

Inhalt

1 Vorbemerkungen.....	5
1.1 Literatur.....	5
2 Grundlagen.....	5
2.1 Was heisst "Programmieren"?.....	5
2.2 Vom Steuern zum Programmieren.....	6
2.3 Höhere Programmiersprachen.....	6
2.3.1 Wie entsteht der Maschinencode ?.....	7
2.4 Grundsätzliches zu Programmiersprachen.....	8
2.5 Vorgehensweise zur SW-Erstellung.....	8
2.5.1 Der Texteditor.....	8
2.5.2 Compiler und Linker.....	9
2.5.3 Der Debugger.....	9
2.5.4 Integrierte Entwicklungsumgebung (IDE).....	9
2.5.5 Die Konsolenanwendung.....	9
3 Grundlagen von C.....	11
3.1 Allgemeine C-Syntaxregeln.....	12
3.2 Basiskomponenten eines C-Programmes.....	12
3.2.1 Kommentare.....	12
3.3 Variablen.....	13
3.3.1 Die Variablen-Definition.....	14
3.3.2 Initialisierung.....	14
3.3.3 Konstanten und Literale.....	15
3.3.4 Die const-Deklaration.....	16
3.4 Gültigkeitsbereiche.....	16
3.5 Die einfachen ausführbaren Anweisungen.....	18
3.5.1 Monitor - Ausgaben in C++.....	18
3.5.2 Eingaben in C++.....	19
3.5.3 Die Zuweisung.....	20
3.5.4 Typwandlung.....	21
3 Operatoren.....	22
3.5 Arithmetische Operatoren.....	23
3.6 Zuweisungsoperatoren.....	23
3.7 Vergleichsoperatoren.....	23
3.8 Postfix- Operatoren.....	24
3.9 Logische Operatoren.....	25
3.10 Sonstige Operatoren.....	26
4.6.1 Bedingungsoperator.....	26
4.6.2 Der sizeof() – Operator.....	26
4.6.3 Der Adressoperator &.....	26
4.6.4 Der Bereichsoperator ::.....	26
4 Kontrollstrukturen.....	27

4.5 Die Verbundanweisung.....	28
4.6 Die for – Schleife.....	28
4.7 Die while – Schleife.....	30
4.8 Die do....while – Schleife.....	31
4.9 Die if – Anweisung.....	31
4.9.5 Die einfache if-Anweisung.....	31
4.9.6 Die if...else – Anweisung.....	32
4.10 Einige typische Fehler.....	33
4.11 Die switch – Anweisung.....	34
4.12 Verschiedene Kontroll-Anweisungen.....	35
4.12.5 Break – Anweisung.....	35
4.12.6 Continue – Anweisung.....	35
4.12.7 goto – Anweisung (Sprung – Anweisung).....	36
5 Felder (Arrays).....	36
5.5 Deklaration von Feldern.....	37
5.5.5 Eindimensionale Felder.....	37
5.5.6 Mehrdimensionale Felder.....	37
5.6 Verwendung von Feldern.....	37
5.7 Initialisierung von Feldern.....	38
6 Modularisierung.....	39
6.5 Funktionen in C.....	39
6.6 Funktionsdeklaration.....	39
6.7 Verwendung von Funktionen: Der Funktions-Aufruf.....	41
6.7.5 Parameter und Ergebniswerte.....	42
6.8 Referenzübergabe.....	43
6.9 Standardparameter (Defaultargument).....	44
6.10 Übergabe von Feldern an Funktionen.....	44
6.11 Gültigkeitsbereiche.....	45
6.12 Statische Variable und Funktionen.....	45
6.12.5 Globale Deklaration.....	45
6.12.6 Lokale Deklaration.....	46
6.13 Inline Funktionen.....	47
6.14 Überladen von Funktionen.....	47
6.15 Rekursion.....	48
6.16 Variable Parameterliste.....	48
6.17 Kommandozeilenparameter.....	49
6.18 Aufgliederung in mehrere Quellcodedateien.....	50
6.18.5 Funktionsdeklaration und – Implementierung.....	50
6.18.6 Die Header-Datei.....	51
6.18.7 Externe Variablen.....	52
6.19 Namensbereiche (Namespaces).....	53
6.19.5 Deklaration von Namensbereichen.....	53
6.19.6 Zugriff auf Elemente aus Namensbereichen.....	53

6.20 Die Standard Bibliothek (RunTime-Library – RTL).....	54
7 Der Präprozessor.....	55
7.5 Übersicht der Präprozessor-Direktiven.....	56
7.6 Dateien einfügen mit der Direktive include.....	57
7.7 Makro und define-Direktive.....	57
7.7.5 Das einfache Makro.....	57
7.7.6 Parametrisches Makro.....	58
7.7.7 Der #-Operator.....	58
7.7.8 Der ##-Operator.....	59
7.8 Bedingte Compilierung.....	59
7.9 Sonstige Direktiven.....	60
7.9.5 Makro assert.....	60
7.9.6 Direktive line.....	60
7.9.7 Direktive error.....	60
7.9.8 Direktive pragma.....	60
7.10 Vordefinierte Größen.....	61
8 Formatierte Ein- und Ausgaben.....	62
8.5 Ausgabe mit printf().....	62
8.6 Eingabe mit scanf().....	64
8.7 Eingabe von Zeichenketten mit Leerstellen.....	65
9 Dateioperationen.....	66
9.5 Auf eine Datei schreiben.....	67
9.6 Drucken.....	68
9.7 Lesen von Datei.....	68
9.8 Neuere Versionen der RTL-Funktionen.....	69
9.9 Einige weitere RTL-Funktionen für Dateioperationen.....	69
10 Selbstdefinierte Datentypen.....	70
10.5 Synonyme definieren mit typedef.....	70
10.6 Der Aufzählungstyp (Enumeration).....	70
10.7 Strukturen (C).....	72
10.7.5 Syntax der Definition.....	72
10.7.6 Deklaration von Variablen vom Typ einer Struktur.....	73
10.7.7 Zugriff auf Variablen vom Typ einer Struktur.....	74
10.7.8 Arrays von Strukturen.....	74
10.7.9 Die tm – Struktur aus der RTL.....	74
10.8 Bitfelder.....	75
10.9 Die Union.....	76
10.10 Ein- und Ausgaben.....	76
11 Zeiger.....	77
11.5 Zeiger – Deklaration.....	77
11.6 Wertzuweisung an Zeigervariable.....	78
11.7 Dereferenzieren.....	78

11.8 Typwandlung bei Zeigern.....	78
11.9 Zeiger auf void.....	79
11.10 Der Null-Zeiger.....	79
11.11 Referenzübergabe bei Zeigern.....	79
11.12 Zeiger- Arithmetik.....	79
11.13 Dynamische Speicherverwaltung.....	80
11.13.5 Dynamische Speicherreservierung.....	81
11.13.6 Speicher-Freigabe.....	81
11.13.7 Speicherplatzgrenzen.....	82
11.14 Zeiger auf Variablen vom Typ einer C - Struktur.....	82
11.15 Verkettete Listen.....	82
11.16 Funktionszeiger.....	84
11.16.5 Der Typ einer Funktion.....	85
11.16.6 Deklaration eines Funktionszeigers.....	85
11.16.7 Zuweisung eines Wertes an einen Funktionszeiger.....	85
11.16.8 Funktionsaufruf mittels Funktionszeiger.....	85
12 Zeichenketten.....	86
12.5 Bibliotheks-Funktionen für Strings.....	87
12.6 Sicherheitsaspekte.....	88
13 Hinweise zur Programmerstellung.....	88

1 Vorbemerkungen

Der Nachfolgende Text behandelt die Programmierung mit der Programmiersprache C/C++ ohne die objektorientierte Programmierung, wie sie von C++ unterstützt wird. Zu Beginn wird jedoch die Eingabe mit `cout` und `cin` verwendet, welche bereits SW-Elemente verwendet, die selbst objektorientiert programmiert sind. Dies erfolgt wegen der Einfachheit und da die Verwendung selbst keine Kenntnisse des objektorientierten Programmierens erfordert.

Immer wenn im Text auf C Bezug genommen wird („Dies wird in C realisiert durch...“) gilt Entsprechendes auch für C++.

Am Ende jedes Kapitels sind einige Programmbeispiele gegeben. Es empfiehlt sich

- Den Code der Beispiele zu analysieren und damit die Funktionsweise zu verstehen!
- Den Code der Beispiele abzuschreiben und die erhaltenen Resultate mit den Vorgaben zu vergleichen!

1.1 *Literatur*

- ANSI C 2.0 – Grundlagen Programmierung, Herdt-Verlag, www.herd.com
- C/C++ - Referenz, Franzis-Verlag, ISBN 3-7723-6353-9

2 Grundlagen

2.1 *Was heisst "Programmieren"?*

Das zentrale Bauteil eines Computers ist der Mikroprozessor. Dabei handelt es sich um einen kleinen Silizium-Chip, der in schwarzes Plastik eingegossen ist. Mit der Außenwelt verbindet den Chip eine Reihe von kurzen Leitungen, die wie die Beine eines Käfers aussehen.

Diese "Beine" sind Schalter, mit denen bestimmt werden kann, was im Inneren des Prozessors geschieht. Sie werden "eingeschaltet", indem eine Spannung von wenigen Volt (3.3 oder 5 V) angelegt wird. Man bezeichnet diesen Zustand mit "1". Ist keine Spannung angelegt, bezeichnet man ihn mit "0".

Alle Schalter haben eine unterschiedliche Bedeutung. Einige sind mit dem Arbeitsspeicher verbunden, andere mit weiteren Bausteinen des Computers. So gibt es u. a. einen Taktgeber, der einige Millionen mal pro Sekunde einen bestimmten Schalter des Prozessors ein- und wieder ausschaltet.

Ein Mikroprozessor wird gesteuert, indem aus dem Arbeitsspeicher nacheinander eine Folge von Bytes geladen wird und diese als unterschiedliche Spannungen an die verschiedenen "Beine" oder Schalter des Prozessors angelegt werden.

Einige der Schalterkombinationen machen Sinn, andere nicht: So bringen Sie Ihren Videorekorder durcheinander, wenn Sie die *Eject*- und *Play*-Tasten gleichzeitig drücken - er reagiert nicht. Ein Prozessor ist da weniger "gutmütig", er "stürzt ab". Gleiches passiert bei einem Mikroprozessor, wenn an bestimmte Schalter gleichzeitig eine Spannung angelegt wird, wo dies eigentlich nicht der Fall sein darf.

Die sinnvollen Kombinationen von angelegten Spannungen nennt man Anweisungen oder Befehle und die Menge aller Anweisungen bildet den Befehlssatz des Mikroprozessors. Der **Befehlssatz** (=die Menge der verfügbaren Befehle) eines bestimmten Mikroprozessor-Typs wird vom Hersteller festgelegt.

Da es keine Standards gibt, sind die Befehlssätze verschiedener Mikroprozessor - Typen in aller Regel unterschiedlich. Die Anweisungen, die ein Prozessor von *Motorola* "versteht", würden bei *Intel*- Prozessoren zu einem komplett anderen Ergebnis führen, in der Regel sogar zu einem Absturz. Natürlich gibt es auch hier Ausnahmen. So bietet z.B. *AMD* günstige Prozessoren an, die denselben Befehlssatz verstehen wie die Original-Prozessoren von *Intel*. Dies ist aber eher die Ausnahme. Intel wird sich hüten, die Funktionsweise ihrer Prozessoren der Konkurrenz zu verraten.

Jeden Befehl kann man somit in der Form 0001101 ... darstellen, man spricht von der Binärdarstellung eines Befehls:

In dieser Darstellung ist jeder Schalter, an dem eine Spannung anliegt, mit einer 1, alle anderen sind mit einer 0 markiert. Man spricht bei den Schaltern dann von Bits.

Da es recht mühsam ist, einen Mikroprozessor auf diese Weise zu steuern, wurden bald eine Reihe von Abkürzungen für diese Anweisungen festgelegt:

Beispiel

- **LD** für eine Ladeanweisung
- **TST** für eine Testanweisung

Diese Darstellung der Anweisungen nennt man Assembler - und genauso heißen auch die Programme, die aus diesen Anweisungen aufgebaut werden.

□

2.2 Vom Steuern zum Programmieren

Bisher war immer von "Steuern" die Rede und nicht von "Programmieren". Programmieren geht einen Schritt weiter: Da Mikroprozessoren heute mehrere Millionen solcher Anweisungen pro Sekunde ausführen, macht es natürlich keinen Sinn, jede dieser Anweisungen selber "einzuschalten". Programmieren besteht nun darin, sich innerhalb von Tagen oder Wochen eine Folge von Anweisungen auszudenken und aufzuschreiben (zu "speichern") und diese vom Mikroprozessor innerhalb von wenigen Sekunden ausführen zu lassen.

Tauchen dabei Fehler auf (ein Absturz wegen einer falschen Schalterstellung oder falsche Ergebnisse), sucht man den oder die Fehler, ändert die Anweisungen und lässt sie erneut ausführen. Diese aufgeschriebene Folge von Anweisungen nennt man ein Programm.

□

2.3 Höhere Programmiersprachen

Das Programmieren in Assembler ist umständlich, da die Befehle sehr kryptisch sind. Man braucht z. B. schon vier oder fünf dieser Anweisungen, um zwei Zahlen zu addieren.

Außerdem sind die Anweisungen in Assembler oder im Maschinencode auf jedem Prozessor anders, d.h. ein Programm, das auf einem Rechner geschrieben wurde, läuft nicht auf einem anderen Rechnertyp mit anderem Prozessor.

Im Laufe der Zeit wurden deshalb eine ganze Reihe von verschiedenen - so genannten "höheren" - Programmiersprachen entwickelt, die anschaulicher und leichter zu verstehen sind und deren Code leichter zu lesen ist.

Das Programmieren in diesen Sprachen ist im Vergleich zu Assembler sehr viel einfacher. Und es gibt Programme, welche die "höheren" Programmiersprachen nach Assembler oder in den Maschinencode übersetzen. Diese Programme nennt man **Compiler**.

Eine der höheren Programmiersprachen ist C, aus welcher später durch Weiterentwicklung C++ entstand. Ein wesentliches Ziel der höheren Programmiersprachen war die **Plattformunabhängigkeit**, das bedeutet, dass Programme, die in einer höheren Programmiersprache geschrieben wurden, nicht mehr an einzelne, bestimmte Prozessoren gebunden sind:

- Man schreibt ein Programm in einer Programmiersprache und lässt es erst danach von einem Compiler in den Maschinencode eines speziellen Prozessors übersetzen. Damit kann das Programm von diesem Prozessor ausgeführt werden.
- Man kann dasselbe Programm mit einem anderen Compiler aber auch in den Maschinencode eines anderen Prozessors übersetzen, sodass es dann nach dem Übersetzen auch auf dem anderen Prozessor läuft.

Folglich muss man jeweils die passenden Compiler verwenden.

2.3.1 Wie entsteht der Maschinencode ?

Die Datei, welche die vom Programmierer in einer höheren Programmiersprachen Anweisungsfolge – den **Programmcode** oder **Quelltext** – enthält heisst die **Quellcodedatei**. Mittels des Compiler wird daraus die Datei erstellt, welche die Anweisungsfolge enthält, welche der Prozessor direkt verarbeiten kann, den sogenannten **Maschinencode**. Diese Datei nennt man eine **ausführbare Datei**

Grafisch kann man dies in folgender Weise darstellen



Häufig tritt in der Praxis jedoch die Situation auf, dass man gleichartige Funktionen in verschiedenen Programmen verwenden kann und damit auch möchte. Genauso wie in anderen Bereichen der Technik liegt es hier nahe, die Lösungen – hier Quellcode - welche bereits verfügbar sind, an anderer Stelle wieder zu verwenden. Häufig werden solche SW-Bausteine in Funktionsbibliotheken zusammengefasst

Weiterhin ist es bei grossen SW-Projekten normal, dass mehrere oder viele SW-Entwickler daran arbeiten. Dazu benötigt jeder eine eigene Quellcodedatei.

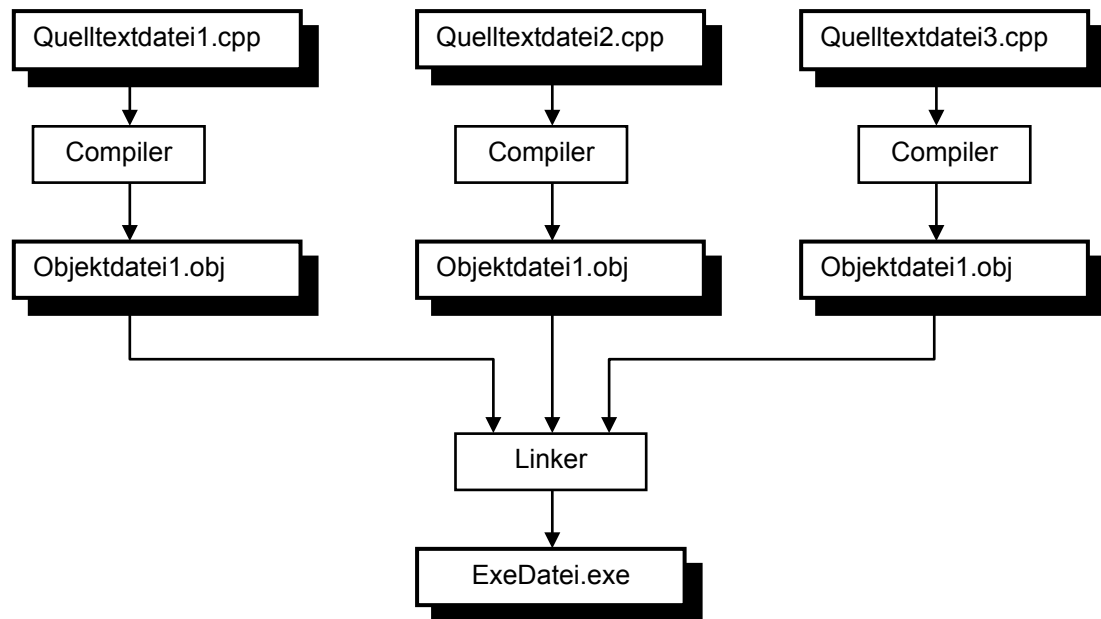
Beide Gründe führen dazu, dass SW-Projekte i.d.R. aus mehreren Quelltextdateien, den sogenannten **Modulen**, bestehen. Daraus muss jedoch 1 ausführbare Datei erzeugt werden! Folglich kann der Übersetzungsprozess nicht so einfach geschehen wie oben dargestellt. Normalerweise wird er als 2-stufiger Prozess ausgeführt:

- In der 1. Stufe wird zunächst durch den eigentlichen Compiler jedes Modul für sich alleine compiliert (übersetzt). Dabei wird auch die Syntax überprüft und ggf **Syntaxfehler** detektiert. Syntaxfehler führen zum Abbruch des Vorgangs und Generierung von entsprechenden Fehlermeldungen. Gibt es keine Syntaxfehler, erzeugt der Compiler für jedes Modul den sogenannten **Objektcode** (Objekt-Datei).
- In der 2. Stufe wird dann durch eine spezielle SW – aus den Objekt – Dateien und ggf aus SW-Bibliotheken (in Form von Dateien) die ausführbare Datei zusammengefügt. Man spricht dabei von Binden, engl. Linken. Die entsprechende SW heisst deshalb **Linker**.

Nachfolgend wird jedoch weiterhin vom Compiler und Compilierungsprozess die Rede sein, wobei der komplette Prozess gemeint ist.

Hinweis: Die oben beschriebene Vorgehensweise wird zwar von vielen, jedoch nicht allen Programmiersprachen verwendet.

Die folgende Darstellung zeigt somit den tatsächlichen Ablauf am Beispiel eines SW-Projektes mit 3 Quellcodedateien:



2.4 Grundsätzliches zu Programmiersprachen

Da der Quell-Code höherer Programmiersprachen automatisch in ausführbare Dateien übersetzt wird, unterliegen sie sehr strengen **Syntaxregeln** (Rechtschreibregeln). Geringste Abweichungen führen zu Syntaxfehlern, deren Vorteil jedoch ist, dass sie automatisch durch den Compiler erkannt werden.

Ausserdem verfügt jede Programmiersprache über einen eigenen Satz sogenannter **Schlüsselwörter**. Dabei handelt es sich um Zeichenfolgen, wie z.B. `const` oder `float`. Diese haben eine genau definierte Bedeutung und dürfen nur dieser Bedeutung entsprechend verwendet werden.

2.5 Vorgehensweise zur SW-Erstellung

Gute und zuverlässige SW erfordert intensive Vorentwicklung (Analyse- und Entwurfsphase) bevor die eigentliche Code-Erstellung begonnen werden kann. Dies ist nicht Thema des vorliegenden Dokumentes

Zur Erstellung von SW- werden folgende Hilfsmittel benötigt:

- Text-Editor
- Compiler
- Linker
- Debugger

2.5.1 Der Texteditor

Grundsätzlich ist jeder Editor verwendbar, welcher reine Text-Dateien (d.h. ohne Formatierungsinformationen usw. in der Datei) erzeugt geeignet. Allerdings gibt es heute spezialisierte Editoren, welche den Programmierer unterstützen, indem sie spezielle Elemente des Codes farblich hervorheben.

2.5.2 Compiler und Linker

Deren Aufgabe wurde bereits beschrieben. Sie müssen Code erzeugen, welche auf der jeweiligen **Plattform** (Mikroprozessor und Betriebssystem) lauffähig ist. DA die Sprache C und C++ international standardisiert ist, sollte man davon ausgehen, dass verschiedene Compiler sich bei demselben Quellcode identisch verhalten. Dies ist jedoch nicht generell der Fall. So kann es sein, dass dieselbe Quellcodedatei von einem Compiler als fehlerfrei akzeptiert wird, wohingegen ein anderer Compiler Syntaxfehler meldet. Die Gründe hierfür sind

- Ungenauigkeiten in der Sprachdefinition
- „Verbesserungen“, welche manche Compilerhersteller über die Sprachdefinition hinaus in ihr Produkt einbauen

Fehlermeldungen (Errors): Detektiert der Compiler schwerwiegende Abweichungen von den Syntaxregeln (s.u.), erzeugt er eine entsprechende Meldung und bricht den Vorgang ab. Es besteht jedoch die Möglichkeit, dass ein und derselbe Syntaxfehler mehrere Fehlermeldungen generiert, so dass es sich in der Praxis empfiehlt, bei der Korrektur mit dem ersten Fehler zu beginnen.

Warnungen: Viele Compiler sind in der Lage, Warnungen zu generieren. Hierbei handelt es sich um Hinweise auf möglich Probleme an den Programmentwickler.

2.5.3 Der Debugger

Bei der Programmieren werden Fehler begangen. Syntaxfehler entdeckt der Compiler. Eine andere Kategorie von Fehlern sind **Laufzeitfehler**. Unter **Laufzeit** versteht man, wenn die ausführbare Datei vom Mikroprozessor tatsächlich ausgeführt wird. Laufzeitfehler führen oft zu einem „Programmabsturz“ oder in ungünstigen Fällen auch zum „Computerabsturz“. Sie in i.d.R. schwieriger zu entdecken. Beim Debugger handelt es sich um eine spezielle SW, welche verschiedene Verfahren zur Entdeckung oder Analyse von Laufzeitfehlern anbietet.

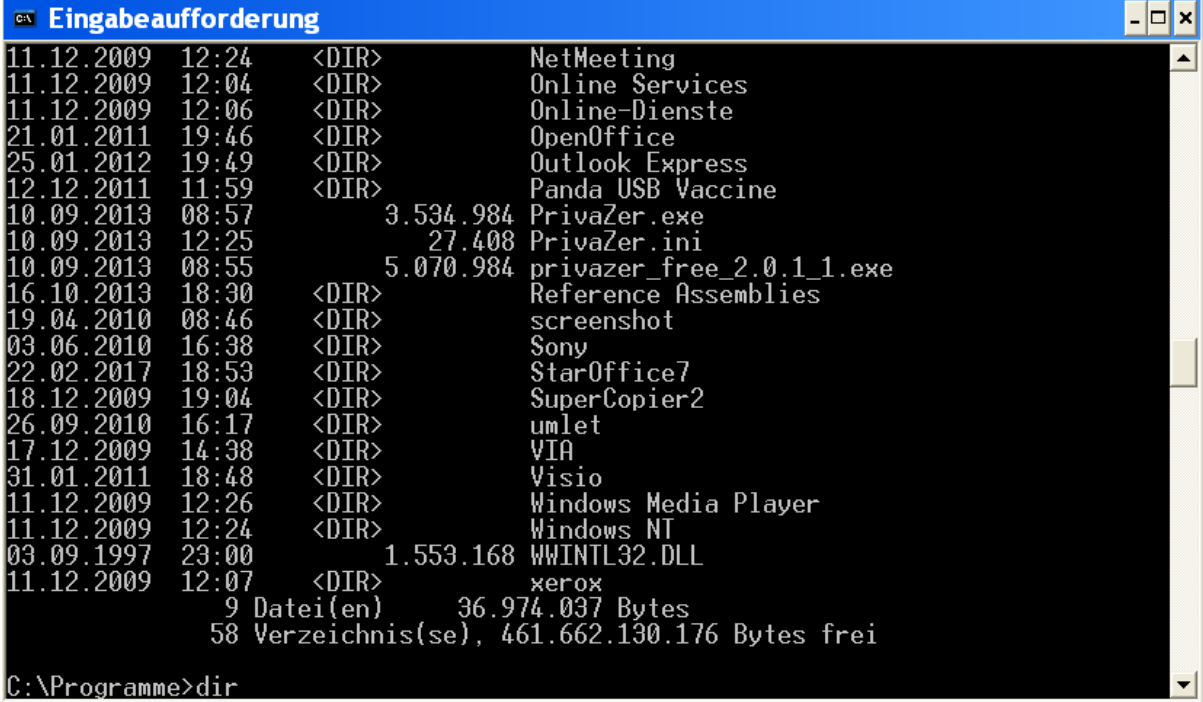
2.5.4 Integrierte Entwicklungsumgebung (IDE)

Moderne Code-Entwicklung verwendet integrierte Entwicklungsumgebungen, welche die Funktionen von Text-Editor, Compiler, Linker und Debugger unter einer einheitlichen Bedienoberfläche zusammenfassen. Häufig ist dies ergänzt durch eine eingebaute Projektverwaltung, welche sich z. B. automatisch darum kümmert, dass all jene Quellcodedatei vor dem linken neu kompiliert werden, welche verändert worden sind. Auch bieten die IDE's häufig automatische Code-Generatoren an, welche z. B. den Rumpf des Hauptprogrammes (s.u) automatisch erzeugen (Spart Tipp-Arbeit)

2.5.5 Die Konsolanwendung

Die Programmiersprache C/C++ ermöglicht nicht die Erstellung von Programmen mit grafischer Oberfläche. D. h. Die hiermit erstellten Programme sind beschränkt auf Eingaben über die Tastatur und textuelle Ausgaben auf dem Bildschirm. Man nennt solche Anwendungen

Konsolanwendungen. Betriebssysteme mit grafischer Oberfläche (wie z.B. Windows) erzeugen für die Ausführung solcher Anwendungen ein (simuliertes) Konsolenfenster. In diesem Fenster können direkt Kommandos des BS eingegeben werden, das Kommando *dir* listet z.B. den Inhalt des aktuellen Verzeichnisses auf. Die folgende Abbildung zeigt beispielhaft das Konsolenfenster:



```
C:\> dir C:\Programme

11.12.2009 12:24 <DIR> NetMeeting
11.12.2009 12:04 <DIR> Online Services
11.12.2009 12:06 <DIR> Online-Dienste
21.01.2011 19:46 <DIR> OpenOffice
25.01.2012 19:49 <DIR> Outlook Express
12.12.2011 11:59 <DIR> Panda USB Vaccine
10.09.2013 08:57 3.534.984 PrivaZer.exe
10.09.2013 12:25 27.408 PrivaZer.ini
10.09.2013 08:55 5.070.984 privazer_free_2.0.1_1.exe
16.10.2013 18:30 <DIR> Reference Assemblies
19.04.2010 08:46 <DIR> screenshot
03.06.2010 16:38 <DIR> Sony
22.02.2017 18:53 <DIR> StarOffice7
18.12.2009 19:04 <DIR> SuperCopier2
26.09.2010 16:17 <DIR> umlet
17.12.2009 14:38 <DIR> VIA
31.01.2011 18:48 <DIR> Visio
11.12.2009 12:26 <DIR> Windows Media Player
11.12.2009 12:24 <DIR> Windows NT
03.09.1997 23:00 1.553.168 WWINTL32.DLL
11.12.2009 12:07 <DIR> xerox
          9 Datei(en)      36.974.037 Bytes
        58 Verzeichnis(se), 461.662.130.176 Bytes frei

C:\Programme>dir
```

3 Grundlagen von C

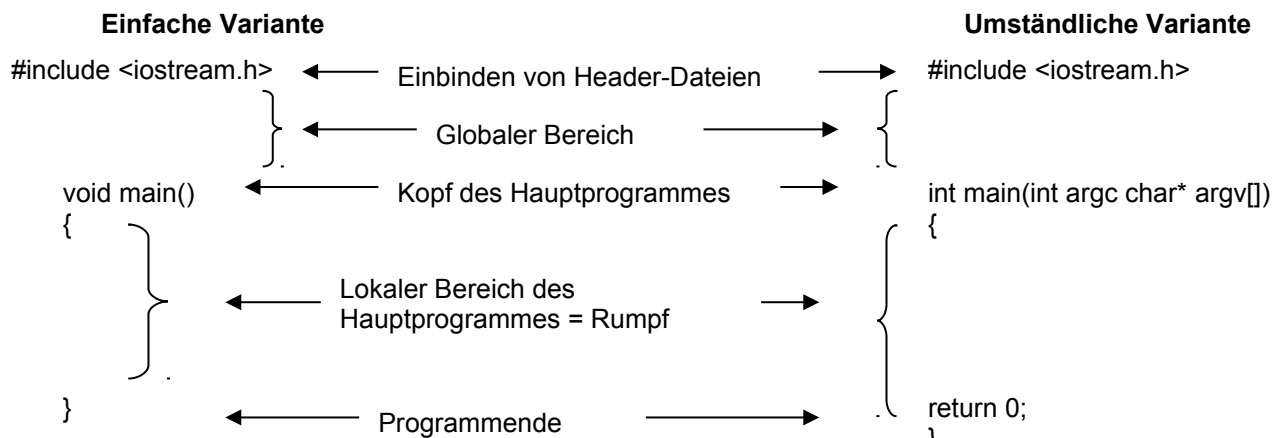
C und C++ sind die Programmiersprachen, welche aktuell am verbreitetsten sind. Einer der Gründe hierfür ist sicher, dass sie einen schnellen Maschinencode erzeugen. Einer anderer Grund kann darin liegen, dass so gut wie alle Ideen und Konzepte, welche im Verlauf der Entwicklung der verschiedenen Programmiersprachen entwickelt wurden, in C und insbesondere in C++ in irgend einer Form mit eingeflossen sind. Ergebnis ist eine ziemlich komplizierte Sprachdefinition, wie folgendes Zitat zeigt:

**Auf jeden C++ - Programmierer, der alle
Feinheiten der Sprache begriffen hat kommen
etwa 1 Million C++ - Programmierer, die mit der
extrem komplexen Sprachdefinition nicht
zurechtkommen**

T Lauer in Internet, Verlag MuT

Die nachfolgenden erläuterten Grundlagen gelten ebenso für C++.

Grundaufbau eines C Programmes



- Die Ausführung eines C- / C++ -Programmes beginnt immer in der main –Zeile, d.h am Kopf des Hauptprogrammes. Die Zeichenfolge `main` darf in jedem C Programm nur 1 mal vorkommen
- Der Rumpf des Hauptprogrammes enthält die Anweisungen des Programmes (Deklarationen und ausführbare Anweisungen)

In nachfolgenden Text wird bei allen Beispielen die einfache Variante verwendet werden. Viele Lehrbücher zu C / C++ verwenden in ihren Beispielen jedoch die andere Variante. Auch gibt es Compiler, welche obligatorisch die umständliche Variante verlangen. Beides ist sachlich nicht gerechtfertigt.

3.1 Allgemeine C-Syntaxregeln

- Jede Anweisung in C endet mit ;
- Der Programmcode wird immer von links nach rechts und von oben nach unten gelesen.
- Leerstellen und Zeilensprünge haben keine Bedeutung (Dienen nur der besseren Lesbarkeit für den Programmierer). Von dieser Regel gibt es einige Ausnahmen, welche gesondert behandelt werden
- Schlüsselwörter werden durch Leerstellen abgegrenzt.
- Jede Art von Klammern treten immer paarweise auf, d.h. zu jeder öffnenden Klammer (, { , [gibt es eine zugehörige schliessende Klammer] , } ,) !

3.2 Basiskomponenten eines C-Programmes

3.2.1 Kommentare

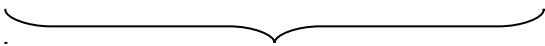
Da reale Programme i.d.R. sehr umfangreich ist, d.h. sehr viele (> 1000) Anweisungen umfasst und der Quellcode in vielen Fällen nicht einfach zu lesen, dienen Kommentare dazu, in den Quellcode für den Programmierer bestimmten erläuterten Text einzufügen. Dieser Text wird durch den Compiler ignoriert, hat somit auf die Programmausführung keine Auswirkung. Zu diesem Zweck muss dieser Text entsprechend als Kommentar gekennzeichnet werden. In C++ gibt es folgende Kommentare:


- **Zeilenkommentar**
- **Blockkommentar**

In reinem C gibt es nur den Blockkommentar.

2.5.5.1 Der Zeilenkommentar

Der Kommentar steht in einer Zeile. Er beginnt mit der Zeichenfolge `//` und endet mit dem Zeilensprung

```
// Dies ist ein Zeilenkommentar
    
      

int variable1; //Deklaration der Variablen
```

2.5.5.2 Blockkommentar

Dieser beginnt mit der Zeichenfolge `/*` und endet (in Leserichtung!) mit der Zeichenfolge `*/`. Der Blockkommentar kann sich über mehrere Zeilen erstrecken. Beispiele:

```
int variable1; /*Kommentar*/ int variable2;

/*  Kommentarbeginn
viele Erklärungen
Kommentarende */
```

3.3 Variablen

In der Formel (Berechnung des Kreisumfanges) $u = 3.1415 \cdot 2 \cdot r$

Sind u und r Variablen und 3.1415 ist eine Zahlenkonstante. \cdot ist ein Operator. r kann beliebige Werte annehmen und damit kann u dann berechnet werden. Variablen beim Programmieren haben eine entsprechende Funktion, allerdings entspricht jeder Variablen auch ein fester Platz im Arbeitsspeicher (RAM). D.h. mittels des Namens einer Variablen wird auf den entsprechenden Ort im RAM zugegriffen.

Bevor Variablen verwendet werden können müssen sie bekannt gemacht werden. Dies heisst **Variablen-Deklaration**. Variablen können beliebige Namen erhalten, es müssen jedoch die folgenden Regeln eingehalten werden:

- Alle alphanumerischen Zeichen einschliesslich '_' (aber keine Umlaute)
- Länge ist beliebig, signifikant sind in C nur die ersten 31 Zeichen (In C++ nicht begrenzt)
- Nicht mit einer Ziffer beginnen
- Es wird zwischen Groß- und Kleinschreibung unterschieden
- Verwendung von Schlüsselwörtern verboten!
- Möglichst aussagekräftige Namen wählen
- Jeder Variablenname darf in einem Gültigkeitsbereich nur 1 Mal verwendet werden. In verschiedenen, auch untergeordneten Gültigkeitsbereichen dürfen sich Variablennamen wiederholen

Da die Verwendung von Schlüsselwörtern für Variablen verboten ist, folgt hier die Liste aller C/C++ - Schlüsselwörter, ihre jeweilige Bedeutung wird erst im weiteren Verlauf deutlich werden.

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

Da beim Programmieren mit verschiedenen Arten von Daten operiert wird, muss auch festgelegt werden, welche Arten von Daten eine Variable aufnehmen kann. Zu diesem Zweck sind in C die in folgender Tabelle aufgelisteten **elementaren Datentypen** definiert.

Datentyp	Bit	min Wert	max Wert	Genauigkeit
int	16(32)	-32768 (16Bit)	32767 (16Bit)	
unsigned int	16(32)	0	65535 (16Bit)	
short int	16	-32768	32767	
unsigned short int	16	0	65535	
Long int	32(64)	-2147483648 (32Bit)	2147483647 (32Bit)	
unsigned long int	32	0	4294967295 (32Bit)	
float	32	ca +/- 3.4E-38	ca +/- 3.4E+38	>= 6 Ziffern
double	64	ca +/- 1.7E-308	ca +/- 1.7E+308	>= 10 Ziffern
long double	80	ca +/- 1.2E-4932	ca +/- 1.2E+4932	>= double
char	8	-128	127	
unsigned char	8	0	255	

Für den Bit-Code von Werten des Typs `char` wird das **ASCII** (American Standard Code for Information Interchange) Code-System verwendet. Die ASCII-Tabelle kann im Internet gefunden

werden. Z. B. hat das Zeichen 'A' in ASCII den Code 65 (Das bedeutet, die Bitfolge von 'A' entspricht der Zahl 65), und das Zeichen 'a' hat den ASCII Code 97.

In C++ wurde ein weiterer elementarer Datentyp `bool` eingeführt. Ihm können nur Wahrheitswerte (`false`, `true`) zugewiesen werden. `false` wird durch 0, und `true` durch 1 repräsentiert.

3.3.1 Die Variablen-Definition

Die Variablen-Deklaration einer Variablen legt folgendes fest:

- Name der Variablen
- Datentyp der Variablen

Zusätzlich wird durch die **Variablen-Definition** noch der **erforderliche Speicherplatz** für die Variable im Arbeitsspeicher reserviert, so dass ihr ein Wert zugewiesen werden kann.

Obwohl Variablen-Definitionen im Programm stehen, zählen sie zu den **nicht ausführbaren Anweisungen** (da sie während des Programmlaufs nicht mehr ausgeführt werden).

Beispiele:

```
int i1;           - eine Variable vom Typ int mit dem Namen i1 wird definiert.
float f_1, _f2;   - 2 Variablen vom Typ float mit Namen f_1, _f2 werden definiert.
```

Regeln:

- In einer Variablen-Definition können beliebig viele Variablen desselben Typs mit verschiedenen Namen definiert werden, die einzelnen Variablen-Namen sind durch Komma getrennt.
- Variablen verschiedenen Typs können nicht in einer Definition definiert werden.
- Jede Variable muss vor ihrer ersten Verwendung deklariert werden:

Beispiel:

```
int i1, i2;
char c1, c2, c3;
float f1, char c3; ← Fehler !
cin >> i1;          ← Erste Verwendung von i1
```

Es ist zu unterscheiden:

- **Globale Variable:** Die Definition steht im globalen Bereich des Quellcodes
- **Lokale Variable:** Die Definition steht in einem lokalen Bereich des Quellcodes, z.B. innerhalb des Rumpfs des Hauptprogrammes

3.3.2 Initialisierung

Beim Starten eines Programmes wird der gesamte Maschinencode des Programmes von der Festplatte in den Arbeitsspeicher kopiert. In diesem Bereich (Stack genannt) liegt dann auch der Speicherort für die Variablen des Programmes. Nun war dieser Bereich des Arbeitsspeichers zuvor irgendwann durch ein anderes Programm belegt, d.h. es besteht die Möglichkeit, dass auf den Bytes noch Werte eingetragen sind, welche ein früheres Programm dort abgelegt hat. Dies hängt i.d.R. vom Betriebssystem ab. Es gibt Betriebssysteme, welche alle nicht mehr einem Programm zugewiesenen

Bytes des RAM grundsätzlich in einen Grundzustand versetzen (z.B. alle Bits auf 0 oder 1), es gibt auch Betriebssysteme, welche aktuell unbenutzten RAM-Speicher in seinem Zustand belassen. Aus der Sicht der höheren Programmiersprache muss somit davon ausgegangen werden, dass die Speicherorte der Variablen nach dem Laden des Programmes in einem undefinierten Zustand sind. C geht nun in folgender Weise vor:

- Globale Variablen werden generell mit 0 initialisiert
- Lokale Variablen werden nicht initialisiert

Unter **Initialisierung** versteht man die Zuweisung eines **Anfangswertes**. Es ist somit eine Aufgabe des Programmierers, den Variablen Anfangswerte zuzuweisen. Dies kann auch bei globale Variablen erforderlich sein, wenn der Anfangswert von 0 abweichen soll.

In C kann die Initialisierung gleich in der Variablen-Definition vorgenommen werden, wie folgendes Beispiel zeigt:

```
int ivar1=-5, ivar2 =77, ivar3=1000;
float fvar1=1.111, fvar2=-0.234;
char ch1='0';
```

Sollen mehrere Variablen mit demselben Wert initialisiert werden, so ist folgende Syntax nicht möglich:

```
int ivar1=ivar2=ivar3=1000; //Falsch
```

3.3.3 Konstanten und Literale

Steht in einer Programmanweisung ein Wert (Zahl, Zeichen oder Zeichenfolge) so spricht man von einer **Konstanten** oder einem **Literal**. Sie dienen dazu, an Variablen z.B. im Rahmen der Initialisierung Werte zuzuweisen, oder es handelt sich um Texte für die Monitorausgabe.

2.6.7.1 Ganzzahl - Konstanten (Literale)

- Eine Oktalzahl wird durch vorangestelltes **0** gekennzeichnet
- Hexadezimalzahl wird durch vorangestelltes **0X** oder **0x** gekennzeichnet
- Suffix **L** oder **l** kennzeichnen eine `long` Zahl
- Suffix **U** oder **u** kennzeichnen eine `unsigned int` Zahl

Beispiele:

```
30 = 036 (oktal) = 0x1E (hex)
unsigned int wert = 456U;
long int neu = 321L;
```

2.6.7.2 Gleitkomma – Konstanten (Literale)

- Suffix **F** oder **f** kennzeichnen eine `float` Zahl
- Suffix **L** oder **l** kennzeichnen eine `long double` Zahl
- Keine Angabe: Automatisch `double`

Beispiel:

```
long double fwert=0.345L;
```

Steht bei Gleitkomma-Konstanten kein Suffix, so wird durch den Compiler standardmässig der Typ `double` angenommen. Dies führt zu Compilerwarnungen in Anweisungen wie der folgenden:

```
float f_var=1.1; //Typ links = float, Typ rechts = double
```

Diese Warnungen lassen sich vermeiden, wenn man Folgendes schreibt:

```
float f_var=1.1f;
```

2.6.7.3 Zeichen und Zeichenketten

Notation für einzelne Zeichenkonstanten:

```
char zeichen = 'a';
```

Notation für einzelne Zeichenkettenkonstanten:

```
cout<<"Zeichenkette";  
char dateiname[] ="C:\eigene\uebung.cpp";
```

Regeln:

- Einzelne Zeichen stehen in Hochkomma
- Keine Sonderzeichen (Umlaute, ß) verwenden !
- Zeichenketten stehen in Anführungszeichen
- **Zeichenketten** dürfen nicht durch Zeilensprung getrennt werden

Zeichenketten heissen auf englisch **string**

3.3.4 Die const-Deklaration

Eine Variable kann in der Definitionsanweisung als konstant deklariert werden. Das bedeutet, dass ihr Wert nachfolgend nicht mehr geändert werden kann. Beispiele:

```
const float PI=3.1415;  
const short MWST=19; // MwSt in % !
```

Es gelten folgende Regeln:

- Die Zuweisung eines Wertes an eine als `const` definierte Variable muss in der Definition erfolgen (in Form einer Initialisierung). Eine spätere Änderung ist nicht möglich
- Ansonsten können `const` Variablen wie andere Variablen verwendet werden
- Wird in der `const` - Definition kein Datentyp angegeben, so gilt standardmässig der Typ `int`
- `const` - Definitionen sind in C++ möglich, jedoch nicht in reinem C

Es ist üblich, Variable, welche als `const` deklariert sind, durch die Schreibweise in Grossbuchstaben hervorzuheben.

3.4 Gültigkeitsbereiche

Die Hauptaufgabe der **Gültigkeitsbereiche** besteht darin, die Bereiche des Quellcodes festzulegen, in welchen Variablennamen eindeutig sein müssen. Das bedeutet, dass jeder Variablenname innerhalb eines Gültigkeitsbereiches nur für 1 Variable verwendet werden kann. Gleichzeitig bestimmen sie auch die Lebensdauer von Variablen. Eine Variable existiert nur so lange im RAM, wie die Ausführung in ihrem Gültigkeitsbereich ist. D. h. verlässt die Ausführung den Gültigkeitsbereich in welchem eine Variable definiert ist, so verschwindet diese Variable!

Lokale Gültigkeitsbereiche werden durch je ein paar Klammern { . . . } festgelegt. Eine Ausnahme hiervon ist nur die `enum`-Deklaration (s. weiter unten)

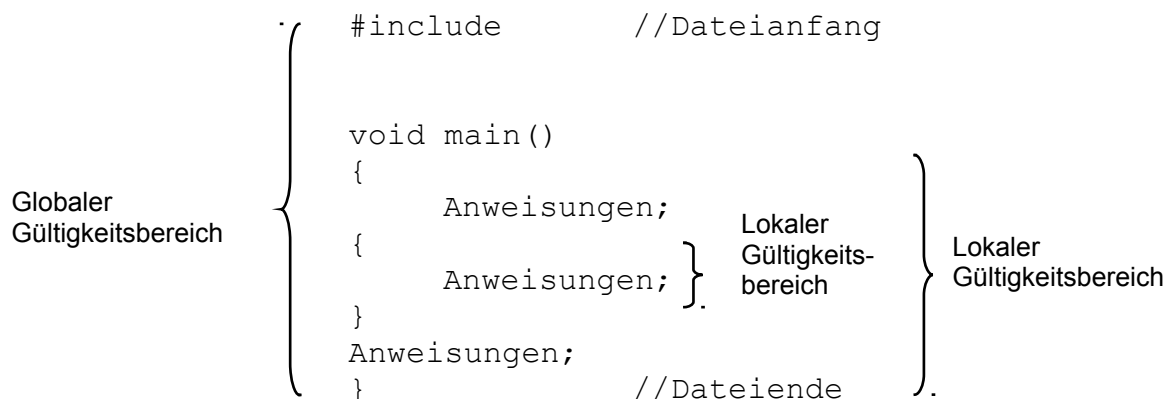
Der Codebereich ausserhalb { . . . } heisst **globaler Gültigkeitsbereich**. Gültigkeitsbereiche können wie in folgender Abbildung dargestellt, ineinander geschachtelt werden. Es ist nützlich, in

Quellcode die Gültigkeitsbereiche durch Einrückungen kenntlich zu machen. Auch dies stellt die Abbildung dar.

Gültigkeitsbereiche können ineinander geschachtelt sein, wie das folgende Beispiel verdeutlicht. Dann gilt, dass in einem untergeordneten Gültigkeitsbereich alle Variablen der übergeordneten Gültigkeitsbereiche sichtbar sind, d.h. dass auf diese zugegriffen werden kann.

Folgerung:

- Die Definition einer globalen Variablen steht im globalen Gültigkeitsbereich
- Die Definition einer lokalen Variablen steht in einem lokalen Gültigkeitsbereich



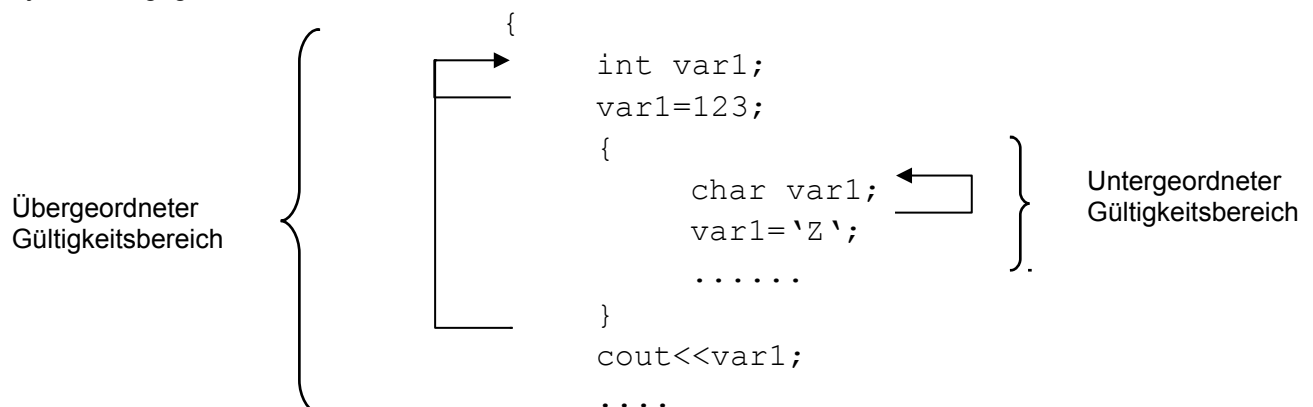
Es gilt die Regel:

Eine Variable ist innerhalb ihres Gültigkeitsbereiches bekannt (und verwendbar), das umfasst auch die untergeordneten Gültigkeitsbereiche

Damit kann sich folgendes Problem ergeben: Wenn in einem Gültigkeitsbereich ein Variablenname verwendet wird und in einem untergeordneten Gültigkeitsbereich derselbe Variablenname (für eine andere Variable!), auf welche der Variablen wird dann zugegriffen. Hier gilt die Regel:

Zugriff erfolgt immer auf jene Variable, die in dem Gültigkeitsbereich definiert ist, in welchem auch der Zugriff erfolgt

Die folgende Darstellung veranschaulicht den Zusammenhang, die Pfeile zeigen, auf welche Variable jeweils zugegriffen wird:



}

3.5 Die einfachen ausführbaren Anweisungen

Unter **ausführbaren Anweisungen** versteht man Anweisungen, welche während der Programmausführung zu Änderungen des Zustandes führen. Die einfachsten sind in der folgenden Tabelle zusammengestellt:

Anweisung	Bewirkte Zustandsänderung
Ausgabe	Es werden Daten auf dem Monitor dargestellt
Eingabe	Daten, welche der Anwender eingibt, werden RAM gespeichert
Zuweisung	Eine Variable ändert ihren Wert, d.h. es werden Daten auf dem der Variablen entsprechenden Ort im RAM gespeichert

Generell gilt, dass diese Anweisungen beliebig oft und in beliebiger Reihenfolge verwendet werden können. (Natürlich muss dabei etwas sinnvolles ausgeführt werden!)

Hinweis: Die nachfolgend vorgestellten Ein- und Ausgabeanweisungen können nur in C++ verwendet werden!

3.5.1 Monitor - Ausgaben in C++

Der einfachste Weg für Monitor – Ausgaben ist die cout-Anweisung wie in folgenden Beispielen:

```
cout<<"Ein einfaches Programm";
```

Ergebnis: Der Text *Ein einfaches Programm* erscheint auf dem Monitor. << heisst der **Ausgabeoperator**.

Somit gilt: Text, welcher hinter dem Ausgabeoperator in Anführungszeichen aufgeführt wird, wird auf dem Monitor ausgegeben

```
cout<<"Ein einfaches Programm"<<endl<<endl;
```

Ergebnis: Der Text *Ein einfaches Programm* erscheint auf dem Monitor, danach wird ein Zeilensprung ausgeführt

```
variable1=55;  
cout<<variable1;
```

Ergebnis: Der Wert, den die angegebene Variable hat (im Beispiel 55), erscheint auf dem Monitor. Dabei ist es unerheblich, von welchem elementaren Datentyp die angegebene Variable ist.

```
variable1=55;  
cout<<variable1<<endl;
```

Der Wert, den die angegebene Variable hat (im Beispiel 55), erscheint auf dem Monitor, danach wird ein Zeilensprung ausgeführt. Wird der Ausgabeoperator in einer Anweisung mehrfach verwendet, so nennt man dies **Verkettung**.

```
variable1=55;  
cout<<"Wert von variable1="<<variable1<<endl;
```

Die Ausgabe sieht folgendermassen aus: Wert von variable1=55
Danach wird noch ein Zeilensprung ausgeführt

```
variable1=55; variable2=-55.5;
cout<<variable1<<variable2<<endl;
```

Die Werte der angegebenen Variablen erscheinen auf dem Monitor, danach wird ein Zeilensprung ausgeführt. Zwischen den Werten wird kein Abstand erzeugt!

```
variable1=55; variable2=-55.5;
cout<<variable1<<"          "<<variable2<<endl;
```

Die Werte der angegebenen Variablen erscheinen auf dem Monitor, danach wird ein Zeilensprung ausgeführt. Vor der Ausgabe des Wertes der 2. Variablen springt die Ausgabe zur nächsten Tabulatormarke, so dass ein Abstand entsteht

Hinweis: Vor der Verwendung von `cout` muss am Beginn des Quelltextes folgende Anweisung stehen:

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

3.5.2 Eingaben in C++

Eingaben durch den Anwender erfolgen über die Tastatur. Dazu müssen entsprechende Anweisungen im Programmtext stehen. Ist die Programmausführung bis zu einer Eingabeanweisung fortgeschritten, erscheint ein dem Bildschirm ein blinkender Cursor und die Ausführung wartet, bis der Benutzer einen Wert eingegeben hat. Während der Benutzer Werte eingibt, erscheinen diese auf dem Monitor. Der Benutzer muss die Eingabe abschliessen durch drücken der Enter-Taste. Danach wird die Programmausführung fortgesetzt. Während der

```
float variable1;
cin>>variable1;
```

Ergebnis: Hat der Benutzer die Werteingabe mit der Enter-Taste abgeschlossen, so hat `variable1` den vom Benutzer angegebenen Wert. `>>` heisst der **Eingabeoperator**.

Wichtig: Die Eingabe-Anweisung wie in obigem Beispiel ist unabhängig vom Datentyp der Variablen. Allerdings muss der Benutzer Werte eingeben, welche zu dem Datentyp der Variablen passen. In obigem Beispiel dürfen somit keine Zeichen eingegeben werden. Gibt der Benutzer trotzdem Zeichen ein, so erfolgt eine fehlerhafte Programmausführung!

Aus diesem Grunde ist es sinnvoll, dem Benutzer vor der Eingabe sinnvolle Informationen über die einzugebenden Daten auf den Monitor zu geben, wie in folgendem Beispiel:

```
float mwst, red_mwst;
cout<<"Eingabe normaler MWst-Satz : " ;cin>>mwst;
cout<<"Eingabe reduzierter MWst-Satz : " ;cin>>red_mwst;
```

Auch der Eingabeoperator kann verkettet werden. Dies zeigt folgendes Beispiel:

```
cin>>mwst>>red_mwst;
```

Allerdings ist dies wenig zu empfehlen, da ungünstig für den Benutzer!

Hinweis: Vor der Verwendung von `cout` muss am Beginn des Quelltextes folgende Anweisung stehen:

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

3.5.3 Die Zuweisung

Das folgende Beispiel zeigt eine Zuweisung: `variable1 = 123.4;`

Es wird der **Zuweisungsoperator** = verwendet. Die Variable, welche links davon steht ändert ihren Wert. Programmobjekte, welche links vom Zuweisungsoperator stehen können, heissen **L-Value(L-Wert)**. Jede Variable ist ein L-Wert.

Folgende Anweisung ist falsch und führt zu einer Fehlermeldung des Compilers:

```
123.4 = variable1;
```

Soll derselbe Wert an mehrere Variable zugewiesen werden, so ist folgende Zuweisung möglich:

```
variable1 = variable2 = variable3 =123.4;
```

Natürlich sind auch Zuweisungen der folgenden Art möglich:

```
variable2 = variable1;
```

Nach Ausführung haben beide Variable denselben Wert.

Schreibzugriff: Die Variable links vom Zuweisungsoperator ändert ihren Wert, d.h. an dem ihr zugeordneten Speicherort im RAM wird der Wert eingetragen. In obigem Beispiel erfolgt somit ein Schreibzugriff auf `variable2`;

Lesezugriff: Die Variable(n) rechts vom Zuweisungsoperator ändert (ändern) ihren Wert nicht, ihr Wert wird lediglich abgelesen. In obigem Beispiel erfolgt somit ein Lesezugriff auf `variable1`;

Es ist zu beachten, dass das Symbol = hier nicht nur Gleichheit von linker und rechter Seite bedeutet, sondern dass sich der Zustand des RAM ändert. Aus diesem Grunde ist es korrekt, von einem Operator zu sprechen.

2.8.7.1 Typsicherheit

Es ist ein wichtiges Element der Entwicklung zuverlässiger Programme, dass an eine Variable nur Werte zugewiesen werden, welche zu ihrem Datentyp passend sind. D.h. es sollen Fehler der folgenden Art vermieden werden:

```
char ch1='A';  
int i1=0;  
i1=ch1;
```

Denn selbst, wenn es zu einer lauffeit-fehlerfreien Programmausführung kommt – was nicht sicher ist – werden die gelieferten Resultate falsch sein. Aus diesem Grunde wird in C eine Typprüfung durch den Compiler ausgeführt. Dieses Konzept wird als **Typisierung** bezeichnet. Stimmen die Datentypen rechts- und linksseitig des Zuweisungsoperators nicht überein, wird eine Fehlermeldung oder eine Warnung erzeugt.

Eine Warnung wird in den Fällen erzeugt, wo eine **automatische Typwandlung**, auch **implizite Typwandlung** genannt, möglich ist. Diese funktioniert in den Fällen

- Von `float` nach `int`: Die Nachkommastellen der float-Zahl gehen verloren

- Von `int` nach `float`: Die Nachkommastellen der `float`-Zahl enthalten den Wert 0

Beispiel:

```
float fvar=0.999;
int ivar;
ivar = fvar;    //Compilerwarnung
cout<<ivar;    //Ausgabe ist 0 !
```

Es gibt auch eine automatische Typwandlung `char` \Leftrightarrow `int` bzw `char` \Leftrightarrow `float`, die Resultate sind in der Regel jedoch nicht sinnvoll. Das 2. Beispielprogramm am Ende dieses Kapitels demonstriert jedoch, dass es durchaus Anwendungen für diese Typwandlung geben kann

3.5.4 Typwandlung

In der Programmier-Praxis kommt es immer vor, dass Zuweisungen von Werten vorgenommen werden müssen, wobei rechts- und linksseitig des Zuweisungsoperators verschiedene Datentypen verwendet werden. So kann z.B die Berechnung von `y = n * x` erforderlich sein, wobei `n` eine Ganzzahl sein soll, welche von 0 bis n_{\max} variiert und `x` eine beliebige Kommazahl. Das Ergebnis ist natürlich ebenfalls eine Kommazahl. Auf der Basis der implizite Typwandlung kann dieser Ausdruck programmiert werden, generiert jedoch – wie dargelegt – Compilerwarnungen.

Die Compilerwarnungen lassen sich vermeiden mittels der **expliziten Typwandlung**, wie in folgendem Beispiel.

```
float fvar=0.999;
int ivar;
ivar = int(fvar);    // keine Compilerwarnung
cout<<ivar;          //Ausgabe ist 0 !
```

Für die expliziten Typwandlung stehn in C folgende Alternative Syntaxvarianten zur Verfügung:

```
ivar = int(fvar);    //Syntaxvariante 1
ivar = (int)fvar;    //Syntaxvariante 2
```

Neben der Vermeidung von Compilerwarnungen gibt es auch Situationen – Beispiele hierfür werden im weiteren Verlauf auftreten – von die expliziten Typwandlung notwendig erforderlich ist.

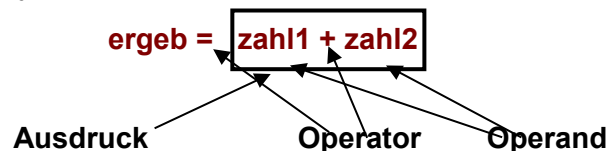
3 Operatoren

In der Mathematik dienen **Operatoren** (+, -, *, /) dazu, Ausdrücke zu formulieren, welche Zahlen zu neuen Zahlen verbinden, kurz um Rechenformeln zu notieren, wie in folgendem Beispiel (Berechnung Verkaufspreis VKP aus Einkaufspreis EKP und MwSt):

$$\text{VKP} = \text{EKP} * 1,19$$

Selbstverständlich müssen auch in der Programmiertechnik in ähnlicher Weise Rechenoperationen möglich sein. Die hierfür benötigten Operatoren werden in diesem Kapitel vorgestellt. Daneben gibt es noch eine grosse Anzahl weiterer Operatoren, deren Bedeutung und Anwendung teilweise jedoch erst in den weiteren Kapiteln erläutert werden kann. Hier sind sie der kompletten Übersicht wegen aufgeführt.

Ausdrücke bestehen aus Operatoren und **Operanden**. Operatoren bestimmen, „Was mit den Operanden gemacht wird“



Enthält ein **Ausdruck** mehrere Operatoren, so muss festgelegt werden, in welcher Reihenfolge die entsprechenden Operationen ausgeführt werden.

Beispiel: Regel „Punkt vor Strich“, in der Mathematik.

Hier wird das gesteuert über den Rang eines Operators. In den nachfolgenden Tabellen ist für jeden Operator auch sein Rang angegeben. Ein kleiner Rang-Wert für einen Operator bedeutet, dass er vor einem anderen Operator mit höherem Rang-Wert ausgeführt wird.

Sollten Abweichungen von der durch die Rang-Werte bestimmten Ausführungsreihenfolge erforderlich sein, so wird dies über Klammern gesteuert, welche genau wie in mathematischen Formeln gesetzt werden:

Beispiel: $a = b / (c + 10)$ als Code: `a = b / (c + 10);`

Generell werden Operatoren unterschieden nach folgenden Eigenschaften unterschieden

- Assoziativität: Abarbeitung von links wie von rechts möglich
- Priorität (Rang): Beispiel 'Punkt vor Strich'
- Anzahl der Operanden (Unär oder binär)
- Funktion

Unärer Operator: Der Operator hat nur einen Operanden

Binärer Operator: Der Operator hat 2 Operanden

Bzgl ihrer Funktion sind folgende Gruppen von Operatoren zu unterscheiden:

- Arithmetische Operatoren
- Vergleichs - Operatoren
- Logische Operatoren
- Zuweisungsoperatoren
- Postfix - Operatoren
- Zeigeroperatoren

3.5 Arithmetische Operatoren

Die **arithmetischen Operatoren** entsprechen den bekannten Operatoren für die Grundrechenarten

Operator	Bedeutung	Rang	int-Operand	float-Operand	Beispiel
+	Addition	5	X	X	a + b
-	Subtraktion	5	X	X	a - b
*	Multiplikation	4	X	X	a * b
/	Division	4	X	X	a / b
%	Modulo	4	X		a % b

Haben die Operanden unterschiedliche Datentypen, dann wird der Typ des Operanden mit niedrigerem Datentyp in den des Operanden mit mächtigerem Datentyp gewandelt.

Der **Modulo – Operator** liefert den Rest einer Ganzzahl-Division : $15 \% 4 = 3$

3.6 Zuweisungsoperatoren

Neben dem bereits bekannten = gibt es noch weitere Zuweisungsoperatoren, wie in folgender Tabelle dargestellt:

Operator	Bedeutung	Rang	Beispiel
=	Zuweisung	15	a = b
+=	Zuweisung nach Addition	15	a += b
-=	Zuweisung nach Subtraktion	15	a -= b
*=	Zuweisung nach Multiplikation	15	a *= b
/=	Zuweisung nach Division	15	a /= b
%=	Zuweisung nach Modulo	15	a %= b
<<=	Zuweisung nach bitweisem Linksschieben	15	a <<= b
>>=	Zuweisung nach bitweisem Rechtsschieben	15	a >>= b
&=	Zuweisung nach bitweisem Und	15	a &= b
=	Zuweisung nach bitweisem Or	15	a = b
^=	Zuweisung nach bitweisem Xor	15	a ^= b

Beispiele: $a = b = c = 3;$

a += 2;	entspricht	a = a + 2;	→ Wert von a = 5
a %= 2;	entspricht	a = a % 2;	→ Wert von a = 1
a+=b*=c;	entspricht	a=a+b*c;	→ Wert von a = 12

3.7 Vergleichsoperatoren

In der SW-Erstellung müssen häufig Vergleiche von Werten durchgeführt werden. Diesem Zweck dienen die Vergleichsoperatoren

Operator	Bedeutung	Rang	Beispiel
<	Kleiner als	7	a < b
<=	Kleiner oder gleich	7	a <= b
>	Größer als	7	a > b
>=	Größer oder gleich	7	a >= b
==	Gleich	8	a == b
!=	Ungleich	8	a != b

Regeln:

- Die Operanden müssen vom gleichen Typ sein!
- Das Ergebnis einer Vergleichsoperation ist wahr (true) oder falsch (false)
- Wahr wird dargestellt durch 1
- Falsch wird dargestellt durch 0

Beispiele: `int a = 2 , b = 3, c, d;`

`c = a < b;` \rightarrow Wert von c = 1

`d = a == b;` \rightarrow Wert von d = 0

Achtung: Es ist der Unterschied in der Schreibweise des Zuweisungsoperators = und des Vergleichsoperators == zu beachten!

3.8 Postfix- Operatoren

Operator	Bedeutung	Rang	Beispiel
++	Inkrement	1 oder 2	<code>a++; ++a;</code>
--	Dekrement	1 oder 2	<code>a--; --a;</code>
()	Funktionsaufruf	1	<code>sin()</code>
[]	Arrayelement	1	<code>feld[3]</code>
.	Feld von Struktur oder Union	1	<code>tag.dat</code>
->	Zeiger	1	<code>point->element</code>
(type)	explizite Typwandlung	2	<code>(int) f_var</code>

Der Unterschied zwischen Prefix und Postfix-Variante der Dekrement - und Inkrement – Operatoren soll an folgendem Beispiel erläutert werden:

Prefix:

```

a = 1;
a++;           //Entspricht a=a+1;
cout<<a;        $\rightarrow$            Ausgabe: 2
b = 21;
a + = ++b;    //zuerst b inkrementieren, dann Zuweisung an a
cout<<a;        $\rightarrow$            Ausgabe: 24
cout<<b;        $\rightarrow$            Ausgabe: 22

```

Postfix:

```

b = 21; a=2;
a + = b++;    // zuerst Zuweisung an a, dann b inkrementieren
cout<<a;        $\rightarrow$            Ausgabe: 23
cout<<b;        $\rightarrow$            Ausgabe: 22

```

Entsprechendes gilt für den Dekrement –Operator.

3.9 Logische Operatoren

Logische Operatoren dienen dazu, Wahrheitswerte zu verbinden. Ein praktisches Beispiel:

„Wenn es Wochenende ist *und* wenn die Sonne scheint dann fahre ich mit dem Motorrad“

In C unterscheidet man logische Operationen auf der Ebene der einzelnen Bits – die **Bitoperatoren** – und logische Operationen auf der Ebene der Variablenwerte. Die Operationen AND, OR; XOR und Negation realisieren dabei die entsprechenden Booleschen Operationen.

Operator	Bedeutung	Rang	Beispiel
&	bitweises AND	9	c = a & b
 	bitweises OR	11	c = a b
^	bitweises XOR	10	c = a ^ b
<<	bitweises Linksschieben (um n Bits)	6	c = a << n
>>	bitweises Rechtsschieben (um n Bits)	6	c = a >> n
~	Einerkomplement	2	c = ~a
&&	logisches AND	12	c = a && b
 	logisches OR	13	c = a b
!	logische Negation	2	c = !a

In den folgenden Tabellen sind die Funktionen der Operatoren mittels Wahrheitstabellen erklärt:

Logisches AND, OR, Negation:

a	b	!a	a && b	a b
true	true	false	true	true
true	false	false	false	true
false	true	true	false	true
false	false	true	false	false

Definition der Wahrheitswerte: Der Wert einer Variablen beliebigen Typs gilt als *true*, wenn er „verschieden von 0“ ist. Konkret bedeutet dies: Wenn ein beliebiges Bit der Variablen den Wert 1 hat, dann hat die gesamte Variable den Wert *true*. Entsprechend hat eine Variable den Wert *false*, wenn jedes Bit 0 ist!

Bit-Operatoren

Beim bitweisen AND, OR bzw. XOR werden innerhalb der Variablen die Bits auf den gleichen Positionen verglichen und daraus die logischen Resultate ermittelt. Die Variablen müssen Ganzzahlen sein !

	Dezimal	Bit - Darstellung							
int a = 3;	3	0	0	0	0	0	0	1	1
~a;	252	1	1	1	1	1	1	0	0
a << 2;	12	0	0	0	0	1	1	0	0
a >> 1;	6	0	0	0	0	0	1	1	0
int b = 14;	14	0	0	0	0	1	1	1	0
c=a & b;	2	0	0	0	0	0	0	1	0
c=a b;	15	0	0	0	0	1	1	1	1
c=a ^ b;	13	0	0	0	0	1	1	0	1

- Beim Linksschieben werden von rechts Nullen nachgezogen !

- Beim Rechtsschieben werden von links Nullen nachgezogen, falls der erste Operand positiv oder unsigned ist !
- Beim Rechtsschieben werden von links Einsen nachgezogen, falls der erste Operand negativ ist !

Hinweis: Das Linksschieben um 1 Stelle der Multiplikation des Wertes mit 2. Entsprechend entspricht das Rechtsschieben um 1 Stelle der Division des Wertes durch 2.

3.10 Sonstige Operatoren

Die beiden nachfolgend beschriebenen Operatoren sind Sonderfälle, werden in C jedoch zu den Operatoren gezählt

4.6.1 Bedingungsoperator

Der **Bedingungsoperator** stellt eine Ausnahme dar insofern er 2 Operatoren verwendet und 3 Operanden hat.

Syntax: `a ? b : c;`

Funktion: Wenn a wahr ist (true), dann gilt b, sonst c

Beispiel:

```
short max, var1=1, var2=2;
max = (var1 > var2) ? var1 : var2;
```

4.6.2 Der sizeof() – Operator

Von der Syntax her betrachtet gehört der sizeof() – Operator zu den C-Funktionen, deshalb wird zu Näherem auf Kap 5 verwiesen.

Funktion: Liefert die *Grösse des Arguments* in Bytes

Beispiel:

```
long l_var;
float feld[10];
cout << sizeof( int );           Ausgabe: 4
cout << sizeof( l_var );         Ausgabe: 8
cout << sizeof( feld );          Ausgabe: 40
```

4.6.3 Der Adressoperator &

Der Adressoperator steht vor einer Variablen und gibt statt ihres Wertes ihre RAM-Adresse zurück, wie folgendes Beispiel demonstriert:

`cout<<&Variable1;` → Ausgabe-Beispiel: 0X12AABB

4.6.4 Der Bereichsoperator ::

Der Bereichsoperator ermöglicht den Zugriff auf Variablen des globalen Gültigkeitsbereiches, wenn im lokalen Bereich Variablen mit demselben Namen deklariert sind. Weiterhin wird er für den Zugriff auf Namensbereiche benötigt, siehe dazu unten.

Syntax (Ohne Leerstelle!): ::

4 Kontrollstrukturen

Mit den bisher bekannten Syntaxelemente können lediglich sequentiell abzuarbeitende Anweisungsfolgen realisiert werden. Dies ist in der Praxis völlig unzureichend. Dies verdeutlichen folgende Problemstellungen:

- Eine Ganzzahl (Zähler) soll durch eine andere Ganzzahl (Nenner) dividiert werden. Da nicht durch 0 dividiert werden kann, muss vor der Division zunächst geprüft werden, ob der Nenner = 0 ist. Ist dies nicht der Fall, muss die Rechnung abgebrochen und gegebenenfalls noch eine Warnmeldung ausgegeben werden. Hierbei handelt es sich um die einfache Alternative
- In einem Haushalt gelte ein fester Wochenspeiseplan wie folgt:
 - *Montag: Gericht A*
 - *Dienstag: Gericht B*
 - *Mittwoch: Gericht C*
 - *Usw*

Hierbei handelt es sich um eine Mehrfachauswahl

- Es sollen die y-Werte der Funktion $y=x^2 + 2x - 6$ berechnet werden für folgende Werte von x: $-3 < x < 3$: Damit sind alle Werte für x und auch die Anzahl der verschiedenen x-Werte bekannt. Hierbei handelt es sich um eine Wiederholung (Schleife) mit (für den Programmierer) bekannter Anzahl von Wiederholung
- Es sollen Umfang und Fläche eines Kreises berechnet werden. Hierzu muss der Benutzer den Radius eingeben. Da eine Radius nicht < 0 sein kann, ist nach der Eingabe zu prüfen, ob ein gültiger Wert (>0) eingegeben wurde. Ansonsten ist die Eingabe zu wiederholen (solange bis ob ein gültiger Wert) eingegeben wurde. Hierbei handelt es sich um eine Wiederholung (Schleife) mit (für den Programmierer) unbekannter Anzahl von Wiederholung

Entsprechende Situationen treten in der Programmierpraxis auf. Somit verfügt der Programmiersprache über Sprachkonstrukte, welche es erlauben, entsprechend komplexere Programmabläufe zu realisieren. Man die diese Sprachelemente Kontrollstrukturen, da sich hiermit der Programmablauf steuern (kontrollieren) lässt. Es handelt sich um Kontrollstrukturen für

- einfache Alternative
- Mehrfachauswahl
- Wiederholung bekannter Anzahl von Wiederholung
- Wiederholung unbekannter Anzahl von Wiederholung

Daneben gibt es noch die Sprunganweisung (Fahre (unter einer bestimmten Bedingung) bei der Anweisung mit der Nr ... fort).

Da in der Praxis SW-Programme i.d.R. komplexe Ablaufstrukturen aufweisen, wurden als Hilfsmittel zur übersichtlichen Darstellung des Programmablaufs spezielle standardisierte grafische Darstellungsarten entwickelt, dabei handelt es sich um

- Darstellungsart **Programmablaufplan** (PAP)
- Darstellungsart **Struktogramm** (Nassi-Schneidermann-Diagramm)

Diese sind als Alternativen zu betrachten. Zu den nachfolgend im Detail erläuterten Sprachelementen in C sind für beide Diagrammtypen jeweils die entsprechenden Komponenten angegeben. Die Texte, welche in den Zeichnungskomponenten stehen sind frei, d.h. es können aussagekräftige Texte geschrieben werden.

In C / C++ gibt es folgende Kontrollstrukturen:

- Die for – Schleife
- Die while – Schleife
- Die do ... while – Schleife
- Die if – Anweisung
- Die switch – Anweisung
- break, continue, goto

Sie werden nachfolgend im Detail erläutert

4.5 Die Verbundanweisung

Alle erwähnten Kontrollstrukturen verwenden Verbundanweisungen. Dabei handelt es sich um sequentielle Anweisungen innerhalb eines speziellen Programmpfades, z.B. innerhalb einer Schleife. Diese werden immer in ein paar Klammern `{ }` gesetzt. Ausnahme, wenn die Verbundanweisung aus einer einzigen Klammer besteht, dann kann das Klammerpaar entfallen (Muss aber nicht). Eine Verbundanweisung nennt man auch einen **Block**. Jeder Block bildet gleichzeitig auch einen eigenen Gültigkeitsbereich, d.h. innerhalb eines Blocks können lokale Variable deklariert werden, welche nur innerhalb dieses Blocks verwendet werden können!

Beispiele:

```
for()
{
    short local1;    //lokale Variable
    cin>> local1;
    cout<<local1*5;
}
```

} Block = lokaler Gültigkeitsbereich

```
for()
{
    cout<<k<<endl;
}
```

for() cout<<k<<endl;

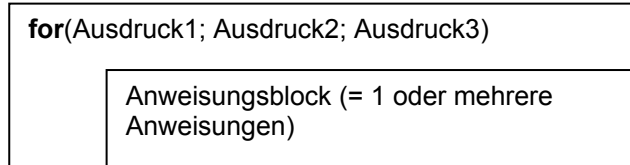
↔ Gleichwertig

4.6 Die for – Schleife

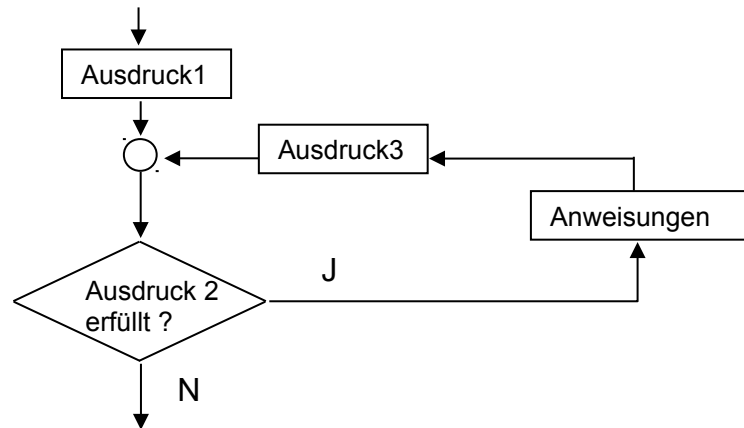
Syntax:

```
for(Ausdruck1; Ausdruck2; Ausdruck3)
{
    //Anweisungen; (Verbundanweisung)
}
```

Struktogramm:



Flussdiagramm (=PAP):



Beispiel:

```
for (int k=0; k<10;k++)      cout<<k;
```

Hier wird die Ausgabeanweisung genau 10-mal ausgeführt.

Die **Verbundanweisung** wird so oft (wiederholt) ausgeführt, bis die in Ausdruck2 formulierte Bedingung nicht mehr erfüllt ist!

Ausdruck1, Ausdruck2 und Ausdruck3 bilden die **Schleifensteuerung**.

Regeln:

- Der Ausdruck1 (Initialisierung) legt die Anfangsbedingung fest
- Der Ausdruck2 wird vor jeder Ausführung der Verbundanweisung ausgewertet.
 - Bei true wird die Verbundanweisung ausgeführt
 - Bei false wird Schleife beendet
- Der Ausdruck3 (Fortschaltausdruck) wird nach Abschluss der Verbundanweisung berechnet
- Nach Beendigung der Schleife wird mit der ersten auf die Verbundanweisung folgenden Anweisung fortgefahren
- Es können ein, zwei oder alle Ausdrücke der Schleifensteuerung entfallen: `for(;;)`
- Die Zählvariable kann in der Schleife manipuliert werden:

```
for (int k=0; k<10;)
{
    k++;
    cout<<k;
}
```

- for –Schleifen können ineinander verschachtelt werden

```
for (int k=10; k>0;k--)
{
    // Anweisungen;
    for (int m=0; m<100;m++)
    {
        // Anweisungen;
    }
    // Anweisungen;
}
```

- Auch folgendes ist möglich:

```
for(int k=0, m=3; m<10; k--, m++)
```

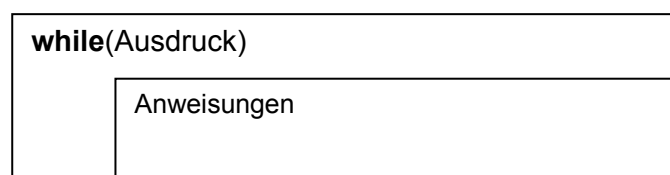
Die for - Schleife wird typischerweise angewandt, wenn die Anzahl der Schleifendurchläufe bekannt ist und gezählt werden kann.

4.7 Die while – Schleife

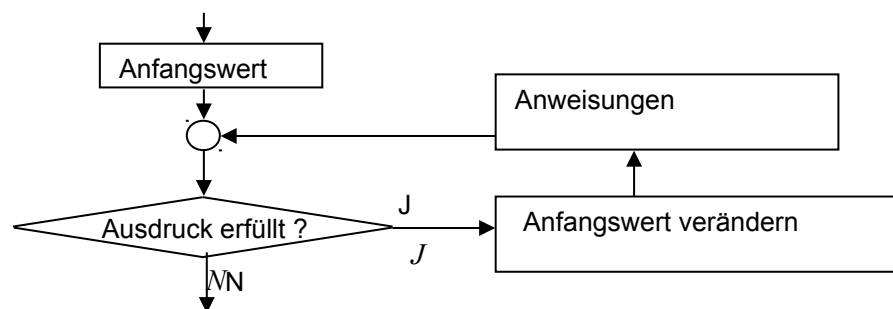
Syntax:

```
while (Ausdruck)
{
    //Anweisungen;
}
```

Struktogramm:



Flussdiagramm (PAP):



Beispiel:

```
int cnt=100, min;
cout << "Gib Minimalwert ein:"; cin>>min;
while(cnt > min)
{
    Anweisungen;
    cout<<cnt<<endl;
    cnt--;
    //Nicht vergessen!
}
```

Der hinter dem Schlüsselwort while stehende Ausdruck ist die **Prüfbedingung**. Solange diese Bedingung erfüllt ist wird die Verbundanweisung wiederholt.

Auch while - Schleifen können verschachtelt werden.

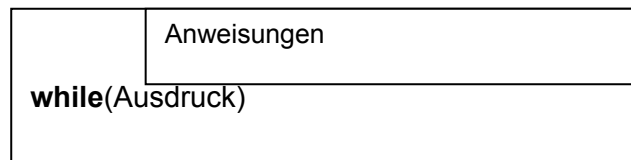
Sowohl bei der for- wie auch bei while – Schleife besteht die Möglichkeit, dass die Verbundanweisung überhaupt nicht ausgeführt werden, wenn nämlich der Prüfausdruck bereits bei der allerersten Prüfung nicht erfüllt ist. Man nennt diese Konstrukte aus diesem Grund **abweisende Schleifen**. Da die Prüfung vor der Ausführung der Verbundanweisung erfolgt, heißen sie auch **kopfgesteuerte Schleifen**.

4.8 Die do....while – Schleife

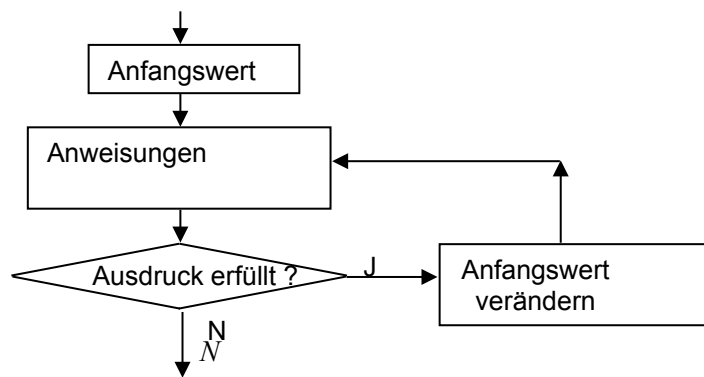
Syntax:

```
do
{
    //Anweisungen;
}while (Ausdruck);
```

Struktogramm:



Flussdiagramm: (PAP)



Beispiel:

```
int cnt=100, min;
cout << "Gib Minimalwert ein:"; cin>>min;
do
{
    //Anweisungen;
    cout<<cnt<<endl;
    cnt--;    //Nicht vergessen!
} while(cnt > min);
```

Im Gegensatz zur `while` - Schleife und zur `for`-Schleife wird die `do... while` - Schleife auf jeden Fall mindestens 1 – mal ausgeführt. Man spricht aus diesem Grund von einer **annahmenden Schleife** oder auch von einer **fussgesteuerten Schleife**.

Auch `do...while` -Schleifen können verschachtelt werden.

4.9 Die if – Anweisung

Mit ihr werden alternative Ausführungspfade realisiert.

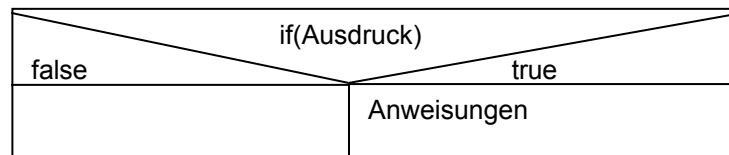
4.9.5 Die einfache if-Anweisung

Syntax:

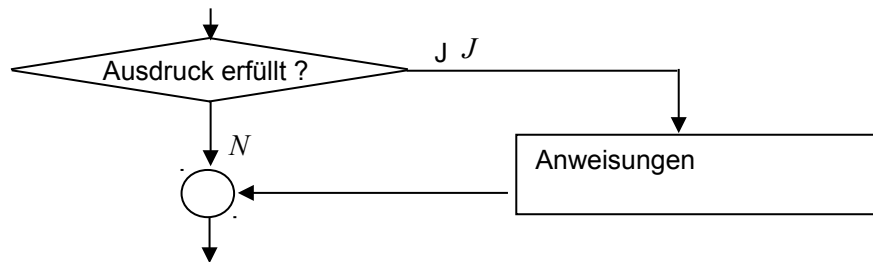
```
if (Ausdruck)
{
    Anweisungen;
}
```

Wenn Ausdruck erfüllt ist, d.h wenn `Ausdruck != 0`, dann wird die Verbund-Anweisung ausgeführt, andernfalls wird sie übersprungen.

Struktogramm:



Flussdiagramm: (PAP)



Beispiel:

```
char chr;
int cnt=100, min;
cout << "Ist die Bedingung erfüllt (J/N) :";
cin>>chr;
if(chr == 'J' || chr == 'j')
{
    Anweisungen;
    cout<<cnt<<endl;
}
```

Natürlich können `if` - Anweisungen verschachtelt werden.

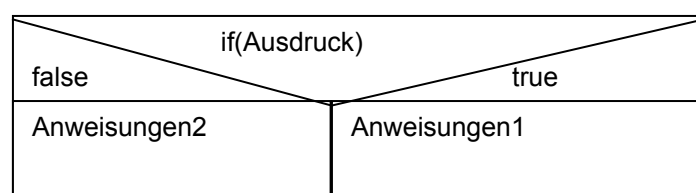
4.9.6 Die if...else – Anweisung

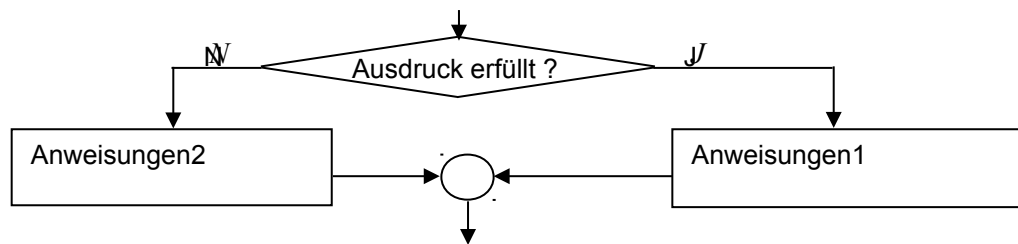
Syntax:

```
if(Ausdruck)
{
    Anweisungen1;
}
else
{
    Anweisungen2;
}
```

Wenn Ausdruck erfüllt ist ($\neq 0$), dann werden die Verbundanweisung `Anweisungen1` ausgeführt, ansonsten die Verbundanweisung `Anweisungen2`, d.h. hier liegen 2 alternative Ausführungspfade vor, wie dies auch das Flussdiagramm deutlich veranschaulicht.

Struktogramm:



Flussdiagramm: (PAP)Beispiel:

```

float wert1=???, wert2;
if(wert1 >= 0.f)    wert2=sqrt(wert1);
else
{
    cout<<"Wurzel aus negativer Zahl!"<<endl;
    wert2=sqrt((-1.) * wert1);
}
  
```

if ...else - Anweisungen können verschachtelt werden, wie folgt

```

if(...)
{ ...}
else
{
    if(...)
    { ...}
    else { ...}
}
  
```

4.10 Einige typische Fehler

- Semikolon hinter der `if`-Anweisung: `if (...) ;`
Das Semikolon hinter der schliessenden Klammer des Prüfausdrucks bildet die **leere Anweisung**. Die leere Anweisung ist eine syntaxmässig gültige Anweisung, bei deren Ausführung jedoch tatsächlich nichts ausgeführt wird.

In obigem Beispielfall bedeutet dies: Ist der Prüfausdruck erfüllt wird der Programmpfad der leeren Anweisung ausgeführt, jedoch ohne jeden Effekt.

Während die leere Anweisung somit syntaxmässig erlaubt ist, stellt sie normalerweise einen logischen Fehler im Code dar. Auch hinter der schliessenden Klammer der Steueranweisungen der `for`-schleife ist die leere Anweisung erlaubt.

- Verwechseln von Zuweisung und logischer Bedingung

<code>if(a = 22)</code>	//Zuweisung!
<code>if(a == 22)</code>	//Bedingung

Im ersten Fall erhält `a` den Wert 22 und damit ist der Prüfausdruck immer erfüllt!

- Falsche `else` Zuordnung:
Annahme: Im folgenden Beispiel soll die `else`-Klausel dem ersten `if` zugeordnet werden:

```

if (...)
if(...)
else          //Dieses else wird dem zweiten if zugeordnet !
{...}

```

Korrekte Lösung: In diesem Fall muss die korrekte Zuordnung durch Verwenden eines Paares der Klammer {} realisiert werden, auch wenn die Klammern ggf syntaxmässig nicht erforderlich wären.

```

if (...)
{
    if(...)    {....}
}
else {....}

```

4.11 Die switch – Anweisung

Mit der switch-Anweisung wird die **Mehrfachauswahl** (mehr als 2 Alternativen) realisiert

Syntax:

```

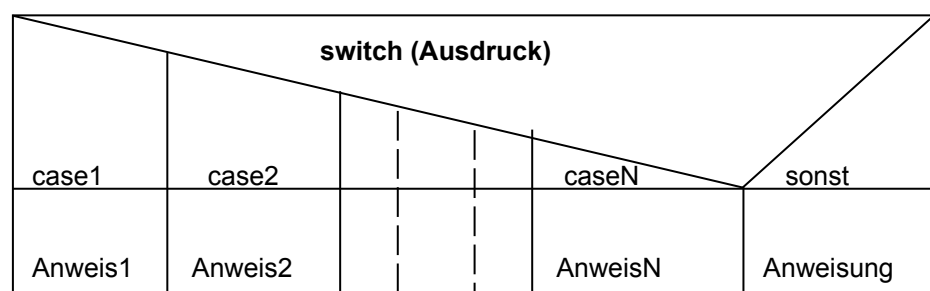
switch(Ausdruck)
{
case 1: {Anweisungen1; break;}
case 2: {Anweisungen2; break;}
...
case N: {AnweisungenN; break;}
default: Anweisungen;
}

```

Beschreibung:

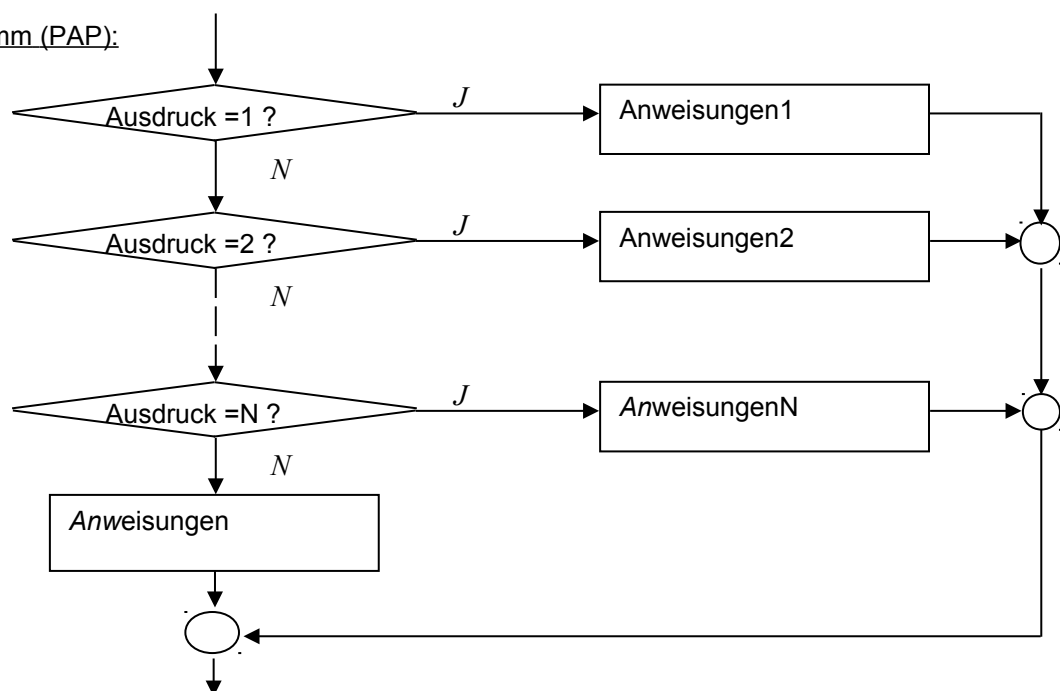
- **Ausdruck** wird ausgewertet, d.h. Ausdruck liefert einen Wert, anschliessend wird derjenige Pfad ausgeführt, dessen Wert (Hinter dem Schlüsselwort **case**) dem **Ausdruck** entspricht. Man kann sich das so vorstellen, dass mittels des Wertes, den **Ausdruck** liefert, eine Sprungmarke angesprungen wird.
- Wird ein Anweisungsblock nicht mit **break;** abgeschlossen, so werden alle nachfolgenden Anweisungsblöcke ausgeführt.
- Ein Anweisungsblock muss nicht in {} eingeschlossen werden (kann es jedoch).
- Der **default** – Pfad ist optional, d.h. er kann entfallen. Wenn er jedoch vorhanden ist, dann wird er genau dann ausgeführt, wenn keiner der **case** –Werte auftritt
- Liefert der Prüfausdruck ein Ergebnis vom Typ **char**, so können die Vergleichswerte in den **case**-Anweisungen entweder in der Schreibweise für Zeichen ('a') oder direkt im ASCII – Code angegeben werden (z.B. 97 für a)

Struktogramm:



Beispiel

```
//Taschenrechner
float wert1, wert2, res;
char op;
cout<<"Gib 2 Zahlen ein:"; cin>> wert1>>wert2;
cout<<"Waehle die Operation: (+,-,*,/)";cin>>op;
switch(op)
{
    case '+': res= wert1+wert2; cout<<res; break;
    case '-': res= wert1-wert2; cout<<res; break;
    case '*': res= wert1*wert2; cout<<res; break;
    case '/': res= wert1/wert2; cout<<res; break;
    default: cout<<"Es wurde kein Operator
gewählt!";
}
```

Flussdiagramm (PAP):

4.12 Verschiedene Kontroll-Anweisungen

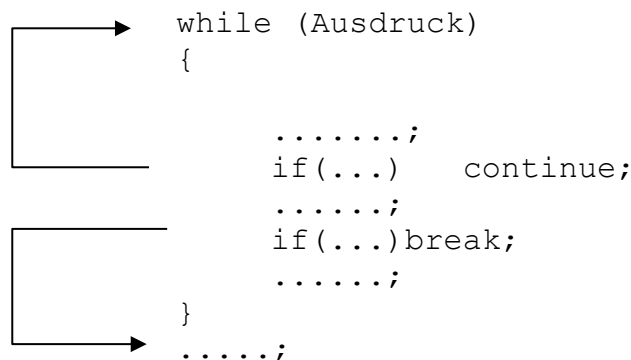
4.12.5 Break – Anweisung

Diese ermöglicht das Verlassen einer Verbundanweisung. Die Ausführung wird mit der ersten Anweisung fortgesetzt, welche der schliessenden Klammer der Verbundanweisung folgt. Ihre Anwendung im Rahmen der `switch-case`-Struktur wurde oben bereits erläutert.

4.12.6 Continue – Anweisung

Diese unterbricht die Abarbeitung der Verbundanweisungen einer Schleife und führt zur vorzeitigen Neuberechnung der Ausdrucksbedingung und ggf erneutem Eintritt in die Ausführung der Verbundanweisung (also Wiederholung).

Die folgende Abbildung veranschaulicht die Funktionsweise sowohl der `continue` wie auch der `break`-Anweisung



4.12.7 goto – Anweisung (Sprung – Anweisung)

Diese ermöglicht das Springen zu einer vordefinierten Marke, wie in nachfolgendem Beispiel:

```

void main()
{
    short zaehler, nenner;
    float quotient;
    cin>>zaehler;
    cin>>nenner;
    if(nenner==0) goto xyz;
    cout<<float(zaehler)/nenner;
    xyz: return;
}
  
```

Die vielfache Verwendung der Sprung – Anweisung führt zu unübersichtlichem und verwirrendem Code mit der Gefahr schwer entdeckbarer Fehler (Spaghetti-Code), weswegen es in der modernen Programmiertechnik üblich ist, diese nicht zu verwenden. Neuere Programmiersprachen wie Java kennen die Sprung – Anweisung überhaupt nicht mehr.

5 Felder (Arrays)

In der Datenverarbeitung tritt häufig der Fall auf, dass eine Menge Daten desselben Typs, welche eine zusammenhängende Einheit bilden, verarbeitet werden muss.

Beispiele:

- In einer meteorologischen Messreihe soll an einem Messpunkt über einen Monat jeweils die Mittagstemperatur gemessen werden
- Für statistische Auswertungen werden in einer Schule von allen Schülern Körpergewicht und Körpergröße erhoben
- Eine Digitalkamera erzeugt für jedes Bild eine 2-dimensionale Anordnung von Ganzzahlen

Um effektiv solche Datenmenge handhaben und auf einzelne Daten davon zugreifen zu können werden Datenfelder, oder einfach **Felder**, engl. **Arrays**, verwendet.

D.h ein Array besteht aus mehreren oder vielen **Feldelementen**, wobei alle Feldelemente denselben Datentyp haben. D.h für jedes Feldelement steht so viel Speicher im RAM zur Verfügung, wie der Datentyp angibt.

Jedes Feldelement kann wie eine Variable des entsprechenden Typs behandelt werden.

Hinweis: Das Überschreiten der Feldgrenzen ist leicht möglich und führt normalerweise zu schweren Laufzeitfehlern

5.5 Deklaration von Feldern

In der Deklaration von Feldern werden Datentyp, der Name des Feldes, sowie die Anzahl und Anordnung der Feldelemente angegeben.

Genauso wie einfache Variable können Arrays lokal oder global deklariert werden.

5.5.5 Eindimensionale Felder

Hierbei handelt es sich um eine lineare Anordnung von Werten. Diese sind auch im RAM linear und zusammenhängend gespeichert.

```
int feld1[10];
```

Name Größe = Zahl der Feldelemente

Mittels obiger Deklaration sind somit $4 \cdot 10 = 40$ Bytes belegt.

5.5.6 Mehrdimensionale Felder

Für jede Dimension wird die Größe in einem Klammerpaar `[]` angegeben.

```
char feld2[10][21][3];    //3 Dimensionen
float bild[1000][500];    //2 Dimensionen
```

Für das Feld `feld2` sind $10 \cdot 21 \cdot 3 = 630$ Bytes belegt, für das Feld `bild` entsprechend $4 \cdot 1000 \cdot 500 = 2000000$ Bytes.

5.6 Verwendung von Feldern

Auf Feldelemente wird einzeln wie auf eine Variable zugegriffen. Das einzelne Feldelement wird hierzu durch den Namen des Feldes sowie eine Nr, den sogenannten **Feldindex**, oder einfach **Index**, identifiziert. Die folgenden Beispiele demonstrieren dies an Hand der Felder aus dem vorigen Abschnitt:

```
feld1[0] = 234;
int k = feld1[i];
feld1[12] = 5;    //Achtung Fehler
                  //wegen Feldgrenzenüberschreitung!!
```

Regeln:

- Indexwerte können Konstanten sein oder Variablen eines Ganzzahltyps.
- Das **erste Element** hat den Index **0** !
- Für jede Dimension wird ein Index benötigt.

```
geld2[0][1][2] = 'A' ;
```

Es folgt, dass der grösste gültige Index gleich der Grössenangabe in der Deklaration -1 ist !

Beispiel: `bild[999][499] = -10;`

5.7 Initialisierung von Feldern

Die Werte der Feldelemente lokaler Felder sind wie bei normalen Variablen undefiniert. Zu Initialisierung gibt es die Möglichkeit der Zuweisung von Werten an die einzelnen Elemente wie folgt:

```
for(short k=0;k<10;k++) feld1[k]=0;
```

Bereits bei der Feld-Deklaration kann die Initialisierung mittels des **Initialisierers** erfolgen:

```
char feld2[5]={'a', 'b', 'c', 'd'};
```

Regeln:

- Enthält der Initialisierer weniger Elemente als das Feld, werden die übrigen Feldelemente mit **0** initialisiert
- Enthält der Initialisierer genau so viele Elemente wie das Feld, dann kann die Angabe der Feldgrösse entfallen:

```
int feld3[]={1,2,3,4}; //Feld mit 4 Elementen
```

Die Initialisierung mehrdimensionaler Felder mittels Initialisierer zeigt folgendes Beispiel:

Alternative1

```
int seasonTemp[3][4] = {26,34,22,17,24,32,19,13,28,38,25,20};
```

Alternative1

```
int seasonTemp[3][4] =
{
{26, 34, 22, 17},
{24, 32, 19, 13},
{28, 38, 25, 20}
};
```

6 Modularisierung

Ebenso wie in der HW-Entwicklung und Produktion die Projekte bzw. Produkte in Komponenten aufgliedert werden, ist entsprechendes auch bei der SW erforderlich. Entsprechende Unterstützung bietet auch die Programmiersprache C. Folgende 2 Möglichkeiten der Modularisierung sind zu unterscheiden:

- Funktionale Modularisierung durch Gliederung in C-Funktionen
- Quellcode –Modularisierung durch Aufteilung Codes in mehrere Quellcode-Dateien

Die Quellcode –Modularisierung ist nur auf der Basis der funktionale Modularisierung durch C-Funktionen möglich

6.5 Funktionen in C

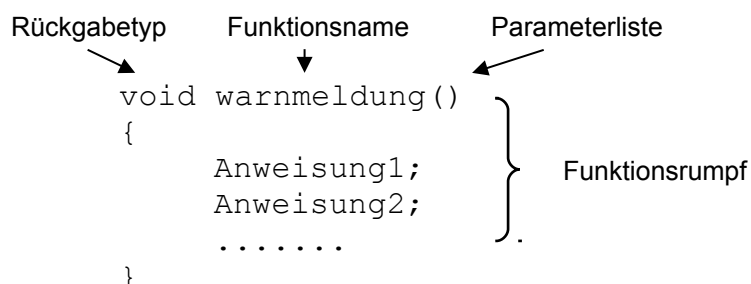
Funktionen sind **Unterprogramme**. Ihre Aufgabe besteht darin, eine modulare SW-Strukturierung zu ermöglichen. Dies dient dazu

- zusammenfassbare Funktionalitäten in eigene Einheiten auszugliedern, was die Übersichtlichkeit und Handhabbarkeit des Codes verbessert
- Funktionalitäten, welche wiederholt benötigt werden, als Einheiten auszugliedern
- Wiederverwendbare Einheiten zu ermöglichen, d.h Teile eines SW-Programmes, welche in gleicher Weise auch in anderen Programmen Verwendung finden. Funktionen sind somit grundlegend für SW-Bibliotheken
- Die Aufgliederung des Code in mehrere Quellcode-Dateien. So ist es z.B. nicht möglich, die erste Hälfte des Hauptprogrammes in eine Quellcode-Datei zu schreiben, und die zweite Hälfte in eine andere Quellcodedatei.

Andere häufig gebrauchte Begriffe für Funktionen sind **Routine** oder **Prozedur**. Auch das Hauptprogramm ist eine Routine.

6.6 Funktionsdeklaration

Das folgende Beispiel zeigt den allgemeinen Aufbau einer C-Funktion. Man erkennt die Ähnlichkeit mit dem Aufbau der Routine `main`.



- Der **Funktionsname** dient der Identifikation der Funktion
- Wenn eine Funktion einen Ergebniswert liefert, dann gibt der **Rückgabotyp** an, welchen Datentyp dieser Ergebniswert hat
- Mittels der **Parameterliste** bekommt die Funktion Werte, die sie benötigt, damit sie ihre Aufgaben erfüllen kann, im obigen Beispiel hat die Funktion eine leere Parameterliste

- Im Funktionsrumpf stehen die Anweisungen, d.h. das, was „Die Funktion tun soll“. Alle bereits bekannten Sprachelemente können hier verwendet werden, Insbesondere können hier auch lokale Variable deklariert werden.

Begriffsdefinitionen:

- Rückgabetyp, Funktionsname und Parameterliste bilden den **Funktionskopf**
- Funktionsname und Parameterliste bilden die **Signatur** der Funktion

Das folgende Beispiel zeigt eine Funktion, welche mittels Parametern Werte bekommt und einen Ergebniswert liefert, welcher gleichzeitig auch auf dem Monitor ausgegeben wird.

```
float summe3(float w1, float w2, float w3)  
{  
    float ergebnis;  
    ergebnis=w1+w2+w3;  
    cout<<"Summe="<<ergebnis<<endl;  
    return ergebnis;  
}
```

- Der Name der Funktion ist `summe3`
- Der Rückgabetyp ist `float`, d.h. die Funktion liefert einen Ergebniswert vom Typ `float`
- Die Funktion übernimmt die Parameter `w1`, `w2`, `w3`. Sie heissen **formale Parameter**. Die Werte dieser Parameter werden benötigt, um die Summe berechnen zu können. Parameter können im Rumpf der Funktion wie lokale Variable verwendet werden. Eine andere gebräuchliche Bezeichnung für Parameter ist **Argumente**.
- Es wird die lokale Variable `ergebnis` deklariert.

Die `return`-Anweisung führt dazu, dass die Funktion an dieser Stelle verlassen wird, d.h. Anweisungen, welche auf eine `return`-Anweisung folgen, werden nicht ausgeführt. (Allerdings kann eine `return`-Anweisung auch hinter einer `if`-Anweisung stehen, so dass sie nur unter gewissen Bedingungen ausgeführt wird). Liefert eine Funktion einen Ergebniswert, so wird dieser in der `return`-Anweisung angegeben. Dabei kann es sich handeln um

- Eine Konstante wie z.B.: `return 0;`
- Eine lokale Variable wie z.B.: `return ergebnis;`
- Einen Ausdruck wie z.B.: `return w1+w2+w3;`

Es gelten folgende Regeln:

- Ist der Rückgabetyp einer Funktion `void`, dann ist die `return`-Anweisung optional, d.h. sie kann weggelassen werden.
- Ist der Rückgabetyp einer Funktion von `void` verschieden, dann muss der Typ des Ergebniswertes, welcher hinter `return` steht, identisch sein mit dem Rückgabetyp

Hinweis: Werden in der `return`-Anweisung mehrere Werte angegeben, wie z.B.

```
return w1, w2;
```

so erzeugt dies keine Compiler-Fehlermeldung, es wird jedoch nur der letzte Wert zurückgegeben

Für die Syntax der Parameterliste gelten folgende Regeln

- Für jedes Argument muss der Typ angegeben werden
- Die einzelnen Argumente werden durch Komma getrennt
- Gibt es keine Argumente, dann steht eine leere Parameterliste
- Im Prinzip sind beliebig viele Argumente möglich

Beispiele

```
fName(char c1, int i1, float f1)
```

```
fName(int i1, int i2, float f1, float f2, char c1)
```

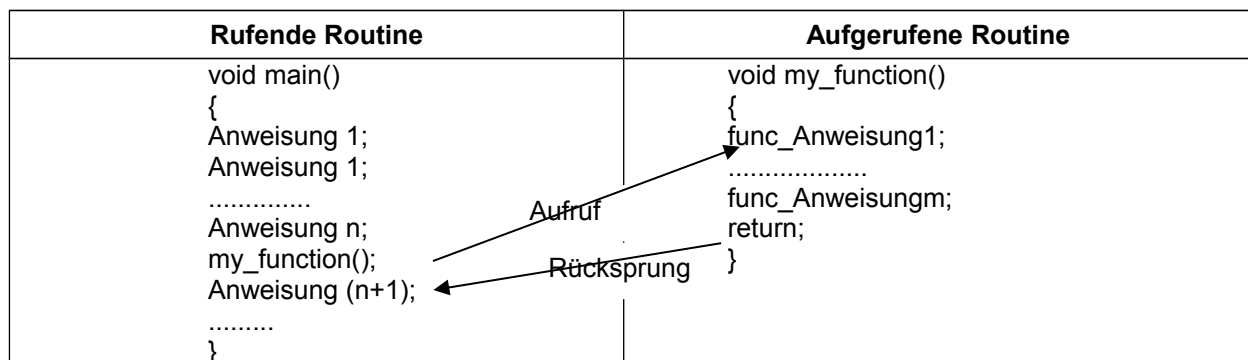
Folgendes ist falsch:

```
fName(int i1, i2, char c1)
```

6.7 Verwendung von Funktionen: Der Funktions-Aufruf

Damit die Anweisungen einer Funktion ausgeführt werden, muss diese von einem anderen Programmteil aufgerufen werden. Dann wird die Ausführungsfolge der Funktion mit der ersten ausführbaren Anweisung. Trifft die Ausführung auf eine `return`-Anweisung, so wird die Funktion verlassen und die Ausführung fortgesetzt in der rufenden Routine, und zwar in der ersten Anweisung, welche dem Funktionsaufruf folgt.

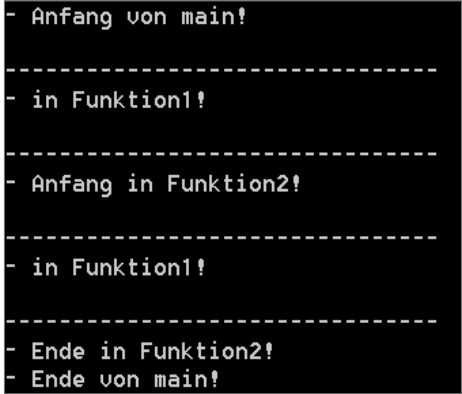
Der Funktionsaufruf erfolgt unter Verwendung des Funktionsnamens. Voraussetzung ist, dass die Funktion zuvor deklariert worden ist (vergl. Variablen-Deklaration). Der Funktionsaufruf wird wie jede Anweisung mit ; abgeschlossen.



Es gelten folgende Regeln:

- Funktionen können nicht nur aus dem Hauptprogramm aufgerufen werden, sondern auch aus anderen Funktionen, so dass letztlich eine ganze Aufrufkaskade entstehen kann.
- Eine Funktion kann beliebig oft aufgerufen werden. Beim mehrfachen Aufruf einer Funktion bleiben die Werte der lokalen Variablen nicht erhalten. Da lokale Variable bei Aufruf einer Funktion automatisch erstellt und beim Verlassen wieder entfernt werden nennt man diese auch **automatische Variable**

Beides demonstriert das folgende Beispiel, bei welchem sich der Ablauf der Ausführung an Hand der jeweiligen Ausgaben genau verfolgen lässt.

Code	Monitorausgabe
<pre> void trennlinie() { cout<<endl; cout<<"-----"<<endl; } void funktion1() { cout<<"- in Funktion1!"<<endl; } void funktion2() { cout<<"- Anfang in Funktion2!"<<endl; trennlinie(); funktion1(); trennlinie(); cout<<"- Ende in Funktion2!"<<endl; } void main() { cout<<"- Anfang von main!"<<endl; trennlinie(); funktion1(); trennlinie(); funktion2(); cout<<"- Ende von main!"<<endl; } </pre>	 <pre> - Anfang von main! ----- - in Funktion1! ----- - Anfang in Funktion2! ----- - in Funktion1! ----- - Ende in Funktion2! - Ende von main! </pre>

6.7.5 Parameter und Ergebniswerte

Das Beispiel der Funktion `summe3` ist eine Funktion mit 3 Parametern, welche einen Wert vom Typ `float` zurückgibt. Sie kann auf folgende 2 alternative Weisen aufgerufen werden

Möglichkeit 1: Ergebniswert wird nicht verwendet

```
summe3(2.2, 3.3, 4.4);
```

Möglichkeit 2: Ergebnis wird verwendet; `eee` nimmt in der rufenden Routine den Ergebniswert auf

```
float eee;
eee = summe3(2.2, 3.3, 4.4);
```

- Die **aktuellen Parameter** ersetzen die formalen Parameter, sie dienen der Übergabe von Werten an die Funktion, d.h. 'innerhalb' der Funktion haben die dortigen Parameter dann die Werte `w1 = 2.2`, `w2 = 3.3` und `w3 = 4.4`
- Gibt eine Funktion einen Wert zurück, so kann dieser in der aufrufenden Routine verwendet werden, muss jedoch nicht
- Statt Konstanten wie in obigem Beispiel können als aktuelle Parameter auch Variablen der rufenden Routine angegeben werden, wie in folgendem Beispiel:

```
float f1=1.01, f2=123.567, f3=-987.01;
float eee = summe3(f1, f2, f3);
```

6.8 Referenzübergabe

Bisher gibt es nur 2 Möglichkeiten für Funktionen, an die aufrufenden Routine Ergebnisse zurückzugeben:

- Mit globalen Variablen
- Als Rückgabewert
Die Verwendung globaler Variablen sollte weitmöglichst vermieden werden, da es schwierig ist (zumindest bei grosser SW), die Übersicht auf den Zugriff hierauf behalten. Als Rückgabewert wiederum kann eine Funktion nur 1 Werte liefern, was in der Praxis oft nicht genügt. Aus diesem Grund können via Parameterliste auch weitere Ergebnisse an die aufrufenden Routine zurückgegeben werden. Dies geschieht mittels Referenzübergabe. Hierbei erlaubt die aufrufende Routine der gerufenen Funktion direkten Zugriff auf den Speicherplatz einer ihrer lokalen Variablen. Damit hat die aufgerufenen Funktion dann die Möglichkeit, zusätzliche Werte an die rufende Routine zu übertragen.

Demgegenüber wird bei der normalen Übergabe, auch **Wertübergabe** genannt, für den Parameter innerhalb der Funktion ein RAM-Speicherplatz angelegt und in diesen der als aktueller Parameter beim Funktionsaufruf übergebene Wert eingetragen. Beides wird nachfolgend veranschaulicht:

Art der Übergabe	Wertübergabe	Referenzübergabe
Übertragungsrichtung von Werten	<pre> graph LR Main --> Funktion </pre>	<pre> graph LR Main <--> Funktion </pre>
Im RAM	<pre> graph LR subgraph "rufende Routine" LV1[localVar = WERT] end subgraph "aufgerufene Routine" P[Parameter = WERT] end P -- WERT --> LV1 </pre>	<pre> graph LR subgraph "rufende Routine" LV1[localVar = WERT] end subgraph "aufgerufene Routine" Z[Zugriff] end Z --> LV1 </pre>

Die Referenzübergabe wird mittels des Adressoperators & in der Parameterliste einer Funktion festgelegt. Sie muss für jedes Argument extra angegeben werden. Beispiel:

```
float refDemoFunktion(char c1, float &f1, float f2, int& i1)
```

- In diesem Beispiel ist Referenzübergabe für f1 und i1 definiert.
- Folgende Schreibweisen sind erlaubt:

```
float &f1
float&f1
float& f1
```

Im folgenden Beispiel wird die Referenzübergabe dazu verwendet, 2 Ergebniswerte an die rufende Routine zurückzugeben.

Code	Monitorausgabe
<pre> Void kreis(float radius, float &flaeche, float &umfang) { float umfang; umfang=2.f*3.1415f*radius; flaeche=3.1415f*radius*radius; } void main() { float rad, um, fl; cout<<"- Eingabe Kreisradius"; cin>>rad; kreis(1.1, fl, um); cout<<"Umfang="<<um<<endl; cout<<"Flaeche="<<fl<<endl; } </pre>	<pre> - Eingabe Kreisradius2.0 Umfang=12.566 Flaeche=12.566 </pre>

6.9 Standardparameter (Defaultargument)

In Fällen, in welchen gewisse Parameter aus der Parameterliste einer Funktion normalerweise bestimmte *typische Werte* aufweisen, können diese typischen Werte in der Definition innerhalb der Parameterliste angegeben werden. Sollen diese Standardwerte bei einem Aufruf der Funktion verwendet werden, so kann auf deren Angabe im Aufruf verzichtet werden. D.h. nur dann, wenn diese typischen Werte nicht verwendet werden sollen, dann muss beim Aufruf an der Stelle der Standardparameter ein aktueller Wert angegeben werden

Standardargumente müssen innerhalb einer Parameterliste als letzte aufgeführt werden. Weiter gelten folgende Regeln:

- Standardargumente sind in der Funktionsdeklaration anzugeben, in der Implementierung (s.u.) kann darauf verzichtet werden
- Standardargumente nur in C++ möglich

Beispiel:

Ein Programm zur Ausgabe einer Fehlermeldung

```

void Fehler (char *message, int severity = 0)
{
    cout<< message<<"Schweregrad: "<< severity;
}

```

Aufrufbeispiele:

```

Fehler ("Division durch null", 3); //severity hat den Wert 3
Fehler ("Rundungsfehler");        // severity hat den Wert 0

```

6.10 Übergabe von Feldern an Funktionen

Sollen Arrays an Funktionen übergeben werden, so geschieht dies ebenfalls mittels der Parameterliste. Dabei in die bereits bekannte Schreibweise für Arrays zu verwenden. Es gelten folgende Regeln

- Bei Arrays erfolgt (Automatisch) immer Referenzübergabe
- Innerhalb der Funktion ist darauf zu achten, dass die Feldgrenzen nicht überschritten werden.
- Ist das Feld nur eindimensional, so kann in der Parameterliste auf die Angabe der Feldgrösse verzichtet werden (muss aber nicht)

D.h. folgende Beispiele sind bzgl der Syntax korrekt:

```
void Feldinitialisierung1(short groesse, float array[111])
void Feldinitialisierung1(short groesse, float array[])
void Feldinit2(float array[10][10], short groese1, short groese2)
```

Beim Aufruf einer Funktion mit einem Array in der Parameterliste wird an der Position des Arrays nur der Name des zu übergebenden Feldes der rufenden Routine angegeben, wie folgt:

```
float localFeld[111];
Feldinitialisierung1(111, localFeld);
```

6.11 Gültigkeitsbereiche

Funktionen definieren eigene lokale Gültigkeitsbereiche, d. h. es ist erlaubt, Variablennamen zu verwenden, die in anderen Gültigkeitsbereichen bereits verwendet wurden, wie in folgendem Beispiel:

```
#include <iostream.h>
float var1=103.444f;      //Globale Variable
void funk()
{
    int var1=27;    //Lokale Variable
    cout<<"In der Funktion:"<<endl;
    cout<<"Lokale Variable :"<<var1<<endl;
    cout<<"Globale Variable :"<<::var1<<endl;
}

void main()
{
    char var1= 'A';      //Lokale Variable
    cout<<"Lokale Variable :"<<var1<<endl;
    cout<<"Globale Variable :"<<::var1<<endl;
}
```

6.12 Statische Variable und Funktionen

6.12.5 Globale Deklaration

Bei Programmen, welche aus mehreren Quelldateien bestehen, ist es gelegentlich sinnvoll, den Zugriffsbereich von *globalen* Variablen und Funktionen nur auf die Datei zu beschränken, in welcher

sie deklariert und verwendet werden. Dies erfolgt mit dem Speicherklassen-Spezifizierer (storage class specifier) **static**.

Beispiel: Diese Funktion ist nur verwendbar in der Quellcodedatei, in welcher sie steht

```
static int FindNextRoute (void)
{
    //...
}
```

Man nennt dies eine statische (globale) Funktion. In entsprechender Weise können auch statische globale Variablen deklariert werden.

6.12.6 Lokale Deklaration

Oft wünschenswert, dass der Wert ausgewählter lokaler Variablen einer Funktion bei deren Verlassen nicht verlorengeht, damit er beim nächsten Aufruf wieder zur Verfügung steht. Das geschieht durch die Definition statischer lokaler Variablen ebenfalls mittels **static**.

```
void Fehler (char *message)
{
    static int count;    // statische lokale Variable
    if (++count > limit) cout<<"zu viele Fehler";

    //...
}
```

Statische lokale Variable werden automatisch mit 0 initialisiert.

Im folgenden Beispiel zählt eine Funktion selbstständig unter Verwendung einer statischen Variablen wie oft sie aufgerufen wird.

Code	Monitorausgabe
<pre>void AufRufZaehler() { static unsigned int cnt; cnt++; cout<<cnt<<"-ter Aufruf der Funktion"<<endl; } void main() { unsigned int anz; cout<<"Eingabe Anzahl Aufrufe: "; cin>>anz; for(int l=0;l<anz;l++) AufRufZaehler(); }</pre>	<pre>Eingabe Anzahl Aufrufe: 4 1-ter Aufruf der Funktion 2-ter Aufruf der Funktion 3-ter Aufruf der Funktion 4-ter Aufruf der Funktion</pre>

6.13 Inline Funktionen

Wird eine bestimmte Funktion häufig aufgerufen, so führt dies zu einer langsameren Programmausführung durch die Aktionen, welche für Aufrufe ausgeführt werden müssen. Dies wird vermieden, indem eine Funktion als `inline` deklariert wird. Hierzu wird das Schlüsselwort `inline` verwendet. In diesem Fall ersetzt der Compiler den Aufruf der Funktion durch den Rumpf (Anweisungsfolge) der Funktion

Beispiel: Bestimmung des Absolutwertes einer Zahl

```
m = (n > 0 ? n : -n)
```

Implementierung als `inline` Funktion, um obige Anweisung nicht häufig (im Quellcode) schreiben zu müssen:

```
inline int Abs (int n)
{
    return n > 0 ? n : -n;
}
```

6.14 Überladen von Funktionen

Normalerweise müssen Funktionsnamen eindeutig sein. Eine Ausnahme hiervon bietet das Verfahren des Überladens. **Überladen** (Overloading) bedeutet: Verschiedene Funktionen mit *identischem Funktionsnamen!*

Gibt es mehrere Funktionen mit identischem Namen, so müssen sich diese in ihrer Signatur unterscheiden. Konkret bedeutet dies, dass sie verschiedene Parameterlisten aufweisen müssen. Beim Funktionsaufruf wählt der Compiler dann mittels der aktuellen Argumente aus, welche Funktion (-Variante) aufzurufen ist.

Die Parameterliste kann sich unterscheiden

- In der Anzahl der Parameter
- Im Datentyp aller oder einzelner Parameter
- Sowohl in der Anzahl wie auch im Datentyp der Parameter

Gibt es mehrere Funktionen mit identischem Namen, so spricht man von *überladenen Funktionen*. Überladen ist **nur in C++** möglich !

Beispiel: Die Pfeile deuten an, welche der Funktionen jeweils aufgerufen wird. Hier erfolgt die Auswahl über den Typ der Argumente.

```
float flaeche(float r)
{
    //Berechnung der Kreisfläche
    return 3.14152*r*r;
}
float flaeche(float a, float b)
{
    //Berechnung der Rechteckfläche
    return a*b;
}
void main()
{
    float a,b,r;
    float flaech;
    .....
    flaech = flaeche(r);
    ....
    flaech = flaeche(a,b);
}
```

```
...
}
```

6.15 Rekursion

In C kann eine Funktion sich auch selbst aufrufen. Man spricht von Rekursion. Ein Beispiel ist die Berechnung der Fakultät:

Fakultät von 0 ist 1 $0! = 1$

Fakultät von n ist: $n! = 1*2*3*...*n = (n-1)!*n$

Dies wird durch folgende **rekursive Funktion** realisiert:

```
int Factorial (unsigned int n)
{
    return n == 0 ? 1 : n * Factorial(n-1);
}
```

- Eine rekursive Funktion muss eine Abbruchbedingung aufweisen, da sie ansonsten sich selbst unbegrenzt oft aufruft.
- Allgemein können rekursive Funktionen auch als **Iteration** umgeschrieben werden:

```
int Factorial (unsigned int n)
{
    int result = 1;
    while (n > 0) result *= n--;
    return result;
}
```

6.16 Variable Parameterliste

Bei der variablen Parameterliste ergibt sich die Anzahl der Parameter erst zur Laufzeit.

- Die Parameterliste muss mindestens einen bekannten Parameter enthalten
- Auf die unbekannten Parameter wird mittels der Makros (s.u.) **va_start**, **va_arg** und **va_end** sowie eine Zeigervariable vom Typ **va_list** zugegriffen
- Es ist das Einbinden der Header-Datei `<stdarg.h>` erforderlich

Beispiel:


```

float durchschnitt(short erster, ...) //... weist auf
//eine variable Parameterliste hin !
{
    short summe=0, i=erster;
    float cnt=0.;
    va_list zeige;           //Definition der Zeigervariablen
    va_start(zeige, erster); //zeige wird auf den ersten
                           //Parameter gesetzt
    while(i != -1)
    {
        summe += i;    cnt++;
        //Lesen des jeweils nächsten Parameters
        i=va_arg(zeige, short);
    }
    //setzt zeige auf NULL zurück
    va_end(zeige);
    return (summe ? summe/cnt : 0.F);
}

void main()
{
    cout<<"Durchschnitt=: "<<durchschnitt(1,2,-1)<<endl;
    cout<<"Durchschnitt=: "<<durchschnitt(-1) <<endl;
    cout<<"Durchschnitt=: "<<durchschnitt(9,3,7,4,-1);
}

```

Hier steht eine Typangabe! Ende der Parameterliste !

- Das Ende der Parameterliste wird nicht automatisch erkannt, hierfür muss eine Vereinbarung getroffen werden (in obigem Beispiel der Wert -1). Dieser Wert muss beim Aufruf als letzter Wert angegeben werden.
- Der 2. Parameter in `va_arg` gibt den Typ des Parameters, dessen Wert zurückgeliefert werden soll

6.17 Kommandozeilenparameter

Werden Programme über die Kommandozeile gestartet (d.h. in der Form einer Befehlseingabe) so kann man oftmals für den Start gewisse Parameter eingeben. Man nennt dies die Kommandozeilenparameter.

Das folgende Beispielprogramm addiert die eingegebenen Parameter und gibt die Summe davon aus:

```

#include <iostream.h>
#include <stdlib.h>      // für atof erforderlich

int main (int argc, const char *argv[])
{
    double sum = 0;
    for (int i = 1; i < argc; ++i)
        sum += atof(argv[i]);
    cout << sum << '\n';
    return 0;
}

```

Beispielhafter Start des Programms über die Kommandozeile:

>dateiname 1 2 3 4

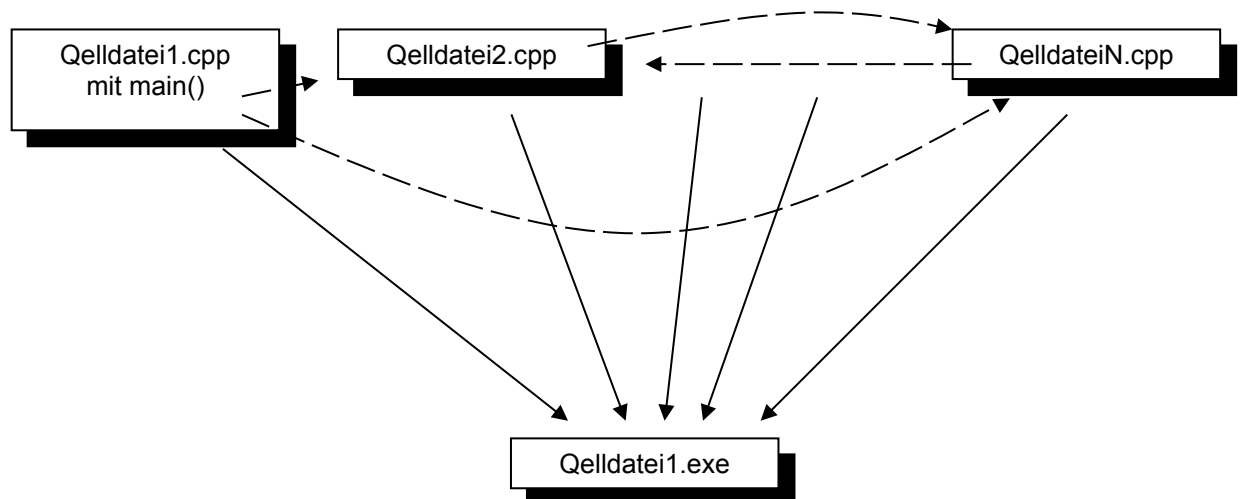
Erläuterung:

Parameter **argc**: Anzahl der Kommandozeilparameter = Elementzahl im Feld `argv[]`
Der Name des Programms zählt als der 1. Parameter, so dass `argc >= 1` ist.

Parameter **argv[]**: Feld mit Zeigern (s.u) auf die einzelnen Kommandozeilenparameter.
Das erste Element zeigt auf den Namen der exe-Datei des ausgeführten Programmes

6.18 Aufgliederung in mehrere Quellcodedateien

Bereits in 1.3.2 wurde darauf hingewiesen, dass grosse SW-Projekte aus mehreren Quellcodedateien bestehen, wie in folgender Darstellung veranschaulicht:



Da das Hauptprogramm als Einheit in einer Quellcodedatei stehen muss, folgt hieraus unmittelbar, dass aus dem Hauptprogramm heraus Funktionsaufrufe erfolgen müssen auf Funktionen, welche sich in anderen Quellcodedateien befinden. Darüberhinaus werden aber auch ansosnsten Aufrufe von Funktion erfolgen, welche sich in anderen Quellcodedateien befinden. Dies ist in der Abbildung durch die gestrichelten Pfeile veranschaulicht. Der Vorteil ist

- Mehrere SW-Entwickler können parallel am gleichen Projekt arbeiten
- Vorhandene SW-Module können wiederverwendet werden
- Die Quellcodedateien sind kleiner und damit besser handhabbar

Diese Vorgehensweise wird in C- unterstützt durch die Aufteilung in

- Funktionsdeklaration
- Funktionsimplementierung

6.18.5 Funktionsdeklaration und – Implementierung

Die Funktionsdeklaration – ein anderer Begriff hierfür ist **Funktionsprototyp** oder einfach Prototyp – enthält alle Informationen, die zum Verwenden einer Funktion (Aufruf) notwendig sind, d.h. es handelt sich genau um den Funktionskopf. Die Funktionsdeklaration muss im Code vor dem ersten Aufruf stehen, sie wird abgeschlossen mit: ;

Die Funktionsimplementierung – ein anderer Begriff hierfür ist **Funktionsdefinition** – enthält die Anweisungen, d.h. den Rumpf der Funktion. Sie kann an beliebiger Stelle innerhalb der Quellcode Dateien stehen – sofern die Syntaxregeln eingehalten werden.

Im folgenden Beispiel steht die Funktionsdeklaration vor dem Hauptprogramm und damit vor dem ersten Aufruf, die Funktionsimplementierung jedoch hinter dem Hauptprogramm:

```
#include <iostream.h>

float summe3(float w1, float w2, float w3); ← Deklaration

void main()
{
    float s;
    s=summe3(2.2f, 3.3f, 4.4f);
}

float summe3(float w1, float w2, float w3) } Definition
{
    float sum=w1+w2+w3;
    return sum;
}
```

Man erkennt: Der Kopf der Funktion am Ort ihrer Implementierung ist identisch mit der Deklaration. In C kann aus diesem Grunde

- In der Funktionsdeklaration auf die Namen der Parameter verzichtet werden, d.h es genügt die Angabe der Datentypen.
- In der Funktionsimplementierung auf die Datentypangaben der Parameter verzichtet werden, d.h es genügt die Angabe der Namen der Parameter.

Obiger Code wäre also auch folgendermassen korrekt :

```
#include <iostream.h>

float summe3(float, float, float);

void main()
{....}

float summe3(w1, w2, w3)
{
    float sum=w1+w2+w3;
    return sum;
}
```

Diese Vorgehensweise empfiehlt sich jedoch nicht.

6.18.6 Die Header-Datei

Um das Einfügen der Funktionsdeklaration in all jenen Quellcodedateien zu vereinfachen, in welchen sie benötigt werden, kann man geeignete Header-Dateien erstellen, welche sodann mit einer include-Direktive (s. Präprozessor) eingebunden werden. Die Header-Datei ist gekennzeichnet durch die Endung `.h`. Die include-Direktive führt dazu, dass durch den Präprozessor der Inhalt der angegebenen Datei an der Ort der include-Direktive kopiert wird, wodurch sie für den Compiler am erforderlichen Ort steht. Damit kann dann die Funktionsimplementierung in einer anderen Quellcodedatei stehen. Obiges Beispiel kann somit unter Verwendung einer Header-Datei in folgender Weise in 2 Module aufgeteilt werden:

Quellcodedatei1.cpp	<pre>#include <iostream.h> #include "HeaderDatei.h" void main() { float s; s=summe3(2.2f, 3.3f, 4.4f); }</pre>
Quellcodedatei2.cpp	<pre>float summe3(float w1, float w2, float w3) { float sum=w1+w2+w3; return sum; }</pre>
HeaderDatei.h	<pre>float summe3(float w1, float w2, float w3);</pre>

Die Datei für die Funktionsimplementierungen und die zugehörige Header-Datei bilden somit eine logische Einheit!

6.18.7 Externe Variablen

Besteht ein SW-Projekt aus mehreren Modulen, so kann der Fall eintreten, dass in einem Modul eine globale Variable deklariert wird, welche in anderen Modulen benötigt wird. Eine nochmalige globale Deklaration ist aber unzulässig. Hier findet die **extern – Deklaration** Verwendung.

Beispiel:

Datei AAA.cpp:

```
int abc;
void main() {...}
```

//Global !! Die Variable wird nur definiert

Datei BBB.cpp:

```
extern int abc;
void f1() {...}
```

//Die Variable wird
//deklariert, nicht definiert

Bei der extern-Deklaration wird kein Speicherplatz reserviert (sondern nur die Variable ‚bekanntgemacht‘)!

Fehler: `extern int size=0;` //Die ist eine Definition!!

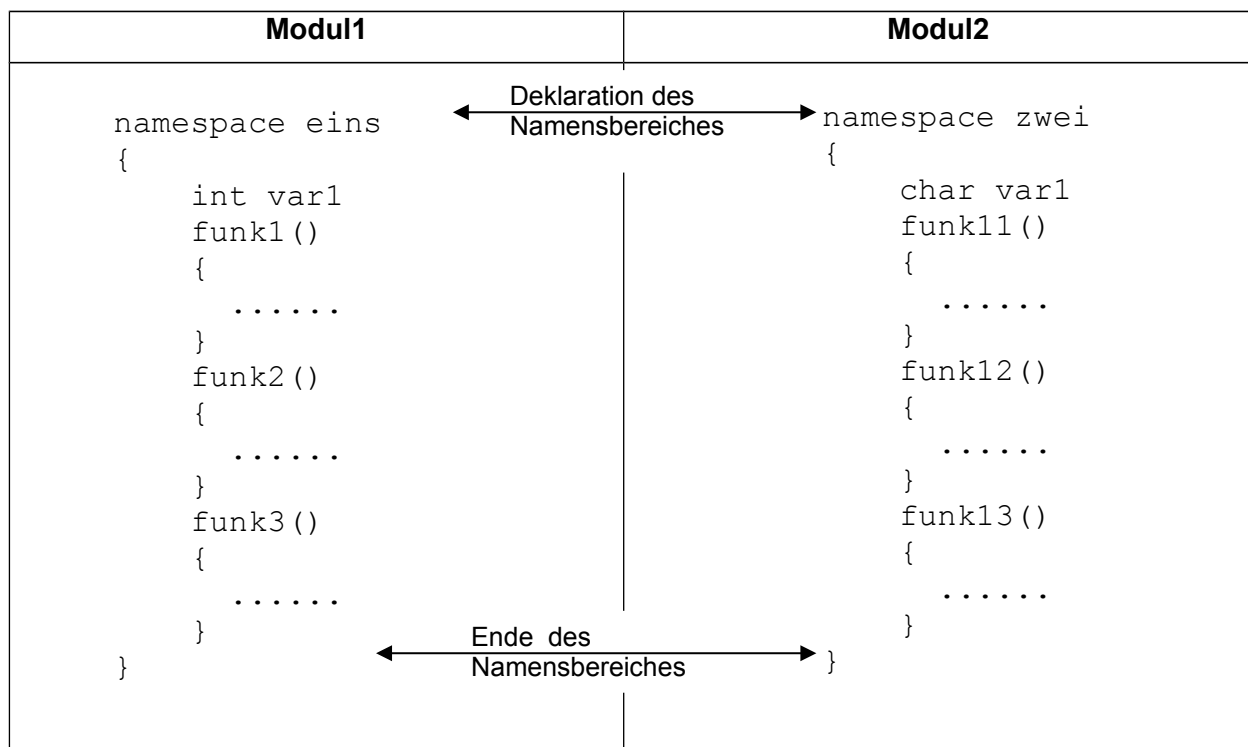
Hinweis: Extern –Deklarationen werden sinnvollerweise in Header-Dateien untergebracht

6.19 Namensbereiche (Namespaces)

Bei grossen SW-Projekten erstreckt sich der globale Gültigkeitsbereich normalerweise über viele Quellcodedateien, welche z.B auch gekaufte SW-Module wie Funktionsbibliotheken enthalten können. In solchen Situationen kann der Fall auftreten, dass identische Namen für verschiedene globale Variablen verwendet werden sollen. So kann z. B. das Modul1 die Funktionen `funk1`, `funk2`, `funk3` enthalten, welche die globale Variable `var1` verwenden. Im Modul2 befinden sich die Funktionen `funk11`, `funk12`, `funk13` enthalten, welche die ebenfalls eine globale Variable mit dem Namen `var1` verwenden.

D.h der globale Gültigkeitsbereich teilt sich auf in 2 Bereiche, nämlich den Bereich der Funktionen `funk1`, `funk2`, `funk3` in Modul1 und den der Funktionen `funk11`, `funk12`, `funk13` in Modul2. Um zu ermöglichen, dass der Variablenname `var1` in beiden Bereichen unabhängig verwendet werden kann, verwendet man Namensbereiche (namespace)

6.19.5 Deklaration von Namensbereichen



Regeln

- Die Verwendung von Namensbereichen ist optional
- Auf globale Variablen, die innerhalb eines Namensbereiches deklariert sind, kann nur innerhalb dieses Namensbereiches zugegriffen werden. Dies gilt auch, wenn die verschiedenen Namensbereiche innerhalb eines Moduls deklariert sind
- Funktionen, die innerhalb eines Namensbereiches deklariert sind, können nur innerhalb dieses Namensbereiches aufgerufen werden.

6.19.6 Zugriff auf Elemente aus Namensbereichen

Nun kann jedoch der Bedarf bestehen, z.B. in obigem Beispiel aus der Funktion `funk1` die Funktion `funk11` des anderen Namensbereichs aufzurufen. Hierzu gibt es folgende Alternativen:

- Mit Bereichsoperator `::`
- Mit der `using` – Deklaration: In dieser wird die Verwendung eines Namensbereiches angegeben; nachfolgend können in diesem Modul die globalen Variablen aus dem angegebenen Namensbereich und die Funktionen aufgerufen werden

Das folgende Beispiel demonstriert dies bezüglich der obigen Beispiele

Beispiel:

```
using namespace eins ; //Nachfolgend kann alles aus Namens-
                        //bereich eins verwendet werden !

void main()
{
    int a,b,c;
    float d,e,f;
    a=25+var1;          //Verwendung von var1 aus namespace eins
    a=zwei::funk11(wert1); //Aufruf von funk11 aus namespace
                        //zwei
}
```

6.20 Die Standard Bibliothek (RunTime-Library – RTL)

Eine Menge von häufig benötigten Funktionen wurden standardisiert und werden vom Compilerhersteller mitgeliefert. Ihre Header-Dateien befinden sich in dem speziellen automatisch installierten Verzeichnis `include`. Die Funktionen werden in der Literatur sowie im mitgelieferten Hilfesystem erläutert. Eine kleine Auswahl davon sind in der nachfolgende Tabelle aufgeführt.

Funktionsname	Beschreibung	Header-Datei
<code>printf()</code>	Formatierte Ausgabe	<code>stdio.h</code>
<code>scanf()</code>	Formatierte Eingabe	<code>stdio.h</code>
<code>time()</code>	Ausgabe von Systemdatum und Systemzeit	<code>time.h</code>
<code>localtime()</code>	Ausgabe von Systemdatum und Systemzeit als locale Zeit	<code>time.h</code>
<code>sin()</code>	Berechnung sinus-Funktion	<code>math.h</code>
<code>cos()</code>	Berechnung cosinus-Funktion	<code>math.h</code>
<code>sqrt()</code>	Berechnung Quadratwurzel	<code>math.h</code>
<code>exp()</code>	Berechnung Exponentialfunktion	<code>math.h</code>
<code>pow(X, Y)</code>	Berechnet X^Y , Y darf auch eine Komma-Zahl sein. So wird z. B. für $Y=1/3$ die dritte Wurzel berechnet	<code>math.h</code>
<code>fabs()</code>	Berechnung des Betrags einer Fließkommazahl	<code>math.h</code>
<code>rand()</code>	Liefert eine Zufallszahl	<code>stdlib.h</code>
<code>strcat()</code>	Aneinanderhängen von Zeichenketten	<code>string.h</code>

strcpy()	Kopieren von Zeichenketten	string.h
strcmp()	Vergleichen von Zeichenk	string.h
abs()	Berechnung des Betrags einer Int-Zahl	stdlib.h
atof()	Umwandlung von ASCII nach float	stdlib.h
atoi()	Umwandlung von ASCII nach int	stdlib.h

Hinweise für die Verwendung

- Die mathematischen Funktionen arbeiten normalerweise mit Argumenten vom Typ `double` und als Ergebniswert liefern sie ebenfalls `double`
- Die trigonometrischen Funktionen aus der RTL erwarten das Argument im Bogenmass,

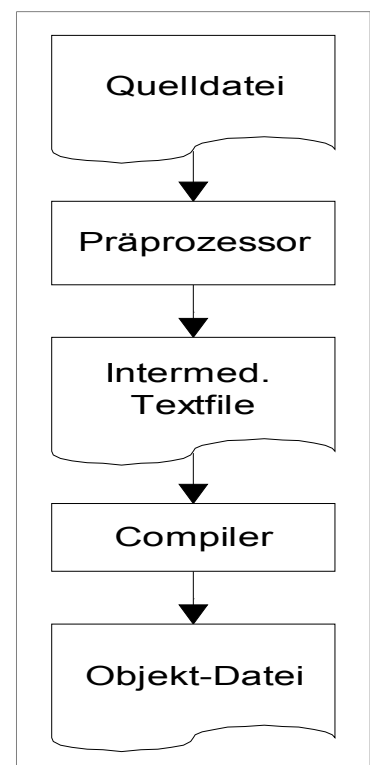
7 Der Präprozessor

Unter dem Präprozessor versteht man ein SW-Programm, welches den vom Programmierer erstellten Quellcode noch vor dem eigentlichen Compiler bearbeitet. Dieses Verfahren ist spezifisch für C. Das Ergebnis der Bearbeitung durch den Präprozessor ist immer noch eine C-Quelldatei, welche jedoch keine Präprozessor-Anweisungen mehr enthält, da genau diese durch den Präprozessor bearbeitet wurden. Eine wichtige Folgerung davon ist, dass die Präprozessor-Anweisungen so erstellt werden, dass der resultierende Quellcode exakt die C-Syntax-Anforderungen erfüllt!

Dazu gibt es natürlich eigenen Syntax-Regeln für Präprozessor-Anweisungen.

Übersicht der Aufgaben des Präprozessors:

- Umformen der Quelldatei in eine Textdatei durch Ausführen der Präprozessor-Anweisungen (Direktiven)
- Entfernen von Kommentaren
- Einfügen anderer Dateien
- Makro-Expansion
- Fehler-Check innerhalb der Präprozessor-Anweisungen



Folgerung:

Präprozessoranweisungen gelten nur für die aktuelle Übersetzungseinheit (Modul)

Hinweis: Das Entfernen der Kommentare erfolgt *vor* der Auswertung der Präprozessor-Anweisungen, somit ist es möglich, Präprozessor-Anweisungen vorübergehend unwirksam zu machen, indem

man ihnen den Zeilenkommentar voranstellt, wie folgt:

```
// #include <...>
```

Syntax

Allgemeiner Aufbau einer Präprozessor-Anweisung:

include <.....>

└────────┘ └────────┘

Direktive Tokens

- Alle Präprozessor-Anweisungen beginnen mit # gefolgt von einer Präprozessor-Direktive. Zwischen beiden kann eine Leerstelle stehen
- Die Präprozessor-Direktiven entsprechen den Schlüsselworten der Programmiersprache. Sie dienen der Erkennung
- Die Tokens stellen Parameter für die jeweilige Direktive dar. Sie sind durch Leerstellen von der Direktive getrennt. Gibt es mehrere Tokens, so sind auch diese durch Leerstellen getrennt
- Eine Präprozessor-Anweisungen endet mit dem Zeilenendzeichen (Zeilensprung)

Beispiele für äquivalente Anweisungen:

```
#define PI 3.1415
#      define      PI      3.1415
```

Soll eine Präprozessor-Anweisung in der nächsten Zeile fortgesetzt werden, so geschieht dies mit dem **Zeilenverlängerungszeichen** \

Beispiele äquivalente Anweisungen:

Version A:

```
#define CheckError \
        if (error) \
            exit(1)
```

Version B: (Äquivalent zu A)

```
#define CheckError if (error) exit(1)
```

7.5 Übersicht der Präprozessor-Direktiven

Directive	Kurze Erklärung
#define	Definiert ein Macro
#undef	Ende der Definition eines Makro
#include	Einfügen von Text aus einer Datei
#ifdef	Macht die Kompilierung abhängig davon, ob eine Makro definiert ist
#ifndef	Macht die Kompilierung abhängig davon, ob eine Makro nicht definiert ist
#endif	Kennzeichnet das Ende eines Bereiches für bedingte Kompilation
#if	Macht die Kompilierung abhängig davon, ob eine Bedingung erfüllt ist
#else	Spezifiziert einen else-Teil für #ifdef, #ifndef, or #if
#elif	Kombination von #else und #if

#line	Ändert die aktuelle Zeilennummer und ggf Dateiname
#error	Gibt eine Meldung auf Bildschirm aus
#pragma	Implementations-spezifisch
#	Zero (ignored)

7.6 Dateien einfügen mit der Direktive include

```
#include "filexyz.cpp"
```

Der vollständige Inhalt Datei *filexyz.cpp* wird an der Position der Direktive eingefügt. Der Inhalt selbst wird unverändert übernommen. Wenn die einzufügende Datei sich nicht im aktuellen Verzeichnis befindet muss der gesamte Pfad angegeben werden. Man nennt dies Einbinden einer Datei
Beispiel:

```
#include "C:\\eigene_programme\\test1\\filexyz.cpp"
```

Beindet sich die einzubindende Datei im Standard – Include –Verzeichnis, so ist folgende Schreibweise erforderlich:

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

Das Standard – Include –Verzeichnis wird bei der Installation des C-Compilers automatisch angelegt und enthält die Header-Dateien für die mitgelieferten SW-Module

Bei einer *include*-Anweisung kann nur ein Token angegeben werden, d.h. mit einer *include*-Anweisung kann immer nur 1 Datei eingebunden werden

Haupt-Anwendung der *include*-Anweisung ist das Einbinden von Header-Dateien!

7.7 Makro und define-Direktive

Für ein **Makro** verwendet man die Direktive *define*. Mit der Anweisung

```
#define ANAXAMANDER
```

wird eine Art Präprozessor-,Variable' ANAXAMANDER definiert, welche nachfolgend in Präprozessor-Anweisungen verwendet werden kann.

7.7.5 Das einfache Makro

```
#define identifier token
```

Aktion: Nachfolgend werden alle Vorkommen von *identifier* durch *token* ersetzt. Dies gilt auch für reine C-Anweisungen!

Beispiel:

```
#define PI 3.14152f
```

Code vor Präprozessor	Code nach Präprozessor
<pre>#define PI 3.14152f void main() { umfang=2.f*PI*radius; }</pre>	<pre>void main() { umfang=2.f*3.14152f *radius; }</pre>

7.7.6 Parametrisches Makro

```
#define identifier(par1, par2,..) token
```

Aktion: Ähnlich einem Funktionsauf. Nachfolgend werden alle Vorkommen von `identifier` durch `token` ersetzt. Die **formalen Parameter** `par1` usw können in `token` vorkommen. Beim Aufruf des Makro muss für jeden der Parameter `par1` usw ein Wert angegeben werden, welcher dann an entsprechender Stelle in `token` eingefügt wird. Man nennt dies **Makro-Expansion**.

Beispiel: `#define MAX(x,y) ((x) > (y) ? (x) : (y))`

Aufruf: `m = MAX (n - 2, k +6);`

Code vor Präprozessor	Code nach Präprozessor
<pre>#define MAX(x,y) \ ((x) > (y) ? (x) : (y)) void main() { short m,n,k; m = MAX(n-2,k+6); }</pre>	<pre>void main() { short m,n,k; m = ((n-2) > (k+6) ? (n-2) : (k+6)); }</pre>

Das Beispiel zeigt auch, das es sinnvoll ist, im Makro um die Parameternamen innerhalb des token-Bereiches Klammern zu setzen, um hierdurch sicherzustellen, dass nach der Expansion die gewünschte Auswertungsreihenfolge eingehalten wird.

7.7.7 Der #-Operator

Wird '#' einem formalen Parameter innerhalb eines parameterischen Makros vorangestellt, wird beim Aufruf des Makros das aktuelle Argument als Zeichenkettenkonstante dargestellt

Beispiel:

```
#include <iostream.h>
#define TEST(var) cout<<#var
void main()
{
    TEST(Hallo);
}
```

Dies führt zu der Monitorausgabe: **Hallo**

7.7.8 Der ##-Operator

Dieser Operator weist den Präprozessor an, 2 Zeichenfolgen zusammenzufassen.

Beispiel:

```
#include <iostream.h>
#define TEST(wert)  cout<<var##wert
void main()
{
    int var1=0, var2=0, var3=7;
    TEST(1);
    TEST(2);
    TEST(3);
}
```

Nach Expansion

```
cout<<var1;
cout<<var2;
cout<<var3;
```

führt zu der Monitorausgabe:

007

7.8 Bedingte Compilierung

Mit der bedingten Compilierung ist es möglich, ganze Codebereiche aus dem Quellcode zu entfernen, welchen der Compiler erhält (im Originaltext wird nichts verändert!)

- Wenn steht `#define VERSIONA`, dann bleibt der Codebereich der Version A erhalten und der Codebereich der Version B wird entfernt (im Originaltext natürlich nicht)
- Wenn nicht steht `#define VERSIONA`, dann bleibt der Codebereich der Version B erhalten und der Codebereich der Version A wird entfernt (im Originaltext natürlich nicht)

```
#define VERSIONA

#ifdef VERSIONA
    Code der Programmversion A
#else
    Code der Programmversion B
#endif
```

Codebereich der Version A

Codebereich der Version B

Zusammengesetzte Bedingungen möglich:

```
#if defined ALPHA || defined BETA
```

Äquivalente Schreibweisen:

```
#if defined BETA
#ifdef BETA
```

Die bedingte Compilierung bietet die Möglichkeit, einen Codeabschnitt von Compilierung auszuschliessen (vorübergehend, z.B. für Testzwecke):

```
#if 0
...code to be omitted
#endif
```

Vermeidung des mehrfachen Einfügens von Code:

```
#ifndef _file_h_
#define _file_h_
    contents of file.h goes here
#endif
```

7.9 Sonstige Direktiven

7.9.5 Makro assert

Definition: `assert(expression)`

Das `assert`-Makro gibt eine Diagnosemeldung aus, wenn `expression` zu `false` (`=0`) ausgewertet wird, und ruft dann [abort](#) auf, d.h. die Programmausführung wird beendet.

Es wird keine Aktion ausgeführt, wenn `expression` `true` (ungleich null) ist.

Die Diagnosemeldung enthält den Ausdruck, für den der Fehler aufgetreten ist, den Namen der Quelldatei und die Nummer der Zeile, in der der Assertionsfehler aufgetreten ist.

Aufruf – Beispiel

```
cin>>radius;
assert(radius>0.);
```

Erforderliche Header-Datei: `assert.h`

7.9.6 Direktive line

```
#line Zeilennummer dateiname
```

- Erzwingt Änderung der Zeilennummer des Quellcode ab dieser Stelle
- Angabe von `dateiname` ist optional

7.9.7 Direktive error

```
#error error
```

Führt zu sofortigem Abbruch der Compilierung an dieser Stelle und Ausgabe des Token `error`

7.9.8 Direktive pragma

Implementationsspezifisch, wird somit durch den Compiler-Hersteller definiert.

7.10 Vordefinierte Größen

Die nachfolgend angegebenen Größen sind Bestandteil jedes C-Programmes und mit den entsprechenden Werten initialisiert.

Hinweis zur Schreibweise: Vor und hinter den Namen sind doppelte Unterstriche!

Namen	Bedeutung
__FILE__	Name der ausgeführten exe-Datei
__LINE__	Nr der Quellcodezeile, in welcher sich die Anweisung befindet
__DATE__	Das Datum der letzten Compilierung als Zeichenkette ("Dec 24 2019")
__TIME__	Die Zeit der letzten Compilierung als Zeichenkette ("12:30:55")

Die nachfolgenden Konstanten enthalten als Wert die grössten bzw. kleinsten Werte der entsprechenden Datentypen:

Konstante	Bedeutung (Wert)	Header-Datei
FLT_MAX	Grösste darstellbare float - Gleitkommazahl	float.h
FLT_MIN	Kleinste darstellbare float - Gleitkommazahl	float.h
INT_MAX	Grösste darstellbare int - Ganzzahl	limits.h
INT_MIN	Kleinste darstellbare int - Ganzzahl	limits.h

Für die anderen elementaren Datentypen gibt es entsprechende Konstante.

8 Formatierte Ein- und Ausgaben

Die Ein- Ausgaben mittels `cout` und `cin` sind in C nicht verwendbar (nur in C++). Dazu stehen Funktionen der RTL zur Verfügung, welche auch eine detaillierte **Formatierung** (Gestaltung des Aussehens der Ausgabe) erlauben. Nachfolgend werden nur die grundlegenden Funktionen behandelt.

8.5 Ausgabe mit `printf()`

Erforderliche Header – Datei: `#include <stdio.h>`

Einfaches Beispiel: Nur Ausgabe eines Textes

```
printf("Guten Morgen");
```

Der erste Parameter der `printf`-Funktion steht in "..." und heisst **Kontrollzeichenkette**. Er steuert das Aussehen der Ausgabe. An dieser Stelle angegebene Zeichenketten werden als solche auf dem Monitor ausgegeben werden. Tabulatorsprünge, Zeilensprünge u.a. werden durch die als **escape-Sequenzen** bezeichneten Symbole erzeugt. Sie sind in folgender Tabelle aufgelistet:

Escape-Sequenz	Ausgelöste Aktion
<code>\a</code>	Piepton (Alarm)
<code>\b</code>	Rücksetzen um ein Zeichen
<code>\f</code>	Seitenvorschub
<code>\n</code>	Neue Zeile
<code>\r</code>	Wagenrücklauf
<code>\t</code>	Horizontaler Tabulator
<code>\v</code>	vertikaler Tabulator
<code>\0</code>	Kein Zeichen
<code>\?</code>	Fragezeichen
<code>\'</code>	Hochkomma
<code>\"</code>	Anführungszeichen
<code>\\</code>	Umgekehrter Schrägstrich in der Ausgabe
<code>\033</code>	ESC

Beispiel: `printf("\tGuten Morgen\n\t Ich bin auch schon da\n");`

Hinweis: Die escape-Sequenzen können auch innerhalb von Zeichenketten bei der Ausgabe mittels `cout` verwendet werden und führen dann zu entsprechenden Aktionen!

Beispiel: `cout<<"\tGuten Morgen\n\t Ich bin auch schon da\n";`

Ausgabe von Variablen-Werten mittels `printf`:

Die `printf`-Funktion ist eine Funktion mit variabler Parameterzahl. Nach dem obligatorischen Parameter Kontrollzeichenkette kann somit eine beliebige Anzahl von Variablen als Parameter aufgelistet

werden, deren Werte auf dem Monitor ausgegeben werden sollen. Hierzu ist es erforderlich, dass die Art der Ausgabe (Formatierung) mittels eines **Formatelementes** innerhalb der Kontrollzeichenkette gesteuert wird. Es gilt

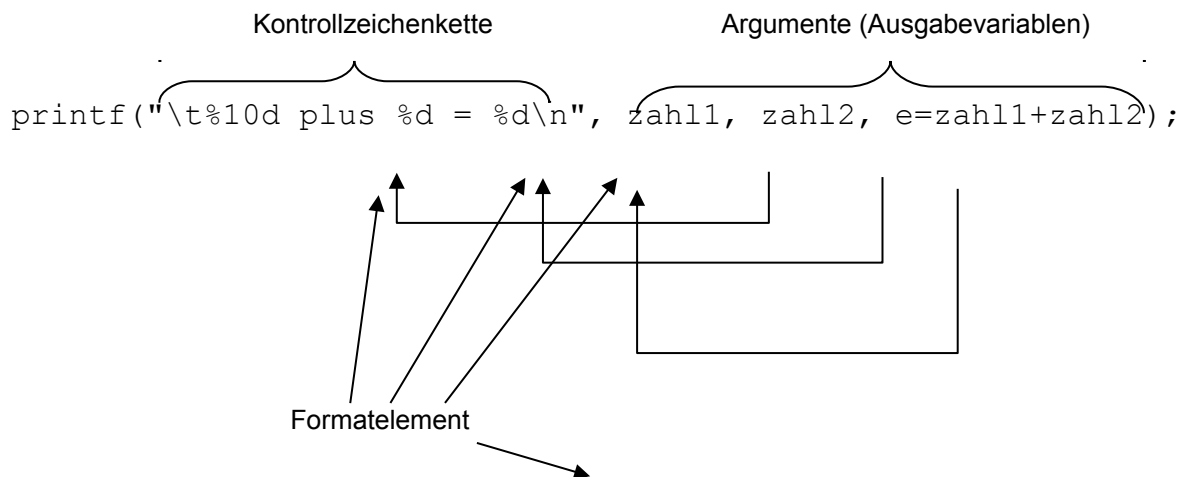
- für jede auszugebende Variable ist ein Formatelement erforderlich
- die Reihenfolge der Formatelemente entspricht der Reihenfolge der Variablen

Beispiel:

```
char var1 = 'Z';
printf("Wert von var1= %c\n", var1);
```

Erzeugte Ausgabe: **Wert von var1 = Z**

Beispiel:



Allgemeine Syntax des Formatelementes:

%Fn.dU

Es bedeuten:

- % - leitet Formatelement ein (Erkennungsmerkmal!)
- F - Formatierungszeichen
- n - minimale Feldlänge (bei Bedarf wird mit Leerzeichen ergänzt)
- . - Trennung der Zahlenangaben n und d
- d - max. Feldlänge oder Anzahl der Ziffern nach Dezimalpunkt
- U - Umwandlungszeichen

Mit Ausnahme des Umwandlungszeichens sind alle anderen Elemente optional.

Verfügbare Formatierungszeichen:

Formatierungszeichen	Bedeutung
-	optional, linksbündige Ausgabe
Keine Angabe	rechtsbündige Ausgabe
%	das % - Zeichen wird ausgegeben
L	optional, folgendes d,o,x,u spezifizieren ein long int Argument
H	optional, folgendes d,o,x,u spezifizieren ein short int Argument

Die **Umwandlungszeichen** steuern die Darstellungsart von Werten entsprechend ihrem Datentyp!

Übersicht der Umwandlungszeichen:

Umwandlungszeichen	Bedeutung
C	Einzelnes Zeichen
s	Zeichenkette
d	Ganzzahl als Dezimalzahl, Vorzeichen möglich
i, u	Ganzzahl, Vorzeichen möglich Bei Oktalzahl wird eine 0 vorangestellt Bei hex-Zahl wird 0x vorangestellt
f	float oder double als dezimale Gleitpunktzahl
g, G	Kürzeste Darstellung (Format d, e oder f)
e, E	float oder double in Exponentialdarstellung
o	int in oktaler Schreibweise
x, X	int in hexadezimaler Schreibweise

Mittels der optionalen Angaben n und d (im Formatelement) wird die Positionierung der Ausgabe gesteuert

Beispiele:

Ganzzahlen: `%10d` → 10-stellige Ausgabe rechtsbündig

Kommazahlen: `%10.2f` → 10-stellige Ausgabe rechtsbündig, 2 Nachkommastellen

8.6 Eingabe mit `scanf()`

- `scanf()` entspricht weitgehend der Ausgabefunktion `printf()`.
- Kontrollzeichenkette gefolgt von einer variablen Liste von Variablen.
- Es gelten die Formatierungs- und Umwandlungszeichen wie bei `printf()`, soweit sinnvoll.
- Den Eingabe - Variablen muss jedoch der Adressoperator `&` vorangestellt werden, da für die Variablen Zeigerübergabe (s.u) spezifiziert ist. Wird dies nicht gemacht, kommt es zu Laufzeitfehlern. Dies gilt jedoch nicht bei Arrays, wie sie z.B. für Zeichenketten verwendet werden, denn bei Arrays ist der Name desselben bereits ein Zeiger!
- Header : `#include <stdio.h>`

Beispiel:

```
void main()
{
    float oper1, oper2, summe;
    int i1, i2;
    //2 Gleitkommazahlen eingeben
    printf("\n Gib 2 Zahlen ein:");
    scanf("%f %f", &oper1, &oper2);
    summe=oper1+oper2;
    printf("\n\t sum= %-10.4f\n", summe);
    //2 Integerzahlen eingeben
```



```
printf("\n Und jetzt 2 Int-Zahlen:");
scanf("%d %d",&i1,&i2);
printf("\n\t sum= %5d\n",i1+i2);
}
```

Hinweis: Die `scanf`-Funktion kann Probleme machen, wenn man Zeichen einlesen möchte und zuvor eine andere Eingabe gemacht worden ist., also in folgender Situation:

```
int i1;    scanf("%d",&i1);
char c1;
scanf("%c",&c1);    //Wird übersprungen
```

Die zweite Eingabe wird übersprungen. Der Grund hierfür ist folgender:

Die erste Eingabe wurde durch den Benutzer durch Eingabe von Enter abgeschlossen. Das Enter-Zeichen verbleibt im Eingabe-Puffer und im folgenden Aufruf der `scanf`-Funktion findet diese bereits das Enter-Zeichen im Eingabe-Puffer, was sie als gültiges Zeichen und gleichzeitig als Ende der Eingabe interpretiert!

Die Lösung besteht darin, dass 2 Aufrufe von `scanf` vorgenommen werden:

```
int i1;    scanf("%d",&i1);
char c1;
scanf("%c",&c1);    //Wird übersprungen
scanf("%c",&c1);    //Hier Eingabe möglich
```

Das Problem besteht nicht bei Eingabe von anderen Datentypen und auch nicht bei Eingabe mittels `cin`. Allerdings tritt es ebenfalls bei Verwendung von `getline()` auf (s.u)

8.7 Eingabe von Zeichenketten mit Leerstellen

Beispiel:

```
char buffer[200];
cin>>buffer;
```

Macht man hier folgende Eingabe:

Robert Mayer

so macht man die Feststellung, dass in `buffer` nur die Zeichenkette **Robert** ankommt. Der Grund hierfür ist, dass die Eingabe eines Leerzeichens (Abstand zwischen 2 Worten) als Ende der Eingabe interpretiert wird (jedoch erst bei Drücken der Enter-Taste). Die verbleibenden Zeichen (**Mayer** im Beispiel) werden für die nachfolgende Eingabe verwendet und stören damit alles).

Nachfolgend wird beschrieben, wie sich auch Zeichenketten mit Leerzeichen einlesen lassen.

Mit `getline()` in C++

```
#include <iostream.h>

void main()
{
    char buffer[100] };
    cin.getline(buffer,100);
}
```

```

        cout<< buffer <<endl;
    }

```

Der 2. Parameter von `getline()` gibt die maximale Anzahl zu lesender Zeichen an. Er bietet damit eine Schutzfunktion, damit die Grösse von `buffer` nicht überschritten wird. Die Anzahl zu gelesener Zeichen ist immer um 1 grösser als die Zahl tatsächlich eingegebener Zeichen (wg des Stringendezeichens, s. dazu Kap 13)!

Mit `_cgets()` in C

```

#include <iostream.h>
#include <conio.h>      // Für _cgets

void main()
{
    char buffer [30]={27};
    char *buf;
    buf=_cgets(buffer);
    cout<<buf<<endl;
}

```

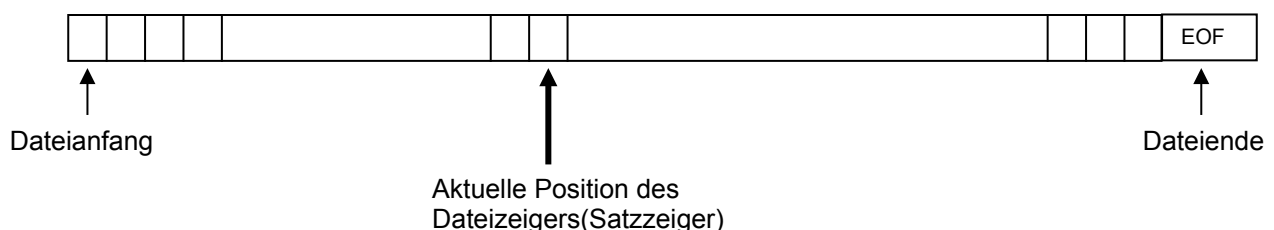
Im ersten Feld von `buffer` muss die maximale Anzahl zu lesender Zeichen eingetragen sein. `buf` ist ein Zeiger und zeigt auf das 2. Feld von `buffer`. Dort steht das erste eingegebene Zeichen. Zu Zeigern siehe unten!

9 Dateioperationen

In diesem Abschnitt werden die grundlegenden Funktionen für Dateioperationen, d.h. Speichern auf Festplatte und Lesen von der Festplatte vorgestellt. Sie arbeiten in sehr ähnlicher Weise wie die Funktionen `printf` und `scanf`. Allerdings ist es erforderlich, einen Zugriff auf die entsprechende Datei zu haben. Dies geschieht durch **Öffnen einer Datei** bzw. **Schliessen einer Datei**. Der **Zugriff auf eine Datei** erfolgt mittels des Dateizeigers. Dieser ist immer vom Typ `FILE*`. Er enthält alle Informationen, welche das Betriebssystem benötigt, um auf die geöffnete Datei zugreifen zu können. Eine geöffnete Datei wird üblicherweise als **stream** bezeichnet.

Vom diesem Dateizeiger, der eine Variable des Programmes ist, ist ein anderer Dateizeiger zu unterscheiden, nämlich ein interner Zeiger des Betriebssystems, welcher auf genau die Stelle der geöffneten Datei zeigt, an welcher die letzte Schreib- oder Leseoperation stattgefunden hat. Dieser Zeiger kann mit speziellen RTL-Funktionen positioniert werden. Das Prinzip veranschaulicht die folgende Darstellung.

Hinweis: Die Daten sind auf der Festplatte byteweise linear, d.h. einfach in aufeinanderfolgenden Bytes, abgespeichert. Hinter dem letzten Byte mit Daten wird durch das Betriebssystem ein spezielles Ende-Symbol eingetragen, das **EOF-Zeichen**



9.5 Auf eine Datei schreiben

Zunächst ist die Datei mit der Funktion `fopen` zu öffnen. Sie enthält als Parameter den Dateinamen und den **Modus des Dateizugriffs**:

```
stream=fopen("C:\\zahlentab.txt", "w");
```

Erster Parameter (Dateinamen):

- Befindet sich die Datei im lokalen Verzeichnis der Anwendung genügt die Angabe des Dateinamens
- Befindet sich die Datei nicht im lokalen Verzeichnis der Anwendung muss der gesamte Pfad zu der Datei angegeben werden. Bei Windows-Betriebssystemen ist das Trennsymbol für die Ordernamen `\` als doppelter `\\` zu schreiben, siehe dazu die Tabelle zu den Escape-Sequenzen!

Zweiter Parameter (Modus): Er legt fest, „was die Anwendung alles darf“.

"w" erlaubt Schreibzugriff. Existiert die Datei nicht, wird sie erzeugt. Existiert die Datei bereits, wird ihr Inhalt gelöscht

"a" erlaubt Schreibzugriff, allerdings werden die (neuen) Daten an die Daten angehängt, welche bereits in der Datei stehen

"r" erlaubt nur Lesezugriff.

"w+" Schreib- und Lesezugriff. Existiert die Datei nicht, wird sie erzeugt. Existiert die Datei bereits, wird ihr Inhalt gelöscht

"a+" Schreib- und Lesezugriff, allerdings werden die (neuen) Daten an die Daten angehängt, welche bereits in der Datei stehen

"r+" Schreib- und Lesezugriff. Datei muss bereits existieren.

Hinweis: Aus Sicherheitsgründen ist es sinnvoll, eine Datei immer im Lesemodus zu öffnen, wenn klar ist, dass die Daten nur gelesen werden sollen

Der Rückgabewert von `fopen` ist ein Zeiger auf eine Variable, welche alle Informationen zum Zugriff auf die geöffnete Datei enthält.

Beispiel: Im folgenden Beispiel werden Ganzzahlen in verschiedener Schreibweise (Dezimal, Oktal und Hexadezimal) in 3 Spalten auf eine Datei geschrieben. Die Datei lässt sich bequem mit einem Editor betrachten und damit der Erfolg kontrollieren.

Für die eigentlichen Schreiboperationen wird die `fprintf`-Funktion verwendet, welche als ersten Parameter den Zeiger auf die geöffnete Datei benötigt, ansonsten aber völlig identisch wie die Funktion `printf` funktioniert.

```
#include <stdio.h>

void main()
{

    FILE *stream;           //Dateizeiger

    //Datei im Modus Schreiben öffnen
    stream=fopen("C:\\zahlentab.txt", "w");
```

```

fprintf(stream, " ++++  Zahlentabelle  ++++\n\n");
fprintf(stream, " Dezimal - Oktal - Hexadezimal\n");

for(int n=0; n<20;n++)
    fprintf(stream, "%3d \t %3o \t %3x \n",n,n,n);

fclose(stream);          //Datei schliessen
}

```

Die `fprintf`-Funktion liefert als Rückgabewert eine Ganzzahl, welche angibt, wieviele Bytes gespeichert worden sind

Schliessen einer Datei: Dies erfolgt mittels der RTL-Funktion `fclose`. Sie sorgt dafür, dass das EOF-Zeichen wieder korrekt angebracht wird.

Die Standard-streams:

<code>stdout</code>	Ausgabe geht auf die Standard-Ausgabe(normalerweise Monitor)
<code>stderr</code>	Ausgabe geht auf die Standard-Ausgabe(normalerweise Monitor)
<code>stdin</code>	Eingabe über die Standard-Eingabe(normalerweise Tastatur)

Die Standard-streams sind immer automatisch geöffnet

9.6 Drucken

Ist ein Drucker direkt am LPT-Anschluss des Rechners angeschlossen, so lässt sich ganz entsprechend dem Schreiben auf eine Datei in einfacher Weise drucken, indem der Drucker wie eine Datei behandelt wird. Es ist lediglich in der Anweisung zum Öffnen der Datei der Dateiname durch "LPT1" zu ersetzen:

```
stream=fopen("LPT1", "w");
```

Es empfiehlt sich, als letzte Druckausgabe einen Seitenvorschub anzuweisen:

```
fprintf(stream, "\f");
```

9.7 Lesen von Datei

So wie die Funktion `fprintf` zum Schreiben auf eine Datei verwendet wird, gibt es die Funktion `fscanf` zum Lesen von einer Datei. Auch sie entspricht in ihrer Funktionsweise der Funktion `scanf`, benötigt aber ebenfalls den Dateizeiger als ersten Parameter. Weiterhin ist auch hierfür zunächst die Datei zu öffnen und zum Abschluss wieder zu schließen.

Das folgende Beispielprogramm liest die angegebene Datei Zeichen für Zeichen und gibt jedes Zeichen direkt auf dem Bildschirm aus. Man bezeichnet dies auch als einen Dump der Datei. Da hier nicht bekannt ist, wie lang die Datei ist, wird bei jedem Byte geprüft, ob es das EOF-Symbol enthält. Zu diesem Zweck gibt die `fscanf`-Funktion das gelesene Zeichen als Rückgabewert zurück.

```

#include <stdio.h>

void main()

```

```

{
    char sw='p', zeichen;
    FILE *datei;           //Dateizeiger
    //Datei zum Lesen öffnen
    datei =fopen("C:\\zahlentab.txt", "r");
    //Einzelne Zeichen lesen und ausgeben
    while (fscanf(datei,"%c",&zeichen)!=EOF)
        printf("%c", zeichen);

    fclose(datei);
}

```

9.8 Neuere Versionen der RTL-Funktionen

Die Funktionen fopen, fprintf und fscanf gelten als potentiell fehlerträchtig. Deswegen wurden modernere Version fopen_s, fprintf_s und fscanf_s (das _s steht für „sicher“) eingeführt, welche wie folgt aufgerufen werden:

```

fopen_s(&datei, "Dateiname", "w");
fprintf_s(datei, "%d", iwert);
fscanf_s(datei, "%d", &i1);

```

9.9 Einige weitere RTL-Funktionen für Dateioperationen

Neben den beschriebenen Funktionen enthält die RTL weitere Funktionen für spezielle Dateioperationen. Eine Übersicht gibt die folgende Tabelle:

Funktionsname	Aktion	Header-Datei
feof	Prüft, ob Dateiende erreicht ist	stdio.h
ferror	Status eines Streams	stdio.h
fgetc, getc	Lesen eines einzelnen Zeichens aus e. Datei	stdio.h
fgetpos	Liefert die augenblickliche Position des Dateizeigers	stdio.h
fgets, gets	Liest einen String	stdio.h
fputc, putc	Schreibt einzelnes Zeichen auf e. Datei	stdio.h
fputs , puts	Schreibt einen String auf e. Datei	stdio.h
fseek	Positioniert Dateizeiger innerhalb einer Datei	stdio.h
ftell	Liefert die augenblickliche Position des Dateizeigers	stdio.h
rewind	Positioniert Dateizeiger auf den Anfang einer Datei	stdio.h

10 Selbstdefinierte Datentypen

In 2.4 wurden die elementaren Datentypen eingeführt. Schlüsselwörter wie `int`, `short`, ... fungieren dabei als Namen für die Datentypen. Bei durch den SW-Entwickler definierten (benutzerdefinierten) neuen Datentypen haben die Typnamen dieselbe Aufgabe. Nach Deklaration des neuen Datentyps können sie somit völlig analog wie die Typnamen der elementaren Datentypen verwendet werden. D. h. man kann

- Variablen dieser Datentypen definieren
- Felder dieser Datentypen deklarieren
- Diese Datentypen in Parameterlisten von Funktionen oder als Rückgabewert verwenden
- Zeiger auf diese Datentypen definieren (s.u.)

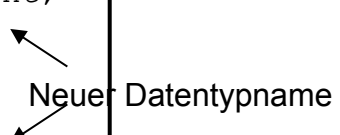
Außerdem wird die Typprüfung durchgeführt !

10.5 *Synonyme definieren mit typedef*

Dies dient nur dazu, neue Typnamen zu definieren. Findet Anwendung, um die Lesbarkeit eines Programmes verbessern, indem Daten mit speziellen Bedeutungen sprechende Namen gegeben werden können

```
typedef unsigned int uint;
uint var1;

typedef unsigned int zeit;
zeit z1;
```



Hier werden die Typnamen `uint` und `zeit` definiert. `var1` und `z1` sind in Wirklichkeit einfach Variablen vom Typ `unsigned int` !

Soll jedoch eine Variablen vom Typ `zeit` (oder `uint`) einer anderen Variablen vom Typ `unsigned int` zugewiesen werden (was im Prinzip ja problemlos möglich ist) so ist eine explizite Typwandlung nötig:

```
typedef unsigned int zeit;
zeit z1=65432;
unsigned int var=(unsigned int)z1;
```

10.6 *Der Aufzählungstyp (Enumeration)*

Es gibt Situationen, wo Variablen nur eine endliche und relativ kleine Anzahl Werte haben können, welche man, somit durchnummerieren kann, so kann man z.B. die Wochentage wie folgt durchnummerieren:

Tag	Nr
Montag	1
Dienstag	2
Mittwoch	3
Donnerstag	4
Freitag	5
Samstag	6
Sonntag	7

Entsprechendes gilt auch für Monatsnamen. Man spricht von einer Aufzählung. Es ist somit möglich, Monatsnamen oder Wochtage mittels Variablen vom Typ `short` oder zu realisieren, wobei die Bedeutung der zugewiesenen Werte entsprechend gegeben ist. Da eine Variable vom Typ `short` (`int`,...) jedoch noch viele andere Werte aufnehmen kann, besteht hier die Gefahr von logischen Programmierfehlern. Dem kann der Aufzählungstyp vorbeugen.

Aufzählungstypen sind somit sinnvoll für Variablen mit begrenztem Wertebereich

Beispiel

```
enum Nachmittag {Nickerchen, Zeitunglesen, Sport,  
BesuchMachen};
```

```
Nachmittag d; //Variable vom Typ Nachmittag
```

- `Nachmittag` ist ein Datentyp-Name
- In der Aufzählungsliste darf jeder Wert nur einmal vorkommen.
- Die in der Aufzählungsliste angegebenen Werte sind nachfolgend Konstanten des Programmes (innerhalb ihres Gültigkeitsbereiches)
- Einer Variablen vom Typ `Nachmittag` können nur Werte zugewiesen werden, die in der Aufzählung aufgeführt sind. D.h. eine Zuweisung von `int`-Werten ist nicht möglich. also

```
d = Nickerchen;  
d = 15; //nicht möglich
```

- Den Bezeichnern der möglichen Werte werden automatisch `int`-Werte für die interne Darstellung zugeordnet. Dies erfolgt in aufsteigender Weise beginnend mit 0, d.h. in obigem Beispiel ist
 - `Nickerchen = 0`
 - `Zeitunglesen = 1`
 - `Sport = 2`
 - `BesuchMachen = 3`
- Diese automatische Wertezuordnung kann durch explizite Angaben beliebig geändert werden:

```
enum Nachmittag { Nickerchen, Zeitunglesen,  
Sport = -3, BesuchMachen = 7};
```

Dies führt zu folgender Wertezuordnung

- Nickerchen = 0
- Zeitunglesen = 1
- Sport = -3
- BesuchMachen = 7

- Die Bezeichner der möglichen Werte können auch als Konstanten in Vergleichen verwendet werden, wie z.B:

```
if(d == Zeitunglesen)
```

Ein häufig benutzter enum -Typ ist:

```
enum Bool {false, true};  
Bool switch1=false, switch2;
```

In C++ ist dieser Datentyp bereits definiert.

10.7 Strukturen (C)

Adressen sind ein Beispiel für **zusammengesetzte Datenstrukturen**, da jede Adresse **Datenelemente** enthält wie

- Name
- Vorname
- Strassenname
- Haus-Nr
- PLZ
- Ortsname
- ...

Dabei können die einzelnen Datenelemente von verschiedenem Datentyp sein.

Für viele Anwendungsfälle ist es nützlich, Variablen solcher zusammengesetzten Datenstrukturen verwenden zu können, welche sich somit aus verschiedenen Datenelementen zusammensetzen. Dies ermöglicht die C-Struktur.

10.7.5 Syntax der Definition

```
// Struktur für Adressen  
typedef struct adresse  
{  
    char Name[25];           //Datenlement der Struktur  
    char Vorname[25];       //Datenlement der Struktur  
    int Plz;                 //Datenlement der Struktur  
    char Ortsname[25];      //Datenlement der Struktur  
    int lfd_nr;             //Datenlement der Struktur  
};  
// Struktur für komplexe Zahlen  
struct complex  
{  
    float re, im;  
};
```


- `adresse` ist der Namen des Datentyps
- das Schlüsselwort `struct` leitet die Definition ein, davor kann `typedef` stehen
- Eine Definition kann auch in einem lokalen Gültigkeitsbereich erfolgen
- Die Definition einer Struktur ist keine Allokation, d.h. für die Datenelemente steht kein Speicher im RAM zur Verfügung. eine Initialisierung innerhalb der Definition der Struktur ist nicht möglich:

```
struct complex
{
    float re=0.f, im=0.f;    //FALSCH
};
```

Man kann eine Struktur-Definition als "Bauplan" für einen zusammengesetzten Datentyp bezeichnen

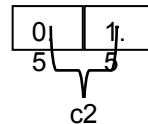
10.7.6 Deklaration von Variablen vom Typ einer Struktur

Beispiele:

```
complex c1, c2={0.5f, 1.5f};    //Initialisierung von c2
adresse einadresse1;           //Variable vom Typ adresse
struct adresse einadresse2;    //Variable vom Typ adresse
```

- Zur Initialisierung kann die Syntax des Feldinitialisierers verwendet werden
- Bei der Variablendeklaration kann optional das Schlüsselwort `struct` vorangestellt werden

Die Variable `c2` sieht im RAM also etwa folgendermassen aus:



Gleichzeitige Definition einer Struktur und Deklaration einer Variablen davon:

```
struct complex
{
    float re, im;
}compl;
```

`compl` ist eine Variable des Typs `complex` !

Ist nur eine einzige Variable vom Strukturtypus erforderlich, so kann auf die Definition eines Datentypnamens verzichtet werden

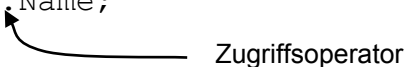
```
struct
{
    char vatername[20];
    char muttername[20];
    char tochtername[20];
    char sohnname[20];
}familie;
```

Vom Typ dieser Struktur gibt es nur die eine Variable `familie`

10.7.7 Zugriff auf Variablen vom Typ einer Struktur

Der Zugriff nur mit dem Variablennamen genügt nicht, hier muss noch das interne Datenelement identifiziert werden, auf welches zugegriffen werden soll. Dies geschieht mittels des **Zugriffsoperators**:

```
cin >> einadresse1.Name;
```



Weitere Beispiele:

```
cin >> einadresse1.Vorname;
einadresse1.Plz = 4711;
cin >> einadresse1.Ortsname;
einadresse1.lfd_nr = 2501;
c1.re=c1.im=1.0f;
.....
cout << einadresse1.Name;
```

Bei einer Zuweisung der folgenden Art

```
complex2 = complex1;
```

werden die Werte der Datenelemente sinnvoll umkopiert, d.h.

```
complex1.re      —————> complex2.re
complex1.im      —————> complex2.im
```

usw.

Achtung: Handelt es sich bei den Datenelementen um Felder (wie bei Vorname usw in adresse), so erfolgt Übertragung der Referenz und nicht des Wertes !

10.7.8 Arrays von Strukturen

Deklaration: adresse adressen[10];

Zugriff: adressen[1].lfd_nr=1;
cin>>adressen[n].Name;

10.7.9 Die tm – Struktur aus der RTL

Zur Aufnahme von Zeit- und Datums-Daten ist in der RTL die Struktur `tm` deklariert Sie wird z.B in der RTL-Funktion `localtime()` verwendet. Nachfolgend wird ihre Definition angegeben.

```
struct tm
{
int tm_sec;        //Seconds after minute (0 - 59)
int tm_min;        //Minutes after hour (0 - 59)
int tm_hour;       //Hours after midnight (0 - 23)
```

```

int tm_mday;    //Day of month (1 - 31)
int tm_mon;     //Month (0 - 11; January = 0)
int tm_year;    //Year (current year minus 1900)
int tm_wday;    //Day of week (0 - 6; Sunday = 0)
int tm_yday;    //Day of year (0 - 365; January 1 = 0)
int tm_isdst;
};

```

Beispiel für die Anwendung:

```

time_t zeit;
time(&zeit);
tm *tm_zeit = localtime(&zeit);
cout<< tm_zeit ->tm_min;

```

Die RTL-Funktion `time` liefert aktuelle Computerzeit als Ganzzahl, welche dann von `localtime` umgerechnet wird in übliche Zeit- und Datumsangaben

10.8 Bitfelder

Bitfelder ermöglichen es, jedem einzelnen Bit oder Gruppen von Bits innerhalb einer Variablen einen eigenen Namen zu geben und sie damit über diese Namen anzusprechen. Die Syntax entspricht weitgehend derjenigen von C-Strukturen.

Deklaration

```

struct flags
{
    unsigned int  flag1:1, flag2:1, flag3:2, flag4:3;
};

```

Stellt 32 Bit zur Verfügung

Bit - Gruppen des Bitfeldes

Grösse der Gruppen in Bit

Wertebereich von flag1: 0, 1

Wertebereich von flag2: 0, 1

Wertebereich von flag3: 0, 1, 2, 3

Wertebereich von flag4: 0, 1, 2, 3, 4, 5, 6, 7

Es müssen nicht für alle verfügbaren Bits (oben 32 Bit) Bitgruppen definiert werden !

Verwendung

```

void main()
{
    flags  my_flags;
    my_flags.flag1=1;
    my_flags.flag2=0;
    my_flags.flag3=3;
}

```

```
my_flags.flag4=6;
cout<< my_flags.flag2;
}
```

Die Gruppen eines Bitfeldes können nur einzeln angesprochen werden

10.9 Die Union

Ausser der Verwendung des Schlüsselwortes `union` entspricht dieses Sprachelement der Struktur. Alles Syntaxregeln sind identisch. Während jedoch bei der Struktur für jedes Datenelement ein eigener Speicherplatz zur Verfügung steht und somit jedes Datelement einen Wert aufnehmen kann, steht bei der Union für alle Datenelemente derselbe Speicherplatz zur Verfügung. Dessen Gesamtgrösse bemisst sich nach dem Bedarf jenes Datenelemente, welches den grössten Speicherplatz benötigt.

Dies bedeutet, dass bei einer Variablen vom Union –Typ die Datenelemente nur aufeinanderfolgend Werte aufnehmen können. In der Praxis findet die Union somit relativ wenig nützliche Anwendungen. Ein Beispiel ist in den nachfolgenden Programmbeispielen angegeben. Da sich bzgl. der Syntax nichts Neues ergibt wird hier nur die Syntax der Definition angegeben:

```
union beispiel
{
    short eineZahl;
    double andereZahl;
};
```

10.10 Ein- und Ausgaben

Mittels der Funktionen `printf`, `scanf`, bzw. `cout`, `cin` sind keine direkten Ein- und Ausgaben möglich. Für zusammengesetzte Datentypen, d.h C-Strukturen ist dies unmittelbar einsichtig, denn bei einer Anweisung der Art

```
cout<<einadressel;
```

ist nicht klargestellt, welche interne Datenelement gemeint ist. Hier kann jedoch so vorgegangen werden, dass für jedes interne Datenelement eine Ein- bzw. Ausgabeanweisung geschrieben wird:

```
cout<<einadressel.Name;
cout<<einadressel.Vorname;
```

Aber auch bei Aufzählungstypen gibt es Einschränkungen. Bezogen auf das Beispiel aus 10.2

```
enum Richtung {Nord, Sued, Ost, West};
Richtung d=West;
cout<<d<<endl;
cin>>d;
```

bedeutet dies, dass in der Ausgabeanweisung nicht, wie vielleicht erwartet, ‚West‘ auf dem Bildschirm steht, sondern die Zahl 3, mit welcher der Wert ‚West‘ intern repräsentiert wird. Die Eingabe-Anweisung funktioniert überhaupt nicht.

Der Grund hierfür liegt darin, dass den Ein- und Ausgabefunktionen der Zusammenhang zwischen der internen Repräsentation der Werte und ihrer Schreibweise nicht bekannt ist. Der Programmierer muss zu diesem Zweck eigene Funktionen schreiben. Ein Beispiel dafür ist in den folgenden Programmbeispielen angegeben.

11 Zeiger

Eine **Zeiger-Variable** enthält als Wert die Adresse eines Ortes im Arbeitsspeicher. Sie gibt eine alternative Möglichkeit, auf Daten im Arbeitsspeicher zuzugreifen. D. h. Wert einer Zeiger-Variable ist die Adresse, auf die sie zeigt.

Aus Gründen der Typsicherheit muss ein Zeiger passend zum Datentyp deklariert werden (Zeiger auf int, Zeiger auf float,). D.h eine Zeigervariable enthält die Information über den Datentyp, auf welchen sie zeigen kann.

Einer Zeigervariablen kann mittels des **Referenzoperators &** (Adress-Operator) ein Wert zugewiesen werden, dabei handelt es sich um die Adresse einer Variablen: Der Zeiger zeigt auf die Variable.

11.5 Zeiger – Deklaration

Syntax

```
int*pname1, *pname2;  
int * pname;  
char * cname;  
float *pname;
```

Datentyp Verweisoperator Name der Zeigervariablen

- Mit dem **Verweisoperator**, der eine Typinformation darstellt, wird eine Zeigervariable gekennzeichnet
- Für jede Zeigervariable muss der Verweisoperator vor den Variablennamen geschrieben werden
- Zeigervariable können auch in Parameterlisten von Funktionen, als Rückgabewert oder als Datenelemente von Strukturen verwendet werden

In folgendem Beispiel sind `chr1`, `chr2` normale Variable, `pch` jedoch eine Zeigervariable

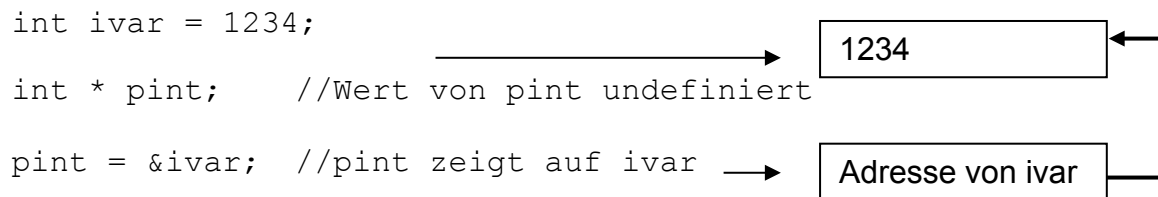
```
char chr1, *pch, chr2;
```

Hinweis: Die Namen von Arrays sind ebenfalls Zeiger, welche als Wert die Adresse des ersten Feldelementes enthalten, also des Feldelementes mit dem Index 0!

11.6 Wertzuweisung an Zeigervariable

Diese erfolgt mit dem Adressoperator

Im RAM



11.7 Dereferenzieren

Hierunter versteht man den Zugriff auf den Inhalt der Adresse, auf welche ein Zeiger zeigt: Dies geschieht mit dem **Dereferenzoperator** `*`.

Beispiel:

```
float fvar1=0.3456, fvar2=4.23232;
float *pfloat;
pfloat = &fvar1;
cout << *pfloat << endl;
fvar2 *= *pfloat;
```

↑ ↖
Multiplikaton Dereferenzieren

Weitere Beispiele:

```
float * pfloat = &fvar1;
//Multiplikation von fvar2 mit fvar1 durch Dereferenzieren von
//pfloat!
fvar3 = fvar2 ** pfloat;
```

Als Regel kann man sich merken:

Wird ein Zeiger ohne `*` benutzt geht es um Adressen
Wird ein Zeiger mit `*` benutzt geht es um Werte

11.8 Typwandlung bei Zeigern

Obwohl Zeiger als Wert immer Adressen enthalten ist ggf eine Typwandlung erforderlich. Hiermit wird die Information über den Typ, auf welchen ein Zeiger zeigen darf, geändert.

Syntax:

```
ptr2 = (char*) ptr1;
```

Dabei ist darauf zu achten, dass die Daten an dem Speicherort, auf welchen ein Zeiger zeigt, die gewünschte Typwandlung auch erlauben!

11.9 Zeiger auf void

Dieser kann auf jeden Datentyp zeigen

Beispiel:

```
int ivar=3434;
float fvar=321.1F;
int *iptr;
float *fptr;
void* pt = &ivar;    //Zeiger auf void
iptr = (int*) pt;    //Type-Cast
cout<<*iptr<<endl;
pt = &fvar;
fptr = (float*) pt;  //Type-Cast
cout<<*fptr<<endl;
```

11.10 Der Null-Zeiger

Der **Null-Zeiger** dient zur Initialisierung von Zeigervariablen, denn diese sind wie andere lokale Variable ebenso undefiniert, wenn ihnen kein Wert explizit zugewiesen wurde. Da ein Zeiger im Prinzip auf den gesamten Adressraum zeigen kann, sind undefinierte Zeiger potentiell sehr gefährlich (Laufzeitfehler)

Zur Initialisierung ist die Konstante NULL definiert

Syntax:

```
char *cpoint = 0;
char *cpoint = NULL; }
```

 Äquivalent

11.11 Referenzübergabe bei Zeigern

Soll in der Parameterliste für eine Zeigervariable Referenzübergabe definiert werden, so geschieht dies mit folgender Syntax

```
void funk(float *&pfloat)
```

11.12 Zeiger- Arithmetik

Mittels der Zeiger-Arithmetik können die Werte von Zeiger-Variablen verändert werden. Dies dient dazu, auf anderen, normalerweise benachbarte Speicherorte zu verweisen. Dabei wird der Wert von Adressen so verändert wie es dem Datentyp des Zeigers entspricht. Durch die Anweisung

```
pointer++;
```

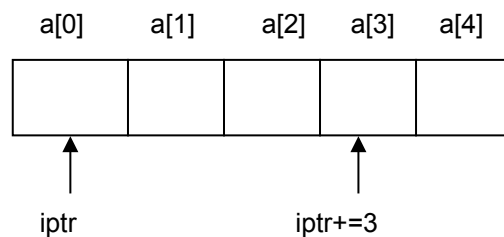
wird der Wert von `pointer` somit um 1 erhöht (=1 Byte) wenn `pointer` ein Zeiger auf `char` ist. Er wird jedoch um 4 erhöht (=4 Byte) wenn `pointer` ein Zeiger auf `float` ist.

Syntax:

```
float *fpoint = &Wert;
fpoint +=3;
fpoint--;
```

Beispiel:

```
int a[5];
int *iptr = a;
iptr+=3;
```



Hierbei besteht allerdings die Gefahr des Überschreitens der Feldgrenzen!

Beachte folgende Alternativen:

Variante 1:

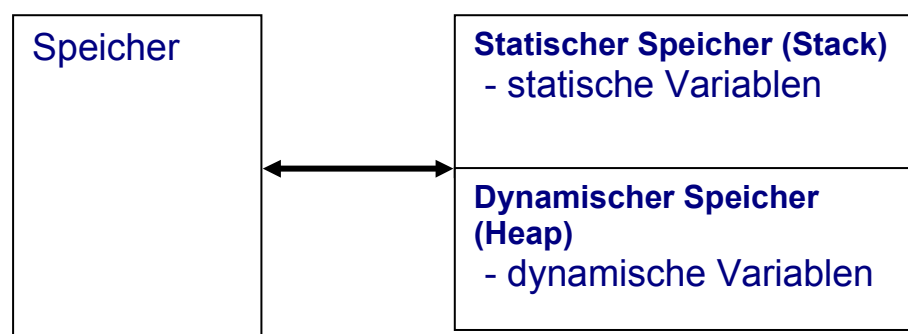
```
float my_feld[10];
float *pfloat=& my_feld[0];
for(int l=0;l<10;l++)
{
    *pfloat=float(l);
    pfloat++;
}
```

Variante 2:

```
float my_feld[10];
float *pfloat=& my_feld[0];
for(int l=0;l<10;l++)
    *pfloat++=float(l);
```

11.13 Dynamische Speicherverwaltung

Der RAM-Adressraum lässt sich prinzipiell in 2 Bereiche unterteilen:



Die Grenze zwischen beiden Bereichen ändert sich dynamisch. Deklarierte Variable finden ihren Platz im Stack-Bereich, man nennt sie **statische Variable**.

```
int ivar=1;
float f_var=7.1;
char *string="Hallo";
```



statische Variable

Da es in vielen Situationen während der SW-Entwicklung nicht absehbar ist, welche Mengen an Daten ein Programm in Zukunft bearbeiten soll, ist es sinnvoll, wenn das Programm selbstständig nach Bedarf weiteren RAM-Speicher vom Betriebssystem anfordern kann. Man spricht hierbei von **dynamischer Speicherverwaltung**. Der zusätzliche Speicherbedarf wird aus dem **Heap** gedeckt.

Das Anfordern von Speicher aus dem Heap heißt auch Speicherreservierung, dabei entsteht eine **dynamische Variable**. Auf diese kann nur über die Adresse zugegriffen werden, d.h. mittels einer

Zeigervariablen. Ebenso muss dem Betriebssystem mitgeteilt werden, dass dynamisch angeforderter Speicher nicht mehr benötigt wird. Dies nennt man Speicherplatzfreigabe.

In C++ stehen für die dynamischer Speicherverwaltung die Befehle `new` und `delete` zur Verfügung, deren Verwendung nachfolgend beschrieben wird.

In **C** stehen für die dynamische Speicherverwaltung die RTL-Funktionen `malloc` und `free` zur Verfügung (wird hier nicht behandelt)

11.13.5 Dynamische Speicherreservierung

Eine dynamische Variable wird erzeugt mittels des Operators **new**, er benötigt als Argument (jedoch nicht in Klammern geschrieben) den gewünschten Datentyp und liefert als Rückgabewert die Adresse des Speicherortes. Dieser Prozess heisst auch **Speicherallokation**.

Beispiel:

```
int *ptr = new int;
char *str = new char[n];           //Dyn. reserviertes Feld

int * ptr1;
....
ptr1 = new int;
*ptr1 = 32;
str[6] = 's';                      //Zugriff mittels Index!
```

Mit der Anweisung `*ptr = new int;`

wird genau so viel dynamischer Speicher reserviert, wie für eine `int`-Variable (2 oder 4 Bytes)

Mit der Anweisung `*str = new char[10];`

werden sequentiell 10 Bytes reserviert, d.h. für 10 Variable vom Typ `char`. Hier handelt es sich um ein dynamisch reserviertes Array

Wichtig: Auf die Daten, welche in dynamischen Variablen angelegt sind, kann nur mittels der Adresse zugegriffen werden. D.h. geht dieser Wert (die Adresse) verloren, sind diese Daten nicht mehr erreichbar.

Dynamisch erzeugte Variable bestehen standardmässig so lange, bis das Programm terminiert. D.h. werden dynamische Variable lokal innerhalb von Funktionen allokiert, so bestehen diese (mit den darin enthaltenen Daten) bis zum Ende des Programmlaufes. Dies gilt auch, wenn die Zeigervariable, welche die Adresse der dynamischen Variablen enthält, lokal innerhalb der Funktion ist und damit beim Verlassen der Funktion verlorengeht. D.h. nach Verlassen der Funktion sind in diesem Fall die Daten nicht mehr erreichbar, obwohl sie noch im Arbeitsspeicher vorhanden sind!

11.13.6 Speicher-Freigabe

Dynamisch belegter Speicher muss durch den Programmierer explizit freigegeben werden. Diese erfolgt mittels des Befehles `delete`.

Syntax

```
delete ptr;           // delete an object
delete [] str;        // delete an array of objects
```

- Bei der Freigabe eines Arrays ist keine Grössenangabe erforderlich, es wird automatisch die korrekte Menge an Bytes freigegeben !
- Wird `delete` angewandt auf einen Zeiger, der nicht auf eine dynamisch erzeugte Variable zeigt, so führt dies zu einem Laufzeitfehler !
- `delete` kann ohne Probleme auf den Nullzeiger angewandt werden

11.13.7 Speicherplatzgrenzen

Wegen des begrenzten Speichers besteht immer die Möglichkeit, dass kein dynamischer Speicher mehr zur Verfügung steht. `new` gibt in diesem Fall den Wert `0` zurück (Sollte jedenfalls). Der Programmierer muss die entsprechende Reaktion implementieren.

11.14 Zeiger auf Variablen vom Typ einer C - Struktur

Es gilt generell, dass Zeigervariablen auch auf benutzerdefinierte Datentypen deklariert werden können. Syntaxmässig Neues ergibt sich jedoch lediglich bei Zeigern auf zusammengesetzte Datentypen (CStruktur, Union).

Als Zugriffsoperator und gleichzeitig zur Dereferenzierung dient :



Beispiel:

```
#define MAX_LAENGE 256

struct buch //Definition der Struktur buch
{
    char title[MAX_LAENGE ];
    char autor[MAX_LAENGE ];
    char verlag[MAX_LAENGE ];
    int nr;
};

void main()
{
    buch * pbuch = new buch; //Variable dynamisch erzeugt

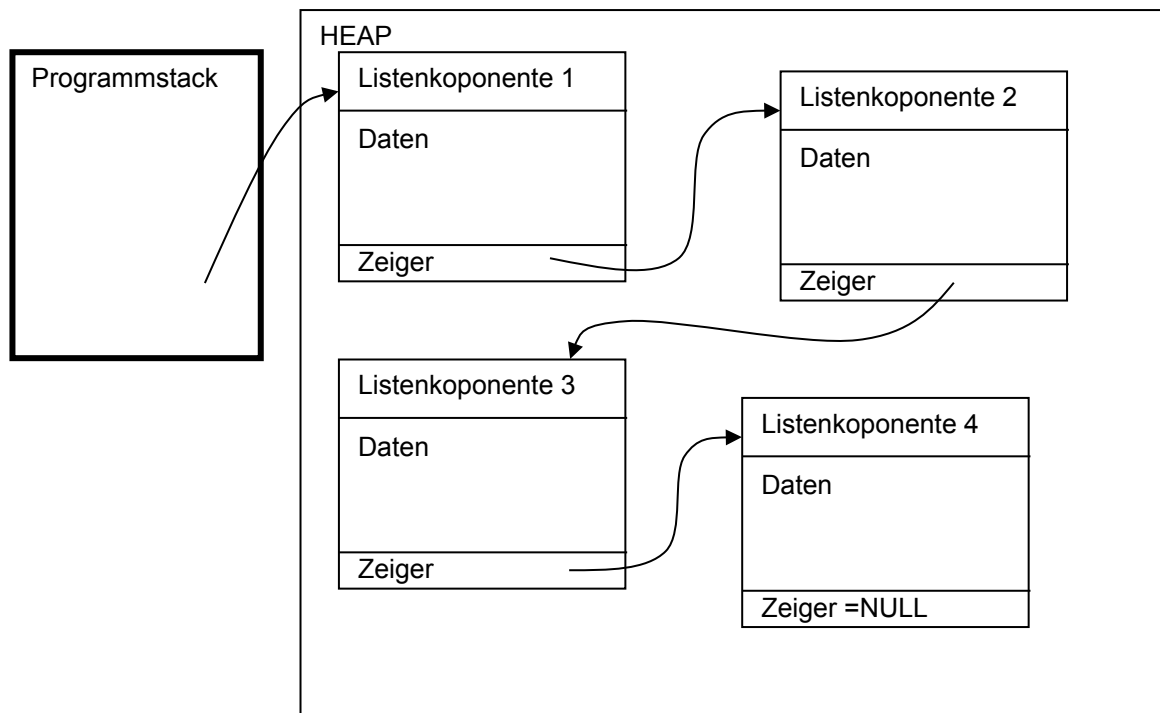
    cin>>pbuch->title;
    cin>>pbuch->autor;
    cin>>pbuch->verlag;
    pbuch->nr=4;

    cout<<endl<<pbuch->title<<endl<<pbuch->autor<<endl;
    cout<<pbuch->verlag<<endl<< <<endl;
}
```

11.15 Verkettete Listen

Mittels verketteter Listen lässt sich in sehr flexibler Weise mit dynamischem Speicher operieren. Die Idee kann man folgendermassen darstellen.

Dynamisch reservierter Speicher wird aus mehreren gleichartigen ‚Portionen‘ zusammengesetzt, erweitert, umgebaut, verkleinert, eben je nach Bedarf. Dazu definiert man die Komponenten in der Weise, dass sie eine Menge an Daten aufnehmen und einen Zeiger auf eine Komponenten desselben Typs:



Der Vorteil des Verfahrens liegt darin, dass, dass innerhalb des Programms eine einzige Variable genügt – ein Zeiger auf den Datentyp, aus welchem die Komponenten bestehen – um damit auf ein potentiell sehr grosses Speichervolumen flexibel zugreifen zu können. Man muss dazu quasi einfach innerhalb der Liste von Komponenten zu Komponente ‚springen‘, bis man bei der letzten angelangt ist. Man erkennt an dieser Stelle auch, dass es wesentlich ist, die letzte Komponenten daruch zu kennzeichnen, dass deren Zeiger = NULL gesetzt wird! Die Verbindung von einem Listenelement zum andern mittels Zeiger nennt man **Verkettung**.

Zum Aufbau der verketteten Liste benötigt man somit das Listenelement und hierfür den entsprechenden Datentyp. Hierfür verwendet man die C-Struktur:

```

struct adresslist //Definition der Struktur
{
    int laufende_Nr;
    char name[20];
    char vorname[20];
    char ort[20];
    int plz;
    adresslist *next;
};
  
```

} Datenbereich

← Dient der Verkettung

Im Programm benötigt man einen Zeiger, welcher auf den Beginn der Liste zeigt, ein 2. Zeiger desselben Typs dient dazu, auf ein beliebiges Listenelement zu zeigen. Im folgenden

Programmausschnitt wird innerhalb einer `for`-Schleife eine Anzahl von Listenelementen erzeugt, was eher nicht der üblichen Vorgehensweise entspricht, wo ein Listenelement nach dem anderen je nach Bedarf erzeugt wird. Hier soll jedoch das Verfahren erläutert werden.

```

.....
adresslist *first; //Deklaration von 2 Zeigern vom Typ
test
adresslist *current;
first=NULL;
.....
//Erzeugen der verketteten Liste
for(int i=0;i<5;i++)
{
    current =new adresslist; //Neues Listenelement
    current-> laufende_Nr =i;
    current->next=first;      //Verkettung
    first=current;
}

```

Nach Ende der Schleife zeigt `first` auf den Anfang der verketteten Liste. Dieses ist das erste Listenelement, jedoch jenes, welcher zuletzt erzeugt wurde. Die Liste wird über die Verkettung in umgekehrter Reihenfolge der Erzeugung durchlaufen!

Hinweis: Man macht sich den Entstehungsprozess der Liste am Besten klar, indem man die Stufen Schritt für Schritt grafisch veranschaulicht (Zeichnet) und dabei Zeiger in der Darstellung einzeichnet

Ausgabe aller Werte der Listenelemente, indem die Liste sequentiell durchlaufen wird

```

.....
current=first; //current auf den Anfang der Liste setzen

while(current!=NULL) //Prüfen, ist Ende der Liste
erreicht
{
    cout<<current->name<<endl;
    .....
    current=current->next; //Zum nächsten LE weitergehen
}

```

In verketteten Listen lassen sich bequem an beliebiger Stelle weitere Listenelemente einfügen, entfernen, man kann weitere Listenelemente anfügen, man kann sie nach Werten durchsuchen, usw. Die Verkettung kann auch in beiden Richtungen erfolgen, so dass die mittlere **Zugriffszeit** auf die Listenelemente verkürzt wird.

Weiterhin ist darauf hinzuweisen, dass der Dateninhalt der Listenelemente von der Verkettung und der Vorgehensweise für den Umgang mit der verketteten Liste vollkommen unabhängig ist. Die verkettete Liste stellt somit ein universelles und flexibles Instrument dar!

11.16 Funktionszeiger

Funktionszeiger nehmen die Adresse einer Funktion auf. Sie haben in der Praxis eine eher geringe Bedeutung und werden hier nur kurz behandelt

11.16.5 Der Typ einer Funktion

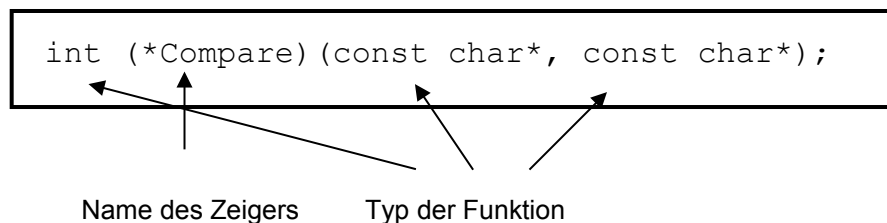
Unter dem Typ einer Funktion werden die Angaben von Rückgabetyt und Parametertypen verstanden

Beispiel: `void funktion(int*, char, char)`

Typ des Beispiels: `void(int*, char, char)`

11.16.6 Deklaration eines Funktionszeigers

Syntax



Der Zeiger aus dem Beispiel heisst "Compare". Eine Funktion, auf die er weist, kann einen beliebigen Namen haben, sie muss jedoch

- denselben Rückgabetyt und
- dieselbe Parameterliste h

haben. Dies gilt im Beispiel für die RTL-Funktion `strcmp`, welche dazu dient, 2 Zeichenketten zu vergleichen (s.u)

11.16.7 Zuweisung eines Wertes an einen Funktionszeiger

Folgende Alternativen sind möglich:

Alternative 1: `Compare = &strcmp;`

Alternative 2: `Compare = strcmp;`

Deklarieren und Initialisieren in einer Anweisung:

```
int (*Compare)(const char*, const char*) = strcmp;
```

11.16.8 Funktionsaufruf mittels Funktionszeiger

Dies wird an obigem Beispiel demonstriert:

```
Compare = strcmp;
strcmp("Tom", "Tim");           // direkter Aufruf
(*Compare)("Tom", "Tim");       // indirekter Aufruf
Compare("Tom", "Tim");          // indirekter Aufruf (verkürzt)
```

12 Zeichenketten

Bei Zeichenketten besteht das Problem darin, dass ihre Länge (=Anzahl der Zeichen) extrem variabel ist.

Um einen einfachen Zugriff auf die gesamte Zeichenkette – die ja eigentlich einen einzigen ‚Wert‘ darstellt – zu ermöglichen wird sie in C in folgender Weise realisiert

- In einem Zeiger auf char wird die Adresse des ersten Zeichens gespeichert
- Die Zeichenkette selbst wird sequentiell im RAM abgespeichert.
- Als Abschlusszeichen wird in dem dem letzten Zeichen folgenden Byte \0 eingetragen

Was hiermit beschrieben wurde nennt man einen **C-String**.

Beispiele: Beide alternative Schreibweisen sind möglich

```
char str1[] = "HELLO";
char *str2 = "Guten Morgen";
```

Darstellung im RAM

		G	u	t	e	n		M	o	r	g	e	n	\0			
--	--	---	---	---	---	---	--	---	---	---	---	---	---	----	--	--	--

Der Compiler sorgt bei den obigen Deklarationen automatisch dafür, dass das Ende-Zeichen eingefügt wird. Somit belegt der String `str1` 6 Bytes und der String `str2` 13 Bytes. D.h. diese Speicheradressen sind für den jeweiligen String belegt.

Wird allerdings ein Feld vom Typ `char` mit vorgegebener Länge definiert, so wird bei der Initialisierung das Endezeichen `'\0'` nicht eingetragen:

Beispiel: `char str3[] = {'H', 'E', 'L', 'L', 'O'};`

Hier ist `str3` ein Feld mit 5 Elementen und kein C_String!

Damit `str3` ein C-String ist, müsste es folgendermassen initialisiert werden:

```
char str3[] = {'H', 'E', 'L', 'L', 'O', '\0'};
```

Man erkennt, obwohl das **Stringendezeichen** mittels 2 Zeichen im Code notiert wird, gilt es als 1 Zeichen!

Der Unterschied zwischen normalen Zeigern und C_Strings ist nun, dass die Ein- und Ausgabefunktion das Dereferenzieren überflüssig machen. In folgende Beispiel

```
int *pint=new int;
*pint=555;
cout<<pint;    //Ausgabe der Adresse
char* str="Beispiel";
cout<<str<<endl;    //Ausgabe der Zeichenkette
```

wird bei der Ausgabe von `print` die Adresse, auf welche `print` zeigt, dargestellt. Bei der Ausgabe von `str` hingegen wird die Zeichenkette ‚Beispiel‘ dargestellt.

Bei diesem Zugriff (hier zur Ausgabe) wird die Zeichenfolge im Speicher beginnend vom ersten Byte, auf das der Zeiger `str` zeigt, entsprechend aufsteigenden Adressen soweit durchlaufen, bis das Endezeichen `'\0'` angetroffen wird. Das bedeutet, dass bei fehlendem Endezeichen in Speicherbereiche eingedrungen wird, in welchen sich fremde Daten befinden (=Feldüberschreitung)!

Die Eingabefunktionen (`scanf`, `fscanf`, `cin`) fügen automatisch das Stringendezeichen an.

Hinweis: Um über einen `char *`-Zeiger Zugriff auf ein einzelnes Zeichen eines Strings zu bekommen muss man dereferenzieren:

`cout<<*str2<<endl;` \longrightarrow Ausgabe: **G**

12.5 Bibiotheks-Funktionen für Strings

Für die einfachere Verwendung von C-Strings stellt die RTL spezielle Funktionen zur Verfügung. Diese arbeiten mit Zeigern als Argumente.

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

Funktion	Beschreibung
<code>strcpy(char *dest, const char* source)</code>	Kopiert String von <code>source</code> nach <code>dest</code>
<code>strcat(char *dest, const char* source)</code>	Hängt String <code>source</code> an <code>dest</code> an
<code>Int strlen(char* source)</code>	Bestimmt die Länge des übergebenen Strings und liefert den Wert als Rückgabewert
<code>int strcmp(char* st1, char* st2)</code>	Vergleicht die Strings <code>st1</code> und <code>st2</code> , als Rückgabewert wird nur dann 0 geliefert, wenn beide Strings völlig gleich sind!

Beispiele:

```
char str1[10];
cin>>str1;      //Die Eingabe von mehr als 9 Zeichen ist
möglich,
                //führt jedoch zu einem Absturz des Programmes!
scanf("%s",str1);      //Die Eingabe von mehr als 9 Zeichen
ist //möglich, führt jedoch zu einem Absturz des
Programmes!

char str2[5];
strcpy(str2,str1); //Kopiert den Inhalt von str1 nach str2. Da
//für str2 in diesem Fall nicht genügend Speicher
```

```
reserviert //wurde, werden fremde Daten überschrieben →  
Absturz des //Programmes!
```

```
char str3[25];  
strcpy(str3,str1); //Kopiert den Inhalt von str1 nach str3.  
Hier //stehtgenügend reservierter Speicher zur Verfügung
```

12.6 Sicherheitsaspekte

Die Funktionen `strcpy`, `strcat` fügen zwar das erforderliche Stringendezeichen an. Sie überschreiben jedoch beliebig Bytes im RAM, falls an dem Ort, wohin sie kopieren, nicht genügend Speicherplatz zur Verfügung steht. Dies kann Laufzeitfehler zur Folge haben. Sie gelten deshalb als unsichere Funktionen und der Anwender muss sicher stellen, dass am Zielort genügend freier Speicherplatz zur Verfügung steht.

Aus diesen Gründen gibt es die moderneren Funktionen `strcpy_s` und `strcat_s`, welche als zusätzlichen Parameter die Grösse des Zielzeichenfolgenpuffers haben, damit diese Grösse nicht überschritten wird:

```
strcpy_s(char *dest, size_t numberOfElements, char* source)  
strcat_s(char *dest, size_t numberOfElements, char* source)
```

`size_t` ist ein in C vordefinierter vorzeichenloser Ganzzahltyp (=unsigned int)

Ein einfacherer Weg besteht jedoch darin, die im Fall der Verwendung von `strcpy`, `strcat` erzeugten Warnungen oder Fehler zu unterdrücken. Dazu gibt es 2 Wege:

- Man schreibt folgende Anweisung als erste in die Quellcodedatei:

```
#define _CRT_SECURE_NO_WARNINGS
```

- Bei Windows-BS verwendet man die Anweisungen:

```
#if defined( _WIN32 )  
#pragma warning(disable:4996)  
#endif
```

13 Hinweise zur Programmerstellung

Nachfolgend einige Empfehlungen zum Programmierstil, deren Zweck darin besteht, auf möglichst einfache Weise übersichtlichen und möglichst fehlerfreien Code zu erstellen

- Das Mindeste, das jedem SW-Entwickler zu empfehlen ist, ist, für sich einen eigenen passenden Programmierstil zu finden und konsequent anzuwenden. Nur wenn man sich im eigenen Programmen "zurechtfindet"; ist eine sinnvolle Fehlersuche und Programmpflege möglich, und nur dann können gute und bereits bestehende Teillösungen in neue Projekte einfließen. Erst dies macht ein effektives, ökonomisches und erfolgreiches Arbeiten möglich.

- Nachdem die Aufgabenstellung vorliegt, erfolgt die Aufgabenanalyse, wo man das "Problem gedanklich durchdringt". Anschließend erfolgt das Entwerfen alternativer Lösungskonzepte. Hier findet Ihre eigentliche kreative Arbeit statt. Hierbei sollte man sich Zeit nehmen und entspannt arbeiten. Es hilft, Ideen, Entwürfe, Schnittstellendefinitionen usw. festzuhalten (aufzunotieren). Denn die Zeit, die man hier investiert, spart man später bei der Fehlersuche und Programmweiterungen wieder ein.
- Datenkapselung einsetzen. Auf globale Daten, Funktionen verzichten
- Das Rad nicht jedes Mal neu erfinden, d.h. bestehenden Code aus anderen Projekten, Standard-bibliotheken verwenden
- Namen für Variablen usw. sollten so kurz wie möglich, aber aussagekräftig sein. Es sollte aus dem Namen schon hervorgehen, ob es sich um eine Variable, Konstante usw. handelt. Hierzu gibt es z. B. die "ungarische Notation". Oder man verwendet eigene kurze Präfixe oder setzt gezielt Groß- und Kleinschrift ein. Eine andere Möglichkeit ist z. B. die Verwendung von Substantiven für Variablen, Verben für Klassen usw.. Bei Abkürzungen daran denken, dass der Wortanfang wichtiger als das Wortende ist und Konsonanten wichtiger als Vokale.
- Lesbare Programme schreiben. Auch wenn viele C - Kritiker es nicht wahrhaben wollen, ein "stilistisch" gut geschriebenes C – Programm kann leserlicher als z.B. ein schlecht geschriebenes PASCAL – Programm sein.
- In jede Zeile nur eine Anweisung
- Durch redundante Leerzeichen und Klammern die Ausdrücke leserlich gestalten.
- Bei Kontrollstrukturen (Blöcken) Einrückungen vornehmen. Zusammengehörende öffnende und geschweifte Klammern an dieselbe Position der Zeile (übereinander) schreiben.
- C++ bietet mittels `/**` die Möglichkeit, viele ergänzende Kommentare einzusetzen, ohne die Übersichtlichkeit der Programmstruktur zu zerstören, indem diese rechts neben Anweisungen geschrieben werden.
- Den Quelltext durch Leerzeilen und Bereiche für Deklarationen, Definitionen und Anweisungen gliedern.
- Einen standardisierten Programmkopf erstellen, der z.B. den Namen des Programms, eine kurze Funktionsbeschreibung, Version, Datum und Autor enthalten kann
- Objektorientierte Lösungsstrategien suchen, d.h. man sollte ein System von Objekten vor Augen haben, die untereinander kommunizieren, sich gegenseitig zu Aktivitäten auffordern und Daten untereinander austauschen.

Programmierrichtlinien:

- Viele Firmen haben eigene Programmierrichtlinien eingeführt, die schriftlich fixiert sind und an welche sich die Mitarbeiter der Firma zu halten haben.
- Bei grösseren Projekten können firmenübergreifend Programmierrichtlinien festgelegt und angewendet werden.
- In der Automobilindustrie wird verbreitet der Programmierstandard MISRA-C angewendete, deren sowohl feste Regeln wie auch Empfehlungen enthält.