Grundlagen der Informationsverarbeitung II

Skript der Vorlesung

Sommersemester 2013

Dipl.-Ing. (FH) Thomas Braun

Inhalt

Referenzen

[1] http://de.wikipedia.org/wiki/Von-Neumann-Architektur

[2] http://msdn.microsoft.com/de-de/library/3bstk3k5(v=vs.100).aspx

[3] http://msdn.microsoft.com/en-us/library/126fe14k.aspx

Lektionen der Vorlesung

Lektion	Thema	Behandelte Topics
1	Einführung, Aufbau eines PC	Vorstellung des Dozenten Literaturhinweise Vor-/Nachteile C++ Zahlensysteme Aufbau eines Rechners Prozesse und Threads C++ Code Generierungsprozess
2	Grundlegendes der C++ Sprache	Dateitypen Elemente einer C++ Datei Präprozessor Direktiven Trennung Deklaration und Definition Namensregeln Einfache „main“ Funktion Negative Digitale Zahlen Wichtige Standardbibliotheken Ein- und Ausgabe auf der Konsole Variablen Konstanten Integrale Datentypen
3	Datentypen und Operationen	Arithmetische Operatoren Kombinierte Operatoren Datentyp Konvertierung Overflow / Underflow Boolsche Datentyp Logik Operationen Bit Manipulationen
4	Kontrollstrukturen	Blockanweisung if-Anweisung switch-Anweisung while-Schleife do-while-Schleife for-Schleife break/continue
5	Funktionen	Funktionen Felddatentypen Aufzählung Strukturdatentyp
6	Zeichenketten und Zeiger	char für Zeichen char-Feld

		Nullterminierung ASCII Tabelle String-Funktionen Konstante Strings Strings als Argumente
7	Speicher	Zeiger als Speicher von Adressen Operatoren für Zeiger Zeiger und Arrays Globaler Speicher Stapel-Speicher Heap-Speicher Operatoren new und delete
8	Dateizugriff	Textdateien lesen und schreiben Binärdateien lesen und schreiben
9	Klassen	Klassenaufbau Konstruktor Destruktor Methoden Eigenschaften Vererbung Sichtbarkeit von Klassen
10	Klassen Teil 2	Virtuelle Methoden Statische Methoden Überladen von Methoden Überladen von Operatoren
11	Datenstrukturen	Array, Liste, Dictionary, Queue, Stack
12	Bibliotheken	Quellcode Bibliothek Statische Bibliothek Dynamische Bibliothek Weiterführende Themen

Lektionen des Praktikums

Übung	Thema	Behandelte Topics
1	Arbeiten mit einer IDE	Projekte verwalten Ansichten für den Zugriff Codeeditor Projekte Bauen Debuggen Breakpoints Watchfenster Memory Call-Stack
2	Datenein-/ausgabe und Variablen Mathe Bibliothek und Datentypen	cin und cout Ausgabe formatieren Integrale Datentypen Fließkommazahlen Wertebereiche von Datentypen Funktionen der Math.h Bibliothek Parallelschaltung zweier Widerstände berechnen
3	Kontrollstrukturen und Felder Funktionen und Konstanten	If-Anweisung While Schleife For Schleife Arbeiten mit Feldern Funktionen Parameterübergabe - By Reference - By Value Switch Anweisung Menü mit Konsole
4	Zeichenketten, Strukturen und Dateien	ASCII Tabelle Kopieren von Strings Vergleichen von Strings Strukturen Textdatei lesen und schreiben Binäre Datei lesen und schreiben
5	CNC Fräsmaschine	Klassen und Objekte
6a	Projekt	Signalgenerator bauen
6b	Projekt	Eine WAV Datei lesen, bearbeiten und abspeichern - Datei binär einlesen - Strukturen - Zeiger - Datenformat einer WAV Datei - Tiefpass - Samplen von Daten - Datei speichern

Lektion 1

Vorstellung des Dozenten

Mein Name ist Thomas Braun. Ich habe 8 Semester an der Fachhochschule der Deutschen Telekom in Dieburg Nachrichtentechnik studiert und habe mit dem Dipl.-Ing. (FH) das Studium abgeschlossen.

Ich habe nach meinem Studium ca. 3 Jahre bei „Schneider Electric GmbH“ in Seligenstadt als Softwareingenieur gearbeitet und vor allem im Bereich Windows Programmierungen Erfahrungen gesammelt.

Danach habe ich ca. 4 Jahre bei „Vibradorm GmbH“ in Michelstadt als Entwicklungsingenieur gearbeitet. Dabei habe ich Elektronik Produkte für die Ergonomiemöbelbranche entwickelt. Meine Tätigkeiten beinhalteten Hardware- und Softwareentwicklungen, d.h. Gehäuseentwicklung, Leiterplattenlayout und Fertigung, Mikrocontrollerprogrammierung und Desktopanwendungen. Die Produkte sind von der Idee bis zur Lieferung an den Kunden zu betreuen gewesen.

Seit 2007 arbeite ich als Softwareingenieur und Unternehmensberater bei “Xox Industrie IT GmbH“ in Aachen. Dort erstelle ich Lösungen für steuer- und messtechnische Probleme in der Industrie, die wir mit Software umsetzen. Hier kommen vor allem Windowssystem oder Echtzeitbetriebssystem wie etwa QNX zum Einsatz.

Seit 2010 führe ich regelmäßig einen Lehrauftrag an der FH Aachen im Fachbereich 10 Elektrotechnik aus.

Ich bin über meine Emailadresse thomas.braun2@fh-aachen.de für alle Fragen und Anmerkungen erreichbar.

Literaturhinweise

- Einführung in die Programmierung mit C++
  Bjarne Stroustrup
  Pearson Studium
  ISBN 978-3868940053
  (Einführung)

- Object-Oriented Programming in C++
  Nicolai M. Josuttis
  Wiley
  ISBN 978-0470843994
  (Einführung)

- C++ Kurs – technisch orientiert
  G. Schmitt
  Oldenburg Verlag
  ISBN 3-486-24318-7
  (Einführung)

- Die C++ Programmiersprache
  Bjarne Stroustrup
  Addison-Wesley

ISBN 3-89319-386-3
(für Ambitionierte)

- The Art of Computer Programming, Volumes 1-4
  Donald E. Knuth
  Addison-Wesley Longman
  ISBN 978-0321751041
  (für Ambitionierte)

- Webseite: www.cpluplus.com
  (Gute Tutorien und Referenz)

- Webseite: www.c-plusplus.de
  (Links zu kostenlosen Tutorien und weitere Informationen)

- Webseite: http://www.cpp-tutor.de
  (Gutes Tutorium)

Vor-/Nachteile C++

Die folgende Tabelle listet verschiedene Aspekte der Sprache C++ auf, die nicht direkt als Vergleich mit anderen Sprachen zu verstehen ist.

Vorteile	Nachteile
Grundsätzlich für alle Anwendungen einsetzbar.	Schwierige Einarbeitung, da die Sprache sehr umfangreich ist und die Syntax als kompliziert empfunden wird.
native Sprache, das Ergebnis sind Maschinenbefehle die direkt vom Prozessor ausgeführt werden.	Fehleranfällig, da vieles erlaubt ist, können auch leicht Fehler gemacht werden. Das Vermeiden von Fehler muss durch das Lernen von Konzepten und Strategien umgesetzt werden.
Hardwarenahe Programmierung möglich.	Typsicherheit und Objektorientierung sind nicht vollständig.
leicht erweiterbar durch Bibliotheken, von denen bereits eine Menge vorhanden und sogar zum Standard gehören.	Bestimmte Aufgaben können sehr kompliziert oder aufwendig formuliert sein.
C++ ist typsicher	Bibliotheken sind nicht immer besonders einfach aufgebaut.
hohe Kontrolle über die Performance (Speicherverbrauch und Ausführungsgeschwindigkeit) der Anwendung möglich	Graphische Benutzeroberflächen sind meist aufwendig zu programmieren, auch wenn es dafür Hilfen und Werkzeuge gibt.
Portierbare Programmierung möglich.	Arbeiten mit Zeigern ist fehleranfällig und wird von Anfängern nur schwer verstanden.
Es wurde bereits viel Erfahrung mit der Sprache gesammelt und diese ist in die Weiterentwicklung der Sprache und in verfügbare Hilfen eingeflossen.	Zeichenketten gibt es in vielen Varianten, die immer wieder zu Verwirrungen, komplizierten Implementierungen und zu Fehlern führen.
Objektorientierung hilft große Projekte gut zu strukturieren.	Trägt einige Altlasten mit sich, um eine gewisse Kompatibilität zu C erhalten.

Zahlensysteme

Binäre Zahlen

Computer arbeiten binär. Das heißt sie kennen nur die Zustände 0 und 1. Physikalisch kann z.B. durch zwei unterschiedliche Spannungspotentiale realisiert werden. Ein Potential von ca. 0 V entspricht dem Zustand 0 und ein Potential von ca. 5 V entspricht dem Zustand 1. Die Wahl der Potentiale ist beliebig und muss nur unterscheidbar sein.

Ein Kondensator ist ein Stromspeicher, der wenn er geladen ist ein Potential ungleich 0 V hat und wenn er geladen ist ein Potential ungleich 0 V hat. Damit ist es möglich binäre Zustände zu speichern. Der Zustand eines Kondensators bildet damit den Speicher für ein Bit.

Mehrere Kondensatoren (Bitspeicher) können kombiniert werden, so dass mehrere Zustände parallel gespeichert werden können. Wenn man z.B. 8 Bitspeicher logisch zusammenfasst, dann nennt man das ein Byte. Physikalisch ist also ein Byte nichts weiter als der Ladungszustand von 8 Kondensatoren.

Die Bitspeicher in einem Byte werden gerne von 0 bis 7 durchnummeriert. Mit den 8 Bitspeichern hat man die Möglichkeit 256 (2^8) Zustände zu repräsentieren.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0		Dezimal	Hexadezimal
0	0	0	0	0	0	0	0		0	0x00
0	0	0	0	0	0	0	1		1	0x01
0	0	0	0	0	0	1	0		2	0x02
0	0	0	0	0	0	1	1		3	0x03
0	0	0	0	0	1	0	0		4	0x04
0	0	0	0	0	1	0	1		5	0x05
0	0	0	0	0	1	1	0		6	0x06
0	0	0	0	0	1	1	1		7	0x07
0	0	0	0	1	0	0	0		8	0x08
0	0	0	0	1	0	0	1		9	0x09
0	0	0	0	1	0	1	0		10	0x0A
0	0	0	0	1	0	1	1		11	0x0B
0	0	0	0	1	1	0	0		12	0x0C
0	0	0	0	1	1	0	1		13	0x0D
0	0	0	0	1	1	1	0		14	0x0E
0	0	0	0	1	1	1	1		15	0x0F
...									...	
1	1	1	1	1	1	1	0		254	0xFE
1	1	1	1	1	1	1	1		255	0xFF

Um den Dezimalwert eines Bytes zu berechnen, kann die Summe der Binärstellen multipliziert mit der Zweierpotenz und der Stellennummer als Exponent gerechnet werden.

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0

$$Z_{binär} = b_7 \cdot 2^7 + b_6 \cdot 2^6 + b_5 \cdot 2^5 + b_4 \cdot 2^4 + b_3 \cdot 2^3 + b_2 \cdot 2^2 + b_1 \cdot 2^1 + b_0 \cdot 2^0$$

Beispiel:

$$1101 = 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 1 \cdot 8 + 1 \cdot 4 + 0 + 1 \cdot 1 = 13$$

Binär kann man jede beliebige Natürliche Zahl darstellen. Je nachdem wie groß die Zahl ist, benötigt die binäre Darstellung unterschiedlich viele Bit-Speicher. Mit einem Byte können die Zahlen von 0-255 dargestellt werden. Nimmt man 2 Byte (das wird auch als ein Wort oder engl. Word bezeichnet) können 2^{16} also 65536 verschiedene Zahlen dargestellt werden. Nimmt man 4 Byte (32 Bit) bekommt man 4.294.967.296 verschiedene Kombinationsmöglichkeiten, kann also rund 4 Milliarden Zahlen darstellen.

Hexadezimale Zahlen

Binäre Zahlen werden sehr schnell unübersichtlich. Daher verwendet man üblicherweise das hexadezimale Zahlensystem. So wie beim Binären jede Stelle der Zahl 2 Zustände und beim dezimalen System jede Stelle 10 Zustände hat, hat das hexadezimale System 16 Zustände für jede Stelle der Zahl.

Da es nur 10 Ziffern gibt, werden die verbleibenden 6 Zustände mit den Buchstaben von A bis F gekennzeichnet. Die Berechnung erfolgt analog zum Binären System, nur dass anstatt mit Zweierpotenzen mit 16er-Potenzen gerechnet wird.

16^7	16^6	16^5	16^4	16^3	16^2	16^1	16^0
b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0

$$Z_{hex} = b_7 \cdot 16^7 + b_6 \cdot 16^6 + b_5 \cdot 16^5 + b_4 \cdot 16^4 + b_3 \cdot 16^3 + b_2 \cdot 16^2 + b_1 \cdot 16^1 + b_0 \cdot 16^0$$

Als Kennzeichen das die geschriebene Zahl im hexadezimalen Format ist, wird entweder ein „0x“ vor die Zahl geschrieben oder ein „h“ angehängt. Bei binären Zahlen wird manchmal ein „b“ angehängt.

Beispiel:

0x0A = 0Ah = 10

Das hexadezimale Zahlensystem hat die vorteilhafte Eigenschaft, dass eine Stelle im hexadezimalen System, 4 Stellen im binären System zusammenfasst.

Beispiel:

binär	1	0	0	1	0	1	1	0
hexadezimal	9				6			

Zum dezimalen Zahlensystem gibt es einen solchen Zusammenhang nicht.

Negative binäre Zahlen

Um anzuzeigen, dass eine Zahl negativ ist, wird per Konvention das höchstwertigste Bit der Zahl gesetzt. Aber auch die anderen Bits bekommen eine andere Bedeutung. Um eine negative binäre Zahl in eine Dezimalzahl umzurechnen, muss dazu das Zweierkomplement gebildet werden. Dazu

werden alle Stellen der binären Zahl invertiert und dann der Wert eins hinzuaddiert. Diese Zahl ergibt den Betrag der Dezimalzahl ohne Vorzeichen.

Beispiel:

binär	1	1	1	1	1	1	1	1
invertiert	0	0	0	0	0	0	0	0
+1	0	0	0	0	0	0	0	1
dezimal	|-1|							

Die Darstellung der negativen Zahlen im Zweikomplement hat den Vorteil, dass nach der 0 bei der kein Bit gesetzt ist die Zahl kommt bei der alle Bits gesetzt sind. Es entsteht eine Art rollierendes Zahlensystem.

Festkommazahlen

Jetzt werden bei Berechnungen nicht nur Ganze Zahlen benötigt, sondern auch Reelle Zahlen, Kommazahlen. Diese können in der Regel nicht vollständig, sondern nur näherungsweise abgelegt werden. Eine Möglichkeit sind Festkommazahlen. Hier werden eine feste Anzahl an Bits für die Ziffern vor dem Komma verwendet und eine feste Anzahl an Bits für die Stellen nach dem Komma. Diese Variante kommt aber bei PCs kaum zum Einsatz.

Fließkommazahlen

Üblicherweise kommen in Computersystemen Fließkommazahlen zum Einsatz. Eine Fließkommazahl besteht aus der Mantisse (m), der Basis (b) und einem Exponenten (e).

$$m \cdot b^e$$

Da der Computer binär rechnet ist hier die Basis = 2. die Mantisse ist eine „normalisierte“ Form der Vorkommazahl. Normalisiert bedeutet hier, dass das Komma der Zahl zur höchsten Stelle verschoben wurde.

Da die Basis 2 ist, entspricht der Exponent der Anzahl die das Komma nach links (bei positivem Exponent) oder nach rechts (bei negativem Exponent) verschoben werden muss.

Beispiel:

Dezimal	12,0									
Binär	1010,0									
Normalisiert binär	$0{,}10100b \cdot 10b^{0100b}$									
Speicherbild	VZ	Mantisse						Exponent		
	0	1	0	1	0	0	0	1	0	0

Je nachdem wie viele Stellen für die Mantisse und den Exponenten verwendet wird, können unterschiedliche Genauigkeiten erreicht werden. Bei den heute üblichen Systemen werden Fließkommazahlen einfacher Genauigkeit in 32 Bit kodiert. Dazu werden 23 Bit für die Mantisse und 8 Bit für den Exponenten verwendet. Das entspricht ungefähr 7 Nachkommastellen einer dezimalen Zahl. Bei Fließkommazahlen doppelter Genauigkeit werden 64 Bit verwendet. 52 Bit für die Mantisse und 11 Bit für den Exponenten.

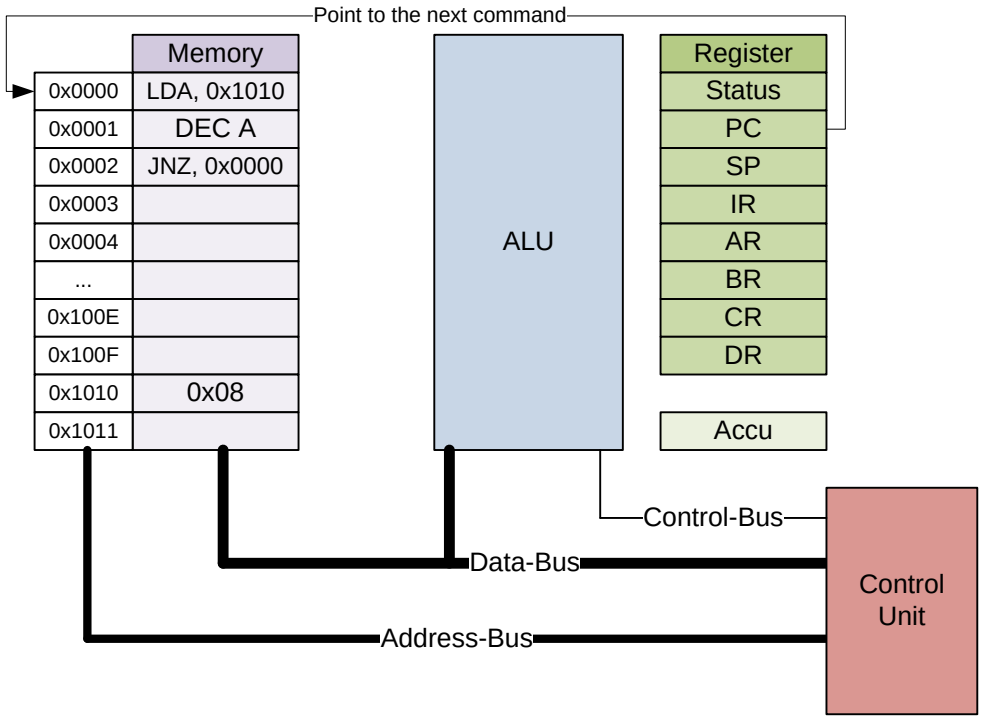
Aufbau eines Rechners

Die Prinzipien eines Rechnersystems aus der Vorlesung „Grundlagen der Informationsverarbeitung I“ werden für diese Vorlesung vorausgesetzt. Das bedeutet sie müssen mit den Prinzipien der von-Neumann-Architektur vertraut sein und die Grundlagen eines Betriebssystems verstanden haben. Sollten sie hier noch Nachholbedarf für sich sehen empfehle ich die Seite von Herrn Prof. Dr. Hoffmann nochmals zu besuchen (http://www.fh-aachen.de/menschen/jhoffmann/GIV/GIV/) oder das Internet zu nutzen. Ein paar Hinweise sind auch nochmals hier im Skript enthalten.

Es gibt verschiedene Architekturen um einen Computer zu realisieren. Für die heutigen PCs ist die „Von-Neumann-Architektur“ die Basis. Diese definiert bestimmte Komponente

Komponenten

Ein Computer besteht aus verschiedenen Komponenten.



- Rechenwerk (ALU - Arithmetische Logik Einheit)
    - Registerspeicher
        - Statusregister
            - Übertragsbit (Carry-Bit – C)
            - Überlaufbit (Overflow-Bit – V)
            - Nullbit (Zero-Bit – Z)
            - Paritätsbit (Parity-Bot – P)
        - Programmzähler (PC)

        - Stackpointer (SP)
        - Counterregister (CR)
        - Indexregister (DR)
        - General Purpose Register (AR/BR)
    - Akkumulator
    - Arithmetik-Einheit (Z.B. Addierer/Subtrahierer/Multplizierer/Dividierer)
        - Ganzzahl/Festkomma/Gleitkomma
    - Logikeinheit (AND/OR/XOR/NOT)
    - Sonstige Funktionen (Z.B.Schiebe- und Rotationsoperationen)
- Steuerwerk
- Bus-System
    - Adressbus
    - Datenbus
    - Steuerbus
- Speicherwerk
    - Datenspeicher
    - Programmspeicher
- Ein-/Ausgabewerk

Speicher

Der Speicher eines Computersystems wird in der Regel byteweise organisiert. Jedes Byte Speicher bekommt eine Adresse. Selbst wenn der Daten-Bus mehr als ein Byte gleichzeitig übertragen kann.

Werden Zahlen abgelegt die mehr als ein Byte Speicherplatz benötigen abgelegt, dann werden diese in aufeinanderfolgende Bytes abgelegt. Dabei werden zwei unterschiedliche Varianten verwendet. Entweder werden die Bytes der Zahl vom niederwertigsten Bit in der niedrigsten Adresse abgelegt (Little-Endian Format) oder es wird mit dem höchstwertigen (Big-Endian Format) begonnen.

Beispiel:

Die folgende 32 Bit Zahl wird im Speicher abgelegt

Hexadezimale Schreibweise 0x12345678

Speicher Adresse:	0x0000	0x0001	0x0002	0x0003
Little-Endian	0x78	0x56	0x34	0x12
Big-Endian	0x12	0x34	0x56	0x78

Es ist zu beachten, dass nicht die Reihenfolge aller Bits sondern nur der Bytes vertauscht wird.

Busse

Die Bus-Systeme sind parallel laufende Leitungen, die zu einem Bestimmten Zeitpunkt einen Binären Zustand von einer Komponente zu einer Anderen übertragen. Je nach Anzahl der Leitungen spricht

man von unterschiedlichen Busbreiten. Über die Busbreite ist auch die gleichzeitig übertragbare Anzahl an Daten definiert bzw. der maximal adressierbare Speicher.

Hat ein System einen Adress-Bus von 32 Bit Breite, dann können maximal 4 Gigabyte Speicher ausgewählt werden. Mehr Speicher kann vom System nicht verwaltet werden und muss z.B. auf eine Festplatte ausgelagert werden.

Hat ein System einen Daten-Bus von 16 Bit Breite, dann können maximal 2 Byte gleichzeitig übertragen werden. Ist die ALU in der Lage 32 Bit Daten zu verarbeiten, dann müssen 2 Datenübertragungen erfolgen, bevor eine Berechnung gemacht werden kann.

Programmablauf

1. Das Steuerwerk liest den nächsten Befehl aus dem Speicher, der mit dem Programmzähler angegeben wird. Eventuell benötigte Daten aus dem Speicher werden in die Register geladen.
2. Der Befehl wird ausgeführt.
3. Der Befehlszähler wird um eins erhöht.

Befehle

Die Befehle liegen im Programmspeicher des Computers als Opcodes vor. Die Opcodes sind Zahlenkombinationen mit bestimmten Bedeutungen. Manche Opcodes können auch noch Informationen für die auszuführende Funktion enthalten. Soll z.B. ein Speicherregister geladen werden, dann wird neben der Zahlenkombination für das Laden aus dem Datenspeicher auch noch die Adresse die zu laden ist angegeben.

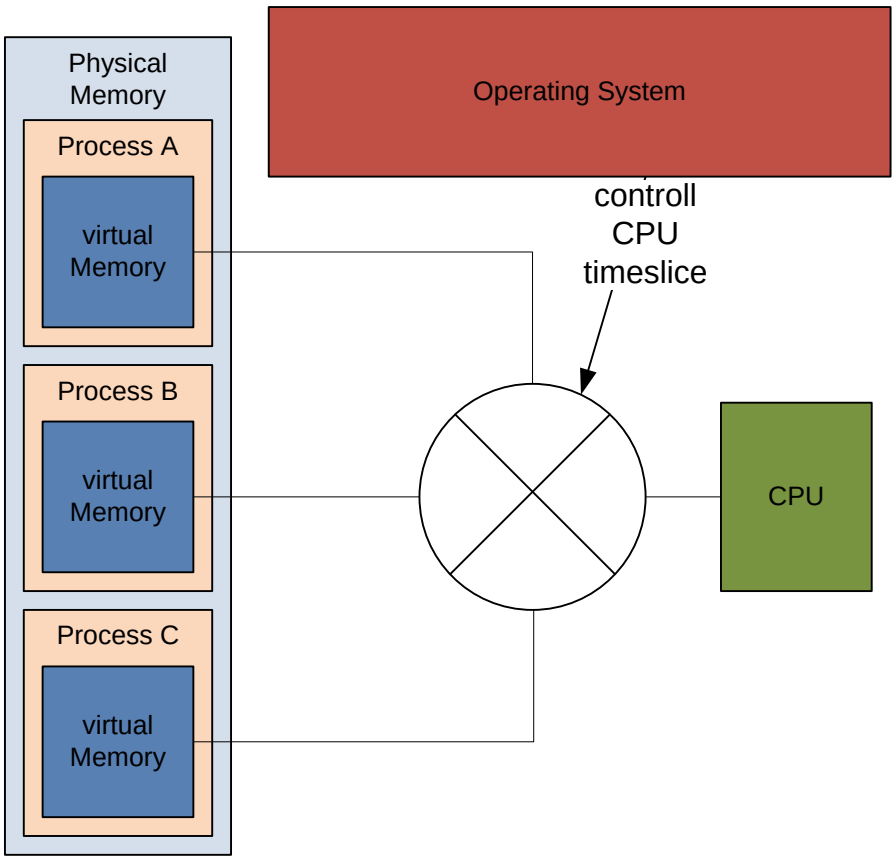
- Ladebefehle um Daten aus dem Speicher in den Akku, in ein Register oder in einen anderen Speicher zu laden. Wenn Konstanten in einen Speicher oder Register geladen werden sollen, dann ist das entweder ein bestimmter Bereich im Datenspeicher oder ist mit im Befehl kodiert.
    - Direkte Adressierung: Es werden die Speicheradressen vollständig angegeben.
    - Indiziert Adressierung: Der Inhalt des Index-Registers und eine Offset addiert ergeben die Adresse.
- Arithmetische und Logische Befehle um Daten im Akkumulator mit Registerwerten oder Daten aus dem Speicher zu berechnen. Das Ergebnis wird meistens in den Akkumulator abgelegt.
- Sprungbefehle schreiben einen neuen Wert in das Programzähler-Register.
    - Absolut, die Adresse wird vollständig angegeben
    - Relativ, zur aktuellen Adresse wird ein Wert addiert oder subtrahiert
    - Bedingt, der Befehl wird nur ausgeführt, wenn ein Status-Bit gesetzt ist oder nicht.
- Stapel-Befehle um Daten auf einen speziellen Datenbereich zu legen, bzw. von dort zu holen.
- Aufrufbefehle überschreiben den Programmzähler, wie ein Sprungbefehl aber sichern zusätzlich bestimmte Register um an die Stelle zurückkehren zu können.

Prozesse und Threads

Die wichtigsten Aspekte sind, dass ein Rechner aus einer Recheneinheit (CPU) und einem oder mehreren Speichern besteht, die über Busse miteinander verbunden sind. Jede Recheneinheit für sequentiell Maschinenbefehle aus, wobei der nächste Befehl erst ausgeführt wird, wenn der vorige beendet wurde. Rechner haben heute mehrere Recheneinheiten und können auf diese Weise

tatsächlich mehrere Befehle gleichzeitig ausführen, aber sogenanntes Multi-Processing wird in dieser Veranstaltung nicht weiter behandelt.

Außerdem sollten sie verstanden haben, dass Betriebssysteme die Ressourcen eines Rechnersystems verwalten und Anwendungen in eigenen Prozessen starten. Das Betriebssystem ist dafür verantwortlich, dass die Prozesse ausgeführt werden. Dazu sorgt es dass ein Prozess für eine bestimmte Zeit von der CPU ausgeführt werden kann. Sobald die Zeit um ist, wird der Prozess angehalten, der Zustand gesichert und die CPU wird einem anderen Prozess zur Verfügung gestellt. Sobald alle aktiven Prozesse im System dran waren, wird wieder von vorne begonnen und der Prozess bekommt wieder Prozessorzeit zugeteilt.

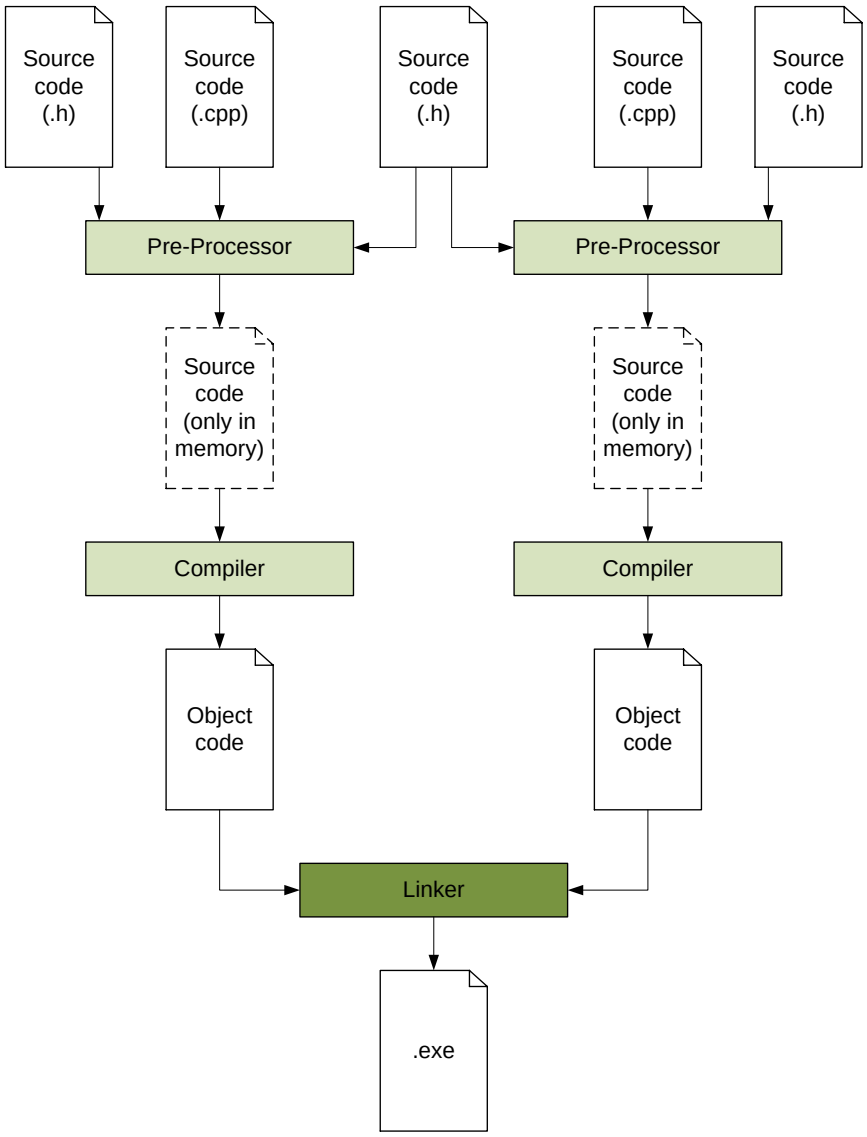


Zu guter Letzt ist noch wichtig, dass ein Programm immer einen Einsprungspunkt hat, das heißt das Betriebssystem will wissen, an welcher Adresse im Speicher der erste auszuführende Befehl für ein Programm ist.

So wie verschieden Prozesse quasi parallel ausgeführt werden, können innerhalb eines Prozesses auch mehrere Befehlssequenzen parallel ausgeführt werden. Diese Einheiten werden als Threads bezeichnet und verfügen im Gegensatz zu den Prozessen nicht über einen eigenen Speicherbereich sondern teilen sich den virtuellen Speicher eines Prozesses. Jeder Prozess hat mindestens einen Thread, der mit dem Prozess gestartet wird. Hier liegt auch der Einsprungspunkt für eine Anwendung.

C++ Codegenerierung

Der Quellcode von C++ ist ein Text, der in Maschinenbefehle übersetzt werden muss, damit diese Befehle ausgeführt werden können. Das C++ Programm wird mit einem Texteditor geschrieben und dann übersetzt. Die Übersetzung erfolgt in mehreren Schritten.



Zuerst wird der Text von einem Präprozessor bearbeitet. Bestimmte Texte werden durch andere Texte ersetzt (Präprozessor-Makros) oder bestimmte gekennzeichnete Codeabschnitte werden ausgelassen. Um solche Makros auszuwerten wird der Quellcode von einem Präprozessor bearbeitet. Das Ergebnis des Präprozessors ist in der Form des ursprünglichen Quellcodes ähnlich und wird an den Kompilierer übergeben.

Der Kompilierer macht aus dem C/C++ Quellcode Anweisungen in Maschinensprache (Assembler). Dabei werden allerdings nicht alle Adresse aufgelöst sondern für die Adressen Platzhalter hinterlegt. Zum Beispiel Sprungadressen, ab denen die Ausführungsfolge der Befehle weiter gehen soll, können unter Umständen zu diesem Zeitpunkt nicht bestimmt werden. Auch die Position von Variablen ist nicht unbedingt bekannt. Der kompilierte Quellcode wird in einer Objekt-Code Datei abgelegt. [Details siehe]

Eine oder mehrere Objekte-Code Dateien werden zu einer ausführbaren Datei zusammengelegt. Sprung- und Speicheradressen werden im Objektcode aufgelöst und mit den korrekten werden eingefügt. Die ausführbare Datei kann vom Betriebssystem geladen und direkt ausgeführt werden.

Lektion 2

Präprozessor-Direktiven

Im ersten Schritt bei der Generierung einer Anwendung mit C++ wird der Quellcode vom Präprozessor verarbeitet. Der Präprozessor sucht nach Befehlen die er kennt, die sogenannten Präprozessordirektiven. Sie dienen dazu den Quellcode an bestimmte Bedingung anpassen zu können. Zum einen wird mit der „#include“ Direktive bereits bestehender Quellcode mit eingebunden, zum anderen können Programmteile aktiviert, bzw. deaktiviert werden. Das kann man z.B. dazu nutzen, um je nachdem für welches Betriebssystem der Code generiert werden soll, andere Funktionen aktiviert zu können. Außerdem können Symbole definiert werden, die dann im Quellcode durch eine Zahl oder ganze Sequenz ersetzt werden. Es ist z.B. üblich konstante Zahlenwerte nicht direkt im Quellcode zu verwenden, sondern einen symbolischen Text, der die Konstante beschreibt (#define M_PI 3.14…).

Die für uns wichtigen möchte ich hier kurz vorstellen.

Direktive	Beschreibung
#define	Definiert ein Symbol und durch was es ersetzt werden soll.
#error	gibt eine Fehlermeldung aus und beendet den Kompiliervorgang.
#if, #elif, #else, #endif	Schließt Code-Blöcke von der Kompilierung ein, bzw. aus, wenn eine Bedingung erfüllt ist.
#ifdef, #ifndef	Schließt Code-Blöcke von der Kompilierung ein, bzw. aus, wenn ein Symbol definiert, bzw. nicht definiert ist.
#include	Fügt den Inhalt einer anderen Quellcode-Datei ein.
#undef	Löscht die Definition für ein Symbol, das zuvor mit #define angelegt wurde.

Namensräume (Namespace)

Für den Kompilierer besteht der Quellcode aus verschiedenen Symbolen, denen eine Bedeutung zugeordnet ist. Z.B. ist der Name einer Variablen nichts weiter, als eine Speicheradresse, in die Daten geschrieben, oder aus der Daten gelesen werden können. Funktionsnamen sind die Speicheradresse ab der eine Kommandosequenz beginnt.

Namen können aber in bestimmten Kontexten andere Bedeutungen haben. Solche Kontexte werden allgemein als Namensraum bezeichnet. Module, Klassen, Funktionen stellen schon selbst einen Namensraum dar. Es gibt aber auch die Möglichkeit einen Namensraum zu erstellen. Innerhalb eines Namensraums müssen Bezeichner eindeutig sein. D.h. zwei Variablen oder zwei Funktionen dürfen nicht den gleichen Namen haben. Auch darf eine Variable nicht wie eine Funktion heißen. Somit ist es mit Namensräumen möglich, Kollisionen von Bezeichnern zu verhindern. Gerade wenn mehrere Menschen an einer Software arbeiten, ist durch die Verwendung von eigenen Namensräumen sichergestellt, dass Bezeichner nicht doppelt vergeben werden.

Die Standardbibliotheken verwenden als Hauptnamensraum „std“. Wenn also Funktionen aus diesem Bereich verwendet werden, dann muss normalerweise immer der Bezeichner „std“ vor den Namen geschrieben werden. Da dies sehr lästig sein kann, gibt die „using“ Anweisung die Möglichkeit dem Kompilierer mitzuteilen, dass er einen Namensraum verwenden soll um ein Symbol aufzulösen.

```
using namespace std;
```

Anwendungsstart und Mainfunktion

Wenn wir ein Programm in C++ erstellen, dann wird durch den Bauprozess (Präprozessor, Kompilierer, Linker) eine Datei, das ausführbare Programm erzeugt. Möchte man dieses Programm starten, dann lädt das Betriebssystem die ausführbare Datei in den Speicher und sucht darin eine Startfunktion. In C/C++ heißt diese Funktion immer „main“. Eigentlich übergibt diese Funktion ein paar Parameter und liefert einen Ergebniswert zurück, aber dies ist beim Erstellen eines Programms nicht zwingend erforderlich. Die einfachste Version einer „main“-Funktion hat keine Parameter was durch leere runde Klammern nach dem Namen ausgedrückt wird und gibt auch keinen Wert zurück. Um das zu kennzeichnen wird vor dem Namen der Datentyp „void“ angegeben. In geschweifte Klammern ist das auszuführende Programm.

```
void main()
{
}
```

Elemente einer Anwendung

Die Quellcodedatei einer C++ Anwendung hat üblicherweise die Dateiendung „.cpp“. Dabei handelt es sich um eine einfache Textdatei, die man auch mit einfachen Texteditoren erstellen könnte.

Kommentare

Da viele Anweisungen nicht selbsterklärend sind, gibt es die Möglichkeit Kommentare im Quellcode unterzubringen. Diese Texte werden dann vom Präprozessor und vom Kompilierer ignoriert.

Es gibt mehrere Möglichkeiten Kommentare einzufügen.

Ab dem doppelten Schrägstrich „//“ wird der Text bis zum Zeilenende ignoriert und gilt als Kommentar.

Ab dem Schrägstrich mit einem Stern „/*“ werden Zeichen solange ignoriert, bis die umgekehrte Zeichenfolge „*/“ kommt. Zwischen diesen beiden Stellen ist alles ein Kommentar.

Am Anfang einer Quellcodedatei wird üblicherweise ein sogenannter Kommentarkopf erstellt, der etwas über den Inhalt und den Autor der Datei aussagt.

```
//------------------------------------------------------------------------
// main.cpp
//
// Contains the main function of the example application.
//
// Date: 2012-03-27
// Author: Thomas Braun
// Email: thomas.braun2@fh-aachen.de
//------------------------------------------------------------------------
```

Include-Anweisungen

Nach dem Kommentarkopf folgen üblicherweise eine Reihe von Include-Anweisungen (Präprozessor-Direktiven) um bestimmte Standardfunktionen, Funktionen aus anderen Modulen oder bestimmte Deklarationen einzubinden.

Für Konsolenanwendungen braucht man z.B. das Standard-Stream-Objekt „cout“ um Daten auf der Konsole auszugeben. Dieses Objekt ist in einer Standardbibliothek bereits enthalten und braucht nur noch mit Hilfe der Datei „iostream“ eingebunden werden.

```
#include<iostream>
```

Datenausgabe auf der Konsole

Da C++ eine objektorientierte Sprache ist, werden für viele Dinge sogenannte Objekte verwendet. Das erste Objekt, dass wir kennenlernen ist „std::cout“ („std“ ist der Namensraum und „cout“ der Objektname). Es handelt sich dabei um ein „Stream-Objekt“ da hier Daten wie ein Datenstrom betrachtet werden, die in das Objekt hinfließen. Die Daten die hineinfließen werden mit einem doppelten Kleiner-Als-Zeichen angehängt. Dabei können mehrere Daten hintereinander angehängt werden. Alle Daten die in „std::cout“ fließen werden auf der Konsole (dem Textausgabefenster) ausgeben. Soll ein Zeilenumbruch dargestellt werden, dann gibt es den Wert „std::endl“ der in „std::cout“ einfließen darf.

```
//---------------------------------------------------------------------------
// main.cpp
//
// Contains the main function of the example application.
//
// Date: 2012-03-27
// Author: Thomas Braun
// Email: thomas.braun2@fh-aachen.de
//---------------------------------------------------------------------------

#include<iostream>

void main()
{
	std::cout << "Hello World!" << std::endl;
}
```

Damit ein Programm sich nicht sofort beendet, wenn es seine Aufgabe erfüllt hat und das Ergebnis dem Benutzer noch dargestellt werden soll, kann eine Standardfunktion „system“ verwendet werden. Diese braucht als Parameter einen Systembefehl. Wir verwenden den Systembefehl „PAUSE“.

```
system("PAUSE");
```

Alle Anweisungen müssen in C/C++ mit einem Semikolon abgeschlossen werden.

Variablen

Möchte man Daten speichern, dann muss man zuerst entscheiden, von welchem Typen diese Daten sind. Der Datentyp bestimmt wie viel Speicher zum Speichern des Datenwertes reserviert wird. Außerdem muss darauf geachtet werden dass der zu Speichernde Wert innerhalb des Wertebereichs für den angegebenen Datentyp ist. So kann der Datentyp „char“ nur Werte zwischen 0 und 255 speichern. Größere Werte würden verfälscht abgelegt.

Datentypen können einfache Datentypen sein, die einen einzelnen Wert speichern können oder es können komplexe Datentypen sein, die aus mehreren einfachen Datentypen aufgebaut sind.

Category	Type	Contents
Integral	**char**	Type **char** is an integral type that usually contains members of the execution character set — in Microsoft C++, this is ASCII.
		The C++ compiler treats variables of type **char**, **signed char**, and **unsigned char** as having different types. Variables of type **char** are promoted to **int** as if they are type **signed char** by default, unless the /J compilation option is used. In this case they are treated as type **unsigned char** and are promoted to **int** without sign extension.
	bool	Type **bool** is an integral type that can have one of the two values **true** or **false**. Its size is unspecified.
	short	Type **short int** (or simply **short**) is an integral type that is larger than or equal to the size of type **char**, and shorter than or equal to the size of type **int**.
		Objects of type **short** can be declared as **signed short** or **unsigned short**. **Signed short** is a synonym for **short**.
	int	Type **int** is an integral type that is larger than or equal to the size of type **short int**, and shorter than or equal to the size of type **long**.
		Objects of type **int** can be declared as **signed int** or **unsigned int**. **Signed int** is a synonym for **int**.
	__int*n*	Sized integer, where *n* is the size, in bits, of the integer variable. The value of *n* can be 8, 16, 32, or 64. (**__int***n* is a Microsoft-specific keyword.)
	long	Type **long** (or **long int**) is an integral type that is larger than or equal to the size of type **int**.
		Objects of type **long** can be declared as **signed long** or **unsigned long**. **Signed long** is a synonym for **long**.
	long long	Larger than an unsigned **long**.
		Objects of type **long long** can be declared as **signed long long** or **unsigned long long**. **Signed long long** is a synonym for **long long**.
Floating	**float**	Type **float** is the smallest floating type.
	double	Type **double** is a floating type that is larger than or equal to type **float**, but shorter than or equal to the size of type **long double**.1
	long double1	Type **long double** is a floating type that is equal to type **double**.
Wide-character	**__wchar_t**	A variable of **__wchar_t** designates a wide-character or multibyte character type. By default, **wchar_t** is a native type but you can use /Zc:wchar_t- to make **wchar_t** a typedef for **unsigned short**. Use the L prefix before a character or string constant to designate the wide-character-type constant.

Quelle http://msdn.microsoft.com/en-us/library/cc953fe1.aspx

Type Name	Bytes	Other Names	Range of Values
int	4	signed	–2,147,483,648 to 2,147,483,647
unsigned int	4	unsigned	0 to 4,294,967,295

Type Name	Bytes	Other Names	Range of Values
__int8	1	char	–128 to 127
unsigned __int8	1	unsigned char	0 to 255
__int16	2	short, short int, signed short int	–32,768 to 32,767
unsigned __int16	2	unsigned short, unsigned short int	0 to 65,535
__int32	4	signed, signed int, int	–2,147,483,648 to 2,147,483,647
unsigned __int32	4	unsigned, unsigned int	0 to 4,294,967,295
__int64	8	long long, signed long long	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
unsigned __int64	8	unsigned long long	0 to 18,446,744,073,709,551,615
bool	1	none	false or true
char	1	none	–128 to 127 by default 0 to 255 when compiled with /J
signed char	1	none	–128 to 127
unsigned char	1	none	0 to 255
short	2	short int, signed short int	–32,768 to 32,767
unsigned short	2	unsigned short int	0 to 65,535
long	4	long int, signed long int	–2,147,483,648 to 2,147,483,647
unsigned long	4	unsigned long int	0 to 4,294,967,295
long long	8	none (but equivalent to __int64)	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
unsigned long long	8	none (but equivalent to unsigned __int64)	0 to 18,446,744,073,709,551,615
enum	varies	none	See Remarks.
float	4	none	3.4E +/- 38 (7 digits)
double	8	none	1.7E +/- 308 (15 digits)
long double	same as double	none	same as double
wchar_t	2	__wchar_t	0 to 65,535

Quelle: http://msdn.microsoft.com/en-us/library/s3f49ktz.aspx

Wenn mit einem Datentyp nicht nur Werte, sondern auch Funktionen verbunden sind, dann spricht man von einer Klasse. Der Speicherplatz, der für eine Klasse reserviert wird, ist die Objektinstanz, also das konkrete Objekt.

Um Speicher für einen Datentyp in einem Programm verfügbar zu machen, muss der Speicher deklariert werden. Dazu wird der Datentyp gefolgt von einem Namen für den reservierten Speicher (Variablenname) geschrieben. Dieser benannte Speicherbereich wird in der Informatik als Variable bezeichnet.

Der Variablen kann auch gleich ein Wert zugewiesen (initialisiert) werden. Dazu muss nach dem Namen noch ein Gleichheitszeichen „=“ gefolgt vom Wert ergänzt werden. Die folgende Zeile reserviert 4 Byte für eine Integer-Variable und belegt diese mit dem Wert „4711“ vor.

```
int myVar = 4711;
```

Es gilt als guter Stil immer eine Variable zu initialisieren, damit sie einen definierten Wert hat. Durch die Initialisierung können bestimmte Fehler bei der Programmierung vermieden werden, da man schneller erkennen kann, das man auf eine Variable zugreift, ohne ihr einen Wert zugewiesen zu haben. Was wiederum zu sehr heimtückischen Fehlern führen kann.

Sobald eine Variable definiert wurde, kann sie verwendet werden. Sie kann lesend verwendet werden, zum Beispiel in dem der Wert der Variable auf der Konsole ausgegeben wird oder der Wert einer anderen Variablen zugewiesen wird.

```
// Declare variable
int myVar = 4711;

// Output to console
cout << myVar << endl;

// Declare other variable
int otherVar = 0;

// Assign to another variable
otherVar = myVar;
```

Die Variable kann auch schreibend verwendet werden, wie im oberen Beispiel die Variable „otherVar“ mit dem Wert von „myVar“ überschrieben wurde. Das Setzen des Wertes einer Variable erfolgt mit dem Zuweisungsoperator, dem einfachen Gleichheitszeichen („=“). Dabei wird der Wert des Ausdrucks auf der rechten Seite, dem Ausdruck auf der linken als Wert zugewiesen. Es ist dabei zu beachten, dass auf der linken Seite nur Ausdrücke stehen dürfen, denen auch Werte zugewiesen werden können. Damit sind wir aber schon beim nächsten Thema.

Ausdrücke (engl. Expressions)

In der Informatik versteht man unter einem Ausdruck eine Anweisung hinter der sich ein Wert verbirgt. Dabei kann ein Ausdruck ein einfacher konstanter Wert, wie etwa „10“ sein oder aus einer komplexen Folge von Operationen und Funktionsaufrufen sein.

```
// Examples for expressions
8;      // constant expression -> result is value is 8

8 + 1; // simple operation with constant expression -> result is value is 9

6 + 7 * 10 / 5; // multiple operations with constant expression -> result is value is 20

7 + sin(3.14); // constant expression with function call -> result is value is ~7.0
```

```
myVar; // variable expression  -> result is value is 4711

myVar + 10; // variable expression with constant expression -> result is value is 4721

myVar +10 * cos(3.14); // variable, constant and function call -> result is value is ~4721.0

&myVar + 10; // Address of variable and constant expression -> result is value is 0xff45103
```

Die Beispiele zeigen reine Ausdrücke, die zwar ein gültiger C++ Code darstellen, aber direkt nach der Berechnung verloren gehen. Sie müssen also in irgendeiner Form gespeichert werden. Genau dazu können die Variablen und der Zuweisungsoperator („=“) verwendet werden. Sowohl auf der linken als auch auf der rechten Seite des Zuweisungsoperators stehen Ausdrücke. Allerdings muss der Kompilierer auf der linken Seite einen ***nicht***-konstanten Ausdruck haben.

```
// Examples of assignment
myVar = 10;
myVar = 20 * sin(0.0);

// not allowed because of constant expression on the left
// 10 = 45 - 35;
```

Konstante Ausdrücke können durch mehrere Arten erstellt werden. Erstens es werden sogenannte Literale verwendet. Literale sind die Zahlen wie (10, 2.1, usw.) und es gibt noch Literale für Zeichen und Zeichenketten (engl. String). Literale für Zeichen werden durch einfache Hochkommata eingeschlossen und sind vom Typ „char“. Zeichenkettenliterale werden durch doppelte Anführungszeichen eingeschlossen und sind ein Feld „char“-Zeichen. Außerdem kennt C++ noch die Literale für den booleschen Datentyp („bool“), die durch die Zeichenfolge „true“ und „false“ angegeben werden.

```
// Examples of literals
8; // numeric literal
'f'; // character literal
"Hello World!"; // string literal
false; // boolean literal
```

Alle Ausdrücke sind von einem bestimmten Datentyp. So ist der Ausdruck „8“ vom Datentyp „int“. Sind Ausdrücke durch Operationen wie „+“, „*“, usw. verknüpft, dann hängt das Ergebnis von der Operation und vom Datentyp der Operanden ab.

Alle arithmetischen Operationen, bei denen die Operanden alle vom gleichen Datentyp sind, liefern ein Ergebnis das ebenfalls vom gleichen Typ ist. Sind hingegen Ganzzahlige und Fließkomma Datentypen gemischt, ist das Ergebnis ein Fließkomma Datentypen. Boolesche Datentypen werden automatisch in die Werte „0“ für „false“ und „1“ für „true“ gewandelt. Sollten bei arithmetischen Operationen Zeichen angegeben werden, dann werden deren ASCII-Codes verwendet.

Logische und vergleichende Operationen (z.B. „&&“, „||“, „==“, „!=“) liefern als Ergebnis einen Booleschen Datentyp, d.h. das Ergebnis ist immer „true“ oder „false“. Alle numerischen Ausdrücke werden dabei in einen Booleschen Wert gewandelt. Alle Werte die gleich „0“ sind, werden als „false“ gewertet. Alle Werte die ungleich „0“ sind werden als „true“ definiert.

Da es verschiedene numerische Datentypen gibt und einem numerischen Literal der Datentyp nicht anzusehen ist, gibt es hier Regeln und Hilfen. Ganze Zahlen sind grundsätzlich vom Datentyp „int“. Fließkommazahlen sind vom Datentyp „double“. Da eine Ganzzahl auch einen anderen Datentyp darstellen kann gibt es Ergänzungen für die Literale, um zu kennzeichnen welcher Datentyp für das

Literal gemeint ist. Außerdem können Literale auch als hexadezimale Zahlen angegeben werden, in dem das Präfix „0x“ vorangestellt wird.

```
8; // literal of type int
8L; // literal of type long
8ul; // literal of type unsigned long

5.1; // literal of type double
5.1f; // literal of type float

0x10; // literal of type int in hexdecimal form
```

Eine andere Art um konstante Ausdrücke zu erzeugen, sind Variable als Konstant zu deklarieren. Dabei wird vor die Typangabe das Schlüsselwort „const“ gesetzt. Eine konstante Variable muss immer auch gleich initialisiert werden, da sich ihr Wert ja nicht mehr ändern darf.

```
// Example of a constant variable
const int myConstIntVar = 10;

// myConstIntVar = 12; Invalid because not changeable anymore.
```

Anweisungen (engl. Statements)

Anweisungen sind Programmelemente, die steuern ob und in welcher Reihenfolge Objekte manipuliert werden.

Mit Anweisungen kann entschieden werden ob eine einzelne oder mehrere Anweisungen ausgeführt oder wiederholt werden sollen. Wichtige Beispiele sind hier:

- if
- while
- do-while
- for
- switch
- Auswertung eines Ausdrucks
- Blockanweisung - Mehrere Anweisungen zusammenführen ({ … })
- Sprunganweisungen
- Deklarationsanweisungen

Viele Anweisungen verwenden Ausdrücke, um zu entscheiden, ob bestimmte Aktionen erfolgen sollen. So verwendet die „if“ Anweisung einen booleschen Ausdruck, um zu bestimmen ob die folgende Anweisung ausgeführt wird.

```
// Example for statements
int a = 0;
if (a == 10) // if needs a boolean expression
{
	cout << "a is ten" << endl;
	cout << "now you can go home" << endl;
}


if (a == 10) // if you forgot the bracess
	cout << "a is ten" << endl; // only this line depends on if clause
	cout << "now you can go home" << endl; // this line is executed in any
case
```

Die Blockanweisung führt dazu, dass die enthalten Anweisungen wie eine Anweisung aufgefasst werden können. Werden also nach einer „if“-Anweisung die geschweiften Klammern vergessen, dann wird nur die erste Anweisung ausgeführt, da die „if“-Anweisung sich nur auf die nächste folgende Anweisung bezieht. Da das schnell zu Fehlern führt, werden grundsätzlich für „if“, „for“ und „while“ Anweisungen die geschweiften Klammern verwendet, auch wenn nur eine Anweisung enthalten ist.

Die Sprunganweisungen veranlassen, dass nicht die im Quelltext folgende Anweisung ausgeführt wird, sondern dass an eine andere Stelle gesprungen wird, bzw. eine Funktion verlassen wird. Beispiele hierfür sind „break“, „continue“ und „return“. Die Sprunganweisung „goto“ ist verpönt und wird schon sehr lange nicht mehr verwendet.

Deklarationsanweisungen führen neue Bezeichner und deren Bedeutung in den aktuellen Namensraum ein. Also die Deklaration von Typenamen (Klassennamen, Strukturnamen, usw.) oder Variablen- und Objektnamen und Funktionsnamen.

Bezeichner und Schlüsselworte

Bezeichner sind eine Folge von Zeichen im C++ Quellcode der eines der folgenden Elemente bezeichnet.

- Objekte oder Variablen
- Namen von einfachen Datentypen
- Klassen, Strukturen oder Unions
- Name einer Enumeration
- Member einer Klasse, Struktur, Union oder Enumeration
- Funktionen oder Methoden
- Parameter einer Funktion oder Methode
- Label Namen
- Makro Namen
- Makro Parameter

Alle Bezeichner die sie im Quellcode verwenden müssen vorher definiert werden. Dazu können sie die Bezeichner selbst definieren, z.B. bei der Variablendeklaration definieren sie den Bezeichner „myVar“ und erklären das es sich um eine einfache Variable vom Datentyp „int“ handelt. Im nachfolgenden kann der Bezeichner „myVar“ verwendet werden und der Kompilierer weiß, dass es sich um die eine feste Speicherstelle handelt, bzw. der Wert der aktuell in dieser Speicherstelle abgelegt wurde.

```
// Declare variable
int myVar = 4711;
```

Weitere Möglichkeiten um Bezeichner zu definieren, ist eine vorgefertigte Definition aus einer Bibliothek, bzw. aus der dazugehörigen Headerdatei zu verwenden. Das haben wir bereits für das Objekt „std::cout“ getan. Hier verwenden wir die Definition die mit der Präprozessor-Direktive „#include<iostream>“ in unser Programm eingebunden wird.

Außerdem kennt der Kompilierer von sich aus auch schon Bezeichner. Diese Bezeichner werden als Schlüsselwörter bezeichnet. Schlüsselwörter sind also nichts weiter als vom Kompilierer vordefinierte Bezeichner. Beispiele hierfür sind die Basisdatentypen „int“ oder „double“. Aber auch Anweisungen wie „if“ oder „while“.

Visual Studio 2010 stellt in den Standardeinstellungen die Schlüsselworte in blau da. Da es nicht erlaubt ist die Schlüsselworte für eigendefinierte Bezeichner zu verwenden, kann man damit schnell erkennen, dass man ausversehen ein Schlüsselwort für einen Bezeichner verwendet hat.

Die Bezeichner dürfen nur aus bestimmten Zeichen bestehen. Folgende Zeichen sind zugelassen:

- Die Buchstaben „a“ bis „z“
- Die Buchstaben „A“ bis „Z“
- Die Ziffern „0“ bis „9“
- Der Unterstrich „_“

Dabei ist zu beachten, dass ein Bezeichner nicht mit einer Ziffer beginnen darf.

Lektion 3

Boolesche Datentyp

C++ kennt neben den numerischen Datentypen auch noch einen booleschen Daten, der genau 2 unterschiedliche Zustände (Werte) einnehmen kann, nämlich „wahr“ (engl. true) oder „falsch“ (engl. false). Der Datentyp heißt „bool“ und hat für die Werte die Schlüsselwörter „true“ und „false“ reserviert.

In C hatte man in der Vergangenheit hatte man diesen Datentyp dadurch realisiert, dass man den Datentyp „int“ verwendet hat. Dabei ist dann der Wert „0“ gleichbedeutend mit „false“ und ein Wert ungleich „0“ entspricht „true“. Die Konvertierung ist auch heute noch gültig und intern wird der Datentyp „bool“ als „char“ gespeichert. Das hat zur Folge, dass eine Variable vom Typ „bool“ - auf einem x86 Zielsystem - statt 1 Bit immer noch 8 Bit benötigt. Die Größe des Datentyps ist in der Spezifikation von C++ undefiniert. Das 1 Byte dafür verwendet wird ist eine Microsoft spezifische Implementierung.

Operatoren

Operatoren sind wie Funktionen die einen Ausdruck zurückliefern. Der Operator verwendet die sogenannten Operanden um das Ergebnis des Ausdrucks zu ermitteln. Es gibt verschiedene Arten von Operatoren, die unterschiedlich viele Operanden benötigen.

- Unäre Operatoren benötigen 1 Operanden.
- Binäre Operatoren benötigen 2 Operanden.
- Ternäre Operatoren benötigen 3 Operanden.

Außerdem besitzen die Operatoren eine Assoziativität, das bedeutet eine Reihenfolge in der die Operatoranden ausgewertet werden.

Beispiel:

```
1 + 2 // Binärer Operator der von links nach rechts ausgewertet wird.
```

Weiterhin besitzen die Operatoren einen Rang. Besteht ein Ausdruck aus mehreren Operatoren, dann wird der Operator mit dem niedrigsten Rang zuerst ausgewertet. Auf diese Art und Weise wird die Punkt-vor-Strichrechnung realisiert. Die Addition hat den Rang 5 während die Division den Rang 4 besitzt.

Rank	Operator	Name or Meaning	Associativity
0	::	Scope Resolution	None
1	.	Member Selection (object)	Left to right
1	->	Member Selection (pointer)	Left to right
1	[]	Array subscript	Left to right
1	()	Function call or member initialization	Left to right
1	++	Postfix increment	Left to right
1	--	Postfix decrement	Left to right
1	typeid()	type name	Left to right
1	const_cast	Type cast (conversion)	Left to right
1	dynamic_cast	Type cast (conversion)	Left to right
1	reinterpret_cast	Type cast (conversion)	Left to right
1	static_cast	Type cast (conversion)	Left to right
2	sizeof	Size of object or type	Right to left
2	++	Prefix increment	Right to left
2	--	Prefix decrement	Right to left
2	~	One's complement	Right to left
2	!	Logical not	Right to left
2	-	Unary minus	Right to left
2	+	Unary plus	Right to left
2	&	Address-of	Right to left
2	*	Indirection	Right to left
2	new	Create object	Right to left
2	delete	Destroy object	Right to left
2	()	Cast	Right to left
3	.*	Pointer-to-member (object)	Left to right
3	->*	Pointer-to-member (pointers)	Left to right
4	*	Multiplication	Left to right
4	/	Division	Left to right
4	%	Modulus	Left to right
5	+	Addition	Left to right
5	-	Substraction	Left to right
6	<<	Left shift	Left to right
6	>>	Right shift	Left to right
7	<	Less than	Left to right
7	>	Greater than	Left to right
7	<=	Less than or equal	Left to right
7	>=	Greater than or equal	Left to right
8	==	Equality	Left to right
8	!=	Inequality	Left to right
9	&	Bitwise AND	Left to right
10	^	Bitwise exclusive OR	Left to right
11	|	Bitwise inclusive OR	Left to right
12	&&	Logical AND	Left to right
13	||	Logical OR	Left to right
14	expr1 ? expr2 : expr3	Conditional	Right to left
15	=	Assignment	Right to left
15	*=	Multiplication assingment	Right to left
15	/=	Division assignment	Right to left
15	+=	Addition assignment	Right to left

15	-=	Substration assignment	Right to left
15	%=	Modulus assignment	Right to left
15	<<=	Left-shift assignment	Right to left
15	>>=	Right-shift assignment	Right to left
15	&=	Bitwise AND assignment	Right to left
15	|=	Bitwise inclusive OR assignment	Right to left
15	^=	Bitwise exclusive OR assignment	Right to left
16	throw expr	throw expression	Right to left
17	,	Comma	Left to right

Tabelle 1: Operatoren, Quelle siehe [3]

Man kann die Operatoren noch in verschiedene Gruppen einteilen.

Arithmetische Operatoren

Die arithmetischen Operatoren führen die bekannten arithmetischen Operationen (Addition, Subtraktion, usw.) aus. Bei ganzzahligen Operanden ist das Ergebnis ebenfalls ein ganzzahliger Ausdruck. Bei Fließkommazahlen ist das Ergebnis ein Fließkommaausdruck. Werden ganzzahlige und Fließkommaoperanden gemischt, dann ist das Ergebnis ein Fließkommaausdruck.

Zuweisungsoperatoren

Der Zuweisungsoperator weist den ausgewerteten Ausdruck der rechten Seite dem Ausdruck der linken Seite zu. Dabei muss der Ausdruck auf der linken Seite den Ausdruck auch aufnehmen können. D.h. es darf kein konstanter Ausdruck sein, sondern muss eine Speicherstelle repräsentieren. Außerdem muss der Typ der Ausdrücke linke und rechts gleich sein oder automatisch konvertiert werden können.

Da häufig der Wert einer Variablen geändert werden soll, gibt es auch Kombinationen von Arithmetischen- und Zuweisungsoperatoren. Diese stellen eine Kurschreibweise dar.

Berechnung und Zuweisung	Kurzform
`a = a + b`	`a += b`
`a = a - b`	`a -= b`
`a = a * b`	`a *= b`
`a = a / b`	`a /= b`
`a = a + 1`	`a++`
`a = a - 1`	`a--`

Tabelle 2: Kurzformen für Arithmetische Operationen mit Zuweisung

Logische und Vergleichsoperatoren

Logische und Vergleichsoperatoren vergleichen die Operanden und das Ergebnis ist immer ein boolescher Ausdruck.

Schiebeoperatoren und Bit Manipulationen

Integrale Datentypen können auch immer als ein Feld von Bits verstanden werden. Mit den Bit-manipulierenden Operationen lassen sich die einzelnen Stellen der Bitfelder manipulieren. Man kann einzelne Bits setzen, löschen oder invertieren.

Die Schiebeoperatoren verschieben das Bit-Feld um die angegebene Anzahl von Stellen. Die „rausgeschobenen“ Stellen gehen verloren, während die „hereingeschobenen“ Stellen mit „0“ belegt werden.

Datentyp Konvertierung

C++ bietet viele Konvertierungen automatisch an. So kann ein Wert vom Datentyp „char“ problemlos einer Variablen vom Datentyp „int“ zugewiesen werden. Dies nennt man implizite Konvertierung. Ist keine Implizite Konvertierung möglich, kann man mit dem Cast-Operator eine explizite Konvertierung vornehmen. Die Cast-Operatoren (xxx_cast) können dabei noch einige Bedingungen prüfen, um sicherzustellen, dass die Konvertierung auch erlaubt ist.

Sonstige Operatoren

Es gibt noch einen Menge anderer Operatoren, zum Beispiel um die Adresse oder Größe einer Variablen zu ermitteln.

Lektion 4

Dateneingabe von der Konsole

Mit dem Objekt „std::cin“ aus der Standardbibliothek „iostream“ kann man Daten von der Konsole einlesen. Um die eingegebenen Daten einer Variablen zuzuweisen wird der Stream-Operator verwendet.

Da es sich bei „std::cin“ um ein Objekt handelt, gibt es verschiedene Methoden um bestimmte Aktionen auszulösen, bzw. Informationen über die letzte Dateneingabe zu erhalten. Beim Einlesen wird intern ein Puffer (Speicher) verwendet, der die Eingaben erst einmal zwischen speichert. Die Daten liegen alle als Text vor. D.h. auch die Zahlen werden erst einmal als Textzeichen und nicht als numerische Werte verstanden. Soll die Eingabe einer Variablen eines numerischen Datentyps zugewiesen werden, dann muss der Text in eine Zahl gewandelt werden. Ist das nicht möglich, dann wird der Variablen kein neuer Wert zugewiesen. Stattdessen hält das „std::cin“ Fehler-Flags, die gesetzt werden.

Mit den Methoden „fail()“, bzw. „good()“ können die Zustände der Fehler-Flags geprüft werden. Die Fehler-Flags werden nicht automatisch gelöscht. Zum Löschen muss die Methode „clear()“ aufgerufen werden. Mit der Methode „ignore(anzahl)“ kann der Puffer geleert werden. Dazu übergibt man als Anzahl die Puffergröße. Am besten verwendet man hierfür die Anweisung „numeric_limits<std::streamsize>::max()“, die den größtmöglichen Wert für den Datentyp „std::streamsize“ liefert.

```
// simple form without test
int a = 0;
cin >> a;

// variation with test
int b = 0;
do
{
	// clear all bad flags
	cin.clear();
	// clear internal buffer
	cin.ignore(numeric_limits<streamsize>::max());
	// read from console
	cin >> b;

} while (!cin.good()); // repeat if error occurs
```

Alternative Entscheidung

Mit der „if“-Anweisung besteht die Möglichkeit zu entscheiden, ob eine Anweisung oder eine alternative Anweisung ausgeführt werden soll. Verzichtet man auf die Alternative, dann wird die Anweisung ausgeführt oder eben nicht.

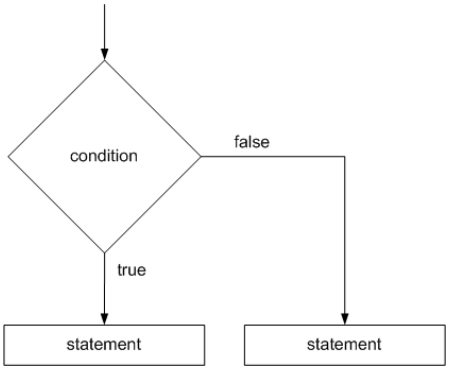
Die Entscheidung ob die nach der „if“-Anweisung stehende Anweisung ausgeführt werden soll wird über einen booleschen Ausdruck getroffen. Der Ausdruck steht in runden Klammern hinter dem Schlüsselwort „if“. Die alternativ auszuführende Anweisung steht hinter dem Schlüsselwort „else“.

Beispiel:

```
// only 1 statement
if (boolean_expression)
	do_something();
else
	do_something_else();

// multiple statements
if (boolean_expression)
{
	do_something1();
	do_something2();
}
else
{
	do_something_else1();
	do_something_else2();
}
```

Wenn die Entscheidung mehr als eine Anweisung betrifft, dann müssen die auszuführenden Anweisungen in geschweifte Klammern (Blockanweisung) gesetzt werden. Da es häufig passiert, dass man zuerst nur eine Anweisung hatte und daher auf die geschweiften Klammern verzichten konnte und jetzt doch mehrere Anweisungen einschließen möchte, aber die Klammern vergessen hat nachträglich zu setzen, wird grundsätzlich immer Klammern gesetzt. Damit entgeht man einer tückischen Falle.



Auswahl

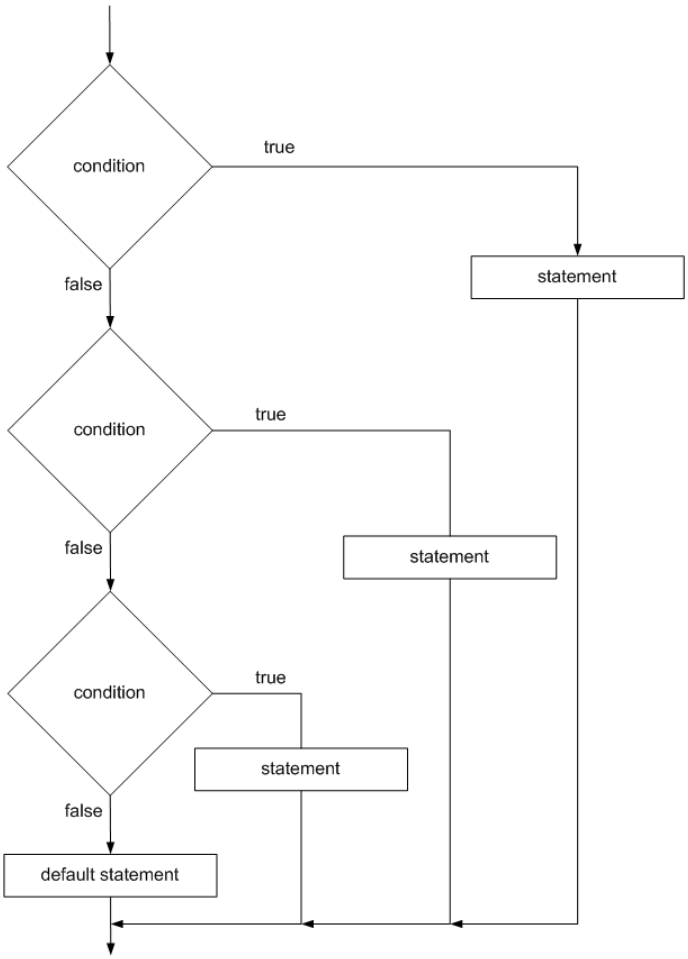
Bei der Auswahl wird ein ganzzahliger numerischer Ausdruck ausgewertet und je nach dem welchen Wert der Ausdruck annimmt, eine bestimmte Code-Stelle gesprungen. Der Wert des Ausdrucks muss konkret benannt werden und es können keine Wertebereiche angegeben werden. Allerdings gibt es den Marker „default“ zu dem gesprungen wird, falls keiner der vorigen Werte zutraf.

Die Anweisung beginnt mit dem Schlüsselwort „switch“ gefolgt mit den integralen Ausdruck (char, short, int, long,…) in runden Klammern. Die Sprungmarken werden in eine gemeinsame geschweifte Klammer gesetzt.

Beispiel:

```
// switch statement
switch (integer_expression)
{
case 1:
       do_something1();
       break;
case 2:
       do_something2();
       break;
default:
       do_something_else();
}
```

Jede Sprungmarke beginnt mit dem Schlüsselwort „case“ dem Wert des integralen Ausdrucks für den die Sprungmarke steht und einem Doppelpunkt („:“). Die nachfolgenden Anweisungen werden so lange abgearbeitet, bis das Ende der Switch-Anweisung erreicht ist oder eine „Break“ Anweisung folgt. Das Schlüsselwort „break“ bewirkt, dass der Block der Switch-Anweisung verlassen wird.



Kopfschleife

Die Kopfschleife prüft einen booleschen Ausdruck und wiederholt die folgende Anweisung solange wie der Ausdruck den Wert „true“ annimmt. Nimmt der Ausdruck den Wert „false“ an, wird die

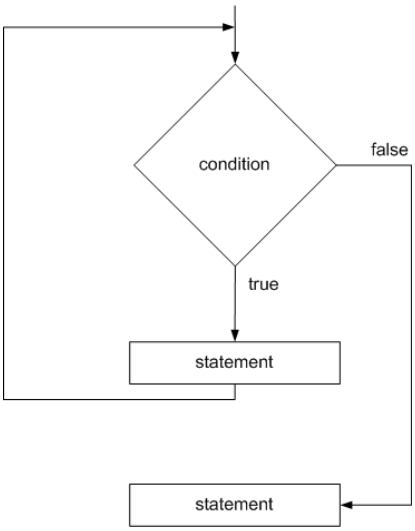
nachfolgende Anweisung nicht weiter wiederholt. Sollen mehrere Anweisungen wiederholt werden, dann müssen diese in geschweifte Klammern (Blockanweisung) gesetzt werden. Auch hier gilt, dass die Klammer möglichst immer verwendet werden sollten, auch wenn nur eine Anweisung wiederholt werden muss.

Beispiel:

```
// head-loop with 1 statement
while (boolean_expression)
	do_something();

// head-loop with multiple statements
while (boolean_expression)
{
	do_something1();
	do_something2();
}
```

Die Anweisung für eine Kopfschleife beginnt mit dem Schlüsselwort „while“ und der Prüfbedingung in runden Klammern.



Fußschleife

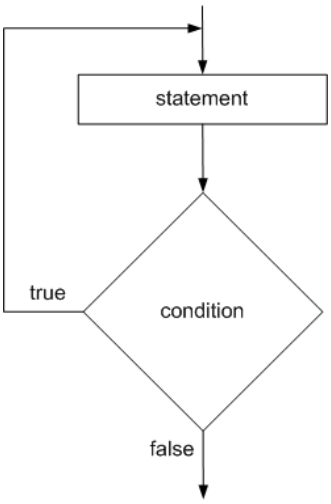
Die Fußschleife ist der Kopfschleife sehr ähnlich. Der wesentliche Unterschied liegt darin, dass erst eine Anweisung ausgeführt wird und danach geprüft wird, ob diese Anweisung wiederholt werden muss. Auch hier ist ein boolescher Ausdruck für die Prüfung notwendig.

Beispiele:

```
// foot-loop with 1 statement
do
	do_something();
while (boolean_expression);

// foot-loop with multiple statements
do
{
	do_something1();
	do_something2();
}
while (boolean_expression);
```

Die Fußschleife wird durch das Schlüsselwort „do“ eingeleitet, gefolgt von der zu wiederholenden Anweisung. Abschließend kommt das Schlüsselwort „while“ mit dem Prüfausdruck in runden Klammern.



Zählschleife

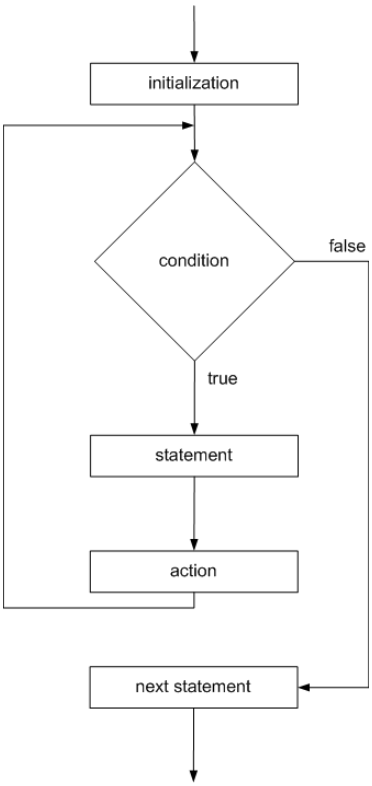
Die Zählschleife ist eine Kurzschreibweise für eine Kopfschleife mit einer Anweisung die einmalig vor der Schleife ausgeführt werden soll (Initialisierung), der booleschen Prüfbedingung und einer Aktion die nach jedem Schleifendurchlauf durchgeführt werden soll.

Beispiel:

```
// counting-loop with 1 statement
for (init_statement; boolean_expression; action_statement)
       do_something();

// counting-loop with multiple statements
for (init_statement; boolean_expression; action_statement)
{
       do_something1();
       do_something2();
}
```

Die Zählschleife beginnt mit dem Schlüsselwort „for“ und den 3 Teilen (Initialisierung, Prüfbedingung, Aktion) in runden Klammern und durch Semikolon getrennt und der zu wiederholenden Anweisung. Sollen mehrere Anweisungen wiederholt werden, dann müssen diese wieder in einer geschweiften Klammer (Blockanweisung) zusammengefasst werden. Auch hier sollte die Blockanweisung immer angewendet werden, um Fehler zu vermeiden.



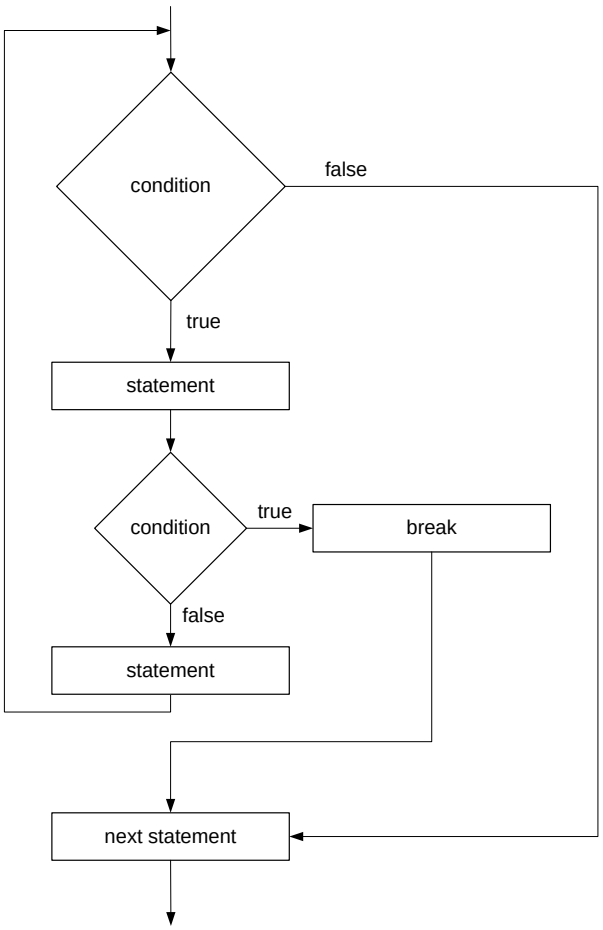
Sprunganweisungen

Es gibt in C++ 4 Sprunganweisungen von denen 3 in der Praxis relevant sind. Der „goto“ Befehl ist verpönt und wird heute nur noch selten verwendet.

Die „break“ Anweisung sorgt dafür, dass ein Block verlassen, bzw. eine Schleife beendet wird und die nächsten Anweisungen nach dem Block bzw. der Schleife abgearbeitet werden.

Beispiel:

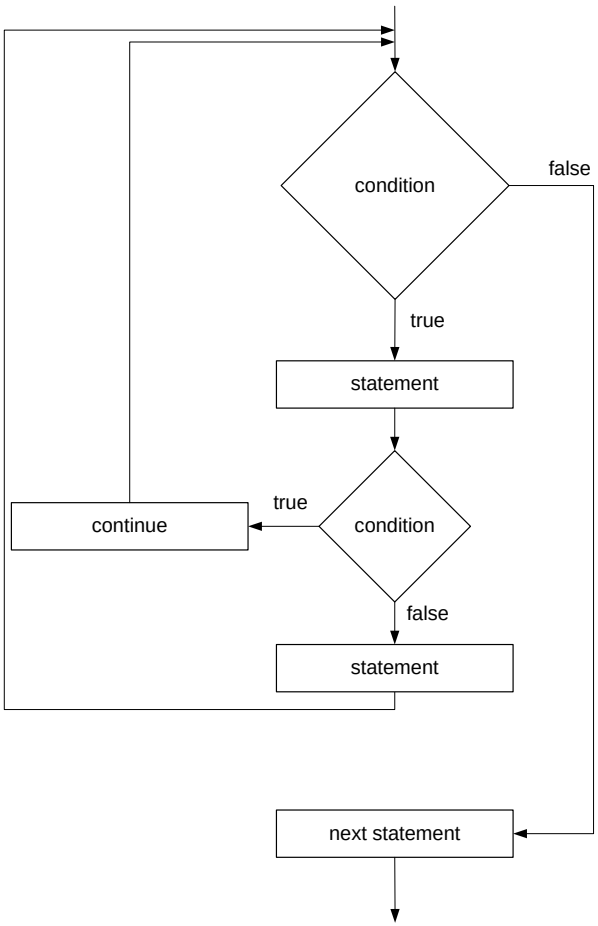
```
// Example of a break in a loop
while (boolean_expression)
{
	do_something();
	if (boolean_expression)
	{
		break; // Here we leave the loop.
			// Next statement is do_something2();
	}
	do_something();
}
do_something2();
```



Die „continue“ Anweisung sorgt dafür, dass ein Schleifendurchlauf abgebrochen wird und erneut die Prüfung durchgeführt wird. Ergibt die Prüfung, dass der Prüfausdruck immer noch „true“ ist wird die zu wiederholenden Anweisung weiter ausgeführt. Bei der Zählschleife wird auch bei einer „continue“ Anweisung immer noch vor der Prüfung die „Aktion“ ausgeführt.

Beispiel:

```
// Example of a continue in a loop
while (boolean_expression)
{
	do_something();
	if (boolean_expression)
	{
		continue;	// Here we jump to the while condition.
				// for this loop do_something1 is not executed.
	}
	do_something1();
}
do_something2();
```



Die „return“ Anweisung beendet eine Funktion oder Methode und springt zum Aufrufer der Funktion oder Methode zurück. Hat die Funktion oder Methode ein Rückgabewert, dann muss nach dem Schlüsselwort „return“ noch ein Ausdruck stehen, der als Rückgabewert mit an den Aufrufer übergeben werden kann. Dabei muss der Ausdruck vom Typ des Rückgabewertes übereinstimmen.

Lektion 5

Funktionen

Funktionen helfen den Quellcode zu strukturieren und bieten die Möglichkeit sich wiederholende Aufgaben nur einmal implementieren zu müssen. Funktionen können einfache Berechnungen ausführen, wie z.B. die Funktion „sin()“ aus der „cmath“ Standardbibliothek oder einen Algorithmus also eine Folge von Anweisungen ausführen.

```
double f = sin(3.14);
```

Der Programmteil, der eine Funktion aufruft, wird als ‚Aufrufer der Funktion bezeichnet. Die Ausführung stoppt im aufrufenden Programmteil und springt zum Anfang der Funktion. Dort werden der Reihe nach die Anweisungen ausgeführt, bis die Funktion zu Ende ist oder eine „return“-Anweisung erfolgte. Ab dort springt die Ausführung zum gestoppten Programmteil und führt die nächsten Anweisungen aus.

```
int MyFunction(int parameter)
{
```

```
        // do something
        return parameter+1;
}
```

Ruft sich eine Funktion selbst auf, dann wird das als Rekursion oder rekursiver Funktionsaufruf bezeichnet. Rekursive Aufrufe brauchen eine Abbruchbedingung, da sonst irgendwann ein Speicherüberlauf erfolgt und es zu einem Programmabsturz kommt. Die Abbruchbedingung sorgt dafür, dass die Anweisung sich selbst aufzurufen nicht mehr ausgeführt wird.

```
void RekursiveFunction(int counter)
{
        counter++;

        if (counter < 10) // This is the break condition
        {
                // call the function itself
                RekursiveFunction(counter);
        }
}
```

Eine Funktion kann einen Wert zurückliefern. Wenn eine Funktion keinen Wert zurückliefert, dann ist Datentyp des Rückgabewertes „void“. Damit kann eine solche Funktion nicht Teil eines Ausdrucks sein.

```
void FunctionWithoutReturnValue()
{
        // do something
        return; // this statement is optional
}
```

Wenn eine Funktion einen Wert rückliefert, dann ist der Datentyp des Rückgabewertes ungleich „void“ angegeben. Somit kann die Funktion als Teil eines Ausdrucks verwendet werden, da nachdem die Funktion aufgerufen wurde diese einen Wert eines bestimmten Datentyps darstellt. Wenn eine Funktion einen Wert zurückliefert, dann muss die letzte Anweisung innerhalb der Funktion die „return“-Anweisung mit einem Ausdruck der den gleichen Datentyp darstellt wie der angegebene Datentyp des Rückgabewertes.

```
        // Return type of MyFunction is int,
        // so we can assign it to an int variable.
        int a = MyFunction(10);
```

Eine Funktion muss einen Namen haben. Der Name muss innerhalb seines Namensraums eindeutig. Allerdings gibt es die Möglichkeit Funktionen zu überladen. D.h. zwei Funktionen innerhalb des gleichen Namensraums dürfen den gleichen Bezeichner haben, aber die Parameterliste muss verschieden sein. Dies dient dazu die gleiche Funktionalität implementieren zu können, die aber für unterschiedliche Situationen verschiedene Parameter braucht. Das was die Funktion macht ist aber vom Prinzip her gleich.

```
// Function with 2 Parameters
double CalcSerialResistance(double R1, double R2)
{
        return R1+R2;
}

// Overloaded function with 3 Parameters
double CalcSerialResistance(double R1, double R2, double R3)
{
```

```
	return R1+R2+R3;
}

// This is NOT allowed, because only the return type is different
int CalcSerialResistance(double R1, double R2)
{
	return R1+R2;
}
```

Jede Funktion hat eine Parameterliste. Die Liste kann leer sein. In der Liste werden Parameter unter Angabe des Datentyps und des Bezeichners aufgelistet. Wenn mehrere Parameter vorhanden sind, dann müssen diese mit einem Komma („,“) getrennt werden. Die Parameter werden in der Funktion wie eine Variable verwendet, deren Wert vom Aufrufer gesetzt wurde.

```
// This is the definition of the function
int MyFunction(int parameter1, double parameter2, char parameter3)
{
	// do something
	if (parameter1 < 0)
	{
		return 10;
	}
	return 0;
}
```

Die Angaben Rückgabewert, Name und Parameterliste werden als Signatur der Funktion bezeichnet, da diese die Schnittstelle eindeutig beschreibt. Verwendet man nur die Signatur mit einem abschließenden Semikolon, dann wird das als „Funktionsdeklaration“ bezeichnet. Ab der Funktionsdeklaration ist die Funktion für den Compiler bekannt und kann in nachfolgenden Anweisungen verwendet werden. Genau genommen gehören die Namen der Parameter nicht zur Signatur und können bei der Deklaration weggelassen werden. Das ist auch wichtig für das Überladen von Funktionen. Überladenen Funktionen müssen sich, in den Datentypen der Parameter unterscheiden. Die Namen der Parameter und der Rückgabewert spielen keine Rolle.

```
// This is the signature of the function
int MyFunction(int parameter1, double parameter2, char parameter3);
```

Wenn der Signatur der Funktion die Implementierung in geschweiften Klammern folgt, dann spricht man von der „Funktionsdefinition“. Diese darf in einem Programm nur einmalig erfolgen. Ist die Funktionsdefinition vor der Verwendung im Programm erfolgt, dann ist keine Funktionsdeklaration mehr notwendig. Die Definition muss zum Zeitpunkt des Linkens verfügbar sein.

Eine Funktion wird aufgerufen, indem der Name der Funktion gefolgt von runden Klammern verwendet wird. In den Klammern müssen Werte, bzw. Ausdrücke, für die Parameter angegeben werden. Diese Werte werden als „Argumente“ der Funktion bezeichnet. Im täglichen Leben werden die Begriffe Parameter und Argument gerne vertauscht. Der Parameter ist eine Speicherstelle von einem bestimmten Typ, der beim Aufruf der Funktion mit einem Wert (Argument) gefüllt wird. Aus dem Kontext ist aber fast immer zu erkennen was gemeint ist.

```
	int argument1 = 10;
	double argument2 = 2.0;
	char argument3 = 'x';

	// Return type of MyFunction is int,
	// so we can assign it to an int variable.
```

```
	int a = MyFunction(argument1, argument2, argument3);
```

Die Besonderheit der Parameter ist, dass sie für jeden Funktionsaufruf ihren eigenen Speicher zugewiesen bekommen. Das hat zur Folge, dass wenn eine Funktion sich selbst aufruft, die Parameter nicht überschrieben werden sondern, nachdem die Funktion zurückkehrt wieder die alten Werte hat.

Auch Variablen und Objekte die innerhalb einer Funktion angelegt werden, werden mit jedem Funktionsaufruf neu angelegt. Möchte man, dass für eine Variable, bzw. ein Objekt immer derselbe Speicherbereich verwendet wird, dann muss man die Variable, bzw. das Objekt mit dem Schlüsselwort „static“ deklarieren.

```
// Function with a static variable
int CountUp()
{
	// Variable counter is only initialized on the first call.
	// With the next call it has the value of a previous call.
	static int counter = 0;
	counter++;
	return counter;
}

// Here we use the CountUp function multiple times
void UseCounter()
{
	for (int i=0; i<10; i++)
	{
		cout << CountUp << endl;
	}
}
```

Grundsätzlich sind alle angegebenen Parameter sogenannte „Wertparameter“, d.h. die Argumente werden in einen neuen Speicherbereich kopiert. Wurde als Argument eine Variable angegeben, dann kann die Variable bei einem Wertparameter nicht selbst verändert werden.

```
// A function with a value parameter
void ShowValueParameter(int thisIsAValueParameter)
{
	thisIsAValueParameter = 10;

}

// Here we call a function with a value parameter
void UseFunctionWithValueParameter()
{
	int argument = 1;

	ShowValueParameter(argument);

	cout << argument << endl; // Result is still 1!
}
```

Wird nach dem Datentyp des Parameters ein Kaufmannsund („&“) angegeben, dann ist der Parameter ein „Referenzparameter“. Bei Referenzen wird nicht der Wert eines Variablen, bzw. eines Objektes übergeben, sondern die Variable, bzw. das Objekt selbst. Damit ist es möglich die Variable eines Aufrufers innerhalb einer Funktion zu verändern.

```
// A function with a reference parameter
```

```
void ShowReferenceParameter(int& thisIsAReferenceParameter)
{
	thisIsAReferenceParameter = 10;

}

// Here we call a function with a reference parameter
void UseFunctionWithReferenceParameter()
{
	int argument = 1;

	ShowReferenceParameter(argument);

	cout << argument << endl; // Result is now 10!
}
```

Für Referenzparameter können keine Literale angegeben werden. Es muss sich also um Variablen oder Objekte handeln. Ist ein Argument ein konstanter Ausdruck (Schlüsselwort „const“ bei der Instanziierung) dann muss auch der Parameter als „const“ deklariert werden. Ansonsten kommt es zu Fehlermeldungen beim Kompilieren.

```
// A function with a constant reference parameter
void ShowConstReferenceParameter(const int& thisIsAConstReferenceParameter)
{
	// This is not allowed, because const parameters must not be changed.
	//thisIsAConstReferenceParameter = 10;

	if (thisIsAConstReferenceParameter < 10)
	{
		// do something
	}
}
```

Funktionen können immer unter dem Gesichtspunkt des Informationsflusses betrachtet werden. Hier stellt sich die Frage, welche Informationen fließen in eine Funktion und welche kommen aus der Funktion zurück an den Aufrufer. Wertparameter und konstante Referenzparameter fließen immer in die Funktion herein und werden damit als „In-Parameter“ bezeichnet. Der Rückgabewert ist wie der Name schon sagt eine Information die zurückfließt und damit ein „Out-Parameter“ Die nicht konstanten Parameter können „Out-Parameter“, „In-Parameter“ oder beides „In-Out-Parameter“ sein. Ist das Argument beim Aufruf egal, sondern wird in der Funktion der Parameter gesetzt und danach vom Aufrufer verwendet, dann ist es ein „Out-Parameter“. Wird der Parameterwert nur innerhalb der Funktion verwendet und nicht mehr vom Aufrufer benutzt. Dann ist es ein „In-Parameter“ in diesem Fall sollte der Parameter mit „const“ deklariert werden. Das ist guter Stil, aber nicht zwingenden. Wird das Argument des Aufrufers in der Funktion verwendet und dann mit einem neuen Wert überschrieben, um vom Aufrufer weiterverwendet zu werden, dann spricht man von einem „In-Out-Parameter“.

Da mit dem Rückgabewert nur ein einziger Wert, bzw. Objekt zurückgegeben werden kann, kann man mit den Referenzparametern mehrere Werte zurückliefern.

Felder

Bisher wurden immer nur mit einzelnen Informationen gearbeitet. Es passiert aber sehr häufig, dass man mehrere Informationen vom gleichen Typ verarbeiten möchte.

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

0x1000	0
0x1004	1
0x1008	2
0x100C	3
0x1010	4
0x1014	5
0x1018	6
0x101C	7

myArray <---> 0x1000

Ein Beispiel sind 10 Messwerte die in einer Messreihe aufgenommen wurden. Jeder Messwert kann durch den Datentyp „double“ dargestellt werden. Jetzt wäre es hilfreich, wenn man nicht 10 Variablen einzeln anlegen müsste, sondern eine Variable mit 10 Elementen. Außerdem sollte jedes Element über einen Ausdruck (also z.B.) eine Indexnummer zugreifbar sein, damit das Einlesen, Ausgeben und Verarbeiten mit der Hilfe von Zählschleifen erfolgen kann.

```
// if you don't use arrays your code would look like this
double measureValue0 = 0.0;
double measureValue1 = 0.0;
double measureValue2 = 0.0;
double measureValue3 = 0.0;
double measureValue4 = 0.0;
double measureValue5 = 0.0;
double measureValue6 = 0.0;
double measureValue7 = 0.0;
double measureValue8 = 0.0;
double measureValue9 = 0.0;

cin >> measureValue0;
cin >> measureValue1;
cin >> measureValue2;
cin >> measureValue3;
cin >> measureValue4;
cin >> measureValue5;
cin >> measureValue6;
// ...
```

Genau das leisten Felder (engl. Arrays). Sie sind Variablen die aus einer bestimmten Anzahl von Daten eines Datentyps bestehen. Sie werden wie eine normale Variable angelegt, aber hinter dem Namen werden in eckigen Klammern die Anzahl der Elemente angegeben.

Beispiel:

```
// this is how it looks like if you use arrays
```

```
double measureValues[10];

for (int i=0; i<10; i++)
{
	cin >> measureValues[i];
}
```

Um auf ein Element zugreifen zu können muss der Feldname gefolgt von eckigen Klammern und der Indexnummer des Feldes angegeben werden. Sowohl beim Lesen als auch beim Schreiben. Das erste Feld hat immer den Index 0. Das letzte Element hat demnach die Anzahl der Elemente-1als Index. Es ist ratsam die Feldgröße über eine Konstante festzulegen und entsprechend im Code die Konstante zu verwenden. Auf diese Weise kann die Feldgröße sehr einfach geändert werden.

```
// it is useful to use a constant to define the size of the array
// this makes your code more flexible.
const int arraySize = 10;
double myArray[arraySize];

for (int i=0; i<arraySize; i++)
{
	myArray[i] = 0.0;
}
```

Wird nur der Feldname verwendet, dann wird dieser als Adresse des ersten Elementes aufgelöst. Zu beachten ist, dass man den Namen eines Feldes nicht direkt in „std::cin“ oder „std::cout“ verwenden kann um sich den Inhalt aller Elemente ausgeben lassen zu können, da hiermit nur die Adresse des ersten Elementes gemeint ist.

```
// cin >> myArray; // This is not allowed

cout << myArray; // prints out a memory adress and not the elements
```

Die eckigen Klammern sind der Index-Operator. Dieser erwartet vor der Klammer die Angabe einen Ausdruck der eine Speicheradresse und einen Datentyp darstellt. In der Klammer wird angegeben um wie viel Elemente von dieser Startadresse im Speicher weiter gerückt werden muss. Da mit dem Datentyp eine Speichergröße verbunden ist, wird die Adresse des Elementes mit der folgenden Formel berechnet.

$$Elementadresse = Startadresse + Elementgröße \cdot Indexnummer$$

Das führt dazu, dass der Indexoperator auch auf die später erläuterten Zeiger angewendet werden kann.

Eine Feldvariable kann Initialisiert werden, in dem eine Liste von Elementen in geschweiften Klammern mit einem Gleichheitszeichen („=“) zugewiesen wird. Die Elemente werden mit einem Komma getrennt.

```
// Array definition with initializiation
double initializedArray[3] = { 1.1, 2.2, 3.3};
```

Die Größe von Feldern müssen beim Instanziieren angegeben werden. Allerdings kann die Größe auch beim Kompilieren aus der Initialisierung berechnet werden. Der Kompilierer nimmt die Anzahl der Elemente die bei der Initialisierung angegeben wurde als Feldgröße.

```
// This array has 4 elements
```

```
	char automaticSizedArray[] = { 'a','b','c', 0 };
```

Felder können auch als Parameter verwendet werden. Sie werden wie bei der Variablendeklaration mit eckigen Klammern nach dem Parameternamen und der Feldgröße angegeben.

```
// This function receives an array as a parameter
void FunctionWithArrayParameter(int arrayParameter[3])
{
	for (int i=0; i<3; i++)
	{
		cout << arrayParameter[i];
	}
}

// call a function and hand over an array.
void CallFunctionWithArrayParameter()
{
	int arrayArgument[3] = {1, 2, 3};
	FunctionWithArrayParameter(arrayArgument);
}
```

Um ein Feld mit den Werten eines anderen Feldes zu füllen (Zuweisung), müssen diese Elementweise kopiert werden. Die Schreibweise „feld1 = feld2;“ funktioniert nicht.

```
	int array1[3];
	int array2[3] = {1, 2, 3};

	// array1 = array2; // this is not allowed

	// you must copy each element;
	for (int i=0; i<3; i++)
	{
		array1[i] = array2[i];
	}
```

Da die Elemente eines Feldes im Speicher immer direkt hintereinander stehen, kann mit der Standard-Bibliotheksfunktion „memcpy(destinationAddress, sourceAddress, sizeInByte)“ auch ein ganzer Speicherblock kopiert werden. Als Startadresse wird der Feldnamen und für die Größe die Feldgröße („sizeof(array)“) angegeben.

```
	// or use memcpy from standard library
	memcpy(array1, array2, sizeof(array1));
```

Lektion 6

Aufzählungen (Enumerationen)

Häufig hat man Werte, die einen bestimmten Zustand darstellen sollen. Gerade in Statusmaschinen aber auch in vielen anderen Aufgaben. Da es nicht besonders eingängig ist sich die Zahlen zu merken ist es sinnvoll, diesen Zahlen einen beschreibenden Namen zu geben. Das kann zum Beispiel durch die Prä-Prozessordefinition oder durch konstante Variablen erfolgen. Wenn aber eine Maschine verschiedene Zustände hat, dann wäre es gut einen Datentyp zu haben, der genau diese Zustände darstellen kann. Dies lässt sich mit Enumerationen erreichen.

Enumerationen sind intern vom Datentyp int, also können entsprechend viele Werte annehmen. Beim Definieren des Datentyps werden bestimmten Werten Namen zugewiesen. Wobei die Nummernvergabe für die Namen auch automatisch erfolgen kann.

Eine Enumeration beginnt mit dem Schlüsselwort „enum“ gefolgt für den Namen der Enumeration. Danach folgen, in geschweiften Klammern, die Liste der Namen und deren zugewiesenen Werte. Die Werte werden mit dem Gleichheitszeichen („=“) zugewiesen. Ist kein Wert angegeben, dann wird der vorige Wert +1 zugewiesen. Wurde nie ein Wert angegeben, dann wird als erstes der Wert 0 zugewiesen. Die verschiedenen Elemente werden durch ein Komma getrennt.

```
// Sample of an enumeration
enum SystemStates
{
	Invalid = -1,
	Initializing = 0,
	Initialized = 1,
	Started, // compiler assing value 2 automatically
	Running,
	Stopped,
	NumberOfState // useful trick to enumerate the states in a for-loop
};
```

Da eine Aufzählung ein Zahlentyp ist, kann er für die Selektion („switch“-Anweisung) verwendet werden.

```
// Here you see the usage of the enumeration in many cases.
// First of all the enumeration is a datatype and can be used like int or double in
the parameter list
void PrintCurrentState(SystemStates currentState)
{
	// Here we use the values for the case-blocks of a switch statement
	switch(currentState)
	{
	case SystemStates::Initialized:
		cout << "System is initialized." << endl;
		break;
	case SystemStates::Started:
		cout << "System is started." << endl;
		break;
	case SystemStates::Stopped:
		cout << "System is stopped." << endl;
		break;
	case SystemStates::Running:
		cout << "System is running." << endl;
		break;
	default:
		cout << "Panik!!! System is in an invalid state!" << endl;
	}
}
```

Die Verwendung von Aufzählungen kann die Lesbarkeit von Quellcode deutlich erleichtern, da nicht kryptische Zahlen, sondern sprechende Namen verwendet werden.

Die Aufzählung ist ein Datentyp der vom Entwickler erstellt wurde. Damit ist es auch möglich einem Parameter mehr Klarheit zu geben. Ist der Typ eines Parameters eine Enumeration, dann ist klar dass

nur die Werte der Enumeration sinnvolle Werte sind. Der Datentyp „int“ würde genauso funktionieren, aber weniger semantische Information liefern.

Wie bei allem in C++ gilt, dass es erst deklariert sein muss, bevor es verwendet werden kann.

Strukturen

In C++ sind Enumeration nicht die einzige Möglichkeit eigene Datentypen zu erstellen. Um Datentypen zu erstellen, die aus mehreren unterschiedlichen Datentypen bestehen, gibt es die Strukturdatentypen. Diese müssen im Programm zuerst deklariert werden, bevor sie dann verwendet werden können. Üblicherweise erfolgt eine solche Deklaration in einer Header-Datei.

Eine Struktur-Deklaration beginnt mit dem Schlüsselwort „struct“ gefolgt von dem Namen der Struktur. Danach muss in geschweiften Klammern {…} eine Liste mit den Elementen folgen. Die Deklaration wird mit einem **Semikolon** abgeschlossen

Beispiel:

```
// Declare a structure of type complex
struct SComplex
{
          double real;
          float imag;
};
```

Strukturdatentypen werden wie einfache Datentypen instanziiert. Um die Strukturvariable zu initialisieren muss nach dem Gleichheitszeichen die Werte für die Elemente der Struktur in geschweiften Klammern angegeben werden. Die Werte müssen in der Reihenfolge wie sie in der Definition stehen und durch Komma getrennt angegeben werden.

```
      // Definition of a complex variable and initialization
      SComplex complexNumber = {1.0, 2.0};
```

Ein Strukturvariable (Variable von einem Strukturdatentyp) kann ich direkt über den Variablennamen mit „std::cin“ beschrieben, bzw. mit „std::cout“ ausgegeben werden. Hinter dem Namen der Strukturvariablen verbirgt sich, ähnlich wie bei Feldern, die Adresse des ersten Elementes.

```
      // cout << complexNumber; // This does not work. Cout don't know about your
type.
```

Um auf die Elemente einer Strukturvariablen zugreifen zu können muss der Variablenname gefolgt von einem Punkt („.“) und dem Elementnamen verwendet werden.

```
      // We must acces each member separatly
      cout << "(" << complexNumber.real << "," << complexNumber.imag << ")" << endl;
```

Eine Besonderheit ergibt sich beim Zuweisen von Strukturen. Wird eine Strukturvariable einer anderen Strukturvariable zugewiesen, dann werden die Inhalte der Elemente der einen Variablen in die Elemente der anderen Variablen kopiert. Das ist daher besonders, weil die Variablennamen eigentlich als Adressen ausgewertet werden. Trotzdem findet hier ein elementweises Kopieren statt.

```
      SComplex complexNumber1;
      SComplex complexNumber2 = {2.3, 5.1};

      // This copies the elements and work fine
      complexNumber1 = complexNumber2;
```

Strukturen haben noch weitere Eigenschaften auf die in diesem Kurs nicht näher eingegangen werden soll. So können z.B. Konstruktoren und Destruktoren, Operatoren und Methoden definiert werden. Auch die Sichtbarkeit von Elementen und Methoden kann eingeschränkt werden. Damit haben Strukturen sehr große Ähnlichkeit zu Klassen. Allerdings wollen wir hier die Strukturen nur als Sammeldatentyp verwenden, um verschiedene Informationen unterschiedlicher Datentypen als eine Einheit behandeln zu können.

Zeichenketten

Wir haben bereits einzelne Zeichen verwendet und sogar ganze Texte die als Zeichenketten vorliegen, ohne diese genauer betrachtet zu haben. Jetzt ist es aber so weit, dass wir uns hier mit beschäftigen können.

Der Computer kennt eigentlich nur Zahlen, die er als Bitkombinationen verarbeiten und speichern kann. Zeichen und Texte kann er direkt nicht verarbeiten. Um aber auch Zeichen und Text verarbeiten zu können, werden die Zeichen mit Code-Nummern versehen. Es wird also jedem Zeichen eine Nummer zugewiesen. Eine solche Zuordnung könnte sehr willkürlich sein, daher gibt es Standards die diese Zuordnung vereinheitlichen. Leider gibt es verschiedene Standards die immer wieder zu Verwirrungen führen können.

Klassisch wurde unter dem Betriebssystem mit dem ASCII-Zeichensatz gearbeitet. Dieser ordnet nicht nur den Ziffern und Buchstaben Code-Nummer zu, sondern kodiert auch Sonderzeichen, Steuerbefehle (wie den Zeilenumbruch) und weiter Symbole. Der ASCII-Zeichensatz verwendet 8 Bit für die Codierung eines Zeichens, so dass 256 verschiedene Zeichen, Ziffern und Symbole kodiert werden können.

000	NUL	033	!	066	B	099	c	132	ä	165	Ñ	198	ã	231	þ
001	Start Of Header (SOH)	034	"	067	C	100	d	133	à	166	ª	199	Ã	232	Þ
002	Start Of Text (STX)	035	#	068	D	101	e	134	å	167	º	200	╚	233	Ú
003	End Of Text (ETX)	036	$	069	E	102	f	135	ç	168	¿	201	╔	234	Û
004	End Of Transmission (EOT)	037	%	070	F	103	g	136	ê	169	®	202	╩	235	Ù
005	Enquiry	038	&	071	G	104	h	137	ë	170	¬	203	╦	236	ý
006	Acknowledge (ACK)	039	'	072	H	105	i	138	è	171	½	204	╠	237	Ý
007	Bell	040	(	073	I	106	j	139	ï	172	¼	205	═	238	¯
008	Backspace (BS)	041	)	074	J	107	k	140	î	173	¡	206	╬	239	´
009	Horizontal Tab	042	*	075	K	108	l	141	ì	174	«	207	¤	240	-
010	Line Feed **(LF)**	043	+	076	L	109	m	142	Ä	175	»	208	ð	241	±
011	Vertical Tab	044	,	077	M	110	n	143	Å	176	░	209	Ð	242	‗
012	Form Feed **(FF)**	045	-	078	N	111	o	144	É	177	▒	210	Ê	243	¾
013	Carriage Return **(CR)**	046	.	079	O	112	p	145	æ	178	▓	211	Ë	244	¶
014	Shift Out	047	/	080	P	113	q	146	Æ	179	│	212	È	245	§
015	Shift In	048	0	081	Q	114	r	147	ô	180	┤	213	ı	246	÷
016	Dataline Escape (DLE)	049	1	082	R	115	s	148	ö	181	Á	214	Í	247	¸
017	DC 1 (XON)	050	2	083	S	116	t	149	ò	182	Â	215	Î	248	°
018	DC 2	051	3	084	T	117	u	150	û	183	À	216	Ï	249	¨
019	DC 3 (XOFF)	052	4	085	U	118	v	151	ù	184	©	217	┘	250	·
020	DC 4	053	5	086	V	119	w	152	ÿ	185	╣	218	┌	251	¹
021	Negative Acknowledge (NAK)	054	6	087	W	120	x	153	Ö	186	║	219	█	252	³
022	Synchronous Idle	055	7	088	X	121	y	154	Ü	187	╗	220	▄	253	²
023	End Of Transmission Block	056	8	089	Y	122	z	155	ø	188	╝	221	¦	254	■
024	Cancel	057	9	090	Z	123	{	156	£	189	¢	222	Ì	255	
025	End Of Medium	058	:	091	[	124	|	157	Ø	190	¥	223	▀		
026	Substitude	059	;	092	\	125	}	158	×	191	┐	224	Ó		
027	Escape **(ESC)**	060	<	093	]	126	~	159	ƒ	192	└	225	ß		
028	File Seperator	061	=	094	^	127 (DEL)	⌂	160	á	193	┴	226	Ô		
029	Group Seperator	062	>	095	_	128	Ç	161	í	194	┬	227	Ò		
030	Record Seperator	063	?	096	`	129	ü	162	ó	195	├	228	õ		
031	Unit Seperator	064	@	097	a	130	é	163	ú	196	─	229	Õ		
032	*SPACE* (SP)	065	A	098	b	131	â	164	ñ	197	┼	230	µ		

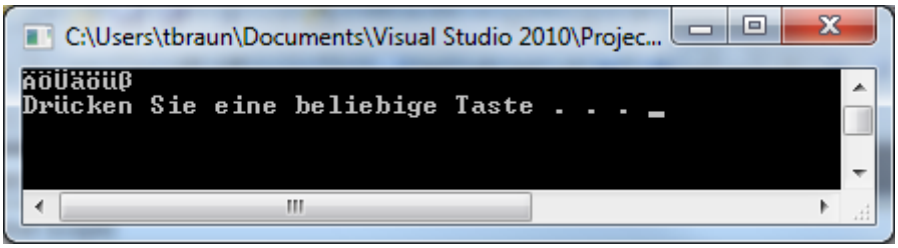
Tabelle 3: ASCII Tabelle

Mittlerweile gibt es sehr viele Standards z.B. den ISO 8859-1 der ebenfalls Zeichen in 8 Bit kodiert. Ein anderer Zeichensatz der bei Windows zum Einsatz kommt, heißt „Windows-1252“ oder auch „CP 1252“ (CP steht für Code-Page). Das ist auch der Grund warum es in unseren Programmen immer wieder dazu kommt, dass die Zeichen die wir im Quellcode ausgegeben haben nicht immer die sind die dann auch ausgegeben werden. Es kommt darauf an, welche Kodierung die ausgebende Komponente verwendet. Wenn wir also mit „std::cout“ Text ausgeben, dann wird für die angegeben Zeichencodes das Zeichen ausgegeben, dass in „std::cout“ für diese Codenummer vorgesehen ist. Es besteht aber die Möglichkeit die zu verwendete Kodierung einzustellen. Dafür gibt es die Bibliotheksfunktion „std::local::global(std::local)“. Als Parameter der Funktion muss eine Spracheinstellung angegeben werden. Wir können uns aber auch einfach die Zeile merken, die das macht.

```
#include <iostream>

void main()
{
	// Sorgt dafür, dass auch die deutschen Umlaute erkannt werden.
	std::locale::global(std::locale("German_germany"));
	// Ausgabe der Umlaute
	std::cout << "ÄÖÜäöüß" << std::endl;

	system("PAUSE");
}
```



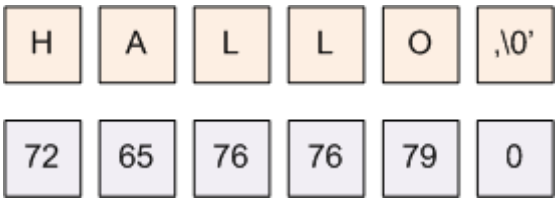
Da die Globalisierung verlangt, dass Computerprogramme Texte in allen Sprachen der Welt ausgeben können müssen und 256 Kombinationen nicht genug dafür sind, ist heute der Unicode Standard für die Textcodierung. Dieser Code ist etwas flexibler. Er kann bis zu 4 Byte pro Zeichen verwenden. Die Tatsache der er unterschiedlich viele Bytes pro Zeichen verwendet ist er doch etwas komplizierter. Für die genauen Möglichkeiten möchte ich auf die weiterführende Literatur und das Internet verweisen. Es genügt erst mal zu wissen, dass es unterschiedliche Kodierungstabellen gibt, und dass diese Unterschiedlich viele Bytes pro Zeichen verwenden können.

ANSI-C Strings

Da wir hier mit 1 Byte Zeichensätzen arbeiten, können die Zeichen im Datentyp „char“ gespeichert werden. Daher hat dieser Datentyp auch seinen Namen. „std::cout“ interpretiert auch Daten vom Typ „char“ automatisch als Zeichen und gibt das zugehörige Zeichen aus anstatt des Zahlenwertes.

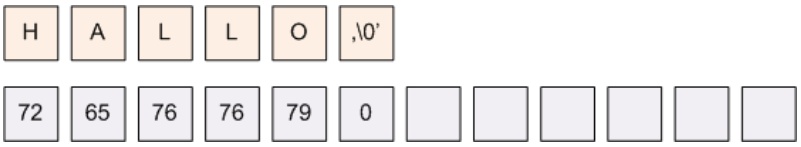
Möchte man mehr als ein Zeichen ausgeben, dann ist das eine Sequenz von „char“ Daten und folgerichtig werden Zeichenketten als „char“-Felder gespeichert. Dabei gibt es eine Besonderheit. Das Ende eines Textes wird durch den Zeichenwert „0“ gekennzeichnet.

```
char myString[6] = "HALLO";
```



Wird also ein Text aus einem „char"-Feld gelesen, dann wird das Feld so lange ausgelesen bis der Wert „0" gefunden wird, auch wenn das Feld noch mehr Zeichen aufnehmen könnte. Dies „0" wird als „terminierende Null" bezeichnet.

```
char myString[12] = "HALLO";
```



Wurde hingegen die terminierende Null ausversehen vergessen, dann wird so lange weiter gelesen, bis im Speicher zufällig eine „0" gefunden wurde. Was zu Fehlern und Programmabstürzen führen kann.

Bei Zeichenketten-Literalen, also den Texten die wir in doppelten Anführungszeichen angeben, wird automatisch die terminierende Null hinzugefügt. Wird statt den doppelten Anführungszeichen, die einfachen Anführungszeichen verwendet, dann wird keine terminierende Null angehängt.

Bibliotheksfunktionen für Zeichenketten

Da Zeichenketten nichts weiter als Felder sind, gelten auch die ganzen Regeln für Felder. Das heißt, eine Zeichenkette kann nicht einfach zugewiesen werden. Allerdings kann eine Feld mit einer Zeichenkette initialisiert werden.

```
char myString[6] = "HELLO";

//Error you can't assign directly
myString = "Hello Again";
```

Auch das Vergleichen mit Vergleichsoperatoren ("==" oder „!=") geht entsprechend schief. Daher gibt es eine Standard-Bibliothek „string.h", die Funktionen für Strings bereitstellt. Wie aus dem Namen hervorgeht ist das eine „C" Bibliothek und demnach nicht objekt-orientiert.

Function	Explanation	Sample	Sample result

char* **strcat**(strDestination, strSource)	Append one string to another	`char strDest[255] = „Hello";`
		`const char strSource[] = „ World!";`
		`strcat(strDest, strSource);`
		`cout << strDest << endl;`
			Hello World!

int **strcmp**(str1, str2)	Compare two strings
	< 0 : str1 < str2
	= : str1 == str2
	> 0 str1 > str2
		`int result = 0;`
		`result = strcmp(„after","less");`
		`cout << result << endl;`
			-1

char* **strcpy**(strDestination, strSource)	Copy one string to another	`char strDest[255] = „Hello“; const char strSource[] = „World!“; strcpy(strDest, strSource); cout << strDest << endl;`	World!
size_t **strlen**(str)	Find length of string	`cout << strlen(„Hello World!“) << endl;`	12
char* **_strlwr**(str)	Convert string to lowercase	`char myString[16] = „HELLO“; _strlwr(myString); cout << myString << endl;`	hello
char* **_strupr**(str)	Convert string to uppercase	`char myString[16] = „hello“; _strupr(myString); cout << myString << endl;`	HELLO

Die Funktion „strcat()“ verbindet 2 Zeichenketten.

Die Funktion „strcmp()“ vergleicht zwei Zeichenketten. Sind die Zeichenketten gleich, dann liefert sie den Wert „0“, was zu etwas ungewohnt Bedingungsprüfungen führt.

Die Funktion „strcpy()“ kopiert eine Zeichenkette in eine „char“-Feld, also in eine andere Zeichenkettenvariable.

Die Funktion „strlen()“ liefert die Anzahl an Zeichen i einer Zeichenkettenvariablen. Es werden alle Zeichen aus der terminierenden Null gezählt. Auch wenn das Feld noch mehr freie Felder hat, wird nur bis zur terminierenden Null gezählt.

Die Funktionen „_strlwr()“ und „_strupr()“ sind nicht standardkonform und beginnen daher mit einem Unterstrich. Sie wandeln alle Buchstaben in Kleinbuchstaben, bzw. in Großbuchstaben.

C++ Zeichenketten

Die Bibliotheksfunktionen in „string.h“ sind Funktionen und nutzen demnach nicht die Vorteile der Objektorientierung. Sind sind aber immer noch Hilfreich, wenn man Zeichenketten direkt manipulieren muss, was bei der Mikrocontroller Programmierung immer wieder vorkommt. Handelt es sich aber um eine Server oder Desktop Anwendung, dann sollte man nicht auf die Vorteile von C++ verzichten. Die Standardbiliothek „string“ definiert einen Datentyp, genau gesagt eine Klasse für Zeichenketten. Die Klasse hat den Name „std::string“ liegt also im Standardnamensraum.

Die Klasse überlädt Vergleichs-, Zuweisungs-, und den Plusoperator, um komfortabler mit Zeichenketten arbeiten zu können.

```
// String-Objekt erstellen und initialisieren
std::string myString = "Hallo objektorientierte Welt!";
// Gleich mal ausgeben
std::cout << myString << std::endl;

// Vergleich von String-Objekten ist jetzt möglich
if (myString == "Hello")
{
	// Do something
```

```
}

// Die Zuweisung funktioniert auch sehr gut.
myString = "geänderter String";
std::cout << myString << std::endl;
```

Hier ist der Wehmutstropfen, dass die Klasse keine Methode für das Konvertieren in Klein- bzw. Großschreibung bereithält. Hier muss man sich dann selbst helfen.

```
// Mit Length bekommt man die Anzahl Zeichen einer Zeichenkette
for (int i=0; i<myString.length(); i++)
{
	// und mit dem Index-Operator kann man auf die einzelnen Zeichen zugreifen.
	std::cout << myString[i] << std::endl;
}
```

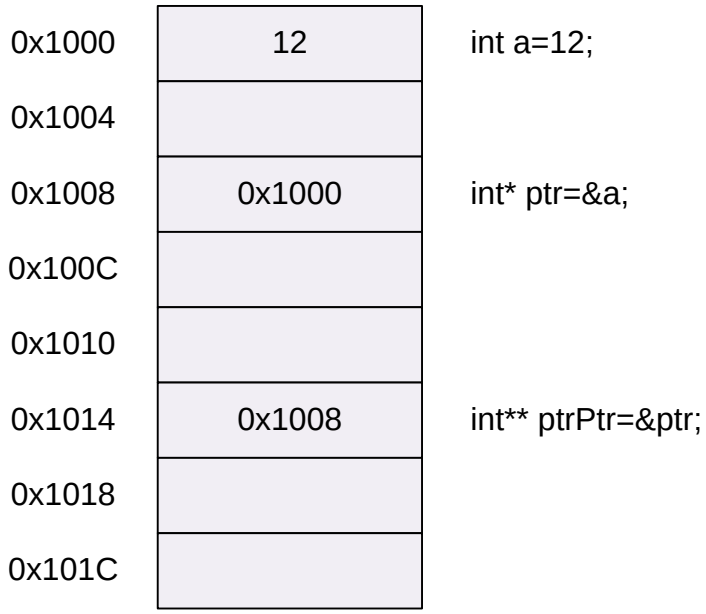
Lektion 7

Zeiger

Die Basisdatentypen sind Wertdatentypen. Das heißt sie stellen den Wert, bzw. die Information direkt da. Operatoren wirken auf die Werte. Felder sind indizierte Datentypen. Das heißt die Informationen werden mit einem Index erreicht. Da alle Variablen eine Speicheradresse besitzen ab der der Variableninhalt zu finden ist, kann man sich auch einen indirekten Zugriff vorstellen, bei dem nicht direkt auf die Speicherstelle zugriffen wird, sondern man sich nur die Adresse merkt und den Datentyp der ab der Adresse zu finden ist. Um mit solchen Adressangaben zu arbeiten gibt es den Datentyp der Zeiger. Zeigervariablen haben einen Datentyp, aber anstatt eine Information dieses Datentyps zu speichern wird nur die Adresse abgespeichert ab der diese Information zu finden ist.

Die Zeiger haben den Vorteil, dass man große Datenmengen nicht immer kopieren muss, wenn man sie z.B. an eine Funktion übergibt. Statt eine Kopie an die Funktion zu übergeben, wird nur die Adresse ab der die Information zu finden ist kopiert. Ein anderer Vorteil von Zeigern ist, dass man sie zur Laufzeit verändern kann, also auf eine andere Stelle zeigen lassen kann. Es ist also möglich im die gleiche Zeigervariable im Programm zu verwenden, aber die damit verbundene Information zu verändern. Damit lassen sich z.B. verkettet Listen oder Ringpuffer erstellen. Dazu aber später mehr. Wichtig ist auch, dass man mit Zeigern auf Adressen operieren kann, ohne in seinem Programm explizit den Adresswert angeben zu müssen.

Es bleibt nochmals festzuhalten: „Zeiger sind Variablen die auf andere Variablen zeigen und der Inhalt einer Zeigervariable ist die Adresse der Variablen, auf die die Zeigervariable zeigt.“



Eine Zeigervariable wird im Quellcode definiert, in dem ein Stern („*“) vor den Variablennamen, bzw. nach dem Datentyp gesetzt wird.

Wird eine Zeigervariable mit „std::cout“ ausgegeben, dann wird folgerichtig eine Adresse ausgeben. Möchte man die Variable auf die gezeigt wird verändern, dann muss man vor die Variable einen Stern („*“) setzen. In diesem Fall ist der Stern der Dereferenzierungsoperator. Für das Beschreiben der Variablen gilt das gleiche.

Bei der Definition ist es wichtig anzugeben, auf was der Zeiger zeigt. Die Zeigervariable ist ja nur eine Adresse und dahinter kann sich ja alles Mögliche verbergen. Daher muss bei der Definition nicht nur ein Stern angegeben werden sondern auch ein Datentyp. Zeigt eine Zeigervariable auf eine andere Zeigervariable, dann müssen entsprechend 2 Sterne nach dem Datentyp verwendet werden.

Beispiel:

```
int a=12;

int* ptr2a = &a;

int ** ptr2ptr2a = &ptr2a;

std::cout << "Content of a: " << a << std::endl;
// Output: Content of a: 12

std::cout << "Address of a: " << &a << std::endl;
// Output: Adress of a: 0x34A2F54

std::cout << "Address where ptr2a points to: " << ptr2a << std::endl;
// Output: Address where ptr2a points to: 0x34A2F54

std::cout << "Content where ptr2a points to: " << *ptr2a << std::endl;
// Output: Content where ptr2a points to: 12

std::cout << "Address of ptr2a: " << &ptr2a << std::endl;
// Output: Address of ptr2a: 0x24A2FA2

std::cout << "Address where ptr2ptr2a points to: " << ptr2ptr2a << std::endl;
// Output: Address where ptr2ptr2a points to: 0x24A2FA2

std::cout << "Content where ptr2ptr2a points to: " << *ptr2ptr2a << std::endl;
// Output: Content where ptr2ptr2a points to: 0x34A2F54
```

```
std::cout << "Content of the content where ptr2ptr2a points to: " << **ptr2ptr2a << std::endl;
// Output: Content of the content where ptr2ptr2a points to: 12

std::cout << "Address of ptr2ptr2a: " << &ptr2ptr2a << std::endl;
// Output: Address of ptr2ptr2a: 0x1DA2E20
```

Zeigerarithmetik

Für Zeiger gilt eine eigene Arithmetik. D.h. die Operationen „+“ und „-„ verhalten sich etwas anders als es für z.B. ganzzahlige Datentypen gilt. Wird einer Zeigervariable der Wert 1 hinzuaddiert, dann wird nicht die gespeicherte Adresse nicht unbedingt um ein Byte erhöht, sondern um so viel wie für den Datentyp, auf den gezeigt wird, benötigt wird. Z.B. addiert man eine 1 auf einen Zeiger, der auf „double“ Datenzeigt, wird die im Zeiger gespeicherte Adresse um 8 Byte erhöht.

```
double ptr = 0x1000;

std::cout << "Content of ptr: " << ptr << std::endl;
// Output: Content of ptr: 0x1000

ptr++;

std::cout << "Content of ptr: " << ptr << std::endl;
// Output: Content of ptr: 0x1008
```

Indizierter Zugriff

Möchte man ab einer bestimmten Adresse auf verschieden weit entfernte Elemente zugreifen, kann man genau wie bei Feldern den Indexoperator verwenden. Hier ist auch der Zusammenhang zu Feldern gegeben. Der Inhalt einer Feldvariablen ist die Adresse des ersten Elementes und der Index gibt an um wie viele Elemente das gesuchte Element davon entfernt ist. Somit ist es auch möglich, eine Feldvariable direkt einer Zeigervariablen zuzuordnen.

```
// Create an array with 10 elements an initialize it.
int myArray[10] = {0,1,2,3,4,5,6,7,8,9};
// Create an int pointer and set it to the first element of the array
int* ptrElement = myArray;

// Modify the 5 element started from the pointer
ptrElement[5] = 50;

// Set the pointer to the 5th element of the array
ptrElement = &myArray[5];

// modify the 5th element of the array indirect.
*ptrElement = 5;
```

Zeiger können auch für Funktionsparameter und stellen damit eine Alternative für Referenzparameter da. Wenn ein Feld als Parameter übergeben werden soll, aber nicht klar ist wie viele Elemente im Feld sein werden, dann kann man auch den Zeiger verwenden und einen zusätzlichen Parameter der angibt wie viele Elemente das Feld enthält.

```
void PrintArray( double* ptrArray, int size)
{
	for (int i=0; i<size; i++)
	{
		std::cout << ptrArray[i] << std::endl;
	}
}
```

Zeigt ein Zeiger auf eine Struktur oder auf eine Klasse, dann will man beim Derefenzieren meistens auf ein Element der Struktur, bzw. Klasse, zugreifen. Da erweist sich der Stern für die Derefenzierung als etwas umständlich. Stattdessen kann man einen Pfeil („->“) aus einem Minuszeichen und eine Größer-als-Zeichen verwenden.

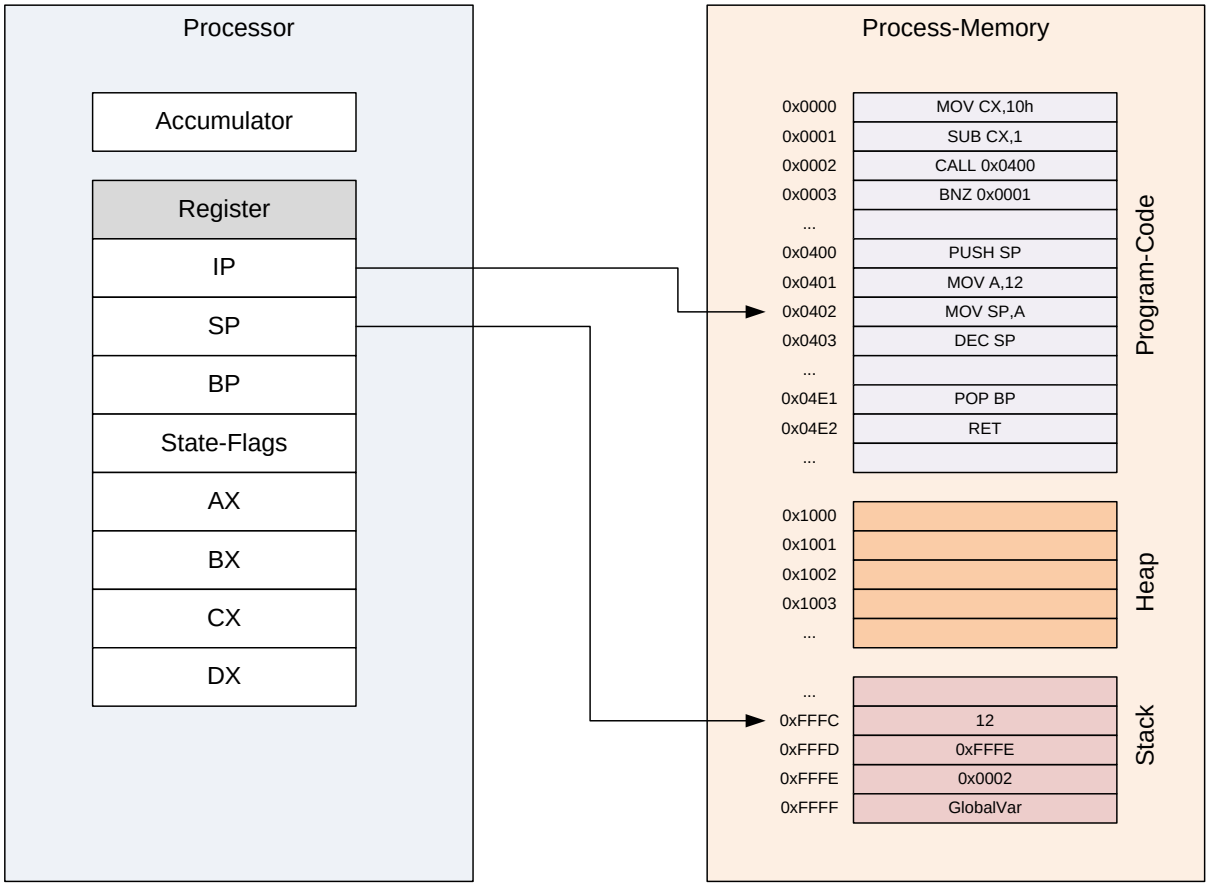
```
SComplex cmp = {1.1 , 2.2};
SComplex* ptr2cmp = &cmp;

// Following lines are equal and do exactly the same
(*ptr2cmp).real = 1.2;
ptr2cmp->real = 1.2;
```

Speicherverwaltung

In Betriebssystemen wie Windows entspricht jede Anwendung einem Prozess. Ein Prozess bekommt einen virtuellen Speicher zugeteilt in dem die Anweisungen und die Daten der Anwendung abgelegt werden. Der Speicher ist virtuell, da er nicht mit dem physikalisch eingebauten Speicher übereinstimmen muss. Aber für unser Programm sieht es so aus, als hätte es diesen Speicher und zwar nur für sich allein.

Wird ein Programm gestartet, dann wird vom Betriebssystem ein Prozess angelegt, der Prozess erhält einen virtuellen Speicher und die Anweisungen werden in den virtuellen Speicher geladen. Zusätzlich wird für die Anwendung ein Speicherbereich bereitgehalten, der zur Programmlaufzeit angefordert werden kann und dann auch wieder freigeben werden muss. Dieser Speicher nennt sich „Heap“. Die Laufzeitumgebung von C++ verwaltet diesen Speicher. Um im Programm Speicher auf dem „Heap“ zu reservieren gibt es den „new“-Operator. Der Operator „delete“ gibt den Speicher wieder frei.



Die andere Möglichkeit um Speicher zu reservieren ist der „Stapel“ (engl. Stack). Der Stapelspeicher hat die Eigenschaft, dass man das was man zuletzt drauf getan hat als erstes wieder herunternehmen muss.

Der Stapelspeicher wird ebenfalls im virtuellen Speicher des Prozesses erstellt. Für den Stapelspeicher gibt es eine Hardware-Unterstützung. Der Prozessor hat ein Register, in der er die aktuelle Position des letzten Elementes auf dem Stapel speichert und es gibt Befehle um Daten vom Speicher in den Prozessor zu laden („pop“), bzw. auf dem Stapelspeicher abzulegen („push“). Damit ist der Stapel sehr effizient benutzbar.

Alle lokalen Variablen werden auf dem Stapelspeicher angelegt. Sobald die Variable aus ihrem Scope geht, wird sie vom Stapelspeicher gelöscht.

Wird eine Funktion aufgerufen, dann werden die Rücksprungadresse und die zu übergebenden Argumente auf den Stapel gelegt. Wird die Funktion verlassen, werden die Argumente und die Rücksprungadresse vom Stapel geholt. Damit steht der Speicher wieder zur Verfügung.

Heap Speicher verwenden

Möchte man zur Laufzeit eine Variable auf dem Heap anlegen dann muss das Schlüsselwort „new“ gefolgt von einem Datentyp verwendet werden. Optional kann in runden Klammern noch Werte für die Initialisierung angegeben werden.

Der „new“-Operator liefert die Adresse ab der die Variable im Heap-Speicher zu finden ist. Demnach muss das Ergebnis des Operators einer Zeigervariablen vom gleichen Typ zugewiesen werden.

Sobald die Variable nicht mehr benötigt wird, muss die Variable mit dem „delete“-Operator wieder freigeben werden. Nach dem Schlüsselwort „delete“ muss die Zeigervariable angegeben werden, die auf den freizugebenden Speicher zeigt. Die Anzahl der freizugebenden Bytes ergibt sich aus dem Datentyp.

```
// Reserve memory from heap and initialize it
int* heap = new int(1);
// Free memory
delete heap;
```

Möchte man hingegen ein Feld auf dem Heap reservieren muss nach dem Datentyp die Anzahl der Elemente in eckigen Klammern angegeben werden. Beim Freigeben des Feldes ist darauf zu achten, dass zwischen „delete“ und der Zeigervariablen die eckigen Klammern ohne weitere Angaben stehen. Werden die Klammern ausversehen vergessen wird nur das erste Element freigeben und nach langer Laufzeit des Programms kann es dazu führen, dass der gesamte Prozessspeicher aufgebraucht ist und das Programm abstürzt.

```
// Reserve multiple memory (array)
double* dynArray = new double[10];
// Free all memory
delete [] dynArray;
```

Globaler Speicher

Globaler Speicher ist Speicher, der außerhalb irgendeiner Funktion oder Klasse erstellt wird. Dieser Speicher ist also mit dem Programmstart vorhanden und muss niemals freigeben werden. Der Speicherverbrauch von globalem Speicher ist damit fix über die gesamte Programmlaufzeit.

Technisch kann der globale Speicher entweder als erstes auf den Stapel gelegt werden oder wird gemeinsam mit dem Programmcode im virtuellen Prozessraum reserviert.

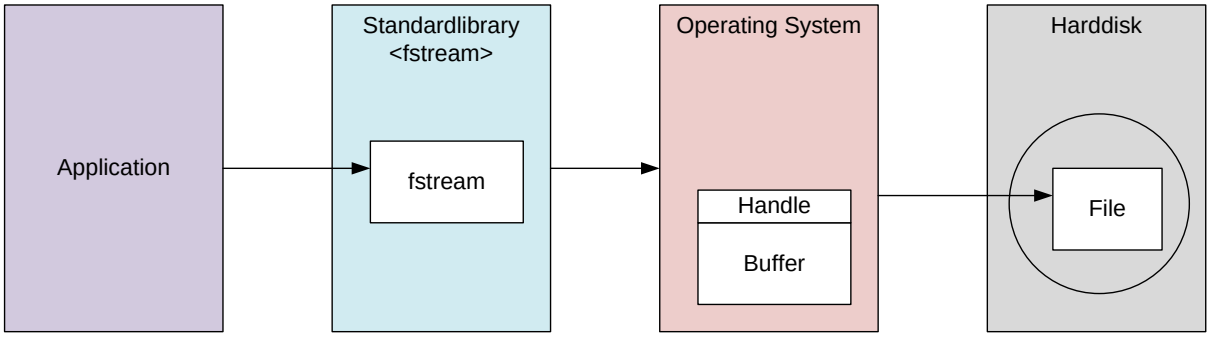
Lokaler Speicher

Lokaler Speicher wird innerhalb der Funktionen und Methoden auf dem Stapelspeicher angelegt. Daraus folgt, dass der Speicher erst dann reserviert wird, wenn eine Funktion oder Methode aufgerufen wird. Wird die Funktion, bzw. Methode verlassen, dann wird auch der Speicher wieder freigegen. Anlog verhält es sich, wenn eine Variable in einer Blockanweisung definiert wurde. Sobald der Block verlassen wird, wird auch die Variable wieder vom Stapel gelöscht.

Lektion 8

Dateien

Dateien (engl. Files) sind Daten die auf Speichermedien geschrieben bzw. gelesen werden können. Das Betriebssystem stellt Funktionen bereit um den Zugriff auf Dateien zu realisieren. Weiterhin gehört eine Standardbibliothek zum Sprachumfang von C++ um mit Dateien zu arbeiten.



Dateien im Betriebssystem

Es ist Sache des Betriebssystems wie es den Zugriff auf die Dateien regelt. Für gewöhnlich versucht das Betriebssystem über den Festplattentreiber mit der Festplatte zu kommunizieren und damit die Daten in den Hauptspeicher zu laden. Zur Identifikation der Datei wird vom Betriebssystem ein sogenanntes Handle vergeben mit dem Aktionen wie Lesen und Schreiben ausgeführt werden müssen. Der Zwischenspeicher (engl. Buffer) wird verwendet, um die von der Festplatte gelesenen bzw. auf die Festplatte zu schreibenden Daten zwischen zu speichern. D.h. die Daten sind nicht unbedingt alle auf einmal vorhanden und werden auch nicht unbedingt sofort weggeschrieben.

Das Betriebssystem stellt Funktionen bereit, um mit der Datei arbeiten zu können. Dabei sind folgende Aktionen üblich:

Aktion	***Beschreibung***
Datei öffnen	Das Betriebssystem nimmt Verbindung mit der Festplatte auf um den Datenaustausch zu ermöglichen. Beim Öffen muss mit angegeben werden, ob zum Lesen oder zum Schreiben der Datei. Beim Schreiben kann noch angegeben werden, ob am Ende einer bestehenden Datei weiter geschrieben werden soll oder ob der aktuelle Inhalt verworfen wird.
Datei lesen	Daten in den Hauptspeicher einlesen.
Datei schreiben	Daten in die Datei, bzw. den Zwischenspeicher schreiben
Lese oder Schreibposition	Da nicht immer alle Daten sofort gelesen werden, sondern immer

ändern.	Stück für Stück, ist ein Positionszeiger vorhanden, der anzeigt an welcher Stelle in der Datei weiter gelesen werden soll.
Zwischenspeicher leeren	Sicherstellen, dass der Zwischenspeicher in die Datei übernommen wurde. Das nicht immer alles sofort geschrieben wird ist eine Geschwindigkeitsoptimierung. Da bei jedem Schreibvorgang die Festplatte erst wieder die genaue Position anfahren muss (was relativ lange dauert) wird versucht möglichst viele Daten auf einmal zu schreiben.
Datei schließen	Die Verbindung zur Festplatte wird nicht mehr gebraucht und aufgehoben. Ab jetzt kann keine weitere Interaktion mehr mit der Datei unter dem gegebenem Handle erfolgen.

Tabelle 4:Dateiaktionen eines Betriebssystems

Dateien mit der Standardbibliothek „fstream“

Die Funktionen um auf eine Datei zuzugreifen hängen vom verwendeten Betriebssystem ab. Zwar gibt es auch hier Standards, z.B. der Posix Standard, aber nicht alle Betriebssysteme halten sich daran. Der C/C++ Standard hat deshalb eine Standardbibliothek, um einheitlich auf Dateien zugreifen zu können. Je nachdem wie das Zielsystem des C/C++ Programms ist, ruft die Standardbibliotheksfunktion die korrekten Funktionen des Betriebssystems auf.

Für C++ kann die Standardbibliothek „fstream“ verwendet werden. Das Verhalten ist ähnlich dem Arbeiten mit der Konsole, da der Datenzugriff ebenfalls als Datenstrom realisiert ist.

Die Standardbibliothek enthält eine Klasse „fstream“ die für den Dateizugriff verwendet werden kann. Dazu instanziiert man ein Objekt dieser Klasse und ruft die entsprechenden Methoden auf. Da es sich um ein Datenstromobjekt handelt, werden dafür dann auch die Datenstromoperatoren („<<“ und „>>“) bereitgestellt.

Beispiel:

```
//------------------------------------------------------------------------------
// main.cpp
//
// writing a text to a file
//
// Date: 17.05.2011
// Author: Thomas Braun
// Email: thomas.braun@fh-aachen.de
//
//------------------------------------------------------------------------------

#include <iostream>
#include <fstream>

using namespace std;

void main()
{
          // Create a file stream object
          fstream myFile;

          // open the file only for writing
          myFile.open("D:\\test.txt", ios::out);

          // write some text into the file
          myFile << "Hello World" << endl;

          // Close the file
          myFile.close();


          system("PAUSE");
```

}

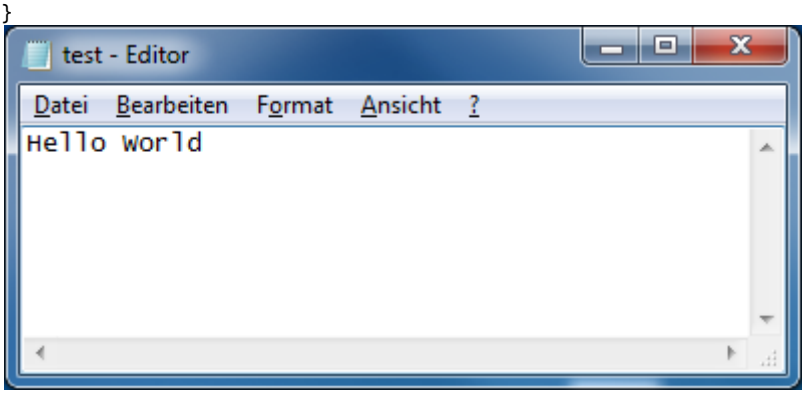


Abbildung 1: Erstellte Textdatei Text.txt

Beispiel:

```
//------------------------------------------------------------------------
// main.cpp
//
// reading a text from a file
//
// Date: 17.05.2011
// Author: Thomas Braun
// Email: thomas.braun@fh-aachen.de
//
//------------------------------------------------------------------------

#include <iostream>
#include <fstream>

using namespace std;

void main()
{
          // Create a file stream object
          fstream myFile;

          // Buffer for reading file content
          char text[255];

          // open the file only for reading
          myFile.open("D:\\test.txt", ios::in);

          // read some text from the file
          myFile >> text;

          // give out to the console the text read
          cout << text << endl;

          // Close the file
          myFile.close();


          system("PAUSE");
}
```

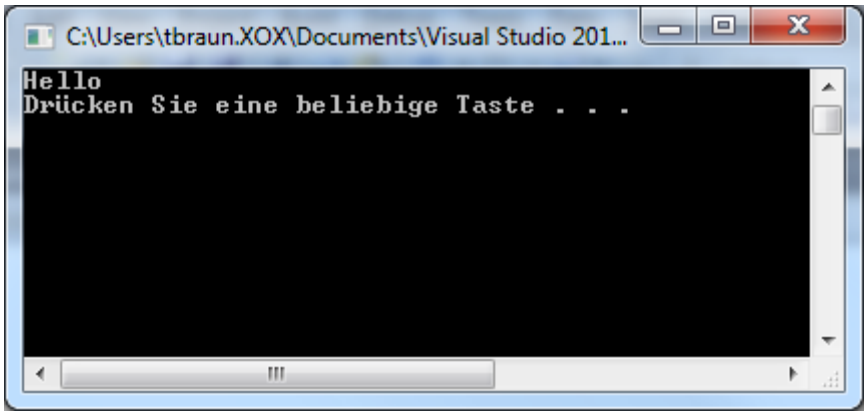


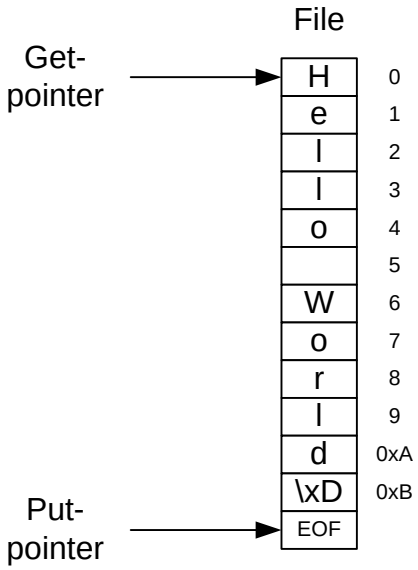
Abbildung 2: Ausgabe der gelesenen Datei auf der Konsole

Wie bei cin liest der Stream-Operator bis zu einem Trennzeichen ein. Standard sind hier die sogenannten White-Characters (Leerzeichen, Tab, Zeilenumbruch). Auf diese Weise kann eine Datei Elementweise eingelesen werden. Es gibt aber noch weitere Methoden von fstream um mit einer Datei zu arbeiten.

Aktion	**Methode**	**Argumente**	**Argumentbeschreibung**
Datei öffnen	open(path,mode)	**mode**	
		ios::in	zum Lesen öffnen
		ios::out	zum Schreiben öffnen
		ios::in|ios::out	zum Lesen und Schreiben öffnen
		ios::binary	Datei wird als binäre Datei geöffnet.
		ios::ate	Öffnet die Datei und setzt den Dateizeiger ans Ende
		ios::app	Öffnet die Datei und Schreiboperationen fügen ans Ende der Datei an.
		ios::trunc	Öffnet die Datei und verwirft den aktuellen Inhalt.
Datei schließen	close()		
Datei lesen	getline(string,size)		
	operator >>		
	read(buffer, size)		
Datei schreiben	operator <<		
	write(buffer, size)		
Position des Lesezeiger setzen	seekg(position)		
Position des Schreibzeigers setzen	seekp(position)		
Position des Lesezeigers relativ setzen	seekg(offset,direction)		
Position des Schreibzeigers relativ setzen	seekp(offset,direction)	**direction**	
		ios:beg	vom Anfang der Datei
		ios::cur	von der Aktuellen Position des Zeigers
		ios::end	vom Ende der Datei

Position des Lesezeigers	tellg()		
Position des Schreibzeigers	tellp()		
Fehler beim Formatieren des gelesenen Stroms	fail()		
Fehler beim Lesen oder Schreiben	bad()		
Ende der Datei ist erreicht.	eof()		
Es ist kein Fehler aufgetreten.	good()		

Hier ist nochmal eine schematische Darstellung einer Datei. Der Lesezeiger zeigt auf den Anfang der Datei und der Schreibzeiger ist zeigt auf das Ende.



Die Kennung EOF bedeuted End of File und zeigt das es keine weiteren Daten mehr zu lesen gibt.

Die Unterscheidung zwischen Textdateien und Binärdateien ist, dass beim Lesen und Schreiben von Textdateien immer davon ausgegangen wird, dass die Datei nur Strings enthält. Die Binäre Datei ist hingegen eine Sammlung von Bytes die auch noch verschieden interpretiert werden kann. Z.B. können die ersten 4 Byte für eine Integer Zahl stehen und die nachfolgenden 8 Bytes für eine Double Zahl. Die genau Bedeutung wird als Format einer Binärdatei bezeichnet und muss beim Programmieren bekannt sein. Es gibt dann eine Vereinbarung wie eine solche Datei zu lesen, bzw. zu interpretieren ist.

Textdatei

Der Inhalt einer Textdatei wird als Strings also Zeichenketten interpretiert. Diese Textdateien werden Trennzeichenorientiert eingelesen. Trennzeichen können Leerzeichen, Semikolon, Komma, Tabs oder Zeilenumbrüche sein. Ist das Trennzeichen der Zeilenumbruch, dann spricht man auch vom zeilenweisen Lesen einer Textdatei.

Typische Textdateien sind einfache Texte die mit Notepad erzeugt werden oder CSV Dateien um Daten abzulegen.

Um Messwerte abzulegen wird häufig das CSV Format verwendet, da viele Programme dieses Format wieder einlesen können, um die Daten weiter zu verarbeiten. Excel ist ein typischer Vertreter dafür.

Als Trennzeichen in einer CSV Datei wollen wir hier das Semikolon und den Zeilenumbruch festlegen.

Um zum Beispiel durch einen Zufallsgenerator erzeugt Zahlen in eine Datei wegzuschreiben kann man folgendes Programm schreiben.

Beispiel:

```
//---------------------------------------------------------------------------
// main.cpp
//
// Sample Write CSV file
//
// Date: 20.05.2011
// Author: Thomas Braun
// Email: thomas.braun2@fh-aachen.de
//
//---------------------------------------------------------------------------

#include <iostream>
#include <fstream>


using namespace std;

void main()
{
          // Number of lines we want to generate
          const int cMaxRandomValues = 10;

          // Instatiate a file stream object
          fstream CsvFile;

          // Create/open it for writing, filename is D:\test.csv
          // you can find it there on harddisc
          CsvFile.open("D:\\test.csv",ios::out);

          // Write a first line for header information
          CsvFile << "x-value" << ";" << "y-value" endl;

          // create and write the lines
          for (int i=0;i<cMaxRandomValues; i++)
          {
                    // The function rand() creates pseudo random numbers
                    // between 0 and RAND_MAX
                    CsvFile << rand() << ";" << rand() << endl;
          }

          // close the file
          CsvFile.close();
}
```

Werte die in einer Textdatei stehen können ähnlich wieder eingelesen werden.

Beispiel:

```
//---------------------------------------------------------------------------
// main.cpp
//
// Sample read a CSV file
//
// Date: 20.05.2011
```

```
// Author: Thomas Braun
// Email: thomas.braun2@fh-aachen.de
//
//------------------------------------------------------------------------

#include <iostream>
#include <fstream>


using namespace std;

void main()
{
          fstream DataFile;

          DataFile.open("D:\data.txt",ios::in);

          double data = 0.0;
          double sum = 0.0;

          while (!DataFile.eof() )
          {
                    DataFile >> data;

                    sum += data;
          }

          cout << "Sum: " << sum << endl;

          system("PAUSE");
}
```

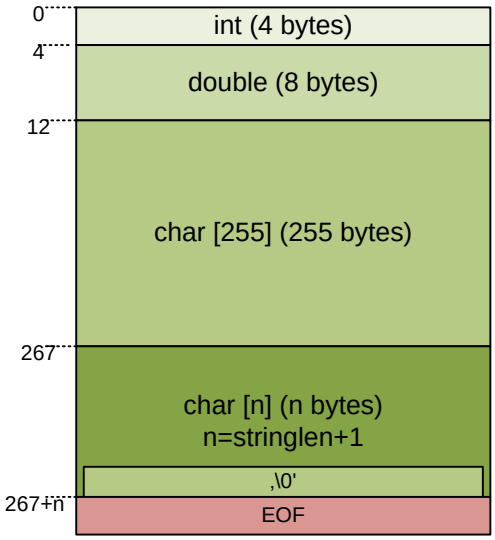
Binärdateien

In Binärdateien werden Zahlenwerte nicht in einen String konvertiert, sondern direkt die binäre Form abgespeichert. Damit wird eine Zahl vom Datentyp „int“ mit 4 Bytes auf der Festplatte gespeichert und entsprechend der Datentyp „double“ mit 8 Bytes.

Zeichenketten können ebenfalls in eine Binärdatei gespeichert werden, sie werden dann aber als eine Folge von ASCII Codes gespeichert und das Programm ist dann für die richtige Behandlung zuständig. So ist z.B. darauf zu achten, dass die terminierende Null gespeichert wird. Beim Lesen ist es die Aufgabe des Programms die Länge einer Zeichenkette zu ermitteln, bzw. kann es durch das Format der Datei definiert sein.

Wenn für Zeichenketten eine feste Anzahl in der Datei definiert ist, dann kann die terminierende Null beim Schreiben auch weggelassen werden und muss dann beim Lesen wieder hinzugefügt werden.

Hier ein Beispiel, wie binäre Daten gespeichert und gelesen werden können. Das Format der Binärdatei schreibt erste ein 4 Byte großes „int“ Datum, danach 8 Bytes für einen „double“-Wert und dann 255 Zeichen für eine Zeichenkette und der Rest dann für eine weitere Zeichenkette vor. Für die erste Zeichenkette werden immer 255 Zeichen gespeichert, auch wenn die Zeichenkette kürzer ist. Die zweite Zeichenkette wird nur in der benötigten Länge inklusive der terminierenden Null gespeichert.



Eine Funktion schreibt eine solche Datei und „D:\\test.bin“ auf der Festplatte ab. Die andere Funktion liest die Datei wieder ein.

Beispiel:

```
//---------------------------------------------------------------------------
// main.cpp
//
// Sample for binary files
//
// Author: Thomas Braun
// Date: 21.04.2012
// Email: thomas.braun2@fh-aachen.de
//---------------------------------------------------------------------------

#include <iostream>
#include <fstream>
#include <string.h>

using namespace std;

//---------------------------------------------------------------------------
// example shows how to write binary data to a file
//---------------------------------------------------------------------------
void WriteBinaryData()
{
        int intdata = 12;

        double doubledata = 134.1;

        char stringarray[255] = "HelloWorld";

        const char* ptrString = "This is string with a pointer";

        // Instantiate a file stream object
        fstream binFile;

        // open file stream for binary writing. Delete content if file already exists.
        binFile.open("D:\\test.bin", ios::out|ios::binary|ios::trunc);

        //write the integer data (4 bytes)
        binFile.write((const char*)&intdata,sizeof(intdata));

        //write the double data (8 bytes)
        binFile.write((const char*)&doubledata,sizeof(doubledata));

        // write a string array (255 Bytes, because sizeof returns the complete array)
        binFile.write(stringarray,sizeof(stringarray));

        // write a string hold by a string pointer (29 bytes for the text and 1 byte for the
terminating 0)
```

```
        binFile.write(ptrString,strlen(ptrString)+1);

        //close file stream
        binFile.close();
}

//------------------------------------------------------------------------------
// funktions read the data that was written before
// It is import to read it in the same order as it was written.
//------------------------------------------------------------------------------
void ReadBinaryData()
{
        int intdata = 0;
        double doubledata = 0.0;
        char stringarray[255] = {0};

        // you have to reserve enough memory for your string. So I take an array with 255 bytes.
        char ptrString[255] = {0};

        // Instantiate a file stream object
        fstream binFile;

        // open file stream for binary writing. Delete content if file already exists.
        binFile.open("D:\\test.bin", ios::in|ios::binary);

        // read the integer data, size is 4 bytes
        binFile.read((char*)&intdata,sizeof(intdata));

        // read the double data, size is 8 bytes
        binFile.read((char*)&doubledata, sizeof(doubledata));

        // read the string array (255 bytes)
        binFile.read(stringarray,sizeof(stringarray));

        // read the pointer string. You have to read until you reach the terminating zero!!!
        // When you use a loop you should always check for enf of file!!
        int i=-1;
        do
        {
                i++;
                // read a character
                binFile.read(&ptrString[i],sizeof(char));
                // until \0 is found or end of file is reached or the string buffer is full
        } while (ptrString[i]!= 0 && !binFile.eof() && i<255);


        // Printout to show
        cout << "Integer data: " << intdata << endl;
        cout << "Double data: " << doubledata << endl;
        cout << "String array: " << stringarray << endl;
        cout << "Ptr String: " << ptrString << endl;

        // close file stream
        binFile.close();
}


//------------------------------------------------------------------------------
// main function calls the examples for reading and writing binary files.
//------------------------------------------------------------------------------
void main()
{

        WriteBinaryData();
        ReadBinaryData();

        system("PAUSE");

}
```

Die folgende Funktion liest nur den „double“-Wert aus der Datei aus, in dem mit der Methode „fstream::seekg(pos)“ direkt zur Dateiposition gesprungen wird und erst dann gelesen wird.

Beispiel:

```
//--------------------------------------------------------------------------
// function read only the double data that was written before
//--------------------------------------------------------------------------
void ReadOnlyPartOfBinaryData()
{
        double doubledata = 0.0;

        // Instantiate a file stream object
        fstream binFile;

        // open file stream for binary writing. Delete content if file already exists.
        binFile.open("D:\\test.bin", ios::in|ios::binary);

        // set the files read-pointer to position 4 (because intdata was before with 4 bytes)
        binFile.seekg(4);

        // read the double data, size is 8 bytes
        binFile.read((char*)&doubledata, sizeof(doubledata));

        // Printout to show
        cout << "Double data: " << doubledata << endl;

        //close file stream
        binFile.close();
}
```

Es ist bei binären Formaten üblich eine Kennung am Anfang zu schreiben, um beim Lesen prüfen zu können, dass die Datei im richtigen Format ist. Diese Prüfung kann zwar zufällig erfolgreich sein, wenn ein anderes Format zufällig mit der gleichen Folge anfängt, gibt aber schon mal eine erste Sicherheit. Als Kennung werden gerne ASCI-Code Folgen verwendet, die etwas mit dem Formatname zu tun haben. Weitere Sicherungsmöglichkeiten ist die Dateilänge und eine Checksumme zu Beginn oder direkt ans Ende zu schreiben. Beim Lesen werden die Kennung und die abgespeicherte Dateigröße geprüft. Wenn die Nutzdaten gelesen wurden kann auch die Checksumme geprüft werden.

Lektion 9

Klassen

Klassen beschreiben den Aufbau von Objekten. Sie sind vergleichbar mit den bereits bekannten Strukturen, nur dass Klassen, bzw. Objekte aus Daten und Funktionen bestehen. Dabei werden die Daten als Attribute oder Member-Variablen und die Funktionen als Methoden bezeichnet. Die neuen Bezeichner sollen auf ein anderes Denkparadigma hinweisen. Während beim strukturierten Programmieren die Aufgabenstellung nach Funktionalitäten untersucht wird, wird beim objekt-orientierten Programmieren nach Objekten gesucht, die Attribute und Methoden haben. Die Methoden dienen dazu die Aufgaben des Objektes auszuführen oder die Attribute des Objektes zu manipulieren.

Die Objekte müssen beim objekt-orientierten Programmieren nicht physikalische Objekte sein. Es werden häufig auch logische Elemente als Objekte gesucht. In unsere Aufgabe sind solche Objekte zum Beispiel zu erstellenden Figuren oder eine Befehlsliste. Es ist die Arbeit eines Software-Designers solche Objekte zu identifizieren und deren Attribute und Methoden festzulegen.

Typischerweise wird bei einer Klasse zwei Dateien verwendet. Eine Headerdatei (*.h) für die Deklaration und ein Moduldatei (*.cpp) für die Definition.

Die Deklaration einer Klasse beginnt mit dem Schlüsselwort „class“ gefolgt von dem frei wählbaren Namen der Klassen. In geschweiften Klammern wird dann aufgelistet aus welchen Elementen die Klasse besteht. Für gewöhnlich werden erst die Attribute und dann die Methoden aufgelistet. Die Klassendeklaration endet mit einem Semikolon.

Die Sichtbarkeit, bzw. Zugreifbarkeit der Klassenelemente wird mit den Schlüsselwörtern

- ***public***, jeder darf die folgenden Elemente zugreifen
- ***protected***, die Elemente der Klasse und davon abgeleitete Klassen dürfen auf die folgenden Elemente zugreifen
- ***privat***, nur die Elemente der Klasse selbst dürfen auf die folgenden Elemente zugreifen.

geregelt. Das Schlüsselwort wird gefolgt von einem Doppelpunkt in der Elementliste verwendet. Die Sichtbarkeit gilt für Elemente die nach dieser Zeile stehen bis eine neue Anweisung für die Sichtbarkeit angeben wird. Die Standardsichtbarkeit für Klassen ist „privat“.

Beispiel 01

```
//---------------------------------------------------------------------------
// Shape.h
//
// Simple Shape Class - Declaration
//
// Date: 08.05.2011
// Author: Thomas Braun
// Email: thomas.braun2@fh-aachen.de
//---------------------------------------------------------------------------

#pragma once

enum EShapeType
{
        eInvalid=0,
        ePoint=1,
        eLine=2,
        eCircle=3
};

class CShape
{
protected:
        EShapeType m_type;

        double m_x1,m_y1;
        double m_x2,m_y2;
        double m_r;

public:
        void SetParameter(EShapeType type, double x1, double y1, double x2, double y2, double r);
        void Print();
};
```

Die Definition der Klassen erfolgt normalerweise außerhalb der Deklaration und wird in einer eigenen Datei gespeichert. Um den Bezug zur Deklaration herzustellen muss die zugehörige Headerdatei als Include-Anweisung mit angegeben werden. Die Methoden haben vor dem Namen noch den Namen der Klasse gefolgt von zwei Doppelpunkten vorangestellt. Damit wird klar, dass die Methode keine allgemeine Funktion ist, sondern zu der Klasse gehört.

```
//---------------------------------------------------------------------------
// Shape.cpp
//
// Simple Shape Class - Definition
//
// Date: 08.05.2011
```

```
// Author: Thomas Braun
// Email: thomas.braun2@fh-aachen.de
//------------------------------------------------------------------------

#pragma once
#include "Shape.h"
#include <iostream>

using namespace std;


void CShape::SetParameter(EShapeType type, double x1, double y1, double x2, double y2, double r)
{
        m_x1 = x1;
        m_y1 = y1;
        m_x2 = x2;
        m_y2 = y2;
        m_r = r;
        m_type = EShapeType::eLine;
}


void CShape::Print()
{
        switch (m_type)
        {
        case ePoint:
                cout << "point " << ";" << m_x1 << ";" << m_y1 << endl;
                break;
        case eLine:
                cout << "line " << ";" << m_x1 << ";" << m_y1 << ";" << m_x2 << ";" << m_y2 << endl;
                break;
        case eCircle:
                cout << "circle " << ";" << m_x1 << ";" << m_y1 << ";" <<  m_r << endl;
                break;
        default:
                cout << "error" << endl;
                return;
        }
}
```

Objekte

Objekte sind die konkreten Instanzen von Klassen. Klassen beschreiben nur wie das Objekt aufgebaut ist, also ein Art Typbeschreibung. Die Objekte sind dann das womit das Programm arbeitet. Eine Klasse „Figur“ beschreibt, dass es Parameter für die Figur geben muss und eine Methode zum Ausgeben des zugehörigen CNC-Befehls hat. Das Objekt „myShape“ verbraucht dann Speicher, um die zugehörigen Koordinaten und den Radius für den Kreis zu speichern. Es können mehrere Objekte von einer Klasse erstellt werden, aber jedes Objekt kann nur von einem Typ sein.

Das Instanziieren eines Objektes ist gleich dem Erstellen einer Variablen. Zuerst wird der Name der Klasse (wie der Datentyp) und dann der Name der Objektinstanz angegeben.

```
//------------------------------------------------------------------------
// main.cpp
//
// shapes in a class
//
// Date: 04.05.2011
// Author: Thomas Braun
// Email: thomas.braun2@fh-aachen.de
//
//------------------------------------------------------------------------

#include "Shape.h"
#include <iostream>

using namespace std;
```

```
void main()
{
        // create instance
        CShape myPoint;
        CShape myLine;
        CShape myCircle;

        // make a point with x1=2.1 y1=4.3
        myPoint.SetParameter(ePoint, 2.1, 4.3, 0,0,0);
        // make a line with x1=3.7 y1=5.6 x2=8.9 and y2==21.5
        myLine.SetParameter(eLine, 3.7, 5.6, 8.9, 21.5, 0);
        // make a circle with x1=6.3 y1=11.6 and r=4.1
        myCircle.SetParameter(eCircle, 6.3, 11.6, 0,0, 21.5);

        // Print out to console
        myPoint.Print();
        myLine.Print();
        myCircle.Print();

        system("PAUSE");
}
```

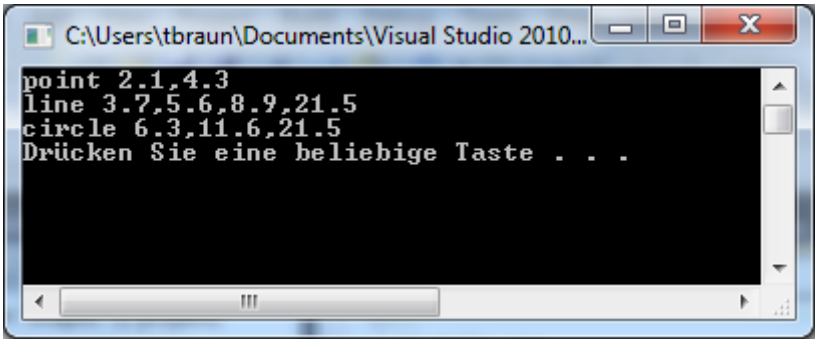


Abbildung 3: Ausgabe des Shape Beispiels

Konstruktor

Der Konstruktor einer Klasse ist eine besondere Methode der Klasse, die beim Instanziieren eines Objektes aufgerufen wird. Hier hat man die Möglichkeit die Objektinstanz zu initialisieren. Es können unterschiedliche Konstruktoren erstellt werden, die jeweils andere Parameter haben.

Der Konstruktor hat den gleichen Namen wie die Klasse, hat aber keinen Rückgabewert, auch nicht „void“.

Beispiel 02

```
//---------------------------------------------------------------------------
// Shape.h
//
// Simple Shape Class - Declaration
//
// Date: 08.05.2011
// Author: Thomas Braun
// Email: thomas.braun2@fh-aachen.de
//---------------------------------------------------------------------------

#pragma once

enum EShapeType
{
        eInvalid=0,
        ePoint=1,
        eLine=2,
        eCircle=3
};

class CShape
{
```

```
protected:
        EShapeType m_type;

        double m_x1,m_y1;
        double m_x2,m_y2;
        double m_r;

public:
        // Default-Construktor
        CShape(void);
        // Constructor for a point
        CShape(double x1, double y1);
        // Constructor for a line
        CShape(double x1, double y1, double x2, double y2);
        // Constructor for a circle
        CShape(double x1, double y1, double r);

        void SetParameter(EShapeType type, double x1, double y1, double x2, double y2, double r);
        void Print();
};
```

Hier ist jetzt noch die Definition für die Konstruktoren.

```
//----------------------------------------------------------------------------
// Shape.cpp
//
// Simple Shape Class - Definition
//
// Date: 08.05.2011
// Author: Thomas Braun
// Email: thomas.braun2@fh-aachen.de
//----------------------------------------------------------------------------

#pragma once
#include "Shape.h"
#include <iostream>

using namespace std;

//----------------------------------------------------------------------------
// Default-Constructor
//----------------------------------------------------------------------------
CShape::CShape(void)
{
        m_type = EShapeType::eInvalid;
}

//----------------------------------------------------------------------------
// Constructor for a point
//----------------------------------------------------------------------------
CShape::CShape(double x1, double y1)
{
        m_x1 = x1;
        m_y1 = y1;
        m_type = EShapeType::ePoint;
}

//----------------------------------------------------------------------------
// Constructor for a line
//----------------------------------------------------------------------------
CShape::CShape(double x1, double y1, double x2, double y2)
{
        m_x1 = x1;
        m_y1 = y1;
        m_x2 = x2;
        m_y2 = y2;
        m_type = EShapeType::eLine;
}

//----------------------------------------------------------------------------
// Constructor for a circle
//----------------------------------------------------------------------------
CShape::CShape(double x1, double y1, double r)
```

```
{
        m_x1 = x1;
        m_y1 = y1;
        m_r = r;
        m_type = EShapeType::eCircle;
}

//---------------------------------------------------------------------------
// Methode setting a shape
//---------------------------------------------------------------------------
void CShape::SetParameter(EShapeType type, double x1, double y1, double x2, double y2, double r)
{
        m_x1 = x1;
        m_y1 = y1;
        m_x2 = x2;
        m_y2 = y2;
        m_r = r;
        m_type = type;
}


//---------------------------------------------------------------------------
// Methode print out for the shape command
//---------------------------------------------------------------------------
void CShape::Print()
{
        switch (m_type)
        {
        case ePoint:
                cout << "point " << ";" << m_x1 << ";" << m_y1 << endl;
                break;
        case eLine:
                cout << "line " << ";" << m_x1 << ";" << m_y1 << "," << m_x2 << "," << m_y2 << endl;
                break;
        case eCircle:
                cout << "circle " << ";" << m_x1 << ";" << m_y1 << ";" <<  m_r << endl;
                break;
        default:
                cout << "error" << endl;
                return;
        }
}



//---------------------------------------------------------------------------
// main.cpp
//
// shapes in a class
//
// Date: 04.05.2011
// Author: Thomas Braun
// Email: thomas.braun2@fh-aachen.de
//
//---------------------------------------------------------------------------

#include "Shape.h"
#include <iostream>

using namespace std;

void main()
{
        // Create instances for point, line and circle
        CShape myPoint(2.1,4.3);
        CShape myLine(3.7,5.6, 8.9, 21.5);
        CShape myCircle(6.3, 11.6, 4.1);

        // Print out the
        myPoint.Print();
        myLine.Print();
        myCircle.Print();

        system("PAUSE");
}
```

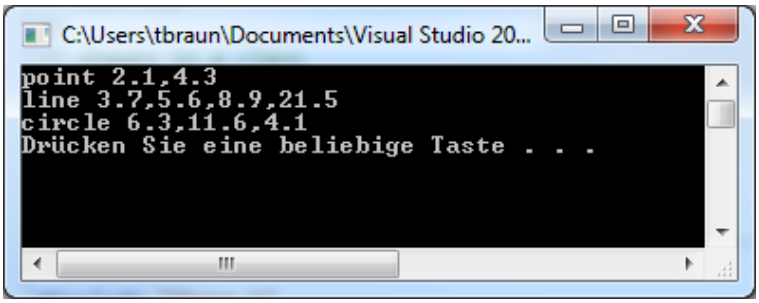


Abbildung 4: Ausgabe Beispiel 02

Destruktor

Der Destruktor wird aufgerufen, bevor ein Objekt zerstört wird. Der Destruktor wird ähnlich erstellt wie ein Konstruktor, nur dass er keine Parameter hat und noch eine Tilde (~) vorangestellt hat.

```
class CClass
{
private:
        int m_attrib;

public:
        // Default-Constructor
        CClass();
        // Constructor initialising attrib
        CClass(int attrib);

        // Destructor
        ~CClass();

        // public method
        void publicMethod();
};
```

In unserer Beispielanwendung können wir die Befehlsliste auch als Objekt implementieren. Dazu müssen wir eine Klasse „CShapeContainer“ erstellen. Die Klasse hat eine Zeigerliste auf „CShapePoint“ Objekte und die Position der nächsten freien Listenposition als auch die maximale Anzahl an Elementen. Außerdem muss man der Liste Elemente hinzufügen können und wieder löschen können.

Beim Erzeugen der Liste soll das Zeigerfeld mit NULL-Zeigern initialisiert werden. Beim Zerstören sollen alle Listenelemente ebenfalls zerstört werden.

```
//---------------------------------------------------------------------------
// ShapeContainer.h
//
// Declaration of a Shape list
//
// Date: 08.05.2011
// Author: Thomas Braun
// Email: thomas.braun2@fh-aachen.de
//---------------------------------------------------------------------------
#pragma once

#include "ShapePoint.h"

class CShapeContainer
{
        static const int cShapeContainerSize = 10;
        CShapePoint* m_shapeList[cShapeContainerSize];
        int m_lastShapeListPosition;

public:
        CShapeContainer(void);
```

```
        ~CShapeContainer(void);

        void AddShape(CShapePoint* ptrShape);
        void DeleteLastShape();
        void Print();
};

//---------------------------------------------------------------------------
// ShapeContainer.cpp
//
// Definition of a Shape list
//
// Date: 08.05.2011
// Author: Thomas Braun
// Email: thomas.braun2@fh-aachen.de
//---------------------------------------------------------------------------

#include "ShapeContainer.h"
#include <iostream>

using namespace std;

//---------------------------------------------------------------------------
// Constructor - Initialize the pointer array with NULL
//---------------------------------------------------------------------------
CShapeContainer::CShapeContainer(void)
{
        m_lastShapeListPosition = 0;
        for (int i = 0; i< cShapeContainerSize; i++)
        {
                m_shapeList[i] = NULL;
        }
}

//---------------------------------------------------------------------------
// Destructor - destroys all stored objects.
//---------------------------------------------------------------------------
CShapeContainer::~CShapeContainer(void)
{
        while (m_lastShapeListPosition)
        {
                m_lastShapeListPosition--;
                delete m_shapeList[m_lastShapeListPosition];
                m_shapeList[m_lastShapeListPosition] = NULL;
        }
}

//---------------------------------------------------------------------------
// Add a shape to the list. The object is created outside.
//---------------------------------------------------------------------------
void CShapeContainer::AddShape(CShapePoint* ptrShape)
{
        if (m_lastShapeListPosition < cShapeContainerSize)
        {
                m_shapeList[m_lastShapeListPosition] = ptrShape;
                m_lastShapeListPosition++;
        }
}

//---------------------------------------------------------------------------
// Delete the last shape in the list
//---------------------------------------------------------------------------
void CShapeContainer::DeleteLastShape()
{
        if (m_lastShapeListPosition > 0)
        {
                m_lastShapeListPosition--;
                delete m_shapeList[m_lastShapeListPosition];
                m_shapeList[m_lastShapeListPosition] = NULL;
        }
}

//---------------------------------------------------------------------------
// Printout all CNC Commands to the console
//---------------------------------------------------------------------------
void CShapeContainer::Print()
```

```
{
        for (int i = 0; i< m_lastShapeListPosition; i++)
        {
                m_shapeList[i]->Print();
        }
        cout << endl;
}
```

Wir können mit „new“ direkt Objekte erzeugen und der Methode „Add“ übergeben. Der Type-Case ist nicht unbedingt anzugeben, da der Compiler das auch implizit macht.

```
//---------------------------------------------------------------------------
// main.cpp
//
// shapes in a class
//
// Date: 04.05.2011
// Author: Thomas Braun
// Email: thomas.braun2@fh-aachen.de
//
//---------------------------------------------------------------------------

#include "ShapePoint.h"
#include "ShapeLine.h"
#include "ShapeCircle.h"
#include "ShapeContainer.h"
#include <iostream>

using namespace std;

void main()
{
        // create a list object
        CShapeContainer shapes;

        // now fill the list

        // a point with x1=2.1 y1=4.3
        shapes.AddShape( new CShapePoint(2.1, 4.3) );
        // a line with x1=3.7 y1=5.6 x2=8.9 and y2==21.5
        shapes.AddShape( (CShapePoint*)new CShapeLine(3.7, 5.6, 8.9, 21.5) );
        // a circle with x1=6.3 y1=11.6 and r=4.1
        shapes.AddShape( (CShapePoint*)new CShapeCircle(6.3, 11.6, 21.5) );

        // Printout each element to the console
        shapes.Print();

        system("PAUSE");
}
```

Lektion 10

Vererbung

Bei der Vererbung übernimmt eine erbende Klasse die Attribute und Methoden der Basisklasse. Wird also ein Objekt vom Typ einer Abgeleiteten Klasse instanziiert, dann können auch die public-Methoden der Basisklasse aufgerufen werden. Man hat damit die Möglichkeit ein Klasse zu erweitern oder nur punktuell das Verhalten zu verändern.

In der Deklaration einer Klasse muss nach dem Klassennamen die Basisklasse angegeben werden. Die Sichtbarkeit der Basisklasse ist hier mit „public“ gewählt. Die Sichtbarkeit bei der Ableitung bedeutet, dass alle geerbten Attribute und Methoden maximal die Sichtbarkeit haben können die hier angegeben wird. Im Falle von „public“ bedeutet das, dass sich nichts ändert. Würde mit der Sichtbarkeit von „protected“ abgeleitet, dann bedeutet das, dass alle „public“ Methoden und

Attribute der Basisklasse jetzt über die abgeleitete Klasse auch nur noch „protected“ zugreifbar sind. Methoden und Attribute mit einer geringeren Sichtbarkeit werden nicht beeinflusst.

Gehen sie in Zukunft davon aus, dass wir die Ableitung immer mit der Sichtbarkeit „public“ wählen.

```
//---------------------------------------------------------------------------
// Declaration of a base class
//---------------------------------------------------------------------------
class CBaseClass
{
protected:
        int m_baseAttrib;

public:
        void publicBaseMethod() { // Do something }
};

//---------------------------------------------------------------------------
// Declaration of a derived class, inherit from CBaseClass
//---------------------------------------------------------------------------
class CDerivedClass : public CBaseClass
{
private:
        double m_derivedAttrib;

public:
        void publicDerivedMethod() { // Do something }
};

//---------------------------------------------------------------------------
// the main function using base and derived class
//---------------------------------------------------------------------------
void main()
{

        CBaseClass myBaseClass;
        myBaseClass.publicBaseMethod();

        CDerivedClass myDerivedClass;
        myDerivedClass.publicDerivedMethod();
        // method of base class can be called too
        myDerivedClass.publicBaseMethod();
}
```

Durch die Vererbung wird auch eine logische Beziehung hergestellt. Eine Möglichkeit ist es gemeinsame Eigenschaften von unterschiedlichen Objekttypen zusammenzufassen.

In unserem Beispiel können wir erkennen, dass alle Objekte mindestens die Koordinaten x1 und y1 haben. Außerdem musste für alle Typen der Typ als Enumeration angegeben werden. Der Typ kann aber auch durch die Klasse selbst ausgedrückt werden.

Wir wählen ein Design bei dem wir sagen alle Figuren basieren auf einem Punkt. Dann geht daraus direkt hervor, dass wir eine Basisklasse Punkt brauchen, die wir wie folgt Implementieren.

```
//---------------------------------------------------------------------------
// ShapePoint.h
//
// Declaration of a point class
//
// Date: 08.05.2011
// Author: Thomas Braun
// Email: thomas.braun2@fh-aachen.de
//---------------------------------------------------------------------------
#pragma once

class CShapePoint
{
```

```
protected:
        double m_x1,m_y1; // Coordinates of the point

public:
        CShapePoint();
        CShapePoint(double x1, double y1);

        void SetParameter(double x1, double y1);
        void Print();
};



//--------------------------------------------------------------------------
// ShapePoint.cpp
//
// Definition of a point class
//
// Date: 08.05.2011
// Author: Thomas Braun
// Email: thomas.braun2@fh-aachen.de
//--------------------------------------------------------------------------

#include "ShapePoint.h"
#include <iostream>

using namespace std;

//--------------------------------------------------------------------------
// Default-Construktor for a point
//--------------------------------------------------------------------------
CShapePoint::CShapePoint()
{
        m_x1 = 0.0;
        m_y1 = 0.0;
};

//--------------------------------------------------------------------------
// Construktor for a point
//--------------------------------------------------------------------------
CShapePoint::CShapePoint(double x1, double y1)
{
        m_x1 = x1;
        m_y1 = y1;
}

//--------------------------------------------------------------------------
// set parameter for a point
//--------------------------------------------------------------------------
void CShapePoint::SetParameter(double x1, double y1)
{
        m_x1 = x1;
        m_y1 = y1;
}

//--------------------------------------------------------------------------
// printout CNC command
//--------------------------------------------------------------------------
void CShapePoint::Print(void)
{
        cout << "point " << ";" << m_x1 << ";" << m_y1 << endl;
}
```

Die beiden anderen Figuren werden dann von CShapePoint abgeleitet.

```
//--------------------------------------------------------------------------
// ShapeLine.h
//
// Declaration of a point class
//
// Date: 08.05.2011
// Author: Thomas Braun
```

```
// Email: thomas.braun2@fh-aachen.de
//-----------------------------------------------------------------------------
#pragma once

#include "ShapePoint.h"

class CShapeLine : public CShapePoint
{
        // start-point is part of the base-class
        double m_x2,m_y2; // End-point of line

public:
        CShapeLine();
        CShapeLine(double x1, double y1, double x2, double y2);

        void SetParameter(double x1, double y1, double x2, double y2);
        void Print();
};



//-----------------------------------------------------------------------------
// ShapeLine.cpp
//
// Definition of a line class
//
// Date: 08.05.2011
// Author: Thomas Braun
// Email: thomas.braun2@fh-aachen.de
//-----------------------------------------------------------------------------

#include "ShapeLine.h"
#include <iostream>

using namespace std;

//-----------------------------------------------------------------------------
// Default-Constructor for a line
//-----------------------------------------------------------------------------
CShapeLine::CShapeLine()
{
        m_x1 = 0.0;
        m_y1 = 0.0;
        m_x2 = 0.0;
        m_y2 = 0.0;
};

//-----------------------------------------------------------------------------
// Construktor for a line
//-----------------------------------------------------------------------------
CShapeLine::CShapeLine(double x1, double y1, double x2, double y2)
{
        m_x1 = x1;
        m_y1 = y1;
        m_x2 = x2;
        m_y2 = y2;
}

//-----------------------------------------------------------------------------
// set parameter for a line
//-----------------------------------------------------------------------------
void CShapeLine::SetParameter(double x1, double y1, double x2, double y2)
{
        m_x1 = x1;
        m_y1 = y1;
        m_x2 = x2;
        m_y2 = y2;
}

//-----------------------------------------------------------------------------
// printout for CNC command
//-----------------------------------------------------------------------------
void CShapeLine::Print(void)
{
        cout << "line " << m_x1 << "," << m_y1 << "," << m_x2 << "," << m_y2 << endl;
}
```

```
//---------------------------------------------------------------------------
// ShapeCircle.h
//
// Declaration of a circle class
//
// Date: 08.05.2011
// Author: Thomas Braun
// Email: thomas.braun2@fh-aachen.de
//---------------------------------------------------------------------------
#pragma once

#include "ShapePoint.h"

class CShapeCircle : public CShapePoint
{
        // middle of the circle is used from base-class
        int m_r; // radius of the circle

public:
        CShapeCircle(void);
        CShapeCircle(double x1, double y1, double r);

        void SetParameter(double x1, double y1, double r);
        void Print();
};



//---------------------------------------------------------------------------
// ShapeCircle.cpp
//
// Defintion of a circle class
//
// Date: 08.05.2011
// Author: Thomas Braun
// Email: thomas.braun2@fh-aachen.de
//---------------------------------------------------------------------------

#include "ShapeCircle.h"
#include <iostream>

using namespace std;

//---------------------------------------------------------------------------
// Default-Constructor for a circle
//---------------------------------------------------------------------------
CShapeCircle::CShapeCircle(void)
{
        m_x1 = 0.0;
        m_y1 = 0.0;
        m_r = 0.0;
}

//---------------------------------------------------------------------------
// Constructor for a circle
//---------------------------------------------------------------------------
CShapeCircle::CShapeCircle(double x1, double y1, double r)
{
        m_x1 = x1;
        m_y1 = y1;
        m_r = r;
}

//---------------------------------------------------------------------------
// set parameter for a line
//---------------------------------------------------------------------------
void CShapeCircle::SetParameter(double x1, double y1, double r)
{
        m_x1 = x1;
        m_y1 = y1;
        m_r = r;
```

```
}

//------------------------------------------------------------------------------
// printout for CNC command
//------------------------------------------------------------------------------
void CShapeCircle::Print()
{
        cout << "circle " << ";" << m_x1 << ";" << m_y1 << ";" << m_r << endl;
}
```

In diesem Design ist jetzt die Enumerationsvariable nicht mehr notwendig, da der Typ der Figur aus der Klasse hervorgeht. In der Print-Methode muss auch keine Switch-Anweisung mehr den Typ unterscheiden. In der Deklaration der Klassen „CShapeLine“ und „CShapeCircle“ wird auch nicht mehr x1 und y1 angegeben, da diese von der Basisklasse geerbt werden.

In der Hauptfunktion ändert sich nun folgendes.

```
//------------------------------------------------------------------------------
// main.cpp
//
// shapes in a class
//
// Date: 04.05.2011
// Author: Thomas Braun
// Email: thomas.braun2@fh-aachen.de
//
//------------------------------------------------------------------------------

#include "ShapePoint.h"
#include "ShapeLine.h"
#include "ShapeCircle.h"
#include <iostream>

using namespace std;

void main()
{
        // create instances
        // a point with x1=2.1 y1=4.3
        CShapePoint myPoint(2.1, 4.3);
        // a line with x1=3.7 y1=5.6 x2=8.9 and y2==21.5
        CShapeLine myLine(3.7, 5.6, 8.9, 21.5);
        // a circle with x1=6.3 y1=11.6 and r=4.1
        CShapeCircle myCircle(6.3, 11.6, 21.5);

        // Print out to console
        myPoint.Print();
        myLine.Print();
        myCircle.Print();

        system("PAUSE");
}
```

Die Ausgabe entspricht immer noch den vorigen Ausgaben.

Dynamisches Erzeugen

Eine Klasse kann auch dynamisch mit „new“ erzeugt und mit „delete“ wieder zerstört werden. Mit dem Erzeugen wird dann auch wieder der Konstruktor aufgerufen.

```
void main()
{
        // Make a pointer variable and assign a dynamically created object
        CShapePoint* myPoint= new CShapePoint;

        // call the methods of the object
        myPoint->SetParameter(2.1, 4.3);
```

```
        myPoint->Print();

        // destroy object
        delete myPoint;
        myPoint = NULL;

        // Make a pointer variable and assign a dynamically created object
        // and do the initialisiation
        CShapeCircle* myCircle = new CShapeCircle(6.3, 11.6, 21.5);

        // call the methods of the object
        myCircle->Print();

        // destroy object
        delete myCircle;
        myCircle = NULL;

        system("PAUSE");
}
```

Gestalt ändern

Man kann ein Objekt einer abgeleiteten Klasse so verwenden, dass sie sich nur wie die Basisklasse verhält. Das geht besonders einfach, wenn man mit Zeigern auf Objekte arbeitet. Es ist möglich eine Objekt einer ableiteten Klasse zu instanziieren und dann mit einem Type-Cast einer Zeigervariablen auf die Basisklasse zuzuweisen. Die Zeigervariable der Basisklasse verhält sich dann so als ob es nur die Basisklasse gäbe.

```
//---------------------------------------------------------------------------
// Declaration of a base class
//---------------------------------------------------------------------------
class  CBaseClass
{
protected:
        int m_baseAttrib;

public:
        void publicBaseMethod() { // Do something }
};

//---------------------------------------------------------------------------
// Declaration of a derived class, inherit from CBaseClass
//---------------------------------------------------------------------------
class CDerivedClass : public CBaseClass
{
private:
        double m_derivedAttrib;

public:
        void publicDerivedMethod() { // Do something }
};

void main()
{
        // Dynamically created instance of a derived class
        CDerivedClass* myDerivedClass = new CDerivedClass;

        // Assign only the base-class part of the derived class to the pointer
        CBaseClass* myBaseClass = (CBaseClass*) myDerivedClass;

        // the base-class pointer can only use the base-class
        myBaseClass->publicBaseMethod();

        system("PAUSE");
}
```

Für unsere Beispielanwendung benötigen wir eine List mit Anweisungen. Die Elemente der Liste könnten die Figuren sein die wir bereits erstellt haben.

Wir wollen eine einfache Form für eine Liste wählen, indem wir ein Feld mit Zeigern auf „CShapePoint“ Objekte erzeugen. Die Anzahl der Elemente wird fest mit 10 vorgegeben. Damit können maximal 10 Figuren gespeichert werden. Eine Zählervariable speichert die letzte freie Position in der Liste.

```
//---------------------------------------------------------------------------
// main.cpp
//
// shapes in a class
//
// Date: 04.05.2011
// Author: Thomas Braun
// Email: thomas.braun2@fh-aachen.de
//
//---------------------------------------------------------------------------

#include "ShapePoint.h"
#include "ShapeLine.h"
#include "ShapeCircle.h"
#include <iostream>

using namespace std;

void main()
{
        // create a list with max_shape elements and a last position variable
        const int c_max_shapes = 10;            // Max elements in the list
        CShapePoint* shapeList[c_max_shapes];   // The list itself as a pointer array
        int lastShapeListPosition = 0;          // Last free position in the array

        // now fill the list

        // a point with x1=2.1 y1=4.3
        shapeList[lastShapeListPosition] = new CShapePoint(2.1, 4.3);
        lastShapeListPosition++; // increment last free position

        // a line with x1=3.7 y1=5.6 x2=8.9 and y2==21.5
        shapeList[lastShapeListPosition] = (CShapePoint*)new CShapeLine(3.7, 5.6, 8.9, 21.5);
        lastShapeListPosition++; // increment last free position

        // a circle with x1=6.3 y1=11.6 and r=4.1
        shapeList[lastShapeListPosition] = (CShapePoint*)new CShapeCircle(6.3, 11.6, 21.5);
        lastShapeListPosition++; // increment last free position


        // Iterate the array until the last free position
        for (int i=0; i<lastShapeListPosition; i++)
        {
                // Printout each element to the console
                shapeList[i]->Print();
        }

        system("PAUSE");
}
```

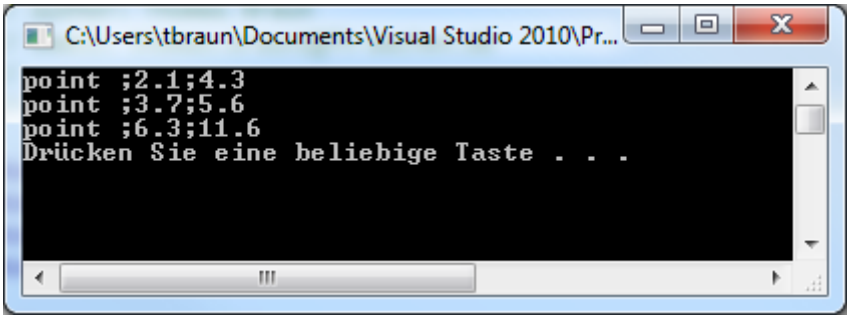


Abbildung 5: Ausgabe ohne virtuelle Methoden

Bei der Ausgabe erkennen wir, dass die Print-Methode immer nur die der Basisklasse aufruft. Es werden also 3 Punkte ausgegeben, anstatt einem Punkt, einer Linie und einem Kreis. Dieses Problem kann behoben werden, indem man die Print-Methode in der Basisklasse als virtuell deklariert. Das hat zur Folge, dass die Methode der Basisklasse von der abgeleiteten Klasse überschrieben wird.

```
//---------------------------------------------------------------------------
// ShapePoint.h
//
// Declaration of a point class
//
// Date: 08.05.2011
// Author: Thomas Braun
// Email: thomas.braun2@fh-aachen.de
//---------------------------------------------------------------------------
#pragma once

class CShapePoint
{
protected:
        double m_x1,m_y1; // Coordinates of the point

public:
        CShapePoint();
        CShapePoint(double x1, double y1);

        void SetParameter(double x1, double y1);
        virtual void Print();
};
```

Damit ist die Ausgabe dann wieder korrekt.

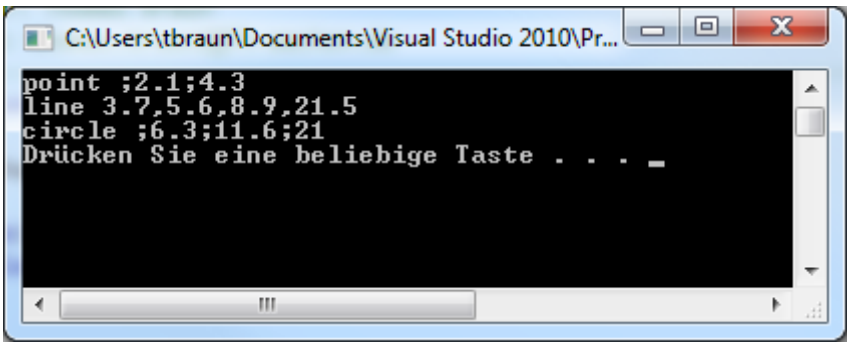


Abbildung 6: Ausgabe mit virtuellen Methoden

Methoden überschreiben

Eine abgeleitet Klasse kann eine Methode haben die exakt gleich heißt, die gleichen Parameter hat und auch den gleichen Rückgabewert wie die Methode einer Basisklasse. In diesem Fall spricht man davon, dass die Methode der Basisklasse von der abgeleiteten Klasse überschrieben wurde. Solange es die Methode in der abgeleiteten Klasse nicht gab, wäre immer die Methode der Basisklasse aufgerufen worden. Jetzt kommt es aber auf den Kontext, bzw. die Schnittstelle die man aufruft. Ruft man ein Objekt mit der Schnittstelle der Basisklasse auf, dann wird auch die Methode der Basisklasse ausgeführt. Wird das Objekt mit der Schnittstelle der abgeleiteten Klasse aufgerufen, dann wird auch die Methode der abgeleiteten Klasse aufgerufen. Neue Methoden der abgeleiteten Klasse erweitern die Basisklasse um neue Funktionen. Überschriebene Methoden der Basisklasse verändern die Funktion der Basisklasse ohne dass die Basisklasse selbst geändert wird.

Virtuelle Methoden

Wenn Methoden einfach überschrieben werden, dann ist die Ausführung der Funktion noch immer an die Aufrufschnittstelle gebunden. D.h. wird eine überladene Methode über die Schnittstelle der Basisklasse aufgerufen, dann wird auch die Methode der Basisklasse aufgerufen. Das ist aber nicht immer gewünscht. Möchte man z.B. Objekte die geometrische Formen repräsentieren in einem Container verwalten, dann wäre es gut, wenn der Container nur mit der Schnittstelle der Basisklasse (z.B. „CShape“) arbeiten kann. Der Container könnte für jedes Element im Container über die Schnittstelle die Methode „Print“ aufrufen, um die Daten der geometrischen Form auszugeben. Die Printmethode sollte aber in der abgeleiteten Klasse überschrieben werden und auch dann aufgerufen werden, wenn die Schnittstelle der Basisklasse verwendet wurde. Das erreicht man mit virtuellen Methoden.

Virtuelle Methoden werden in der Basisklasse mit dem Schlüsselwort „virtual“ vor dem Methodennamen gekennzeichnet. Damit weiß der Kompilierer, dass sobald eine abgeleitete Klasse die Methode überschreibt, die Methode auch für die Basisklasse überschrieben wird. Die Ursprüngliche Implementierung der überladenen Methode kann nur noch unter expliziter Angabe der Basisklasse vor dem Methodennamen aufgerufen werden.

Manchmal macht es keinen Sinn, dass es eine Implementierung für eine virtuelle Methode in der Basisklasse gibt. In dieser Situation, kann in der Deklaration nach der Signatur der Methode „= 0“ geschrieben werden. Dann spricht man von einer abstrakten Methode. Eine Klasse die mindestens eine abstrakte Methode enthält kann nicht instanziiert werden. Man nennt eine solche Klasse ebenfalls abstrakt.

Überladen von Methoden

Innerhalb einer Klasse kann es mehrere Methoden mit gleichem Namen, aber unterschiedlichen Parameterlisten geben. Das nennt man überladen. Es wird immer die Methode aufgerufen, deren Argumente mit den Typen der Parameter übereinstimmen. Ist keine Übereinstimmung möglich, wird noch versucht, eine implizite Typkonvertierung vorzunehmen. Ist auch das nicht möglich, dann wird ein Fehler beim Kompilieren ausgegeben.

Operatoren überladen

Allgemein können alle Operatoren in C++ überladen werden. Ein Operator wird definiert in dem das Schlüsselwort „operator“ gefolgt von dem Operatorsymbol und der Parameterliste geschrieben wird. Auch der Rückgabe typ muss angegeben werden. Im Grunde wird ein Operator wie eine Funktion oder Methode definiert. Nur die Namensregel ist etwas anders.

Klassen können ebenfalls Operatoren definieren. Auf diese Weise lässt sich eine klasse „CComplex“ für komplexe Zahlen erstellen und die Operationen „+“ und „-„ definieren. Andere Anwendungsmöglichkeiten sind Smart-Pointer. Smart-Pointer befreien den Programmierer davon das „new“ und „delete“ immer beachten zu müssen. Die Klasse beinhaltet einen normalen Pointer und ruft in den internen Methoden/Konstruktor/Destruktor das „new“ bzw. das „delete“ auf.

Lektion 11

Container

Objekte die Daten aufnehmen und verwalten werden als Container bezeichnet. Dabei werden verschieden Containerarten unterschieden. Jede Art hat seine Vor- und Nachteile, was Zugriffsgeschwindigkeit und Speicherverbrauch angeht. Manche Containerarten sind besonders schnell beim beliebigen Zugriff. Andere sind besonders schnell beim Einfügen an einer beliebigen Stelle. Manche Arten sind bei der Zugriffsgeschwindigkeit unabhängig von der Anzahl der Element die bereits im Container gespeichert sind. Andere werden dadurch langsamer. Im Folgenden werden einige wichtige Containerarten kurz charakterisiert.

Die hier dargestellten Containertypen sind Idealtypen. Es gibt aber auch viele Mischformen die versuchen die Eigenschaften für eine bestimmte Aufgabe zu kombinieren und damit zu optimieren.

Felder (engl. Array)

Das Feld ist bereits bekannt. Dadurch dass das die Daten immer direkt hintereinander im Speicher liegen, kann mit Hilfe der Indexnummer sehr schnell die Position eines Elementes im Speicher berechnet werden. Demnach ist der Zugriff auf ein beliebiges Element sehr schnell. Nachteil ist, das Einfügen eines neuen Elementes sehr aufwendig ist.

0	
1	
2	
...	
N-1	

Abbildung 7: Felder

Felder können so angewendet werden, dass sie bereits mehr Speicher reservieren, als aktuell im Feld Elemente gespeichert sind. Die Gesamtmenge die gespeichert werden kann wird dann Kapazität des Feldes genannt. Ist die Anzahl der gespeicherten Elemente kleiner als die Kapazität, dann ist auch noch das Anfügen am Ende des Feldes sehr effizient. Wird in diesem Fall ein Element an einer beliebigen Stelle eingefügt müssen die Nachfolgende Element um eine Stelle verschoben werden und dann kann das neue Element an dieser Stelle gespeichert.

0	
1	
2	
...	
N-1	

0	
1	
2	
...	
M-2	
M-1	

Abbildung 8: Feldelement einfügen an beliebiger Stelle

Ist hingegen die Kapaziät eines Feldes erschöpft, dann muss ein neues Feld erzeugt werden und die Elemente vom alten Feld in das neue Feld kopiert werden. Wird das neue Feld an beliebiger Stelle eingefügt, dann müssen erst alle Elemente vor dem neuen Feld kopiert werden, dann das neue Element einfügen und dann die nachfolgende Elemente umkopiert werden.

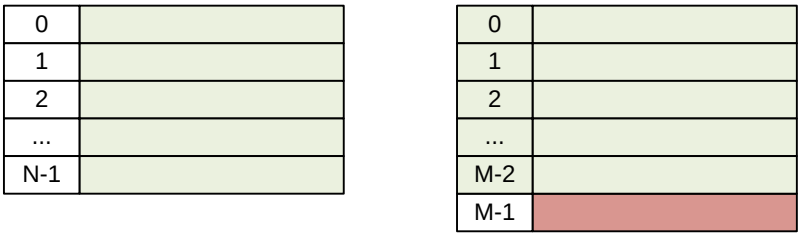


Abbildung 9: Feldelement anhängen

Verkettete Liste

Die Liste wird als einfach verkettete Liste oder als doppelt verkettete Liste verwendet. Bei der einfach verketteten Liste hat jedes Element einen Zeiger auf das nächste Element. Es gibt einen Zeiger der immer auf den Anfang zeigt und das letzte Element zeigt auf eine Ende-Kennung, z.B. auf „NULL“.

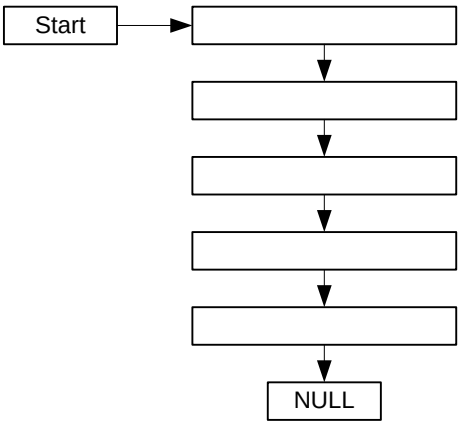


Abbildung 10: Einfach verkettete Liste

Bei der doppelt verketteten List zeigt jedes Element noch zusätzlich auf seinen Vorgänger. So kann man bei der doppelt verketteten Liste in beide Richtungen durch die Liste laufen (iterieren), während die einfach verkettete Liste nur in eine Richtung durchlaufen werden kann.

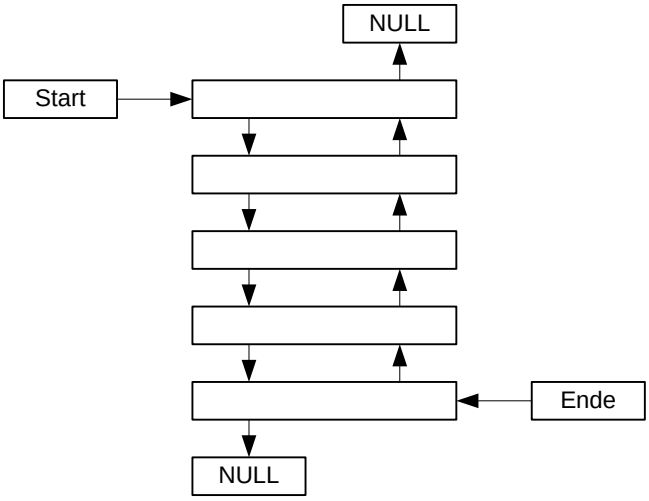


Abbildung 11: Doppelt verkettete Liste

Der Vorteil der verketteten Listen ist, dass man sehr schnell neue Elemente Einfügen kann. Da hier einfach Zeiger verbogen werden müssen. Allerdings ist der beliebige Zugriff sehr langsam und verschlechtert sich mit der Anzahl der Elemente.

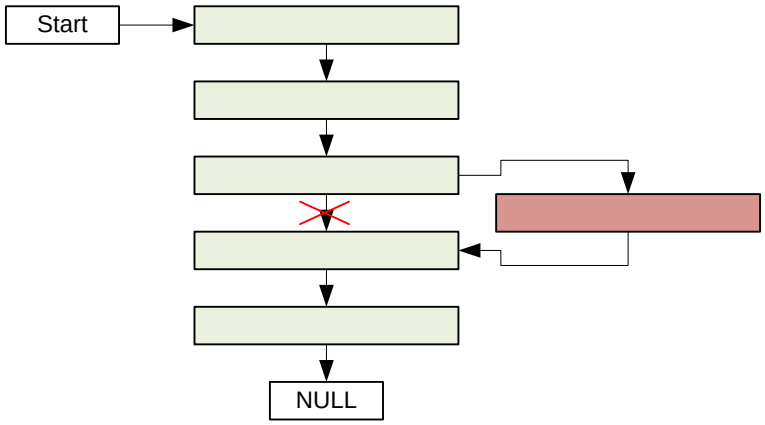


Abbildung 12: Einfügen in eine verkettete Liste

Werden auf die Elemente von einem der Enden der Reihe nach zugegriffen, dann ist die verkettete Liste wieder sehr schnell.

Der Speicherverbrauch ist höher als beim Feld, da für jedes gespeicherte Element noch ein, bzw. sogar zwei Zeiger gespeichert werden müssen.

Stapel (engl. Stack)

Den Stapel haben wir bereits bei der Speicherverwaltung kennengelernt. Stapelspeicher hat wird durch seine Zugriffsmöglichkeiten charakterisiert. Es kann immer nur auf das Element zugegriffen werden, was als letztes auf den Stapel gelegt wurde. Es besteht nicht die Möglichkeit ein Element an beliebiger Stelle einzufügen oder darauf zu zugreifen. Man nennt das auch LIFO Speicher (Last In First Out).

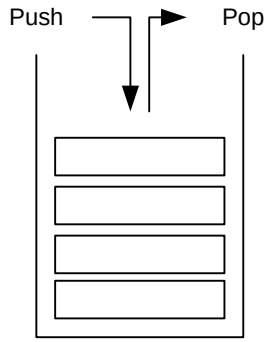


Abbildung 13: Stapelspeicher

Die Methoden um auf einen Stapel zuzugreifen heißen „Push“ um Daten auf dem Stapel abzulegen und „Pop“ um Daten vom Stapel zu entnehmen.

Warteschlange (engl. Queue)

Die Warteschlange ist im Gegensatz zum Stapel ein „FIFO“ Speicher (First In First Out). Das Element, das zuerst in die Warteschlange eingefügt wurde muss als erste auch wieder entnommen werden. Der wahlfreie Zugriff ist nicht vorgesehen.

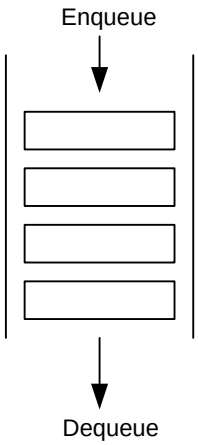


Abbildung 14: Warteschlange

Der Nutzen einer Warteschlange liegt darin das eine Reihenfolge der Elemente beim Zugriff erzwungen wird, was in bestimmten Situationen erwünscht wird.

Assoziatives Feld – Verzeichnis (engl. Dictionary)

Das Assoziative Feld ordnet einem Element einen eindeutigen Schlüssel zu unter dem das Element zu finden ist. Dabei darf ein Schlüsselwert nur einmal verwendet werden.

„Peter“	
„Hans“	
„Lucilla“	
...	
„Karl“	

Abbildung 15: Assoziatives Feld

Der Datentyp des Schlüssels kann beliebig sein. Es könnte ein einfacher Integer, ein String oder auch ein andere komplexer Datentyp sein. Häufig wird aber eine Zeichenkette verwendet.

Ringspeicher

Der Ringspeicher hat die Eigenschaft, dass er besonders gut als Pufferspeicher verwendet werden kann.

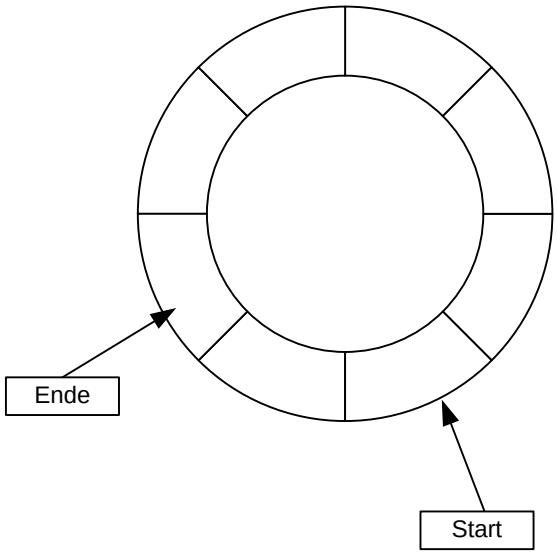


Abbildung 16: Ringspeicher

Der Ringspeicher hat eine feste Kapazität. Es wird immer nach dem letzten gespeicherten Element geschrieben und das zuerst geschriebene Element wieder ausgelesen. Je nach Variante kann das älteste Element überschrieben und damit verworfen werden, wenn der Speicher bereits voll ist.

Lektion 12

Um Programmcode für verschiedene Anwendungen wiederverwenden zu können, kann man ihn in eine Bibliothek auslagern. Es gibt verschiedene Arten von Bibliotheken, die ihre Vor- und Nachteile haben, bzw. ihren jeweiligen Anwendungszweck haben.

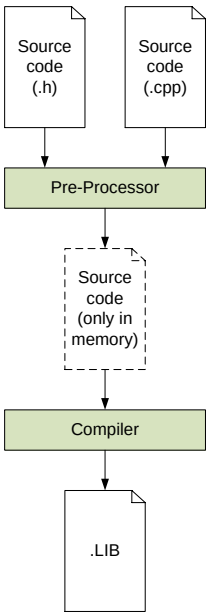
Quellcode Bibliothek

Bei der Quellcode Bibliothek wird der Quellcode in Form von Quellcodedateien bereitgestellt. Meistens sind das Headerdateien (*.h) in der z.B. Konstanten definiert sind oder auch generische Funktionen. Die Quellcodebibliotheksdateien müssen einfach in das Anwendungsprojekt mit eingebunden werden und können dann im Quellcode verwendet werden.

Quellcodebibliotheken sind nicht vorkompiliert und müssen demnach immer mit der Anwendung zusammen kompiliert werden.

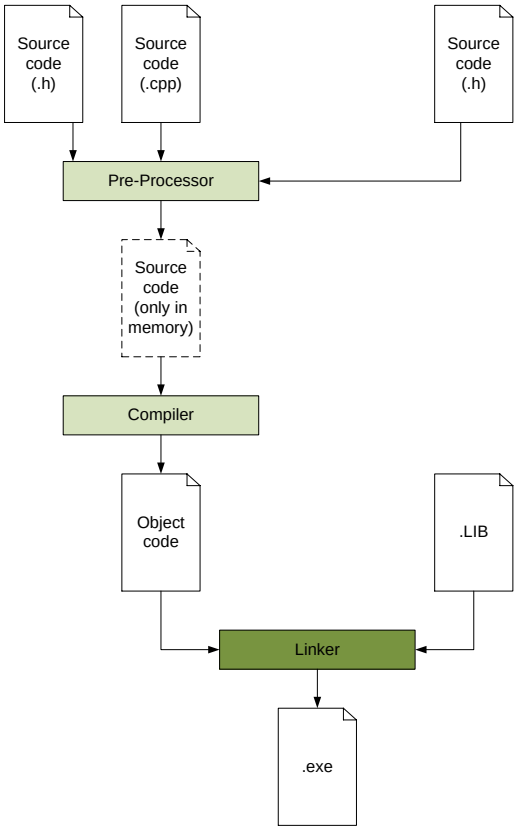
Statische Bibliothek

Die statische Bibliothek ist eine vorkompilierte Bibliothek.



Unter Windows haben statische Bibliotheken die Dateierweiterung „.LIB“ unter Linux ist das „.a“. Sie sind ähnlich aufgebaut wie die Objektcode-Dateien und sind für sich alleine noch nicht ausführbar.

In einem Anwendungsprojekt wird die Bibliothek vom Linker in den Code der Ausführbaren Datei (*.exe) eingebunden. Damit der Anwendungsprogrammierer die Dateien aufrufen kann stellt man eine oder mehrere Headerdateien mit der Bibliothek zusammen zur Verfügung.



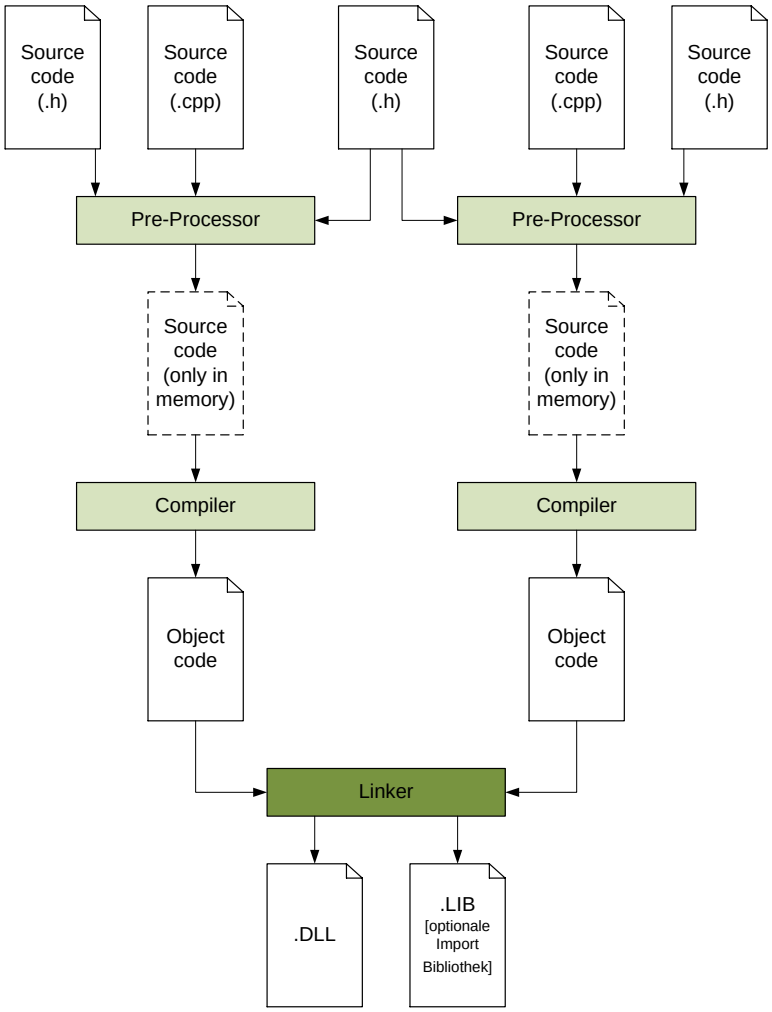
Ein Nachteil der statischen Bibliotheken ist, dass falls etwas an der Bibliothek korrigiert werden muss, alle Anwendungen die diese Bibliothek verwenden neu kompiliert und ausgeliefert werden müssen. Das kann sehr aufwendig sein.

Dynamische Bibliothek

Eine dynamische Bibliothek ist im Prinzip eine ausführbare Datei, nur dass sie nicht eine Prozess starten kann, sondern von einem bestehenden Prozess zur Laufzeit in den Prozess geladen wird.

Unter Windows haben dynamische Bibliotheken die Dateiendung „*.DLL“ (Dynamic Link Library), unter Linux ist das „*.so“ (Shared Object).

Das Laden der Bibliothek erfolgt über Betriebssystemfunktionen (API) außerdem müssen die Aufrufbaren Funktionen bekannt sein. Um das zu erleichtern wird meist noch zusätzlich eine statische Bibliothek erstellt die die Funktionen zum dynamischen Laden der dynamischen Bibliothek enthält. Die statische Bibliothek wird dann in die ausführbare Anwendung eingebunden. Diese statische Bibliothek wird auch als Import-Bibliothek bezeichnet.



Vorteil der dynamischen Bibliothek ist, dass sie einfach ausgetauscht werden kann, ohne den Rest der Anwendung anfassen zu müssen. Auch das Kompilieren der Anwendung und der Bibliothek ist unabhängig.

Nachteilig ist, dass inkompatible Versionen der Bibliothek bestimmte Anwendungen unbrauchbar machen können. Sollte eine Anwendung eine ältere Version erwarten als die Version, die gerade mit einer anderen Anwendung installiert wurde, wird die alte Anwendung nicht mehr funktionieren.

Weitere Aussichten

Dieser Kurs versucht sich auf wichtige Grundlagen zu stützen und muss dabei eine Auswahl der Themen treffen, die für einen ersten Einstieg wichtig sind. Im Folgenden möchte ich Ihnen noch ein paar Stichpunkte geben, zu Themen die noch nicht behandelt wurden, aber für den Interessierten als nächstes angegangen werden sollte.

- Type-Casting
- Typdefinitionen
- Funktionszeiger
- Template Funktionen und Template Klassen
- Referenzdatentypen
- Die Standard Template Bibliothek (STL)
- Konstruktion von Klassen und deren Verwendung beim Type-Casting
- Copy-Konstruktoren

- Default-Parameteter
- Ausnahmebehandlung (Exceptions)
- Runtime Type Information (RTTI)
- …