, k kann kein 2. Mal gebunden werden
```

4 Ausdrücke und Operatoren

4.1 Einfache Ausdrücke

Ausdrücke (Expression) sind Vorschriften zur Manipulation von Daten. Sie können aus arithmetischen, logischen, vergleichenden oder zuweisenden Operationen bestehen. Ein wesentlicher Gesichtspunkt ist, dass jeder Ausdruck ein Ergebnis, also einen Wert besitzt. Die Reihenfolge der Anwendung von Operationen wird aus der Reihenfolge der Teilausdrücke sowie der Präzedenz der Operatoren bestimmt. Die Präzedenz ist eine Art Rangfolge in der Hierarchie der Operatoren, die den gegenseitigen Vorrang angibt.

Wie schon gesagt, können Ausdrücke Teilausdrücke enthalten; die Schachtelungstiefe ist nicht explizit begrenzt. Es ist allerdings guter Programmierstil, Ausdrücke lesbar zu notieren, selbst wenn dadurch einmal eine Zwischenvariable benötigt werden sollte.

4.2 Operatoren

Die Operatoren geben, wie in der Mathematik, Verknüpfungen zwischen Datenobjekten an. Diese Datenobjekte können Konstanten oder Variablen sein. Sie sind nicht beschränkt auf rein rechnerische Verknüpfungen, wie sie z.B. die Grundrechenarten darstellen, sondern umfassen alle in der Datenverarbeitung üblichen Elementaroperationen. Dabei ist zu beachten, dass C/C++ im wesentlichen als Implementierungssprache konzipiert wurde. Das hat für die Menge der verfügbaren Operatoren zur Folge, dass der Schwerpunkt auf den Basisoperationen liegt und nicht auf höheren Funktionen, die sich alle durch die Basisoperationen darstellen lassen.

4.2.1 Arithmetische Operatoren

C/C++ kennt folgende arithmetische Operatoren:

```
*   /   %       : Multiplikation, Division und Modulo-Funktion
+   -           : Addition und Subtraktion
```

Diese Operatoren werden alle in zweistelliger Form benutzt; d.h. in der Anwendung haben sie die Form:

```
EXPR1 OP EXPR2
```

also z.B.

```
i * 10
j + k
```

Der Subtraktionsoperator kann auch noch in einstelliger Form verwendet werden, wie in der Mathematik gebräuchlich:

```
-10          -k
```

Der Modulo-Operator % berechnet den Rest bei der Division. Ganz allgemein gilt also:

```
Modulo = a - (a / b) * b
```

Aus dieser Gleichung ist auch ersichtlich, dass der Divisionsoperator bei der Division von ganzen Zahlen den Divisionsrest abschneidet:

```
10 / 3  --> 3
10 % 3  --> 1
```

Der Modulo-Operator kann **nicht auf Gleitpunktzahlen** angewandt werden.

Die Reihenfolge der Auswertung entspricht der gewohnten Weise, die Operatoren * / % werden vor + und - ausgewertet:

```
2+4*7  entspricht  2+(4*7)
```

Bei gleicher Präzedenzstufe wird von links nach rechts ausgewertet.

Bei den assoziativen Operatoren * bzw. + ist dem Compiler (selbst bei der Klammerung!!) überlassen, in welcher Reihenfolge der Ausdruck ausgewertet wird:

```
(a+b)+c  kann u.U. auch als  a+(b+c)
```

ausgewertet werden.

Die Behandlung von Fehlersituationen wie Over- und Underflow ist von der Implementierung abhängig.

********** **Aufgaben** **********

9) Welchen Wert hat x

```
x = 20 % 6 + 4 * 3;
```

10) Klammern Sie folgende Ausdrücke entsprechend der mathematischen Präzedenz

```
a *  b + c

x % y / z

z +  u - d * e
```

4.2.2 Vergleichs- und logische Operatoren

Die Vergleichs- und logischen Operatoren liefern als Ergebnis einen Wert, der den Wahrheits-werten `true` und `false` entspricht. Vereinbarungsgemäss entspricht in C/C++ der Wert 0 dem Wahrheitswert `false`, alle anderen Werte dem Wahrheitswert `true`. Die Vergleichsoperatoren haben als Ergebnis für true den Wert 1.

Die logischen Operatoren sind von der Form:

```
EXPR_1  OP  EXPR_2
```

Zu den Vergleichsoperatoren gehören

```
>   >=   <=   <
```

Sie stehen alle auf gleicher Präzedenzstufe. Von niedriger Präzedenz sind die zwei logischen Operatoren für Gleichheit bzw. Ungleichheit:

```
==   !=
```

Beide Gruppen haben aber geringere Präzedenz als die arithmetischen Operatoren. Somit gilt :

```
i + j > k + l      entspricht     (i + j) > (k + l)
i == j > k         entspricht     i == (j > k)
```

Sehr wichtig in C/C++ sind die logischen Operatoren `&&`, `¦¦` und `!`. Am besten lassen sie sich mit einer Wahrheitstafel beschreiben. `&&` (AND) repräsentiert die logische Und-Verknüpfung, `¦¦` (OR) die logische inklusive Oder-Verknüpfung und der einstellige logische Operator `!` die Negation.

X	Y	X && Y	X ¦¦ Y	!X
true	true	true	true	false
true	false	false	true	false
false	true	false	true	true
false	false	false	false	true

Man sieht leicht, dass die AND-Funktion nur dann den Wert `true` erhält, wenn beide Operanden ebenfalls `true` sind. Analog gilt bei der OR-Funktion, dass sie nur dann den Wert `false` ergibt, wenn beide Operanden den Wert `false` haben. Das gilt auch für zusammengesetzte Ausdrücke der Art:

a) `e1 && e2 && e3 && e4` --> `true` falls alle `ei == true`

bzw.

b) `e1 || e2 || e3 || e4` --> `false` falls alle `ei == false`

Liegt also ein solcher logischer Ausdruck vor, dann kann das Ergebnis des Ausdrucks schon bekannt sein, bevor alle Teilausdrücke ausgewertet wurden; das ist genau dann der Fall, wenn im obigen Beispiel a) der Teilausdruck `e1` den Wert `false` ergibt. In diesem Fall ist das Ergebnis des Gesamtausdrucks unabhängig von den anderen Teilausdrücken `false`. Man könnte sich für diesen Fall also die Auswertung der Teilausdrücke ersparen. Genau das leisten die Operatoren `&&` und `||`. Die Teilausdrücke werden von **links nach rechts** ausgewertet, bis der Wert des Gesamtausdrucks bekannt ist.

Beispiel:
```
if( x != 0  && (y / x) > 3 )
      y = 4;
```

Hat x den Wert 0, dann wird der Teilausdruck `((y / x) > 3)` nicht mehr berechnet und somit ein "zero divide" vermieden!

********** **Aufgaben** **********

11) Klammern Sie entsprechend den angegebenen Präzedenzregeln:

```
i == j > k

i >= k && l

y / x > 3

i == l != m
```

12) Geben Sie einen Ausdruck an, der für die Zeichenvariable c den Wert true liefert, falls c einer Zahl entspricht (ASCII Wert 0 bis 9).

4.2.3 Inkrement- und Dekrementoperatoren

Die Addition bzw. Subtraktion von 1 zu einer Variablen ist in den meisten Programmen eine sehr häufige Operation. C/C++ bietet dafür zwei spezielle Operatoren an, nämlich ++ und `--`.

Ungewöhnlich an diesen Operatoren ist, dass man sie sowohl in Präfix- `(++i)` als auch als Postfix-- Notation `(i++)` verwenden kann. In beiden Fällen wird die Variable um 1 erhöht. In der Präfix-Version wird sie erhöht, bevor ihr Wert benutzt wird, in der Postfix-Notation erst danach.

Beispiel:

```
                int i = 5, k, m;

a)      k = i++;          =>    k = i;
                                i = i + 1;

b)      m = ++i;          =>    i = i + 1;
                                m = i;
```

In a) wird `k` auf den Wert 5 und `i` auf den Wert 6 gesetzt. In b) hingegen wird `i` zuerst um eins erhöht (also auf 7) und erst dann `m` zugewiesen. Die Operatoren ++ und `--` dürfen nur auf Variablen angewandt werden; ein nicht zulässiger Ausdruck wäre z.B. `(i+m)++`.

4.3 Typensichere Ein-/Ausgabe

Mit `cin, cout` und `cerr` kann sehr einfach auf den Standardoutput geschrieben bzw. vom Standardinput gelesen werden. `cin, cout` und `cerr` sind Variablen vom Typ ostream respektive istream, die durch `<iostream>` deklariert und natürlich durch den Linker dazu gefügt werden. Im Kapitel 10 wird die Ein/Ausgabe detaillierter beschrieben.

Die beiden Funktionen

```
      ostream& operator<<(ostream&, TYPE);
```

und

```
      istream& operator>>(istream&, TYPE);
```

sind überladen auf der Basis von TYPE. In `<iostream>` sind Versionen für alle eingebauten Datentypen TYPE vorhanden. Der Programmierer kann diese beiden Operatoren z.B. für struct-Typen selber überladen.

Beispiel:

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

int main( )
{
      int zahl;

      cout << "Zahl eingeben: ";
      cin >> zahl;
      cout << endl << "Eingabe: " << zahl << "\tQuadrat: "
      << zahl * zahl << endl;
}
```

4.4 Listen Ausdrücke

Mehrere Ausdrücke lassen sich zu einem Ausdruck zusammenfassen, indem man sie - durch Kommas getrennt - hintereinander schreibt.

Beispiele: a)

```
i = 2, j++, w = w - q + 12;
```

gleichbedeutend wie

```
i = 2;
j++;
w = w - q + 12;
```

b)

```
if(i = i + 2, i < 10)
      j = 12;
```

gleichbedeutend wie

```
i = i + 2;
if (i < 10)
      j = 12;
```

Der letzte Ausdruck der Liste wird ausgewertet, hier `i < 10.` Um die Lesbarkeit eines Programm's zu erhöhen, sollten Listen aber vermieden werden. Ausnahmen bilden Variablendefinitionen (`int i, k, m;`) und Initialisierung von `for` Schleifen (siehe Kap. 5.6).

4.5 sizeof Operator

In C/C++ gibt es einen unären Operator `sizeof`, der vom Compiler bewertet wird und der dazu benutzt werden kann, die Grösse eines beliebigen Objektes festzustellen.
Die Ausdrücke

```
sizeof Objekt
```

und

```
sizeof (Typname)
```

liefern jeweils eine ganze Zahl, nämlich die Grösse des angegebenen Objekts oder Datentyps gemessen in Byte. Ein Objekt kann dabei eine Variable, ein Array oder eine Struktur sein.

4.6 Präzedenz von Operatoren

Die Präzedenzstufe eines Operators gibt an, in welcher Reihenfolge er in einem Ausdruck ausgewertet wird; Operatoren mit höherer Präzedenzstufe "binden" also stärker, wie man in dem Beispiel

```
i + j * k
```

sieht. '*' bindet stärker, deshalb wird zuerst der Teilausdruck

```
j * k
```

und dann erst die Summe berechnet. Will man eine andere Reihenfolge, muss der gewünschte Teilausdruck geklammert werden:

```
( i + j ) * k
```

In der Anordnung der Präzedenzstufen ist Vorsicht geboten bei gleichzeitiger Verwendung der logischen Operatoren `& | ^` (siehe Kapitel 4.8) einerseits und den Vergleichsoperatoren `== !=` andererseits:

bei einer Bit-Test-Operation wie

```
if((x & MASK) == 1) ...
```

ist die Klammerung des Teilausdrucks (`x & MASK`) notwendig, da der Ausdruck sonst wie

```
if( x & (MASK == 1)) ...
```

interpretiert würde.

Die Präzedenz der Operatoren ist in der Tabelle auf der nächsten Seite beschrieben:

Höchste Präzedenz		
Symbol	Name oder Bedeutung	Auswertung
`(...)` *`::`*	Klammerung *Scope Operator*	
`fname(...)` `[...]` `.` `->` `++` `--`	Funktionsaufruf Array Element Struktur oder Union Komponente Pointer auf Struktur Komponente Postfix-Inkrement “a++“ Postfix-Dekrement “a- -“	links nach rechts
`!` `~` `-` `+` `++` `--` `&` `*` `(type)` `sizeof` *`new`* *`delete`* *`new[ ]`* *`delete[ ]`*	logisches NOT Einer Komplement negatives Vorzeichen *positives Vorzeichen* Präfix-Inkrement “++a“ Präfix-Dekrement “- -a“ Adresse von Pointer (Dereferenzierung) Cast Operator Grösse in Bytes (erzeugt Konstante zur Compilationszeit) *Objekt anlegen* *Objekt freigeben* *Array von Objekten anlegen* *Array von Objekten freigeben*	rechts nach links
`*` `/` `%`	Multiplikation Division Modulo	links nach rechts
`+` `-`	Addition Subtraktion	links nach rechts
`<<` `>>`	bitweises links schieben bitweises rechts schieben	links nach rechts
`<` `<=` `>` `>=`	kleiner kleiner oder gleich grösser grösser oder gleich	links nach rechts
`==` `!=`	gleich ungleich	links nach rechts
`&`	bitweises AND	links nach rechts
`^`	bitweises exklusiv OR (XOR)	links nach rechts
`|`	bitweises OR	links nach rechts
`&&`	logisches AND (Auswertung bis zum ersten False)	links nach rechts
`||`	logisches OR (Auswertung bis zum ersten True)	links nach rechts
`? :`	Bedingter Ausdruck	rechts nach links
`=` `*=, /=,` `%=, +=, -=,` `<<=, >>=,` `&=, ^=, |=`	Zuweisung Zusammengesetzte Zuweisung: “a *op*= b“ entspricht “a = a *op* b“	rechts nach links
`,`	Folge von Ausdrücken	links nach rechts
Niedrigste Präzedenz		

4.7 Konversionen

Auf Maschinenebene werden alle Datenobjekte als Bitkette dargestellt. Diese Bitkette wird vom System dann je nach Typ zum Beispiel als `int`- oder als `char`-Objekt interpretiert. Wird eine Variable eines Datentyps in einen anderen Datentyp umgewandelt, so bleibt die ursprüngliche Bitkette zwar erhalten, allerdings ändern sich Interpretation und Länge der Bitkette (`int`-Zahlen haben zum Beispiel eine Länge von zwei beziehungsweise vier Bytes, `char`-Werte sind ein Byte lang).

Können bei der Typumwandlung signifikante Stellen verloren gehen, so spricht man von unsicheren Konvertierungen (Unsafe Conversion), andernfalls von sicheren Konvertierungen (Safe Conversion). Beispiele für unsichere Typumwandlungen sind die Konvertierung von Gleitpunktwerten auf ganzzahlige Werte oder von `long`-Zahlen auf den Datentyp `int`.

In C++ wird zwischen Standard-Typumwandlungen und expliziten Typumwandlungen unterschieden. Standard-Konvertierungen werden vom System automatisch vorgenommen, explizite hingegen vom Programmierer. Daneben existiert noch die Möglichkeit, Typum-wandlungen selbst zu definieren.

Ausdrücke werden implizit (= automatisch) in Ausdrücke anderen Typs umgewandelt, wenn der tatsächliche Typ eines Ausdrucks und der ,,erwartete`` Typ nicht übereinstimmen. Möglich ist das zum Beispiel, wenn ein Ausdruck

- als Operand eines Ausdrucks verwendet wird und der erwartete Typ nicht mit dem Ergebnistyp dieses Ausdrucks übereinstimmt,
- in einer Bedingung (logischer Ausdruck) verwendet wird,
- an einer Stelle verwendet wird, an der ein Integer-Ausdruck (zum Beispiel in einer case-Anweisung) erwartet wird, oder
- in einer Initialisierung verwendet wird und der Typ des zu initialisierenden Objekts nicht mit dem Typ des Ausdrucks übereinstimmt.

Derartige Umwandlungen werden Standard-Typumwandlungen genannt.

Integer und Zeichen

Integer und Zeichen-Operanden können in arithmetischen Ausdrücken grundsätzlich beliebig gemischt werden; jeder `char`-Operand wird automatisch in ein `int`-Objekt konvertiert. Als Beispiel dafür dienen folgende Anweisungen, die drei Zeichen in eine Integer Zahl konvertieren:

```
{
      char a = '1';
      char b = '5';
      char c = '3';
      int i;

      i = (a - '0') * 100 + (b - '0') * 10 + c - '0';
}
```

Der Teilausdruck `a - '0'` ergibt den numerischen Wert der Ziffer `a`, da im ASCII-Zeichensatz die Ziffern `'0'` bis `'9'` in dichter, aufsteigender Reihenfolge vercodet sind. `i` wird hier also der Wert `153` zugewiesen.

Andere Konversionen

Die übrigen Konversionen werden nach dem Prinzip durchgeführt, dass vom kurzen zum längeren Typ umgewandelt wird. Im einzelnen führt das zu folgenden Regeln:

```
char  -->   int
short -->   int
float -->   double
```

Konversionen in Zuweisungen

Auch in Zuweisungsausdrücken wird konvertiert; es liege die Zuweisung

lval = rval (left- bzw. right value)

vor.

lval-Typ	<--	rval-Typ	Konversionsregel
`int`		`char`	wie oben beschrieben
`char`		`int`	die höheren Bits werden abgeschnitten
`float`	`int`		übliche Konversion
`int`		`float`	Abschneiden des gebrochenen Teils
`float`	`double`		Rundung
`int`		`long int`	Abschneiden der höherwertigen Bits
`char`		`long int`	Abschneiden der höherwertigen Bits

Bool-Typumwandlungen

Jeder rval eines Ausdruck, der vom Typ `int`, `enum` oder Pointer ist, kann auf einen bool-Wert umgewandelt werden. Ein Wert 0 sowie jeder Nullzeiger wird zum bool-Wert *`false`*, jeder andere Wert zu *`true`*.

Erzwungene Typenkonversionen

Die bisher aufgeführten Konversionen sind alle implizit, d.h. sie werden vom Compiler automatisch generiert. Der Programmierer hat aber auch explizit die Möglichkeit, den Typ eines Ausdrucks anzugeben.
C++ kennt sechs verschiedene Typumwandlungsoperatoren (Cast-Operatoren). Ein Cast-Operator (()) stammt noch aus dem C-Subset von C++, ein zweiter (ebenfalls ()) stellt eine alternative Notation dar, und die anderen vier *(`const_cast, static_cast, reinterpret_cast` und `dynamic_cast`)* sind erst seit kurzem Teil der Sprache.

Möchte man z.B. eine `float` Division zweier `int` Zahlen erzwingen, dann muss dies mit dem **cast** Operator dem Compiler mitgeteilt werden (siehe folgendes Beispiel).

Beispiel : Implizite Konversionen und cast- Operator

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

int main( ) {
   int x = 10, y = 3;
   float fx = 10.0, fz;

   cout << setiosflags( ios::showpoint);

   fz = x / y;     // ganzzahlige Division
   cout << "Fall 1 : \tfz = 10/3 \t\t=> " << fz << endl;

   fz = fx / y;    // float Division (implizite Konversion)
   cout << "Fall 2 : \tfz = 10.0/3 \t\t=> " << fz << endl;

   fz = (float) x / y;  // Cast int-float, C-Stil
   cout << "Fall 3 : \tfz = (float) 10/3 \t=> " << fz << endl;

   fz = float (x) / y;  // Cast int-float, Funktionsstil
   cout << "Fall 4 : \tfz = float (10)/3 \t=> " << fz << endl;
   return 0;
}
```

Dieses Programm erzeugt folgenden Output:

```
Fall 1 :  fz = 10/3          => 3.00000
Fall 2 :  fz = 10.0/3        => 3.33333
Fall 3 :  fz = (float) 10/3  => 3.33333
Fall 4 :  fz = float (10)/3  => 3.33333
```

Die Verwendung dieser ,,alten`` Typumwandlungsoperatoren wird aber nicht mehr empfohlen - sie sind in der Sprache aufgrund der nötigen Kompatibilität zu ,,alten`` Programmen aber weiterhin vorhanden.

Neue Cast-Operatoren

Bei der Einführung von neuen, sicheren Cast-Operatoren wurde darauf geachtet, mehrere Operatoren für die verschiedenen Arten von Typumwandlungen zu verwenden.

Eine Typumwandlung mit dem Operator *const_cast* wandelt einen Ausdruck vom Typ T mit den optionalen Qualifikatoren `const` und `volatile` in einen Ausdruck desselben (Basis-)Typs ohne den Qualifikator const um. Der Sinn und Zweck eines Casts ist ausschliesslich die (vorübergehende) ,,Entfernung`` des const-Qualifikators. Dazu ein Beispiel:

Folgende Funktion sei **unveränderbar** vorgegeben. Sie sucht nach einem SubString `substr` in der Zeichenkette `str`.

```
char *find(char *str, char *substr)   {
      char *p;
      bool ok=true;
      for (int i=0, j=0;ok && str[i] != '\0'; i++) {
            p = &str[i];
            while (str[i++] == substr[j++])
                  ok = false;
            j--;
      }
      return p;
}
```

Mit dieser Funktion möchte man nun die Funktion `find` benutzen:

```
const char *findSubString(const char * str, const char *subStr)
{
      return find(str, subStr);      // Fehler! find hat nur char*-Argumente
}
```

Die Argumente der Funktion `findSubString` sind vom Typ `const char*`, da keiner der Parameter verändert wird. Die Funktion find hat aber Parameter vom Typ `char*`. Der Compiler meldet beim Übersetzen der `return`-Anweisung einen Fehler, da versucht wird, `const`-Objekte als Aktualparameter für nicht konstante Formalparameter zu verwenden.

Grundsätzlich sind drei Lösungen des Problems möglich:

- Eine Lösung des Problems besteht darin, die Parameter der Funktion `findSubString` nicht als `const` zu deklarieren. Das widerspricht aber der Richtlinie, Eingangsparameter als `const` zu deklarieren. Ein Benutzer der Funktion, der nur die Schnittstelle kennt, kann keine const-Objekte als Parameter übergeben und würde zudem annehmen, dass die Parameter verändert werden könnten.

- Eine andere Lösung liegt in der Verwendung von Casts im C-Stil. Dann ist allerdings nicht klar, was der Programmierer mit dem Typumwandlungsoperator bezweckt. Zudem kann ein derartiger Cast bei einer Änderung der Aktualparametern zu ungewünschten Fehlern führen.

  ```
  return find((char*)str,(char*)subStr);
  ```

- Wesentlich besser ist es, den const-Qualifikator der Argumente mit einem *`const_cast`* zu entfernen. Damit ist einerseits die Schnittstelle der Funktion `findSubString` weiterhin korrekt definiert, und andererseits die Absicht des Programmierers klar:

  ```
  return find(const_cast<char*>(str), const_cast<char*>(subStr));
  ```

Wichtig ist zu wissen, dass ein `const_cast` nicht auf ,,echte`` const-Objekte angewandt werden darf. Echte const-Objekte sind Objekte, die bei ihrer Deklaration als const vereinbart sind. Das Ergebnis eines derartigen Casts ist undefiniert, da eine Kennzeichnung eines Objekts als const dem Compiler mitteilt, dass dieses Objekt nicht verändert wird. Der Compiler kann damit spezielle Optimierungen durchführen. Werden derartige Objekte in non-const-Objekte umgewandelt, kann es zu Problemen kommen! Unvorhersehbare Fehlerfälle sind die Folge.

```
const char ch = 'a';
cout << const_cast<char> (ch);  // Problematisch!
```

Typumwandlungen, die vom Compiler zur Übersetzungszeit als gültig beurteilt werden können, werden als ,,definiert`` bezeichnet. Sie werden mit dem Operator `static_cast` verwirklicht. Definierte Typumwandlungen sind Umwandlungen von Objekten einer Klasse auf Objekte einer Basisklasse oder die Umwandlung mittels einer Umwandlungsfunktion.
Beispiele für derartige Typumwandlungen werden in den jeweiligen Abschnitten gegeben.

Alle Typumwandlungen, die nicht von `static_cast` und `const_cast` durchgeführt werden können, werden mittels `reinterpret_cast` verwirklicht. Der Name des Operators stammt daher, dass eine derartige Typumwandlung eine neue Interpretation der zugrundeliegenden Bitkette, die das Objekt darstellt, erfordert.

```
char*  p = new char[20];
...
int* pi = reinterpret_cast<int*> (p);
```

Eine vierte Art der Typumwandlung ist die mit Hilfe von `dynamic_cast`. Diese Typumwandlung steht in engem Zusammenhang mit dem Typsystem von C++.

`dynamic_cast` und `static_cast` sind nur bei objektorientierter Programmierung von Bedeutung.

Übersicht

Situation	Cast-Operator
Vorübergehende Entfernung von const	*const_cast*
Definierte (= gültige) Umwandlung zwischen Speicherobjekten	*reinterpret_cast*
Umwandlung von polymorphen Objekten	*dynamic_cast*
Undefinierte Umwandlungen (= alle anderen Situationen)	*static_cast*

********** **Aufgaben** **********

13) Welche Konversionen finden statt bei

```
int i = 0x344, k = 0x284f;
double z = 45.77;
char c = '3';

i = c;

c = k;

k = z;

z = (double) i;
```

14) Wie sieht ein Algorithmus aus, der alle Zeichen 'a'...'z' in die Zeichen 'A'...'Z' konvertiert?

4.8 Bitweise Operatoren

Die Programmiersprache C/C++ bietet die sehr wichtige Möglichkeit, einzelne Bits bzw. Gruppen von Bits zu manipulieren. Die bitweisen Operatoren erlauben dies mit Hilfe der bekannten logischen Basisoperationen.

Die bitweisen Operatoren dürfen nicht auf Gleitpunktzahlen angewandt werden.

&	bitweises AND
¦	bitweises OR
^	bitweises XOR
<<	schieben links
>>	schieben rechts
~	1-Komplement

Die Wirkungsweise der einzelnen Operatoren zeigen folgende Wahrheitstafeln:

&	0	1
0	0	0
1	0	1

¦	0	1
0	0	1
1	1	1

^	0	1
0	0	1
1	1	0

~	0	1
	1	0

Der rechte Teilausdruck der Schiebe-Operatoren << und >> gibt die Zahl der Stellen an, um die geschoben werden soll. `x << 3` bedeutet Schieben nach links um 3 Stellen; von rechts werden Nullen nachgeschoben.

Beim Rechtsschieben ist Vorsicht geboten: bei vorzeichenlosen Werten werden von links Nullen nachgeschoben, bei vorzeichenbehafteten hängt der entsprechende "Nachschiebewert" von der Implementierung ab. Im allgemeinen wird das Vorzeichenbit nachgeschoben.

Die Wirkungen der Operatoren lassen sich am besten an einem Beispiel darlegen. Es soll ein logischer Ausdruck mit den drei Variablen `x, p` und `n` betrachtet werden. Aus dem Wert `x` sollen von der Position `p` `n`-Bits ausgefiltert werden.

Werte:
```
unsigned int x = 0x94E9, Resultat;
int p = 6, n = 7;
```

```
          <---  n=7 --->
x: 1 0 0 1'0 1 0 0'1 1 1 0'1 0 0 1          (Binärdarstellung)
                    ^           ^
                    |           |
                Bit 6 (p)    Bit 0
```

```
Resultat = (x >> p) & ~(~0 << n);
```

Resultat: 0 0 0 0'0 0 0 0'0 1 0 1'0 0 1 1 bzw. 0x53

Um diesen Ausdruck zu verstehen, zerlegen wir ihn in seine Teilausdrücke:

`(x >> p)`	:	schiebt das gewünschte Feld an das rechte Ende
`~0`	:	erzeugt lauter 1-Werte
`(~0 << n)`	:	schiebt um `n` Positionen nach links, wobei 0-Werte von rechts nachgeschoben werden.
`~(~0 << n)`	:	erzeugt eine Maske mit `n` 1-Werten am rechten Ende.

********** **Aufgaben** **********

15) Warum ist das rechte Bit im Ergebnis des Ausdrucks `x & (x-1)` immer 0?

16) Setzen Sie in der Variablen i die Bits 11 und 12 auf 1 (Bitnummerierung einer `unsigned int`: Bit 0 bis Bit 31).

17) Löschen Sie in i die Bits 2 und 4.

4.9 Bit-Felder

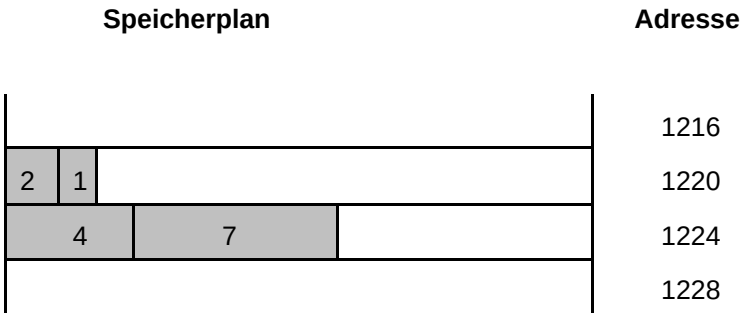
Eine spezielle Form von Komponenten erlaubt die Darstellung von Bit-Feldern gewünschter Länge. 'Bit-fiddling' wird damit u.U. vereinfacht, insbesondere, wenn eben ein Maschinenwort mehrere Felder enthält, die unterschiedlich bearbeitet werden müssen (Statusregister etc.).

```
struct Bit_feld {                   // mögliche Werte
      unsigned int four : 2;        // 0, 1, 2 oder 3
      unsigned int ok   : 1;        // 0 oder 1

      unsigned int      : 0;        // neues Wort
      unsigned int stat : 4;        // 0 - 15
      unsigned int val  : 7;        // 0 - 127
};

Bit_feld flags;
```

definiert eine Variable `flags`, welche vier Bitfelder enthält (der Feldtyp muss `int, signed int` oder `unsigned int` sein). Das erste Feld besteht aus zwei Bits, das zweite aus einem Bit das dritte Feld besteht aus 4 Bits und das vierte aus 7 Bits.
Solcherart definierte Felder sind jedoch oft mit Pferdefüssen behaftet, denn in gewissen Fällen spielt es eine vorrangige Rolle zu wissen, ob Bits von rechts nach links oder von links nach rechts gezählt werden. Felder sind übrigens keine Arrays, sie haben auch **keine Adressen**. Die maximale Feldgrösse ist implementationsabhängig. Ein namenloses Feld mit der Länge 0 bewirkt, dass das nächste Feld auf eine Wortgrenze gelegt wird (siehe im obigen Beispiel die dritte Zeile '`unsigned int : 0`').



Der Zugriff auf ein Bit_feld erfolgt mit dem von der Struktur her gewohnten **.** Operator:

Beispiele:

```
flags.ok = 1;
flags.val = flags.four;
```

4.10 Arithmetische und logische Zuweisungsoperatoren

Neben dem gewöhnlichen Zuweisungsoperator = gibt es noch eine Reihe von weiteren Zuweisungsoperatoren, die alle die gleiche Form haben:

```
E1 OP= E2
```

wobei E1 und E2 Ausdrücke sind und OP einer der zweistelligen Operatoren

```
+  -  *  /  %  <<  >>  &  ^  |
```

ist.

Die Bedeutung von E1 OP= E2 ist äquivalent zu

```
E1 = (E1) OP (E2)
```

mit der Einschränkung bzw. dem Vorteil, dass E1 nur einmal berechnet wird.

Beispiel:

```
x *= y + 1;
```
entspricht
```
x = x * (y + 1);
```

Der Vorteil dieser Konstruktion ist eine prägnantere Schreibweise, die der natürlichen Sprechweise entgegenkommt ("erhöhe x um 2", also `x += 2`). Ausserdem kann sie zu wesentlich kompakterem Code führen, wie z.B in:

```
table[i] += k;
```
statt
```
table[i] = table[i] + k;
```

Die Berechnung der Zieladresse ist nur einmal nötig (Adresse von table + i).

An dieser Stelle sei noch einmal darauf hingewiesen, dass in C/C++ die Zuweisungen Ausdrücke sind. Das hat zur Folge, dass jede Zuweisung einen Wert hat. Diesen Wert kann man wieder zuweisen und so eine Kette von Zuweisungen bilden:

```
a = b = c = d = 3;
```
entspricht
```
a = (b = (c = (d = 3) ) );
```

Damit lassen sich mehrere Variablen in einer Anweisung setzen.

********** **Aufgabe** **********

18) Welchen Wert hat j nach

a)
```
j  = 11;
j *= j + 1;
```

b)
```
j  = 11;
j += ++j;
```

4.11 Bedingte Ausdrücke

Die Eigenart, häufig vorkommende Sequenzen durch prägnante Ausdrücke zu vereinfachen, haben wir in C/C++ schon öfters gesehen. Besonders auffällig ist dies beim bedingten Ausdruck, der im praktischen Gebrauch sehr oft eingesetzt werden kann.

Es soll z.B. das Maximum von a bzw. b bestimmt werden. Konventionell geht das mit der Anweisung

```
if(a > b)
      z = a;
else
      z = b;
```

Mit dem dreistelligen Operator ?: lässt sich das wie folgt formulieren

```
z = (a > b) ? a : b;
```

Allgemein hat der bedingte Operator die Form

```
E1 ? E2 : E3
```

Der Wert von E1 wird berechnet; hat er den Wert `true`, dann bekommt der gesamte Ausdruck den Wert von E2, andernfalls den Wert von E3.

Mit diesem Operator lassen sich sehr kompakte Anweisungen schreiben. Als Beispiel soll eine Schleife angegeben werden, welche die Elemente eines Arrays mit N Elementen ausdruckt. Nach jedem Element soll ein Leerzeichen, nach jedem 10. Element und nach dem letzten Element ein NEWLINE ausgegeben werden.

Lösung:

```
for(i = 0; i < N; ++i)
      cout << a[i]  << ((i % 10 == 9 || i == N - 1) ?  '\n' : ' ');
```

********** **Aufgaben** **********

19) Der Variablen c soll das Minimum von a und b zugewiesen werden.

20) Welchen Wert hat m nach den Anweisungen

```
i = 10; j = 11;
m = i * 11 > j * 10 ? i < j ? 4 : 5 : i > j ? 6 : 7;
```

Ist diese Schreibweise erlaubt und sinnvoll?

4.12 Überladene Operatoren

Für die Standard-Datentypen hat C/C++ verschiedene unäre und binäre Operatoren vorgesehen.

Beispiel binärer Operator '+':
```
int i = 13, j = 205, k;
k = i + j;
```

Da solche Operationen nur für die Standard-Datentypen gelten, wäre es wünschenswert, sie auch auf benutzereigene Datentypen übertragen zu können.

Beispiel:
```
Complex a, b, c;
c = a + b;
```

Dies kann nun mit Überladen des Operators erreicht werden, indem der Operator '+' an die Struktur Complex angepasst wird.

Schreibweise: *Typ **operator**Operatorsymbol (Parameter){....}*

Beispiel:
```
struct Complex
{
      int i;
      int j;
};


Complex operator+(Complex a,  Complex b)
{
      a.i = a.i + b.i;
      a.j = a.j + b.j;
      return a;
}

int main( )
{
      Complex a = {121, 355};        // a.i = 121, a.j = 355
      Complex b = {50, -12};         // b.i =  50, b.j = -12
      Complex c = a + b;             // c.i = 171, c.j = 343
      c = c + c;                     // c.i = 342, c.j = 686
}
```

4.13 Scope Operator ::

Mit dem *Scope Operator ::* kann auf eine versteckte Variable zugegriffen werden.

Beispiel:
```
int x = 35;         // globale Variable x

int main( )
{
      int x = 2;    // lokale Variablen x und y, die globale x ist nun verdeckt
      int y;

      y = ::x;      // mit ::x kann globale x gelesen werden, y wird 35 zugewiesen
      ::x = x + 3; // globale x wird mit 5 überschrieben
}
```

5 Kontrollfluss

5.1 Einführung

Der Kontrollfluss eines Programms bezeichnet die Reihenfolge der auszuführenden Anweisungen. Für alle Programmieraufgaben sind drei Typen von Kontrollanweisungen ausreichend.

1) Sequenz

2) Iteration

3) Selektion

Wenn in einer Sprache diese Basiskonstrukte zur Verfügung stehen, lässt sich (theoretisch) jedes Problem formulieren. In den verschiedenen Sprachen haben die Konstrukte unterschiedliche Syntax (Oberfläche) und Semantik (Bedeutung). Insbesondere können dafür mehrere Sprachkonstrukte vorhanden sein (Beispiel in Pascal: REPEAT-UNTIL und WHILE-Schleife für die Iteration). Wenn eine Sprache nur Konstrukte der oben genannten Art enthält, gilt sie als wohlstrukturiert. Es gibt (leider) auch Anweisungen, welche diese Klassifizierung nicht beschreiben, wie z.B. die goto Anweisung.

Die Grundeinheit des Kontrollflusses ist eine Anweisung; sie kann entweder operativen (z.B. Zuweisung) oder kontrollierenden Charakter (z.B. die if Anweisung) haben.

In C/C++ kann jeder Ausdruck zu einer Anweisung gemacht werden, indem man ein Semikolon (Strichpunkt) anhängt.

Aus dem Ausdruck `x = y` wird die Anweisung `x = y;`

Das Semikolon ist also als Abschluss einer Anweisung zu verstehen und nicht als Trennzeichen wie in ALGOL-artigen Sprachen.

5.2 Blockanweisung

Neben den Ausdruck-Anweisungen gehören auch die Blockanweisungen zur Klasse der Sequenz.

Mehrere Anweisungen können zu einer Blockanweisung zusammengefasst werden. So können die Kontrollkonstrukte `if` oder `while`, die eine einzige Anweisung erfordern, über die Blockanweisung entsprechend geschrieben werden. Eine weitere Anwendung ist der Rumpf einer Funktion, die genau eine Blockanweisung sein muss.

Selbstverständlich können Blockanweisungen selbst wieder geschachtelte Blockanweisungen enthalten.

Sie werden dann in geschweifte Klammern ('{' und '}') eingeschlossen. Diese Klammern entsprechen dem BEGIN ... END aus ALGOL bzw. Pascal.

Jede Blockanweisung kann zusätzlich noch lokale Variablen besitzen, die nur in diesem Block gültig sind.
Nach der schliessenden Klammer einer Blockanweisung steht nie ein Semikolon.

Beispiel für eine Blockanweisung mit den lokalen Variablen `i` und `j`:

```
{
      int i, j;
      i = 1;
      j = 3;
}
```

oder

```
{
      int i;
      i = 4;
      {
            int j;
            j = 5;
            ++i;
      }
}
```

5.3 if Anweisung

Die bekannteste Kontrollanweisung dürfte die `if` Anweisung sein, die in mehr oder weniger gleicher Form in nahezu allen Programmiersprachen vorkommt.

Die `if` Anweisung gehört zu den selektiven Anweisungen, d.h. eine Anweisung wird in Abhängigkeit einer Bedingung ausgewählt.

Syntax der if Anweisung:

```
if ( EXPRESSION )
      STATEMENT_1
else
      STATEMENT_2
```

Der `else`-Teil ist nicht erforderlich.

Semantik : Der Ausdruck nach dem Schlüsselwort `if` wird ausgewertet. Hat er den Wert `true` (ist er also ungleich 0), dann wird die Anweisung `STATEMENT_1` ausgeführt. Hat der Ausdruck den Wert `false` (ist er also gleich 0) und ist ein `else`-Teil vorhanden, dann wird `STATEMENT_2` ausgeführt.

Hinweis : Die Syntax der `if` Anweisung ist in dieser Weise nicht immer eindeutig lösbar; die Zweideutigkeit tritt genau dann auf, wenn `if` Anweisungen geschachtelt sind:

Variante a)

```
if(n > 0)
      if(a > b)
            z = a;
      else
            z = b;
```

Variante b)

```
if(n > 0)
      if(a > b)
            z = a;
else
            z = b;
```

Durch die Einrückung sind die beiden Interpretationsmöglichkeiten angedeutet. Diese Problematik wird in der üblichen Weise gelöst: das 'else' gehört zum letzten 'else'-losen 'if'. Im obigen Beispiel ist die Variante a) die korrekte. Möchte man aber gern die andere Variante programmieren, dann kann man mit {...} den entsprechenden Effekt erzielen. Variante b) hätte dann folgendes Aussehen:

```
if(n > 0)
{
      if(a > b)
            z = a;
}
else
      z = b;
```

oder besser

```
if(n > 0)
{
      if(a > b)
      {
            z = a;
}
}
else
{
      z = b;
}
```

Weiterer Hinweis für Pascal-Kenner:

In Pascal darf vor einem `else` nie (!!!) ein Semikolon stehen. Vergessen Sie das Semikolon dennoch nicht in C/C++, denn erst das Semikolon macht aus einem Ausdruck eine Anweisung.

5.4 do while Anweisung

Die Kontrollanweisungen der Iteration erlauben die Wiederholung einer Anweisung bzw. einer Folge von Anweisungen. Die Zahl der Wiederholungen hängt von einer Bedingung ab, die im Fall der `do..while` Anweisung **nach** der Ausführung der Anweisung geprüft wird (in Pascal REPEAT-UNTIL).

Syntax der `do while` Anweisung:

```
do
      STATEMENT
while ( EXPRESSION ) ;
```

Semantik der do while Anweisung:

Die Anweisung `STATEMENT`, der Rumpf der `do while` Anweisung, wird ausgeführt. Dann wird der Ausdruck `EXPRESSION` ausgewertet. Hat er den Wert `true`, wird wieder `STATEMENT` ausgeführt usw. Die Wiederholung endet, wenn `EXPRESSION` den Wert `false` bekommt.

Hinweis :

Die `do while` Anweisung bewirkt, dass die Rumpfanweisung mindestens einmal durchlaufen wird. Die Rumpfanweisung kann selbstverständlich auch eine Blockanweisung sein.

5.5 while Anweisung

Im Gegensatz zur `do while` Anweisung wird bei der `while` Anweisung der Kontrollausdruck **vor** Ausführung der Rumpfanweisung ausgewertet.

Syntax der `while` Anweisung:

```
while ( EXPRESSION )
      STATEMENT
```

Semantik der `while` Anweisung:

Der Ausdruck `EXPRESSION` wird geprüft; hat er den Wert `true`, wird die Anweisung `STATEMENT` ausgewertet, und der Kontrollausdruck wird erneut geprüft. Das wiederholt sich solange, bis der Kontrollausdruck den Wert `false` erzeugt.

5.6 for Anweisung

Die `for` Anweisung wird häufig benutzt, wenn die Zahl der Wiederholungen beim Eintritt in die for Anweisung schon bekannt ist. Allerdings erlaubt die Syntax der `for` Anweisung weit flexiblere Kontrollmöglichkeiten als in anderen Sprachen.

Syntax der `for` Anweisung:

```
for ( EXPRESSION_1 ; EXPRESSION_2 ; EXPRESSION_3 )
      STATEMENT
```

Semantik der `for` Anweisung:

Die `for` Anweisung lässt sich einfach in die Form der `while` Anweisung übertragen:

```
EXPRESSION_1;
while ( EXPRESSION_2 )
{
      STATEMENT
      EXPRESSION_3 ;
}
```

`EXPRESSION_1` entspricht einer Initialisierung, `EXPRESSION_2` einer Kontrollbedingung, und `EXPRESSION_3` schliesst den Rumpf ab. Alle drei Ausdrücke können weggelassen werden, das zugehörige Semikolon muss aber stehen bleiben.

Also sind

```
for ( ; ; )
      STATEMENT

for ( EXPRESSION_1 ; ; )
      STATEMENT
etc.
```

gültige `for` Anweisungen. Die Bedeutung lässt sich aus der `while`-Übertragung der for Anweisung leicht erkennen, wenn man übereinkommt, dass ein fehlender Ausdruck `EXPRESSION_2` die dauernde Bedingung `true` bedeutet. `for(;;) STATEMENT` ist also wie folgt zu interpretieren:

```
while ( true )
      STATEMENT
```

Ob man die `while` oder die `for` Anweisung verwendet, ist auch eine Frage der Lesbarkeit. In dem häufigen Fall einer Zählschleife, in der eine Anweisung z.B. 10 mal wiederholt werden soll, schreibt man am besten:

```
int i, a[10];

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

In C++ ist es möglich eine nur lokal in der Schleife gültige Kontroll- und Indexvariable zu definieren:

```
int a[10];

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

5.7 Zusammenfassendes Beispiel

```
int main( )
{
  int i = 2, k, h =10;                  // Definition, Liste, Initialisierung

  if (i > 0)                            // if/else Verzweigung
  {                                     // Beginn Block, wenn Bedingung wahr
        k = h + 1;
        h = k + 3;
  }                                     // Ende Block
  else
        k = h + 5 * i;                  // Zuweisung, falls Bedingung falsch

  while (i <= k)                        // while Schleife
  {
        k--;                            // Schleifenkörper (Block)
        h = h + i;
  }

  do                                    // do while Schleife
  {
        h = h - 5;                      // Schleifenkörper (Block)
        i = h;
  }
  while (i > 0);

 for (i=0; i < 5; ++i)                  // for Schleife
        h = h + i;                      // Schleifenkörper ohne Block
```

obige for Schleife entspricht folgender while Schleife:

```
  i = 0;
  while (i < 5)
  {
        h = h + i;
        i = i + 1;
  }
}
```

5.8 break Anweisung

Während der Ausführung eines Schleifenrumpfes taucht häufig die Situation auf, dass man den Rumpf direkt verlassen möchte. Insbesondere bei Fehlersituationen kann ein kontrollierter Sprung zur Klarheit eines Programms beitragen. Die `break` Anweisung verzweigt innerhalb einer `do while, while, for` oder `switch` Anweisung zu der Anweisung, die dem entsprechenden Konstrukt folgt. Wenn möglich sollte aber die **break Anweisung vermieden werden**, da solche Sprünge nicht gerade von gutem Programmierstil zeugen. Leider ist das `break` bei der `switch` Anweisung nicht zu vermeiden.

Syntax der `break` Anweisung: `break;`

Beispiel mit `break`

```
int c;

while(true)
{
      c = cin.get();
      if(c == EOF)
            break;
      cout.put(c);
}
```

gleiches Beispiel ohne `break`

```
int c;

while((c = cin.get()) != EOF)
            cout.put(c);
```

5.9 switch Anweisung

Prinzipiell ist es auch mit der `if..else` Konstruktion möglich, aus mehr als einer Alternative zu selektionieren. Das lässt sich dann als Folge geschachtelter `if`'s schreiben. Allerdings führt das meistens zu schlechter Lesbarkeit; deshalb ist in C/C++ eine Mehrwegverzweigung integriert, die dem CASE in Pascal entspricht. Diese `switch` Anweisung prüft den Wert eines Ausdrucks auf Gleichheit mit einer Anzahl von konstanten Werten und verzweigt bei Gleichheit zu der zugehörigen Anweisungsfolge.

Syntax der `switch` Anweisung:

```
switch ( EXPRESSION )
      STATEMENT
```

Die Anweisung `STATEMENT` ist üblicherweise eine Blockanweisung, deren innere Anweisungen Marken der Form

```
case CONSTANT_EXPRESSION : STATEMENT
```

tragen können. Ausserdem kann eine Marke der Form

```
default : STATEMENT
```

vorhanden sein.

Semantik der `switch` Anweisung:

Der Ausdruck `EXPRESSION` wird ausgewertet und der Wert nacheinander mit den Konstanten der `case`-Marken verglichen. Bei Gleichheit wird die folgende Anweisungsfolge ausgeführt. Wird bei keiner `case`-Marke Gleichheit festgestellt, wird - falls vorhanden - zur `default`-Marke verzweigt. Andernfalls wird zur nächsten Anweisung nach der `switch` Anweisung gesprungen. `case`- und `default`-Marken beenden nicht die Anweisungsfolge eines Zweiges; will man einen Zweig verlassen, muss man die `break` Anweisung verwenden.

Beispiel:

```
int expr(int op1, char c, int op2)
{
      int res = 0;
      switch(c)
      {
            case '+' :
                        res = op1 + op2 ;
                        break;
            case '-' :
                        res = op1 - op2 ;
                        break;
            case '*' :
                        res = op1 * op2 ;
                        break;
            case '%' :    ;
            case '/' :
                        if(op2 == 0)
                              cout << "zero divide";
                        else
                              if(c == '/')
                                    res = op1 / op2 ;
                              else
                                    res = op1 % op2 ;
                        break;
            default :
                        cout << "operation not correct";
      }
      return res ;
}
```

Die Funktion `expr` könnte Teil eines Simulationsprogramms für einen Taschenrechner sein. Als Operanden werden zwei integer Zahlen und als Operator ein Zeichen übergeben. Mögliche Aufrufe wären:

```
expr(125, '*', 7);
expr(i, c, j);
```
etc.

5.10 goto Anweisung

Die `goto` Anweisung erlaubt den (beliebig missbräuchlichen) direkten Sprung zu einer mit einer Marke gekennzeichneten Anweisung. Theoretisch und praktisch (!!!) kommt man, mit etwas Aufwand, immer zu einer `goto`-freien Lösung. Dass C/C++ dennoch das `goto`-Konstrukt enthält, hat zum einen wohl historische Gründe, zum anderen gibt es einige ganz wenige Ausnahmen, in denen ein `goto` etwas mehr Klarheit verschaffen kann. Im wesentlichen ist das der Sprung aus einer tief verschachtelten Schleifenstruktur, die mit `break` nicht vollständig verlassen werden kann. Mit Sicherheit kann man aber auch ohne die `goto` Anweisung auskommen.

Syntax der `goto` Anweisung:

```
      goto LABEL;
      ....
      ....
LABEL:
```

Eine Marke besteht aus einem Namen und einem Doppelpunkt (':'). Die Marke kann vor jeder Anweisung der Funktion stehen, in der die `goto` Anweisung steht. Die offizielle C-Beschreibung macht keine Aussage darüber, ob auch in Unterblöcke verzweigt werden darf; aus Gründen der Programmier-disziplin sollte darauf tunlichst verzichtet werden.

5.11 Die leere Anweisung

In manchen Situationen benötigt man formal eine Anweisung, ohne dass tatsächlich eine Operation durchgeführt werden soll. Ein Semikolon (`;`) erfüllt diese Aufgabe. Als Beispiel ist hier ein leerer Schleifenrumpf gezeigt:

```
int main( )
{
      int i;
      char arr[25] = "Das ist ein String";

      for(i = 0; arr[i] != '\0'; ++i)
            ;
      cout << "arr hat " << i " Zeichen belegt\n";
      return 0;
}
```

Hier werden die Anzahl Zeichen einer Zeichenkette gezählt und ausgedruckt.

Der `sizeof` Operator würde bei `sizeof(arr)` die Grösse des definierten Speicherplatzes ausgeben. Hier also 25.

********** **Aufgaben** **********

21) Falls `i`, `k` und `m` einen Wert haben, wie gross ist er im inneren und im äusseren Block?

```
{
      int i = 2, m;
      {
            int k = i;
            int i;
            i = m + k;
      }
      m = i + k;
}
```

22) Gegeben sei folgendes Programm. Was wird hier ausgedruckt?

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

int main( )
{
      int i, ia[]  = {10, 121, 7812, 3, 44};

      for(i = 0; i <= 4; i++)
          cout <<  "ia[" << i << "] = " <<  ia[i];
      cout << endl;
      return 0;
}
```

23) Schreiben Sie eine Funktion `howlong`, die als Wert die Länge einer Zeichenkette (ohne die abschliessende 0) zurückgibt. Das i-te Element einer Zeichenkette heisst `s[i]`, das erste `s[0]`.

```
int howlong(char s[])
{
      int i;









      return i;
}
```

24) Schreiben Sie eine Funktion `umkehr`, welche die Reihenfolge der Zeichen einer Zeichenkette umkehrt (aus "abcde" wird "edcba"). Benutzen Sie dazu `howlong()` von Aufgabe 23).

```
void umkehr(char s[])
{
      char ch;
      int i, j;









}
```

6 Funktionen

6.1 Einführung

Grössere Programme werden aus verschiedenen Gründen in kleinere Einheiten aufgeteilt:

Übersichtlichkeit

Kleinere Funktionen sind besser in ihrer logischen Struktur zu überschauen.

Modularität

Verschiedene Teile können von verschiedenen Programmierern entwickelt werden. Funktionen sind ausserdem von verschiedenen Stellen des Programms aufrufbar.

Verstecken von Implementierungsdetails

In Funktionen können Entwurfsentscheidungen und Implementierungsdetails versteckt werden, die für das logische Verhalten an anderer Stelle irrelevant sind. Ändern sich dann solche Bedingungen, muss nur an wohldefinierten Teilen des Programms geändert werden.

Ein C/C++-Programm besteht im allgemeinen aus mehreren Funktionen und globalen Datendeklarationen. Diese können u.U. über mehrere Files verteilt sein; getrennte Übersetzung und die Verwendung von Bibliotheksfunktionen für Ein/Ausgabe und numerische Funktionen sind möglich.

Der Prozess der Programmentwicklung und die detaillierte Funktionsweise eines C/C++ Compilers hängt allerdings stark vom umgebenden Betriebssystem ab, so dass hier nur auf die entsprechenden Handbücher verwiesen werden kann.

6.2 Funktionsdefinitionen

Grundsätzlich gilt die Regel: **'Deklaration vor Verwendung'**. Dies trifft auch für Funktionen zu, das heisst, bevor eine Funktion verwendet (aufgerufen) werden kann, muss deren Name und der Typ des gelieferten Resultates mittels einer Deklaration der aufrufenden Funktion bekanntgegeben werden. Eine Funktion ist in jenem File, in dem sie definiert wurde, immer sichtbar. Funktionsnamen der Speicherklasse `extern` (Default) werden an den Linker exportiert. Die Speicherklasse `static` unterdrückt den Export eines Funktionsnamens an den Linker, die betreffende Funktion steht also anderen Files nicht zur Verfügung.
Liefert die Funktion keinen Wert, so verwendet man den Datentypen `void`. Die Parameter stehen zwischen (..). Sind keine Parameter zu übernehmen, so wird mit `(void)` die leere Liste angegeben. Parameter werden mit Komma getrennt.
Man beachte, dass nach der schliessenden Klammer der Parameterliste **kein** Semikolon steht.

Die Definition der Funktion besteht also aus einem Funktionskopf gefolgt vom Funktionskörper. Die allgemeine Form lautet :

```
Typ des Resultates Funktionsname (Parameterliste)
{
      Funktionskörper
}
```

Ein gültiger Funktionskopf sieht wie folgt aus:

```
double rechteck(double laenge, double breite)
{
      .....
```

Die Parameterliste kann nicht als 'Listen Ausdruck' (siehe Kap. 4.4) behandelt werden. Jeder Parameter besteht aus Typ und Namen.

Die Deklaration einer Funktion hat die Form :

```
Typ des Resultates Funktionsname (Parameterliste);
```

ANSI-konform sind 2 Varianten :

```
double rechteck(double, double);
```

oder

```
extern double rechteck(double laenge, double breite);
```

Diese Art von Deklaration nennt man **function prototype**. In function prototypes muss der Parameter**name** nicht angegeben werden, der Datentyp der Parameter genügt.

Der Funktionsaufruf erfolgt mit

```
Funktionsname (Argumentenliste);
```

z.B.

```
int main( )
{
      double flaeche, laenge = 30.5, breite = 5.0;

      flaeche = rechteck(laenge, breite);
      .....
```

Wird nur der nackte Funktionsname(ohne Typ, ohne Argumente, ohne Klammern) aufgeführt, so ist der Wert dieser Referenz die Adresse der Funktion. Dies ist zum Beispiel bei der Übergabe der Adresse der Funktion `rechteck` an die Funktion `test` der Fall :

```
errcode = test(rechteck);
```

6.3 return Anweisung

Die Rückkehr aus einer Funktion erfolgt entweder beim Erreichen des textlichen Endes des Funktionskörpers(und dabei wird ein undefinierter Wert an den Aufrufer geliefert) oder beim Erreichen einer `return` Anweisung, wobei mit dem `return` ein Resultat (Ausdruck) an den Aufrufer zurückgegeben werden kann. Eine Funktion kann keine, eine oder mehrere `return` Anweisungen haben. Die aufrufende Funktion kann, braucht aber nicht, den erhaltenen Wert weiterverwenden.

Die Syntax der `return` Anweisung lautet:

```
return EXPRESSION ;
      oder
return ;
```

Im zweiten Fall wird ein undefinierter Wert zurückgegeben.

Beispiel:

```
int f_breite(int, int);

int main(void)
{
      int laenge, breite, flaeche;

      cin >> flaeche >> laenge;
      breite = f_breite(flaeche, laenge);
      cout << "breite = " << breite << endl;
      return 0;
}
```

Kompaktform mit zwei `return`'s:

```
int f_breite(int fl, int la)
{
      if(la)
            return( fl / la );
      else
            return 0;

}
```

oder

besser mit Hilfsvariable und nur einem `return`:

```
int f_breite(int fl, int la)
{
      int br = 0;

      if(la > 0)
            br = fl / la;

      return br;
}
```

Die `return` Anweisung widerspricht in gewisser Weise den Regeln der strukturierten Programmierung, die für jedes Konstrukt nur einen Eingang und einen Ausgang erlaubt. Die Verwendung der `return` Anweisung an beliebiger Stelle bedeutet aber die Schaffung mehrerer Ausgänge und sollte somit mit Bedacht verwendet werden.
Am Ende von main steht hier nun auch eine `return` Anweisung. Da `main` eine Funktion wie jede andere ist, kann sie einen Wert an den Aufrufer zurückliefern. Dieser Wert wird an die Umgebung (Betriebssystem) geliefert, in der das Programm ausgeführt wurde. Null bedeutet, dass das Programm normal beendet wurde; andere Werte deuten auf einen fehlerhaften Abbruch.

6.4 Argumente und Parameter

Parameter können in C und C++ als **Wert** übergeben werden(call by value). In C++ ist auch eine **Referenz**-Übergabe vorgesehen(call by reference); in C ist dies nur mittels Pointernotation möglich.

6.4.1 Wertübergabe

Eine Wertübergabe bedeutet, dass beim Aufruf einer Funktion die Werte der aktuellen Parameter berechnet und dann über den Parameterübergabemechanismus an die aufgerufene Funktion übergeben werden. Die aufgerufene Funktion kann die Argumente **nicht** verändern.

Beispiel:

```
void func(int, int, int);
int main( )
{
      int x=2, y=4, z=12;
      func(x, y, z);          // Funktionsaufruf
}


void func(int x, int y, int z)
{
      x = y + 1;              // verändert x nur lokal, also wird die
      ....                    // Variable x aus main() nicht
}                             // verändert
```

Es versteht sich von selbst, dass die Ordnung (Reihenfolge) der Argumentenliste der Ordnung der Parameterliste entsprechen muss, d.h. das erste Argument wird beim Funktionsaufruf dem ersten Parameter zugeordnet etc.
Es ist auch möglich, Funktionen mit einer unbestimmten Zahl von Parametern zu verwenden. Dazu braucht es aber die Bibliothek `<stdarg.h>` (siehe Kap 11.10).

6.4.2 Referenzübergabe

Werden Adressen von Variablen (Arraynamen, Pointer) als Argumente übergeben, so kann in C, bei entsprechender Auslegung der Funktion, die Wirkung eines call by reference erzielt werden.

In C++ hingegen ist eine Referenzübergabe ohne Pointernotation möglich. Es wurde zusätzlich die Parameterübergabe "Typ& Variable" eingeführt. Dies entspricht dem Pascal "VAR" Parameter.

Beispiel:

```
void myfunc(int&, int&);      // Funktionsdeklaration

int main( )
{
      int a = 2, b = 5;

      myfunc(a, b);           // nach dem Funktionsaufruf hat a den Wert 7
      return 0;               // und b den Wert 36
}

void myfunc(int& a, int &b)   // Funktionskopf
{
      a = a + b;              // lokale a und b in main() werden verändert
      b = a * b + 1;
}
```

Diese beiden gezeigten Parameterübergaben (Wert- und Referenzübergaben) können mit allen

Standarddatentypen verwendet werden, also auch mit `char, double, struct, union` etc. Die einzige Ausnahme bildet der Array.

6.4.3 Übergabe eines Arrays

Ein **Array** kann einer Funktion zwar auch auf 2 Arten übergeben werden. Beide entsprechen aber einem 'call by reference'. Eine Wertübergabe 'call by value' ist nicht möglich.

Beispiel:

```
void func( int [] );            // Funktionsdeklaration

int main( )
{
      int arr[4] = {312, 16, 20};
                                  // Zwei gültige Parameterübergaben:
      func(&arr[0]);              // In beiden Fällen wird die Adresse des
      func(arr);                  // Integer Array's arr[] mitgegeben
}


void func(int a[])                // Funktionskopf
{
      int i;

      a[3] = 2;                   // Auf den Array a kann hier genau gleich mit
      i = a[1];                   // den eckigen Klammern zugegriffen werden.
      a[1] = 442;          // a ist hier aber eigentlich ein Pointer der auf
}                                 // int zeigt; somit ist  int *a auch gültig.
```

Die erneute Angabe der Dimension im Funktionskopf ist nicht nötig, da ja hier kein Speicherplatz reserviert, sondern lediglich der Typ des Parameters deklariert wird.

Achtung : C/C++ prüft im Gegensatz zu Pascal nicht, ob ein Arrayzugriff ausserhalb der erlaubten (d.h. definierten) Arraygrenzen erfolgt.

Mehrdimensionale Arrays als Parameter

Wird in einem Funktionsaufruf der Name eines Arrays als Parameter übergeben, dann wird damit die Anfangsadresse des Arrays an die aufgerufene Funktion weitergegeben. Diese kann dann die Elemente des Arrays direkt verändern (call by reference).

Bei mehrdimensionalen Arrays müssen die Dimensionen, mit Ausnahme der ersten (Anzahl Zeilen), angegeben werden. Dies ist erforderlich, damit eine korrekte Adressarithmetik erfolgen kann.

Beispiel:

```
void func(int mat[][14]);       // Funktionsdeklaration (mindestens Kolonnen angeben)

int main( ) {
      int matrix[7][14];        // Definition der Matrix

      func(matrix);             // Funktionsaufruf mit Adresse von matrix als Parameter
            ....
}

void func(int mat[][14]) {      // Funktionskopf (Zeilen müssen nicht angeben werden)
      mat[1][5] = 341;          // wird in matrix[1][5] abgespeichert
}                               // (Referenzübergabe)
```

C/C++ prüft im Gegensatz zu Pascal nicht, ob ein Arrayzugriff ausserhalb der erlaubten (d.h. deklarierten) Arraygrenzen erfolgt.

6.5 Beispiel zu Funktionen und Parameter

Um den Prozess der Programmentwicklung zu verdeutlichen, soll ein Programm entwickelt werden, das die folgende Aufgabe hat:

Aus einer unbestimmten Zahl von Eingabezeilen sollen diejenigen ausgegeben werden, die eine bestimmte Zeichenkette (hier "nicht") enthalten.

So ergibt sich aus
Lirum, larum Löffelstiel!
Wer das nicht kann,
der kann nicht viel.
Eins, zwei, drei,
du bist frei.

wenn nach der Zeichenkette "nicht" gesucht wird, folgendes :
Wer das nicht kann,
der kann nicht viel.

Die Grobstruktur des Programms lässt sich informell beschreiben mittels:

```
while ( NOCH_EINE_ZEILE )
      if ( ZEICHENKETTE_IN_DER_ZEILE )
            GIB_ZEILE_AUS
```

Es liegt nahe, die drei noch undefinierten Teile durch drei Funktionen zu realisieren:

1) NOCH_EINE_ZEILE entspricht einer Funktion `lieszeile(&zeile[0])`, die eine Zeile in den Array `char zeile[]` einliest.

2) ZEICHENKETTE_IN_DER_ZEILE soll durch eine Funktion `suchen(&zeile[0], &gesucht[0])` realisiert werden, die im Array `zeile[]` den Teilstring `gesucht[]` sucht und, je nachdem, ob sie ihn gefunden hat oder nicht, die Position im String `zeile[]` oder den Wert -1 zurückgibt.

3) GIB_ZEILE_AUS lässt sich durch die Standardfunktion `cout <<` realisieren, auf die später noch detailliert eingegangen wird.

Wenn man die Programmieraufgabe auf diese Weise in Funktionen aufteilt, kann eine spätere Änderung, z.B. der Suchfunktion, nur eine begrenzte Auswirkung haben. Die Gesamtstruktur des Programms bleibt erhalten.

```
#include <iostream>
using namespace std;
const int Maxzeile = 100;
int lieszeile(char []);
int suchen(char [], char []);

/*****************************************************************************************************
* Funktionsname       : main
* Beschreibung        : Eingabezeilen, die ein "nicht" enthalten, sollen ausgegeben werden
* Parameter           : keiner
* Rueckgabewert       : keiner
*****************************************************************************************************/
int main( ) {
      char gesucht[] = "nicht";
      char zeile[Maxzeile];

      while(lieszeile(&zeile[0]) > 0) {
            if(suchen(&zeile[0], &gesucht[0]) >= 0){
                  cout <<  zeile << endl;
            }
      }
      return 0;
}


/*****************************************************************************************************
* Funktionsname       : lieszeile
* Beschreibung        : liest Zeichen von der Tastatur ein
* Parameter           : in zei[ ] werden die Zeichen abgespeichert
* Rueckgabewert       : Anzahl Zeichen
*****************************************************************************************************/
int lieszeile(char zei[]) {
      int i;
      int c;

      for(i=0; i < Maxzeile-1 && (c=cin.get()) != '\n'; ++i)
            zei[i] = (char)c;

      zei[i] = '\0';


      return i;
}




/*****************************************************************************************************
* Funktionsname       : suchen
* Beschreibung        : sucht string "nicht" in der Zeile zei[]
* Parameter           : zei => Eingabezeile, ges => string "nicht"
* Rueckgabewert       : gefunden => Position, nicht gefunden => -1
*****************************************************************************************************/
int suchen(char zei[], char ges[]) {
      int i, j, k, ok = -1;

      for(i = 0; zei[i] != '\0' && ok == -1; ++i) {
            for(j=i, k=0; ges[k] != '\0' && zei[j] == ges[k]; ++j, ++k)
                  ;
            if(ges[k] == '\0')
                  ok = i;
      }
      return ok;
}
```

6.6 Rekursion

Rekursion heisst, dass eine Funktion sich selbst aufruft. Das kann ein direkter oder indirekter Aufruf sein. Der direkte Aufruf sieht so aus:

```
void func(void) {
      .....
      func();            // rekursiver Aufruf
      .....
}
```

Der indirekte Aufruf kommt erst über mehrere Aufrufstufen zu der ursprünglichen Funktion zurück, wie z.B. in:

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

void func2(void) {
      .....
      func1();           //  indirekter rekursiver Aufruf
}
```

Rekursive Programmierung kann sehr elegant und prägnant sein, aber auch undurchsichtig und fehlerhaft.

6.7 Default Parameter

Parameter können in C++ mit Default Werten vorbelegt werden. Diese Parameter können dann beim Aufruf der Funktion weggelassen werden. In der Funktion werden die weggelassenen Parameter durch die Default Werte ersetzt. Nach dem ersten Default Parameter dürfen nur noch Default Parameter folgen.

Beispiel:

Funktionsdeklaration	Aufruf der Funktion	
void fkt1(int x=3, int y=6, int z=2);	*fkt1();*	// x=3, y=6, z=2
	fkt1(4);	// x=4, y=6, z=2
void fkt2(int x, int y=6, int z=2);	*fkt2(4);*	// x=4, y=6, z=2
	fkt2(4,5,1);	// x=4, y=5, z=1
void fkt3(int x = 3, int y, int z = 2);		// Error
void fkt3(int x = 3, int y, int z);		// Error

Die Angabe der Default Parameter erfolgt nur in der Funktionsdeklaration (nicht in der Funktionsdefinition).

Funktionsdefinition :

```
void fkt(int x, int y, int z)
{
      ...
}
```

6.8 Inline Funktionen

Bei sehr kleinen Funktionen kann der Fall eintreten, dass der Funktionsoverhead unverhältnis-mässig hoch ist im Vergleich zu den Anweisungen in der Funktion. Unter Funktionsoverhead versteht man die notwendigen Operationen zur Verwaltung eines Funktionsaufrufes wie z.B. Kopien von Argumenten, Sichern von Maschinenregister, Programmsprung, Stackverwaltung. Mit dem Schlüsselwort `inline` vor der Funktionsdefinition kann in C++ erreicht werden, dass jeder Funktionsaufruf mit der ganzen Funktion ersetzt wird. Dies entspricht eigentlich einem Makro mit dem entscheidenden Vorteil, dass die Übergabetypen überprüft werden. Es ist klar, dass die Definition einer `inline` Funktion nur für kleine, häufig benutzte Funktionen sinnvoll ist. Ein guter Compiler wird es aber unterlassen, beispielsweise eine rekursive `inline` Funktion zu expandieren.

Beispiel: Es soll das Quadrat von Zahlen zwischen 1 und 10 berechnet werden

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

inline int quadrat(int);       // Deklaration

int main(void)
{
    int i=0, value;            // Lokale Variablen Definition

    while(i < 10)
    {
      i++;
      value = quadrat(i);      // quadrat wird durch i * i ersetzt
      cout << "Quadrat von " << i << " = " << value << endl;
    }
    return 0;
}


inline int quadrat(int val)
{
    return val * val;
}
```

6.9 Überladen von Funktionen

Grundsätzlich gibt man verschiedenen Funktionen auch verschiedene Namen. Wenn aber Funktionen die gleichen Aufgaben mit Objekten verschiedenen Typs verrichten sollen, kann es sinnvoll sein, diese Funktionen mit gleichem Namen zu versehen.
Das Verwenden gleicher Namen für Operationen mit verschiedenen Typen wird overloading genannt. Diese Technik ist uns bereits bekannt. So ist der Operator + sowohl für Integer Werte und Double Werte wie auch für Pointer definiert.

Es gelten dabei aber folgende Einschränkungen:

- Funktionen, die sich nur durch den Funktionswert (Rückgabetyp) unterscheiden, dürfen nicht überladen werden

- Typ und Typ & sind als Typen formaler Argumente nicht unterschiedlich genug

- Typ und const Typ sind als Typen formaler Argumente nicht unterschiedlich genug

- Typ * und Typ [] sind als Typen formaler Argumente nicht unterschiedlich genug

- Unterscheidungen von Typen in formalen Argumenten, die lediglich auf einem typedef basieren sind nicht unterschiedlich genug

- Kombinationen von obigen Definitionen führen ebenfalls zu Fehldefinitionen

Beispiele:

// Typendefinitionen
```
typedef P_ZEI char *;
```

// Gültige Funktionsüberladungen
```
void drucke( const char *str, int breite );

void drucke( const char *str );

void drucke( const double wert, int breite );

void drucke( const int wert, int breite );
```

// Ungültige Funktionsüberladungen
```
float drucke(const char *str, int breite);  // Unterschied nur im Funktionswert

void drucke(char *str, int breite);         // Unterschied im 1. Argument nicht ausreichend (const)

void drucke(P_ZEI str, int breite);   // Unterschied im 1. Argument nicht ausreichend (P_ZEI)

void drucke(const char str[], int breite); // Unterschied im 1. Argument nicht ausreichend ([ ])
```

7 Pointer

7.1 Einführung

Die Strukturierung eines Programms besteht nicht nur darin, dem Programmablauf eine bestimmte Form aufzuprägen, sondern auch eine dem Problem angepasste Datenbeschreibung zu finden. Viele Objekte, die durch ein Programm manipuliert werden, sind nicht nur durch Einzelwerte beschrieben. Vielmehr bestehen sie aus einer Reihe von Einzeldaten, die für ein Objekt ganz bestimmte Werte annehmen.

Die Möglichkeit, Daten in einer Programmiersprache zu strukturieren, dient der Klarheit und Verständlichkeit eines Programms in ganz starkem Masse.

Je nachdem, ob die Elemente einer Datenstruktur gleichen oder unterschiedlichen Typs sind, sprechen wir von Daten-Arrays oder von Strukturen.

Eine weitere Strukturierungsmöglichkeit bieten Pointer. Pointer sind Variablen, welche die Adresse (und nicht den Wert) einer anderen Variablen enthalten. Damit lassen sich Objekte zur Laufzeit miteinander verbinden, was die Konstruktion von komplexen Datentypen wie Listen, Bäumen und Schlangen zulässt.

7.2 Definition von Pointern

Jedes Objekt eines C-Programms, das Speicherplatz belegt, hat eine Adresse. Um diese Adressen manipulieren zu können, bietet C/C++ die Möglichkeit, spezielle Pointervariablen zu definieren. Eine solche Pointervariable wird zu einem Type definiert, so dass das Objekt, auf das der Pointer weist, bezüglich seines Typs erkennbar ist. Eine Variable wird zu einer Pointervariablen, indem bei der Definition das Zeichen '*' vorangestellt wird.

```
int *pti;
```

definiert `pti` als einen Pointer, der auf ein Objekt vom Typ `int` zeigt, während

```
Person *pt_to_person;
```

einen Pointer namens `pt_to_person` definiert, welcher auf eine `struct` vom Typ `person` zeigt. Schliesslich ist in

```
int (*funpt)();
```

`funpt` ein Pointer, der auf eine Funktion zeigt, welche als Resultat einen Wert vom Typ `int` liefert. Im klaren Gegensatz dazu ist in

```
int *func(void);
```

mit `func` eine Funktion deklariert, die als Resultat einen Pointer auf eine Variable vom Typ `int` liefert.

Beispiel :

```
int *pti, zahl1, zahl2, zahl3;

zahl1 = 4;
pti = &zahl2;
*pti = zahl1 + 8;
zahl3 = *pti * *pti;
```

Zuerst werden vier Variablen definiert, nämlich `pti` als Pointer auf eine Variable vom Typ `int`, sowie die drei `int`-Variablen `zahl1, zahl2` und `zahl3`. Nach der Zuweisung (`zahl1 = 4`) wird dem Pointer `pti` die Adresse der Variablen `zahl2` zugewiesen. In der nächsten Anweisung wird der Variablen, auf die `pti` zeigt (nämlich `zahl2`), die Summe des Wertes von `zahl1`, also 4, plus den Wert 8 zugewiesen.
Nun hat also

`zahl2` den Wert 12,
`pti` die Adresse von `zahl2` und
`*pti` deshalb auch den Wert 12.

Als nächstes wird der Wert der Variablen, auf die `pti` zeigt, mit sich selbst multipliziert. Das Resultat, 144, wird `zahl3` zugewiesen.

Hier noch einmal die Lesart für die verschiedenen Konstruktionen :

`&zahl`	wird gelesen als "Adresse von `zahl`" (englisch : address of `zahl`).
`*pti = ...`	wird gelesen als "Variable, auf die `pti` zeigt" (englisch : at `pti`).
`...=...*pti ...`	wird gelesen als "Wert der Variablen, auf die `pti` zeigt" (englisch : 'stuff' at `pti`).

Demnach wird mit

```
int i, j;
int *pti, *ptj;

j = 12;
pti = &i;
ptj = &j;

*pti = *ptj;
```

eine Zuweisung von Werten und mit

```
ptj = pti;
```

eine Zuweisung von Adressen vorgenommen.

Hingegen sind

```
*ptj = pti;    oder       ptj = *pti;
```

sozusagen eine unerlaubte Mischung von Kraut und Rüben, oder hier von Adressen und Integer-Werten.

Beispiel:

Man möchte in einer Funktion zwei Werte vertauschen.

```
void swap(int, int);
int main( ) {
      int i = 15, j = 35;

      swap(i, j);        // Uebergabe dieser 2 Werte
}

void swap(int i, int j) {      // Funktion zum Vertauschen zweier Werte
      int temp;

      temp = i;
      i = j;
      j = temp;
}
```

Die Funktion tauscht wohl lokal `i` mit `j` aus, aber diese Variablen sind infolge call-by-value bloss lokale Kopien der Originalwerte des Aufrufers. An den Variablen `i` und `j` in `main()` ändert sich demnach gar nichts.

Verwendet man jedoch

```
void swap(int *pti, int *ptj) {
      int temp;

      temp = *pti;
      *pti = *ptj;
      *ptj = temp;
}
```

so werden mit dem folgenden Programmcode

```
void swap(int *, int *);

int main( ) {
      int i = 15, j = 35;

      swap(&i, &j);      // Uebergabe zweier Adressen
}
```

die Adressen von `i` und `j` auf die Pointer `pti` und `ptj` in der Funktion `swap` kopiert. Damit findet der gewünschte Austausch der Werte statt. Man erreicht also ein call-by-reference.

Wie schon früher erwähnt, ist in C++ eine Referenzübergabe auch ohne Pointernotation möglich:

```
void swap(int&, int&);

int main( ) {
      int i = 15, j = 35;

      swap(i, j);        // Uebergabe dieser 2 Werte als Referenz
}

void swap(int& i, int& j) {    // Funktion zum Vertauschen zweier Werte
      int temp = i;

      i = j;
      j = temp;
}
```

7.3 Operationen mit Pointern

C/C++ ist - im Gegensatz zu Pascal - sehr grosszügig in der Verwendung von Pointern. Das ermöglicht zum einen eine sehr praxisorientierte Flexibilität, zum andern ist die Programmierung aber sehr fehleranfällig. Besonders in der Testphase eines Programms können falsch gesetzte Pointer viel Unheil anrichten. Irgendwer hat einmal gesagt, Pointer seien das goto der Datenstrukturen...

Folgende Operationen lassen sich auf Pointern anwenden:

- Vergleiche: `==  !=  <  <=  >=  >`
- arithmetische Operationen: `+  -  ++  --`
- Zuweisung: `=  +=  -=`

Erfahrungsgemäss bereitet die Art und Weise, wie in C/C++ die Arithmetik bei Pointern definiert ist, nicht nur "C-Novizen" Schwierigkeiten. Wie schon erwähnt, gehört implizit zu jedem Pointer der Typ des Objekts, auf den er weist.

```
*pti = *pti + 1;
```

inkrementiert den Wert der Variablen, auf die `pti` zeigt.

```
(*pti)++;
```

hat den gleichen Effekt, hingegen wird mit

```
*pti++;  oder pti++;
```

die Adresse in der Lokation `pti` inkrementiert (Assoziativität r -> l !). Ebenso muss der Unterschied zwischen

```
i = *pti + 5;
```

sowie

```
i = *(pti + 5);
```

klar gesehen werden. Wo Konstruktionen vom Typ `*pti++` und `*(pti + 5)` legal und sinnvoll sind, werden wir noch sehen. Schliesslich muss noch erwähnt werden, dass

```
&x
```

illegal ist, wenn `x` mit `register int x` definiert wurde. Ebenso ist

```
&3.141
```

sinnlos, und mit der Definition `int i` ist auch

```
&(i + 1)
```

nicht erlaubt. Mit anderen Worten : Der Operand des Adressoperators muss, wie bereits erwähnt, ein lvalue sein, also etwas das links vom Gleichheitszeichen '=' stehen kann.

Wird nun ein Ausdruck der Form

```
POINTER + INTEGER
```

ausgewertet, dann wird nicht einfach der Integer-Wert zur Adresse addiert, sondern es findet eine Konversion des Integerwertes in der Weise statt, dass der Wert multipliziert wird mit der Länge des Objektes, auf das der Pointer weist. Für die Subtraktion gilt sinngemäss dasselbe.

Beispiel:

In einer Implementierung habe ein Zeichen die Länge 1 Byte, ein Integer die Länge 4 Byte.

```
char *ptc, c;          // Annahme: c liegt auf Adresse 1000
int  *pti, i;          // Annahme: i liegt auf Adresse 2000

ptc = &c;              // Inhalt von ptc ist die Adresse 1000
pti = &i;              // Inhalt von pti ist die Adresse 2000

ptc = ptc + 1;         // ptc hat jetzt den Inhalt 1001
pti  = pti + 1;        // pti hat jetzt den Inhalt 2004

ptc = ptc + 2;         // ptc hat jetzt den Inhalt 1003
pti  = pti + 2;        // pti hat jetzt den Inhalt 2012
```

Durch diese Art der Arithmetik erreicht man, dass hintereinander stehende Objekte gleichen Typs mit einfacher Pointermanipulation durchlaufen werden können. Insbesondere ist diese Konstruktion portabel, da über die Länge eines Datentyps keine Angabe gemacht werden muss. Bei einer Übertragung auf einen Rechner mit anderer Datendarstellung ist also keine Änderung nötig.

********** **Aufgabe** **********

19) Falls die einzelnen Definitionen und Anweisungen in den folgenden Zeilen korrekt sind, was bedeuten sie?

```
 1    int main( )
 2    {
 3          int k, i = 4;
 4          int *pti, *ptk, **ppti;
 5
 6          pti         = &i;
 7          ptk         = &k;
 8          ppti        = &pti;
 9
10          (*pti)++;
11
12          *ptk        = **ppti + 3;
13
14          **ppti      = *ptk + 2;
15
16          *ppti       = ptk;
17
18          pti         = &&ppti;
19    }
```

7.4 Pointer und Arrays

Pointer und Arrays haben in C/C++ eine sehr enge Verwandtschaft, ja in gewissen Fällen erscheinen die beiden Objekte als ein und dasselbe. Arrays in der klassischen Form wurden bereits vorgestellt. Hier sollen nun die gewonnenen Erkenntnisse erweitert werden, denn was mit gewöhnlicher Indizierung über [] erreicht werden kann, ist auch über Pointer möglich. Oft ist die Pointer Version sogar besser und schneller.

Definieren wir einen Array, einen Pointer und zwei `int` Variablen wie folgt:

```
int arr[10] = {16, 17, 212}, *p_to_arr, k, i = 2;
```

dann zeigt `p_to_arr` nach der Anweisung

```
p_to_arr = &arr[0];      oder          p_to_arr = arr;
```

auf das 0te Element von `arr`. Entsprechend wird mit

```
k = *p_to_arr;                         // k wird 16 zugewiesen
```

k der Wert von `arr[0]` zugewiesen. Wenn nun `p_to_arr` auf ein Array Element zeigt, dann zeigt `p_to_arr + 1` auf das nächste Element, `p_to_arr + 2` auf das übernächste usw. Also ist nun

```
k = *(p_to_arr + 1)      und    k = arr[1]   // k wird 17 zugewiesen
```

dasselbe, und auch

```
k = *(p_to_arr + i)      und    k = arr[i]   // k wird hier 212 zugewiesen
```

greifen auf das gleiche Element zu.

Diese Aussagen sind immer richtig, egal um was für einen Datentyp es sich handelt. Das heisst, Pointer Arithmetik wird automatisch entsprechend der Grösse der verwendeten Objekte 'skaliert'. Dies ist ein Grund, weshalb bei der Definition eines Pointers angegeben werden muss, auf was für ein Objekt er zeigen soll. Dies ist ebenfalls der Grund, weshalb sich Integer- und Adress Arithmetik grundlegend unterscheiden.

Weitere Beispiele:

```
p_to_arr++;              // Pointer wird inkrementiert
k  = *p_to_arr;          // k wird 17 zugewiesen

--p_to_arr;              // Pointer wird dekrementiert
k  = *p_to_arr;          // k wird 16 zugewiesen
```

Beispiel:

Arrays können bekanntlich nicht direkt kopiert werden. Man möchte eine Funktion haben, die beliebige Zeichenketten kopieren kann. Die Funktion `copy` hat als Parameter die Adressen der Quellenkette `Q` und der Zielkette `Z`.

```
int main( )
{
      char Quelle[15] = "Hello world", Ziel[15];

      copy(&Quelle[0], &Ziel[0]);
}
```

ausführliche Version: **oder** Kurzform:

```
void copy(char *Q, char *Z)
{
      while(*Q != '\0')
      {
            *Z = *Q;
            Z++;
            Q++;
      }
      *Z = '\0';
}
```

```
void copy(char *Q, char *Z)
{
while((*Z++ = *Q++) != '\0')
            ;
}
```

Zur Erläuterung: `*Q` ist das Zeichen, auf das `Q` zeigt. `*Q++` entspricht `*(Q++)`. `Q` wird nach dem Zugriff auf den String um eins erhöht, zeigt also auf das nächste Zeichen. Entsprechend ist links von der Zuweisung der Ausdruck `*Z++` die Speicherstelle, auf die `Z` zeigt; `Z` wird nach dem Speichern des Zeichens um eins erhöht.

********** **Aufgabe** **********

20) Wie kann der ASCII Code (in Hex) von jedem Element eines Arrays ausgegeben werden?

```
int main( )
{
      char s[] = "Hello world";
      char *p;









}
```

7.5 Pointer auf Strukturen

An einem Beispiel soll gezeigt werden, wie ein Pointer auf eine Struktur gesetzt werden kann.
Aufgabe: Eine Funktion `init()` soll folgenden Array von Strukturen initialisieren und zur Kontrolle werden im Hauptprogramm alle Namen ausgedruckt:

```
struct Person       { char name[30];
                      int jahrgang;
                    };

Person liste[10];   // Array von 10 Strukturen
```

Die Übergabe dieser Struktur `liste` an die Funktion `init` kann nun auf drei Arten erfolgen. Das erste Beispiel zeigt eine Referenzübergabe (nur in C++ möglich):

```
#include <iostream>
using namespace std;
struct Person       { char name[30];
                      int jahrgang;
                    };

void init(Person &, char [], int);      //Referenz : &

int main( )
{
    int a=0;
    Person liste[10];

    init(liste[a++], "Roland Meier",  1951 );
    init(liste[a++], "Thomas Zimmer", 1956 );
    init(liste[a++], "Bea Beer",      1953 );

    for(int i=0; i < a ; ++i)
       cout << liste[i].name << endl;
    return 0,
}

void init(Person &element, char name[], int jahr)
{
    for(int i = 0;name[i] != '\0'; i++)
       element.name[i] = name[i];
    element.name[i] = '\0';
    element.jahrgang = jahr;
}
```

Immer jeweils die nächste Struktur aus dem Array von Personen wird als Referenz an die Funktion `init` weitergegeben. Mit dem **Postinkrement** des Indexes a wird nach der Übergabe inkrementiert; somit hat in unserem Beispiel der Index zu Beginn der for-Schleife den Wert 3. In der Funktion `init()` kann dann wie gewohnt mit dem **.** Operator auf die Elemente zugegriffen werden.

Als nächstes Beispiel soll nun die Pointernotation verwendet werden (auch in C möglich):

```
void init(Person *, char [], int);      // Pointer : *

int main( )
{
    int a=0;
    Person liste[10];

    init(&liste[a++], "Roland Meier",  1951 );        // Adresse : &liste[..]
    init(&liste[a++], "Thomas Zimmer", 1956 );
    init(&liste[a++], "Bea Beer",      1953 );

    for(int i=0; i < a ; ++i)
       cout << liste[i].name << endl;
    return 0;
}
```

```
void init(Person *p_liste, char name[], int jahr)
{
      for(int i = 0; name[i] != '\0'; i++)
            (*p_liste).name[i] = name[i];
      (*p_liste).name[i] = '\0'
      (*p_liste).jahrgang = jahr;
}
```

Der Parameter `p_liste` ist also ein Pointer auf eine Struktur vom Typ `Person`. Die Klammer im Ausdruck `(*p_liste).jahrgang` muss unbedingt stehen, da der Punktoperator eine sehr hohe Präzedenz hat; ohne Klammer würde jahrgang als Pointer betrachtet. Weil diese Konstruktion in C/C++ so häufig gebraucht wird, gibt es einen eigenen Operator. Mit dem **`->`** Operator kann somit über die Adresse einer Struktur auf ein Strukturelement zugegriffen werden.

```
void init(Person *p_liste, char name[], int jahr)
{
      for(int i = 0; name[i] != '\0'; i++)
            p_liste->name[i] = name[i];
      p_liste->name[i] = '\0';
      p_liste->jahrgang = jahr;
}
```

Präzedenz von **`.`** und **`->`**

Die Operatoren **`.`** und **`->`** haben eine sehr hohe Präzedenzstufe und binden deshalb sehr stark. In der folgenden Aufgabe wird dies gezeigt.

Gegeben sei:

```
struct Sam  {
                  int i;
                  int *pi;
            };

int x, arr[] = { 3, 8, 4};
Sam st = { 121, &arr[0]};
Sam *pst;

pst = &st;

++pst->i;          // i wird inkrementiert, also auf 122
```

Mit dem Ausdruck **`++pst->i`** wird also `i` und nicht `pst` um eins erhöht, da die Interpretation des Ausdrucks als **`++(pst->i)`** zu lesen ist; Syntax ohne `->` Operator: **`++(*pst).i`** .

************ Aufgaben ************

21) Was bedeuten die folgenden Ausdrücke? Überlegen Sie sich zuerst, was der Ausdruck **ohne** den ++ Operator aussagt bzw. auf was gezeigt wird. Dann überlegen Sie sich, **was** erhöht wird und **wann**.

```
x = *pst->pi;                 // a

*pst->pi++ = 23;              // b

(*pst->pi)++;                 // c

x = pst->i++;                 // d
```

22) Gegeben sei folgendes Programm. Was wird nun in den drei Fällen ausgedruckt?

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

int main( )
{
      struct Eins  {    char c[4];
                        char *s;
                   } ;

      struct Zwei  {    char *cp;
                        Eins ss1;
                   };

      static Eins  s1 = {"abc", "def"};

      static Zwei s2 = {"ghi", { "jkl", "mno" } };


      cout << s1.c[1] << endl << *s1.s << endl;       // 1


      cout <<  s2.cp << "\t" << s2.ss1.s << endl;           // 2


      cout <<  ++s2.cp << "\t" << ++s2.ss1.s << endl; // 3


      return 0;
}
```

7.6 Anlegen von dynamischen Objekten

Das bisher besprochene Speichermanagement zielt auf eine weitgehend automatisierte Objektverwaltung ab: Variablen werden bei der Aktivierung eines Blocks beziehungsweise vor ihrer ersten Verwendung automatisch angelegt und beim Verlassen des Blocks automatisch zerstört. Möchte der Programmierer aber die Grösse von Datenstrukturen zur Laufzeit ändern und den Zeitpunkt des Anlegens von Objekten beziehungsweise seiner Zerstörung selber kontrollieren, wird ein dynamisches Speicherkonzept benötigt.

Dazu gibt es in C++ die Operatoren new und delete (ähnlich den C Funktionen `malloc()` und `free()`).

new Typename

delete Variable

Ein durch new kreiertes Objekt existiert solange, bis es durch `delete` gelöscht wird. Damit wird verhindert, dass der bei einer Definition reservierte Speicherplatz nach Verlassen des entsprechenden Blockes freigegeben wird.

Beispiel:

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

int i = 10;

int *j = new int;                    // allocate a single int

int *arr_i = new int[50];            // allocate an array

float **arr_p = new float *[i];      // allocate 10 pointer to float

struct Complex {int re, im;};

Complex* myVar = new Complex;        // allocate a struct

...
...
delete j;                            // deallocate single int

delete [] arr_i;                     // deallocate an array

delete myVar;                        // deallocate a struct

delete [] arr_p;                     // deallocate an array of pointer
```

Kann kein Speicherplatz angelegt werden, gibt new als Rückgabewert `NULL` zurück.

```
if (j == NULL)
      cerr << "Fehler: new hat nicht geklappt ";
```

Durch die Installation eines sogenannten „New-Handler“ entfällt die Notwendigkeit der Überprüfung des Rückgabewertes. Diese Methode wird aber im vorliegenden Kurs nicht besprochen.

7.7 Argumente beim Programmaufruf

Als Anwendungsbeispiele für kombinierte Datendeklarationen soll die Übergabe von Argumenten beim Programmaufruf dienen. Die Argumente werden in der Kommandozeile an das Hauptprogramm in der folgenden Weise überreicht:

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

Der Parameter `argc` gibt die Zahl der Argumente in der Kommandozeile an, durch die das Programm aufgerufen wurde. Der Parameter `argv` ist ein Array von Pointern, die auf Charakters zeigen. Hier zeigen sie auf die einzelnen Argumente.

Wenn die Kommandozeile

echo alle Argumente

lautet, dann hätte

`argc`	den Wert 3,
`argv[0]`	wäre ein Pointer auf "echo",
`argv[1]`	ein Pointer auf "alle" und
`argv[2]`	ein Pointer auf "Argumente".

Beispiel:

Wir wollen ein Programm (echo.cpp) erstellen, welches nach seinem Aufruf einfach seine Argumente an main wiedergibt.

```
int main(int argc, char *argv[])
{
      int i;
      for(i = 1; argc > i; ++i)
            cout <<  argv[i] <<  "  " ;
      cout << endl;
}
```

Nehmen wir an, obiges Programm sei unter dem Namen 'echo' der Befehlsebene des Betriebssystems bekannt, dann resultiert mit dem Aufruf

echo alle Argumente

die Ausgabe von

alle Argumente

Betrachten wir nun das Programm im Detail:
`argc` hat den Wert 3, weil drei durch blanks getrennte Argumente vorliegen. Solange der Wert von `argc` grösser als 1 ist, wird das Programm die `for`-Schleife durchlaufen. Weil i mit 1 initialisiert wird, zeigt der Pointer `argv[i]` auf das 2. Argument “alle”. Damit wird der Programmname "echo" übersprungen. `*argv[i]` bedeutet das 0te Zeichen dieses Strings bzw. ‘a’.

Speicherplan

Adresse	Inhalt	Zugriff (verschiedene Schreibweisen: Array- und Pointer-Notation)
1000	3	`argc`
1004	1008	`argv`
1008	1020	`argv[0], *argv`
1012	1025	`argv[1], *(argv+1)`
1016	1030	`argv[2], *(argv+2)`
1020	e c h o	`argv[0][0],*argv[0], **argv`
1024	\0	`argv[0][4],*(argv[0]+4),(*argv)[4], *(*argv+4)`
1025	a l l e	`argv[1][0],*argv[1], (*(argv+1))[0],**(argv+1)`
1029	\0	`argv[1][4],*(argv[1]+4),(*(argv+1))[4],*(*(argv+1)+4)`
1030	A r g u	`argv[2][0],*argv[2], (*(argv+2))[0],**(argv+2)`
1034	m e n t	`argv[2][4],*(argv[2]+4),(*(argv+2))[4],*(*(argv+2)+4)`
1038	e \0	`argv[2][8],*(argv[2]+8),(*(argv+2))[8],*(*(argv+2)+8)`

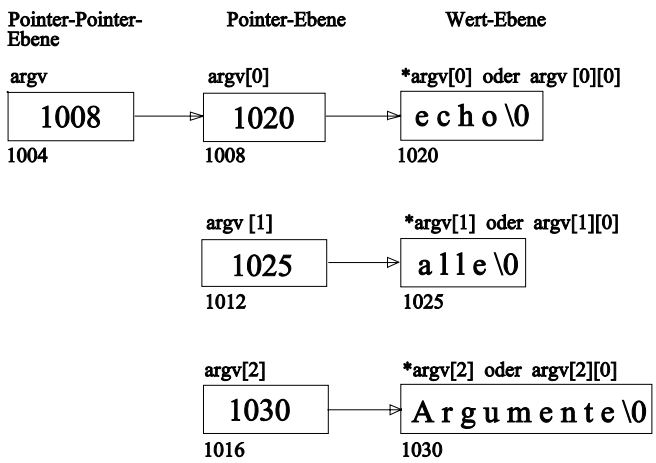
Diese diversen Zugriffsschreibweisen geben einen Überblick über die Pointerarithmetik. In der Praxis wird oft einfacher mittels Inkrement des Pointers `argv` zugegriffen:

```
++argv;                 // argv hat jetzt den Inhalt 1012, zeigt also auf 'alle'
cout.put (**argv);      // damit wird 'a' gedruckt
++(*argv);              // auf  der Adresse 1012 ist jetzt der Inhalt 1026
cout.put (**argv);      // damit wird jetzt 'l' gedruckt
```

Dieser Zusammenhang ist gut in der folgenden Pointerdarstellung ersichtlich.

Pointerdarstellung

Beschreibung:
Bezeichnung:
Inhalt:
Adresse:



Eine andere Variante desselben Programms wäre :

```
void main(int argc, char *argv[])
{
      for( ++argv; argc > 1; --argc, ++argv )
            cout << *argv <<  "  " ;
      cout << endl;
}
```

7.8 Pointer auf Funktionen

In C/C++ ist es möglich, Pointer auf Funktionen zu definieren. Eine solche Funktion kann auch einen Wert zurückgeben.

Beispiel:

Gegeben sind zwei Funktionen, die das Volumen von Körpern berechnen. Diese Funktionen sollen mittels einem Pointer aufgerufen werden. Im zweiten Teil des Hauptprogramms sollen diese Funktionen einer weiteren Funktion 'berech' übergeben werden.

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

double kegel(double, double, double);
double quader(double, double, double);
double berech(double (*) (double, double, double), double, double, double);

int main(void)
{
      double(*func_ptr)(double, double, double);
      double vol1, vol2;
      double x = 12.8, y = 8.2, z = 5.0;

      func_ptr =  kegel;            // Pointer zeigt auf Fkt. kegel
      vol1 = func_ptr(x, y, z);     // Aufruf der Fkt. durch den Pointer
      func_ptr =  quader;           // Pointer zeigt auf Fkt. quader
      vol2 = func_ptr(x, y, z);     // Aufruf der Fkt. durch den Pointer
      cout << "Volumen des Kegels: " << vol1 << " und des Quaders: " << vol2
      << endl;

      vol1 = berech( kegel,x,y,z);  // Adresse der Fkt. kegel wird mitgegeben
      vol2 = berech( quader,x,y,z); // Adresse der Fkt. quader wird mitgegeben
      cout << "Volumen des Kegels: " << vol1 << " und des Quaders: " << vol2
      << endl;
      return 0;
}


double quader(double l, double b, double h)
{
      return(l * b * h);
}

double kegel(double D, double d, double h)
{
      return(3.14 * h * (D*D + D*d + d*d) / 12);
}

double berech(double(*func_p)(double,double,double),double x,double y,double z)
{
      return func_p(x, y, z);       // gleichwertig wäre : vol=(*func_p)(x, y, z);
}
```

Resultat :

```
Volumen des Kegels:  439.652  und des Quaders:  524.800
Volumen des Kegels:  439.652  und des Quaders:  524.800
```

7.9 Typenqualifikation (type Qualifier)

C/C++ kennt die Typen Qualifier `const` und `volatile`. Sie können nur auf Ausdrücke wirken, welche lvalues ergeben (ein lvalue ist etwas, was links vom Zuweisungsoperator = stehen kann).

`const`

verhindert das Verändern eines gekennzeichneten Identifier. Folgende Anwendungen mit `const` sind möglich:

- Zuweisung einer Konstanten:

```
const float pi = 3.141;
```

der Wert der Konstanten `pi` darf nicht verändert werden, er muss also bei der Definition zugewiesen werden. Dies entspricht (mit Einschränkungen) `#define PI 3.141`

- konstanter Pointer (der immer auf das gleiche Objekt zeigt)

```
int *const ptr = &variable;
```

der Pointer `ptr` muss immer auf die gleiche Variable zeigen, also muss diese bei der Definition zugewiesen werden.

- Pointer, der auf konstanten Wert zeigt

```
const int *ptr = &konstante;
```

die Konstante darf ihren Wert nicht verändern, muss also als `const` angelegt sein. Gleiche Bedeutung hat: `int const *ptr = &konstante;`

- konstanter Pointer, der auf konstantes Objekt zeigt

```
const int *const ptr = &konstante;
```

- Funktionsparameter, der nicht verändert werden darf

```
myfunc (const int *parameter) { ... }
```

eigentlich ein 'call by reference'. Der Parameter wird hier aber vor Änderung geschützt.

`volatile`:

Objekte können zusätzlich als `volatile` (flüchtig) definiert werden. `volatile` ist ein Hinweis an den Compiler, eine Optimierung an den Stellen zu vermeiden, an denen solche Objekte auftreten. Dies ist zum Beispiel erforderlich, wenn der Wert des Objekts nicht nur durch das Programm geändert, sondern auch durch andere laufende Programme oder durch eine programmunabhängige Instanz (z.B. DMA) verändert wird. Ein `volatile` Objekt wird jedesmal direkt aus dem Hauptspeicher geladen (forced refetch).

```
volatile int port[0];
```

8 Präprozessor

8.1 Einführung

Der C Präprozessor erlaubt gewisse Manipulationen und Substitutionen im C Quellenprogramm. Alle Anweisungen an den Präprozessor bestehen aus dem Zeichen '#', einem Schlüsselwort und Parametern. Jede Präprozessor-Anweisung muss in einer separaten Zeile des Quellenprogramms stehen.

Die vom Präprozessor verarbeiteten Anweisungen sind :

`#define`	löst eine Textsubstitution aus,
`#undef`	hebt entsprechendes früheres `#define` auf,
`#include`	fügt an seiner Stelle das aufgeführte File ein,
`#if`	Code einfügen, falls Bedingung wahr ist,
`#ifdef`	Code einfügen, falls Makro definiert ist,
`#ifndef`	Umkehrung von `#ifdef`,
`#else`	alternative Klausel für `#if, #ifdef` und `#ifndef,`
`#elif`	Schachtelung von `#if`...,
`#endif`	terminiert `#if`...,
`#line`	steuert die Zeilennummerierung,
`#pragma`	führt implementationsabhängige Anweisungen aus,
`#error`	markiert einen Fehler.

8.2 #define, #undef

Mit der `#define` Anweisung können **Makros** definiert werden.

Syntax:

```
#define  IDENTIFIER  STRING
```

Der Name (IDENTIFIER) wird an allen Stellen des Programms durch die Zeichen substituiert, die hinter dem Namen in der `#define`-Zeile stehen (STRING). Ausgenommen davon sind Zeichenketten der Form "...", in denen der Name ebenfalls vorkommt.

Beispiel:

```
#define wahr  1
#define falsch 0
```

Die Zeile

```
if (wahr)
      i = 10;
```

wird vor der eigentlichen Übersetzung in

```
if (1)
      i = 10;
```

substituiert. Allerdings wird

```
cout << “wahr und falsch“;
```

nicht verändert, da in Zeichenketten nicht substituiert wird.

Diese Makros können auch parametrisiert werden. Die Definition sieht dann so aus:

```
#define  IDENTIFIER(PARAMETER_LIST)  STRING
```

Beispiel:

```
#define sqr(x)((x)*(x))
```

Die Parameter der Parameterliste werden lexikalisch ersetzt,

```
sqr(z+1)
```

ergibt dann

```
((z+1)*(z+1))
```

Man beachte insbesondere die Notwendigkeit der Klammerung.

Mit `#undef IDENTIFIER`

lässt sich sie Makrodefinition für den angegebenen Namen wieder rückgängig machen.

8.3 #include

Eine Zeile der Form

```
#include "FILENAME"
```

bewirkt, dass an dieser Stelle der Inhalt des angegebenen Files in das Quellenprogramm eingefügt wird. Dieser Mechanismus ist für zwei Aufgaben besonders nützlich:

- Die include Files können globale `#define` Anweisungen und Datendeklarationen enthalten
- In den include Files werden Funktionen deklariert, die ihre Aufrufschnittstellen nach aussen bekannt geben müssen. Beispiel dafür sind I/O-Funktionen. Prinzipiell sind damit in C/C++ auch abstrakte Datentypen implementierbar.

Die Semantik "..." respektiv <...> bezieht sich auf den Suchpfad für das File und ist implementations- und umgebungsabhängig.

8.4 #if, #ifdef, #ifndef, #else, #elif, #endif

Der Präprozessor erlaubt noch eine Reihe weiterer Kontrollanweisungen, die eine bedingte Verarbeitung ermöglichen.

Mit drei Anweisungen kann eine Bedingung geprüft werden:

```
#if CONSTANT_EXPRESSION
```

der konstante Ausdruck wird ausgewertet und auf ungleich 0 geprüft.

```
#ifdef IDENTIFIER
```

die Anweisung prüft, ob der angegebene Name schon in einem #define definiert wurde

```
#ifndef IDENTIFIER
```

es wird geprüft, ob der Name noch nicht definiert wurde.

Zusammen mit den Anweisungen `#else` und `#endif` kann denn eine verbundene Präprozessor-anweisung der folgenden Form geschrieben werden:

```
#if CONSTANT-EXPRESSION
      <beliebige Quellenzeilen>
#else
      <beliebige Quellenzeilen>
#endif
```

An Stelle des ersten `#if` können natürlich auch `#ifdef` oder `#ifndef` stehen.
Der Teil

```
#else
      <beliebige Quellenzeilen>
```

ist optional.

Wenn die `#if`-Anweisung den Wert 0 (`false`) hat, werden die Zeilen zwischen `#if` und `#else` oder, falls kein `#else` vorhanden ist, zwischen `#if` und `#endif` überlesen.

Eine gängige Verwendung von `#ifdef` findet man in Header Files und bei der Auslegung von Code für verschiedene Umgebungen.

Beispiel:

```
#define ISPROC 68000

#if ISPROC == 6809
      #define INTLGTH 2
      #define PTRLGTH 2
      lea (reg,offset);
      .....
#else
      #define INTLGTH 4
      #define PTRLGTH 4
      lea (reg,offset);
      .....
#endif
```

8.5 #error

Diese Direktive hat die Form

```
#error message
```

Stösst der Präprozessor auf sie, so wird die message ausgegeben.

Beispiel: Im File test.cpp stehe

```
#define INTSIZE 16

#if INTSIZE < 32
#error INTSIZE zu klein
#endif
```

Versucht man nun das File zu kompilieren, so wird eine Fehlermeldung

```
INTSIZE zu klein
```

ausgegeben und die Compilation abgebrochen.

9 Erstellen modularer C/C++ Programme

9.1 Einleitung

Der Einfachheit halber hatten wir uns in den Übungen zu den C-Programmen auf ein File beschränkt, in dem sowohl das Hauptprogramm als auch alle benötigten Funktionen standen.
Professionelle Softwarepakete bestehen aber - bedingt durch ihre Grösse - aus einer Vielzahl von Modulen. Die Gesamtaufgabe wird aufgegliedert in Einzelpakete, den Modulen. Die Standardbibliothek von C/C++ ist ebenfalls eine solche Sammlung von Modulen. Diese modulare Programmierung bietet sehr viele Vorteile.

9.2 Vorteile einer Aufteilung in Module

- Wiederverwendbarkeit

Ein grosses Programm mit einer Vielzahl unterschiedlicher Aufgaben ist kaum wiederverwendbar. Wird hingegen das Programm aufgeteilt in sinnvolle kleine Einheiten, die Module, so bestehen gute Chancen, die einzelnen Module in Folgeprojekten wieder zu verwenden.

- Testmöglichkeit

Module können einzeln getestet werden und erlauben so bessere Testmöglichkeiten. Dies wirkt sich positiv auf die Qualität des Softwareproduktes aus. Jedes Modul kann so auf jede erdenkliche Art genau getestet werden. Auch die Wiederverwendung trägt zur Qualität bei, da durch mehrfachen Einsatz das Produkt besser getestet und erprobt werden muss.

- Mehrere Bearbeiterinnen und Bearbeiter am selben Projekt

Ohne eine Aufteilung in Module wäre dies nicht möglich. Erst die Modularisierung ermöglicht dem Programmierer, ein für ihn klar definiertes Gebiet zu implementieren und zu testen.

- Übersicht

Ein grosses Programm mit unterschiedlichen Aufgaben ist unübersichtlich. Eine Applikation aus kleinen, einfachen Modulen ist klar gegliedert und viel einfacher wartbar.

- Kapselung (Information Hiding)

Nach dem Geheimnisprinzip von Parnas stellt ein Programm dem Benutzer nur genau so viel Informationen zur Verfügung, als er unbedingt benötigt. Alle Implementierungsdetails werden der Benutzerin und dem Benutzer bewusst verborgen.
Dies bedeutet, dass der Benutzer zu internen Variablen keinen Zugriff hat und deren Aussehen und Typ nicht kennt. Statt dessen liefert ihm das Programm Schnittstellenfunktionen, die ihm erforderliche Auskünfte geben.
Beispiel: Bei einem Robotersteuerungsprogramm soll die Geschwindigkeit des Roboterarmes abgefragt werden. Die interne Geschwindigkeitsvariable ist nach aussen verborgen. Statt dessen liefert die Funktion getSpeed() den Geschwindigkeitswert.
Diese Trennung hat viele Vorteile. Kein anderes Programm hat die Möglichkeit, versehentlich den Speed-Wert zu überschreiben (ein Überschreiben von Werten durch nicht identifizierbare Programmstellen gibt es also nicht mehr), und die interne Darstellung des Speed-Wertes ist freibleibend. Der Benutzer bekommt immer den richtigen Wert in der richtigen Dimension, unabhängig davon, ob der Wert intern als `float` oder `long int` dargestellt wird oder ob der Wert intern in m/s oder mm/s dargestellt ist.
Durch die Modularisierung und Aufteilung der Files in ein Headerfile *.hpp und in ein Implementationsfile *.cpp (siehe unten) wird das Information Hiding unterstützt bzw. erst möglich.
Werden die Schnittstellen zwischen den am Projekt beteiligten Personen zu Beginn definiert und in *.hpp Files abgelegt, ist der Spielraum für alle Beteiligten bereits gegeben.

9.3 Aufteilung von Programmen in Headerfile und Implementationsfile

File: **projekt.cpp** Hauptprogramm

```
#include <iostream>

#include "kurve.hpp"
#include "util.hpp"
#include "check.hpp

int main (int argc, char *argv[ ] )
{
    ......
}
```

File: **kurve.hpp** Modul 1

```
enum pos {UNTEN,MITTE,OBEN};

extern int interpolate (int x, int y);

extern int state;
```

File: **util.hpp** Modul 2

```
extern int getjob (int nr);

extern int showParam (int jobnr);
```

File: **check.hpp** Modul 3

```
extern int checkMode ( int unit );

extern int shutdown ( void );
```

File: **kurve.cpp**

```
#include “kurve.hpp”

int state;  /* globale Variable */

int interpolate (int x, int y)
{
.............
.............
}
```

File : **util.cpp**

```
#include “util.hpp”

int getjob ( int nr)
{
    ........
}

int showParam ( int jobnr )
{
    ........
}
```

File : **check.cpp**

```
#include “check.hpp”

int checkMode ( int unit )
{
    ........
}

int shutdown ( void )
{
    ........
}
```

9.3.1 Das Hauptprogramm

Die Funktion main steht separat in einem getrennten File. In diesem File steht nur `main()` und normalerweise keine weiteren Zusatzfunktionen. Das Hauptprogramm enthält alle notwendigen `#include`-Anweisungen, um die von ihm benötigten Funktionen zu deklarieren.

File: projekt.cpp /*Der Filename des Hauptprogramms ist üblicherweise identisch mit dem Filenamen des lauffähigen Programms */

```
//---------- Include für Standardbibliothek ----------
#include <iostream>
#include <......>

//---------- Include für weitere Module ----------
#include "kurve.hpp"
#include "util.hpp"
#include "check.hpp"

//---------- Präprozessor Direktiven, sofern sie nur für das Hauptprogramm von Interesse sind
#define MAXSPEED 3000
#define MAXAXES 3

//---------- main -----------------
int main(int argc, char *argv[])
{
      ..............
      ..............


}
```

Für das Hauptprogramm gibt es kein Headerfile. Ein Headerfile enthält selber nie ein `#include` eines weiteren *.hpp Files. Hier besteht die Gefahr der Rekursion.

9.3.2 Die Module

Das Programmpaket wird so aufgeteilt, dass von einem Modul entweder

- eine zusammenhängende Teilaufgabe erfüllt wird oder
- eine Sammlung gleichartiger Funktionen entsteht.

Eine zusammenhängende Teilaufgabe wäre z.B. ein Dialogfenster.
Sammlung gleichartiger Funktionen: Die C/C++ Standard Libraries z.B. sind nach diesem Prinzip zusammengefasst. Module mit gleichartigen Funktionen werden gerne zu gross. Es ist deshalb empfehlenswert, Untergruppen von gleichartigen Funktionen zu bilden, um die Module klein und übersichtlich zu halten.

Ein Modul wird in zwei Files aufgeteilt:

- dem Headerfile	**kurve.hpp**	als Modulschnittstelle
- dem Implementationsfile	**kurve.cpp**	als Modulkörper

Ein Modul umfasst normalerweise nur etwa ein bis zwei Seiten Source-Code. Ein grosses Modul mit sehr vielen zusammengehörenden Funktionen sollte nach Möglichkeit nie länger als zehn Seiten werden.

Modulschnittstelle, das Headerfile *.hpp

File: kurve.hpp

```
//------------- Präprozessor Direktiven, die für die Anwender des Moduls von Interesse sind
#ifndef Kurve_H     // verhindert das Mehrfacheinfügen
#define Kurve_H

//------------- Deklarationen, die für die Anwender des Moduls von Interesse sind
typedef long int BIG;
enum pos { UNTEN,MITTE, OBEN};

/*------------- Funktions Prototypen,  hier stehen nur die Funktionen, die dem Benutzer zugänglich
          gemacht werden sollen. */
extern int interpolate(int x, int y);

/*------------- Globale Daten, hier darf nur der Import (extern) von globalen Daten stehen, nicht die
            globale Variable selbst.  Anmerkung: Globale Daten sind zu vermeiden, besser mit
            Zugriffsfunktionen arbeiten. */
extern int state;     // die Variable state ist im zugehörigen Implementationsteil definiert

#endif
```

Das *.hpp File deklariert die gemeinsamen Elemente (Konstanten, Typen, Variablen, Funktionen) zwischen dem Modul und dem übergeordneten Programmteil (Main oder Funktion aus einem anderen Modul).

Modulkörper, das Implementationsfile *.cpp

File: kurve.cpp

```
#include <iostream>
#include "kurve.hpp"      /* Eigenes Modul zur Überprüfung der Deklarationen ebenfalls
                             immer aufführen */

//--------Präprozessor Direktiven, die nur modulintern von Interesse sind----
#define MAXW 1000

//-----globale Daten-----
int state;          // wird im Headerfile kurve.hpp als 'extern int state;' deklariert

//-----modulglobale Daten-----
static int xpos, ypos;

//-----modulglobale Funktions Prototypen -----
static int calibrate(int x, int y);

//------Implementation der Funktionen-------
int interpolate(int x, int y)
 {
 .....
 }

static int calibrate(int x, int y)
 {
 .....
 }
```

9.4 Programmentwicklung mit Modulen

Wird nur mit einem File gearbeitet, so ist der Aufruf von Compiler und Linker höchst einfach. Bei modularer Software sind die Module separat zu kompilieren, und es sind deren Abhängigkeiten untereinander zu beachten. Nach dem Kompilieren müssen die einzelnen Objekt-Files noch gelinkt werden.

9.4.1 Compilation

Die Aufteilung in Module hat auf den Aufruf des Compilers keine Auswirkung. Jedes Modul holt sich über die Includes die jeweiligen Deklarationen bzw. Funktionsprototypen selber. Damit ist der Compiler zufrieden; die zu den Headerfiles zugehörige Implementation braucht er nicht zu kennen.

Beispiele :

cxx -g -w0 -c kurve.cpp	erzeugt ein Objekt File kurve.o (Standard-UNIX Compiler)
g++ -ggdb -Wall -c kurve.cpp	erzeugt ein Objekt File kurve.o (GNU-Compiler)

9.4.2 Linker

Der Linker muss alle beteiligten Module zu einem exec File linken. Dem Linker muss deshalb beim Aufruf eine Liste der beteiligten Objekt Files mitgeteilt werden.

Dies kann geschehen durch:

- Eingabe der Filenamen in der Kommandozeile

  cxx projekt.o kurve.o util.o check.o -o projekt bei mehrfachem Aufruf wird dies lästig

- Batch File (bei UNIX heisst das Script)

  Beispiel analog oben. Dies ist nicht besonders komfortabel.

- Make Utility

  Geänderte Files werden erkannt und automatisch zur Neubesetzung und zum Linken angemeldet. Dabei wird anhand der Definitionen im Makefile deren Abhängigkeit untereinander erkannt. Falls notwendig werden auch nicht geänderte Module neu übersetzt, wenn z.B. ein von ihm benötigtes Headerfile abgeändert wurde.

9.4.3 Make Utility

Bei einem Programmpaket, das aus einer Vielzahl von Modulen besteht, muss darauf geachtet werden, dass nach jeder Programmänderung die entsprechenden Module neu übersetzt werden und umgekehrt bei jedem geänderten Objekt File neu gelinkt wird. Make Utilities automatisieren diesen Vorgang.

Make Utility beim PC

Zum Beispiel bei Borland C/C++ oder Microsoft C/C++ gibt es eine Projektverwaltung. In der Projektverwaltung werden alle zum Projekt gehörenden *.cpp Files eingetragen. Borland und Microsoft C/C++ erzeugen daraus selbständig ein Makefile (Projekt File).

Make Utility unter UNIX

Unter dem Filenamen 'Make' kann bei UNIX ein Makefile angelegt werden. Dazu muss zuvor die Abhängigkeit der Module untereinander bekannt sein (siehe Myers).

10 Ein-/Ausgabe in C++: Streams

10.1 Streams als Abstraktion der Ein-/Ausgabe

Alle Objekte werden durch binäre Zeichenfolgen repräsentiert, die durch die jeweiligen Typen eine spezielle Bedeutung haben. Die Ausgabe eines Objekts kann daher auch als das Problem betrachtet werden, diese binären Zeichenfolgen in eine Folge von für Menschen ,,verständliche`` Textzeichen umzuwandeln. Diese Textzeichen können dann vom System auf dem Bildschirmangezeigt oder in einer Datei gespeichert werden. Die Eingabe von Objekten wiederum ist dann als Umwandlung von Textzeichen in die binäre Repräsentation der Objekte zu sehen.

Ein Stream repräsentiert einen ,,Strom`` (= sequentielle Abfolge) von Zeichen. Auf diese Ströme können Zeichen sequentiell geschrieben und von ihnen gelesen werden.

Für vordefinierte Datentypen stehen für die Ausgabe bereits entsprechende Operator-Funktionen (<<) zur Verfügung. Die Ausgabe von ,,eigenen`` Typen wird realisiert, indem ,,neue`` Operator-Funktionen << implementiert werden, welche die nötige Umwandlung vornehmen.

Ähnliches trifft auf die Eingabe zu. Die Operator-Funktion >> wandelt die lesbare Repräsentation von Objekt-Werten in die entsprechende binäre Form um. Die Eingabe kann ebenso für eigene Datentypen erweitert werden, indem entsprechende Operator-Funktionen >> implementiert werden.

Standardmässig stehen in C++-Programmen vier Ströme für die Ein-/Ausgabe zur Verfügung:

- *cin* stellt den Standard-Eingabestrom dar. Dieser Strom ist (normalerweise) mit der Tastatur ,,verbunden``, so dass jede Leseoperation gleichbedeutend mit dem Lesen einer Zeicheneingabe von der Tastatur ist.

- *cout* ist der Standard-Ausgabestrom, der üblicherweise direkt mit dem Bildschirm ,,verbunden`` ist. Jede Schreiboperation auf *cout* entspricht damit der Darstellung eines Objekts beziehungsweise Zeichens auf dem Bildschirm.

- *cerr* ist ein (ungepufferter, siehe später) Ausgabestrom wie *cout* und ist mit dem Standard-Fehler-Ausgabestrom gekoppelt. Er wird (üblicherweise) ausschliesslich zur Ausgabe von Fehlermeldungen verwendet.

- *clog* ist ebenso wie *cerr* ein Ausgabestrom, der mit dem Standard-Fehler-Ausgabestrom gekoppelt ist.

Mit Strömen können auch beliebige Dateien assoziiert und verarbeitet werden. Dabei wird jede Datei (wie im Fall der Ein-/Ausgabe) als eine Folge von Zeichen gesehen, die gelesen und beschrieben werden kann.

10.2 Ausgabe

Die Klasse *ostream* stellt Methoden zur Ausgabe aller vordefinierten Datentypen (`char, bool, int` etc) zur Verfügung. Alle ,,Ausgabemethoden`` sind überladene Versionen des Operators <<. Die verschiedenen Versionen unterscheiden sich dabei in ihren Parametern und haben etwa folgende Schnittstelle:

```
ostream& operator<<(bool n);
ostream& operator<<(int n);
ostream& operator<<(short n);
ostream& operator<<(double n);
```

Jede der Funktionen wandelt das jeweilige Objekt (angegeben durch den Parameter) in die lesbare Form um, indem die entsprechenden Werte auf den aktuellen Strom geschrieben werden. Der Rückgabewert ist der (veränderte) Strom selbst.

Eine Anweisung

```
cout << "Wert von i: " << i << endl;
```

ist damit nichts anderes als die Sequenz der einzelnen Operator-Funktionen:

```
((cout.operator<<("Wert von i: ")).operator<<(i)).operator<<(endl);
```

Zeichen und Zeichenketten können auch über die Element-Funktionen *put(char c)* beziehungsweise *write(char* s, int n)* ausgegeben werden. *put* gibt ein Zeichen aus, während *write* eine Folge von n Zeichen ausgibt. Diese beiden Funktionen führen eine ,,unformatierte`` Ausgabe durch. Bei den <<-Ausgabefunktionen hingegen kann das Format festgelegt werden und man spricht von ,,formatierter`` Ausgabe.

Wichtig ist auch zu wissen, dass die Ausgabe in der Regel ,,gepuffert`` erfolgt: Das System speichert aus Effizienzgründen x Zeichen zwischen und gibt sie dann ,,gemeinsam`` aus. Durch den Aufruf der Methode *flush* kann die Ausgabe aber auch erzwungen werden. Die Anweisungen

```
cout << i << flush;
cout << j; cout.flush();
```

geben die Werte von i und j unmittelbar aus. Die beiden Möglichkeiten, den Ausgabepuffer zu entleeren, sind gleichwertig.

10.3 Eingabe

Die Eingabe ist ähnlich organisiert wie die Ausgabe. Die Klasse *istream* ist die Abstraktion eines ,,Eingabestroms`` und stellt unter anderem folgende Möglichkeiten zur Verfügung:

```
istream& operator>>(bool& n);
istream& operator>>(int& n);
istream& operator>>(short& n);
istream& operator>>(double& n);
```

Die Operator-Funktion >> ist nichts anderes als die Umwandlung der Textzeichen des Standard-Eingabestroms in die binäre Repräsentation des jeweiligen Objekts.

Neben den Möglichkeiten der ,,formatierten`` Eingabe mit >> stellt *istream* unter anderem folgende Methoden zur ,,unformatierten`` Eingabe zur Verfügung:

```
istream& get(char& c);
istream& getline(char* s,  streamsize n,  char delim=traits::newline());
char& ignore(stream_size n=1, int delim=traits::eof());
int peek();
istream& read(char* s, streamsize n);
stream& putback(char c);
stream& unget();
```

- *get* liest ein einzelnes Zeichen ein, gibt bei EOF 0 zurück.
- *getline* liest eine Folge von maximal n Zeichen ein. *getline* bricht den Einlesevorgang ab, wenn:
    1. das Ende des Eingabestroms erreicht wird,
    2. das Begrenzerzeichen *delim* auftritt oder aber
    3. bereits n-1 Zeichen gespeichert sind.

- *ignore* ,,ignoriert`` n Zeichen beziehungsweise so viele Zeichen, bis das Ende des Eingabestroms oder ein delim-Zeichen gelesen wird. Die Zeichen werden gelesen und ,,weggeworfen``.

- *peek* liefert ein Zeichen, ohne es aus dem Eingabestrom zu entfernen.

- *read* liest maximal n Zeichen des Eingabestroms (beziehungsweise weniger, falls das Eingabeende vorher auftrat).

- *putback* gibt ein beliebiges Zeichen ,,zurück`` in den Eingabestrom. Die nächste Leseoperation liefert dann dieses Zeichen. *unget* führt dieselbe Aktion mit dem zuletzt gelesenen Zeichen durch.

10.4 Formatierte Ein-/Ausgabe

Das Format der Aus- und Eingabe über die Operatoren << und >> kann, wie bereits erwähnt, festgelegt werden. Dazu stellt ios, eine Basisklasse von iostream, verschiedenste Möglichkeiten (linksbündig, rechtsbündig, Hexadezimal, Dezimal, Oktal) zur Verfügung.

Bei der Manipulation gibt es zwei grundsätzliche Möglichkeiten:

- Über das Setzen von sogenannten Formatfelder kann das Format bei der Ein- und Ausgabe festgelegt werden. Dazu wird das entsprechende Flag mit der Methode setf gesetzt. setf weist zwei Parameter auf. Der erste bestimmt den konkreten Wert eines Flags, der zweite, welches Flag verändert wird.

- Daneben existieren spezielle Methoden (Manipulatoren), welche die entsprechenden Flags direkt verändern.

Im folgenden werden die ,,wichtigsten`` Manipulationsmöglichkeiten anhand von Beispielen gezeigt. Für genauere Abhandlungen sei auf das Hilfesystem beziehungsweise die Dokumentation des jeweiligen Entwicklungssystems verwiesen.

`int`-Zahlen können in verschiedenen Formaten ausgegeben werden:

```
int    i = 512;

cout << "\nNormal: " << i;
cout << "\nOktal: " << oct << i;   // oct stellt Ausgabe auf oktal um
cout << "\nOktal: " << i << endl; // kein "oct" noetig, da Einstellung erhalten bleibt
cin  >> oct >> i;                 // Oktale Eingabe
cout << "\nHex:   " << hex << i;  // Hexadezimale Ausgabe

cout.setf(ios::dec, ios::basefield);      // Alternative Möglichkeit: Feld direkt ueber
cout << "\nDezimal:" << i;                // setf setzen
```

Die Ausgabe bei der Eingabe von 17:

```
Normal: 512
Oktal:  1000
Oktal:  1000
Hex:    f
Dezimal:15
```

Auch das Format von Fliesskommazahlen kann festgelegt werden:

```
cout.setf(ios::scientific);
cout << "Scientific: " << 123.456 << endl;
cout.setf(ios::fixed);
cout << "Fixed: " << 123.456 << endl;
```

Die Ausgabe:

```
Scientific: 1.234560e+002
Fixed: 123.456
```

Zudem können für alle Ausgaben Feldlängen festgelegt werden. Die Ausgabe von Objekten erfolgt dann (falls möglich) innerhalb der festgelegten Feldlänge. Mit Methoden wie left, right und internal wird die Ausrichtung der Ausgabe innerhalb des angegebenen Felds bestimmt, mit fill das ,,Füllzeichen``:

```
// Feldlaenge festsetzen
  cout.width(5);
  cout << '(' << "ab" << ')' << endl;              // a)

// Feldlaenge & Fuellzeichen festsetzen
  cout.width(5);
  cout.fill('*');
  cout << '(' << "ab" << ')' << endl;              // b)

// Ausrichtung festsetzen
  cout.width(20);
  cout << "Rechts" << endl;                        // c)
```

Die Ausgabe:

```
    (ab)                 // a)
****(ab)                 // b)
**************Rechts     // c)
```

10.5 Streams und Dateien

Wie bereits beschrieben, sind Ströme Abstraktionen für die zeichenweise Ein-/Ausgabe. Dabei ist es unerheblich, ob die Ausgabe auf den Bildschirm erfolgt oder auf eine Datei, da jede Datei auch als eine sequentielle Folge von Zeichen aufgefasst werden kann. Dateien können also über die ,,normalen`` Möglichkeiten der Ein-/Ausgabe beschrieben werden (`<<, >>, get, write` etc.). Allerdings sind dabei einige ,,Besonderheiten`` zu beachten:

- Dateien müssen geöffnet werden. Das Öffnen einer Datei assoziiert die physikalische Datei mit dem entsprechenden Strom. Existiert die Datei noch nicht, so kann sie beim Öffnen angelegt werden.

- Dateien können zum Lesen (Read Only), Schreiben (Write Only), oder zum Lesen und Schreiben (Read-Write) geöffnet werden.

- Dateien müssen nach ihrer Verarbeitung geschlossen werden. Jede Datei wird automatisch geschlossen, wenn der assoziierte Stream zerstört wird.

- Die Verarbeitung von Dateien kann im Textmodus oder im Binär-Modus erfolgen. Bei der binären Verarbeitung werden Ströme Zeichen für Zeichen ohne jegliche Transformation gelesen beziehungsweise geschrieben. Im Textmodus wird zum Beispiel das Steuerzeichen `endl` in die ,,Plattform-übliche`` Zeilen-Ende-Sequenz umgewandelt (0x0a 0x0d auf DOS, OS/2 etc.).

Der Zugriff auf eine geöffnete Datei schliesslich erfolgt über einen fiktiven ,“Schreib-/Lesekopf“, der die aktuelle Position in der jeweiligen Datei angibt. Diese Position kann über spezielle Methoden verändert werden.

10.5.1 Öffnen von Dateien

Das Öffnen einer Datei erfolgt beim Anlegen eines Stream-Objekts beziehungsweise über die Methode open. Dabei werden der Name der Datei und der Öffnungsmodus als Parameter übergeben. Daneben können weitere, auch implementierungsspezifische Parameter angegeben werden. Dazu benötigt man die Header-Datei <fstream>.

Das folgende Beispiel zeigt eine Verwendung. Es öffnet nacheinander alle als Parameter in der Kommandozeile übergebenen Dateien und gibt deren Inhalt aus (entspricht dem UNIX-Programm cat):

```
// Header-Datei fuer Datei-I/O
#include <fstream>
#include <iostream>
using namespace std;

/* Alle als Parameter uebergebenen Dateien nacheinander oeffnen,
 * ausgeben und schliessen
 */
int main(int argc, char *argv[])
{
      ifstream datei;
      char ch;

      // fuer alle Argumente aus der Kommandozeile
      for(int i=1; i<argc;i++)
      {
            // Datei oeffnen
            datei.open(argv[i]);

            // Inhalt rausschreiben
            while (datei.get(ch) != 0)
                  cout.put(ch);
            // Datei schliessen und Statusbits loeschen
            datei.close();
            datei.clear();
      }
      return 0;
}
```

Die untenstehende Tabelle zeigt die verschiedenen Möglichkeiten, eine Datei zu öffnen. Die Möglichkeiten können (sofern sie sich nicht widersprechen) auch kombiniert werden.

Einige Beispiele:

```
// Default-Modi:
      ifstream readFrom("Eingabe.txt");
      ofstream writeTo("Ausgabe.txt");
      fstream readWrite("EinAus.txt");

// Fuer Eingabe oeffnen
      ifstream readFrom("Eingabe.txt", ios_base::in);

// Binaere Eingabe
      ifstream readFrom("Eingabe.txt", ios_base::in | ios_base::bin);

// Binaere Ausgabe und Datei leeren, falls sie existiert
      ifstream readFrom("Eingabe.txt", ios_base::out | ios_base::bin |
                                       ios_base::trunc);
```

Modus	Kommentar
`in`	Datei für Eingabe öffnen
`out`	Datei für Ausgabe öffnen
`app`	Schreiboperationen am Dateiende ausführen.
`ate`	Nach dem Öffnen der Datei sofort an das Dateiende verzweigen
`trunc`	Zu öffnende Datei zerstören, wenn sie bereits existiert

10.5.2 Lesen und Setzen von Positionen

Dateien müssen nicht streng sequentiell verarbeitet werden. Vielmehr kann die aktuelle Position innerhalb einer Datei über verschiedene Methoden abgefragt und verändert werden. Dazu stehen unter anderem folgende Typen und Methoden zur Verfügung:

- *`streampos`* ist der Typ einer Dateiposition.

- *`seekg(offset, direction)`* zum Beispiel setzt die aktuelle Dateiposition des fiktiven Lesekopfs einer Datei. *`offset`* gibt die Position vom Dateianfang beziehungsweise -ende aus an, `direction` legt fest, von wo aus die Position bestimmt wird: *`ios::beg`* (Dateianfang), *`ios::cur`* aktuelle Position) oder *`ios::end`* (Dateiende).

  `aFile.seekg(-10, ios::end)` setzt die aktuelle Position zehn Bytes vor dem Dateiende und
  `aFile.seekg(10, ios::beg)` zehn Bytes nach dem Dateianfang.
  Der Suffix g in *`seekg`* steht für Get.

- *`tellg`* liefert die aktuelle Position des fiktiven Lesekopfs in der Datei.

- *`seekp`* und *`tellp`* sind die entsprechenden Versionen für die Put-Varianten (in bezug auf Ausgabeströme).

Das folgende Beispiel liest eine Datei und berechnet die Zeilenlänge (bis und mit Newline) und schreibt jeweils zeilenweise die Summe aller Zeichen an das Dateiende.

Die Datei

```
Hallo
das
ist die Test-Eingabedatei.
Zeile vier
und fuenf.
```

wird vom Programm (auf der nächsten Seite) wie folgt verändert:

```
Hallo
das
ist die Test-Eingabedatei.
Zeile vier
und fuenf
6 10 37 48 59 [ 73 ]
```

Die ersten fünf Zahlen (6..59) geben Summe der Zeichen (eine Zahl pro Zeile) in der Datei an, die Zahl in der eckigen Klammern gibt die Gesamtgrösse in Bytes an.

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

int main() {
      int cnt=0;
      char ch;
      fstream inout;
      streampos mark;

      // Falls das File existiert, zum Lesen und Schreiben im Append-Modus oeffnen
      inout.open("test.txt", ios::nocreate|ios::in|ios::app);

      // Ist das Oeffnen eines bestehenden Files gelungen?
      if (!inout.good()) {
            cout << "Fehler: File fehlt!!!!\n";
            exit(1);
      }
      // aktuelle Position = File-Anfang
      inout.seekg(0, ios::beg);

      do {
            inout.get(ch);
            cout.put(ch);
            cnt++;

            // Falls Zeilenende gelesen
            if (ch == '\n') {
                  // aktuelle Position in mark speichern
                  mark = inout.tellg();
                  // Anz. Zeichen am Dateiende anfuegen
                  inout << cnt << ' ';
                  // Wieder zur "alten" Position zurueck
                  inout.seekg(mark);
            }
      } while(!inout.eof());

      // eof trat auf, daher erfolgen keine Ein-/Ausgaben
      // mehr Status muss "geloescht" werden -> clear
      inout.clear();
      // jetzt kann wieder ins File geschrieben werden
      inout << cnt << endl;
      // Endresultat auch auf Bildschirm
      cout << "[ " << cnt << " ]" << endl;
      inout.close();
      return 0;
}
```

Einige Kommentare zum Programm:

- `test.txt` wird als Ein-/Ausgabedatei geöffnet, falls sie existiert . Alle Schreiboperationen erfolgen automatisch am Dateiende (Modus: app).

- Um die Datei von Anfang an zu verarbeiten, wird die Position des fiktiven Lesekopfs auf den Dateianfang gesetzt (*`seekg(0, ios::beg)`*).

- Wenn ein Zeilenumbruch gelesen wurde, wird die aktuelle Position sowie ein Leerzeichen geschrieben. Die aktuelle Position in der Datei wird über die Methode `tellg` ermittelt. Um die Datei nach den Schreiboperationen (die ja am Dateiende erfolgen) weiter zu verarbeiten, wird die Position anschliessend wieder hergestellt.

- Die `while`-Schleife wird verlassen, wenn kein Zeichen mehr gelesen werden kann. Da auftretende Fehler und das Überlesen des letzten Zeichens einer Datei dazu führen, dass keine weiteren Lese- und Schreiboperationen mehr stattfinden, werden die Status-Flags durch `clear` wieder gelöscht.

11 C-Bibliotheken

11.1 Übersicht der ANSI-C Standard Libraries

Die Sprache C verfügt über **keine** 'eingebauten' Funktionen, insbesondere auch nicht für Input/Output (sie soll ja klein sein). Kompensiert wird dieser 'Mangel' durch die Lieferung verschiedener Libraries zusammen mit dem Compiler.

Der ANSI-Standard beschreibt 15 Libraries (genauer : Header-Files). Es sind dies :

`<assert.h>`	ermöglicht das Prüfen von Zusicherungen zur Laufzeit,
`<ctype.h>`	Zeichenmanipulation und -Klassifikation,
`<errno.h>`	Fehlerbehandlung,
`<locale.h>`	Land- und gebietsabhängige Funktionen,
`<math.h>`	mathematische Funktionen,
`<setjmp.h>`	Sprünge über Funktionsgrenzen,
`<signal.h>`	asynchrone Signalisierung,
`<stdarg.h>`	Makros für die Verarbeitung von Funktionen mit einer nicht spezifizierten Anzahl von Parametern,
`<stddef.h>`	Offset eines Feldes innerhalb einer struct,
`<stdio.h>`	Standard Input/Output auf Streams,
`<stdlib.h>`	diverse Funktionen, z.B. Sortierfunktion, dynamische Speicherverwaltung, Binary Search, ...
`<string.h>`	Stringmanipulationen,
`<time.h>`	diverse Zeitumrechnungsfunktionen.
`<float.h>`	definiert Konstanten für den Wertumfang der Gleitpunktarithmetik
`<limits.h>`	definiert Konstanten für den Wertumfang der ganzzahligen Typen

Daneben wird in der Regel mit jedem Compiler noch eine ganze Reihe weiterer Libraries verfügbar gemacht (Schnittstelle zum Betriebssystem, grafische Funktionen, Kommunikation etc.).

Bleibt noch anzufügen, dass, nicht zuletzt aus Effizienzgründen, eine ganze Reihe von Funktionen gar nicht als Funktionen, sondern als Makros implementiert sind. Dies ist für die Benutzerin und den Benutzer eigentlich unwichtig, solange semantisch kein Unterschied besteht. Probleme gibt es aber allenfalls wieder infolge von Nebeneffekten im Funktions- bzw. Makroaufruf.

Beispiel :

Es existiert die Funktion

```
int square(int x)
{
      return x * x;
}
```

Ruft man nun diese Funktion auf mit

```
a = 3;
b = square(a++);
```

so hat danach a den Wert 4 und b den Wert 9, wie wir dies von der Funktion erwarten dürfen. Ist aber die Funktion als Makro

```
#define SQUARE(x)       ((x) * (x))
```

implementiert, so wird der Aufruf

```
a = 3;
b = SQUARE(a++);
```

vom Präprozessor zu

```
a = 3;
b = ((a++) * (a++));
```

expandiert, was schliesslich für a den Wert 5 und für b den Wert 12 ergibt! (Weshalb 12 und nicht 9 oder 16 ?).

Der Unterschied zwischen Funktionsaufruf und Makroaufruf besteht darin, dass im ersten Fall das Argument **einmal** evaluiert und dann an die Funktion übergeben wird, womit auch der Nebeneffekt des Inkrementierens von a nur einmal passiert, wohingegen der Ausdruck a++ beim Makroaufruf **zweimal** berechnet wird und damit der Nebeneffekt zweimal erzeugt wird.

Abhilfe kann nur die Regel schaffen, dass Makros unter keinen Umständen mit Ausdrücken aufgerufen werden dürfen, welche Nebeneffekte verursachen.

Die schon erwähnte Abhängigkeit vom umgebenden Betriebssystem führt dazu, dass die in diesem Kapitel definierten Funktionen in einer anderen Umgebung u.U. etwas anders aussehen. In jedem Fall sollte das Benützerhandbuch genau konsultiert werden.

Folgende neue Datentypen und Konstante werden in den Libraries verwendet:

- `size_t` wird als Rückgabewert vom Operator `sizeof` gebraucht, also für die Anzahl Bytes eines Objektes. `size_t` ist definiert als `unsigned int`.
- `void *` wird als generischer Pointertyp verwendet. Jeder Pointer kann ohne Informationsverlust in diesen Typ verwandelt und wieder zurückverwandelt werden.
- `NULL` wird zum Nullsetzen eines Pointers verwendet. Es entspricht dem Nil in Pascal. Die Definition ist: `#define NULL (void *) 0.`
- `FILE *` ist der Datentyp für einen Filepointer (Filedeskriptor). Ein Filepointer zeigt somit auf ein geöffnetes File.

11.2 Ein- und Ausgabe: <stdio.h>

Die Ein- und Ausgabefunktionen, Typen und Makros, die in `<stdio.h>` vereinbart sind, machen nahezu einen Drittel der Bibliothek aus.

Ein Datenstrom (stream) ist Quelle oder Ziel von Daten und wird mit einem Peripheriegerät verknüpft. Die Bibliothek unterstützt zwei Arten von Datenströmen, für Text und binäre Information, die allerdings bei manchen Systemen und insbesondere bei UNIX identisch sind. Ein Textstrom ist eine Folge von Zeilen; jede Zeile enthält null oder mehr Zeichen und ist mit `'\n'` abgeschlossen. Eine Umgebung muss möglicherweise zwischen einem Textstrom und einer anderen Repräsentierung umwandeln (also zum Beispiel `'\n'` als Wagenrücklauf und Zeilenvorschub abbilden). Ein Binärstrom ist eine Folge unbearbeiteter Bytes zur Aufzeichnung interner Daten. Wird ein Binärstrom geschrieben und auf dem gleichen System wieder eingelesen, so entsteht die gleiche Information.

Ein Strom wird durch Eröffnen (open) mit einem File oder einem Gerät verbunden; die Verbindung wird durch Abschliessen (close) wieder aufgehoben. Eröffnet man ein File, so erhält man einen Pointer auf ein Objekt vom Typ `FILE`, wo alle Information hinterlegt ist, die zur Kontrolle des Stroms nötig ist. Wenn die Bedeutung eindeutig ist, werden wir die Begriffe Filepointer, Filedeskriptor und Datenstrom gleichberechtigt verwenden.

Wenn die Ausführung eines Programms beginnt, sind die drei Ströme `stdin`, `stdout` und `stderr` bereits eröffnet.

11.2.1 Formatierte Ausgabe

Als Beispiel für eine der höhere Ausgabefunktionen soll `printf` erläutert werden, die es erlaubt, Daten in formatierter Darstellung auf die Standardausgabe zu schreiben.
Ein Aufruf von `printf` hat die Form:

```
int printf(const char *format, p1, p2, p3...)
```

format:	Zeichenkette mit Formatanweisungen
p1,p2,p3...:	beliebig viele Parameter.

Der Resultatwert ist die Anzahl der geschriebenen Zeichen; er ist negativ, wenn ein Fehler passiert ist. Die wesentliche Formatsteuerung ist in der ersten Zeichenkette enthalten; sie besteht aus beliebigen Zeichen, die direkt ausgegeben werden, und den Steueranweisungen, die mit dem Zeichen '%' beginnen.

Diese Steueranweisungen sind von der Form

%-<number>.<precision><format>

die Zeichen '-', <number> und .<precision> sind optional, d.h. sie können, müssen aber nicht angegeben werden.

'-'	gibt an, dass linksbündig ausgegeben werden soll.
<number>	ist eine dezimale Zeichenfolge. Sie gibt die minimale Spaltenzahl an. Sollten für die Darstellung des auszugebenden Objekts mehr Zeichen als <number> benötigt werden, so wird die grössere Zahl von Zeichen ausgegeben.
<precision>	gibt bei Gleitpunktdarstellung die Zahl der Stellen hinter dem Komma an.
<format>	gibt die Darstellungsform an.

Folgende Möglichkeiten stehen zur Auswahl:

'd', 'i'	Ausgabe in Dezimalnotation; Vorzeichen nur bei negativen Werten
'o'	Oktalnotation
'x'	Ausgabe in Hexadezimaldarstellung ohne führendes '0x'
'u'	Dezimaldarstellung ohne Vorzeichen
'c'	einzelnes Zeichen (ASCII)
's'	Zeichenkette (Achtung: Adresse übergeben)
'e'	double Notation der Form [-]m.nnnnnnE[+-]xx
'f'	double Notation der Form [-]mmm.nnnnn
'g'	benutzt %e bzw. %f, je nachdem, welche der beiden kürzer ist

Die Verarbeitung geht so vor sich, dass die Steuerzeichen von links her nach %-Formaten durchsucht werden. Diese werden der Reihe nach auf die Parameter p1, p2 usw. von printf angewandt. Es ist darauf zu achten, dass gleichviel %-Formate wie Parameter übergeben werden, da sonst unvorhersehbare Resultate entstehen können.

Beispiel:

```
int summe, zahl1 = 123, zahl2 = 24;

summe = zahl1 + zahl2;
printf("Die Summe aus zahl1 und zahl2 = %d,\tzahl1 ist %d\n",
summe, zahl1);
```

Ausdruck : Die Summe aus zahl1 und zahl2 = 147, zahl1 ist 123

Die erste Formatanweisung %d bezieht sich auf die Variable `summe`; die zweite bezieht sich auf die Variable `zahl1`.

Soll eine `short int` ausgegeben werden, muss ein h dem d vorangestellt werden; soll eine `long int`

ausgegeben werden, muss ein `l` dem d vorangestellt werden.

Die Funktion

```
int sprintf(char *s, const char *format, ...)
```

funktioniert wie `printf`, nur wird die Ausgabe in den Zeichenvektor `s` geschrieben und mit `` `\0` `` abgeschlossen. `s` muss gross genug für das Resultat sein. Im Resultatwert wird `` `\0` `` nicht mitgezählt.

11.2.2 Formatierte Eingabe

Für die Eingabe ist die Funktion `scanf` das Gegenstück zu `printf.` `scanf` wird praktisch gleich benutzt, ausser dass alle Argumente (p1,p2,p3..) **Pointer bzw. Adressen** sein müssen. `scanf` ignoriert Leerzeichen und Tabulatoren in der Format-Zeichenkette. Ausserdem werden in der Eingabe Leerzeichen, Tabulatoren und Zeilentrenner überlesen, wenn nach Eingabewerten gesucht wird.

Ein Aufruf von `scanf` hat die Form:

```
int scanf(const char *format, p1, p2, p3...)
```

`scanf` liefert `EOF`, wenn vor der ersten Umwandlung das Fileende erreicht wird; andernfalls liefert die Funktion die Anzahl der umgewandelten und abgelegten Eingaben. Kann die erste Eingabe nicht gelesen werden, wird 0 zurückgegeben.
Folgende Möglichkeiten stehen zur Wahl :

'd' dezimal integer, `int *`
'i' integer; `int *`. Darstellung oktal (führende 0) oder hexadezimal (führende 0x oder 0X)
'o' oktal integer (mit oder ohne führende 0), `int *`
'u' `unsigned` dezimal `int, unsigned int *`
'x' `int` in Hexadezimaldarstellung ohne führende '0x', `int *`
'c' einzelnes Zeichen (ASCII), `char *`. Das nächste Eingabezeichen wird eingesetzt. Soll das nächste sichtbare Zeichen gelesen werden, benutzen Sie `%1s`
's' Zeichenkette bis **Trennzeichen**, `char *` zeigt auf eine Zeichenkette
'e' `float *`, Notation der Form [-]m.nnnnnnE[+-]xx
'f' `float *`, Notation der Form [-]mmm.nnnnn
'g' benutzt %e bzw. `%f`, je nachdem, welche der beiden kürzer ist

Als **Trennzeichen** gelten: `\t,\n,\r,\v,\f,' '` (Leerzeichen). Den Umwandlungszeichen `d,i,` `o,` u und x muss ein h vorausgehen, wenn das Argument vom Typ `short int` ist, bzw. ein `l`, wenn das Argument vom Typ `long int` ist. Um `double` einzulesen ist auch ein vorangestelltes `l` erforderlich.

Beispiel, um eine Summe mit einzulesenden `double` Zahlen zu berechnen :

```
double zahl, sum = 0;

while(scanf("%lf", &zahl) == 1)
      sum = sum + zahl;
printf("\t %.2f\n", sum);
```

Beispiel, um ein Datum der Form '8 Jan 90' einzulesen :

```
int tag, jahr;
char monat[10];

scanf("%d %s %d", &tag, monat, &jahr);
```

Im weiteren ist

```
int sscanf(char *s, const char *format, ...)
```

äquivalent zu `scanf(...)`, mit dem Unterschied, dass die Eingabezeichen aus der Zeichenkette `s` stammen.

11.2.3 Ein- und Ausgabe von Zeichen

Standardmässig ist für C-Programme die Tastatur Eingabe und der Bildschirm Ausgabe.

Die Funktion

```
int getchar(void)
```

liest vom Standardeingabe-Medium und gibt ein Zeichen bzw. einen `unsigned char` in `int` umgewandelt zurück. Dabei sei darauf hingewiesen, dass bei Fileende (`EOF`) ein spezielles Zeichen (meist -1) zurückgegeben wird, um vom Programm aus eine Endbehandlung durchführen zu können. Welches Zeichen das ist, bleibt der Implementierung überlassen.

Die Funktion

```
int putchar(int c)
```

schreibt das Zeichen `c` auf das Ausgabemedium. Sie gibt das geschriebene Zeichen zurück oder `EOF` für Fehler.

Beispiel:

Das untenstehende Programm kopiert vom Standard-Eingabemedium auf das Standard-Ausgabemedium.

```
int main( )
{
      int c;
      while((c = getchar()) != EOF)
            putchar(c);
}
```

Weitere Funktionen sind :

```
char *gets(char *s)
```

liest die nächste Zeile von `stdin` in den Vektor `s` und ersetzt dabei den abschliessenden Zeilentrenner durch `` `\0` ``. Die Funktion liefert `s` oder `NULL` bei Fileende oder bei Fehler.

```
int puts(const char *s)
```

`puts` schreibt die Zeichenkette `s` und einen Zeilentrenner in stdout. Die Funktion liefert `EOF`, wenn ein Fehler passiert, andernfalls einen nicht-negativen Wert.

11.2.4 Fileoperationen

Die folgenden Funktionen beschäftigen sich mit Fileoperationen. Nicht alle Programme kommen mit den Standard-Ein/Ausgabemedien aus; sei es, weil mehrere Ein- bzw. Ausgabefiles benötigt werden, sei es, weil das File erst während des Programmlaufs bestimmt wird. Für diesen Fall müssen die Files explizit eröffnet werden.

Dazu dient die Funktion:

```
FILE *fopen(const char *filename, const char *mode)
```

die als Ergebnis einen Filedeskriptor (Datenstream) oder `NULL` bei Misserfolg zurückgibt. Die Parameter

'filename' und 'mode' haben folgende Bedeutung:

filename gibt den Namen des Files an; dabei ist filename ein Pointer auf einen String.

mode gibt an, ob das File als Ein- oder als Ausgabefile verwendet werden soll.

"r"	Textfile zum Lesen eröffnen
"w"	Textfile zum Schreiben erzeugen; gegebenenfalls alten Inhalt wegwerfen
"a"	anfügen; Textfile zum Schreiben am Fileende eröffnen oder erzeugen
"r+"	Textfile zum Ändern eröffnen (Lesen und Schreiben)
"w+"	Textfile zum Ändern erzeugen; gegebenenfalls alten Inhalt wegwerfen
"a+"	anfügen; Textfile zum Ändern eröffnen oder erzeugen, Schreiben am Ende

Ändern bedeutet, dass im gleichen File gelesen und geschrieben werden darf; fflush oder eine Funktion zum Positionieren in Files muss zwischen einer Lese- und einer Schreiboperation oder umgekehrt aufgerufen werden. Enthält mode nach dem ersten Zeichen noch b, also etwa "rb" oder "w+b", dann wird auf ein binäres File zugegriffen.

Beispiel:

```
FILE *filepoi;
filepoi = fopen("mydat.txt", "r");
...
```

Das File `mydat.txt` wird zum Lesen `"r"` eröffnet; der Variablen `filepoi` wird ein Pointer zugewiesen, über den das File im folgenden Ablauf angesprochen wird.

Die Funktionen zum expliziten Lesen und Schreiben verwenden den Filedeskriptor, der von `fopen` zurückgegeben wird. Es sind dies unter anderen die Funktionen `int fgetc(FILE *filep)` zum Lesen eines Zeichens und `int fputc(int c, FILE *filep)` zum Schreiben eines Zeichens c in ein File.

Zum Schliessen von Files dient die Funktion

```
int fclose( FILE *filepoi)
```

wobei `filepoi` wieder der durch `fopen` definierte Filedeskriptor ist. Genaugenommen schreibt `fclose` noch nicht geschriebene Daten für `stream`, wirft noch nicht gelesene, gepufferte Eingaben weg, gibt automatisch angelegte Puffer frei und schliesst den Datenstrom. Die Funktion liefert Null oder bei Fehler `EOF`.

Beispiel: Ein File wird geöffnet, zeilenweise gelesen und auf den Bildschirm geschrieben, also inklusiv Sonderzeichen und Return's. Folgende Funktionen werden verwendet:
gets, fgets(), fopen(), fclose(), fscanf(), fprintf()

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 100

int main (void)
{
      char filename[MAX], line[MAX];
      FILE *filepointer;

      printf("\nInput File Name eintippen :\t");
      gets(filename);

      if ((filepointer = fopen(&filename[0], "r")) == NULL)
      {
            fprintf(stderr, "\n\tcan't open input file %s\n\n", filename);
            exit(1);
      }

      while (fgets(line, MAX, filepointer) != NULL)
            fprintf (stdout, "%s", line);

/* Folgende Anweisung kopiert WORT für WORT auf den Bildschirm. Die isspace()
   Zeichen wie '\n', '\t' werden also ausgefiltert.

      while (fscanf(filepointer, "%s", line) != EOF)
            fprintf (stdout, "%s ", line);
*/

      fclose(filepointer);

      return 0;

} /* main */
```

Weiter Funktionen sind :

```
int fprintf(FILE *stream, const char *format, ...)
```

ist äquivalent zu `printf(...);` es muss aber noch der Ausgabestream angegeben werden. Mit `fprintf()` ist es somit auch möglich, in ein File zu schreiben.

```
int fscanf(FILE *stream, const char *format, ...)
```

ist äquivalent zu `scanf(...)`, nur muss noch der Eingabestream angegeben werden.

```
char *fgets(char *s, int n, FILE *stream)
```

`fgets` liest höchstens die nächsten n-1 Zeichen in `s` ein und hört vorher auf, wenn ein Zeilentrenner gefunden wird. Der Zeilentrenner wird im Vektor abgelegt. Der Vektor wird mit `` `\0` `` abgeschlossen. `fgets` liefert s. Bei Fileende oder bei einem Fehler wird `NULL` zurückgegeben..

```
int fputs(const char *s, FILE *stream)
```

`fputs` schreibt die Zeichenkette `s` (die `'\n'` nicht zu enthalten braucht) in `stream`. Die Funktion liefert einen nicht-negativen Wert oder `EOF` bei einem Fehler.

```
int fgetc(FILE *stream)
```

`fgetc` liefert das nächste Zeichen aus `stream` als `unsigned char` (umgewandelt in `int`) oder `EOF` bei Fileende oder bei einem Fehler.

```
int getc(FILE *stream)
```

`getc` ist äquivalent zu `fgetc`, kann aber ein Makro sein und dann das Argument für `stream` mehr als einmal bewerten.

```
int fputc(int c, FILE *stream)
```

`fputc` schreibt das Zeichen c (umgewandelt in `unsigned char`) in `stream`. Die Funktion liefert das ausgegebene Zeichen oder `EOF` bei Fehler.

```
int putc(int c, FILE *stream)
```

`putc` ist äquivalent zu `fputc`, kann aber ein Makro sein und dann das Argument für `stream` mehr als einmal bewerten.

```
int ungetc(int c, FILE *stream)
```

`ungetc` stellt c (umgewandelt in `unsigned char`) in `stream` zurück, von wo das Zeichen beim nächsten Lesevorgang wieder geholt wird. Man kann sich nur darauf verlassen, dass pro Datenstrom ein Zeichen zurückgestellt werden kann. `EOF` darf nicht zurückgestellt werden. `ungetc` liefert das zurückgestellte Zeichen oder `EOF` bei einem Fehler.

```
FILE *freopen(const char *filename, const char *mode, FILE *stream)
```

`freopen` eröffnet das File für den angegebenen Zugriff mode und verknüpft `stream` damit. Das Resultat ist `stream` oder `NULL` bei einem Fehler. Mit freopen ändert man normalerweise die Files, die mit `stdin`, `stdout` oder `stderr` verknüpft sind.

```
int fflush(FILE *stream)
```

Bei einem Ausgabestrom sorgt `fflush` dafür, dass gepufferte, aber noch nicht geschriebene Daten geschrieben werden; bei einem Eingabestrom ist der Effekt undefiniert. Die Funktion liefert `EOF` bei einem Schreibfehler und sonst Null. `fflush(NULL)` bezieht sich auf alle offenen Files.

```
int remove(const char *filename)
```

`remove` entfernt das angegebene File, so dass ein anschliessender Versuch, sie zu eröffnen, fehlschlagen wird. Die Funktion liefert bei Fehlern einen von Null verschiedenen Wert.

```
int rename(const char *oldname, const char *newname)
```

`rename` ändert den Namen eines Files und liefert nicht Null, wenn der Versuch fehlschlägt.

```
FILE *tmpfile(void)
```

`tmpfile` erzeugt ein temporäres File mit Zugriff "wb+", das automatisch gelöscht wird, wenn der Zugriff abgeschlossen wird, oder wenn das Programm normal zu Ende geht. `tmpfile` liefert einen Datenstrom oder `NULL`, wenn das File nicht erzeugt werden konnte.

```
char *tmpnam(char s[L_tmpnam])
```

`tmpnam(NULL)` erzeugt eine Zeichenkette, die nicht der Name eines existenten Files ist, und liefert einen Pointer auf einen internen Vektor im statischen Speicherbereich. `tmpnam(s)` speichert die Zeichenkette in `s` und liefert auch `s` als Resultat; in `s` müssen wenigstens `L_tmpnam` Zeichen abgelegt werden können. tmpnam erzeugt bei jedem Aufruf einen anderen Namen; man kann höchstens von `TMP_MAX` verschiedenen Namen während der Ausführung des Programms ausgehen. Zu beachten ist, dass `tmpnam` einen Namen und keine Files erzeugt.

```
int setvbuf(FILE *stream, char *buf, int mode, size_t size)
```

`setvbuf` kontrolliert die Pufferung bei einem Datenstrom; die Funktion muss vor allen anderen Operationen aufgerufen werden und bevor gelesen oder geschrieben wird. Hat `mode` den Wert `_IOFBF`, so wird vollständig gepuffert, `_IOLBF` sorgt für zeilenweise Pufferung bei Textfiles, und `_IONBF` verhindert Puffern. Wenn `buf` nicht `NULL` ist, wird `buf` als Puffer verwendet; andernfalls wird ein Puffer angelegt. `size` legt die Puffergrösse fest. Bei einem Fehler liefert `setvbuf` nicht Null.

```
void setbuf(FILE *stream, char *buf)
```

Wenn `buf` den Wert `NULL` hat, wird der Datenstrom nicht gepuffert. Andernfalls ist `setbuf` äquivalent zu `(void) setvbuf(stream, buf, _IOFBF, BUFSIZ)`.

11.2.5 Direkte Ein- und Ausgabe

```
size_t fread(void *ptr, size_t size, size_t nobj, FILE *stream)
```

`fread` liest aus `stream` in den Vektor `ptr` höchstens `nobj` Objekte der Grösse `size` ein. `fread` liefert die Anzahl der eingelesenen Objekte; das kann weniger als die geforderte Zahl sein. Der Zustand des Datenstroms muss mit `feof` und ferror untersucht werden.

```
size_t fwrite(const void *ptr, size_t size, size_t nobj, FILE *stream)
```

`fwrite` schreibt `nobj` Objekte der Grösse `size` aus dem Vektor `ptr` in `stream`. Die Funktion liefert die Anzahl der ausgegebenen Objekte; bei Fehler ist das weniger als `nobj`.

11.2.6 Positionieren in Files

```
int fseek(FILE *stream, long offset, int origin)
```

`fseek` setzt die Fileposition für `stream` eine nachfolgende Lese- oder Schreiboperation wird auf Daten von der neuen Position an zugreifen. Für ein binäres File wird die Position auf `offset` Zeichen relativ zu `origin` eingestellt; dabei können die Werte `SEEK_SET` (Fileanfang), `SEEK_CUR` (aktuelle Position) oder `SEEK_END` (Fileende) angegeben werden. Für einen Textstrom muss offset Null sein oder ein Wert, der von `ftell` stammt (dafür muss dann `origin` den Wert `SEEK_SET` erhalten). `fseek` liefert einen von Null verschiedenen Wert bei Fehler.

```
long ftell(FILE *stream)
```

`ftell` liefert die aktuelle Fileposition für `stream` oder -1L bei Fehler.

```
void rewind(FILE *stream)
```

`rewind(fp)` ist äquivalent zu `fseek(fp, 0L, SEEK_SET); clearerr(fp)`.

Beispiel:

```
#include <stdio.h>          /*  Hier wird das Lesen und Schreiben in ein 'File of records' gezeigt. Es  */
#include <string.h>         /*  werden die Funktionen, fseek(), rewind(), ftell(), tempfile(), fread() */
#include <errno.h>          /*  und fwrite verwendet. */

struct RECORD   {
                    short   counter;
                    int     integer;
                    double  realnum;
                };

int main(void)
{
      struct RECORD filerec = { 0, 1, 10000000.0};    /* Define and init. the structure filerec */
      int i, newrec;
      size_t recsize = sizeof( filerec );           /* Size of the structure filerec */
      FILE *recstream;                          /* Filepointer */
      long int recseek, remember;                   /* Fileoffset */

      recstream = tmpfile();                        /* Create and open temporary File. */

      for( i = 0; i < 10; i++ )                     /* Write 10 unique records to File. */
      {
            ++filerec.counter;
            filerec.integer = filerec.integer * 3;
            filerec.realnum = filerec.realnum / (i + 1);
            fwrite( &filerec, recsize, 1, recstream );
      }

      do                                            /* Find a specified record. */
      {     printf( "Enter record betweeen 1 and 10 (or 0 to quit): " );
            if((scanf( "%d", &newrec) && newrec && getchar() == '\n') != 0)
            {
                  recseek = (long)((newrec - 1) * recsize);
                  fseek( recstream, recseek, SEEK_SET );
                  fread( &filerec, recsize, 1, recstream );

                  if(( feof( recstream)  ||  ferror( recstream)) == 0)
                  {
                        printf( "Counter:\t%hd\n", filerec.counter );
                        printf( "Integer:\t%d\n", filerec.integer );
                        printf( "Real number:\t%.2f\n\n", filerec.realnum );
                  }
                  else
                        printf("\nAn error occured by reading the File !!\n\n");
            }
            else
                  while( getchar( ) != '\n') ;      /* To emty the inputbuffer */
      } while( newrec != 0);

      remember = ftell( recstream );                /* To memorize the actual File position */

/* Starting at first record, scan each for specific value. The following line is equivalent to:
  fseek( recstream, 0L, SEEK_SET ); */

      rewind( recstream );

      do      /* Search now the first integer above 1000 in the File of records */
      {     fread( &filerec, recsize, 1, recstream );
      } while( filerec.integer < 1000 );

      recseek = ftell(recstream);               /* To memorize the actual File position */
      printf("\nFirst integer above 1000 is %d in record %ld\n", filerec.integer,
            recseek / recsize);
      fseek(recstream, remember-recsize, SEEK_SET);   /* Go back to the memorized pos. */
      fread(&filerec, recsize, 1, recstream);
      printf("\nRememberd Record Integer: %d\t Counter: %hd\n", filerec.integer,
            filerec.counter);

     return 0;
}

int fgetpos(FILE *stream, fpos_t *ptr)
```

`fgetpos` speichert die aktuelle Position für `stream` bei `*ptr`. Der Wert kann später mit `fsetpos` verwendet werden. Der Datentyp `fpos_t` eignet sich zum Speichern von solchen Werten. Bei einem Fehler liefert `fgetpos` einen von Null verschiedenen Wert.

```
int fsetpos(FILE *stream, const fpos_t *ptr)
```

`fsetpos` positioniert `stream` auf die Position, die von `fgetpos` in `*ptr` abgelegt wurde. Bei einem Fehler liefert `fsetpos` einen von Null verschiedenen Wert.

11.2.7 Fehlerbehandlung

Viele der Bibliotheksfunktionen notieren Zustandsangaben, wenn ein Fileende oder ein Fehler gefunden wird. Diese Angaben können explizit gesetzt und getestet werden. Ausserdem kann der Integer-Ausdruck `errno` (der in `<errno.h>` deklariert ist) eine Fehlernummer enthalten, die mehr Informationen über den zuletzt aufgetretenen Fehler liefert.

```
void clearerr(FILE *stream)
```

`clearerr` löscht die Fileende- und Fehlernotizen für `stream`.

```
int feof(FILE *stream)
```

`feof` liefert einen von Null verschiedenen Wert, wenn für `stream` ein Fileende notiert ist.

```
int ferror(FILE *stream)
```

`ferror` liefert einen von Null verschiedenen Wert, wenn für `stream` ein Fehler notiert ist.

```
void perror(const char *s)
```

`perror(s)` gibt `s` und eine von der Implementierung definierte Fehlermeldung aus, die sich auf die Fehlernummer in `errno` bezieht. Die Ausgabe erfolgt im Stil von `fprintf (stderr, "%s: %s\n", s, "Fehlermeldung")`.

11.3 Tests für Zeichenklassen: <ctype.h>

Das Definitionsfile <ctype.h> vereinbart Funktionen zum Testen von Zeichen. Jede Funktion hat ein `int`-Argument, dessen Wert entweder `EOF` ist oder als `unsigned char` dargestellt werden kann, und der Resultatwert hat den Typ `int`. Die Funktionen liefern einen von Null verschiedenen Wert (wahr), wenn das Argument `c` die beschriebene Bedingung erfüllt; andernfalls liefern sie Null.

`isalnum(c)`	`isalpha(c)` oder `isdigit(c)` ist wahr
`isalpha(c)`	`isupper(c)` oder `islower(c)` ist wahr
`iscntrl(c)`	Steuerzeichen
`isdigit(c)`	dezimale Ziffer
`isgraph(c)`	sichtbares Zeichen, kein Leerzeichen
`islower(c)`	Kleinbuchstabe (aber kein Umlaut)
`isprint(c)`	sichtbares Zeichen, auch Leerzeichen
`ispunct(c)`	sichtbares Zeichen, mit Ausnahme von Leerzeichen, Buchstabe oder Ziffer
`isspace(c)`	Leerzeichen, Seitenvorschub `\f`, Zeilentrenner `\n`, Wagenrücklauf `\r`, Tabulatorzeichen `\t`, Vertikal-Tabulator `\v`
`isupper(c)`	Grossbuchstabe (aber kein Umlaut)
`isxdigit(c)`	hexadezimale Ziffer

Im 7-Bit ASCII-Zeichensatz sind die sichtbaren Zeichen 0x20 (' ') bis 0x7E ('~'); die Steuerzeichen sind 0 (NUL) bis 0x1F (US) sowie 0x7F (DEL).

Zusätzlich gibt es zwei Funktionen zur Umwandlung zwischen Gross- und Kleinbuchstaben:

`int tolower(int c)` wandelt `c` in einen Kleinbuchstaben um
`int toupper(int c)` wandelt `c` in einen Grossbuchstaben um

Wenn `c` ein Grossbuchstabe ist, liefert `tolower(c)` den entsprechenden Kleinbuchstaben; andernfalls ist das Resultat `c`. Wenn `c` ein Kleinbuchstabe ist, liefert `toupper (c)` den entsprechenden Grossbuchstaben; andernfalls ist das Resultat `c`.

11.4 Funktionen für Zeichenketten: <string.h>

In dem Definitionsfile <string.h> werden zwei Gruppen von Funktionen für Zeichenketten vereinbart. Die erste Gruppe hat Namen, die mit str beginnen; die Namen der zweiten Gruppe beginnen mit mem. Sieht man von memmove ab, ist der Effekt der Funktionen undefiniert, wenn zwischen überlappenden Objekten kopiert wird.

In der folgenden Tabelle sind die Variablen s und t vom Typ char *; die Parameter cs und ct haben den Typ const char *; der Parameter n hat den Typ size_t; und c ist ein int-Wert, der in char umgewandelt wird. Die Vergleichsfunktionen behandeln ihre Argumente als unsigned char Vektoren.

char *strcpy(s,ct)	Zeichenkette ct in Vektor s kopieren, inklusive `\0`; liefert s.
char *strncpy(s,ct,n)	höchstens n Zeichen aus ct in s kopieren; liefert s. Mit `\0` auffüllen, wenn ct weniger als n Zeichen hat.
char *strcat(s,ct)	Zeichenkette ct hinten an die Zeichenkette s anfügen; liefert s.
char *strncat(s,ct,n)	höchstens n Zeichen von ct hinten an die Zeichenkette s anfügen und s mit `\0` abschliessen; liefert s.
int strcmp(cs,ct)	Zeichenketten cs und ct vergleichen; liefert <0 wenn cs<ct, 0 wenn cs == ct, oder > 0 wenn cs > ct.
int strncmp(cs,ct,n)	höchstens n Zeichen von cs mit der Zeichenkette ct vergleichen; liefert < 0 wenn cs < ct, 0 wenn cs== ct, oder >0 wenn cs>ct.
char *strchr(cs,c)	liefert Pointer auf das erste c in cs oder NULL, falls nicht vorhanden.
char *strrchr(cs,c)	liefert Pointer auf das letzte c in cs, oder NULL, falls nicht vorhanden.
size_t strspn(cs,ct)	liefert Anzahl der Zeichen am Anfang von cs, die sämtlich in ct vorkommen.
size_t strcspn(cs,ct)	liefert Anzahl der Zeichen am Anfang von cs, die sämtlich nicht in ct vorkommen.
char *strpbrk(cs,ct)	liefert Pointer auf die Position in cs, in der irgendein Zeichen aus ct erstmals vorkommt, oder NULL, falls keines vorkommt.
char *strstr(cs,ct)	liefert Pointer auf erste Kopie von ct in cs oder NULL, falls nicht vorhanden.
size_t strlen(cs)	liefert Länge von cs (ohne `\0`).
char *strerror(n)	liefert Pointer auf Zeichenkette, die in der Implementierung für Fehler n definiert ist.
char *strtok(s,ct)	strtok durchsucht s nach Zeichenfolgen, die durch Zeichen aus ct begrenzt sind; siehe unten.

Eine Folge von Aufrufen von strtok(s,ct) zerlegt s in Zeichenfolgen, die jeweils durch ein Zeichen aus ct begrenzt sind. Beim ersten von einer Reihe von Aufrufen ist s nicht NULL. Dieser Aufruf findet die erste Zeichenfolge in s, die nicht aus Zeichen in ct besteht; die Folge wird dadurch abgeschlossen, dass das nächste Zeichen in s mit `\0` überschrieben wird; der Aufruf liefert dann einen Pointer auf die Zeichenfolge. Jeder weitere Aufruf der Reihe ist kenntlich dadurch, dass NULL für s übergeben wird; er liefert die nächste derartige Zeichenfolge, wobei unmittelbar nach dem Ende der vorhergehenden mit der Suche begonnen wird. strtok liefert NULL, wenn keine weitere Zeichenfolge gefunden wird. Die Zeichenkette ct kann bei jedem Aufruf verschieden sein.

Die mem... Funktionen sind zur Manipulation von Objekten als Zeichenvektoren gedacht; sie sollen eine Schnittstelle zu effizienten Routinen sein. In der folgenden Tabelle haben s und t den Typ void *; die Argumente cs und ct sind vom Typ const void *; das Argument n hat den Typ size_t; und c ist ein int-Wert, der in unsigned char umgewandelt wird.

void *memcpy(s,ct,n)	kopiert n Zeichen von ct nach s; liefert s.
void *memmove(s,ct,n)	wie memcpy, funktioniert aber auch, wenn die Objekte überlappen.
int memcmp(cs,ct,n)	vergleicht die ersten n Zeichen von cs mit ct; Resultat wie strcmp.
void *memchr(cs,c,n)	liefert Pointer auf das erste c in cs oder NULL, wenn das Zeichen in den ersten n Zeichen nicht vorkommt.
void *memset(s,c,n)	setzt die ersten n Zeichen von s auf den Wert c, liefert s.

11.5 Mathematische Funktionen: <math.h>

Die Definitionsfile <math.h> vereinbart mathematische Funktionen und Makros.

Die Makros EDOM und ERANGE, die man in <errno.h> findet, sind von Null verschiedene ganzzahlige Konstanten, mit denen Fehler im Argument- und Resultatbereich der Funktionen angezeigt werden; HUGE_VAL ist ein positiver double-Wert. Ein Argumentfehler (domain error)]iegt vor, wenn ein Argument nicht in dem Bereich liegt, für den eine Funktion definiert ist. Bei einem Argumentfehler erhält errno den Wert EDOM; der Resultatwert hängt von der Implementierung ab. Ein Resultatfehler (range error) liegt vor, wenn das Resultat der Funktion nicht als double dargestellt werden kann. Ist das Resultat absolut zu gross, also bei overflow, liefert die Funktion HUGE_VAL mit dem korrekten Vorzeichen, und errno erhält den Wert ERANGE. Ist das Resultat zu nahe bei Null, also bei underflow, liefert die Funktion null; je nach Implementation kann errno den Wert ERANGE erhalten.

In der folgenden Tabelle sind xund y vom Typ double, das Argument n ist ein int**-Wert**, und alle Funktionen liefern double. Winkel werden bei trigonometrischen Funktionen in Radian angegeben.

sin(x)	Sinus von x
cos(x)	Kosinus von x
tan(x)	Tangens von x
asin(x)	arcsin(x)
acos(x)	rccos(x)
atan(x)	arctan(x)
atan2(y,x)	arctan(y/x)
sinh(x)	Sinus Hyperbolicus von x
cosh(x)	Cosinus Hyperbolicus von x
tanh(x)	Tangens Hyperbolicus von x
exp(x)	Exponentialfunktion
log(x)	natürlicher Logarithmus ln(x), x>0.
log10(x)	Logarithmus zur Basis 10 $\log_{10}(x)$,x>0.
pow(x,y)	x hoch Y. Ein Argumentfehler liegt vor bei x=0 und y<=0, oder bei x<0 und y ist nicht ganzzahlig.
sqrt(x)	Wurzel aus x, x>=0.
ceil(x)	kleinster ganzzahliger Wert, der nicht kleiner als x ist, in Form von double.
fabs(x)	absoluter Wert |x|
floor(x)	grösster ganzzahliger Wert, der nicht grösser als x ist, in Form von double.
ldexp(x,n)	x*2 hoch n
frexp(x, int *exp)	zerlegt x in eine normalisierte Mantisse im Bereich [1/2,1), die als Resultat geliefert wird, und eine Potenz von 2, die in *exp abgelegt wird. Ist x null, sind beide Teile des Resultats null.
modf(x, double *ip)	zerlegt x in einen ganzzahligen Teil und einen Rest, die beide das gleiche Vorzeichen wie x besitzen. Der ganzzahlige Teil wird bei *ip abgelegt, der Rest ist das Resultat.
fmod(x,y)	Gleitpunktrest von x/y, mit dem gleichen Vorzeichen wie x. Wenn y null ist, hängt das Resultat von der Implementierung ab.

11.6 Hilfsfunktionen: <stdlib.h>

Die Definitionsfile <stdlib.h> vereinbart Funktionen zur Umwandlung von Zahlen, für Speicherverwaltung und ähnliche Aufgaben.

```
double atof(const char *s)
```

atof wandelt s in double um; die Funktion ist äquivalent zu strtod(s,(char **) NULL).

```
int atoi(const char *s)
```

atoi wandelt s in int um; die Funktion ist äquivalent zu (int)strtol(s,(char **) NULL, 10).

```
long atol(const char *s)
```

atol wandelt s in long um; die Funktion ist äquivalent zu strtol(s,(char **) NULL, 10).

Beispiel:
```
long int zahl;
char str[] = "7194";

zahl = atol(&str[0]);
printf("zahl = %ld\n", zahl);        /* Ausdruck: zahl = 7194 */
```

```
long strtol(const char *s, char **endp, int base)
```

strtol wandelt den Anfang der Zeichenkette s in long um. Dabei wird Zwischenraum am Anfang ignoriert. Der Pointer *endp zeigt auf die abschliessende `\0` der Zeichenkette s oder auf das erste nicht umwandelbare Zeichen in s. Hat base einen Wert zwischen 2 und 36, erfolgt die Umwandlung unter der Annahme, dass die Eingabe in dieser Basis repräsentiert ist. Hat base den Wert Null, wird als Basis 8, 10 oder 16 verwendet; eine führende Null bedeutet dabei oktal, und 0x oder 0X zeigen eine hexadezimale Zahl an. In jedem Fall stehen Buchstaben für die Ziffern von 10 bis base-1; bei Basis 16 darf 0x oder 0X am Anfang stehen. Wenn das Resultat zu gross werden würde, wird je nach Vorzeichen LONG_MAX oder LONG_MIN geliefert, und errno erhält den Wert ERANGE.

Beispiel:

```
#include <stdio.h>
#include <stdlib.h>            /* enthaelt Funktion strtol */
#include <errno.h>             /* enthaelt integer errno */

int main( )
{
      long int zahl;
      char str[] = "73492345634", *endp = NULL;  /* riesige Zahl im String str[] und ein
                                                    auf NULL gesetzter Hilfspointer */
      errno = 0;               /* errno muss vor der Benutzung auf 0 gesetzt werden */
      zahl = strtol(&str[0], &endp, 10);         /* Umwandlung string -> long */
      if(errno == 0 && *endp == `\0`)  /* ok, wenn endp auf abschliessende \0 zeigt */
            printf("ok! zahl = %ld\n", zahl);    /* Bei str[] = "345", Ausdruck: ok! */
      else                                       /* zahl = 345 */
            perror("Fehler"); /* In diesem Bsp wird “Fehler: Result Too Large” ausge-
}                                druckt */
```

```
unsigned long strtoul(const char *s, char **endp, int base)
```

strtoul funktioniert wie strtol, nur ist der Resultattyp unsigned long, und der Fehlerwert ist ULONG_MAX.

```
double strtod(const char *s, char **endp)
```

`strtod` wandelt den Anfang der Zeichenkette `s` in `double` um, dabei wird Zwischenraum am Anfang ignoriert. Der Pointer `*endp` zeigt auf die abschliessende `` `\0` `` der Zeichenkette `s` oder auf das erste nicht umwandelbare Zeichen in s. Falls das Ergebnis zu gross ist, also bei overflow, wird als Resultat `HUGE_VAL` mit dem korrekten Vorzeichen geliefert; liegt das Ergebnis zu dicht bei Null(also bei underflow), wird Null geliefert. In beiden Fällen erhält `errno` den Wert `ERANGE`.

```
int rand(void)
```

`rand` liefert eine ganzzahlige Pseudo-Zufallszahl im Bereich von 0 bis `RAND_MAX`; dieser Wert ist mindestens 32767.

```
void srand(unsigned int seed)
```

`srand` benutzt `seed` als Ausgangswert für eine neue Folge von Pseudo-Zufallszahlen. Der erste Ausgangswert ist 1.

```
void *calloc(size_t nobj, size_t size)
```

`calloc` liefert einen Pointer auf einen Speicherbereich für einen Vektor von `nobj` Objekten, jedes mit der Grösse `size`, oder `NULL`, wenn die Anforderung nicht erfüllt werden kann. Der Bereich wird mit Null-Bytes initialisiert.

```
void *malloc(size_t size)
```

`malloc` liefert einen Pointer auf einen Speicherbereich für ein Objekt der Grösse `size` oder `NULL`, wenn die Anforderung nicht erfüllt werden kann. Der Bereich ist nicht initialisiert.

```
void *realloc(void *p, size_t size)
```

`realloc` ändert die Grösse des Objekts, auf das p zeigt, in `size` ab. Bis zur kleineren der alten und neuen Grösse bleibt der Inhalt unverändert. Wird der Bereich für das Objekt grösser, so ist der zusätzliche Bereich uninitialisiert. `realloc` liefert einen Pointer auf den veränderten Bereich oder `NULL`, wenn die Anforderung nicht erfüllt werden kann; in diesem Fall ist `*p` unverändert.

```
void free(void *p)
```

`free` gibt den Bereich frei, auf den p zeigt; die Funktion hat keinen Effekt, wenn p den Wert `NULL` hat. p muss auf einen Bereich zeigen, der zuvor mit `calloc, malloc` oder `realloc` angelegt wurde.

Beispiel:
Mit den heap Funktionen `calloc(), realloc()` und `free()` wird die dynamische Datenverwaltung gezeigt

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

int main( )
{
    int *bufint;

    printf("Allocate a 512 element buffer\n" );
    if((bufint =(int *)calloc(512, sizeof(int))) == NULL)
         exit(1);

    if((bufint =(int *)realloc(bufint, 1024 * sizeof(int))) == NULL)
         printf("Can't reallocate");

    free(bufint);        /* Free memory */
}


void abort(void)
```

`abort` sorgt für eine anormale Beendigung des Programms im Stil von `raise(SIGABRT)`.

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

`exit` beendet das Programm normal. `atexit`-Funktionen werden in umgekehrter Reihenfolge ihrer Hinterlegung aufgerufen, die Puffer offener Files werden geschrieben, offene Ströme werden abgeschlossen, und die Kontrolle geht an die Umgebung des Programms zurück. Wie `status` an die Umgebung des Programms geliefert wird, hängt von der Implementierung ab, aber Null gilt als erfolgreiches Ende. Die Werte `EXlT_SUCCESS` und `EXIT_FAILURE` können ebenfalls angegeben werden.

```
int atexit(void(*fcn)(void))
```

`atexit` hinterlegt die Funktion `fcn`, damit sie aufgerufen wird, wenn das Programm normal endet, und liefert einen von Null verschiedenen Wert, wenn die Funktion nicht hinterlegt werden kann.

```
int system(const char *s)
```

`system` liefert die Zeichenkette `s` an die Umgebung zur Ausführung. Hat `s` den Wert `NULL`, so liefert `system` einen von Null verschiedenen Wert, wenn es einen Kommandoprozessor gibt. Wenn `s` von `NULL` verschieden ist, dann ist der Resultatwert implementierungsabhängig.

```
char *getenv(const char *name)
```

`getenv` liefert die zu name gehörende Zeichenkette aus der Umgebung oder `NULL`, wenn keine Zeichenkette existiert. Die Details hängen von der Implementierung ab.

```
void *bsearch(const void *key, const void *base, size_t n, size_t size,
int(*cmp)(const void *keyval, const void *datum))
```

`bsearch` durchsucht `base[0]...base[n-1]` nach einem Eintrag, der gleich `*key` ist. Die Funktion `cmp` muss einen negativen Wert liefern, wenn ihr erstes Argument (der Suchschlüssel) kleiner als ihr zweites Argument (ein Tabelleneintrag) ist, Null, wenn beide gleich sind, und sonst einen positiven Wert. Die Elemente des Vektors `base` müssen aufsteigend sortiert sein. `bsearch` liefert einen Pointer auf das gefundene Element oder `NULL`, wenn keines existiert.

```
void qsort(void *base, size_t n, size_t size,int(*cmp)(const void *, const
void *))
```

`qsort` sortiert einen Vektor `base[0]...base[n-1]` von Objekten der Grösse `size` in aufsteigender Reihenfolge. Für die Vergleichsfunktion cmp gilt das gleiche wie bei `bsearch`.

```
int abs(int n)
```

`abs` liefert den absoluten Wert seines `int` Arguments.

```
long labs(long n)
```

`labs` liefert den absoluten Wert seines `long` Arguments.

```
div_t div(int num, int denom)
```

`div` berechnet Quotient und Rest von `num/denom`. Die Resultate werden in den `int` Komponenten `quot` und `rem` einer Struktur vom Typ `div_t` abgelegt.

```
ldiv_t ldiv(long num, long denom)
```

`ldiv` berechnet Quotient und Rest von `num/denom`. Die Resultate werden in den `long` Komponenten `quot` und `rem` einer Struktur vom Typ `ldiv_t` abgelegt.

11.7 Fehlersuche: <assert.h>

Mit dem `assert`-Makro fügt man Testpunkte zu Programmen hinzu:

```
void assert(int expression)
```

Hat `expression` den Wert Null, wenn `assert(expression)` ausgeführt wird, dann gibt der `assert`-Makro auf `stderr` etwa folgende Meldung aus:

```
Assertion failed: expression, file filename, line nnn
```

Anschliessend wird die Ausführung durch Aufruf von `abort` abgebrochen. Der Filename der Programmquelle sowie die Zeilennummer stammen von den Präprozessor-Makros `_ _FILE_ _` und `_ _LINE_ _` .

Wenn beim Einfügen von `<assert.h>` ein Makroname `NDEBUG` definiert ist, wird der `assert`-Makro ignoriert.

11.8 Funktionen für Datum und Uhrzeit: <time.h>

Die Definitionsdatei `<time.h>` vereinbart Typen und Funktionen zum Umgang mit Datum und Uhrzeit. Manche Funktionen verarbeiten die Ortszeit, die von der Kalenderzeit zum Beispiel wegen einer Zeitzone abweicht. `clock_t` und `time_t` sind arithmetische Typen, die Zeiten repräsentieren, und `struct tm` enthält die Komponenten einer Kalenderzeit:

`int tm_sec;`	Sekunden nach der vollen Minute (0, 61)	(Inkl. mögliche Schaltsekunden)
`int tm_min;`	Minuten nach der vollen Stunde (0, 59)	
`int tm_hour;`	Stunden seit Mitternacht (0, 23)	
`int tm_mday;`	Tage im Monat (1, 31)	
`int tm_mon;`	Monate seit Januar (0, 11)	
`int tm_year;`	Jahre seit 1900	
`int tm_wday;`	Tage seit Sonntag (0, 6)	
`int tm_yday;`	Tage seit dem 1. Januar (0, 365)	
`int tm_isdst;`	Kennzeichen für Sommerzeit	

`tm_isdst` ist positiv, wenn Sommerzeit gilt, Null, wenn Sommerzeit nicht gilt, und negativ, wenn die Information nicht zur Verfügung steht.

```
clock_t clock(void)
```

`clock` liefert die Rechnerkern-Zeit, die das Programm seit Beginn seiner Ausführung verbraucht hat, oder -1, wenn diese Information nicht zur Verfügung steht. `clock()/CLOCKS_PER_SEC` ist eine Zeit in Sekunden.

```
time_t time(time_t *tp)
```

`time` liefert die aktuelle Kalenderzeit oder -1, wenn diese nicht zur Verfügung steht. Wenn `tp` von `NULL` verschieden ist, wird der Resultatwert auch bei `*tp` abgelegt.

```
double difftime(time_t time2, time_t time1)
```

`difftime` liefert `time2-time1` in Sekunden ausgedrückt.

```
time_t mktime(struct tm *tp)
```

`mktime` wandelt die Ortszeit in der Struktur `*tp` in Kalenderzeit um, die so dargestellt wird wie bei `time`. Die Komponenten erhalten Werte in den angegebenen Bereichen. `mktime` liefert die Kalenderzeit oder -1, wenn sie nicht dargestellt werden kann.

Die folgenden vier Funktionen liefern Pointer auf statische Objekte, die von anderen Aufrufen

überschrieben werden können.

```
char *asctime(const struct tm *tp)
```

`asctime` konstruiert aus der Zeit in der Struktur `*tp` eine Zeichenkette der Form

```
      Sun Jan 3 15:14:13 1992\n\0
```

```
char *ctime(const time_t *tp)
```

`ctime` verwandelt die Kalenderzeit `*tp` in Ortszeit; dies ist äquivalent zu `asctime(localtime(tp))`

```
struct tm *gmtime(const time_t *tp)
```

`gmtime` verwandelt die Kalenderzeit `*tp` in Coordinated Universal Time (UTC). Die Funktion liefert `NULL`, wenn UTC nicht zur Verfügung steht. Der Name `gmtime` hat historische Bedeutung.

```
struct tm *localtime(const time_t *tp)
```

`localtime` verwandelt die Kalenderzeit `*tp` in Ortszeit.

```
size_t strftime(char *s, size_t smax, const char *fmt, const struct tm *tp)
```

`strftime` formatiert Datum und Zeit aus `*tp` in `s` unter Kontrolle von `fmt`, analog zu einem `printf` Format. Gewöhnliche Zeichen (insbesondere auch die abschliessende `` `\0` ``) werden nach `s` kopiert. Jede `%c` wird so wie unten beschrieben ersetzt, wobei Werte verwendet werden, die der lokalen Umgebung entsprechen. Höchstens `smax` Zeichen werden in `s` abgelegt. `strftime` liefert die Anzahl der resultierenden Zeichen, mit Ausnahme von `` `\0` ``. Wenn mehr als `smax` Zeichen erzeugt wurden, liefert `strftime` den Wert Null.

%a	abgekürzter Name des Wochentags
%A	voller Name des Wochentags
%b	abgekürzter Name des Monats
%B	voller Name des Monats
%c	lokale Darstellung von Datum und Zeit
%d	Tag im Monat (01-31)
%H	Stunde (00-23)
%I	Stunde (01-12)
%j	Tag im Jahr (001-365)
%m	Monat (01-12)
%M	Minute (00-59)
%p	lokales Äquivalent von AM oder PM
%s	Sekunde (00-59)
%U	Woche im Jahr (Sonntag ist erster Wochentag) (00-53)
%w	Wochentag (0-6, Sonntag ist 0)
%W	Woche im Jahr (Montag ist erster Wochentag) (00-53)
%x	lokale Darstellung des Datums
%X	lokale Darstellung der Zeit
%y	Jahr ohne Jahrhundert (00-99)
%Y	Jahr mit Jahrhundert

Beispiel:

/* Mit den Funktionen `time(), localtime(), ctime(), asctime(), gmtime(), mktime()` und `strftime()` wird auf verschiedene Arten gezeigt, wie Datum und Zeit ausgedruckt werden kann. */

```
#include <stdio.h>
#include <time.h>
#include <string.h>

int main( )
{
      struct tm *today, *gmt, xmas_tm = { 0, 0, 12, 25, 11, 90 };
      char s[40], ampm[] = "AM";
      time_t ltime;

      time(&ltime);           /* Get time and display as number and string. */
      printf("\nTime in seconds since GMT 1/1/70:\t%ld\n", ltime);
      printf("Local time and date:\t\t\t%s", ctime(&ltime));


      gmt = gmtime(&ltime);         /* Display GMT. */
      printf("Greenwich Mean Time:\t\t\t%s", asctime(gmt));

      today = localtime(&ltime);    /* Convert to local time structure and adjust for PM if
                                       necessary*/
      if(today->tm_hour > 12)
      {
            strcpy(ampm, "PM");
            today->tm_hour -= 12;
      }
      /* Note how pointer addition is used to skip the first 11 characters and printf is used to trim off
      terminating characters.*/
      printf("12-hour time:\t\t\t%.8s %s\n", asctime(today) + 11, ampm);

      /* Make time for noon on Christmas, 1990. */
      if(mktime(&xmas_tm) != (time_t)-1)
      printf("Christmas:\t\t\t%s\n", asctime(&xmas_tm));

      /* Display the full weekday name and the day of the year */
      if(strftime(s,40,"%A the %j day in the year%y",localtime(&ltime))!=0)
            printf("Today is %s\n\n\n", s);
}
```

Folgendes wird ausgedruckt:

Time in seconds since GMT 1/1/70:	935497729
Local time and date:	Tue Aug 24 08:28:49 1999
Greenwich Mean Time:	Tue Aug 24 12:28:49 1999
12-hour time:	08:28:49 AM
Christmas:	Tue Dec 25 12:00:00 1990

Today is Tuesday the 236 day in the year 99

11.9 Grenzwerte einer Implementierung: <limits.h>

Die Definitionsdatei <limits.h> definiert Konstanten für den Wertumfang der ganzzahligen Typen. Die nachfolgenden Werte sind zugelassene minimale Grössen; grössere Werte können in einer Implementierung benutzt sein (siehe Compiler Handbuch).

CHAR_BIT	8	Bits in einem char
CHAR_MAX	UCHAR_MAX oder SCHAR_MAX	maximaler Wert für char
CHAR_MIN	0 oder SCHAR_MIN	minimaler Wert für char
INT_MAX	+32767	maximaler Wert für int
INT_MIN	-32768	minimaler Wert für int
LONG_MAX	+2147483647	maximaler Wert für int
LONG_MIN	-2147483648	minimaler Wert für int
SCHAR_MAX	+127	maximaler Wert für signed char
SCHAR_MIN	-128	minimaler Wert für signed char
SHRT_MAX	+32767	maximaler Wert für short
SHRT_MIN	-32768	minimaler Wert für short
UCHAR_MAX	255	maximaler Wert für unsigned char
UINT_MAX	65535	maximaler Wert für unsigned int
ULONG_MAX	4294967295	maximaler Wert für unsigned long
USHRT_MAX	65535	maximaler Wert für unsigned short

11.10 Variable Argumentliste: <stdarg.h>

Die Definitionsdatei <stdarg.h> bietet die Möglichkeit, eine Liste von Funktionsargumenten abzuarbeiten, deren Länge und Datentypen nicht bekannt sind.

Angenommen, lastarg ist der letzte benannte Parameter einer Funktion fkt, die mit einer variablen Anzahl von Argumenten aufgerufen wird. Dann vereinbart man in fkt eine Variable ap vom Datentyp va_list, die der Reihe nach auf jedes Argument zeigen wird:

```
va_list ap;
```

ap muss einmal mit dem Makro va_start initialisiert werden, bevor auf die unbenannten Argumente zugegriffen wird:

```
void va_start(va_list ap, type lastarg);
```

Daran anschliessend liefert jede Ausführung des Makros va_arg einen Wert, der den Datentyp und den Wert des nächsten unbenannten Arguments besitzt. Ausserdem ändert das Makro den Wert von ap so, dass der nächste Aufruf von va_arg das nächste Argument liefert:

```
type va_arg(va_list ap, type);
```

Das Makro

```
void va_end(va_list ap);
```

muss einmal aufgerufen werden, nachdem die Argumente bearbeitet wurden.

Beispiel: Es wird eine Funktion mit variabler Argumentenliste gezeigt. Es werden eine unbestimmte Anzahl Strings aneinandergehaengt und ein Pointer auf den zusammengesetzten String zurückgegeben. Zu diesem Zweck wird die Argumentenliste zweimal durchlaufen. Beim ersten Mal wird die Gesamt länge der zusammenzusetzenden Strings ermittelt und dann genau so viel Speicherplatz alloziert. Beim zweiten Durchlauf werden die Strings effektiv zusammen-hängend abgelegt.

```
#include <stdio.h>
#include <stdarg.h>
#include <stdlib.h>
#include <string.h>
char *multcat(int numargs, char lastfix, ...);

int main(void)
{
      char *result;

      result = multcat(4,'x',"One ","two ","testing",".\n");
      printf("%s", result);
      free(result);
      result = multcat(5,'x',"Two ","three ","still testing",".\n","!!\n");
      printf("%s", result);
      free(result);
      return(EXIT_SUCCESS);
}

/*******************************************************************************
 * Funktionsname       : multcat
 * Beschreibung        : haengt Strings aus variabler Arg.liste aneinander
 * Parameter           : variabel
 * Rueckgabewert       : zusammengesetzter String
 *******************************************************************************/
char *multcat(int numargs, char lastfix, ...)
{
      va_list argptr;     /* zeigt nacheinander auf jedes unbenannte Argument */
      static char *result = NULL;
      int i, size=0;

      va_start(argptr, lastfix);          /* zeigt auf 1. unbenanntes Argument */
      for(i = 0; i < numargs; i++)        /* Berechnung der Laenge aller Args */
            size += strlen(va_arg(argptr, char *));
      va_end(argptr);                     /* aufraeumen */
                                          /* Speicherplatz fuer die Args holen */
      if((result =(char *)calloc(size+1,1)) == NULL)
      {
            fprintf(stderr, "out of space\n");
            exit(EXIT_FAILURE);
      }

      va_start(argptr, lastfix);
      for(i = 0; i < numargs; i++)        /* in den allozierten Speicherplatz werden */
            strcat(result, va_arg(argptr, char *));    /* jetzt die Argumente */
      va_end(argptr);                     /* hintereinander gehaengt */

      return(result);    /* die Startadresse des zusammengehaengten Strings wird */
}                        /* zurückgegeben */
```

Folgendes wird ausgedruckt:

One two testing.
Two three still testing.
!!

Anhang A : Traditionen in C

Gewisse Stilelemente, welche von C Programmierern gepflegt werden, haben eine weite Verbreitung gefunden und sind so selbstverständlich wie anderswo ungeschriebene Gesetze. Ein typisches Beispiel sind die Parameter von main, welche fast ausnahmslos mit argc und argv bezeichnet werden. Es dürfte sinnvoll und nützlich sein, sich an die wichtigsten dieser Stilelemente zu halten, tragen sie nicht zuletzt zur erhöhten 'Portabilität von Programmierern' bei.

Variablennamen : `c i j n p s`

Mit c ist eine Variable vom Typ `char` gemeint, `int i` und `j` werden sehr oft als Schleifenindizes gebraucht, `int n` bezeichnet eine Anzahl von Objekten, p steht für einen Pointer und `s` für einen String.

for - Schleife 1

Die Konstruktion `for(n = 10; n >= 0; --n)` wird in dieser Form für eine dekrementierende Schleife eingesetzt, während `for(c = 0; c <= 127; ++c)` für aufsteigende Schleifen gebraucht wird.

for - Schleife 2

`for(i = 0; s[i] ! = `\0`; ++i)` ist typisch für Schleifen, die beim Auftreten eines bestimmten Wertes abbrechen.

for - Schleife 3

`for(i = 0, j = MAX; i < j; ++i, --j)` illustriert, wie die for-Schleife zwei Kontrollvariablen mitführt.

Test auf Null

`if(c), if(p)` etc. (auch bei `while, for`) machen Gebrauch von der Tatsache, dass der numerische Wert 0 `false` entspricht, und vermeiden damit die explizite Angabe eines Vergleichs auf Null. Bei numerischen Werten und Variablen sollte allerdings der explizite Vergleich (z.B. `if(anz != 0)`) immer ausgeschrieben werden.

'Gepacktes' while

`while((c = getchar()) ! = EOF)` ersetzt, dank der Möglichkeit, Assignment Expressions zu schreiben, die sonst übliche Konstruktion mit separater Initialisierung und Schleifenschritt, wie z. B. in Pascal

```
READ(c) ;
WHILE c <> EOF DO BEGIN
        ...
        READ(c)
END ;
```

Ewige Schleife

`for(;;)` ist klassisch für die 'ewige' Schleife. Mit

```
#define FOREVER for(;;)
```

oder

```
#define LOOP for(;;)
```

wird die Absicht noch deutlicher herausgestellt.

for - Schleife 4

`for(i = 0; i < MAXARR; ++i)` ist typisch für Arrayverarbeitung. Die Initialisierung bezieht sich auf das 0te Element des Arrays und die Laufbedingung auf die Anzahl Elemente im Array (und nicht auf das letzte Element des Arrays).

Array - Initialisierung

`static int arr[3] = 0;` initialisiert alle Elemente des Arrays mit 0. Es ist dies ein Spezialfall (der auch für Strukturen gilt), denn es wird ja nur **ein** Initialwert vorgegeben, die übrigen Elemente erhalten per Default den Wert 0 (wegen `static`). Die exziplite Initialisierung des 0ten Elementes macht dies offenkundiger.

for - Schleife 5

`for(p = s; *p; ++p)` ist Standard für die Stringverarbeitung. Die Laufbedingung wird false, wenn das Ende des Strings, also `NULL`, erreicht wird, was den numerischen Wert 0 hat.

Anhang B : Häufig gemachte Fehler in C

Es gibt eine ganze Reihe **Fehler**, die beim Programmieren in C sowohl von Neulingen als auch von alten Hasen oft gemacht werden. Hin und wieder kommt es vor, dass ein Programm mit irgend einem Fehler behaftet ist, den man beim besten Willen nicht lokalisieren kann. In solchen Fällen ist es oft hilfreich, die nachfolgende Liste zu konsultieren.

Allgemein

- uninitialisierte Variablen
- um 1 daneben (Indizes, Listen etc.)
- Nicht-C-Arrays bearbeitet (C-Arrays beginnen bei 0, nicht bei 1)
- Semikolon vergessen
- Semikolon nach Präprozessor-Anweisung
- zu wenig Klammern in Präprozessor-Makros
- Ausdruck mit Nebeneffekt an Präprozessor übergeben
- Klammern (() und {} !) gehen nicht auf

Typen, Operatoren, Ausdrücke

- `char` zusammen mit `getchar` verwendet
- Slash / und Backslash \ verwechselt
- Funktionsargumente **nach** { definiert
- arithmetischer Overflow
- relationale Operatoren auf Strings angewandt (`s=="end"` statt `strcmp (s,"end")`)
- = statt == gebraucht
- Reihenfolgen von Nebeneffekten ausgenützt (z.B. in `a[n] = n++`)
- Operatorenhierarchie missachtet (reduntante Klammern helfen!)
- negative Zahl nach rechts geshiftet
- `` `\0` `` am Ende des Strings vergessen

Programmfluss

- falsch plaziertes `else`
- `break` in `switch` vergessen
- erster oder letzter Fall in Schleife anormal
- Schleife nie ausgeführt
- Array (beginnt bei 0, hat `n` Elemente, also letztes Element hat Index n-1) falsch
- Semikolon nach Kontrollanweisung, erzeugt leere Anweisung (z.B. `for(...);`)

Funktionen und Programmstruktur

- falsche Typen bei Argumenten
- falsche Reihenfolge bei Argumenten
- falsche Anzahl von Argumenten
- angenommen, `static` werde immer wieder initialisiert

Pointer und Arrays

- Wert statt Pointer (oder umgekehrt) übergeben (klassischer Fall: `scanf`)
- `char` mit `char *` verwechselt
- Pointer auf inexistenten String gebraucht
- Dangling reference, Aliasing, Garbage
- ' und " verwechselt

Anhang C : Regeln zur Erstellung von portablen Codes

Die nachfolgende Liste soll als Anleitung dienen, wenn ein Programm erstellt werden muss, das auf verschiedene Architekturen portiert werden soll.

- **Nie annehmen, int und Pointer hätten dieselbe Grösse.** Und daran denken, dass die meisten Defaults `int` sind.

- **Code vermeiden, der auf Annahmen über irgendeine Ordnung basiert,** es sei denn, eine solche Ordnung sei im Standard beschrieben. Keine Vorschriften bestehen z.B. bezüglich Anordnung der Bytes in einem Wort, Länge der Datentypen, Evaluation von Unterausdrücken, Argumenten etc...

- **Nie 'magische' Zeichen einer Maschine ausnutzen** (so bedeutet ctrl-Z nicht in jedem Betriebssystem `EOF`).

- **Wo immer möglich, symbolische Konstanten (via `#define`) benützen.**

- **Alle maschinenabhängigen Deklarationen und Definitionen in einem Header-File konzentrieren.**

- **Alles, aber auch wirklich alles exziplit deklarieren.** Funktionen, die keinen Funktionswert liefern, sind mit `void` zu deklarieren.

- **Nie annehmen, dass alle Pointer immer gleichartig sind und auf alle Datentypen zeigen können.** Auf gewissen Maschinen ist beispielsweise ein `char *` **länger** als ein `int *`, auf anderen Maschinen zeigt ein Pointer auf eine Funktion auf einen anderen Speicherplatz als ein Pointer auf Daten.

- **`NULL` sollte nicht als explizites Argument für irgendeine Funktion, welche Pointer entgegennimmt, verwendet werden.** In gewissen Umgebungen sind Pointer auf verschiedene Datentypen unterschiedlich gross. Dieses Problem wird übrigens vermieden, wenn der entsprechende Funktions-Prototyp vorhanden ist. Dann nämlich wird `NULL` automatisch zum richtigen Pointertyp konvertiert.

- **Die Funktion main immer mit einem expliziten return oder exit verlassen.** (Zugegeben, das ist im vorliegenden Text nur ausnahmsweise der Fall.)

- **Nie int verwenden!** Eine `int` hat die gleiche Grösse wie ein Register der Maschine. Wo immer diese Grösse eine Rolle spielt, ist `short` oder `long` zu verwenden. Am besten mit einem Header-File und `typedef`'s arbeiten :

```
#ifdef PC
      typedef           short S2BYTE
      typedef           long  S4BYTE
      ...
#endif

#ifdef VAX
      typedef           short S2BYTE
      typedef           int   S4BYTE
      ...
#endif
```

Egal, wie man's anstellt, es wird kaum je möglich sein, hundertprozentig portablen Code zu erstellen. Deshalb sollte man die maschinen- und betriebssystemspezifischen Programmteile in separaten Files zusammenfassen.

Anhang D : Methode zur Zerlegung der Deklarations- und Definitionssyntax

Was ist x im folgenden Beispiel :

```
char  (*(*x())[])()     ?
```

(x ist eine Funktion, welche einen Pointer liefert, der auf einen Array von Pointern auf Funktionen, welche `char` liefern, zeigt !)
Ein zweifellos wenig erbauliches Thema in C ist die Syntax, mit der Definitionen und Deklarationen (nachfolgend DD genannt) von Objekten festgelegt werden. Ein Grund für die Schwierigkeit rührt daher, dass Typenbezeichner an verschiedenen Stellen vorkommen können:

- in Deklarationen. Hier wird ein Typ einem Namen zugeordnet. Beispiel: `extern char c;`
- in Definitionen. Hier werden ein Typ und Wert (nämlich der Speicherplatz) einem Namen zugeordnet. Beispiel: `char c;`
- in Casts. Hier wird die Repräsentation eines Objektes von einem Typ in einen anderen vorgenommen (Typenkonversion). Beispiel : `(int) c;`

Im Rest dieses Abschnittes soll nun versucht werden, das Thema etwas systematischer zu behandeln, so dass der Leser oder die Leserin schliesslich in der Lage sein wird, auch komplizierte DD auf Anhieb sicher zu interpretieren bzw. zu konstruieren.

Grundsätzlich hat eine DD die Form

{storage-class} data-type declarator {= initializer}

wobei die storage-class und initializer fakultativ sind. Beim data-type handelt es sich um einen der bereits dargestellten Basisdatentypen oder um einen typedef-Namen. Schliesslich bleibt noch der declarator, welcher aus folgenden Elementen bestehen kann :

1) identifier mit Typ T wird definiert
2) (declarator) gleich wie declarator
3) *declarator gleich wie declarator in einer DD mit dem Typ 'Pointer auf T'
4) declarator() gleich wie declarator in einer DD mit dem Typ 'Funktion, welche einen Wert vom Typ T liefert'
5) declarator[N] gleich wie declarator in einer DD mit dem Typ 'Array mit N Elementen vom Typ T'

Die verschiedenen Möglichkeiten können beliebig kombiniert werden, mit den folgenden Einschränkungen:

6) Funktionen können keine Arrays oder Funktionen als Funktionswerte liefern
7) Arrays von Funktionen sind nicht erlaubt
8) Funktionen können nicht Komponenten von struct oder union sein
   und schliesslich
9) haben in DD () und [ ] höhere Präzedenz als *

Hier ist eine einfache Methode beschrieben, mit welcher sich DD in ein leicht verständliches, pascal-ähnliches Pseudoenglisch umsetzen lassen.

Die Methode arbeitet wie folgt :

Man schreibe die C-Syntax der DD auf und reduziere diese gemäss Regel 9) Schritt für Schritt von innen nach aussen. Bei jedem Reduktionsschritt wird der C-Ausdruck um ein Element gemäss 1)..5) kleiner und der Pseudoausdruck um ein Element grösser.

*	wird umgesetzt in 'pt-to'
()	wird umgesetzt in 'func-ret'
[ ]	wird umgesetzt in 'arr-of'

Zuletzt wird überprüft, ob der entstandene Ausdruck eine der Regeln 6)..8) verletzt. In den folgenden Beispielen wird jeweils die Position des Variablennamens mit einem . markiert. Ist der Punkt von runden Klammern umschlossen (.) , können diese weggelassen werden.

Beispiel 1 : `int (*pti)[ ]`

1. Schritt:	`int (*.)[]`	: pti
2. Schritt:	`int (.)[]`	: pti pt_to
3. Schritt:	`int`	: pti pt_to arr_of

Resultat 1: 'pti is a pointer to an array of int'

Beispiel 2: `char (*(*x())[])()`

1. Schritt:	`char (*(*.())[])()`	: x
2. Schritt:	`char (*(*.)[])()`	: x func_ret
3. Schritt:	`char (*(.)[])()`	: x func_ret pt_to
4. Schritt:	`char (*.)()`	: x func_ret pt_to arr_of
5. Schritt:	`char .()`	: x func_ret pt_to arr_of pt_to
6. Schritt	`char`	: x func_ret pt_to arr_of pt_to func_ret

Resultat 2: 'x is a function returning a pointer to an array of pointers to functions returning char'

Beispiel 3: `char (w()[])()`

1. Schritt:	`char (.()[])()`	: w
2. Schritt:	`char (.[])()`	: w func_ret
3. Schritt:	`char .()`	: w func_ret arr_of
4. Schritt:	`char`	: w func_ret arr_of func_ret

Resultat 3: 'w is a function returning an array of functions returning char'
Allerdings ist diese DD illegal, weil sie gegen Regeln 6) und 7) verstösst.

Die Umkehrung der Methode, um von einer englischen Umschreibung zur C-Syntax zu gelangen, sollte offensichtlich sein.

********** **Aufgaben** **********

23) Interpretieren Sie die folgenden DD :

```
char *argv[]

int (*comp)()

int *(*k())()[]
```

24) Erstellen Sie die DD für folgende Beschreibung :
a ist ein Pointer auf einen Array von Pointern auf Funktionen, welche `float` liefern.

25) Erstellen Sie die DD für folgende Beschreibung :
`m` ist eine Funktion, welche einen Pointer auf einen Array von Pointer auf `double` liefert.

.

Stichwortverzeichnis

:: 8, 12, 39, 42, 50, 96, 97, 99-101
<assert.h> 102, 119
<ctype.h> 102, 113
<errno.h> 102, 111, 112, 115, 116
<iostream> 12, 36, 97
<limits.h> 102, 122
<math.h> 102, 115
<setjmp.h> 102
<signal.h> 102
<stdarg.h> 64, 102, 122, 123
<stddef.h> 102
<stdio.h> 102, 103
<stdlib.h> 102, 116, 117, 123
<string.h> 102, 111, 114, 121, 123
<time.h> 102, 119, 121
#define 12, 87, 127
#else 87-89
#error 87, 89
#if 87-89
#ifdef 87-89, 127
#include 12, 87, 88
#line 87
#pragma 87
abort 117-119
abs 118
acos 115
ANSI 7
ANSI-Standard Libraries 102
Argument 13, 64
Argumente beim Programmaufruf 82
Argumentenliste 62
Arithmetische Operatoren 32
Array 19, 23
asctime 120, 121
asin 115
assert 102, 119
atan 115
atan2 115
atexit 118
atof 116
atoi 116
atol 116
Ausdrücke 32
auto 14, 21, 22
Bedingte Ausdrücke 49
Beispiele 37, 40, 43, 47, 70, 94, 99
Bibliothek 64, 103
Bit-Felder 47
Bitweise Operatoren 45
Blockanweisung 13, 52, 54, 57
bool 14, 16, 17, 41, 42, 95, 96
break 57
bsearch 118
calloc 117, 123
case 14, 40, 57, 58
CAST-Operator 41
ceil 115
cerr 36, 81, 95
char 16, 41
cin 11, 12, 15, 36, 57, 63, 67, 95, 97
clearerr 110, 112
clock 119
clog 95
const 20, 42, 86
const_cast 14, 41-44
continue 14
cos 115
cosh 115
cout 95
ctime 120, 121
Datendefinition 16
Datendeklaration 16
Datentypen 16
Datentypen Grösse 17
default 57
Definition 16
Deklaration 16
Dekrementoperatoren 36
delete 14, 39, 81
difftime 119
div 118
do-while 54
double 16, 41
else 53
enum 14, 30, 41, 91, 93
exit 117, 118, 123, 127
exp 115
extern 21, 61
fclose 107, 108
feof 110-112
ferror 110-112
fflush 107, 109
fgetc 107, 109
fgetpos 112
fgets 108
FILE * 103
Fileoperationen 107
float 16, 41
floor 115
flush 96
fmod 115
fopen 107, 108
for 55
fprintf 108, 112, 123
fputc 107, 109
fputs 108, 109
fread 110, 111
free 81, 117, 123
freopen 109
frexp 115
fscanf 108
fseek 110, 111
fsetpos 112
fstream 99, 101
ftell 110, 111
Funktion 10, 61
fwrite 110, 111
get 15, 57, 67, 96, 98-101, 121
getchar 106
getenv 118
getline 96
Gleitpunktkonstante 19
gmtime 120, 121
goto 14, 51, 58, 74
Headerfile 90-94

if 53
ifstream 99
Inkrementoperatoren 36
inline 69
int 12, 16
isalnum 113
isalpha 113
iscntrl 113
isdigit 113
isgraph 113
islower 113
isprint 113
ispunct 113
isspace 113
isupper 113
isxdigit 113
Konstante 18, 19, 30, 39, 43, 86, 88, 103
Kontrollfluss 51
Konversion 41, 42, 75
labs 7, 118
ldexp 115
ldiv 118
Library 7
Listen 37
localtime 120, 121
log 115
log10 115
Logische Operatoren 34
long 16, 18, 41
main 12, 63, 82
Makros 87
malloc 81, 117
memchr 114
memcmp 114
memcpy 114
memmove 114
memset 114
mktime 119, 121
modf 115
modulo 32, 39
new 9, 14, 39, 43, 81
NUL 18
NULL 103
ofstream 99
operator 50
Operatoren 16, 32, 34, 36, 38, 90
Parameter 13, 64, 66
Parameterliste 62
perror 112, 116
Pointer 71
Pointer auf Funktionen 85
Pointer auf Strukturen 78
Pointer und Arrays 76, 126
pow 115
Präzedenz 32, 38
printf 104
Programm Beispiele
dynamischer Speicherplatz 117
Fakultäten 11
file of records 111
File öffnen 99
filecopy 108
Lirum 67
Pointer auf Funktionen 85
Positionieren im File 101
strtol 116
variable Argumente 123
Zeitfunktionen 121
put 57, 83, 96, 99-101
putc 109
putchar 106
puts 106
qsort 118
rand 117
realloc 117
Referenzübergabe 64
register 21
rekursiv 11
remove 109
return 13, 63
rewind 110, 111
scanf 12, 105
Schlüsselwörter 14
Scope 16, 39, 50
setbuf 110
setvbuf 110
signed 14, 16, 17, 47, 122
sin 115
sinh 115
size_t 103
sizeof 16, 27, 37
Speicherklassen 14, 21
sprintf 105
sqrt 115
srand 117
sscanf 106
static 21
strcat 114, 123
strchr 114
strcmp 114, 126
strcpy 114, 121
strcspn 114
strerror 114
strftime 120, 121
strlen 114, 123
strncat 114
strncmp 114
strpbrk 114
strrchr 114
strspn 114
strstr 114
strtod 116, 117
strtok 114
strtol 116
strtoul 116
struct 26
Strukturen 26
switch 57
Syntax 13
system 118
tan 115
tanh 115
time 102, 119-121
tmpfile 109, 111
tmpnam 110
tolower 113
toupper 113
typedef 31, 127
Typenkonversion 40, 45
Typenqualifikation 86

Übergabe eines Arrays 65
ungetc 109
union 29
UNIX 7, 94, 99, 103
unsigned 16
Vergleichs Operatoren 34
void 10, 61
void * 103
volatile 14, 20, 42, 86
Wertübergabe 64
while 13, 54
Zeichenkonstante 18
Zuweisungsoperatoren 48