rtr Startseite
Einführung
Inhalt
1. Vorbemerkungen
I. Linux
II. C
III. C++

C/C++-Crashkurs

Eva Brucherseifer, Martin Schneider, Lisa Schramm
Regelungstheorie & Robotik
TU Darmstadt

August 2006

Hinweis: Dieser Online-Kurs wird nicht weiter gepflegt, bleibt aber für Interessenten weiter zugänglich.
Wir verweisen auf die Veranstaltung "Programmierung in der Automatisierungstechnik", in deren Rahmen auch neue Kursmaterialien zur Verfügung gestellt werden..
Dieser Kurs geht auf das Programmieren von Anwendungen unter Linux ein. Dazu wird dem Kursteilnehmer zunächst das Betriebssystem Linux mit den Tools nähergebracht, die man zum Programmieren benötigt.
Teil I beschäftigt sich weiterhin mit den Grundlagen des Compilierens, d.h. des Umwandelns von Programmcode in ausführbare Programme.

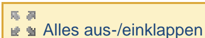
In Teil II wird die Programmiersprache C vorgestellt. Am Ende des Kapitels befinden sich Aufgaben. Dem Kursteilnehmer wird empfohlen sich im Rahmen der Aufgaben mit der Sprache auseinanderzusetzten.

Teil III beschäftigt sich mit der Programmiersprache C++, die auf C aufbaut. Es werden die Basistechniken objektorientierter Programmierung und deren Anwendung in C++ angesprochen.

Wir hoffen, daß sie in diesem Kurs viel lernen. Das wichtigste ist jedoch, daß Sie sich selber bei den Aufgaben und darüber hinaus mit der Programmierung auseinandersetzten. Wie jede Sprache, lernt man auch Programmiersprachen allein durch Übung.

Bei Fragen, Problemen oder Anregungen melden Sie sich bitte bei lschramm(at)rtr.tu-darmstadt.de.
Viel Spaß!

rtr Startseite
Einführung
Inhalt
1. Vorbemerkungen
I. Linux
II. C
III. C++


Alles aus-/einklappen

rtr Startseite

Einführung

Inhalt

1. VORBEMERKUNGEN

1.1 Aufbau des Kurses

I. LINUX

2. ÜBERSICHT

3. DATEISYSTEM

4. SHELL

5. PROGRAMMIEREN UNTER LINUX

6. Literaturhinweise

7. AUFGABEN

II. C

8. Ein C-Programm

9. Variablen / Datentypen

10. WICHTIGE SPRACHELEMENTE

11. FUNKTIONEN

12. Benutzerdefinierte Typen

13. ÜBERSETZEN VON C-PROGRAMMEN

14. ARRAYS

15. ZEIGER

16. AUFGABEN

III. C++

17. DAS KLASSENKONZEPT

18. BASISTECHNIKEN

19. Templates

20. DIE STANDARDBIBLIOTHEK STL

21. KLASSEN II

22. FORTGESCHRITTENE TECHNIKEN

23. VERERBUNG

24. POLYMORPHIE

25. AUFGABEN

rtr Startseite
Einführung
Inhalt
1. Vorbemerkungen
1.1 Aufbau des Kurses
I. Linux
II. C
III. C++

An wen wendet sich diese Einführung?

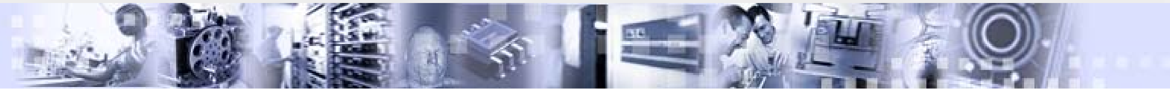
Die vorliegende Einführung richtet sich vorrangig an Studenten, die Interesse an einer Studien- oder Diplomarbeit am Fachgebiet Regelungstheorie und Robotik haben und an die Teilnehmer des alljährlich stattfindenden Robotik-Seminars. Darüberhinaus sind natürlich auch alle aus anderen Gründen Interessierten herzlich willkommen.

Was ist der Sinn dieser Einführung?

Die Studenten sollen auf ihre Arbeit an unserem Fachgebiet vorbereitet werden. Am Fachgebiet Regelungstheorie und Robotik besteht die Arbeitsumgebung im wesentlichen aus Solaris-Servern und Linux-Arbeitsplatzrechnern. Programmiert wird fast durchgängig in den Sprachen C und C++.
Die Einführung möchte folgendes erreichen:

- Die Grundlagen der Programmiersprachen C und C++ sollen vermittelt werden.
- Der Einsatz der Sprachen C und C++ in einer UNIX-Programmierumgebung soll vorbereitet werden.
- Ein Überblick über die Möglichkeiten zur Programmierung graphischer Oberflächen soll gegeben werden.

Grundlegende Voraussetzungen an die Kursteilnehmer, etwa in C oder einer anderen Programmiersprache bestehen nicht, vereinfachen jedoch den Einstieg.



rtr Startseite
Einführung
Inhalt
1. Vorbemerkungen
> 1.1 Aufbau des Kurses
I. Linux
II. C
III. C++

1.1 Aufbau des Kurses

Der Kurs besteht aus 4 Teilen:

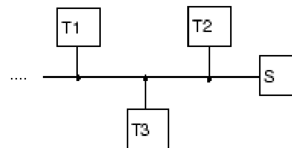
- Einführung in die Programmierung unter Linux mit einer Compiler-Sprache.
- Programmiersprache C
- Programmiersprache C++ und Überblick über Software Engineering
- Graphische Programmierung

Jeder Teil ist begleitet von praktischen Übungen. Die Teilnehmer sollten sich jedoch auch darüber hinausgehend mit den Programmiersprachen beschäftigen.
Eine Literaturliste mit einführenden Büchern, aber auch mit Standardwerken für den erfahrenen Progarmmierer, ist am Ende jeden Teils zu finden.

rtr Startseite
Einführung
Inhalt
1. Vorbemerkungen
I. Linux
> 2. Übersicht
2.1 Graphische Oberflächen in Linux
3. Dateisystem
4. Shell
5. Programmieren unter Linux
6. Literaturhinweise
7. Aufgaben
II. C
III. C++

2. Übersicht

Linux ist ein echtes Mehrbenutzersystem (multi-user).
Das heißt, daß mehrere Benutzer völlig getrennt und ungestört gleichzeitig am selben Rechner arbeiten können. Die Rechenleistung und die Ressourcen des Systems werden dann von Linux zwischen den verschiedenen Benutzern verteilt.
Jeder Benutzer hat immer ein eigenes Homeverzeichnis, in dem seine Einstellungen und persönliche Dateien gespeichert werden. Unix-Rechner können in ein lokales Netzwerk (local area network, LAN) mit einheitlicher Benutzerverwaltung und zentralem Fileserver integriert sein.



Zugang zum System

Um auf einem Linux-Rechner arbeiten zu können, muß man sich erst darauf einloggen, indem man seinen Login-Namen und sein Passwort angibt. Jeder Benutzer hat nur beschränkten Zugriff auf das System. Die Dateien haben verschiedene Rechte, um nur bestimmten Benutzern zu erlauben, sie zu lesen, schreiben, oder auszuführen.
Nur ein Benutzer besitzt die Rechte auf alle Dateien. Dieser Benutzer heißt ``root''. Er ist der Verwalter des Systems, und kann alle Einstellungen ändern, Programme installieren, neue Benutzer einrichten, usw.

Verlassen des Systems

Nach dem Arbeiten verläßt man das System, indem man sich ``ausloggt'', um seine Daten zu schützen.

Prozesse:

Wenn Programme ausgeführt werden, sind sie dem System als Prozesse bekannt.

- Jeder Prozess hat eine eindeutige Nummer: Die PID (Process Identification Number)
- Die Prozesse sind baumartig strukturiert. Wurzel ist PID 0.
- Jeder Prozess ist einem Benutzer zugeordnet.

Shells (Kommandointerpreter):

- Eine Shell interpretiert die Befehlseingabe auf der Kommandozeile
- Aufbau eines Befehls: `<Befehlsname> <Optionen> <Parameter>`
- Beispiel: `man ls`
- Es gibt interne (in die Shell eingebaute) und externe Befehle (Programme) sowie selbstgeschriebene Skripte (sog. Shellskripte, die Aufrufe von internen und externen Befehlen enthalten)
- Die Shell speichert ihre Umgebung in Umgebungsvariablen. Diese können mit dem Befehl `export` angesehen und verändert werden. Die Umgebungsvariablen können in den Dateien `.profile` und `.bashrc` im Homeverzeichnis verändert werden.
- Der Suchpfad für Programme befindet sich in der Umgebungsvariablen PATH.

rtr Startseite
Einführung
Inhalt
1. Vorbemerkungen
I. Linux
2. Übersicht
2.1 Graphische Oberflächen in Linux
3. Dateisystem
4. Shell
5. Programmieren unter Linux
6. Literaturhinweise
7. Aufgaben
II. C
III. C++

2.1 Graphische Oberflächen in Linux

Der X-Server ermöglicht die graphische Darstellung unter Linux und bildet die Schnittstelle zur Graphikkarte. Um ein einfaches Programmieren von Fensterprogrammen zu ermöglichen, gibt es verschiedene Toolkits - auch für Skriptsprachen:

- Tcl/Tk, Perl oder Python als Kleber
- Motif
- Gtk
- QT/KDE

Zum Teil sind diese Toolkits auch auf anderer Betriebssysteme (Windows, Mac OS) portabel. Aufgabe der Toolkits ist es, graphische Elemente (Widgets) wie Buttons, Dialoge, Texteingabe etc. und eine zentrale Event-Loop für die Benutzeraktion zur Verfügung zu stellen.

2.2 Windowmanager, Desktops

Aufbauend auf diesen Toolkits und dem X-Server gibt es für Linux verschiedene Windowmanager bzw. Desktops:

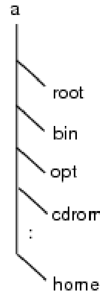
- fvwm
- KDE
- Gnome

Ein Windowmanager sorgt dafür, dass graphische Programme einen Rahmen haben, deren Fenster groß und klein gemacht werden können, daß sie sich überlappen können, etc. Ein Desktop Environment bietet zudem Start- und Taskleisten, Desktop-Icons, ein Clipboard, Drag&Drop-Funktionen für einen guten Workflow.
Zudem gibt es eine Vielzahl von graphischen Programmen.

rtr Startseite
Einführung
Inhalt
1. Vorbemerkungen
I. Linux
2. Übersicht
> 3. Dateisystem
3.1 Bedeutung der Directoryeinträge
3.2 Der Befehl "chmod"
3.3 Besondere Pfade
4. Shell
5. Programmieren unter Linux
6. Literaturhinweise
7. Aufgaben
II. C
III. C++

3. Dateisystem

Um effektiv mit der Shell zu arbeiten, braucht man Kenntnisse über die Datei- und Verzeichnis-Struktur unter Linux. Verzeichnisse sind Ordner, in denen Dateien, Programme oder auch Unterverzeichnisse abgelegt werden können. Alle Verzeichnisse des Systems sind baumartig angeordnet. Das Wurzelverzeichnis ist in der Hierarchie ganz oben und wird mit ``/'' angesprochen. Von hier aus gelangt man zu allen anderen Verzeichnissen.



Die Partitionen einer Festplatte können in diesen Verzeichnisbaum integriert werden, indem sie gemountet werden. Dies gilt auch für Wechselmedien wie Floppys oder CDs. Bevor sie wieder entnommen werden, müssen die Medien mit umount wieder aus dem Verzeichnisbaum entfernt werden, dieser Befehl bewirkt auch, daß aus Performancegründen gecachte Schreibzugriffe zuvor ausgeführt werden.
Mit dem Befehl cd <Verzeichnis> (change directory) kann das Verzeichnis gewechselt werden. Die Eingabe von cd bzw. cd ~ führt den Benutzer stets in sein eigenes Heimatverzeichnis (meistens /home/<Loginname>).
Um sich den Inhalt eines Verzeichnisses anzuschauen, geben Sie den Befehl ls <Verzeichnis> ein. Wenn Sie kein Verzeichnis hinter ls angeben, wird das aktuelle Arbeitsverzeichnis angezeigt. Hier ein typisches Home-Verzeichnis:

```
Desktop  bild.svg  public_html
```

Um mehr Informationen zu erhalten, empfiehlt sich die Option ls -l. Normalerweise gibt es für diesen Befehl den ``Alias'' ll.

```
total 20
drwx------    3 eva      users        4096 2002-06-07 20:38 Desktop
-rw-r--r--    1 eva      users        6202 2002-06-09 11:35 bild.svg
drwxr-xr-x    2 eva      users        4096 2002-06-02 20:26 public_html
```

Und um alle versteckten Dateien zu sehen (z.B. alle Dateien, die mit einem . beginnen), sollten Sie ls -la eingeben. Diese versteckten Dateien enthalten die User-Konfiguration.

```
total 1680
drwxr-xr-t   16 hacking  users        4096 2002-10-20 11:06 .
drwxr-xr-x    5 matze    users        4096 2002-10-04 08:46 ..
-rw-r--r--    1 hacking  users        5742 2002-10-04 08:44 .Xdefaults
-rw-r--r--    1 hacking  users        1305 2002-10-04 08:44 .Xmodmap
lrwxrwxrwx    1 root     root           10 2002-10-04 01:45 .Xresources ->
.Xdefaults
-rw-------    1 hacking  users        2668 2002-10-20 11:07 .bash_history
-rw-r--r--    1 hacking  users        1286 2002-10-04 08:44 .bashrc
-rw-r--r--    1 hacking  users         208 2002-10-04 08:44 .dvipsrc
-rw-r--r--    1 hacking  users        1637 2002-10-04 08:44 .emacs
drwxr-xr-x    4 hacking  users        4096 2002-06-07 11:26 .kde
-rw-r--r--    1 hacking  users         934 2002-10-04 08:44 .profile
drwxr-xr-x    2 hacking  users        4096 2002-09-21 12:57 .qt
-rwxr-xr-x    1 hacking  users        2432 2002-10-04 08:44 .xinitrc
-rw-r--r--    1 hacking  users        1101 2002-10-04 08:44 .xserverrc.secure
-rwxr-xr-x    1 hacking  users        2794 2002-10-04 08:44 .xsession
-rw-------    1 hacking  users     1395772 2002-06-07 20:38 .xsession-errors
drwx------    3 hacking  users        4096 2002-06-07 20:38 Desktop
-rw-r--r--    1 hacking  users        6202 2002-06-09 11:35 bild.svg
drwxr-xr-x    2 hacking  users        4096 2002-06-02 20:26 public_html
```

rtr Startseite
Einführung
Inhalt
1. Vorbemerkungen
I. Linux
2. Übersicht
3. Dateisystem
3.1 Bedeutung der Directoryeinträge
3.2 Der Befehl "chmod"
3.3 Besondere Pfade
4. Shell
5. Programmieren unter Linux
6. Literaturhinweise
7. Aufgaben
II. C
III. C++

3.1 Bedeutung der Directoryeinträge

Wenn Sie die ersten 10 Zeichen einer Zeile der Ausgabe von "ls -l" betrachten, werden Sie recht schnell ein einfaches Muster erkennen. Das erste Zeichen überhaupt gibt den Typ der Datei an. Dabei steht

- d für Directory,
- l für Link und ein
- - für eine normale Datei.

Danach folgen drei Dreierblöcke, welche die Zugriffsrechte regeln. Der erste Dreierblock steht dabei für

- den Benutzer (User = u) selbst (also Sie),
- der zweite Block beschreibt die Gruppenrechte (Group = g) und
- der dritte Block regelt den Zugriff der übrigen Welt (Others = o).

Jeder Block besteht aus drei Flags, welche gesetzt sein können. Das sind im Einzelnen

- r (read = lesen),
- w (write = schreiben) und
- x (execute = ausführen).

Der User sollte in seinem Homeverzeichnis normalerweise alles dürfen, weswegen oft die ersten drei rwx gesetzt sind. Zumindest Schreiben sollte man allen anderen verbieten. Bei Verzeichnissen muß das x-Flag gesetzt sein, damit der entsprechende Benutzer in das Verzeichnis wechseln darf. Wenn Sie also nicht wollen, daß irgendjemand ihre Dateien liest oder ausführt (von Schreiben ganz zu Schweigen), sollten die Berechtigungen einer Datei etwa so aussehen wie für die Datei .bash_history im oberen Beispiel (rw- --- ---).
Der nächste Eintrag ist weniger wichtig, danach kommt der Name des Besitzers der Datei (hier immer root) und dann die Gruppe (auch root). Der nächste Eintrag ist eine Zahl, welche die Größe der Datei in Bytes angibt. Dann folgt das Datum der letzten Änderung und anschließend der Name der Datei.

rtr Startseite
Einführung
Inhalt
1. Vorbemerkungen
I. Linux
2. Übersicht
3. Dateisystem
3.1 Bedeutung der Directoryeinträge
3.2 Der Befehl "chmod"
3.3 Besondere Pfade
4. Shell
5. Programmieren unter Linux
6. Literaturhinweise
7. Aufgaben
II. C
III. C++

3.2 Der Befehl "chmod"

Mit diesem Befehl ändert man die Zugriffsrechte auf eine Datei oder ein Verzeichnis.

Format: `chmod <Optionen> <Maske> <Dateiname>`

Der Dateiname kann eine Datei oder ein Verzeichnis sein.

Die Maske regelt die neuen Zugriffsrechte nach einem einfachen Prinzip: Für die drei Gruppen gibt es die Kürzel ugo (user, group, others), dann + (erlauben) oder - (verbieten) und zuletzt rwx (wie bisher).

Möchten Sie ihre Datei für die Welt lesbar machen, müssen Sie chmod o+r test eingeben. Wollen Sie alle Zugriffe verbieten, geben Sie einfach chmod ugo-rwx test ein. Keine Angst! Als Besitzer der Datei können Sie die Rechte immer noch ändern, auch wenn Sie die Datei jetzt weder lesen noch beschreiben können. Geben Sie jetzt chmod u+rwx test und anschließend chmod g+rx test ein, dann können wenigstens Ihre Gruppenmitglieder die Datei lesen und ausführen.

rtr Startseite
Einführung
Inhalt
1. Vorbemerkungen
I. Linux
2. Übersicht
3. Dateisystem
3.1 Bedeutung der Directoryeinträge
3.2 Der Befehl "chmod"
3.3 Besondere Pfade
4. Shell
5. Programmieren unter Linux
6. Literaturhinweise
7. Aufgaben
II. C
III. C++

3.3 Besondere Pfade

Auf jedem Linux System befinden sich besondere Pfade, die eine bestimmte Funktion haben.

- **/home**:
  Heimatverzeichnis der einzelnen Benutzer. Hier liegen alle Daten, Einstellungen und all das, worauf nur der Benutzer Zugriff haben soll. Im Allgemeinem können diese Daten von allen gelesen, aber von niemand anderem als der Eigentümer verändert werden.
- **/root**:
  Heimatverzeichnis des Administrators root.
- **/usr (Unix System Ressourcen)**:
  Aus diesem Verzeichnis holen sich viele Programme Dokumente, Informationen, Hilfstexte und andere wichtige Daten.
- **/bin und /sbin (Ausführbare Programme)**:
  in diesem Verzeichnissen sind viele systemnahe Programme zu finden, welche schon zum Systemstart benötigt werden.
- **/opt (optionale Software)**:
  Kommerzielle Software oder sehr große Programmpakete wie etwa KDE, Netscape, Mozilla usw. finden hier ihren Platz.
- **/etc (Konfigurationsordner)**:
  Hier sind Dateien zusammengefasst, die die Konfigurationsinformationen für den Rechner enthalten. Sie beinhalten z.B. Informationen zur Internetverbindung, zum Startmodus des Rechners oder einzelnen Programmen wie Backup (Datensicherung).
- **/boot (Ordner für den Systemstart)**:
  Dateien und Programme, die zum Systemstart benötigt werden, z.B. der Kernel.
- **/var (Protokoll-Dateien)**:
  Log-Dateien und Verzeichnisse für temporäre Daten (caches)
- **Sonstige**:
  Daneben gibt es noch weitere Ordner, die Informationen zum System und den angeschlossenen Geräten enthalten: /lib und /usr/lib (Bibliotheken), /var (variable Daten), /proc (Prozesse), /media (auswechselbare Geräte wie Floppy, CD etc.) /dev (alle angeschlossene Geräte wie Drucker, Festplatten, Tastaturen etc.)

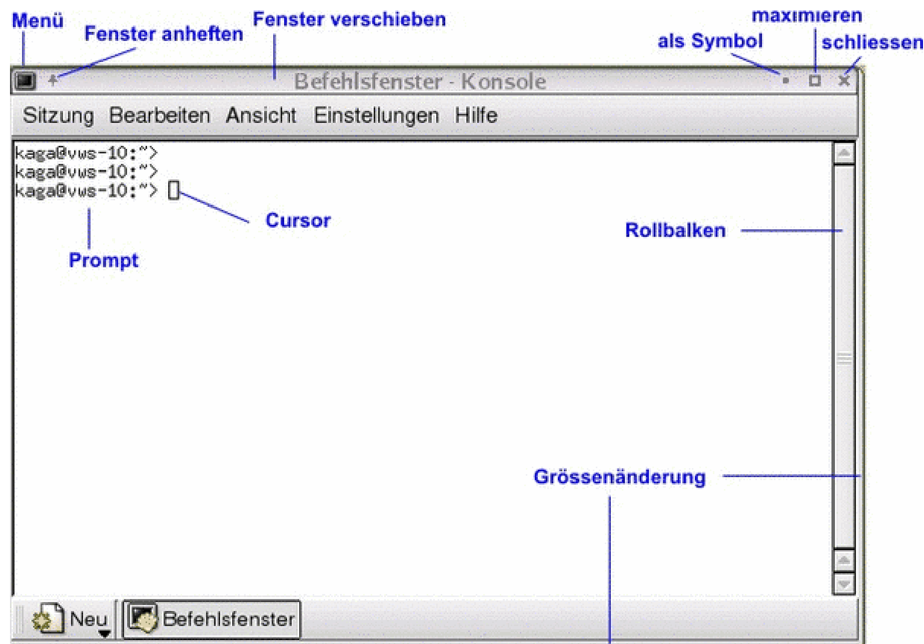
rtr Startseite
Einführung
Inhalt
1. Vorbemerkungen
I. Linux
2. Übersicht
3. Dateisystem
4. Shell
4.1 Befehle
4.2 Shellskripte
5. Programmieren unter Linux
6. Literaturhinweise
7. Aufgaben
II. C
III. C++

4. Shell

Auf der Taskleiste finden Sie ein Icon, das einen Monitor mit einer Muschel (engl. *shell*) darstellt. Wenn Sie mit der Maus auf dieses Symbol klicken, Öffnet sich das ``Konsole''-Fenster, in dem Sie Befehle eingeben können.Dieses Terminal stellt einen Kommandointerpreter in einer graphischen Oberfläche zur Verfügung. Kommandointerpreter stellen die einfachste Schnittstelle zum Betriebssystem dar, indem Befehle per Texteingabe entgegennehmen. Das komplette System lässt sich auf diese Weise steuern.

Auf UNIX/Linux-Betriebssystemen befinden sich standardmäßig eine Reihe von Kommandointerpretern, die grundsätzlich jeweils unterschiedliche Kommandosprachen bereitstellen. Glücklicherweise sind viele dieser Sprachen sehr ähnlich und überschneiden sich stark, da die verschiedenen Interpreter oft Erweiterungen von anderen Interpretern bereitstellen. Die wichtigsten Kommandointerpreter sind die Bash ('`bash`'), C-Shell ('`csh`','`tcsh`') sowie die Korn-Shell ('`ksh`'), von denen hier lediglich auf die Bash eingegangen wird.

Den Shellinterpreter kann man mit dem Befehl `'echo $SHELL'` auf der Kommandozeile abfragen. Sie sehen in der ersten Zeile den sogenannten Prompt "`benutzer@computer:~>`"



rtr Startseite
Einführung
Inhalt
1. Vorbemerkungen
I. Linux
2. Übersicht
3. Dateisystem
> 4. Shell
> 4.1 Befehle
4.2 Shellskripte
5. Programmieren unter Linux
6. Literaturhinweise
7. Aufgaben
II. C
III. C++

4.1 Befehle

Befehle bestehen aus verschiedenen Bestandteilen. Zuerst kommt immer das Befehlswort und dann die Parameter oder Optionen.
Jeder Befehl wird erst ausgeführt, wenn Sie die Eingabetaste drücken.
Als Beispiel wird der Befehl ls angenommen. Geben Sie in der Konsole nur ls ein, wird Ihnen der Inhalt des Verzeichnisses angezeigt, in dem Sie sich gerade befinden. Wenn Sie die Option ``l'' benutzen, und ls -l angeben, werden die Details angezeigt. Wenn sie danach ein Verzeichnissname angeben, z.B. ls /usr/, wird der Inhalt dieses Verzeichniss angezeigt.
Mit der option --help (z.B. ls --help) kann man alle möglischen Optionen eines Befehls abrufen.

```
man  [ - ]  [ -adFlrt ]  [ -M path ]  [  -T macro-package  ]
     [ -s section ] name ...
```

```
Befehlsname [Flags] [-Options] Parameter
```

Eine genauere Beschreibung eines Befehls mit seinen Optionen enthält die man-page. Diese erhält man mit Hilfe des Befehls man.
Mit dem Zeichen `&` am Ende eines Befehls bewirkt man, daß ein Programm als eigenständiger Prozess gestartet wird und die Kommandozeile wieder zur Verfügung steht.

4.1.1 Einige wichtige Befehle

Dateien

- `ls`: Anzeige der Dateien im aktuellen Verzeichnis
- `ls -l` bzw. `-ll`: zusätzliche Angabe der Zugriffsrechte, des Besitzers, der Größe und des Erstellungsdatum
- `ls -a` bzw. `-la`: mit einem Punkt beginnende Konfigurationsdateien werden auch angezeigt
- `mkdir <Name>`: Erstellt ein Unterverzeichnis Name
- `rmdir <Name>`: Löscht das leere Verzeichnis Name
- `rm <Name>`: Löscht die Datei Name
- `cd`: Wechsel in das Home-Verzeichnis
- `cd <Name>`: Wechsel in das Verzeichnis Name
- `cd ..`: Wechsel in das übergeordnete Verzeichnis
- `pwd`: Zeigt das aktuelle Verzeichnis an
- `cp <Quelle> <Ziel>`: Kopiert Quelle nach Ziel
- `mv <Quelle> <Ziel>`: Verschiebt Quelle nach Ziel bzw. kopiert Quelle in das Unterverzeichnis Ziel

Hilfe

- `man <Name>`: Online-Hilfe zum Befehl Name
- `<Name> --help`: Ausgabe der Befehlssyntax von Name

Textdateien

- `more <Name>`:Anzeige der Textdatei Name
- `cat <Name>`:Anzeige der Textdatei Name

Archive

- `tar -cf <archive.tar> <foo> <bar>`: Erstellt das Archiv archive.tar rekursiv mit den Dateien oder Verzeichnissen foo und bar
- `tar -tf <archive.tar>`: Gibt die Liste der Dateien, die sich in der Archive archive.tar befinden
- `tar -xf <archive.tar>`: Extrahiert alle Dateien aus der Archive archive.tar.
- `gzip <archive.tar>`: Komprimieren einer Datei
- `gunzip <archive.tar.gz>`: Entpacken einer Datei

Rechtevergabe

- `chmod`: Ändert die Zugriffsrechte auf eine Datei
- `chown`: Ändert den Besitzer einer Datei

Programmhandling

- `ps`: Zeigt die laufenden Prozesse
- `kill`: Beendet einen laufenden Prozess (mit Hilfe eines Interrupts)
- `nice`: Gibt einem Prozess eine andere Priorität

Suchfunktionen

- `find`: Finden von Dateien
- `grep`: Durchsucht Text-Dateien

rtr Startseite
Einführung
Inhalt
1. Vorbemerkungen
I. Linux
2. Übersicht
3. Dateisystem
4. Shell
4.1 Befehle
4.2 Shellskripte
5. Programmieren unter Linux
6. Literaturhinweise
7. Aufgaben
II. C
III. C++

4.2 Shellskripte

Shellskripte sind Programme, die direkt von dem Shell interpretiert werden. Sie beinhalten Shell-Befehle, die im Shell ausgeführt werden können. Sie fangen immer mit der Zeile

```
#!/bin/bash
```

Damit wird festgelegt, dass die Bash dieses Skript abarbeiten soll. Dann kann ein Programm in Skriptsprache geschrieben werden.
Namensexpandierung:

- *,? (Wildcards)
- 0123456789, abc

Quoting: Man unterscheidet 3 Formen:

- `'command $var'`
  Alles was zwischen `' '` steht, wird als ein String behandelt.
- `"command $var"`
  Alle Variablen die zwischen `" "` stehen, werden zunächst durch ihren Wert ersetzt und daraufhin wird der gesamte Ausdruck als String zurückgegeben.
- `` `command $var` ``
  Alle Variablen die zwischen ` `` ` stehen, werden zunächst durch ihren Wert ersetzt und daraufhin wird der gesamte Command-String zwischen den Quotes als Kommando interpretiert und ausgeführt.

```
Beispiel 1
> echo $$=pwd > welt
> cat welt
28850=pwd

Beispiel 2
> echo "$$=pwd > welt"
28850=pwd > welt

Beispiel 3
> echo '$$=pwd > welt'
$$=pwd > welt

Beispiel 4
> echo "$$=`pwd` > welt"
Directory: /home/petbec
28850= > welt
```

rtr Startseite
Einführung
Inhalt
1. Vorbemerkungen
I. Linux
2. Übersicht
3. Dateisystem
4. Shell
5. Programmieren unter Linux
5.1 Aufbau eines C/C++-Programm
5.2 Programme erstellen
5.3 Schritte der Programmübersetzung
5.4 Der GNU-C-Compiler
5.5 Buildsysteme
5.6 Testen und Debuggen
5.7 CVS
6. Literaturhinweise
7. Aufgaben
II. C
III. C++

5. Programmieren unter Linux

Programmiersprachen kann man unterteilen in Skriptsprachen und Compilersprachen.
Skriptsprachen wie Javascript, Perl oder Python werden nicht compiliert, sondern können von einem Interpreter direkt ausgeführt werden.
C und C++ sind Compiliersprachen. Der Code muß vor dem Ausführen zunächst in Objectcode umgewandelt (compiliert werden), bevor er ausgeführt werden kann. Dieser Objectcode (Maschinencode) ist plattformabhängig, aber schneller ausführbar als Code, der zur Laufzeit erst interpretiert werden muss.

- rtr Startseite
- Einführung
- Inhalt
- 1. Vorbemerkungen
- I. Linux
  - 2. Übersicht
  - 3. Dateisystem
  - 4. Shell
  - 5. Programmieren unter Linux
    - 5.1 Aufbau eines C/C++-Programm
    - 5.2 Programme erstellen
    - 5.3 Schritte der Programmübersetzung
    - 5.4 Der GNU-C-Compiler
    - 5.5 Buildsysteme
    - 5.6 Testen und Debuggen
    - 5.7 CVS
  - 6. Literaturhinweise
  - 7. Aufgaben
- II. C
- III. C++

5.1.1 Eigenschaften von C

- C ist eine funktionale Sprache
- integrierte Variablentypen (int, char, void, ...), strikte Typprüfung
- Erweiterbare Sprache: Einbinden von Bibliotheken

Aufbau eines Programms:

- mindestens die Funktion `int main()`
- Variablendefinitionen
- Funktionsaufrufe

Codebeispiel in C:

```
/* filename: hello_world.c */
#include <stdio.h>;
int main()
{
  printf("Hello world \n");
  return(0);
}
```

Vorteile von C

- schnelle Programme
- höhere Programmiersprache
- große Vielfalt von Bibliotheken verfügbar

Nachteile von C

- viele Fehlertypen sind nicht vom Compiler erkennbar
- zur Lesbarkeit sind eigene Stilkonventionen erforderlich
- viel Funktionalität ist nicht in der Sprache integriert

5.1.2 Eigenschaften von C++:

- enthält C (C89)
- objektorientierte Sprache
- enthält viele moderne Sprachelemente und Datenstrukturen

Codebeispiel in C++:

```
/* filename: hello_world.cpp */
#include <iostream>;
int main()
{
  cout << "Hello world" << endl;
  return(0);
}
```

Die Sprachkonzepte von C und C++ sind sehr verschieden!

rtr Startseite
Einführung
Inhalt
1. Vorbemerkungen
I. Linux
2. Übersicht
3. Dateisystem
4. Shell
5. Programmieren unter Linux
5.1 Aufbau eines C/C++-Programm
5.2 Programme erstellen
5.3 Schritte der Programmübersetzung
5.4 Der GNU-C-Compiler
5.5 Buildsysteme
5.6 Testen und Debuggen
5.7 CVS
6. Literaturhinweise
7. Aufgaben
II. C
III. C++

5.2 Programme erstellen

Editieren von Programmen:

- Programmcode erstellen
- Dokumentieren
- Debug-Code

Der Programmcode besteht aus Header- und Source-Dateien.
Source-Dateien:

- Endung: *.c (in C) oder *.cpp/*.cc (in C++)
- Definition der Funktionen, Implementierung

Header-Dateien:

- Endung: *.h (in C) oder *.h/*.hpp (in C++)
- Deklaration der Funktionen, also Spezifikation der Schnittstellen
- Code, den mehrere Sourcefiles benötigen, das ,,Inhaltsverzeichnis`` der Source-Files

Beispiel

```
/*******************************************************
 c-Datei:
 *******************************************************/
 #include "myfunc.h"
 #include <stdio.h>

float add(float a, float b) // Implementierung der Funktion add()
{
 return (a+b);
}

int main()
{
 float a,b,c;
 a = 10;
 b = 1.7;
 c = add(a,b);
 printf("Result: %f\n", c);
 return 0;
}


/*******************************************************
h-Datei: myfunc.h
*******************************************************/
#ifndef MYFUNC_H
#define MYFUNC_H

float add(float a, float b); // Prototyp der Funktion add()

#endif
```

rtr Startseite
Einführung
Inhalt
1. Vorbemerkungen
I. Linux
2. Übersicht
3. Dateisystem
4. Shell
5. Programmieren unter Linux
5.1 Aufbau eines C/C++-Programm
5.2 Programme erstellen
5.3 Schritte der Programmübersetzung
5.4 Der GNU-C-Compiler
5.5 Buildsysteme
5.6 Testen und Debuggen
5.7 CVS
6. Literaturhinweise
7. Aufgaben
II. C
III. C++

5.3 Schritte der Programmübersetzung

Zunächst werden die `.c` und `.h`-Dateien, die das Programm bilden sollen, editiert. Hierzu kann ein beliebiger Texteditor verwendet werden.

In jedem Linux-Betriebssystem kann der C-Compiler über den Befehl `cc` aufgerufen werden. Dies ist in der Regel eine Art Frontend, das den eigentlichen Compiler aufruft. Oft ist `cc` auch lediglich ein symbolischer Link auf den Compiler (siehe `ls -ll /usr/bin | grep cc`).

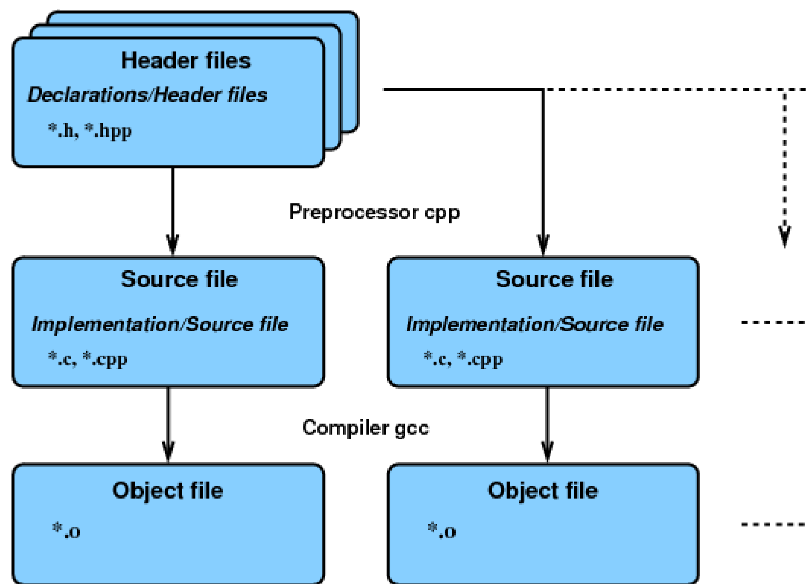
Unter Linux ist das meist verwendete Übersetzungstool der `gcc` (GNU-C-Compiler). Der `gcc` selber ruft beim Übersetzen zahlreiche weitere Programme auf und verarbeitet deren Ausgaben. Oft wird der Begriff `gcc` auch synonym für diese Programmsammlung (GNU-Buildsystem) verwendet.

Das Übersetzen läuft dann am Beispiel des GNU-Buildsystems wie folgt ab:

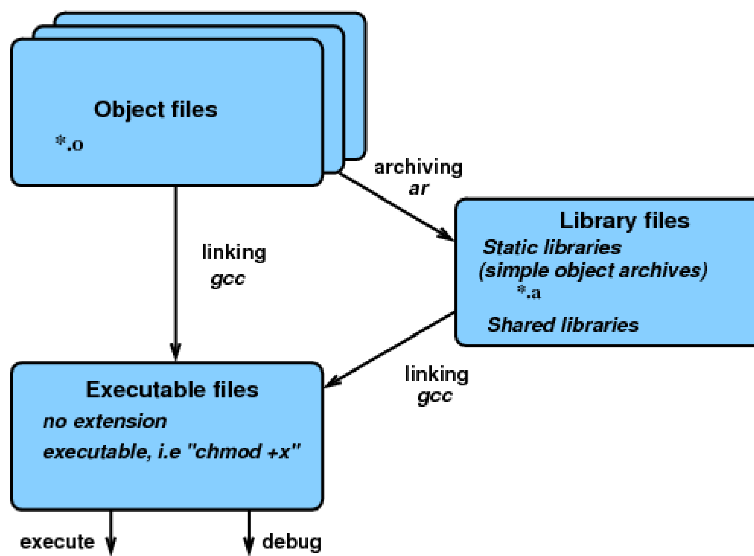
- **Phase 1**: *Präprozessor* `cpp`
  Jede einzelne `.c`-Datei wird zunächst vom Präprozessor verarbeitet. Dieser lädt zuerst alle durch `#include`-Anweisungen angegebenen Dateien zur `.c`-Datei hinzu und ersetzt anschließend alle vorkommenden Makros im Code durch die in den Makrodefinitionen (`#define`-Anweisungen) angegebenen Ausdrücke. Weiterhin werden alle Kommentare durch Leerzeichen ersetzt. In der so entstanden Ansammlung von Programmcode sind nun alle Informationen akkumuliert, die zum Übersetzen benötigt werden und aus reinem C-Code bestehen.
  Mit Hilfe des Präprozessors kann auch eine Anpassung an verschiedene Arbeitsumgebungen (Betriebssystem, Compiler) erfolgen.
- **Phase 2**: *Compiler* `cc / gcc` **bzw.** `g++` **für C++**
  Der vom Präprozessor ausgegebene C-Code kann nun vom eigentlichen Compiler in `Assembler-Code` übersetzt werden. Dies ist bei modernen Compiler ein sehr komplexer Prozess, da oftmals eine Optimierung hinsichtlich Laufzeit oder Größe unter sehr unterschiedlichen Nebenbedingungen und architekturspezifischen Besonderheiten (RICS, CISC, Befehlssätze, etc.) erwünscht ist.
  In der Regel ist der Assembler-Code für den Maschinentyp bestimmt, auf dem auch die Übersetzung stattfindet. Dies muss nicht so sein (Cross-Compiler).
- **Phase 3**: *Assembler* `as`
  Der GNU-Assembler `as` ersetzt schließlich den optimierten Code des Compilers mit dem eigentlichen Maschinen-Hex-Code der spezifischen Prozessor-Architektur. Dieser sogenannte Objekt-Code wird in einer Datei mit der Endung `.o` gespeichert. Diese Datei ist noch nicht ausführbar, da noch externe Abhängigkeiten aufgelöst werden müssen.
- **Phase 4**: *Linker* `ld`
  Der Linker bindet mehrere Objektdateien oder Archive zu einer ausführbaren Datei zusammen, indem er die bestehenden Symbol-Referenzen (Abhängigkeiten) auflöst und die relativen Adressen der Datenblöcke neu berechnet. Meist wird der selbst geschriebene Code gegen Standardbibliotheken gelinkt, in denen der Maschinencode der aufzurufenden Standard-C-Funktionen abgelegt ist.
  Im Gegensatz zu Windows-Programmen haben ausführbare Dateien unter Linux in der Regel keine Dateiendung.

Die einzelnen Schritte müssen glücklicherweise nicht per Hand ausgeführt werden, sondern werden weitestgehend selbständig vom `gcc` organisiert.

Phase 1 - Phase 3



Phase 4



rtr Startseite
Einführung
Inhalt
1. Vorbemerkungen
I. Linux
2. Übersicht
3. Dateisystem
4. Shell
5. Programmieren unter Linux
5.1 Aufbau eines C/C++-Programm
5.2 Programme erstellen
5.3 Schritte der Programmübersetzung
5.4 Der GNU-C-Compiler
5.5 Buildsysteme
5.6 Testen und Debuggen
5.7 CVS
6. Literaturhinweise
7. Aufgaben
II. C
III. C++

5.4 Der GNU-C-Compiler

Der `gcc` führt beim Übersetzen einer einzigen Quelldatei mit einer `main()`-Funktion die obigen Schritte vollkommen automatisch durch, so dass direkt die ausführbare Programmdatei ausgegeben wird. Über sogenannte `Flags` kann das Verhalten des Compiler den eigenen Wünschen entsprechend angepasst werden. Zum Beispiel erzeugt der `gcc` standardmäßig die ausführbare Datei `a.out`, was mit der Option `'-o Outputfilename'` auf den gewünschten Namen `Outputfilename` geändert wird.

Weitere Optionen legen die Sprache und Sprachdialekte wie z.B. ANSI C, C89, C99, etc. fest, andere wiederum die Codeoptimierung, den Prozessortyp oder weitere Suchpfade für Header-Dateien und Bibliotheken. Siehe hierzu die Man-Page zu `gcc`.

Benutzung:

```
gcc [OPTIONEN] [FILES]
```

Hauptoptionen:

```
-o outfile       erzeugt ein Executable namens "outfile"
-ansi            schaltet in den ANSI-C-Modus
-std=c99         prüft auf Kompatibilität zum Sprachstandard C99
-pedantic        besonders genaues Prüfen des C-Programmes
-I/usr/local/include
                 sucht Headerfiles zusaetzlich in dem Verzeichnis
                 "/usr/local/include"
-L/usr/local/lib
                 sucht Objektfiles und Libraries zusätzlich
                 in dem Verzeichnis "/usr/local/lib"
-lmylib          linkt die Objektdateien gegen
                 die Bibliotheksdatei libmylib.so
-g               Übersetzung mit Debug-Informationen für späteres
                 Debuggen
-O / -O1 / -O2   verschiedene Optimierungsmöglichkeiten
-v               zeigt Zusatzinformationen an
-Wall            berichtet alle möglichen Warnungen
--version        zeigt die Version des gcc an
```

Falls das Projekt mehrere Quelldateien besitzt, die alle zu einem Programm gebunden werden sollen, ist das schrittweise übersetzen der einzelnen Quelldateien mit einem anschließenden Linkprozess der entsprechenden Objektmodule notwendig. Der `gcc` kann dafür veranlaßt werden, bei einem bestimmten Schritt der Programmübersetzung abzubrechen und statt dem fertigen Programm z.B. den Assembler-Code oder Objekt-Code auszugeben.

Optionen zur Ausgabe:

```
-E    stoppt nach der Präprozessor-Phase, das Ergebnis ist reiner C-Code
-S    stoppt nach dem Kompilieren, das Ergebnis ist Assembler-Code
-c    stoppt nach dem Assemblieren, das Ergebnis ist Objekt-Code
```

Normalerweise werden diese Einstellungen in einem sog. `Makefile` vorgenommen, das von dem im nächsten Abschnitt behandelten `make`-Programm verarbeitet wird.

rtr Startseite
Einführung
Inhalt
1. Vorbemerkungen
I. Linux
2. Übersicht
3. Dateisystem
4. Shell
5. Programmieren unter Linux
5.1 Aufbau eines C/C++-Programm
5.2 Programme erstellen
5.3 Schritte der Programmübersetzung
5.4 Der GNU-C-Compiler
5.5 Buildsysteme
5.6 Testen und Debuggen
5.7 CVS
6. Literaturhinweise
7. Aufgaben
II. C
III. C++

5.5 Buildsystem: make

Es gibt verschiedene Ansätze um den oben beschriebenen Ablauf zum Bauen von Programmen zu automatisieren. Dazu verwendet man in der Regel sog. Makefiles, die von dem Programm make interpretiert werden. make ist somit eine Art Interpretersprache.
Arbeitsweise von make:

- inkrementelles Bauen: Regeln basieren auf Änderungsdatum
- Ziel: Übersetzungszeit verkürzen
- Darstellung: Abhängigkeitsgraph
- Ziele, Abhängigkeiten, Regeln
- Variablen (Bsp.: CC = cc, $(CC), make CC=gcc)
- Tücken: Tabulator, datumsbasiert, mehrfache Abhängigkeiten

für Fortgeschrittene:

- implizite Regeln (Bsp.: .c.o:)
- Musterregeln Bsp.: (%.o: %.c)
- Makefiles für andere Befehle (Bsp.: make install)

Beispiel für ein Makefile

```
example1_2:example1_2.o libEx.a
   gcc example1_2.o -L. -lEx -o example1_2

example1_2.o:
   gcc -g -c example1_2.c

libEx.a:
   gcc -g -c head.c
   ar -rv libEx.a head.o
```

5.5.1 Komplexere Buildsysteme

Da das Schreiben aufwendigerer Makefiles recht aufwendig ist, kann man diesen Prozess ebenfalls automatisieren. Dazu werden meist die Tools autoconf, automake und libtool verwendet.
autoconf:

- autoconf erzeugt das Konfigurationsskript configure automatisch aus Systemtests
- configure führt Systemstests aus und erstellt auf der Basis von Vorlagen in jedem Unterverzeichnis (Makefile.in) Makefiles (ebenfalls für jedes Unterverzeichnis)
- autoconf ist durch eigene Tests erweiterbar

automake:

- automake erstellt die Vorlagen für autoconf (Makefile.in) aus einfach editierbaren Files: Makefile.am

libtool:

- libtool ist ein Frontend für den Compiler
- libtool weiß von welchen Programmen und mit welchen Optionen auf verscheidenen Systemen mit verschiedenen Compilern Programme compiliert und gelinkt werden.

Inzwischen verwenden fast alle OpenSource-Projekte ein Buildsystem aus autoconf/automake/libtool/make. Die Software, die man frei aus dem Internet herunterladen kann, läßt sich dann mit folgender Befehlssequenz compilieren und installieren:

```
./configure
make
make install
```

Mit dem Aufruf von ./configure --help bekommt man eine Übersicht über die Optionen des jeweiligen configure-Skripts.
Zum Erstellen des Buildvorgangs sind die Tools autoconf, automake und libtool notwendig. Sind einmal das configure-Skript und die Makefile.in-Files vorhanden, benötigt man sie nicht mehr.

TECHNISCHE UNIVERSITÄT DARMSTADT

Fachgebiet Regelungstheorie und Robotik

Impressum | Druckversion

REGELUNGSTHEORIE UND ROBOTIK rtr

rtr Startseite
Einführung
Inhalt
1. Vorbemerkungen
I. Linux
2. Übersicht
3. Dateisystem
4. Shell
5. Programmieren unter Linux
5.1 Aufbau eines C/C++-Programm
5.2 Programme erstellen
5.3 Schritte der Programmübersetzung
5.4 Der GNU-C-Compiler
5.5 Buildsysteme
5.6 Testen und Debuggen
5.7 CVS
6. Literaturhinweise
7. Aufgaben
II. C
III. C++

5.6 Testen und Debuggen

Debuggen (gdb, ddd):

- Anwendung: Fehlersuche (systematisches Aufstellen und Prüfen von Hypothesen)
- schrittweise kontrollierte Ausführung des Programms
- Überwachung von Variablen
- bedingte Programmablaufsteuerung
- graphische Darstellung von Speicherinhalten

Testen:

- eigene Testprogramme
- Unit-Tests mit Unterstützung durch Bibliotheken: cppunit, boost
- z. B.: Memory Leak Checks: valgrind (OpenSource), libefence, purify, Insure, splint

5.6.1 graphische Programmierumgebungen

- KDevelop, Sniff+
- integrierter Zugriff auf die Programmierwerkzeuge und Source-Editoren

TECHNISCHE UNIVERSITÄT DARMSTADT

rtr Startseite
Einführung
Inhalt
1. Vorbemerkungen
I. Linux
2. Übersicht
3. Dateisystem
4. Shell
5. Programmieren unter Linux
5.1 Aufbau eines C/C++-Programm
5.2 Programme erstellen
5.3 Schritte der Programmübersetzung
5.4 Der GNU-C-Compiler
5.5 Buildsysteme
5.6 Testen und Debuggen
5.7 CVS
6. Literaturhinweise
7. Aufgaben
II. C
III. C++

5.7 CVS

CVS = Concurrent Version System

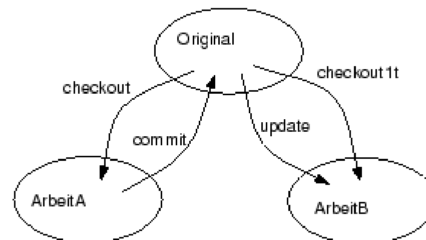
- Versionierung von ganzen Dateibäumen
- Paralleles Arbeiten möglich
- arbeitet auf Textfiles und vergleicht zeilenweise
- Konflikte bei Änderung an gleicher Stelle: Files werden mit merge vereinigt. Wenn 2 Zeilen gleich sind, bleiben beide Versionen erhalten und werden markiert. Der Konflikt muß dann von Hand gelöst werden.

Vorteile:

- Kontrolle über verschiedene Versionen
- Wiederherstellen von ganzen Konfigurationen
- keine Blockaden
- cvs ist ein weitverbreitetes System

Nachteile:

- hoher Plattenplatzbedarf (vor allem bei binären Dateien)
- Konflikte möglich (von Hand aufzulösen)



- cvs import, cvs add: Daten unter CVS-Kontrolle stellen
- cvs checkout: Lokale Kopie
- cvs commit: Einchecken, Log-Eintrag wird abgefragt und ist später einsehbar
- cvs update: Lokales Verzeichnis auf neuesten Stand bringen
- cvs log: Ansehen der Logs
- cvs annotate: Anzeige einer Datei mit Angabe, von wem und aus welchem Version jede Zeile stammt
- cvs tag: Vergabe eines Namens für ein Set von Dateien, evtl. Starten einer neuen, parallelen Entwicklungslinie (branch)
- cvs add: Hinzufügen einer Datei aus dem Repository
- cvs remove: Entfernen einer Datei aus dem Repository, Geschichte der Datei bleibt erhalten

rtr Startseite
Einführung
Inhalt
1. Vorbemerkungen
I. Linux
2. Übersicht
3. Dateisystem
4. Shell
5. Programmieren unter Linux
6. Literaturhinweise
7. Aufgaben
II. C
III. C++

6. Literaturhinweise

1. Unix, eine Einführung in die Benutzung. Vom RRZN der Universität Hannover. 12. Auflage, 1998.
2. Zeller und Krinke: Programmierwerkzeuge. dpunkt-verlag, ISBN 3-932588-70-3
3. Der UNIX-Werkzeugkasten - Programmieren mit UNIX, von B. Kernighan und R. Pike. Carl Hanser Verlag, 1984.
4. Unix, eine Einführung in die Benutzung. Vom RRZN der Universität Hannover. 12. Auflage, 1998.
5. UNIX System V - Professionelles Programmieren von R. Thomas, L. Rogers und J. Yates. OsborneMcGrawHill Verlag, 1987.
6. Die Programmiersprache C: Ein Nachschlagewerk. Vom RRZN der Universität Hannover. 12. Auflage, 1996.
7. Numerical Recipes in C - The Art of Scientific Computing. Von W. Press, B. Flannery, S. Teukolsky und W. Vetterling. Cambridge University Press, 1984.
8. C - the Complete Reference. Von H. Schildt. OsborneMcGrawHill Verlag, 1990.
9. Advanced Programming in the UNIX Environment. Von W. Stevens. Addison-Wesley, 1993.
10. Wieland, Thomas: C++ Entwicklung mit Linux. dpunkt-verlag, ISBN 3-932588-74-6
11. Stroustroup, Bjarne: Die C++ Programmiersprache. 4. aktualisierte und erweiterte Auflage, Studentenausgabe. Addison-Wesley, 2000, Boston. ISBN 3-8273-1756-8 DM 69,90 (paperback)
    Originalausgabe: Stroustroup, Bjarne: The C++ Programming Language, Special Edition. Addison-Wesley, 2000, Boston. ISBN 0-201-70073-5
12. Schildt, Herbert: C++: The Complete Reference, Third Edition. Que Corp., 1999. ISBN 0-7897-2022-1. $39.99, ca. 80 DM (paperback)
13. Meyers, Scott: Effektiv C++ programmieren. 50 Wege zur Verbesserung Ihrer Programme und Entürfe. Addison-Wesley, 1998. ISBN 3-8273-1305-8
    Originalausgabe: Meyers, Scott: Effective C++ ISBN 0-201-92488-9
14. Meyers, Scott: Mehr Effektiv C++ programmieren. 35 neue Wege zur Verbesserung Ihrer Programme und Entwürfe.
    Addison-Wesley, 1999. ISBN 3-8273-1275-2
    Originalausgabe: Meyers, Scott: More Effective C++ ISBN 0-201-63371-x
15. Kalev, Danny: The ANSI/ISO C++ Professional Programmer's Handbook. McGraw-Hill, 1998. ISBN 0-07-882476-1. $39.99, ca. 90 DM (paperback)
16. Dalheimer, M.K., Programming with Qt. O'Reilly, 2002, 2. Auflage. ISBN 0596000642, 48,49 EUR
17. Sweet, David: KDE-2-Programmierung. komponentenbasierte Anwendungen mit Qt, kparts und DCOP. Markt-und-Technik-Verl., 2001, 576 S. ISBN 3-8272-5980-0 erhähltlich bei www.terrashop.de für 13 EUR.
    Originalausgabe: David Sweet, Ph.D., et. al.: KDE 2.0 Development. Guide to KDE development. Macmillan/SAMS, ISBN: 0672318911

rtr Startseite
Einführung
Inhalt
1. Vorbemerkungen
I. Linux
2. Übersicht
3. Dateisystem
4. Shell
5. Programmieren unter Linux
6. Literaturhinweise
7. Aufgaben
7.1 Example 1
7.2 Example 2
7.3 Example 3
7.4 Example 4
7.5 Example 5
II. C
III. C++

7.1 Example 1

In our first example we want to use the editor (kate) and do these steps:

1. change the working directory to `~/example1_1` (You may have to `mkdir ~/example1_1` first!)
2. start kate (type kate) at the terminal prompt
3. type the following lines:

```
/* filename: hello_world.c */
#include <stdio.h>
int main(void)
{
  printf("Hello world \n");
  return(0);
}
```

4. save file by typing `ctrl-s` and specify the filename `hello_world.c`. Quit kate with `ctrl-q`; (You simply can use the menu `file` on top of kate instead, of course).
5. type: `gcc -o hello_world hello_world.c`;
6. type `ls` to list the directory structure you will see the files hello_world.c (Source file) and hello_world (which should be the executable file)
7. if necessary, type `chmod +x hello_world` to make the file `hello_world` executable
8. type `hello_world` and hit return to start the programm. On the screen you see the result.

rtr Startseite
Einführung
Inhalt
1. Vorbemerkungen
I. Linux
2. Übersicht
3. Dateisystem
4. Shell
5. Programmieren unter Linux
6. Literaturhinweise
7. Aufgaben
7.1 Example 1
7.2 Example 2
7.3 Example 3
7.4 Example 4
7.5 Example 5
II. C
III. C++

7.2 Example 2

In the next example we want to compile 2 files together

1. `mkdir ~/example1_2; cd example1_2` then start kate at the terminal prompt
2. type the following lines:

```
/* filename: head.c  */
#include <stdio.h>
int head()
{
  printf("in procedure head \n");
  return(0);
}
```

3. save the file by typing `ctrl-s` and specify the filename `head.c` and quit kate.
4. start kate again at the terminal prompt (Instead of quitting and restarting, you can simply open a new file by `ctrl-o` or just use the menu.)
5. type the following lines:

```
/* filename: example1_2.c */
#include <stdio.h>
int main(void)
{
  printf("in procedure main before calling procedure head \n");
  head();
  printf("in procedure main after calling procedure head \n");
  return(0);
}
```

6. type `gcc -o example1_2 head.c example1_2.c`
7. type `ls` and you will see the files: head.c, example1_2.c, and the file example1_2, the executable If you type `ls -alrt`, you get a detailed list of all files, with the newest files at the end).
8. type `chmod +x example1_2`
9. type `./example1_2` and hit the return -key to start the executable. You will see 3 lines of text which show the chain in which the 2 procedures are called.

rtr Startseite
Einführung
Inhalt
1. Vorbemerkungen
I. Linux
2. Übersicht
3. Dateisystem
4. Shell
5. Programmieren unter Linux
6. Literaturhinweise
7. Aufgaben
7.1 Example 1
7.2 Example 2
7.3 Example 3
7.4 Example 4
7.5 Example 5
II. C
III. C++

7.3 Example 3

In this example we will create a library and insert the function head which is defined in the object-file head.o (see example 1_2 (step6)).

1. remain in `~/example1_2`. Type `gcc -c head.c`. Then look at the files created (`ls -alrt`). What are these *.o files?
2. Create a library typing ar `rv libEx.a head.o`. The library contains the object-file Ex now.
3. now list the contents of the library by typing `ar tv libEx.a`.
4. now delete the file example1_2 by typing `rm example1_2`. If you are asked, if you really want to remove the file type `y` to confirm the remove
5. now delete the file head.o by typing `rm head.o`. If you are asked, if you really want to remove the file type `y` to confirm the remove
6. check with `ls` what happend to the files.
7. it is time to compile the file again. at this time use: `gcc -o example1_3 example1_2.c libEx.a` (You can also use the more sophisticated version `gcc example1_2.c -L. -lEx -o example1_3`).
8. see with `ls` which files are created

rtr Startseite
Einführung
Inhalt
1. Vorbemerkungen
I. Linux
2. Übersicht
3. Dateisystem
4. Shell
5. Programmieren unter Linux
6. Literaturhinweise
7. Aufgaben
7.1 Example 1
7.2 Example 2
7.3 Example 3
7.4 Example 4
7.5 Example 5
II. C
III. C++

7.4 Example 4

In this example we will create a mechanism for automation, which is a file called `Makefile`.

1. start kate
2. type:

```
example1_4:example1_2.o head.o
   gcc example1_2.o head.o -o example1_4

example1_2.o: example1_2.c

head.o: head.c
```

   Remember to use tabs instead of spaces!

3. save file by typing `ctrl-s` and specify the filename `Makefile`. Then quit kate.
4. now delete the file example1_3 by typing rm example1_3. If you are asked, if you really want to remove the file type `y` to confirm the remove.
5. check the result by typing `ls`
6. type `make`
7. check the result by typing `ls`

rtr Startseite
Einführung
Inhalt
1. Vorbemerkungen
I. Linux
2. Übersicht
3. Dateisystem
4. Shell
5. Programmieren unter Linux
6. Literaturhinweise
7. Aufgaben
7.1 Example 1
7.2 Example 2
7.3 Example 3
7.4 Example 4
7.5 Example 5
II. C
III. C++

7.5 Example 5

In this example we will use a debugger to search for possible bugs.

1. start kate and open the Makefile
2. replace the rules to build example1_2.o and head.o by the following lines to write debuggerinformation in your executables and rename example1_3 to example1_4:

```
example1_4:example1_2.o head.o
   gcc example1_2.o head.o -o example1_4

example1_2.o:
   gcc -g -c example1_2.c

head.o:
   gcc -g -c head.c
```

3. save file by typing `ctrl-s` and quit kate.
4. now delete the files example1_4, example1_2.o and head.o by typing `rm example1_4 example1_2.o head.o`. If you are asked, if you really want to remove the file type `y` to confirm the remove.
5. check the result by typing `ls`.
6. type `make`.
7. check the result by typing `ls`.
8. start the debugger DDD by typing `ddd` at the terminal prompt.
9. Choose option `open program` in the menu `file` on top of `ddd`. Choose `example1_4` on the list.
10. Add a breakpoint in line 5 choosing option `breakpoints... ; add break` in the menu `source` or simply clicking right mouse button on line 5 and choose `set breakpoint`.
11. Run the program choosing option `run` in the menu `program` and see what happens.

- rtr Startseite
- Einführung
- Inhalt
- 1. Vorbemerkungen
- I. Linux
- II. C
  - 8. Ein C-Programm
  - 9. Variablen / Datentypen
  - 10. Wichtige Sprachelemente
  - 11. Funktionen
  - 12. Benutzerdefinierte Typen
  - 13. Übersetzen von C-Programmen
  - 14. Arrays
  - 15. Zeiger
  - 16. Aufgaben
- III. C++

Vorbemerkungen

Aufbau

Kapitel 8 bis 13 beschäftigt sich mit den Grundlagen der Sprache C wie Variablendeklarationen, Schleifen, Funktionsdeklarationen und -aufrufen. Es wird auch kurz auf den Übersetzungsprozeß eines Programmes mit Hilfe des make-Utilities eingegangen, der für eine modulare Programmierung wesentlich ist. In den Kapiteln 14 und 15 werden weiterführende Merkmale der Sprache behandelt; u. a. wird die Verwendung und Verwaltung von Zeigern an dieser Stelle angesprochen. Im Anhang finden sich die zwei Übungsbeispiele

- ,,Signalgenerator`` und
- ,,Matrizenverwaltung mit Doppelzeigern``.

Sie sind charakteristisch für viele Problemstellungen, denen sich ein Programmierer an unserem Fachgebiet gegenübersieht.

Literatur

Die Programmiersprache C ist sehr eng mit dem Betriebssystem UNIX und dessen Derivate wie etwas dem freien Nachfolger Linux verbunden*. Die Literaturhinweise umfassen aus diesem Grund generelle Werke zum Programmieren unter UNIX , Einführungsmaterial in die C-Programmierung sowie ein Werk zum wissenschaftlichen Programmieren

---

*) Der Linux-Kernel und die Geräte-Treiber sind bis auf kleine Ausnahmen in C programmiert. Der Assembler-Anteil liegt bei <5%.

rtr Startseite
Einführung
Inhalt
1. Vorbemerkungen
I. Linux
II. C
8. Ein C-Programm
9. Variablen / Datentypen
10. Wichtige Sprachelemente
11. Funktionen
12. Benutzerdefinierte Typen
13. Übersetzen von C-Programmen
14. Arrays
15. Zeiger
16. Aufgaben
III. C++

8. Ein C-Programm

```
#include <stdio.h>    /* Einbinden von Definitionen fuer Standard Ein-/Ausgabe */

#define PI     3.14159                          /* Definition von Konstanten */
#define N     10

int main (void)                     /* jedes Programm besitzt eine Funktion main */
{                                                       /* hier beginnt ein Block */
  int i, s ;             /* Definition von integer Variablen mit Namen i und s */
  float f ;                     /* Definition einer 32 Bit Fliesskomma Variablen */
  double d[N] ;    /* Definition eines Felds von 64 Bit Fliesskomma Variablen */

  f = PI ;                                                     /* Zuweisungen */
  s = 0 ;

  for (i=0; i<N; i++)                                         /* eine Schleife */
  {                                             /* Schleifenrumpf: eigener Block */
    d[i] = 0.0 ;                    /* Zuweisung von 0.0 an das i-te Feldelement */
    s += 10 ;                                  /* Inkrementieren von s um 10 */
  }                                      /* hier endet der Block (for-Schleife) */

  printf("Hallo C-Programmierer!\n") ;      /* formatierte Ausgabe mit printf */
  printf("s = %d  f = %3.2f\n", s, f) ;

  return (0) ;                               /* dies ist _hier_ nicht notwendig */
}                                                 / * hier endet der Block (main) */
```

Bemerkungen:

- Ein Programm besteht aus mehreren *Funktionen*; jedes Programm besitzt mindestens die Funktion `main`. Diese wird beim Programmstart zuerst aufgerufen.
- Die Anweisungen werden mit einem Semikolon abgeschlossen.
- Zusammengehörende Anweisungen werden in Blöcken `{ ... }` zusammengefaßt. Aus Gründen der Übersichtlichkeit sollte man die Blöcke einrücken!
- Ein-/Ausgabe Anweisungen sind nicht Bestandteil der Sprache, jedoch stellt die Standard-Bibliothek viele Funktionen hierzu bereit, z.B. die Funktion `printf`. Um diese nutzen zu können, muß der entsprechende Header `stdio.h` per `#include` eingebunden werden (--> *Präprozessor*).

rtr Startseite
Einführung
Inhalt
1. Vorbemerkungen
I. Linux
II. C
8. Ein C-Programm
9. Variablen / Datentypen
10. Wichtige Sprachelemente
11. Funktionen
12. Benutzerdefinierte Typen
13. Übersetzen von C-Programmen
14. Arrays
15. Zeiger
16. Aufgaben
III. C++

9.1 Variablendefinitionen

Vor ihrer Verwendung müssen Variablen definiert werden. (Auf den formalen Unterschied zwischen Deklaration und Definition gehen wir hier nicht ein.) Eine solche Definition hat die Form

```
typename  varname ;
```

varname ist ein beliebiger Name, der für diese Variable vergeben wird. Der Name darf natürlich nicht mit einem Schlüsselwort der Sprache kollidieren. typename steht für den Datentyp der Variable. Es kann sich sowohl um einen Basistyp, als auch um selbstdefinierte Typen handeln.
Basistypen sind:

`int`	ganzzahlige Größe, Maschinenwortbreite (16 oder 32 Bit)
`float`	Fließkommazahl einfacher Genauigkeit (32 Bit)
`double`	Fließkommazahl doppelter Genauigkeit (64 Bit)
`char`	ein (ASCII)-Zeichen, also ein Byte

Aus jedem Typ entsteht der entsprechende *Zeigertyp*, wenn dem Variablenname bei der Definition ein * vorangestellt wird.

```
int  *i_ptr ;
```

Ein Zeiger (pointer) stellt die Adresse einer Variablen dar. Darauf wird später noch eingegangen (Abschnitt 15).
Variablendefinitionen stehen immer am Blockanfang. Die *Sichtbarkeit* von Variablen erstreckt sich auf den Block, an dessen Anfang die Definition steht, und auf dessen untergeordneten Blöcke. In übergeordneten Blöcken ist die entsprechende Variable jedoch nicht sichtbar.

9.2 Typkonversionen

Zuweisungen einer Variablen vom Typ T1 an eine Variable vom Typ T2 sind in C erlaubt. Hierbei muß allerdings eine Anpassung an den Typ der linken Seite der Zuweisung erfolgen (*type casting*). Dies kann entweder *implizit*, d.h. automatisch durch den Compiler, oder *explizit* durch den Programmierer geschehen. Implizite Typkonvertierungen sind fehleranfällig uns sollten, wenn möglich in ihrer expliziten Form gemacht werden. Explizite Typkonversionen haben folgendes Format:

```
varOfTypeT2 = (T2)varOfTypeT1;
```

Implizite Typkonversionen werden vom Compiler nach bestimmten Regeln vorgenommen; steht auf der rechten Seite einer Zuweisung ein arithmetischer Ausdruck, so werden alle Operanden auf den ,,größten`` auftretenden Datentyp konvertiert und dann der Ausdruck berechnet. Das Ergebnis wird auf den Typ der linken Seite der Zuweisung konvertiert. Beim Umwandeln von Fließkommatypen in ganzzahlige Typen werden die Nachkommastellen abgeschnitten.
Beispiel:

```
int i,j,k;
float a,b,c;

i=2; j=3;
a=4; b=5;

k = i/b;  /* i/b wird in float-Arithmetik berechnet, das Ergebnis wird nach
           * int konvertiert. Demnach hat k anschliessend den Wert 0
           */

c = i/b;  /* wie oben, jedoch entfaellt die Konversion des Ergebnisses
           * c erhaelt den Wert 0.4
           */

c = j/i;  /* Die beiden Operanden j und i sind int-Variablen, es wird erst eine
           * eine Ganzzahl-Division durchgefuehrt. Das Ergenis ist 1 und wird
           * nach float konvertiert. c erhaelt also den Wert 1.0
           */
```

rtr Startseite
Einführung
Inhalt
1. Vorbemerkungen
I. Linux
II. C
8. Ein C-Programm
9. Variablen / Datentypen
10. Wichtige Sprachelemente
10.1 Ein-/Ausgabe mit printf, scanf
10.2 Verzweigungen
10.3 Schleifen
10.4 Operatoren
11. Funktionen
12. Benutzerdefinierte Typen
13. Übersetzen von C-Programmen
14. Arrays
15. Zeiger
16. Aufgaben
III. C++

10.1 Ein-/Ausgabe mit printf, scanf

Die Funktionen scanf und printf stellen die Schnittstelle eines Programms zu den Standardein- bzw. -ausgabemedien dar (Tastatur und Bildschirm, es sei denn, die Standard Ein-/Ausgabe wird beim Programmstart explizit ,,umgebogen``).
Grundsätzlich werden Funktionen aufgerufen duch Angabe des Funktionsnamens (hier also printf bzw. scanf) und eine Liste von *Parametern* (siehe Abschnitt 11).
Bei printf bzw. scanf ist dies der *Formatstring*, der beliebige Texte und Konvertierungszeichen – falls Werte von Variablen aus- bzw. eingegeben werden sollen – enthält. Danach werden die entsprechenden Variablen angegeben.

```
int i ; float f ;  char name[10]
i = 1 ; f = 1.234 ;

scanf("%d", &i) ;
scanf("%f", &f) ;
scanf("%s", name) ;

printf(" Die Werte waren %d und %f\n", i, f) ;
```

Die Konvertierungszeichen müssen zu den Datentypen der Variablen passen! Wichtige Konvertierungszeichen sind:

%d	`int` Variablen
%f	`float` Variablen
%lf	`double` Variablen
%c	`char` Variablen
%s	`char` Felder (Strings)

Der Formatstring enthält außerdem Steuerzeichen, etwa \n für das Zeilenende (carriage return).
Bei scanf ist unbedingt zu beachten, daß die Adressen der Variablen übergeben werden (call by reference, siehe Abschnitt 11.4). Dies geschieht bei Feldern wie dem String name in obigem Beispiel duch Angabe des Feldnamens , bei ,,normalen`` Variablen muß die Adresse durch den Operator & explizit ermittelt werden.

rtr Startseite
Einführung
Inhalt
1. Vorbemerkungen
I. Linux
II. C
8. Ein C-Programm
9. Variablen / Datentypen
10. Wichtige Sprachelemente
10.1 Ein-/Ausgabe mit printf, scanf
10.2 Verzweigungen
10.3 Schleifen
10.4 Operatoren
11. Funktionen
12. Benutzerdefinierte Typen
13. Übersetzen von C-Programmen
14. Arrays
15. Zeiger
16. Aufgaben
III. C++

10.2.1 if-else Anweisung

Abhängig davon ob ein Ausdruck ,,wahr`` oder ,,falsch`` ist wird entweder der if- oder der else-Block ausgeführt. Der else-Teil muß nicht unbedingt vorhanden sein. Verschachtelungen sind natürlich möglich.

```
if (i>0)    {  /* Anweisungen */  }
else        {  /* Anweisungen */  }
```

10.2.2 switch Anweisung

Will man in Abhängigkeit des Wertes eines Ausdrucks (z.B. einer Variablen) verschiedene Blöcke ausführen, bietet sich die switch Anweisung anstelle vieler if's an.

```
switch (i)
{
  case 1:
            /* Anweisungen */
  break ;

  case 2:
            /* Anweisungen */
  break ;

  default : /* Anweisungen */
}
```

Der default-Zweig erlaubt, auf nicht vorhergesehene Werte zu reagieren.
Wichtig: Nach jedem case-Block muß unbedingt ein break stehen, sonst werden alle folgenden Blöcke ebenfalls ausgeführt!

rtr Startseite
Einführung
Inhalt
1. Vorbemerkungen
I. Linux
II. C
8. Ein C-Programm
9. Variablen / Datentypen
10. Wichtige Sprachelemente
10.1 Ein-/Ausgabe mit printf, scanf
10.2 Verzweigungen
10.3 Schleifen
10.4 Operatoren
11. Funktionen
12. Benutzerdefinierte Typen
13. Übersetzen von C-Programmen
14. Arrays
15. Zeiger
16. Aufgaben
III. C++

10.3.1 while-Schleife

```
int i ;
i = 0 ;
while (i < 10)  {    /*Anweisungen */    }
```

Hierbei handelt es sich um einen *pre-checked loop*, d.h. die Abbruchbedingung wird *vor* dem Eintritt in die Schleife geprüft. Innerhalb des Schleifenrumpfs muß natürlich eine Anweisung stehen, die Einfluß auf das Abbruchkriterium hat, hier also i hochzählt.

10.3.2 for-Schleife

Eine for-Schleife beginnt mit dem Schlüsselwort for gefolgt von
(Initialisierung ; Abbruchkriterium ; Weiterschaltanweisung).
Dies ist ebenfalls ein *pre-checked loop.* Der Operator **++** führt zum Inkrementieren der Variablen k um 1. Prinzipiell können aber beliebige Weiterschaltanweisungen angegeben werden.

```
int k ;
for (k=1; k<=10; k++) {    /* Anweisungen */   }
```

10.3.3 do-Schleife

Bei der do-Schleife handelt es sich um einen *post-checked loop*, d.h. die Abbruchbedingung wird nach dem ersten Schleifendurchlauf erstmals geprüft. Auch hier muß der Schleifenrumpf natürlich explizit eine Anweisung enthalten, die sich auf das Abbruchkriterium auswirkt.

```
int m ;
m = 17 ;
do   {     /* Anweisungen */     }   while (m!=0) ;
```

Der Operator != bedeutet ,,ungleich``.
Im Zusammenhang mit Schleifen sind die Anweisungen break und continue zu erwähnen. break führt zum sofortigen Verlassen der Schleife. Unabhängig davon, ob die Abbruchbedingung erfüllt ist, erfolgt der Sprung zur ersten Anweisung nach dem Schleifenrumpf.
Bei for-Schleifen (und nur bei diesen) führt die continue-Anweisung dazu, daß der aktuelle Durchlauf abgebrochen wird. Es wird dann zunächst die Abbruchbedigung geprüft, ggf. die Weiterschaltanweisung ausgeführt und der nächste Durchlauf beginnt.

rtr Startseite
Einführung
Inhalt
1. Vorbemerkungen
I. Linux
II. C
8. Ein C-Programm
9. Variablen / Datentypen
10. Wichtige Sprachelemente
10.1 Ein-/Ausgabe mit printf, scanf
10.2 Verzweigungen
10.3 Schleifen
10.4 Operatoren
11. Funktionen
12. Benutzerdefinierte Typen
13. Übersetzen von C-Programmen
14. Arrays
15. Zeiger
16. Aufgaben
III. C++

10.4 Operatoren

Die wichtigsten Operatoren sind
Vergleichsoperatoren:

`==`	gleich (Achtung: Nicht mit = vertauschen!)
`>`	größer
`>=`	größer oder gleich
`<`	kleiner
`<=`	kleiner oder gleich
`!=`	ungleich

Logische Operatoren:

`&&`	logisches `UND`
`||`	logisches `ODER`
`!`	logische Negation

Arithmetische Operatoren:

`*`	Multiplikation
`/`	Division
`+`	Addition
`-`	Subtraktion

Operatoren zur bitweisen Verknüpfung:

`&`	bitweises `UND`
`|`	bitweises `ODER`
`~`	bitweise Negation

Sonstige Operatoren:

`=`	Zuweisung (Achtung: Nicht mit `==` vertauschen!)
`++`	Inkrement
`-`	Dekrement
`&`	Adresse
`*`	Dereferenzierung eines Zeigers

Bemerkungen:

- Die Inkrement-/Dekrement-Operatoren können vor- oder nachgestellt werden; in Verbindung mit anderen Operatoren oder in logischen Ausdrücken ergeben sich dann unterschiedliche Bedeutungen:

  ```
  int m = 0 ;
  do   {   /* Anweisungen */    }  while  (m++ < 100) ;
  ```

  Hier wird z. B. *erst* der Vergleich ausgeführt und dann `m` erhöht. Hätte man `++ m` geschrieben, wäre die Reihenfolge umgekehrt.

- Die arithmetischen Operatoren `+`, `-`, `*`, `/` sowie die Operatoren zur Bitverknüpfung `&`, `|`, `~` können jeweils mit dem Zuweisungsoperator kombiniert werden. Z. B. führt

  ```
  int i, j, k;

  i+= 10 ;
  j |= k ;
  ```

  zum Inkrementieren von `i` um 10 und zur Zuweisung der bitweisen oder-Verknüpfung von `j` und `k` zurück an die Variable `j`.

- Der `*` Operator steht sowohl für die Dereferenzierung als auch für die Multiplikation. Der Compiler erkennt aus dem Kontext, was gemeint ist.

- Die Operatoren haben verschiedene Prioritäten. Hierzu wird auf die Literatur verwiesen. Im Zweifelsfall sollte man durch Klammerung die Reihenfolge eindeutig festlegen.

rtr Startseite
Einführung
Inhalt
1. Vorbemerkungen
I. Linux
II. C
8. Ein C-Programm
9. Variablen / Datentypen
10. Wichtige Sprachelemente
> 11. Funktionen
> 11.1 Funktionsprototypen
11.2 Funktionsdefinition
11.3 Funktionsaufruf
11.4 Parameterübergabe an Funktionen
12. Benutzerdefinierte Typen
13. Übersetzen von C-Programmen
14. Arrays
15. Zeiger
16. Aufgaben
III. C++

11.1 Funktionsprototypen

Der Kopf einer Funktion hat folgenden Aufbau

```
returntype functionname(parametertype parametername, ...) ;
```

Funktionen liefern im allgmeinen ein Ergebnis zurück, der Datentyp des Ergebnisses wird durch returntype spezifiziert. Funktionen, die kein Ergebnis liefern, also Prozeduren im Sinne von Pascal, besitzen den Ergebnistyp void.

`functionname` ist der Name der Funktion, unter dem sie angesprochen, d.h. von anderen Funktionen aufgerufen werden kann.
In runden Klammern folgen die Parameter (Argumente) der Funktion. Hierbei ist jeweils der Typ paramtype eines Parameters und dann dessen Name anzugeben. Hat eine Funktion mehrere Parameter, so werden diese durch Kommata getrennt aufgelistet.

Die obige Deklaration einer Funktion wird durch ein Semikolon abgeschlossen. Es handelt sich um den *Prototyp*. Es wird hierbei nicht festgelegt, was die Funktion tut (es fehlt ja noch der C-Code), sondern nur ihre Schnittstelle spezifiziert, d.h. angegeben, welchen Typ der Rückgabewert und die Parameter haben. Welchen Sinn hat das?
Funktionen werden von anderen Funktionen aufgerufen. Werden dabei die falschen Parametertypen übergeben oder wird der Rückgabewert falsch interpretiert, verhält sich das Programm nicht korrekt und die Fehlersuche ist sehr mühsam.
Deshalb muß zu jeder Funktion ein Prototyp existieren. Günstig ist es, die Prototypen von Funktionen zur Lösung einer bestimmten Aufgabe in einer Header-Datei zusammenzufassen. Dieser Header wird dann per #include in anderen .c-Modulen, die die Funktionen aufrufen, eingebunden. Dadurch kann der Compiler prüfen, ob die Funktion korrekt verwendet wird.

rtr Startseite
Einführung
Inhalt
1. Vorbemerkungen
I. Linux
II. C
8. Ein C-Programm
9. Variablen / Datentypen
10. Wichtige Sprachelemente
11. Funktionen
11.1 Funktionsprototypen
11.2 Funktionsdefinition
11.3 Funktionsaufruf
11.4 Parameterübergabe an Funktionen
12. Benutzerdefinierte Typen
13. Übersetzen von C-Programmen
14. Arrays
15. Zeiger
16. Aufgaben
III. C++

11.2 Funktionsdefinition returntype functionname(parametertype parametername, ...) ;

Die eigentliche Definition einer Funktion besteht wiederum aus dem Kopf wie oben angegeben, jedoch ohne das abschließende Semikolon. Danach folgt in einem { ... } Block der Code der Funktion. Die letze Anweisung in diesem Block ist die return-Anweisung, die die Rückgabe des Ergebnisses an die aufrufende Funktion bewirkt.

Beispiel - *Berechnung der Fakultät*:
In einem Header fak.h steht der Prototyp: int fak (int n) ;
Die Funktionsdefinition (steht in einem .c-Modul fak.c) sieht wie folgt aus:

```
int fak (int n)
{
  int i, ergebnis;
  ergebnis = 1 ;
  for (i=1; i<= n; i++)
  {
    ergebnis *= i ;      /* Multiplikation und Zuweisung */
  }
  return ergebnis ;
}
```

rtr Startseite
Einführung
Inhalt
1. Vorbemerkungen
I. Linux
II. C
8. Ein C-Programm
9. Variablen / Datentypen
10. Wichtige Sprachelemente
11. Funktionen
11.1 Funktionsprototypen
11.2 Funktionsdefinition
11.3 Funktionsaufruf
11.4 Parameterübergabe an Funktionen
12. Benutzerdefinierte Typen
13. Übersetzen von C-Programmen
14. Arrays
15. Zeiger
16. Aufgaben
III. C++

11.3 Funktionsaufruf

Eine Funktion wird aufgerufen durch Angabe des Funktionsnamens und der aktuellen Parameterwerte in runden Klammern. Besitzt die Funktion einem Ergebnistyp, der von void verschieden ist, kann das Ergebnis direkt einer Variablen vom entsprechenden Typ zugeordnet werden. Im Beispiel der Fakultätsfunktion lauten mögliche Aufrufe:

```
int a, b ;
a = 3 ;
b = fak(a) ;
a = fak(4) ;
fak(a) ;
```

Der letze Aufruf ist zwar auch korrekt, macht aber keinen Sinn, da das Ergebnis ignoriert wird. Durch Einbinden der Headerdatei #include "fak.h", die den Prototyp der Funktions fak enthält, kann der Compiler die Aufrufe der Funktion auf Korrektheit, d.h. Übereinstimmung mit den Prototyp, prüfen.
Rekursive Funktionsaufrufe sind ebenfalls möglich.

rtr Startseite
Einführung
Inhalt
1. Vorbemerkungen
I. Linux
II. C
8. Ein C-Programm
9. Variablen / Datentypen
10. Wichtige Sprachelemente
11. Funktionen
11.1 Funktionsprototypen
11.2 Funktionsdefinition
11.3 Funktionsaufruf
11.4 Parameterübergabe an Funktionen
12. Benutzerdefinierte Typen
13. Übersetzen von C-Programmen
14. Arrays
15. Zeiger
16. Aufgaben
III. C++

11.4 Parameterübergabe an Funktionen

Der Aufruf einer Funktion bedeutet auf Maschinenebene:

- Kopieren der Parameterwerte auf den Stack
- Abarbeitung des Codes der aufgerufenen Funktion
- Kopieren des Ergebniswertes auf den Stack -oder in ein Prozessorregister
- Rücksprung zur aufrufenden Funktion

Die Funktion arbeitet also mit *Kopien* der Parameterwerte. Wenn die Funktion die Parameterwerte manipuliert, sind die Änderungen in der aufrufenden Funktion nicht wirksam. Ist dies erforderlich, so muß nicht der Wert eines Parameters übergeben werden (*call by value*), sondern die Adresse an der dieser Wert steht (*call by reference*). Dazu muß der Typ des entsprechenden Parameters ein Zeigertyp sein.

Beispiel zu *call by value*

```
void myfunc(int a)
{
 a = 100 ;
}

int main (void)
{
 int a ;
 a = 3 ;
 myfunc(a) ;
 printf("%d\n", a) ;
/* immer noch 3 */
}
```

Beispiel zu *call by reference*

```
void myfunc(int *a)
{
*a = 100 ;  /* Dereferenzierung! */
}

int main (void)
{
 int a ;
 a = 3 ;
 myfunc(&a) ; /* Adressuebergabe! */
 printf("%d\n", a) ;
 /* jetzt ist a = 100*/
}
```

Wenn eine Struktur mit vielen Komponenten (siehe Abschnitt 12) als Parameter an eine Funktion übergeben wird, ist es sinnvoll den *call by reference* Mechanismus zu verwenden, auch wenn die Werte in der Funktion nicht geändert werden sollen. Der Grund dafür ist, daß sonst alle Strukturkomponenten auf den Stack kopiert werden, was mit einem nicht unerheblichen Zeitaufwand verbunden ist.

rtr Startseite
Einführung
Inhalt
1. Vorbemerkungen
I. Linux
II. C
8. Ein C-Programm
9. Variablen / Datentypen
10. Wichtige Sprachelemente
11. Funktionen
12. Benutzerdefinierte Typen
13. Übersetzen von C-Programmen
14. Arrays
15. Zeiger
16. Aufgaben
III. C++

12. Benutzerdefinierte Typen

Man kann in C eigene Datentypen definieren. Dies erfolgt normalerweise in einer Header-Datei. Für die selbstdefinierten Typen gilt dasselbe wie für die Basistypen, z.B. daß durch Voranstellen eines * bei der Variablendefinition ein Zeiger vereinbart wird.
Es gibt unterschiedliche Arten von benutzerdefinierten Typen:

`struct`'s	Strukturen, d.h. zusammengesetzte Typen
`unions`'s	ähnlich wie `struct`'s, aber zur Laufzeit veränderbar
`enums`'s	Aufzählungstypen

Wir behandeln hier nur die Strukturen (`struct`'s), d.h. Typen, die aus verschiedenen Komponenten zusammengesetzt sind. Diese sind immer dann sinnvoll einzusetzen, wenn Informationen, die ein bestimmtes Problem charakterisieren, zusammengefaßt werden sollen. Ein klassisches Beispiel hierzu sind die komplexen Zahlen:

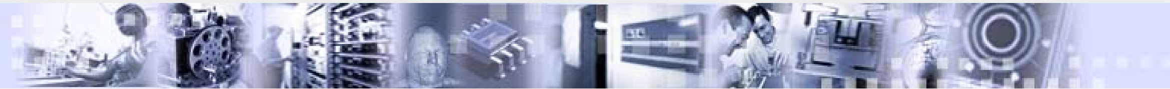
```
typedef struct
{
  double r ,   /* Realteil      */
         i ;   /* Imaginaerteil */
}
Complex ;
```

Hier haben die beiden Komponenten denselben Typ, nämlich double. Allgemein können die Kompontenen von Strukturen aber beliebige Typen haben, also auch Zeigertypen und wiederum Strukturen.
Steht die obige Typdefinition in einem Header complex.h, so ist in einem .c-Modul, das mit diesem Typ arbeitet, dieser Header einzubinden. Anschließend können Variablen vom Typ Complex genauso behandelt werden wie Basistypen.
Beispiel: Addition von komplexen Zahlen:

```
#include<complex.h>

Complex  complexAdd(Complex c1, Complex c2)
{
  Complex sum ;

  sum.r = c1.r + c2.r ;
  sum.i = c1.i + c2.i ;

  return sum ;
}
```

Die Prototypen solcher Funktionen wie der komplexen Addition, sollten ebenfalls im Header complex.h stehen.
Das Beispiel zeigt, wie auf die einzelnen Komponenten einer Strukturvariablen zugegriffen wird. Mit der Konstruktion varname.compname wird die Komponente compname einer Variablen varname angesprochen. varname muß natürlich vom entsprechenden Strukturtyp sein.
Arbeitet man mit Zeigern auf Strukturvariablen, so wird der . ersetzt durch ->. Ist varptr ein Zeiger auf eine Strukturvariable, so ist die Komponente compname anzusprechen durch varptr->compname.
Beispiel: Nullsetzen einer komplexen Zahl:

```
#include<complex.h>

void setComplexZero(Complex *c)
{
  c->r = 0.0 ;
  c->i = 0.0 ;
}
```



rtr Startseite
Einführung
Inhalt
1. Vorbemerkungen
I. Linux
II. C
8. Ein C-Programm
9. Variablen / Datentypen
10. Wichtige Sprachelemente
11. Funktionen
12. Benutzerdefinierte Typen
13. Übersetzen von C-Programmen
13.1 Zerlegung in Module
13.2 Schritte der Programmübersetzung
13.3 Übersetzung mit dem make-Programm
14. Arrays
15. Zeiger
16. Aufgaben
III. C++

13.1 Zerlegung in Module

Wie im Zusammenhang mit Funktionsprototypen und Typedefinitionen bereits ausgeführt, sollten diese Elemente in einer Header-Datei zusammengefaßt werden. Header-Dateien besitzen die Endung .h Sie enthalten auch die Konstantendefinitionen.
Die Funktionsdefinitionen, d.h. der eigentliche Quelltext, wird in einem oder mehreren .c-Modulen untergebracht. Die Aufteilung in mehrere Module führt im allgemeinen zu größerer Übersichtlichkeit und hat Vorteile bei der Übersetzung der Module mithlife des make-Programms (siehe Abschnitt 13.3). Funktionen, die einen bestimmten Problemkreis betreffen, z.B. verschiedene Funktionen zum Rechnen mit komplexen Zahlen, können in einer gemeinsamen .c-Datei stehen. Sinnvoll ist auch, ein eigenes Modul für die main-Funktion zu verwenden.

TECHNISCHE UNIVERSITÄT DARMSTADT

Fachgebiet Regelungstheorie und Robotik

Impressum | Druckversion

REGELUNGSTHEORIE UND ROBOTIK rtr

Sie befinden sich: > TU Darmstadt > ETIT > IAT > RTR > Lehre > E-Learning > C/C++-Onlinekurs > II. C > 13. Übersetzen von C-Programmen > 13.2 Schritte der Programmübersetzung

rtr Startseite
Einführung
Inhalt
1. Vorbemerkungen
I. Linux
II. C
8. Ein C-Programm
9. Variablen / Datentypen
10. Wichtige Sprachelemente
11. Funktionen
12. Benutzerdefinierte Typen
13. Übersetzen von C-Programmen
13.1 Zerlegung in Module
13.2 Schritte der Programmübersetzung
13.3 Übersetzung mit dem make-Programm
14. Arrays
15. Zeiger
16. Aufgaben
III. C++

13.2 Schritte der Programmübersetzung

Zunächst werden die .c und .h-Dateien, die das Programm bilden sollen, editiert. Hierzu kann ein beliebiger Texteditor verwendet werden.
Beim Übersetzen, d.h. beim Aufruf des Compilers mit dem Kommando cc, laufen *automatisch* folgende Schritte ab:

- der *Präprozessor* bearbeitet `#include` und `#define` Anweisungen, d.h. bindet die Header-Dateien ein und ersetzt die Konstantennamen durch die zugeordneten Werte.
- Der eigentliche *Compiler* übersetzt das Programm. Er erzeugt sog. Objektmodule (`.o`-Dateien), die schon Maschinencode enthalten, aber noch nicht lauffähig sind.
- Der *Linker* bindet die Objektmodule mit der Standardbibliothek, sowie evtl. benutzereigenen Bibliotheken zu einem lauffähigen Programm zusammen.

Präprozessor, Compiler und Linker verarbeiten jeweils verschiedene Flags, die z.B. festlegen, in welchen Verzeichnissen die Header-Dateien gesucht werden und welche Art von Programmcode erzeugt werden soll (z.B. Code-Optimierung oder Debug-Version).
Normalerweise werden diese Einstellungen in einem sog. Makefile vorgenommen, das von dem im nächsten Abschnitt behandelten make-Programm verarbeitet wird.

rtr Startseite
Einführung
Inhalt
1. Vorbemerkungen
I. Linux
II. C
8. Ein C-Programm
9. Variablen / Datentypen
10. Wichtige Sprachelemente
11. Funktionen
12. Benutzerdefinierte Typen
13. Übersetzen von C-Programmen
13.1 Zerlegung in Module
13.2 Schritte der Programmübersetzung
13.3 Übersetzung mit dem make-Programm
14. Arrays
15. Zeiger
16. Aufgaben
III. C++

13.3 Übersetzung mit dem make-Programm

Im Verlauf der Programmentwicklung werden häufig Änderungen am Quelltext vorgenommen, das Programm übersetzt, wieder modifiziert, neu übersetzt, usw.
Insbesondere wenn das Programm aus mehreren .c-Modulen besteht, ist der Einsatz des make-Hilfsprogramms sinnvoll. make erkennt, welche Module seit der letzten Übersetzung modifiziert wurden und ruft den Compiler nur für diese auf. Dadurch wird der gesamte Vorgang deutlich beschleunigt. Außerdem können im Makefile die Optionen (Flags) für Präprozessor, Compiler und Linker gesetzt werden. Der Programmierer muß im entsprechenden Verzeichnis nur das Kommando make eingeben, der Rest läuft automatisch ab. Im folgenden ist das Makefile des Einführungsbeispiels (Abschnitt 16.1) abgedruckt.

```
# **** Preprocessor
CPPFLAGS=
# **** Linker
LDFLAGS = -lm
# **** C-Compiler
CFLAGS= -g

LINK.c=$(CC) $(CFLAGS) $(CPPFLAGS) $(LDFLAGS)

# **** Abhaengigkeiten speichern
.KEEP_STATE:

# **** Sources

SRC     = sgen.c sgmain.c
OBJ     = $(SRC:.c=.o)

# **** Regeln

sgen:   $(OBJ)
        $(LINK.c) -o  $@ $(OBJ)
```

In Makefile werden Kommentare durch das Zeichen # eingeleitet. Die Zeile CFLAGS= -g bewirkt, daß der Compiler debug-fähigen Code erzeugt*. Durch LDFLAGS = -lm wird der Linker dazu veranlaßt, die Mathematik-Bibliothek hinzuzubinden (hier wegen der Sinusfunktion).
In den Zeilen

```
SRC     = sgen.c sgmain.c
OBJ     = $(SRC:.c=.o)
```

wird angegeben, welche .c-Module das Programm bilden.
Ein Makefile sieht zunächst etwas ,,kryptisch`` aus, i.allg. schreibt man diese Datei aber nicht zu jedem Programm neu, sondern übernimmt ein bestehendes Makefile und setzt im wesentlichen nur die Namen der .c-Module ein. Bei Entwicklungsumgebungen (auf PC's etwa Visual C oder Borland C) gibt es übrigens auch Makefiles, der Programmierer kommt nur nicht direkt damit in Berührung.

---

*) Dadurch können Fehler mit einem Source-Debugger gesucht werden. Durch andere Flags kann hier eine Code-Optimierung (--> Geschwindigkeit, Speicherplatz) erreicht werden.

TECHNISCHE UNIVERSITÄT DARMSTADT

Fachgebiet Regelungstheorie und Robotik

Impressum | Druckversion

REGELUNGSTHEORIE UND ROBOTIK rtr

rtr Startseite
Einführung
Inhalt
1. Vorbemerkungen
I. Linux
II. C
8. Ein C-Programm
9. Variablen / Datentypen
10. Wichtige Sprachelemente
11. Funktionen
12. Benutzerdefinierte Typen
13. Übersetzen von C-Programmen
14. Arrays
14.1 Character-Arrays, Strings
15. Zeiger
16. Aufgaben
III. C++

14. Arrays

Durch die Definition

```
typename  arrayname[N] ;
```

wird ein Feld von N Variablen vom Typ typename vereinbart. Die Elemente des Arrays werden durch den Feldnamen arrayname gefolgt von dem Index des Elements in eckigen Klammern angesprochen.
Beispiel:

```
double x[10] ;       /* Deklaration eines Arrays von 10 Fliesskommazahlen */
int    i     ;

for (i=0; i<10; i++) x[i] = 0.0 ;  /* Zugriff über Schleife */
```

Arrays können durch eine geklammerte Liste initialisiert werden:

```
int count[5] = { 1, 2, 3, 4, 5 };
```

Bei der Initialisierung mit Werten kann die Angabe der Feldgröße entfallen, sie wird dann vom Compiler bestimmt:

```
int count[] = { 1, 2, 3, 4, 5 };
```

Besonderheiten:

- Das erste Feldelement hat stets den Index **0**. In dem oben dargestellten Beispiel sind daher die Indizes 0...9 gültig.
- Auf die Einhaltung der Feldgrenzen muß der Programmierer selbst achten. Wird ein Feld mit einem ungültigen Index angesprochen, stürzt das Programm in der Regel ab oder es verhält sich nicht wie vorgesehen.
- Zweidimensionale Felder (d.h. Matrizen) lassen sich durch Angabe von Zeilen- und Spaltenzahl vereinbaren und durch Zeilen- und Spaltenindex adressieren:

```
int matrix[10][20] ; /* Deklaration einer Matrix mit 10x20 Elementen */
...
matrix[5][6] = 0 ;
```

- Alle Operationen auf Feldern lassen sich auch mit den im nächsten Abschnitt beschriebenen *Zeigern* formulieren. Der Name des Feldes steht für dessen Basisadresse, d.h. den Zeiger auf den Feldanfang.

rtr Startseite
Einführung
Inhalt
1. Vorbemerkungen
I. Linux
II. C
8. Ein C-Programm
9. Variablen / Datentypen
10. Wichtige Sprachelemente
11. Funktionen
12. Benutzerdefinierte Typen
13. Übersetzen von C-Programmen
14. Arrays
14.1 Character-Arrays, Strings
15. Zeiger
16. Aufgaben
III. C++

14.1 Character-Arrays, Strings

Im Gegensatz zu C++ gibt es in C keinen elementaren Datentyp für Zeichenketten (Strings), daher werden diese stets durch ein Array von einzelnen char-Zeichen repräsentiert. Um also den String "Hallo" in C zu speichern, müsste man schreiben:

```
char Zeichenkette[6];

Zeichenkette = { 'H', 'a', 'l', 'l', 'o', '\0' };
```

Das abschließende Zeichen ist das Null-Zeichen, es ist erforderlich, um den String abzuschließen. Man spricht dann auch von einem **nullterminierten String**. Wird ein String in einem Stream verarbeitet oder einer String-Bibliotheksfunktion übergeben, so kann anhand der Nullterminierung das Ende der Zeichenkette erkannt werden. Wird ein String nicht nullterminiert, so kann ein **Speicherzugriffsfehler** auftreten.
Zur Initialisierung von Zeichenkettenkonstanten gibt es auch eine einfachere Schreibweise:

```
char Zeichenkette[] = "Hallo";
```

Der Compiler hängt das Nullzeichen dabei automatisch an und bestimmt die Größe des Arrays selbständig.
Es gibt eine Reihe von Stringmanipulationsfunktionen, die durch Einbinden von `string.h` zur Verfügung stehen:

```
strlen: bestimmt die Länge eines Strings in Byte

trcat: hängt zwei Strings aneinander

strcpy: kopiert einen String in einen anderen

strcmp: vergleicht zwei Strings
```

TECHNISCHE UNIVERSITÄT DARMSTADT

Fachgebiet Regelungstheorie und Robotik

Impressum | Druckversion

REGELUNGSTHEORIE UND ROBOTIK rtr

Sie befinden sich: > TU Darmstadt > ETIT > IAT > RTR > Lehre > E-Learning > C/C++-Onlinekurs > II. C > 15. Zeiger

- rtr Startseite
- Einführung
- Inhalt
- 1. Vorbemerkungen
- I. Linux
- II. C
  - 8. Ein C-Programm
  - 9. Variablen / Datentypen
  - 10. Wichtige Sprachelemente
  - 11. Funktionen
  - 12. Benutzerdefinierte Typen
  - 13. Übersetzen von C-Programmen
  - 14. Arrays
  - > 15. Zeiger
    - 15.1 Grundlagen
    - 15.2 Zeigerarithmetik
    - 15.3 NULL-Pointer
    - 15.4 Dynamische Speicherverwaltung
    - 15.5 Zeiger auf Zeiger
    - 15.6 Zeiger auf Funktionen
    - 15.7 Funktionen mit variabler Argumentliste
  - 16. Aufgaben
- III. C++

15. Zeiger

Zeiger (*pointer*) werden in C-Programmen häufig verwendet. Sie erlauben sehr ``hardwarenah`` -und damit effizient- zu programmieren. Andererseits können Fehler bei der Verwendung von Zeigern, soweit sie nicht durch den Compiler entdeckt werden, zu Fehlverhalten oder Absturz des Programms führen.

rtr Startseite
Einführung
Inhalt
1. Vorbemerkungen
I. Linux
II. C
8. Ein C-Programm
9. Variablen / Datentypen
10. Wichtige Sprachelemente
11. Funktionen
12. Benutzerdefinierte Typen
13. Übersetzen von C-Programmen
14. Arrays
15. Zeiger
15.1 Grundlagen
15.2 Zeigerarithmetik
15.3 NULL-Pointer
15.4 Dynamische Speicherverwaltung
15.5 Zeiger auf Zeiger
15.6 Zeiger auf Funktionen
15.7 Funktionen mit variabler Argumentliste
16. Aufgaben
III. C++

15.1 Grundlagen

Ein Zeiger ist die *Adresse* einer Variablen. Zu jedem Datentyp type existiert ein entsprechender Zeigertyp type*. Die beiden folgenden Operatoren werde häufig im Zusammenhang mit Zeigern angewandt:

1. Der Adreß-Operator `&` liefert die Adresse der Variablen auf die er angewandt wird.
2. Der Dereferenzierungsoperator `*` liefert den Wert der Variablen, die durch den Zeiger angesprochen (referenziert) wird.

```
int  i      ;      /* Deklaration einer Variablen vom Typ int            */
int *i_ptr ;      /*              einer Variablen vom Typ Zeiger auf int */

i      = 100 ;     /* Zuweisung an i */
i_ptr  = &i   ;    /* Zuweisung der Adresse von i an i_ptr      */
*i_ptr = 200 ;     /* Dereferenzierung von i_ptr und Zuweisung */
```

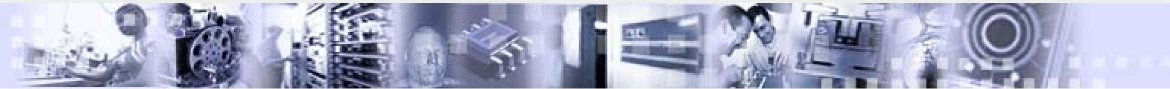
rtr Startseite
Einführung
Inhalt
1. Vorbemerkungen
I. Linux
II. C
8. Ein C-Programm
9. Variablen / Datentypen
10. Wichtige Sprachelemente
11. Funktionen
12. Benutzerdefinierte Typen
13. Übersetzen von C-Programmen
14. Arrays
15. Zeiger
15.1 Grundlagen
15.2 Zeigerarithmetik
15.3 NULL-Pointer
15.4 Dynamische Speicherverwaltung
15.5 Zeiger auf Zeiger
15.6 Zeiger auf Funktionen
15.7 Funktionen mit variabler Argumentliste
16. Aufgaben
III. C++

15.2 Zeigerarithmetik

Zeiger als Adressen von Variablen im Speicher haben unabhängig vom Datentyp der referenzierten Variablen stets dasselbe Format. Trotzdem sind Zeiger in C an bestimmte Typen gebunden. Dadurch wird unter anderem die Zeigerarithmetik vereinfacht.
Im folgenden Beispiel wird jeweils ein Feld von 100 Elementen des Typs int, double und des zusammengesetzen Typs Complex definiert. Dieser benutzerdefinierte Typ enthält zwei Komponenten, nämlich den Real- und Imaginärteil einer komplexen Zahl jeweils als double-Fließkommawert. In den Zeilen 12 bis 14 werden die entsprechend deklarierten Zeigervariablen auf den jeweiligen Feldanfang gesetzt. Hierbei ist *kein* Adreßoperator erforderlich, denn i, d und c sind Felder deren Name bereits die Basisadresse darstellt.

```
01 typedef struct
02 {
03   double r, i ; /* Real- und Imaginaerteil einer komplexen Zahl */
04 } Complex ;
05
06 int main(void)
07 {
08   int       k       ,       i[100]   ,       *i_ptr   ;
09   double    d[100] ,       *d_ptr   ;
10   Complex   c[100] ,       *c_ptr   ;
11
12   i_ptr = i ;
13   d_ptr = d ;
14   c_ptr = c ;
15
16   for (k=0; k<100; k++)
17   {
18     *i_ptr ++ = 0   ;
19     *d_ptr ++ = 0.0 ;
20
21     c_ptr->r  = 0.0 ;
22     c_ptr->i  = 0.0 ;
23     c_ptr ++        ;
24   }
25   return 0;
26 }
```

Nun zum eigentlichen Sinn dieses Beispiels: In der Schleife ab Zeile 16 werden die Feldelemente auf den Wert 0 gesetzt. Dies geschieht indem der jeweilige Zeiger dereferenziert wird (mit dem Operator *) und der Wert 0 zugewiesen wird. In Zeile 18 und 19 wird der Zeiger außerdem noch mithilfe des ++-Operators inkrementiert. Da das ++ nachgestellt ist, erfolgt erst die Zuweisung und dann das Hochzählen der Zeigervariablen15.1. Hier macht sich nun die Typbindung der Zeiger bemerkbar: In beiden Fällen, also sowohl bei dem int- als auch dem double-Zeiger, bewirkt der ++-Operator das Weiterschalten des Zeigers auf das nächste Feldelement. Dies ist nicht ganz selbstverständlich, denn die referenzierten Variablen belegen unterschiedlich viel Speicherplatz! Während int-Variablen maschinenabhängig entweder 2 oder 4 Byte belegen, handelt es sich bei double-Werten um sog. Fließkommazahlen doppelter Genauigkeit, die 64 Bit, also 8 Byte groß sind. Die in den Zeigern abgelegten Adressen müssen also je nach referenzierter Variable unterschiedlich hochgezählt werden. Da dem Compiler die Typen der Zeigervariablen bekannt sind, kann er dem Programmierer diese (fehleranfälligen) Unterscheidungen abnehmen.
Dasselbe gilt auch für den Zeiger auf den zusammengesetzten Typ Complex. In den Zeilen 21 und 21 werden die beiden Kompontenten jeweils auf 0 gesetzt und in Zeile 23 der Zeiger inkrementiert. Da die Struktur Complex zwei double-Werte enthält, bedeutet das ++ hier eine Addition von 16 Byte auf die aktuelle Adresse.



rtr Startseite
Einführung
Inhalt
1. Vorbemerkungen
I. Linux
II. C
8. Ein C-Programm
9. Variablen / Datentypen
10. Wichtige Sprachelemente
11. Funktionen
12. Benutzerdefinierte Typen
13. Übersetzen von C-Programmen
14. Arrays
> 15. Zeiger
15.1 Grundlagen
15.2 Zeigerarithmetik
> 15.3 NULL-Pointer
15.4 Dynamische Speicherverwaltung
15.5 Zeiger auf Zeiger
15.6 Zeiger auf Funktionen
15.7 Funktionen mit variabler Argumentliste
16. Aufgaben
III. C++

15.3 NULL-Pointer

Jeder Zeigervariablen (unabhängig von welchem Typ) kann die Konstante `NULL` zugewiesen werden, um sie als nicht initialisiert zu kennzeichnen. Der Zugriff über nicht initialisierte Zeiger führt zum Programmabsturz. Bevor man einen Zeiger benutzt (dereferenziert), sollte man durch Vergleich mit `NULL` prüfen, ob er überhaupt initialisiert ist. Dies gilt insbesondere beim Zugriff auf dynamisch allokierte Speicherbereiche.

TECHNISCHE UNIVERSITÄT DARMSTADT

Fachgebiet Regelungstheorie und Robotik

Impressum | Druckversion

REGELUNGSTHEORIE UND ROBOTIK rtr

Sie befinden sich: > TU Darmstadt > ETIT > IAT > RTR > Lehre > E-Learning > C/C++-Onlinekurs > II. C > 15. Zeiger > 15.4 Dynamische Speicherverwaltung

rtr Startseite
Einführung
Inhalt
1. Vorbemerkungen
I. Linux
II. C
8. Ein C-Programm
9. Variablen / Datentypen
10. Wichtige Sprachelemente
11. Funktionen
12. Benutzerdefinierte Typen
13. Übersetzen von C-Programmen
14. Arrays
15. Zeiger
15.1 Grundlagen
15.2 Zeigerarithmetik
15.3 NULL-Pointer
15.4 Dynamische Speicherverwaltung
15.5 Zeiger auf Zeiger
15.6 Zeiger auf Funktionen
15.7 Funktionen mit variabler Argumentliste
16. Aufgaben
III. C++

15.4 Dynamische Speicherverwaltung

Felder, die in Abschnitt 14 beschrieben wurden, werden statisch im Datensegment des jeweiligen Programms abgelegt. Ihre Größe wird zur Übersetzungszeit festgelegt. Dagegen bedeutet dynamische Speicherverwaltung, daß zur Laufzeit Speicherblöcke (variabler Größe) aus dem Speichervorrat der Maschine (dem *Heap*) angefordert werden und auch wieder freigegeben werden. Von diesen Mechanismen sollte man Gebrauch machen, wenn

- die Größe der benötigten Speicherblöcke variabel, also erst zur Laufzeit bekannt ist[1).]
- der Speicher nicht ständig gebraucht wird, sondern nur während bestimmter Berechnungen.

in c bzw. der C-Standardbibliothek werden folgende Funktionen zur dynamischen Speicherwaltung angeboten[2)]:

```
void* malloc (size_t size) ;

void free(void *ptr);
```

`malloc` erwartet als Parameter die Größe des angeforderten Speicherblocks *in Byte*. Um die Größe in Byte nicht von Hand bestimmen zu müssen und portable Programme zu schreiben (--> Maschinenabhängigkeit), benutzt man den Operator sizeof(). Dieser liefert angewandt auf einen beliebigen Typbezeichner die Größe der entsprechenden Variablen in Byte. malloc liefert einen Zeiger auf den Beginn des allokierten Speichers zurück, oder falls die Anforderung vom System nicht erfüllt werden konnte, den Wert NULL. Der zurückgelieferte Zeiger ist vom Typ Zeiger auf void. void kennzeichnet den ``leeren`` Typ. Dies bedeutet hier, daß es sich um einen generischen Zeiger handelt, der durch eine Typumwandlung (cast) in den Typ, der den angeforderten Variablen entspricht, konvertiert werden muß.

```
int *i_ptr ,
     n      ;

n = 120 ;

i_ptr = (int*)malloc(n * sizeof(int)) ;

if (i_ptr == NULL) {   /* Fehlerbehandlung! */   }
```

Hier wird ein Block von 120 Integer-Werten allokiert. Das vorangestellte (int*) stellte die oben angesprochene Umwandlung auf den entsprechenden Zeigertyp dar. Durch Multiplikation der Zahl der angeforderten Elemente mit dem sizeof-Operator wird die Zahl der benötigten Bytes ermittelt.
Der Inhalt des von malloc bereitgestellten Speichers ist undefiniert. Der Programmierer muß selbst für eine Belegung mit sinnvollen Werten sorgen. Verwendet man allerdings die Funktion calloc, wird der reservierte Speicher mit Nullen vorbelegt. Der Prototyp von calloc lautet:

void *calloc (size_t nelem, size_t elsize) ;
Wird der Speicherblock nicht mehr benötigt, muß er durch Aufruf von free mit der Basisadresse als Argument wieder freigegeben werden. Durch diese Freigabe kann der Speicherblock bei den folgenden Anforderungen wieder für andere Zwecke verwendet werden:

```
free(i_ptr)  ;
i_ptr = NULL ;
```

Der Inhalt der Zeigervariablen wird durch den Aufruf von free nicht verändert. Daher kann der Zeiger durch explizite Zuweisung von NULL als ungültig markiert werden. Wird in nachfolgenden Programmabschnitten zuerst geprüft, ob der Zeiger gültig ist, kann so der Zugriff auf den nicht mehr verfügbaren Speicher abgefangen werden.

---

1) Natürlich kann man einfach ein (statisches) Feld mit der maximal zu erwartenden Größe vereinbaren. Dies ist aber eine unflexible Lösung (die maximale Feldgröße kann nur durch Neu-Übersetzen modifiziert werden) und man blockiert möglicherweise unnötig viel Speicher, da die Felder entsprechend überdimensioniert sind.

2) Es gibt noch eine 3. Fuktion, nämlich calloc, deren Wirkung ähnlich wie die von malloc ist. Unterschiede bestehen in der Parameterliste und in der Tatsache, daß calloc den allokierten Speicher mit Nullen initialisiert, während bei malloc der Inhalt des angeforderten Speichers undefiniert ist.

rtr Startseite
Einführung
Inhalt
1. Vorbemerkungen
I. Linux
II. C
8. Ein C-Programm
9. Variablen / Datentypen
10. Wichtige Sprachelemente
11. Funktionen
12. Benutzerdefinierte Typen
13. Übersetzen von C-Programmen
14. Arrays
15. Zeiger
15.1 Grundlagen
15.2 Zeigerarithmetik
15.3 NULL-Pointer
15.4 Dynamische Speicherverwaltung
15.5 Zeiger auf Zeiger
15.6 Zeiger auf Funktionen
15.7 Funktionen mit variabler Argumentliste
16. Aufgaben
III. C++

15.5 Zeiger auf Zeiger

Zeiger existieren für beliebige Datentypen. Zu jedem Typ `type` (sowohl Basistypen als auch benutzerdefinierte Typen) existiert ein entsprechender Zeigertyp `type*`. Dementsprechend können auch Zeiger auf Zeigertypen, also Doppel- oder Mehrfachzeiger gebildet werden.

```
int   zahl  ,
     *ptr   ,
    **dptr  ;

ptr = &zahl  ;
dptr = &ptr  ;

zahl    = 100 ;
**dptr  = 200 ;
```

Hier wird `ptr` mit der Adresse der Variablen `zahl` geladen. `dptr` wird die Adresse des Zeigers `ptr` zugewiesen. Der Inhalt der Variablen `zahl` wird zunächst auf den Wert 100 gesetzt und dann durch zweimaliges Dereferenzieren des Doppel-Zeigers und erneute Zuweisung manipuliert. Was könnte der Sinn dieser komplizierten Konstruktion sein?

Der Nutzen von Mehrfach-Zeigern liegt in der effizienten Adressierung mehrdimensionaler Datenbereiche[*]. Dies lässt sich beispielsweise ausnutzen, wenn sehr effiziente Routinen zum Rechnen mit Matrizen geschrieben werden sollen. Auch der sog. *Frame Buffer* (Bildspeicher) (-->Bildverarbeitung, Computergraphik) wird häufig so organisiert, um sehr schnell auf die einzelnen Pixel zugreifen zu können. Der Geschwindigkeitsvorteil ist dadurch begründet, daß die Adreßrechnung ohne Multiplikationen auskommt.
Anhand der Übungsaufgabe in Anhang 16.2 wird diese Technik verdeutlicht.

---

[*]) Der Speicher eines Rechners ist natürlich immer linear organisiert, also eindimensional. In der abstrakten Vorstellungswelt des Programmierers ist aber z.B. eine Matrix ein zweidimensionales Objekt im Speicher.

TECHNISCHE UNIVERSITÄT DARMSTADT

Fachgebiet Regelungstheorie und Robotik

Impressum | Druckversion

REGELUNGSTHEORIE UND ROBOTIK rtr

- rtr Startseite
- Einführung
- Inhalt
- 1. Vorbemerkungen
- I. Linux
- II. C
  - 8. Ein C-Programm
  - 9. Variablen / Datentypen
  - 10. Wichtige Sprachelemente
  - 11. Funktionen
  - 12. Benutzerdefinierte Typen
  - 13. Übersetzen von C-Programmen
  - 14. Arrays
  - 15. Zeiger
    - 15.1 Grundlagen
    - 15.2 Zeigerarithmetik
    - 15.3 NULL-Pointer
    - 15.4 Dynamische Speicherverwaltung
    - 15.5 Zeiger auf Zeiger
    - 15.6 Zeiger auf Funktionen
    - 15.7 Funktionen mit variabler Argumentliste
  - 16. Aufgaben
- III. C++

15.6 Zeiger auf Funktionen

Auch Funktionen lassen sich durch Zeigervariablen referenzieren. Hierbei stellt der Zeiger die Startadresse des Maschinencodes der referenzierten Funktion dar.

Um eine Zeigervariable `fncPtr` zu vereinbaren mit der sich Funktionen mit dem Ergebnistyp `retType` und der Parameterliste (`p1Type p1, p2Type p2`) referenzieren lassen, verwendet man folgende Konstruktion:

```
retType (*fncPtr)(p1Type p1, p2Type p2);
```

Funktionspointer benötigt man z.B. zur Verwendung der qsort Funktion aus der Standardbibliothek. Diese Funktion ist eine Implementierung des Quicksort-Algorithmus zum Sortieren eines Feldes, dessen Elemente einen beliebigen Datentyp haben dürfen. Der Prototyp dieser Funktion (ist in `stdlib.h` enthalten) sieht so aus:

```
void qsort (void *base, size_t noElems, size_t elemSize,
            int (*compFnc)(const void*, const void*));
```

Das zu sortierende Feld wird durch `base` referenziert (Achtung: cast auf `void*`) und enthält `noElems` Elemente, die jeweils `elemSize` Byte belegen. Der letzte Parameter von `qsort` ist ein Pointer auf eine Funktion, die 2 Zeiger auf Feldelemente als Parameter erwartet und eine Vergleichsoperation für diese Elemente realisiert. Diese Funktion muß den Wert -1, 0 bzw. 1 liefern, wenn das erste Element kleiner, gleich bzw. größer als das zweite ist (oder umgekehrt wenn absteigend sortiert werden soll).
Somit ist es möglich, `qsort` auf beliebige (selbst definierte) Datentypen anzuwenden, man muß nur eine passende Vergleichsfunktion definieren.

rtr Startseite
Einführung
Inhalt
1. Vorbemerkungen
I. Linux
II. C
8. Ein C-Programm
9. Variablen / Datentypen
10. Wichtige Sprachelemente
11. Funktionen
12. Benutzerdefinierte Typen
13. Übersetzen von C-Programmen
14. Arrays
> 15. Zeiger
15.1 Grundlagen
15.2 Zeigerarithmetik
15.3 NULL-Pointer
15.4 Dynamische Speicherverwaltung
15.5 Zeiger auf Zeiger
15.6 Zeiger auf Funktionen
> 15.7 Funktionen mit variabler Argumentliste
16. Aufgaben
III. C++

15.7 Funktionen mit variabler Argumentliste

Manchmal ist es sinnvoll, die Zahl der Parameter einer Funktion erst zur Laufzeit festzulegen. Der Programmierer muß also eine *variable Argumentliste* vorsehen. Im folgenden wird dies anhand einer Funktion myFunc erläutert, die neben zwei ,,festen`` Parametern vom Typ int und double* noch eine beliebige Anzahl weiterer Parameter verarbeiten kann.

1. Das C-Modul, in dem der Rumpf der Funktion mit variabler Parameterliste definiert wird, muß den Header `stdarg.h` einbinden.

2. Der Funktionsprototyp hat folgendes Aussehen:

```
void myFunc(int n, double *dPtr, ...);
```

3. Das folgende Gerüst des Funktionsrumpfes zeigt, wie auf die variablen Parameter zugegriffen werden kann:

```
void myFunc(int n, double *dPtr, ...)
{
  va_list ap;      /* Datentyp 'variable Argumentliste' */

  int intParam ;

  /* Initialisierung der Liste. Hier ist der Name des letzten 'regulaeren'
   * Parameters anzugeben. Es muss laut C-Standard mindestens ein solcher
   * Parameter existieren. Dennoch gibt es Compiler, die auch den Fall
   * unterstützen, dass kein 'regulärer' Parameter existiert. In diesem
   * Fall lautet der entsprechende Aufruf var_start(ap,);
   */
  va_start(ap, dPtr);

  /* Mithilfe des var_arg Makros kann jetzt nacheinander auf die verschiedenen
   * Argumente zugegriffen werden. Dabei muss der (erwartete) Typ des
   * Arguments, auf das zugegriffen werden soll, angegeben werden.
   */
   intParam = va_arg(ap,int);

  /* 'Aufraeumen' der va_list Datenstruktur
   */
  va_end(ap);
}
```

Hinweise:

- Es gibt keine Möglichkeit festzustellen, wieviele Argumente tatsächlich übergeben wurden. Wird `va_arg` aufgerufen nachdem das letzte Argument bereits ausgelesen wurde, ist das Verhalten undefiniert!

  Sinnvoll ist daher, in einem ,,regulären`` Argument die Zahl der tatsächlich übergebenen Parameter zu spezifizieren.

- Die Funktionen der `printf`-Familie arbeiten bekanntlich mit variablen Argumentlisten. Sie erkennen die Zahl (und die Typen) der nachfolgenden Argumente anhand des Formatstrings (d.h. an den darin auftretenden Konversionszeichen).

- Schreibt man eine eigene Funktion zur Ausgabe von Fehler- oder Statusmeldungen, so ist der Mechanismus der variablen Argumentliste nützlich, um dabei auch Werte (einer vorab nicht festgelegten Anzahl) von Variablen auszugeben. Hierzu kann man das `va_arg` Makro und die Funktionen `vprintf` bzw. `vfprintf` verwenden. Diesen kann man einen String mit Konversionszeichen und die Argumentliste übergeben.

Viel Spaß beim Experimentieren.

rtr Startseite
Einführung
Inhalt
1. Vorbemerkungen
I. Linux
II. C
8. Ein C-Programm
9. Variablen / Datentypen
10. Wichtige Sprachelemente
11. Funktionen
12. Benutzerdefinierte Typen
13. Übersetzen von C-Programmen
14. Arrays
15. Zeiger
16. Aufgaben
16.1 Ein Signalgenerator in C
16.2 Matrizenverarbeitung mit Doppel-Zeigern in C
16.3 Lösung zum Signalgenerator
16.4 Lösung zur Matrizenverarbeitung
III. C++

16.1 Ein Signalgenerator in C

Zur Verdeutlichung der wichtigsten Sprachelemente wird ein einfacher **Signalgenerator** als C-Programm realisiert:
Wahlweise soll eine der Signalformen

- Sprungfunktion
- Sinusfunktion
- Dreieckschwingung

verwendet werden können. Die Signale sind jeweils durch

- Startzeitpunkt
- Endzeitpunkt
- Amplitude und (außer bei der Sprungfunktion)
- Periodendauer

charakterisiert.
Das Programm fordert den Benutzer auf, eine der drei Signalformen zu wählen und die entsprechende Parametrierung vorzunehmen. Die Ergebnisse, d.h. jeweils Zeitstempel und Signalwert, werden zeilenweise auf die Standardausgabe (Bildschirm oder Umleitung in eine Datei) geschrieben. Mit dem Program gnuplot, einer Public Domain Software zur Visualisierung beliebiger Daten, können die Signalverläufe später angezeigt werden.

Anleitung:

1. Download der Programmdateien:
    1. Starten Sie nach dem Einloggen einen Web-Browser (Konqueror oder Communicator mit den Befehlen `konqueror` oder `communicator`).
    2. Richten Sie in Ihrem Home-Verzeichnis das Verzeichnis sgen ein. Dazu dient der Befehl `mkdir sgen`.
    3. Laden Sie mit Ihrem Browser die Datei `sgen_aufgabe.tar` in das Verzeichnis `sgen` herunter (für Windows-User: sgen_aufgabe.zip).
    4. Entpacken Sie das tar-Archiv mit dem Befehl
       `tar -xvf sgen_aufgabe.tar`.

       Im Verzeichnis `~/sgen` (~ steht für Ihren Benutzernamen) finden Sie nun die folgenden Dateien:

       - `sgen.h` : Definitionen von Konstanten und Typen sowie Prototypen der Funktionen des Moduls `sgen.c`
       - `sgen.c` : Realisierung der Funktionen
       - `sgmain.c` : Dieses Modul enthält das Hauptprogramm, d.h. die Funktion main des Signalgenerators.
       - `Makefile` : Datei zur automatisierten Erstellung des aus- führbaren Programms aus den `.c`-Modulen.

       Die Datei `sgen.c` ist noch nicht vollständig, sondern soll teilweise von Ihnen erstellt werden!

2. Bevor Sie anfangen, sollten Sie sich die obigen Dateien ansehen (jeweils mit dem Kommando `nedit` *filename* in einen Texteditor laden) und sich über folgende Punkte Klarheit verschaffen:

    1. Welchen Zweck hat die Definition eines zusammengesetzten Datentyps (`struct`)? In `sgen.h` wird der Strukturtyp `Signal` (zur Charakterisierung der hier behandelten Funktionsverläufe) definiert. Wie formuliert man den Zugriff auf die einzelnen Komponenten dieser Struktur? Wie unterscheidet sich die Syntax bei Zeigern auf Strukturtypen von der bei ,,normalen`` Variablen des Strukturtyps? (Hinweis: Sehen Sie sich dazu die Formulierungen in `sgmain.c` beim Zugriff auf die Komponenten der Variablen `my_signal` und die Verwendung des Parameters `signal_ptr` in den Funktionen in `sgen.c` an.)

    2. Wozu dienen Funktionsprototypen?

    3. Was ist der Unterschied zwischen *call by value* und *call by reference*, wie werden beide Mechanismen in C realisiert. Wann ist *call by reference* zu verwenden? Was passiert ,,hardwaretechnisch`` bei der Parameterübergabe?

    4. Machen Sie sich die ,,Aufrufhierarchie`` der Funktionen klar.

3. Übersetzen Sie das Programm durch den Aufruf von `make`. Lassen Sie mit dem Kommando `ls` anschließend den Inhalt des Verzeichnisses anzeigen. Sie finden nun die Objektmodule `sgen.o` und `sgmain.o` sowie das ausführbare Programm `sgen*` vor. Starten Sie das Programm (`sgen` eintippen.) Wählen Sie die Signalform 1 (Sprungfunktion). Die Ergebnisse werden auf dem Bildschirm, d.h. dem jeweiligen (virtuellen) Terminal (Sie arbeiten hier unter UNIX), ausgegeben.

   Um den Verlauf zu visualisieren, müssen die Werte in eine Datei geschrieben werden. Hierzu starten Sie das Programm nochmal, wobei Sie jedoch die Standardausgabe auf eine Datei, z.B. `test.dat`, richten:
   `sgen > test.dat`
   Starten Sie nun das Programm `gnuplot`. Geben Sie an dessen Prompt ein:

```
plot "test.dat" with lines
```

4. Ergänzen Sie den fehlenden Programmcode in den Funktionen
   - `sine` (Sinusfunktion)
   - `triangle` (Dreiecksschwingung)

   des Moduls `sgen.c`. Die Funktionen sollen sich analog zu `step` verhalten.

   Hinweise:

   1. Die Standard-Bibliothek der mathematischen Funktionen enthält eine Funktion `double sin(double x) ;` Das Argument `x` muß im Bogenmaß angegeben werden. Der Prototyp dieser Funktion wird durch Einbinden der Header-Datei `math.h` im Modul `sgen.c` sichtbar.
   2. Wenn Sie nach dieser Anleitung vorgegangen sind, werden Sie feststellen, daß bei den erneuten Aufrufen von make nur noch das Modul `sgen.c` übersetzt wird. Das Make-Utility erkennt, welche Module editiert wurden (wie?) und reduziert so (bei komplexeren Projekten) die Übersetzungszeiten deutlich.
   3. Wenn Sie die Datei `test.dat` mehrfach angeben, werden die neuen Daten jeweils am Ende angefügt, wodurch `gnuplot` etwas irritiert wird. Daher sollte `test.dat` zwischendurch immer wieder gelöscht werden (`rm`-Kommando).
   4. Um die Vorteile eines Multitasking/Multiuser Betriebssystems kennenzulernen, sollte `gnuplot` von einem zweiten Terminal (z.B. im ersten Terminal `cmdtool&` eingeben) gestartet werden. Durch Eingabe von `Control-P` wiederholt `gnuplot` übrigens den letzten Befehl.
   5. Probieren Sie aus, was das Programm tut, wenn Sie die Zeile `#include <math.h>` im Modul `sgen.c` weglassen.
   6. Die Musterlösungen finden Sie hier. Sie stehen überdies auch zum Download bereit (`sgen_loesung.tar`, für Windows-User: sgen_loesung.zip). Richten Sie sich am besten in Ihrem Homeverzeichnis ein weiteres Verzeichnis an (etwa `~/sgen_ml`), um die Musterlösungen einzusehen. Das Vorgehen hierzu ist dasselbe wie oben.

rtr Startseite
Einführung
Inhalt
1. Vorbemerkungen
I. Linux
II. C
8. Ein C-Programm
9. Variablen / Datentypen
10. Wichtige Sprachelemente
11. Funktionen
12. Benutzerdefinierte Typen
13. Übersetzen von C-Programmen
14. Arrays
15. Zeiger
16. Aufgaben
16.1 Ein Signalgenerator in C
16.2 Matrizenverarbeitung mit Doppel-Zeigern in C
16.3 Lösung zum Signalgenerator
16.4 Lösung zur Matrizenverarbeitung
III. C++

16.2 Matrizenverarbeitung mit Doppel-Zeigern in C

Besorgen Sie sich wie im vorigen Beispiel die erforderlichen Dateien über das World Wide Web und schaffen Sie sich eine geeignete Programmierumgebung:

1. Starten Sie erneut Ihren Web-Browser (Konqueror, Communicator mit den Befehlen `konqueror` oder `communicator`).
2. Richten Sie in Ihrem Home-Verzeichnis das Verzeichnis `matrix` ein. Dazu dient der Befehl `mkdir matrix`.
3. Wechseln Sie in das Verzeichnis `matrix`, in dem Sie `cd matrix` eintippen.
4. Laden Sie mit Ihrem Browser die Datei `matrix_aufgabe.tar` in das Verzeichnis `matrix` herunter (für Windows-User: matrix_aufgabe.zip).
5. Entpacken Sie das tar-Archiv mit dem Befehl `tar -xvf matrix_aufgabe.tar`.

Im Verzeichnis `~/matrix` (~ steht nach wie vor für Ihren Benutzernamen) finden Sie nun die Dateien `matrix.h`, `matrix.c` und `main.c`.

16.2.1 Makefile

Kopieren Sie das Makefile des Beispiels mit dem Signalgenerator (`cp ~/sgen/Makefile ~/matrix`) und passen Sie es für diese Übungsaufgabe an.

16.2.2 Header

Analysieren Sie die Header-Datei `matrix.h`. Sie finden den Typ Matrix, eine Struktur, die Zeilen und Spaltenzahl der Matrix (jeweils `int`) sowie einen Doppel-Zeiger auf die Matrixelemente, die vom Typ `float` sein sollen, enthält. Weiterhin stehen in diesem Header die Prototypen der Funktionen, die Sie im Modul `matrix.c` realisieren sollen.

16.2.3 Allokieren und Initialisieren einer Matrix

Machen Sie sich die Anweisungen in der Funktion init im Modul `matrix.c` klar. Besonders wichtig sind die Aufrufe von `malloc` sowie der Zugriff auf die einzelnen Matrixelemente. Warum muß der Doppelzeiger beim Aufruf von `scanf` nur einmal de-referenziert werden?

16.2.4 Funktionen

Vervollständigen Sie die folgenden Funktionen in matrix.c:

	Funktion	Beschreibung
1.	`void show(Matrix *m_ptr)`	Matrix zeilenweise formatiert ausgeben.
2.	`int trans(Matrix *m1_ptr, Matrix *m2_ptr)`	Transponierte der durch `m1_ptr` angesprochenen Matrix in `m2_ptr` ablegen.
3.	`void kill(Matrix *m_ptr)`	Speicherplatz der Matrix vollständig freigeben.

Hinweis zu 1.:
Um einen einzelnen Fließkommawert auszugeben (Datentyp `float`) verwenden Sie den folgenden Aufruf von `prinf`:
`printf("%-8.2f", elem) ;` (wobei `elem` der Elementwert ist)
Einen Zeilenumbruch erzeugen Sie mit `printf("\n") ;`

Hinweis zu 2.:
Die Funktion `trans` soll den Wert 0 zurückliefern, wenn das Transponieren fehlerfrei durchgeführt werden konnte. Anderenfalls (bei inkompatiblen Matrizen) wird der Wert 1 zurückgeliefert.

Hinweis zu 3.:
Fügen Sie auch in der Funktion `main` im Modul `main.c` die erforderlichen Aufrufe von `kill` ein.

16.2.5 Musterlösung

Die Musterlösung zu dieser Ausgabe finden Sie im hier. Sie kann heruntergeladen werden (matrix_loesung.tar, für Windows-User: matrix_loesung.zip). Es empfiehlt sich erneut die Einrichtung eines Verzeichnisses (etwa `~/matrix_ml`), um mit der Musterlösung herumzuexperimentieren.

rtr Startseite
Einführung
Inhalt
1. Vorbemerkungen
I. Linux
II. C
8. Ein C-Programm
9. Variablen / Datentypen
10. Wichtige Sprachelemente
11. Funktionen
12. Benutzerdefinierte Typen
13. Übersetzen von C-Programmen
14. Arrays
15. Zeiger
16. Aufgaben
16.1 Ein Signalgenerator in C
16.2 Matrizenverarbeitung mit Doppel-Zeigern in C
16.3 Lösung zum Signalgenerator
16.4 Lösung zur Matrizenverarbeitung
III. C++

16.3.1 Signalgenerator

16.3.1.1 Header sgen.h

```
/* --------------------- Definition von Konstanten -------------------------- */

#define PI             3.14159

#define SAMPLING_TIME 0.1         /* Abtastzeit */

/* ----------------------- Definition neuer Datentypen ---------------------- */

typedef struct                  /* Signalbeschreibung       */
{
  int     type ,                /* Kennung der Signalform   */

          start, stop ;         /* Beginn/End-Zeitpunkte    */

  double span, period  ;        /* Amplitude, Periodendauer */
}
Signal ;

/* ------------------------ Prototypen der Funktionen ----------------------- */

int  init     (Signal* ) ;
void step     (Signal* ) ;
void sine     (Signal* ) ;
void triangle (Signal* ) ;
```

16.3.1.2 Modul sgen.c

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

#include "sgen.h"

/* ------------------------------------------------------------------------- */

int init (Signal* signal_ptr)
{
  int ret_val = 0 ;

  double time ;

  fprintf(stderr, "Signalform (1 ... 3)   : ") ;
  scanf("%d", &signal_ptr->type) ;

  fprintf(stderr, "Startzeitpunkt [sec]   : ") ;
  scanf("%lf", &time) ;
  signal_ptr->start = time/SAMPLING_TIME ;

  fprintf(stderr, "Endzeitpunkt [sec]     : ") ;
  scanf("%lf", &time) ;
  signal_ptr->stop = time/SAMPLING_TIME ;

  fprintf(stderr, "Amplitude              : ") ;
  scanf("%lf", &signal_ptr->span) ;

  if (signal_ptr->type!=1)
  {
    fprintf(stderr, "Periodendauer [sec]    : ") ;
    scanf("%lf", &signal_ptr->period) ;
  }

  if ((signal_ptr->type < 1) || (signal_ptr->type > 3))
  {
    fprintf(stderr, "Ungueltige Signalform!\n") ;
    ret_val = 1 ;
  }

  if ((signal_ptr->start < 0) || (signal_ptr->stop < 0) ||
      (signal_ptr->start > signal_ptr->stop))
```

```
  {
    fprintf(stderr, "Ungueltige Start/Ende - Parameter!\n") ;
    ret_val = 1 ;
  }

  return (ret_val) ;
}
/* ------------------------------------------------------------------------ */
void step (Signal *signal_ptr)
{
  int    k ;
  double val, time ;

  time  = 0.0 ;

  for (k=0; k < signal_ptr->stop; k++)
  {
    if (k < signal_ptr->start)
    {
      val = 0.0 ;
    }
    else
    {
      val = signal_ptr->span ;
    }
    printf ("%5.2f\t%5.2lf\n", time, val) ;

    time += SAMPLING_TIME ;
  }
}
/* ------------------------------------------------------------------------ */
void sine (Signal *signal_ptr)
{
  int     k ;

  double  val, omega, time;

  omega = 2.0 * PI / signal_ptr->period ;
  time  = 0.0 ;

  for (k=0; k < signal_ptr->stop; k++)
  {
    if (k < signal_ptr->start)
    {
      val = 0.0 ;
    }
    else
    {
      val = signal_ptr->span * sin(omega * time) ;

      time += SAMPLING_TIME ;
    }
    printf ("%5.2f\t%5.2lf\n", k*SAMPLING_TIME, val) ;
  }
}
/* ------------------------------------------------------------------------ */
void triangle (Signal *signal_ptr)
{
  int     k ;
  double  inc, val, t_25, t_75, t, time ;

  t_25 = 0.25 * signal_ptr->period ;
  t_75 = 0.75 * signal_ptr->period ;

  inc   = 4.0 * SAMPLING_TIME * signal_ptr->span / signal_ptr->period  ;
  t     = 0.0 ;
  time  = 0.0 ;
  val   = 0.0 ;

  for (k=0; k < signal_ptr->start; k++)
  {
    printf ("%5.2f\t%5.2lf\n", time, val) ;

    time += SAMPLING_TIME ;
  }

  for ( ; k < signal_ptr->stop; k++)
  {
    if (t>=signal_ptr->period)
    {
      t = 0.0 ;
    }
    if (t < t_25)
    {
      val += inc ;
    }
    else
    {
      if (t < t_75)
      {
        val -= inc ;
```

```
      }
      else
      {
        val += inc ;
      }
    }

    printf ("%5.2f\t%5.2lf\n", time, val) ;

    t    += SAMPLING_TIME ;
    time += SAMPLING_TIME ;
  }
}
/* ------------------------------------------------------------------------- */
```

16.3.1.3 Modul sgmain.c

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

#include "sgen.h"

void main (void)
{
  int    error     ;
  Signal my_signal ;

  error = init (&my_signal) ;

  if (!error)
  {
    switch (my_signal.type)
    {
      case 1:
        step(&my_signal) ;
        break ;

      case 2:
        sine(&my_signal) ;
        break ;

      case 3:
        triangle(&my_signal) ;
        break ;

      default:
        fprintf(stderr, "Eingabefehler!\n") ;

    } /* Ende von switch      */
  } /* Ende des else- Zweiges */
} /* Ende von main */
```

rtr Startseite
Einführung
Inhalt
1. Vorbemerkungen
I. Linux
II. C
8. Ein C-Programm
9. Variablen / Datentypen
10. Wichtige Sprachelemente
11. Funktionen
12. Benutzerdefinierte Typen
13. Übersetzen von C-Programmen
14. Arrays
15. Zeiger
16. Aufgaben
16.1 Ein Signalgenerator in C
16.2 Matrizenverarbeitung mit Doppel-Zeigern in C
16.3 Lösung zum Signalgenerator
16.4 Lösung zur Matrizenverarbeitung
III. C++

16.3.2 Matrizenverarbeitung mit Doppelzeigern

16.3.2.1 Makefile

```
# **** C-Compiler
CC = gcc
CFLAGS= -g
LINK.c=$(CC) $(CFLAGS) $(CPPFLAGS) $(LDFLAGS)

# **** Sources
SRC     = matrix.c main.c
OBJ     = $(SRC:.c=.o)

# **** Abhaengigkeiten speichern
.KEEP_STATE:

# **** Regeln
matrix:         $(OBJ)
                $(LINK.c) -o $@ $(OBJ)
clean:
        rm -f $(OBJ)
```

16.3.2.2 Header matrix.h

```
/* -------------------------- Typdefinition --------------------------------- */

typedef struct
{
 int     lines, cols ;    /* Zeilen- und Spaltenzahl            */
 float **mat_ptr     ;    /* Doppel-Zeiger auf Matrizenelemente */
}
Matrix ;

/* ------------------------- Funktionsprototypen ----------------------------- */

int  init  (Matrix *m_ptr) ;
void read  (Matrix *m_ptr) ;
void show  (Matrix *m_ptr) ;
int  trans (Matrix *m1_ptr, Matrix *m2_ptr) ;
void kill  (Matrix *m_ptr) ;
```

16.3.2.3 Modul matrix.c

```
#include <stdio.h>
#include "matrix.h"

int init (Matrix *m_ptr)
{
  int i, j, error = 0 ;

  /* --------------------- Dimension der Matrix festlegen ------------------- */
  printf("Zeilen :  ");
  scanf ("%d", &m_ptr->lines) ;

  printf("Spalten :  ");
  scanf ("%d", &m_ptr->cols) ;

  /* ------------------ Zeilenanfangszeiger allokieren ---------------------- */

  m_ptr->mat_ptr = (float**)malloc(m_ptr->lines * sizeof(float*)) ;

  if (m_ptr->mat_ptr != NULL)
  {
    /* --------------------------- Zeilen allokieren ------------------------ */
    for (i=0; i < m_ptr->lines; i++)
    {
      *(m_ptr->mat_ptr+i) = (float*)malloc(m_ptr->cols * sizeof(float));

      if (*(m_ptr->mat_ptr+i) == NULL)
      {
        error = 1 ;
```

```
        /* ---------- Erfolgreich allokierte Teile wieder freigeben --------- */
        for (j=0; j < i; j++)
        {
          free (*(m_ptr->mat_ptr+j)) ;
        }
        free (m_ptr->mat_ptr) ;

        break ;
      }
    }
  }
  else
  {
    error = 1 ;
  }

  return (error) ;
}
/* ------------------------------------------------------------------------ */
void read (Matrix *m_ptr)
{
  int i, j ;

  for (i=0; i < m_ptr->lines; i++)
  {
    printf("Zeile %d:\n", i+1) ;
    for (j=0; j < m_ptr->cols; j++)
    {
      printf("\t Element %d = ", j+1) ;
      scanf("%f", *(m_ptr->mat_ptr+i)+j) ;
    }
  }
}
/* ------------------------------------------------------------------------ */
void show (Matrix *m_ptr)
{
  int i, j ;

  for (i=0; i < m_ptr->lines; i++)
  {
    for (j=0; j < m_ptr->cols; j++)
    {
      printf("%-8.2f", *(*(m_ptr->mat_ptr+i)+j)) ;
    }
    printf("\n") ;
  }
}
/* ------------------------------------------------------------------------ */
int trans (Matrix *m1_ptr, Matrix *m2_ptr)
{
  int i, j ;

  if ((m1_ptr->lines != m2_ptr->cols) || (m1_ptr->cols != m2_ptr->lines))
  {
    return (1) ;
  }
  else
  {
    for (i=0; i < m2_ptr->lines; i++)
    {
      for (j=0; j < m2_ptr->cols; j++)
      {
        *(*(m2_ptr->mat_ptr+i)+j) = *(*(m1_ptr->mat_ptr+j)+i) ;
      }
    }
    return (0) ;
  }
}
/* ------------------------------------------------------------------------ */
void kill (Matrix *m_ptr)
{
  int i, j ;

  for (i=0; i < m_ptr->lines; i++)
  {
    free (*(m_ptr->mat_ptr+i)) ;
  }
  free (m_ptr->mat_ptr) ;

  m_ptr->mat_ptr = NULL ;
}
```

16.3.2.4 Modul main.c

```
#include <stdio.h>
#include "matrix.h"

void main (void)
{
```

```
  Matrix M1, M2 ;

  /* ------------------------- Matrizen allokieren -------------------------- */

  printf("1. Matrix:\n") ;
  if (init(&M1))
  {
    printf("Speicher konnte nicht allokiert werden!\n") ;
  }
  else
  {
    printf("\n2. Matrix:\n") ;
    if (init(&M2))
    {
      printf("Speicher konnte nicht allokiert werden!\n") ;
    }

    /* ----------------- Elemente der 1. Matrix einlesen -------------------- */
    read(&M1) ;

    /*  ----------------- 1. Matrix formatiert ausgeben -------------------- */
    printf("M1:\n") ;
    show (&M1) ;

    if (trans(&M1, &M2))
    {
      printf("Transponieren nicht moeglich!\n") ;
    }
    else
    {
      printf("\nM1 - transponiert:\n") ;
      show(&M2) ;
    }
    /* ---------------- Speicher beider Matrizen freigeben ------------------ */

    kill(&M1) ;
    kill(&M2) ;
  }
}
```

- rtr Startseite
- Einführung
- Inhalt
- 1. Vorbemerkungen
- I. Linux
- II. C
- III. C++
  - 17. Das Klassenkonzept
  - 18. Basistechniken
  - 19. Templates
  - 20. Die Standardbibliothek STL
  - 21. Klassen II
  - 22. Fortgeschrittene Techniken
  - 23. Vererbung
  - 24. Polymorphie
  - 25. Aufgaben

Einleitung

C++ ist eine eigene Programmiersprache
C++ ist objektorientiert

Motivation für objektorientiertes Programmieren:

- Kostendruck: Wiederverwenden von Quellcode, Wartung
- Qualitätsanforderungen: sicherheitskritische Anwendung, Zufriedenheit der Kunden

Programmiersprache soll den Programmierer bei folgenden Anforderungen unterstützen:

- Wiederverwendbarkeit, Generisches Programmieren
- Wartbarkeit
- Zuverlässigkeit
- Korrektheit
- Unabhängiges Testen von Programmteilen

Überblick über diesen Kurs:

- Grundlagen: Die Klasse
- Unterschiede zwischen C und C++
- Klassen II: Operatoren
- Wiederverwendung von Code
- Die Standardbibliothek
- Software-Engineering: Wie komme ich zu einem guten Programm?

Programmieren in C++

Änderungen:

- Compiler: g++
- Sourcefiles: Endung .cpp (oder .cc)
- Header: Endung .h oder .hpp
- zusätzliche Systembibliothek: libstdc++ (wird automatisch eingebunden)
- zusätzliche Klassen: STL (Standard Template Library)
- Header aus der C++-Bibliothek (libstdc++): ohne .h
  ```
  #include <iostream>
  ```
- Header aus der C-Bibliothek: ohne .h, gekennzeichnet durch ein vorangestelltes c
  ```
  #include <cstdio>
  ```

```
#include <otherlib/header.h>
#include "myheader.h"
```

rtr Startseite
Einführung
Inhalt
1. Vorbemerkungen
I. Linux
II. C
III. C++
> 17. Das Klassenkonzept
17.1 Syntax einer Klassendeklaration
17.2 Implementierung der Klassenmethoden
17.3 Verwendung einer Klasse
17.4 Konstruktor: Erzeugen eines Objekts
17.5 Destruktor
17.6 Trennung von Deklaration und Definition
17.7 Beispiel
18. Basistechniken
19. Templates
20. Die Standardbibliothek STL
21. Klassen II
22. Fortgeschrittene Techniken
23. Vererbung
24. Polymorphie
25. Aufgaben

17. Das Klassenkonzept

Eine Klasse kapselt Daten und Funktionalität in einer Blackbox
Beispiel: Rechnen mit komplexen Zahlen
Rückblick: Realisierung in C

- Implementierung in getrenntem Header/C-File: complex.h, complex.c
- neue Datentypen werden als struct definiert
- neue Funktionen verwenden den Datentyp, implementieren Rechenoperationen
- im Header: Datentypen und Funktionen werden deklariert, um sie in anderen Programmteilen verwenden zu können
- lose Kopplung zwischen Daten und Funktionen, kein Schutz der Daten

Beispiel:

```
typedef struct
{
   double m_re;
   double m_im;
} Complex;

Complex complexAdd(Complex c1,Complex c2)
{
   Complex sum;
   sum.m_re = c1.m_re + c2.m_re;
   sum.m_im = c1.m_im + c2.m_im;
   return sum;
}
```

In C++ werden neue Typen als Klassen implementiert.

- Die Klasse ist eine Typvereinbarung (Deklaration) und besteht aus:
  `Daten`
  `Methoden`(auch: member-functions)
- Eine Variable, deren Typ eine Klasse ist heißt
  `Objekt` oder `Instanz`
- Die Daten und Methoden einer Klasse können frei zugänglich sein, oder gekapselt und somit für die Außenwelt verborgen sein.
- Komplexe Datentypen und Funktionsgruppen sind möglich
- Daten und die darauf arbeitenden Funktionen bilden eine Einheit
- Implementierungsdetails werden verborgen

rtr Startseite
Einführung
Inhalt
1. Vorbemerkungen
I. Linux
II. C
III. C++
> 17. Das Klassenkonzept
> 17.1 Syntax einer Klassendeklaration
17.2 Implementierung der Klassenmethoden
17.3 Verwendung einer Klasse
17.4 Konstruktor: Erzeugen eines Objekts
17.5 Destruktor
17.6 Trennung von Deklaration und Definition
17.7 Beispiel
18. Basistechniken
19. Templates
20. Die Standardbibliothek STL
21. Klassen II
22. Fortgeschrittene Techniken
23. Vererbung
24. Polymorphie
25. Aufgaben

17.1 Syntax einer Klassendeklaration

Die Bekanntgabe (oder Deklaration) einer Klasse erfolgt mit dem Keyword

`class`

Ob von außen auf Variablen und Methoden einer Klasse zugegriffen werden darf, wird über folgende Schlüsselwörter angegeben:

`private` - privat, von außen nicht benutzbar

`public` - öffentlich, von außen benutzbar

Als private deklarierte Daten und Methoden sind nicht nur vor dem Zugriff von außen geschützt, sondern auch vor dem Zugriff von abgeleiteten Klassen. Um diesen dennoch Zugriff zu geben, müssen diese mit folgendem Schlüsselwort markiert werden:

`protected` - geschützt, von außen nicht benutzbar

Der Zugriff auf Methoden und vor allem auf Daten sollte möglichst restriktiv gehandhabt werden. Um dennoch Daten von außen lesbar und veränderbar zu machen, muß man entsprechende Methoden vorsehen. Solche Daten werden auch als Properties einer Klasse bezeichnet.

Deklaration einer Klasse:

```
class ClassName
{
     /* Private Daten und Methoden */
   public:
     /* oeffentliche Daten und Methoden */
   private:
     /* weitere private Daten und Methoden */
};
```

rtr Startseite
Einführung
Inhalt
1. Vorbemerkungen
I. Linux
II. C
III. C++
> 17. Das Klassenkonzept
17.1 Syntax einer Klassendeklaration
> 17.2 Implementierung der Klassenmethoden
17.3 Verwendung einer Klasse
17.4 Konstruktor: Erzeugen eines Objekts
17.5 Destruktor
17.6 Trennung von Deklaration und Definition
17.7 Beispiel
18. Basistechniken
19. Templates
20. Die Standardbibliothek STL
21. Klassen II
22. Fortgeschrittene Techniken
23. Vererbung
24. Polymorphie
25. Aufgaben

17.2 Implementierung der Klassenmethoden

Alle Elemente einer Klasse (Funktionen, Variablen) müssen außerhalb der Klassendeklaration als solche gekennzeichnet werden. Dazu wird der Name der Klasse dem Elementnamen vorangestellt: ClassName::function()
Definition einer Klasse in der Quellendatei:

```
bool ClassName::functionname()
{
   /* do something */
   return true;
}
```

Die in der Klassendeklaration enthaltenen Methoden werden im cpp-File definiert. Die Deklaration der Klasse muß im Header-Files stehen, wenn die Klasse in mehr als einem Source-File verwendet wird.
In Klassenmethoden kann auf alle Daten und Methoden der Klasse direkt zugegriffen werden. Auf Daten von übergebenen Objekten, der gleichen Klasse kann ebenfalls zugegriffen werden. Ein Pointer auf sich selber ist in der Variable **this** in jeder Klasse vorhanden.

```
void Complex::add(Complex c2)
{
   re = re + c2.re;
   this->im = *this.im + c2.im
   return true;
}
```

Der Operator "->" derefenziert einen Pointer und daher gleichwertig mit der Schreibweise "(*this).element".

17.2.1 Beispiel 1

Konstruktion einer Klasse mit privaten Daten und öffentlichen Methoden

```
 1 #include <iostream>
 2
 3 /** declaration of class for complex numbers  */
 4 class Complex
 5 {
 6 private:
 7     /* data */
 8     double m_re;
 9     double m_im;
10
11 public:
12     /* methods */
13     void set(double re, double im)
14     {
15         m_re=re;
16         m_im=im;
17     }
18     void print();
19
20     void setRe(double re) { m_re = re; }
21     double re() { return m_re; }
22     void setIm(double im) { m_im = im; }
23     double im() { return m_im; }
24 };  // end of class Complex
25
26
27 /** Implementation of print()
28     This methods prints the complex number to an output stream */
29 void Complex::print()
30 {
31     cout << m_re << " + j(" << m_im << ")" << endl;
32 }
```

rtr Startseite
Einführung
Inhalt
1. Vorbemerkungen
I. Linux
II. C
III. C++
> 17. Das Klassenkonzept
17.1 Syntax einer Klassendeklaration
17.2 Implementierung der Klassenmethoden
> 17.3 Verwendung einer Klasse
17.4 Konstruktor: Erzeugen eines Objekts
17.5 Destruktor
17.6 Trennung von Deklaration und Definition
17.7 Beispiel
18. Basistechniken
19. Templates
20. Die Standardbibliothek STL
21. Klassen II
22. Fortgeschrittene Techniken
23. Vererbung
24. Polymorphie
25. Aufgaben

17.3 Verwendung einer Klasse

Eine Klasse entspricht einer Typdeklaration und wird ebenso verwendet. Konkrete Objekte werden folgendermaßen erzeugt:

```
Complex c1;
```

Die Methoden einer Klasse sind nicht mit globalen Funktionen vergleichbar. Sie können nur für ein konkretes Objekt verwendet werden und werden über das Objekt referenziert. Die Methoden arbeiten auf den Daten des jeweiligen Objekts und das Resultat hängt somit von dem Objekt ab, für das die Funktion aufgerufen wird.

Beispiel:

```
int main()
{
   Complex c1;
   Complex c2;

   c1.set(10,30);  // setting values of c1
   c2.setRe(5);
   c2.setIm(10);

   c1.print();  // output: "10 + j(30)"
   c2.print();  // output: "5 + j(10)"
   return 0;
}
```

rtr Startseite
Einführung
Inhalt
1. Vorbemerkungen
I. Linux
II. C
III. C++
17. Das Klassenkonzept
17.1 Syntax einer Klassendeklaration
17.2 Implementierung der Klassenmethoden
17.3 Verwendung einer Klasse
17.4 Konstruktor: Erzeugen eines Objekts
17.5 Destruktor
17.6 Trennung von Deklaration und Definition
17.7 Beispiel
18. Basistechniken
19. Templates
20. Die Standardbibliothek STL
21. Klassen II
22. Fortgeschrittene Techniken
23. Vererbung
24. Polymorphie
25. Aufgaben

17.4 Konstruktor: Erzeugen eines Objekts

Ein Objekt einer Klasse wird durch einen Konstruktor erzeugt.
Der Konstruktor wird automatisch ausgeführt, wenn eine Instanz von einer Klasse erzeugt wird. Konstruktoren sind Methoden (member-functions) mit dem gleichen Namen wie die Klasse. Hier können Initialisierungen ausgeführt und Speicher reserviert werden. Return-Werte gibt es keine. Ein Konstruktor kann aber Parameter übergeben bekommen.

```
Complex();
```

In C++ haben eingebaute Datentypen (int, double, etc.) ebenfalls Konstruktoren. Sie reservieren Speicher, führen aber keine Initialisierung durch!
Der Ablauf bei der Konstruktion mit einem Default-Konstruktor sieht folgendermaßen aus:

1. Anlegen der Klasseninformation im Speicher
2. Initialisierung der 1. Variable: Aufruf von `double()` für `m_re`
3. Initialisierung der 2. Variable: Aufruf von `double()` für `m_im`
4. Ausführung der Implementierung des Konstruktors

Eigene Implementierung der Konstruktoren mit Zuweisung:

```
Complex::Complex()
{
    m_re = 0.0;
    m_im = 0.0;
}
```

Eigene Implementierung der Konstruktoren mit Initialisierung der Daten:

```
Complex::Complex()
  : m_re(0.0), m_im(0.0)    // Initialisierung in der Reihenfolge
                             //der Deklaration in der Klasse
{}
```

Wenn kein Konstruktor implementiert ist, wird er vom Compiler automatisch angelegt.

TECHNISCHE UNIVERSITÄT DARMSTADT

Fachgebiet Regelungstheorie und Robotik

Impressum | Druckversion

REGELUNGSTHEORIE UND ROBOTIK rtr

Sie befinden sich: > TU Darmstadt > ETIT > IAT > RTR > Lehre > E-Learning > C/C++-Onlinekurs > III. C++ > 17. Das Klassenkonzept > 17.5 Destruktor

rtr Startseite
Einführung
Inhalt
1. Vorbemerkungen
I. Linux
II. C
III. C++
17. Das Klassenkonzept
17.1 Syntax einer Klassendeklaration
17.2 Implementierung der Klassenmethoden
17.3 Verwendung einer Klasse
17.4 Konstruktor: Erzeugen eines Objekts
17.5 Destruktor
17.6 Trennung von Deklaration und Definition
17.7 Beispiel
18. Basistechniken
19. Templates
20. Die Standardbibliothek STL
21. Klassen II
22. Fortgeschrittene Techniken
23. Vererbung
24. Polymorphie
25. Aufgaben

17.5 Destruktor

Destruktoren zerstören ein Objekt einer Klasse. Sie kümmern sich um das Freigeben des Speichers.
Die Deklaration ist ähnlich wie die des Konstruktors, außer daß sie ein ~ vor dem Klassennamen haben:

    ~Complex();

Ein Destruktor wird automatisch aufgerufen, wenn das Programm beendet oder wenn der Gültigkeitsbereich eines Objekts endet, z.B. am Ende einer Funktion.
Der Ablauf der Zerstörung eines Objekts ist umgekehrt zur Konstruktion:

1. Ausführung der Implementierung des Destruktors
2. Zerstörung der 2. Variable: Aufruf von `~double()` für `m_im`
3. Zerstörung der 1. Variable: Aufruf von `~double()` für `m_re`
4. Freigeben der Klasseninformation im Speicher

Auch hier legt der Compiler automatisch eine optimierte Version an, wenn keine eigene Implementierung vorhanden ist.

TECHNISCHE UNIVERSITÄT DARMSTADT

Fachgebiet Regelungstheorie und Robotik

Impressum | Druckversion

REGELUNGSTHEORIE UND ROBOTIK rtr

Sie befinden sich: > TU Darmstadt > ETIT > IAT > RTR > Lehre > E-Learning > C/C++-Onlinekurs > III. C++ > 17. Das Klassenkonzept > 17.6 Trennung von Deklaration und Definition

rtr Startseite
Einführung
Inhalt
1. Vorbemerkungen
I. Linux
II. C
III. C++
17. Das Klassenkonzept
17.1 Syntax einer Klassendeklaration
17.2 Implementierung der Klassenmethoden
17.3 Verwendung einer Klasse
17.4 Konstruktor: Erzeugen eines Objekts
17.5 Destruktor
17.6 Trennung von Deklaration und Definition
17.7 Beispiel
18. Basistechniken
19. Templates
20. Die Standardbibliothek STL
21. Klassen II
22. Fortgeschrittene Techniken
23. Vererbung
24. Polymorphie
25. Aufgaben

17.6 Trennung von Deklaration und Definition

Deklaration:
Klassendeklaration, im Header - Endung `.h`
Definition:
Definition der Methoden, im C++-File - Endung `.cpp`
Nach Einbinden eines Headers kann eine Klasse verwendet werden.

```
#include "complex.h"
```

Achtung!
Klassendeklarationen sind Typdeklarationen und dürfen pro Compiliereinheit (cpp-File) nur einmal eingebunden werden, da sonst Klassen doppelt deklariert würden. Dies erlaubt der Compiler jedoch nicht.

Abhilfe:
Präprozessor-Defines in jedem Header-File

```
#ifndef COMPLEX_H
#define COMPLEX_H

class Complex
{
    ...
};

#endif
```

rtr Startseite
Einführung
Inhalt
1. Vorbemerkungen
I. Linux
II. C
III. C++
17. Das Klassenkonzept
17.1 Syntax einer Klassendeklaration
17.2 Implementierung der Klassenmethoden
17.3 Verwendung einer Klasse
17.4 Konstruktor: Erzeugen eines Objekts
17.5 Destruktor
17.6 Trennung von Deklaration und Definition
17.7 Beispiel
18. Basistechniken
19. Templates
20. Die Standardbibliothek STL
21. Klassen II
22. Fortgeschrittene Techniken
23. Vererbung
24. Polymorphie
25. Aufgaben

17.7.1 example.h

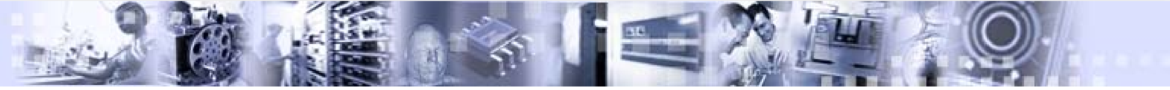
```
#ifndef COMPLEX_H
#define COMPLEX_H

/** class to handle complex numbers  */
class Complex
{
private:
    double m_re;
    double m_im;

public:
    /** Default Constructor */
    Complex() : m_re(0.0), m_im(0.0) {}

    /** Destructor */
    ~Complex(){}

    /* methods */
    void set(double re, double im);
    void print();
    void setRe(double re) { m_re = re; }
    double re()           { return m_re; }
    void setIm(double im) { m_im = im; }
    double im()           { return m_im; }

};  // end of class Complex

#endif
```

17.7.2 example.cpp

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

/* methods of class Complex*/
void set(double re, double im)
{
    m_re=re;
    m_im=im;
}

/** This methods prints the complex number to an output stream */
void Complex::print()
{
    cout << m_re << " + j(" << m_im << ")" << endl;
}
```

17.7.3 main.cpp

```
#include "example.h"

int main()
{
    Complex c1;   // creation of instance c1 with default values

    c1.print();   // output

    return 0;
}
```



rtr Startseite
Einführung
Inhalt
1. Vorbemerkungen
I. Linux
II. C
III. C++
17. Das Klassenkonzept
> 18. Basistechniken
> 18.1 Kommentare
18.2 Speicherverwaltung mit new / delete
18.3 Datentyp bool
18.4 Strings
18.5 Ein- und Ausgabe in Streams
18.6 Überladen von Funktionen
18.7 Default-Parameter
19. Templates
20. Die Standardbibliothek STL
21. Klassen II
22. Fortgeschrittene Techniken
23. Vererbung
24. Polymorphie
25. Aufgaben

18.1 Kommentare

C-Kommentare:

```
double re /* Realteil */
double im /* Imaginärteil */
```

Zusätzlich gibt es in C++ folgende Kommentare:

```
double re // Realteil
double im // Imaginärteil
```

Dies ist auch praktisch zum Auskommentieren von Zeilen:

```
void main()
{
  now.set(10,30,00);
  //now.print();
}
```

Dokumentation schreiben ist wichig, denn man selber vergißt schnell, was man sich beim programmieren gedacht hat.
Dokumentationstools wie z.B. doxygen unterstützen beim Auswerten von Dokumentation. Berücksicht werden Kommentare, die folgendermaßen beginnen:

```
/** Realteil */
double re
//! Imaginärteil
double im
```

Sie werden auf das darauf folgende Syntaxelement bezogen.

rtr Startseite
Einführung
Inhalt
1. Vorbemerkungen
I. Linux
II. C
III. C++
17. Das Klassenkonzept
> 18. Basistechniken
18.1 Kommentare
> 18.2 Speicherverwaltung mit new / delete
18.3 Datentyp bool
18.4 Strings
18.5 Ein- und Ausgabe in Streams
18.6 Überladen von Funktionen
18.7 Default-Parameter
19. Templates
20. Die Standardbibliothek STL
21. Klassen II
22. Fortgeschrittene Techniken
23. Vererbung
24. Polymorphie
25. Aufgaben

18.2 Speicherverwaltung mit new / delete

Wie auch in C muß bei der Verwendung von Zeigern für dynamisches Speichermanagement Speicher reserviert und wieder freigegeben werden.
Statt `malloc` und `free` verwendet man in C++ diese Operatoren:

- `new` - Aufruf des Konstruktors
- `delete` - Aufruf des Destruktors

Dazu muß man nicht die Größe des benötigten Speicherplatzes kennen. Statt dessen wird der Konstruktor der Klasse und seiner Daten-Elemente aufgerufen. Der Operator new liefert einen Zeiger auf das im Speicher erzeugte Objekt zurück.
`malloc/free` benutzt ein anderes Speichermanagement als `new/delete`. Verwende nie beide Stile zusammen!!!
Eine Variable, die mit `new` erzeugt wurde, muß auch mit `delete` wieder gelöscht werden.

```
Complex *p_c1 = new Complex;            // Pointer auf die Instanz von Complex
delete p_c1;                            // Speicher freigeben für p_c1
Complex *p_c2 = new Complex(10,20);     // mit Initialisierung
Complex *p_c3 = new Complex(&p_c2);     // als Kopie (der Zeiger auf p_c1
                                        // muß dereferenziert werden)
delete p_c2;                            // Speicher freigeben für p_c2
delete p_c3;                            // Speicher freigeben für p_c3
long *var = new [4] long;               // Array vom Typ long mit Größe 4
delete [] var;                          // Achtung: bei Arrays wird [] benötigt!
```

Mit dynamischer Speicherverwaltung hängt die Lebenszeit eines Objekts nicht mehr von seinem Gültigkeitsbereich ab, sondern kann (und muß) vom Programmierer frei gewählt werden. D.h. daß der Programmierer auch dafür Sorge tragen muß, daß einmal angelegter Speicher auch wieder freigegeben wird.

TECHNISCHE UNIVERSITÄT DARMSTADT

Fachgebiet Regelungstheorie und Robotik

Impressum | Druckversion

REGELUNGSTHEORIE UND ROBOTIK rtr

Sie befinden sich: > TU Darmstadt > ETIT > IAT > RTR > Lehre > E-Learning > C/C++-Onlinekurs > III. C++ > 18. Basistechniken > 18.3 Datentyp bool

rtr Startseite
Einführung
Inhalt
1. Vorbemerkungen
I. Linux
II. C
III. C++
17. Das Klassenkonzept
18. Basistechniken
18.1 Kommentare
18.2 Speicherverwaltung mit new / delete
18.3 Datentyp bool
18.4 Strings
18.5 Ein- und Ausgabe in Streams
18.6 Überladen von Funktionen
18.7 Default-Parameter
19. Templates
20. Die Standardbibliothek STL
21. Klassen II
22. Fortgeschrittene Techniken
23. Vererbung
24. Polymorphie
25. Aufgaben

18.3 Datentyp bool

Der Datentyp `bool` kann zwei Werte annehmen:

`true` oder `false`

Vergleiche wie a<b liefern bool zurück.
Bei der Typkonvertierung von `bool` nach `int` wird folgendermaßen ersetzt:

0 <=> `false`

1 <=> `true`

Bei der Konvertierung von `int` nach `bool` wird 0 zu `false` und jeder andere Wert wird zu `true`.

TECHNISCHE UNIVERSITÄT DARMSTADT

Fachgebiet Regelungstheorie und Robotik

Impressum | Druckversion

REGELUNGSTHEORIE UND ROBOTIK rtr

Sie befinden sich: > TU Darmstadt > ETIT > IAT > RTR > Lehre > E-Learning > C/C++-Onlinekurs > III. C++ > 18. Basistechniken > 18.4 Strings

rtr Startseite
Einführung
Inhalt
1. Vorbemerkungen
I. Linux
II. C
III. C++
17. Das Klassenkonzept
18. Basistechniken
18.1 Kommentare
18.2 Speicherverwaltung mit new / delete
18.3 Datentyp bool
18.4 Strings
18.5 Ein- und Ausgabe in Streams
18.6 Überladen von Funktionen
18.7 Default-Parameter
19. Templates
20. Die Standardbibliothek STL
21. Klassen II
22. Fortgeschrittene Techniken
23. Vererbung
24. Polymorphie
25. Aufgaben

18.4 Strings

Die Klasse `string` implementiert Zeichenketten und ist wesentlich besser handhabbar als `char*` .

Anwendungsbeispiele:

```
string wort1;
string wort2("test");

wort1 = "nochmal test";
string wort3 = wort1 + " " + wort2;  // ergibt: "nochmal test test"
char c = wort1[1];                    // c ist 'o'

if (!wort1.empty())
  wort1.clear()                       // jetzt ist wort1 leer
```

Wenn eine C-Funktion verwendet wird, die als Parameter char* erwartet, erhält man den sogenannten C-String mit der Funktion

```
ofstream outputfile(filename.c_str());
```

rtr Startseite
Einführung
Inhalt
1. Vorbemerkungen
I. Linux
II. C
III. C++
17. Das Klassenkonzept
18. Basistechniken
18.1 Kommentare
18.2 Speicherverwaltung mit new / delete
18.3 Datentyp bool
18.4 Strings
18.5 Ein- und Ausgabe in Streams
18.6 Überladen von Funktionen
18.7 Default-Parameter
19. Templates
20. Die Standardbibliothek STL
21. Klassen II
22. Fortgeschrittene Techniken
23. Vererbung
24. Polymorphie
25. Aufgaben

18.5 Ein- und Ausgabe in Streams

In C und C++ geschieht die Ein- und Ausgabe über Streams. Ein Stream ist ein Gerät (device), in das Daten geschrieben und aus dem Daten gelesen werden können, z.B eine Datei, ein Drucker, eine Netzwerk-Verbindung oder der Bildschirm. Auf dem Bildschirm gibt es 3 Streams:

- Input stream
- Output stream
- Error stream

Unter Unix kann man das auch auf der Konsole tun. Man liest aus einem File, indem man "<file" schreibt, or schreibt in ein File mit ">file". Fehlermeldungen aus dem Error Stream lenkt man mit "2> file" in eine Datei.
Was man in C mit `printf` und `scanf` gemacht hat, wird in C++ von den Streams `cin`, `cout` und `cerr` erledigt.

Man kann eine Variable mit folgendem Ausdruck einlesen:

```
cin >> test_variable;
```

oder schreiben mit:

```
cout << "Dies ist die Testvariable: " << test_variable << endl;
```

Das I/O über Streams ist im Header `iostream` deklariert, der den C-Header stdio.h ersetzt:

```
#include <iostream>
```

Bemerkung:
`cerr` wird vor `cout` verarbeitet - wenn man beide Streams verwendet, kann die Ausgabe somit in anderer Reihenfolge erfolgen, in der man die Ausgabe gemacht hat.

Beispiel für eine Ausgabe in eine Datei:

```
#include <fstream>

...
ofstream* out = new ofstream("out.txt");
out << "Diese Datei heisst out.txt" << endl;
delete out;
...

fstream out;
string first_word;
out.open("out.txt");
out >> first_word;
out << first_word;
out.close();
...
```

rtr Startseite
Einführung
Inhalt
1. Vorbemerkungen
I. Linux
II. C
III. C++
17. Das Klassenkonzept
> 18. Basistechniken
18.1 Kommentare
18.2 Speicherverwaltung mit new / delete
18.3 Datentyp bool
18.4 Strings
18.5 Ein- und Ausgabe in Streams
> 18.6 Überladen von Funktionen
18.7 Default-Parameter
19. Templates
20. Die Standardbibliothek STL
21. Klassen II
22. Fortgeschrittene Techniken
23. Vererbung
24. Polymorphie
25. Aufgaben

18.6 Überladen von Funktionen

Unter dem Überladen von Funktionen versteht man die Deklaration und Implementierung von Funktionen mit gleichem Funktionsnamen, aber unterschiedlichen Parameterlisten:

```
void printErrorMsg(const char* prefix, const char* text, int linenumber);
void printErrorMsg(const char* text, int linenumber);
void printErrorMsg(const char* prefix, const char* text);
```

Das Überladen ermöglicht es, Funktionen und Methoden mit gleicher Funktionalität (wie hier die Ausgabe einer Fehlermeldung) zusammenzufassen. Dies erhöht die Flexibilität und Lesbarkeit eines Programmes.

Bemerkung:
Die Verwendung Angabe von Default-Werten für Funktionsparamter ist auch eine Art des Überladens.

rtr Startseite
Einführung
Inhalt
1. Vorbemerkungen
I. Linux
II. C
III. C++
17. Das Klassenkonzept
> 18. Basistechniken
18.1 Kommentare
18.2 Speicherverwaltung mit new / delete
18.3 Datentyp bool
18.4 Strings
18.5 Ein- und Ausgabe in Streams
18.6 Überladen von Funktionen
> 18.7 Default-Parameter
19. Templates
20. Die Standardbibliothek STL
21. Klassen II
22. Fortgeschrittene Techniken
23. Vererbung
24. Polymorphie
25. Aufgaben

18.7 Default-Parameter

C++ erlaubt bei Funktionsdeklarationen die Angabe von Default-Werten, falls die entspechenden Übergabewerte beim Aufruf fehlen. Das entspricht einem Überladen von Methoden.
Beispiel:

```
void myFunction( int i, int j=0, double d=47.11, MyClass* class=0 )

myFunction(1);
myFunction(1, 2);
myFunction(1, 2, 3.141);
myFunction(1, 2, 3.141, classpointer);
```

Wichtig:
Die Parameter, für die Default-Werte angegeben werden, müssen am Ende der Parameterliste stehen.

rtr Startseite
Einführung
Inhalt
1. Vorbemerkungen
I. Linux
II. C
III. C++
17. Das Klassenkonzept
18. Basistechniken
19. Templates
20. Die Standardbibliothek STL
21. Klassen II
22. Fortgeschrittene Techniken
23. Vererbung
24. Polymorphie
25. Aufgaben

19.1 Templates: Parametrisierbare Klassen

Interne Datentypen können durch Template-Parameter variiert werden.

Beispiel:

```
class Complex
{
private:
double m_re;
double m_im;
public:
Complex(double re, double im);
double re();
double im();
Complex operator+(const Complex c1, const Complex c2);
}
```

Complex verwendet den Datentyp double, dies ist hart kodiert. Evtl. sind aber auch komplexe Zahlen vom Typ float und int wünschenswert. Implementierung von `ComplexFloat`, `ComplexInt`?

Realisierung als Template:

```
template <class TRe, class TIm>
class Complex
{
private:
TRe m_re;
TIm m_im;
public:
Complex(TRe re, TIm im);
TRe re();
TIm im();
Complex& operator+(const Complex& c1, const Complex& c2);
}

template <class TRe, class TIm>
TRe Complex<TRe,TIm>::re()
{
return m_re;
}
```

Wenn eine solche Klasse verwendet wird, müssen die gewünschten Datentypen angegeben werden:

```
Complex<double,double> c1(1.2,3);
cout << c1.re();
```

Die Implementierung von Templates ist sehr fehleranfällig, die Anwendung einfach. Für Templates gibt es nur Header und keine Sourcefiles, da ein Template nur eine Vorlage für eine Klasse ist. Diese Vorlage kann erst compiliert werden, wenn ein Typ angegeben ist.

Tip für's Implementieren:

Zunächst Klasse für einen Datentyp implementieren und dann in Template umwandeln.

TECHNISCHE UNIVERSITÄT DARMSTADT

Fachgebiet Regelungstheorie und Robotik

Impressum | Druckversion

REGELUNGSTHEORIE UND ROBOTIK rtr

rtr Startseite
Einführung
Inhalt
1. Vorbemerkungen
I. Linux
II. C
III. C++
17. Das Klassenkonzept
18. Basistechniken
19. Templates
20. Die Standardbibliothek STL
20.1 STL: Existierende wiederverwendbare Klassen
20.2 Iteratoren
20.3 Algorithmen
20.4 Andere Bibliotheken
21. Klassen II
22. Fortgeschrittene Techniken
23. Vererbung
24. Polymorphie
25. Aufgaben

20.1 STL: Existierende wiederverwendbare Klassen

Abkürzung für: **Standard Template Library**
Die STL ist Teil der Standard-C++-Bibliothek und enthält u. a.:

- Streams (für die Ein-/Ausgabe)
- Container (zum Aufbewahren von mehreren Objekten des gleichen Typs)
- Funktionsobjekte: z.B. Iteratoren
- Algorithmen, die auf Containern arbeiten

Beispiele für Container:

- vector
- list
- map
- string
- (complex)

Verwendung:

```
#include <iostream>
#include <string>
#include <list>

int main()
{
string wort1("wort1");
string wort2("wort2");

list<string> mylist;
mylist.push_back(wort1);
mylist.push_back(wort2);

if (!mylist.empty())
cout << "mylist enthaelt " << mylist.size()+1 << " Elemente." << endl;
return 0;
}
```

Ausgabe:
mylist enthaelt 2 Elemente.

rtr Startseite
Einführung
Inhalt
1. Vorbemerkungen
I. Linux
II. C
III. C++
17. Das Klassenkonzept
18. Basistechniken
19. Templates
20. Die Standardbibliothek STL
20.1 STL: Existierende wiederverwendbare Klassen
20.2 Iteratoren
20.3 Algorithmen
20.4 Andere Bibliotheken
21. Klassen II
22. Fortgeschrittene Techniken
23. Vererbung
24. Polymorphie
25. Aufgaben

20.2 Iteratoren

Iteratoren sind Klassen, die sich an Struktur-Klassen wie z.B. list oder vector "entlang hangeln" können. Iteratoren verhalten sich **ähnlich wie Zeiger** - über Dereferenzierung erhält man Zugriff auf das Objekt, an dem er gerade steht.

Beispiel:

```
#include <iostream>
#include <string>
#include <list>


int main()
{
string wort1("wort1");
string wort2("wort2");


list<string> mylist;
mylist.push_back(wort1);
mylist.push_back(wort2);


list<string>::iterator myIter;


myIter = mylist.begin();
cout << *myIter << " "; // Ausgabe: wort1
myIter++;
cout << *myIter << endl; // Ausgabe: wort2
// Alternative:
for (myIter=mylist.begin(); myIter!=mylist.end() ; myIter++)
cout << *myIter << " ";
return 0;
}
```

Ausgabe:

```
wort1 wort2
wort1 wort2
```

rtr Startseite
Einführung
Inhalt
1. Vorbemerkungen
I. Linux
II. C
III. C++
17. Das Klassenkonzept
18. Basistechniken
19. Templates
20. Die Standardbibliothek STL
20.1 STL: Existierende wiederverwendbare Klassen
20.2 Iteratoren
20.3 Algorithmen
20.4 Andere Bibliotheken
21. Klassen II
22. Fortgeschrittene Techniken
23. Vererbung
24. Polymorphie
25. Aufgaben

20.3 Algorithmen

Die STL bringt fertige Algorithmen zur Manipulation von Container mit.
Beispiel: Suche nach dem ersten Element, das gleich 7 ist

```
void f(list<int> & c)
{
list<int>::iterator p = find(c.begin(), c.end(), 7);
}
```

oder: Suche nach dem ersten Element, das kleiner 7 ist

```
bool kleiner_als_7(int v)
{
return v<7;
}


void f(list<int> & c)
{
list<int>::iterator p = find(c.begin(), c.end(), kleiner_als_7);
}
```

Andere Algorithmen sind u.a.:

- for_each()
- count()
- search()
- transform()
- copy()
- swap()
- fill()
- generate()
- sort()
- merge()
- min_element()
- ...

rtr Startseite
Einführung
Inhalt
1. Vorbemerkungen
I. Linux
II. C
III. C++
17. Das Klassenkonzept
18. Basistechniken
19. Templates
20. Die Standardbibliothek STL
20.1 STL: Existierende wiederverwendbare Klassen
20.2 Iteratoren
20.3 Algorithmen
20.4 Andere Bibliotheken
21. Klassen II
22. Fortgeschrittene Techniken
23. Vererbung
24. Polymorphie
25. Aufgaben

20.4 Andere Bibliotheken

Weitere Quellen für fertige Klassen:

- www.boost.org
  Von Mitgliedern des ANSI C++-Kommitees gegründet, bewerben sich für den Standard Beispiele: Random-Klasse, Tokenizer, Matrizen, Graphen-Bibliothek, ...
- www.sourceforge.net
  OpenSource-Projekte. Bibliotheken und fertige Programme. Wenn eine Bibliothek unter der LGPL oder einer BSD-Lizenz steht, darf sie gegen kommerziellen Code gelinkt werden.
- ftp.llp.fu-berlin.de/lsoft/
  Linksammlung zu wissenschaftlichen Projekten unter Linux

Ein abschliessendes Wort zur Standard-Bibliothek: Der ANSI-C++-Standard ist vergleichsweise jung (1998), aber inzwischen unterstützen ihn alle aktuellen Compiler weitgehend. Man sollte versuchen soweit wie möglich ANSI-konform zu programmieren, um Software auch auf anderen Systemen oder mit anderen Compilern compilieren zu können.

rtr Startseite
Einführung
Inhalt
1. Vorbemerkungen
I. Linux
II. C
III. C++
17. Das Klassenkonzept
18. Basistechniken
19. Templates
20. Die Standardbibliothek STL
> 21. Klassen II
> 21.1 Alternative Konstruktoren
21.2 Copy-Konstruktor
21.3 Zuweisungsoperator
21.4 Automatisch generierte Funktionen
22. Fortgeschrittene Techniken
23. Vererbung
24. Polymorphie
25. Aufgaben

21.1 Alternative Konstruktoren

Es ist möglich mehr als einen Konstruktor zu implementieren, sie müssen sich nur in der Anzahl und/oder dem Typ der Argumente unterscheiden.

Standardkonstruktor:

`Complex();`

Konstruktor mit Werten zur Initialisierung:

`Complex(double re, double im);`

Wenn in der Klasse kein Konstruktor enthalten ist, erzeugt der Compiler eine Default-Implementierung. Der Default-Konstruktor, den der Compiler anlegt, ist meist sehr gut optimiert. Man sollte deswegen nur Konstruktoren anlegen, wenn dies nötig ist. Im Fall der Complex-Klasse ist dies sinnvoll, da die Variablen re und im sonst beliebige Werte enthalten können.

Eigene Implementierung der Konstruktoren mit Initialisierung der Daten:

```
Complex::Complex()
   : m_re(0.0), m_im(0.0)
{}

Complex::Complex(double re, double im)
   : m_re(re), m_im(im)
{}
```

Die Konstruktoren werden folgendermaßen benutzt:

```
int main()
{
   Complex c1;       // Standardkonstruktor
   Complex c2();     // Standardkonstruktor
   Complex c3(4,5);  // Konstruktor mit Initialisierungswerten
   return 0;
}
```

rtr Startseite
Einführung
Inhalt
1. Vorbemerkungen
I. Linux
II. C
III. C++
17. Das Klassenkonzept
18. Basistechniken
19. Templates
20. Die Standardbibliothek STL
> 21. Klassen II
21.1 Alternative Konstruktoren
> 21.2 Copy-Konstruktor
21.3 Zuweisungsoperator
21.4 Automatisch generierte Funktionen
22. Fortgeschrittene Techniken
23. Vererbung
24. Polymorphie
25. Aufgaben

21.2 Copy-Konstruktor

Wenn der Konstruktor ein Objekt vom gleichen Typ übergeben bekommt, nennt man ihn Copy-Konstruktor, da er eine Kopie des gleichen Datentyps anlegt.

```
Complex(const Complex& c);
```

Wenn in der Klasse kein Copy-Konstruktor enthalten ist, erzeugt der Compiler eine Default-Implementierung.
Implementierung:

```
Complex::Complex(const Complex& c)
   : m_re(c.m_re), m_im(c.m_im)
{}
```

Die Konstruktoren werden folgendermaßen benutzt:

```
int main()
{
   Complex c1;        // Standardkonstruktor
   Complex c4(c1);    // Kopie von c1
   return 0;
}
```

rtr Startseite
Einführung
Inhalt
1. Vorbemerkungen
I. Linux
II. C
III. C++
17. Das Klassenkonzept
18. Basistechniken
19. Templates
20. Die Standardbibliothek STL
> 21. Klassen II
21.1 Alternative Konstruktoren
21.2 Copy-Konstruktor
> 21.3 Zuweisungsoperator
21.4 Automatisch generierte Funktionen
22. Fortgeschrittene Techniken
23. Vererbung
24. Polymorphie
25. Aufgaben

21.3 Zuweisungsoperator

Bei der Zuweisung einer Variablen auf eine andere ist manchmal eine besondere Handhabung von Daten notwendig:

```
c2 = c1;
```

Daher gibt es einen speziellen Zuweisungsoperator, der wie die Konstruktoren und Desktruktoren vom Compiler erzeugt wird, wenn keiner implementiert wurde:

```
Complex& operator=(const Complex& rhs);
```

Die Funktion bekommt den rechten Teil der Zuweisung als Parameter und es wird die Zuweisungsoperation des Objekts auf der linken Seite aufgerufen, rhs steht für ,,right hand side``. Das Ergebnis wird als Rückgabewert übergeben, damit eventuelle Kettenzuweisungen durchgeführt werden können.

```
c3 = c2 = c1;
```

Die Implementierung eines Zuweisungsoperators sollte folgende Schritte enthalten:

1. testen, ob rechte und linke Seite der Zuweisung identisch sind
2. alte Daten löschen (betrifft vor allem Pointer)
3. alle Daten der rechten Seite den lokalen Daten zuweisen

```
Complex& Complex::operator= (const Complex& rhs)
{
   if (this==&rhs)
     return *this;

   /* loeschen der Daten hier nicht notwendig */
   m_re = c.m_re;
   m_im = c.m_im;

   return *this;
}
```

rtr Startseite
Einführung
Inhalt
1. Vorbemerkungen
I. Linux
II. C
III. C++
17. Das Klassenkonzept
18. Basistechniken
19. Templates
20. Die Standardbibliothek STL
> 21. Klassen II
21.1 Alternative Konstruktoren
21.2 Copy-Konstruktor
21.3 Zuweisungsoperator
> 21.4 Automatisch generierte Funktionen
22. Fortgeschrittene Techniken
23. Vererbung
24. Polymorphie
25. Aufgaben

21.4 Automatisch generierte Funktionen

Für eine leere Klasse generiert der Compiler automatisch eine Reihe von Methoden, wenn diese im Code benötigt werden. Einige haben wir bereits kennengelernt, hier ist eine komplette Liste:

```
class MyClass{};
```

entspricht

```
class MyClass
{
public:
    MyClass(){};                             // Default-Konstruktor
    MyClass(const MyClass& rhs) ;            // Copy-Konstruktor
    ~MyClass();                              // Destruktor

    MyClass& operator=(const MyClass& rhs);  // Zuweisungsoperator
    MyClass* operator*();                    // Dereferenzierungsoperator
    const MyClass* operator&();              // Adressoperator
};
```

rtr Startseite
Einführung
Inhalt
1. Vorbemerkungen
I. Linux
II. C
III. C++
17. Das Klassenkonzept
18. Basistechniken
19. Templates
20. Die Standardbibliothek STL
21. Klassen II
22. Fortgeschrittene Techniken
22.1 Konstanten
22.2 Konstante Methoden
22.3 Referenzen
22.4 Deklarationen
22.5 Operatoren
22.6 Statische Member-Variablen und -Methoden
22.7 Beispiel
23. Vererbung
24. Polymorphie
25. Aufgaben

22.1 Konstanten

In C werden Konstanten mit der Präprozessoranweisung **#define** vereinbart:

```
#define PI 3.14159
```

Nachteile dieser Vorgehensweise sind:

- keine Typprüfung
- keine Anzeige im Debugger

Konstanten werden in C++ mit dem Schlüsselwort `const` gekennzeichnet:

```
const double PI = 3.14159
```

Die Konstantheit ist Bestandteil des Datentyps und kann nur, wenn es denn unbedingt notwendig ist, durch spezielle Casts (Typkonvertierungen) entfernt werden.

Bemerkung:

Die Verwendung von globalen Konstanten sollte wenn immer möglich vermieden werden und eine Deklarierung mit lokaler Gültigkeit vorgezogen werden.

rtr Startseite
Einführung
Inhalt
1. Vorbemerkungen
I. Linux
II. C
III. C++
17. Das Klassenkonzept
18. Basistechniken
19. Templates
20. Die Standardbibliothek STL
21. Klassen II
22. Fortgeschrittene Techniken
22.1 Konstanten
22.2 Konstante Methoden
22.3 Referenzen
22.4 Deklarationen
22.5 Operatoren
22.6 Statische Member-Variablen und -Methoden
22.7 Beispiel
23. Vererbung
24. Polymorphie
25. Aufgaben

22.2 Konstante Methoden

In C++ können auch Methoden einer Klasse das const-Schlüsselwert haben. Dies bedeutet, daß diese Methode das Objekt, über das sie aufgerufen wird, nicht verändert.

```
class MyClass
{
int value;
// die Methode getValue verändert keine Daten in MyClass
void getValue() const
{ return value; }
}
```

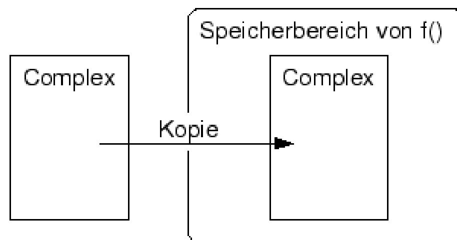
`const` sollte verwendet werden, wo immer es möglich ist.

rtr Startseite
Einführung
Inhalt
1. Vorbemerkungen
I. Linux
II. C
III. C++
17. Das Klassenkonzept
18. Basistechniken
19. Templates
20. Die Standardbibliothek STL
21. Klassen II
22. Fortgeschrittene Techniken
22.1 Konstanten
22.2 Konstante Methoden
22.3 Referenzen
22.4 Deklarationen
22.5 Operatoren
22.6 Statische Member-Variablen und -Methoden
22.7 Beispiel
23. Vererbung
24. Polymorphie
25. Aufgaben

22.3 Referenzen

Wenn einer Funktion auf gewöhnliche Art und Weise Parameter übergeben werden, werden diese in lokale Variablen kopiert. Das bedeutet auch, daß sich Änderungen einer Variablen nur innerhalb der Funktion auswirken. (*call by value*)

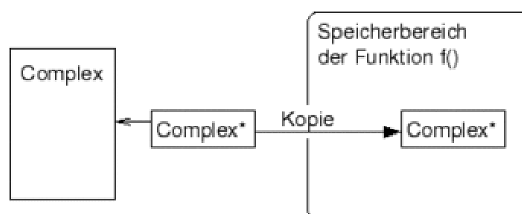
```
void
myFunctionValue(int i)
{
// nur die lokale Kopie
// wird verändert
i++;
}
```



Call by Value

Um größere Datenstrukturen nicht unnötig zu kopieren und dadurch die Laufzeit des Programms sowie den Speicherbedarf zu erhöhen, übergibt man die Adresse einer Variable, statt dem Objekt selbst. Zudem können auf diese Weise auch externe Variablen verändert werden. Die Variable wird der Funktion sozusagen ``zur Bearbeitung'' übergeben. (*call by reference*)

```
void myFunctionPointer(int* i)
{
(*i)++;
}
```

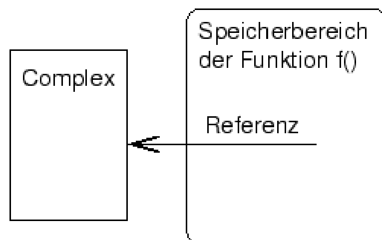


Call by Reference, C-Stil

```
int myI=9;
int* pmyI;          // * gehoert zur Typdeklaration,
                    //         Typ: Pointer auf int
pmyI = &myI;        // & ist der Adress-Operator.
*pmyI = 5;          // * ist hier der Dereferenzier-Operator,
                    //         myI hat jetzt den Wert 5

myFunctionValue(myI);      // myI hat immer noch den Wert 5
myFunctionPointer(pmyI);   // myI hat nun den Wert 6
```

In C++ gibt es eine zusätzliche Form der Parameterübergabe: die Referenz. Referenzen sind eine Art konstanter Zeiger, bei denen der Programmierer nicht mit Adressen arbeiten muß. Referenzen sind ein bestimmter Datentyp und werden mit dem Symbol & nach dem Variablentyp gekennzeichnet.

```
void myFunctionReference(int& i)
{
i++;
}
```



Call by Reference, C++-Stil

```
int myI=9;
int& rmyI = myI;     // & gehoert zur Typdeklaration,
                     //        Typ: Referenz auf int
                     // die referenzierte Variable kann nur
                     //        bei der Initialisierung festgelegt werden
                     // die Referenz kann spaeter nicht geaendert werden.
                     // myI und rmyI sind jetzt quasi identisch.
rmyI = 5;            // myI hat jetzt den Wert 5

myFunctionValue(myI);        // myI hat immer noch den Wert 5
myFunctionPointer(myI);      // myI hat nun den Wert 6
myFunctionReference(rmyI);  // myI hat nun den Wert 7
```

Vorteil ist, daß jetzt bei einem Funktionsaufrauf noch nicht mal mehr die Adresse kopiert wird, was wiederum geringe Laufzeitvorteile bringt. Zudem sind Referenzen vom Programmierer wesentlich einfacher zu handhaben. Es treten seltener Fehler auf als bei Zeigern.

Referenzen in Funktionsaufrufen

Wird der Parameter nur zur Benutzung innerhalb der Funktion übergeben, aber dort nicht verändert werden, übergibt man den Parameter als Konstanten. Der const-Operator ist dann als Zusicherung zu verstehen, daß der Wert der Variable nicht verändert wird. Auf diese Weise enthält die Schnittstelle der Funktion bereits eine Information, was in der Methode passiert:

```
// in myFunction wird d nicht verändert.
void myFunction(const double& d) {...}
```

Bemerkungen:

- besonders bei größeren Klassen sollten Parameter nie über call by value übergeben werden, da die kopierten Datenmengen sehr groß werden können
- Bei Funktionsdeklarationen sind Referenzen gegenüber Pointern zu bevorzugen
- Als reine Variablentypen werden Referenzen fast nie verwendet.
- Wenn eine Variable als Referenz übergeben wird, aber innerhalb einer Funktion nicht verändert werden soll, ist eine Deklaration als const-Parameter sinnvoll

rtr Startseite
Einführung
Inhalt
1. Vorbemerkungen
I. Linux
II. C
III. C++
17. Das Klassenkonzept
18. Basistechniken
19. Templates
20. Die Standardbibliothek STL
21. Klassen II
22. Fortgeschrittene Techniken
22.1 Konstanten
22.2 Konstante Methoden
22.3 Referenzen
22.4 Deklarationen
22.5 Operatoren
22.6 Statische Member-Variablen und -Methoden
22.7 Beispiel
23. Vererbung
24. Polymorphie
25. Aufgaben

22.4 Deklarationen

Deklarationen müssen nicht am Anfang eines Blocks stehen. Sie können an belieben Stellen im Quelltext gemacht werden.
Weiterhin gilt, daß sie ab der Deklaration und bis zum Blockende des aktuellen Blocks (z.B. for-Schleife) gilt.
Bei einer Deklaration in dem Kopf einer for-Schleife ist die Variable nur innerhalb der for-Schleife bekannt. Da die Handhabung bei den existierenden Compilern nicht einheitlich ist, sollte statt dessen die Laufvariable vorher angelegt werden:

```
int i;
for (i=0;i<n;i++)
{
  array[i]++;
}
```

TECHNISCHE UNIVERSITÄT DARMSTADT

rtr Startseite
Einführung
Inhalt
1. Vorbemerkungen
I. Linux
II. C
III. C++
17. Das Klassenkonzept
18. Basistechniken
19. Templates
20. Die Standardbibliothek STL
21. Klassen II
22. Fortgeschrittene Techniken
22.1 Konstanten
22.2 Konstante Methoden
22.3 Referenzen
22.4 Deklarationen
22.5 Operatoren
22.6 Statische Member-Variablen und -Methoden
22.7 Beispiel
23. Vererbung
24. Polymorphie
25. Aufgaben

22.5 Operatoren

In C++ ist es möglich, für eigene Klassen Operatoren zu definieren, wie sie für builtin-Datentypen wie int, float, etc. zur Verfügung stehen. Diese Überladen die Basis-Operatoren. Sinn dieser Operatoren ist es folgende Schreibweise für eigene Datentypen zu erlauben:

```
ergebnis = x * y + z;
x + 1.0;
```

Verschiedene operator-Implementierungen:

Operator 1: zweistelliger Operator für 2 komplexe Zahlen

```
Complex& operator+(const Complex& lhs, const Complex& rhs)
{
Complex result(lhs.re() + rhs.re(), lhs.im() + rhs.im());
return result;
}
```

Operator 2: zweistelliger Operator für komplexe Zahl plus double

```
Complex& operator+(const Complex& lhs, const double rhs)
{
Complex result(lhs.re() + rhs, lhs.im());
return result;
}
```

Operator 3: einstelliger Operator

```
const Complex operator+(const Complex& rhs)
{
tis->m_re + rhs.re();
this->m_im + rhs.im();
return *this;
}
```

Ein solcher Operator kann dann folgendermaßen verwendet werden:

```
Complex compl1(2,3);
Complex compl2(4,5);
double wert = 5;

compl1 = compl2 + wert;
compl1 = compl1 + compl2;
compl2 + compl1;
```

Überladbare Operatorfunktionen

```
+    -    *    /    %    ^    &    |    ~    >>   <<   ++   --
+=   -=   *=   /=   %=   ^=   &=   |=   ~=   >>=  <<=
=    <    >    !    !=   ==   <=   >=   &&   ||
->   ->*  ,    []   ()   new  new[]  delete  delete[]
```

Implizite Typumwandlung

In C++ sind implizite Typumwandlungen möglich, wenn ein entsprechender Konstruktor existiert. Der Konstruktor für die Typumwandlung von double nach Complex würde beispielsweise so aussehen:

```
Complex( const double d )
    : m_re(d), m_im(0.0)
{}
```

Wenn ich also nun eine Funktion (wie beispielsweise den operator+) mit einem double-Wert als Argument statt einem Complex-Objekt aufrufe, wird dieser erst implizit mit Hilfe des Konstruktors in ein Complex-Objekt konvertiert und dann damit die Funktion aufgerufen.
Mit dem oben angegebenen Konstruktor kann man sich also z.B. für die Addition der komplexen Zahlen folgende Operatoren sparen:

```
Complex& operator+(const Complex& lhs, const double rhs)
Complex& operator+(const double lhs, const Complex& rhs)
Complex& operator+(const double lhs, const double rhs)
```

Möchte man, daß eine solche Typkonvertierung nicht implizit ausgeführt wird, kann man einen Konstruktur als `explicit` deklarieren:

```
explicit Complex( const double d )
    : m_re(d), m_im(0.0)
{}
```

member-Funktion oder nicht?

- Nur globale Funktionen erlauben implizite Typumwandlungen. Möchte man diese nutzen, muß man den Operator als globale Funktion deklarieren.

- Ein Operator kann nur dann eine member-Funktion einer Klasse sein, wenn der Ergebnistyp auf der linken Seite den Typ der Klasse hat. Deshalb können `operator<<` und `operator>>` keine member-Funktionen sein (siehe unten).
- Alle anderen Operatoren, auf die obiges nicht zutrifft, sollten Elementfunktionen sein.
- Soll ein Operator in einer erbenden Klassen überladen werden (siehe auch 24.2), muß der Operator eine Elementfunkion sein.

friend-Funktionen

Das Schlüsselwort **friend** bewirkt, daß die entsprechende Funktion auf die privaten Elemente der Klasse, in der die Methode als friend deklariert ist, zugreifen kann.
Ein solches Beispiel sind die Ein- und Ausgabefunktionen für Streams operator<< und operator>>, die häufig Zugriff auf private Daten benötigen:

Header

```
//Deklaration der globalen Funktion
ostream& operator<<(ostream& output, const Complex& c1);

class Complex
{    ...
   public:
      friend ostream& operator<<(ostream& output, const Complex& c1);
}
```

Sourcefile

```
ostream& operator<<(ostream& output, const Complex& c1)
{
   output << m_re << " + j(" << m_im << ")";
}
```

Einige Beispiele für Operatordefinitionen

Rechenoperationen

```
   Complex operator-(Complex& c1, double d);
   Complex& operator+=(const Complex& c1);
   Complex& operator-=(const Complex& c1);
   Complex& operator++(const Complex& c1);
   Complex& operator--(const Complex& c1);
```

Vergleiche

```
   bool operator==(const Complex& c1, const Complex&  c2);
   bool operator!=(const Complex& c1, const Complex&  c2);
   bool operator>(const Complex& c1, const Complex&  c2);
   bool operator<(const Complex& c1, const Complex&  c2);
```

Zuweisung

```
   Complex& operator=(const Complex& c1);
```

Ein-/Ausgabe

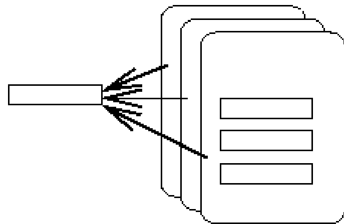
```
   istream& operator>>(istream& input, Complex& c1);
   ostream& operator<<(ostream& output, const Complex& c1);
```

rtr Startseite
Einführung
Inhalt
1. Vorbemerkungen
I. Linux
II. C
III. C++
17. Das Klassenkonzept
18. Basistechniken
19. Templates
20. Die Standardbibliothek STL
21. Klassen II
22. Fortgeschrittene Techniken
22.1 Konstanten
22.2 Konstante Methoden
22.3 Referenzen
22.4 Deklarationen
22.5 Operatoren
22.6 Statische Member-Variablen und -Methoden
22.7 Beispiel
23. Vererbung
24. Polymorphie
25. Aufgaben

22.6 Statische Member-Variablen und -Methoden

Manchmal möchte man Eigenschaften einer Klasse haben, die für alle Instanzen gelten. Oder man möchte Methoden benutzen können, die zwar zur Funktionalität einer Klasse gehören, aber nicht unbedingt eine konkrete Instanz benötigen.
Dazu gibt es das Konzept der statischen Member-Variablen und -Methoden. In diesem Beispiel hat die Klasse A die statische Variable m_count, um die Anzahl der existierenden Objekte zu zählen.
Mit der statischen Funktion count() kann der Zählerstand abgefragt werden:

```
class A
{
static int m_count;
public:
static int count();
A();
~A();
...
}
```



Die Objekte einer Klasse haben Zugriff auf eine gemeinsame statische Variable.

Eine statische Variable muß außerhalb der Klassendeklaration **definiert** werden, dies wird im cpp-File gemacht.
Eine gleichzeitige Initialisierung ist möglich:

```
int A::m_count =0;
```

Die statische Funktion wird ebenfalls hier definiert:

```
int A::count() { return m_count; }
```

Da eine statische Funktion/Variable nicht an ein konkretes Objekt gekoppelt ist, wird sie über die Klassenzugehörigkeit angesprochen:

```
int number_of_class_A_objects = A::count();
```

Innerhalb der Klasse kann sie direkt verwendet werden:

```
A::A()  {  m_count++; }
A::~A() {  m_count--;  }
```

rtr Startseite
Einführung
Inhalt
1. Vorbemerkungen
I. Linux
II. C
III. C++
17. Das Klassenkonzept
18. Basistechniken
19. Templates
20. Die Standardbibliothek STL
21. Klassen II
22. Fortgeschrittene Techniken
22.1 Konstanten
22.2 Konstante Methoden
22.3 Referenzen
22.4 Deklarationen
22.5 Operatoren
22.6 Statische Member-Variablen und -Methoden
22.7 Beispiel
23. Vererbung
24. Polymorphie
25. Aufgaben

22.7.1 example.h

```
#ifndef COMPLEX_H
#define COMPLEX_H

/** class to handle complex numbers  */
class Complex
{
private:
    double m_re;
    double m_im;

public:
    /** Constructors and destructor */
    Complex() : m_re(0.0), m_im(0.0) {}
    Complex( double re, double im ) : m_re(re), m_im(im) {}
    Complex( double re ) : m_re(re), m_im(0.0) {}
    Complex( const Complex& c ) : m_re(c.m_re), m_im(c.m_im) {}
    ~Complex() {}

    /* Assignment operator */
    Complex& operator= (const Complex& rhs);
    /* overloaded assignement operator */
    Complex& operator= (const double rhs);

    /* methods */
    void set(double re, double im);

    void setRe(const double re) { m_re = re; }
    double re() const           { return m_re; }
    void setIm(const double im) { m_im = im; }
    double im() const           { return m_im; }

    Complex& operator+=(const Complex& rhs);
    Complex& operator-=(const Complex& rhs);

};  // end of class Complex


ostream& operator>> (ostream& out, const Complex& c);
istream& operator<< (istream& out, const Complex& c);

Complex& operator+(const Complex& lhs, const Complex& rhs);
Complex& operator-(const Complex& lhs, const Complex& rhs);

/* very unusual that we have complete access to the data via
 * the interface, so we make these functions global to allow
 * type conversions on the left hand side
 */
bool operator==(const Complex& lhs, const Complex& rhs);
bool operator!=(const Complex& lhs, const Complex& rhs);

#endif
```

23.7.2 example.cpp

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

/*=========================================*/
/* methods of class Complex*/

Complex& Complex::operator= (const Complex& rhs)
{
    if (&rhs == this)
        return *this;
    m_re = rhs.m_re;
    m_im = rhs.m_im;
    return *this;
}

Complex& Complex::operator= (const double rhs)
{
    m_re = rhs;
    m_im = 0.0;
    return *this;
```

```
}

void set(double re, double im)
{
    m_re=re;
    m_im=im;
}

Complex& Complex::operator+=(const Complex& rhs)
{
    m_re += rhs.m_re;
    m_im += rhs.m_im;
    return *this;
}

Complex& Complex::operator-=(const Complex& rhs)
{
    m_re -= rhs.m_re;
    m_im -= rhs.m_im;
    return *this;
}


/*=========================================*/
/* globale Funktionen */

ostream& operator<< (ostream& out, const Complex& c)
{
    out << c.re() << " + j( " << c.im() << ")";
    return out;
}

istream& operator>> (istream& in, const Complex& c)
{
    double re, im;
    in >> re;
    in >> im;
    c.set(re,im);
    return in;
}

Complex& operator+(const Complex& lhs, const Complex& rhs)
{
    Complex c(lhs);
    c += rhs;
    return c;
}

Complex& operator-(const Complex& lhs, const Complex& rhs)
{
    Complex c(lhs);
    c -= rhs;
    return c;
}

bool operator==(const Complex& lhs, const Complex& rhs)
{
    return (lhs.re()==rhs.re() && lhs.im()==rhs.im());
}

bool operator!=(const Complex& lhs, const Complex& rhs);
{
    return !(lhs==rhs);
}
```

23.7.3 main.cpp

```
#include "example.h"

int main()
{
    Complex* p_c1;    // creation of instance c1 with default values
    p_c1 = new Complex(10,30);

    Complex c2(5,6);

    cout << *p_c1 << endl;
    cout << c2 << endl;

    Complex c3;
    if (c3 == c2 )
        cout << "c2 und c3 sind gleich" << endl;

    c2 = c3 + *p_c1;
    c2 += c3;
    c2 += 5;

    delete p_c1;

    return 0;
}
```

rtr Startseite
Einführung
Inhalt
1. Vorbemerkungen
I. Linux
II. C
III. C++
17. Das Klassenkonzept
18. Basistechniken
19. Templates
20. Die Standardbibliothek STL
21. Klassen II
22. Fortgeschrittene Techniken
23. Vererbung
23.1 Objektorientierte Konzepte zur Code-Wiederverwendung
23.2 Vererbung
23.3 Ergänzen von Funktionen
23.4 Konstruktion
23.5 Destruktion
23.6 Zuweisung
23.7 Beispiel für Vererbung
24. Polymorphie
25. Aufgaben

23.1 Objektorientierte Konzepte zur Code-Wiederverwendung

Das Konzept der Klasse ermöglicht in der OO - Programmierung

- **das Verstecken von Information**
  Implementierungsdetails bleiben dem Benutzer der Klasse verborgen
- **die Kapselung**
  Daten und Methoden können versteckt werden, Klassen haben eine klare Schnittstelle

Zum Wiederverwenden von Implementierungen bietet C++ folgende weitergehende Konzepte:

- **Vererbung**
  zum Weitergeben von Klassenschnittstellen und -implementierungen and darauf aufbauende Klassen
- **Abstraktion**
  zur Spezifikation von allgemeinen Klassenschnittstellen
- **Polymorphie**
  wenn mehrere Klassen die gleiche Aufgabe auf verschiedene Weisen erledigen und die Klassen austauschbar sein sollen
- **Templates**
  zum flexiblen Einsatz von Algorithmen auf parametrisierbaren internen Datentypen

23.1.1 Beispiel

Wir möchten Buch-, Film- und Musik-Information verwalten. Dazu legen wir verschiedene Klassen an, die sich haupsächlich in den Daten und in der Ausgabemethode unterscheiden:

```
class Book;
class Film;
```

Die Objekte mit den konkreten Daten möchten wir in einer Liste verwalten. Es soll möglich sein, auf allen Datensätzen Suchoperationen auszuführen. Außerdem soll es möglich sein, alle Informationen in verschiedenen Formaten auszugeben.
Als Datenstruktur für die Liste der verschiedenen Medientypen verwenden wir ein Array. Dieser hat jedoch einen festgelegten Datentyp, wie beispielsweise den Typ **Media**.

```
Media*[5] medialist;
```

Problem:

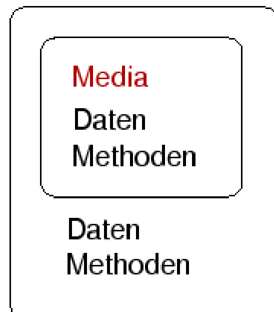
- Verwaltung der Objekte in einer gemeinsamen Liste
- Einheitliche Abfragen auf Objekten ausführen, die unterschiedliche Typen haben
- einfacher Austausch von Ausgabestrategien

rtr Startseite
Einführung
Inhalt
1. Vorbemerkungen
I. Linux
II. C
III. C++
17. Das Klassenkonzept
18. Basistechniken
19. Templates
20. Die Standardbibliothek STL
21. Klassen II
22. Fortgeschrittene Techniken
23. Vererbung
23.1 Objektorientierte Konzepte zur Code-Wiederverwendung
23.2 Vererbung
23.3 Ergänzen von Funktionen
23.4 Konstruktion
23.5 Destruktion
23.6 Zuweisung
23.7 Beispiel für Vererbung
24. Polymorphie
25. Aufgaben

23.2 Vererbung

Bei der **Vererbung** stehen den erbenden Klassen Schnittstellen und Implementierungen der Elternklasse zur Verfügung. Die erbende Klasse setzt sich dann zusammen aus der Elternklasse plus der neuen Daten und Methoden in der Kindklasse.
Syntax der Vererbung:

```
class Media
{
    private:
        int m_i;      // steht in Book
                      // nicht direkt zur Verfuegung
    public:
        void i();     // auch extern verfügbar
    protected:
        void setI(); // kann nur in erbenden
                      // Klassen verwendet werden.
};

class Book : public Media
{
...
};
```



Geerbt werden alle Daten und Funktionen, Zugriff besteht jedoch nur, wenn sie in der Elternklasse als `public` oder `protected` deklariert sind. Mehrfachvererbungen sind auch möglich:

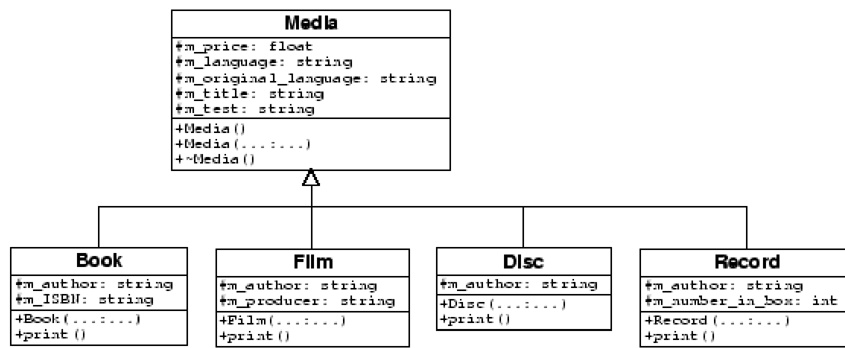
```
class Book : public Media, public Printable
{
...
};
```

Es können neue Daten und Methoden hinzugefügt werden. Methoden können überladen werden, wenn sie in der Elternklasse public oder protected sind.
Vorteil der Vererbung:

- mehrere Klassen können so die Implementierung bzw. die Schnittstelle der Elternklasse mitbenutzen
- einfach zu realisieren

Die Verwendung einer erbenden Klasse unterscheidet sich nicht von der einer einfachen Klasse.

23.2.1 Vererbungshierarchie



Hinweis:

Klassenhierarchien sollten möglichst flach gehalten werden und Elternklassen sollten möglichst allgemeine Implementationen enthalten.

rtr Startseite
Einführung
Inhalt
1. Vorbemerkungen
I. Linux
II. C
III. C++
17. Das Klassenkonzept
18. Basistechniken
19. Templates
20. Die Standardbibliothek STL
21. Klassen II
22. Fortgeschrittene Techniken
23. Vererbung
23.1 Objektorientierte Konzepte zur Code-Wiederverwendung
23.2 Vererbung
23.3 Ergänzen von Funktionen
23.4 Konstruktion
23.5 Destruktion
23.6 Zuweisung
23.7 Beispiel für Vererbung
24. Polymorphie
25. Aufgaben

23.3 Ergänzen von Funktionen

Möchte man eine Funktion überladen, so deklariert man die Methode auch in der erbenden Klasse mit gleichem Namen und gleicher Parameterliste. Die Implementierung der Kind-Klasse überschreibt dann die der Eltern-Klasse. Möchte man allerdings nicht die komplette Funktion überschreiben, kann man in der Kind-Klasse auch die Methoden der Elternklasse direkt aufrufen. Mit dem **::-Operator** macht man deutlich aus welcher Klasse die Methode stammt:

```
void Book::print(ostream& out)
{
    Media::print(out);
    out << m_author;
    out << m_ISBN;
}
```


Ererben von Methoden

TECHNISCHE UNIVERSITÄT DARMSTADT

Fachgebiet Regelungstheorie und Robotik

Impressum | Druckversion

REGELUNGSTHEORIE UND ROBOTIK rtr

Sie befinden sich: > TU Darmstadt > ETIT > IAT > RTR > Lehre > E-Learning > C/C++-Onlinekurs > III. C++ > 23. Vererbung > 23.4 Konstruktion

rtr Startseite
Einführung
Inhalt
1. Vorbemerkungen
I. Linux
II. C
III. C++
17. Das Klassenkonzept
18. Basistechniken
19. Templates
20. Die Standardbibliothek STL
21. Klassen II
22. Fortgeschrittene Techniken
23. Vererbung
23.1 Objektorientierte Konzepte zur Code-Wiederverwendung
23.2 Vererbung
23.3 Ergänzen von Funktionen
23.4 Konstruktion
23.5 Destruktion
23.6 Zuweisung
23.7 Beispiel für Vererbung
24. Polymorphie
25. Aufgaben

23.4 Konstruktion

Der Ablauf bei der Konstruktion einer erbenden Klasse passiert folgendermaßen:

1. Anlegen der Klasseninformation im Speicher
2. Aufruf des Konstruktors der ererbten Klasse
    1. Anlegen der Klasseninformation im Speicher
    2. Initialisierung der Variablen der Elternklasse
    3. Ausführung der Implementierung des Eltern-Konstruktors
3. Initialisierung der eigenen Variablen
4. Ausführung der Implementierung des Konstruktors

Ererben von Daten

Wenn man Parameter an den Konstruktor der Eltern-Klasse übergeben möchte, kann man diesen explizit vor der Initialisierung der eigenen Variablen aufrufen:

```
Book::Book()
    : Media()
{}

Book::Book(const string& author, const string& ISBN, float price,
           const string& language, const string& title)
    : Media(price, language, title),
      m_author(author), m_ISBN(ISBN)
{}

Book::Book(const Book& rhs)
    :  Media(rhs),
       m_author(rhs.m_author), m_ISBN(rhs.m_ISBN)
{}
```

TECHNISCHE UNIVERSITÄT DARMSTADT

Fachgebiet Regelungstheorie und Robotik

Impressum | Druckversion

REGELUNGSTHEORIE UND ROBOTIK rtr

rtr Startseite
Einführung
Inhalt
1. Vorbemerkungen
I. Linux
II. C
III. C++
17. Das Klassenkonzept
18. Basistechniken
19. Templates
20. Die Standardbibliothek STL
21. Klassen II
22. Fortgeschrittene Techniken
23. Vererbung
23.1 Objektorientierte Konzepte zur Code-Wiederverwendung
23.2 Vererbung
23.3 Ergänzen von Funktionen
23.4 Konstruktion
23.5 Destruktion
23.6 Zuweisung
23.7 Beispiel für Vererbung
24. Polymorphie
25. Aufgaben

23.5 Destruktion

Auch bei der Desktruktion muß die Elternklasse berücksichtigt werden:
Der Ablauf der Zerstörung eines Objekts ist umgekehrt zur Konstruktion:

1. Ausführung der Implementierung des Destruktors
2. Zerstörung der eigenen Variablen
3. Aufruf des Destruktors der Elternklasse
4. Freigeben der Klasseninformation im Speicher

rtr Startseite
Einführung
Inhalt
1. Vorbemerkungen
I. Linux
II. C
III. C++
17. Das Klassenkonzept
18. Basistechniken
19. Templates
20. Die Standardbibliothek STL
21. Klassen II
22. Fortgeschrittene Techniken
23. Vererbung
23.1 Objektorientierte Konzepte zur Code-Wiederverwendung
23.2 Vererbung
23.3 Ergänzen von Funktionen
23.4 Konstruktion
23.5 Destruktion
23.6 Zuweisung
23.7 Beispiel für Vererbung
24. Polymorphie
25. Aufgaben

23.6 Zuweisung

Die Implementierung eines Zuweisungsoperators muß nun auch die Zuweisung des Elternteils beachten:

1. testen, ob rechte und linke Seite der Zuweisung identisch sind
2. Aufruf des Zuweisungsoperators der ererbten Klasse
3. alte Daten löschen (betrifft vor allem Pointer)
4. alle Daten der rechten Seite den lokalen Daten zuweisen

```
Book& Book::operator= (const Book& rhs)
{
   if (this==&rhs)
     return *this;

   Media::operator=(rhs);

   /* loeschen der Daten hier nicht notwendig */
   m_author = rhs.m_author;
   m_ISBN = rhs.m_ISBN;

   return *this;
}
```

TECHNISCHE UNIVERSITÄT DARMSTADT

Fachgebiet Regelungstheorie und Robotik

Impressum | Druckversion

REGELUNGSTHEORIE UND ROBOTIK rtr

Sie befinden sich: > TU Darmstadt > ETIT > IAT > RTR > Lehre > E-Learning > C/C++-Onlinekurs > III. C++ > 23. Vererbung > 23.7 Beispiel für Vererbung

rtr Startseite
Einführung
Inhalt
1. Vorbemerkungen
I. Linux
II. C
III. C++
17. Das Klassenkonzept
18. Basistechniken
19. Templates
20. Die Standardbibliothek STL
21. Klassen II
22. Fortgeschrittene Techniken
23. Vererbung
23.1 Objektorientierte Konzepte zur Code-Wiederverwendung
23.2 Vererbung
23.3 Ergänzen von Funktionen
23.4 Konstruktion
23.5 Destruktion
23.6 Zuweisung
23.7 Beispiel für Vererbung
24. Polymorphie
25. Aufgaben

23.7.1 Beispiel 1

examle1.h

```
 1 #include <iostream>
 2 #include <string>
 3
 4 class Media
 5 {
 6 protected:
 7     float m_price;
 8     string m_language;
 9     string m_title;
10     string m_test;
11 public:
12     Media(float price, const string& language,const string& title);
13     ~Media(){};
14     void print();
15 };
16
17 class Book : public Media   // Book is subclass of Media
18 {
19 protected:
20     string m_author;
21     string m_ISBN;
22 public:
23     Book(const string& author, const string& ISBN, float price,
24          const string& language, const string& title);
25     void print();
26 };
27
28 class Film : public Media   // Film is subclass of Media
29 {
30 protected:
31     string m_author;
32     string m_producer;
33 public:
34     Film(const string& author, const string& producer,
35          float price, const string& language,const string& title);
36     void print();
37 };
```

24.7.2 Beispiel 1

examle1.cpp

```
 1 #include <iostream>
 2 #include <string>
 3
 4 #include "example1.h"
 5
 6 Media::Media(float price, const string& language,const string& title)
 7    : m_price(price), m_language(language),m_title(title), m_test("o.k.")
 8 {}
 9
10 void Media::print()
11 {
12     cout << m_test << endl;
13 }
14
15 Book::Book(const string& author, const string& ISBN, float price,
16            const string& language, const string& title)
17    : Media(price, language, title), m_author(author), m_ISBN(ISBN)
18 {}
19
20 void Book::print()
21 {
22     cout << "price:" << m_price << endl;
23     cout << "language:"<< m_language << endl;
24     cout << "title:" << m_title << endl;
25     cout << "author:" << m_author << endl;
```

```
26      cout << "ISBN:" << m_ISBN << endl;
27 }
28
29 Film::Film(const string& author, const string& producer, float price,
30            const string& language, const string& title)
31    : Media(price, language, title), m_author(author), m_producer(producer)
32 {}
33
34 void Film::print()
35 {
36      cout << "price:" << m_price << endl;
37      cout << "language:"<< m_language << endl;
38      cout << "title:" << m_title << endl;
39      cout << "author:" << m_author << endl;
40      cout << "producer:" << m_producer << endl;
41 }
42
43 int main(void)
44 {
45      cout << "Book:" << endl;
46      Book* mybook = new Book("Stroustrup, Bjarne","3-8273-1756-8",69.90,
47              "deutsch","Die C++ Programmiersprache");
48      mybook->print();
49
50      cout << endl << "Media:" << endl;
51      Media* mymedia;
52      mymedia = new Book("Stroustrup, Bjarne","3-8273-1756-8",69.90,
53                         "deutsch","Die C++ Programmiersprache");
54      mymedia->print();
55 }
```

TECHNISCHE UNIVERSITÄT DARMSTADT

Fachgebiet Regelungstheorie und Robotik

Impressum | Druckversion

REGELUNGSTHEORIE UND ROBOTIK rtr

Sie befinden sich: > TU Darmstadt > ETIT > IAT > RTR > Lehre > E-Learning > C/C++-Onlinekurs > III. C++ > 24. Polymorphie > 24.1 Dynamisches Binden

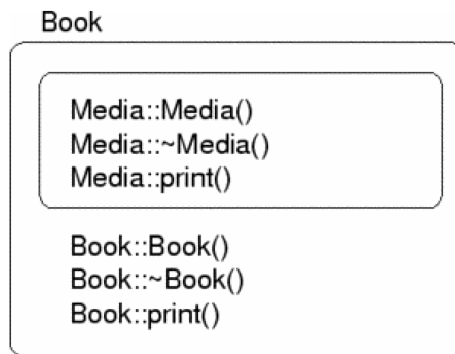
rtr Startseite
Einführung
Inhalt
1. Vorbemerkungen
I. Linux
II. C
III. C++
17. Das Klassenkonzept
18. Basistechniken
19. Templates
20. Die Standardbibliothek STL
21. Klassen II
22. Fortgeschrittene Techniken
23. Vererbung
24. Polymorphie
24.1 Dynamisches Binden
24.2 Abstrakte Klassen und "pure virtual"-Funktionen
25. Aufgaben

24.1 Dynamisches Binden

Polymorphie ist ein wichtiges Element objektorientierter Programmierung und macht Klassenhierarchien zu einem mächtigen Instrument.
Polymorphie ist die die ,,Vielgesichtigkeit`` eines Zeigers bzw. einer Referenz. D.h. ein Zeiger oder eine Referenz kann vom Typ her ein Pointer auf eine Elternklasse sein, aber trotzdem auf Objekte der Kindklassen zeigen.
Normalerweise wird in C/C++ die **frühe Bindung** durchgeführt, d.h. der Typ, der referenziert wird, wird anhand des Typs des Pointers, bzw. der Referenz zur Compile-Zeit bestimmt.

```
Media* mymedia;
mymedia = new Book ("Stroustrup, Bjarne",
"3-8273-1756-8",69.90,"deutsch",
"Die C++ Programmiersprache");
mymedia->print();
```

-> Welche print-Funktion wird ausgeführt?



Ererben von Methoden

Damit hier die print-Funktion der Klasse Book statt von Media benutzt wird, muß die Funktion print() als **virtuelle Funktion** deklariert sein.
Dadurch wird der Mechanismus der **späten Bindung** ausgewählt, das ist die Typenbestimmung zur Laufzeit. Das Programm prüft dann erst, von welchem Typ das referenzierte Objekt tatsächlich ist, und ruft dann dessen virtuelle Methode auf.

```
class Media
{
...
virtual void print();
};
```

Das Schlüsselwort virtual bewirkt also, daß zur Laufzeit die Funktion print() von dem Objekt aufgerufen wird, auf das der Pointer zeigt.
Für das Beispiel der Medien-Datenbank heißt das, das wir nun Bücher und Filme in einem Array abspeichern können und durch Aufruf der Methode print() für alle Elemente, alle Datensätze mit der jeweils richtigen Methode der Kindklasse ausgeben können.
Zusätzliche Funktionen der Kindklasse können allerdings über einen Pointer auf die Elternklasse nicht aufgerufen werden. Deshalb ist eine sorgfältige Schnittstellen-Deklaration der Elternklasse wichtig.

rtr Startseite
Einführung
Inhalt
1. Vorbemerkungen
I. Linux
II. C
III. C++
17. Das Klassenkonzept
18. Basistechniken
19. Templates
20. Die Standardbibliothek STL
21. Klassen II
22. Fortgeschrittene Techniken
23. Vererbung
24. Polymorphie
24.1 Dynamisches Binden
24.2 Abstrakte Klassen und "pure virtual"-Funktionen
25. Aufgaben

24.2 Abstrakte Klassen und "pure virtual"-Funktionen

Abstrakte Klassen
- dienen der **Definition von Schnittstellen** und
- haben keine Instanzen.

Sie können Datenobjekte und Implementierungen beinhalten, die alle Kind-Objekte einer abstrakten Klasse teilen. Der eigentliche Sinn liegt jedoch in der Definition von sogenannten **pure virtual functions**:

```
virtual void print()=0;
```

Diese werden in der abstrakten Klasse ohne Implementierung angelegt. Alle Kindklassen müssen diese Funktion überladen und somit mit einer Implementierung versehen.
Es gibt kein Schlüsselwort für eine abstrakte Klasse, aber sobald sie eine pure virtual-Funktion enthält, kann sie nicht instanziiert werden.

Abstrakte Klassen legen den Schwerpunkt auf **Schnittstellen-Vererbung**. D.h. sie deklarieren eine Schnittstelle, definieren sie aber nicht selber, sondern zwingen alle Kindklassen dazu, diese selber zu implementieren. Wenn alle Kindklassen das gleiche Interface haben, muß bei der Verwendung nicht festgelegt werden, welche Kindklasse verwendet wird. Alle lassen sich gleich handhaben.

Im Gegensatz dazu steht die **Implementierungsvererbung**. Hierbei werden in der Elternklasse Funktionalitäten implementiert, die dann an die Kinder vererbt werden. Dies erzeugt Implementierungsabhängigkeiten zwischen Subsystemen.

25.2.1 Beispiel

example3.h

```
 1 #include <string>
 2
 3 class Media
 4 {
 5 protected:
 6     float m_price;
 7     string m_language;
 8     string m_title;
 9     string m_test;
10 public:
11     Media(){};
12     Media(float price, const string& language,const string& title);
13     ~Media(){};
14     virtual void print()=0;
15 };
16
17 class Book : public Media  // Book is subclass of Media
18 {
19 protected:
20     string m_author;
21     string m_ISBN;
22 public:
23     Book(const string& author, const string& ISBN, float price,
24          const string& language,const string& title);
25     virtual void print();
26 };
27
28 class Film : public Media  // Film is subclass of Media
29 {
30 protected:
31     string m_author;
32     string m_producer;
33 public:
34     Film(const string& author, const string& producer,
35          float price, const string& language,const string& title);
36     virtual void print();
37 };
```

example3.cpp

```
 1 #include <iostream>
 2 #include <string>
 3
 4 #include "example3.h"
 5
 6 Media::Media(float price, const string& language,const string& title)
```

```
 7    : m_price(price), m_language(language),m_title(title), m_test("o.k.")
 8 {}
 9
10 Book::Book(const string& author, const string& ISBN, float price,
11            const string& language, const string& title)
12    : Media(price, language, title), m_author(author), m_ISBN(ISBN)
13 {}
14
15 void Book::print()
16 {
17     cout << "price:" << m_price << endl;
18     cout << "language:"<< m_language << endl;
19     cout << "title:" << m_title << endl;
20     cout << "author:" << m_author << endl;
21     cout << "ISBN:" << m_ISBN << endl;
22 }
23
24 Film::Film(const string& author, const string& producer, float price,
25            const string& language, const string& title)
26    : Media(price, language, title), m_author(author), m_producer(producer)
27 {}
28
29 void Film::print()
30 {
31     cout << "price:" << m_price << endl;
32     cout << "language:"<< m_language << endl;
33     cout << "title:" << m_title << endl;
34     cout << "author:" << m_author << endl;
35     cout << "producer:" << m_producer << endl;
36 }
37
38 int main(void)
39 {
40     cout << "Book:" << endl;
41     Book b1("Stroustrup, Bjarne","3-8273-1756-8",69.90,"deutsch",
42             "Die C++ Programmiersprache");
43     b1.print();
44
45     cout << endl << "Media:" << endl;
46     Media* mymedia;
47     mymedia = new Book("Stroustrup, Bjarne","3-8273-1756-8",69.90,
48                        "deutsch","Die C++ Programmiersprache");
49     mymedia->print();
50 }
```

rtr Startseite
Einführung
Inhalt
1. Vorbemerkungen
I. Linux
II. C
III. C++
17. Das Klassenkonzept
18. Basistechniken
19. Templates
20. Die Standardbibliothek STL
21. Klassen II
22. Fortgeschrittene Techniken
23. Vererbung
24. Polymorphie
25. Aufgaben
25.1 Aufgabe 1: STL
25.2 Aufgabe 2: Klassen
25.3 Aufgabe 3: Zur Vererbung von Klassen
25.4 Aufgabe 4: Zum Überladen von Methode
25.5 Loesungen

25.1 Aufgabe 1: STL

Anleitung:

1. Laden Sie die Files zur Aufgabe:
   .tar.gz oder .zip
2. Entpacken Sie die Aufgabe:

```
> gunzip stl.tar.gz
> tar xf stl.tar
```

3. In dem Projekt ist ein Makefile enthalten, mit dem das Projekt compiliert werden kann:

```
> make
```

Beschreibung

Die STL-Klassen sind hilfreich in typischen Programmieraufgaben, wie Datenverwaltung in Containern, Ein- und Ausgabe. In dieser Aufgabe werden wir uns diese Dinge genauer ansehen.
Dazu ist in dem Projekt bereits eine Datei mit Daten vorhanden, die in der Aufgabe eingelesen und bearbeitet werden sollen.

Aufgabe

- Fragen Sie den Benutzer nach dem Filenamen einer Datei mit Daten.
- Was passiert, wenn Sie einen nicht existenten Dateinamen angeben. Implementieren Sie eine sinnvolle Fehlerbehandlung, z.B. unter Verwendung der Methode `is_open` von `fstream`.
- Legen sie 2 Vektoren $\underline{x}$ und $\underline{t}$ an, um eine Zeitreihe $\underline{x}(t)$ zu speichern. Die Werte von x sind Dezimalzahlen und t ist ganzzahlig. Wählen sie die Datentypen für die Vektoren entsprechend.
- Öffnen Sie die Datei `"daten.txt"` und lesen sie die Werte ein. Speichern Sie die Werte in den Vektoren $\underline{t}$ und $\underline{x}$.
- Geben Sie die Zeitreihe auf dem Bildschirm aus.
- Bestimmen Sie das Maximum der Zeitreihe und geben Sie es aus.

rtr Startseite
Einführung
Inhalt
1. Vorbemerkungen
I. Linux
II. C
III. C++
17. Das Klassenkonzept
18. Basistechniken
19. Templates
20. Die Standardbibliothek STL
21. Klassen II
22. Fortgeschrittene Techniken
23. Vererbung
24. Polymorphie
25. Aufgaben
25.1 Aufgabe 1: STL
25.2 Aufgabe 2: Klassen
25.3 Aufgabe 3: Zur Vererbung von Klassen
25.4 Aufgabe 4: Zum Überladen von Methode
25.5 Loesungen

25.2 Aufgabe 2: Klassen

Anleitung:

1. Laden Sie die Files zur Aufgabe:
   .tar.gz oder .zip
2. Entpacken Sie die Aufgabe:

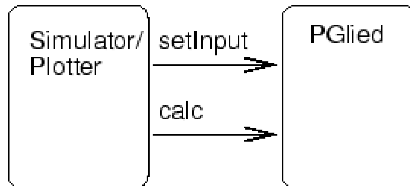
```
> gunzip klassen.tar.gz
> tar xf klassen.tar
```

3. In dem Projekt ist ein Makefile enthalten, mit dem das Projekt compiliert werden kann:

```
> make
```

Beschreibung

In der Aufgabe sind zunächst 2 Klassen implementiert: `SprungSimulator` und `PGlied`.
Die Klasse `SprungSimulator` hat die Aufgabe, einen Sprung zu simulieren. Dazu kann zum einen Parameter für einen Sprung konfiguriert werden. Und zum anderen kann ein Sprung mit der Methode `start()` ausgeführt werden. Die Klasse sorgt selber dafür, daß alle Werte ausgegeben werden. Das System, daß der `SprungSimulator` anregen soll, bekommt er als Referenz der Methode `start()` übergeben.
Die Klasse `PGlied` beinhaltet das Systemverhalten - in diesem Fall ein Proportional-Glied -, das wir gerne betrachten möchten. Ein P-Glied beinhaltet den Verstärkungsfaktor als Parameter, den wir über die Methoden `setK()` verändern können. Zum Simulieren gibt es die Methoden `setInput()` und `calc()`. `calc()` liefert uns den simulierten Wert zu dem Eingangswert, den wir mit `setInput()` gesetzt haben.



In `main()` ist eine Anwendung der beiden Klassen implementiert:
Zunächst werden Instanzen `pglied` und `sprung` von beiden Klassen angelegt und konfiguriert. Mit dem Aufruf der Methode `start()` von dem Objekt `sprung` mit pglied als Parameter wird die Simulation ausgeführt.

Aufgabe:

1. Die Klasse `SprungSimulator` ist noch nicht vollständig implementiert. Vervollständigen Sie die fehlenden Teile, so daß sie eine komfortable Klasse ist.
2. Testen Sie das System, beispielsweise durch Verwendung von gnuplot.
3. Statt einem P-Glied soll nun das Verhalten eines PT1-Gliedes simuliert werden. Dazu gibt es bereits eine Vorlage für eine Klasse `Pt1Glied`. In der Datei pt1glied.cpp gibt es eine Anleitung, wie man ein solches System digital simuliert. Vervollständigen Sie diese und testen Sie das System.
4. Implementieren Sie in der `main()`-Funktion ein interaktives Einlesen der Sprung- und System-Parameter.

Fragen:

1. Wie müßte man die Klassen `PGlied` bzw. `Pt1Glied` ändern, damit man das System in einem Schritt simulieren kann?
2. Wie läßt sich die Beziehung zwischen dem Simulator und dem `PGlied` bzw. `Pt1Glied` charakterisieren?
3. Wie könnte man die Klasse `SprungSimulator` ändern, damit man eine Versuchsreihe mit einem gegebenen System machen kann? Wie verändert sich dadurch die Bindung zwischen den Klassen `SprungSimulator` und `PGlied` bzw. `Pt1Glied`?
4. Welchen Nachteil hat die Klasse `SprungSimulator`, wenn man verschiedene Systeme in einem Programm untersuchen will?

rtr Startseite
Einführung
Inhalt
1. Vorbemerkungen
I. Linux
II. C
III. C++
17. Das Klassenkonzept
18. Basistechniken
19. Templates
20. Die Standardbibliothek STL
21. Klassen II
22. Fortgeschrittene Techniken
23. Vererbung
24. Polymorphie
25. Aufgaben
25.1 Aufgabe 1: STL
25.2 Aufgabe 2: Klassen
25.3 Aufgabe 3: Zur Vererbung von Klassen
25.4 Aufgabe 4: Zum Überladen von Methode
25.5 Loesungen

25.3 Aufgabe 3: Zur Vererbung von Klassen

Anleitung:

1. Laden Sie die Files zur Aufgabe:
   .tar.gz oder .zip
2. Entpacken Sie die Aufgabe:

```
> gunzip vererbung.tar.gz
> tar xf vererbung.tar
```

3. In dem Projekt ist ein Makefile enthalten, mit dem das Projekt compiliert werden kann:

```
> make
```

Beschreibung:

Im Rahmen dieser Aufgabe soll das Programm zur Simulation eines PT1-Gliedes aus der letzten Ausgabe erweitert bzw. an entscheidenden Stellen verbessert werden. Dabei sollen die neu eingeführten Verfahren der Vererbung und des dynamischen Bindens zur Anwendung kommen.
Zunächst soll von der gegebenen Oberklasse UeGlied (siehe `ueglied.h`), von der schon die Klasse `PGlied` abgeleitet wurde, die Klasse `Pt1Glied` abgeleitet werden. Dabei ist zu beachten, dass die Klassen jetzt etwas anders aufgebaut sind, als in der ersten Übung. Um sie möglichst allgemein zu gestalten, besitzt die Oberklasse für Parameter und Zustände Vektoren aus der STL), die dynamisch anhand den Erfordernissen der abgeleiteten Klasse erstellt werden. D.h. die abgeleiteten Klassen übergeben in ihrem Konstruktor die Anzahl der benötigten Parameter an die Oberklasse (vgl. `PGlied`).

Für das `Pt1Glied` werden jeweils zwei a und b Parameter benötigt. `b0 = 0.0` und `a0 = 1.0`, b1 und a1 berechnen sich wie in der letzten Übung aus K und T1. Die Deklaration der neuen Klasse kommt - wie es sich gehört -nach `ueglied.h`, die Methoden in `ueglied.cpp`.

Wenn dies geschehen ist, lohnt es sich einen Blick auf die Methode `SprungSimulation::start()` in `main.cpp` zu werfen! Man sieht, daß sie dieses Mal nicht mehr verändert werden muss, wenn ein anderes Übertragungsglied übergeben werden soll.

Ferner schreibt die Funktion die Werte nicht direkt auf den Bildschirm, sondern übergibt sie an ein Objekt der Klasse Senke (siehe `senke.h`). Als zweiter Teil dieser Übung soll nun als Alternative zu der gegeben - sehr einfachen - Klasse Senke eine Klasse Datei von Senke abgeleitet werden. In `senke.cpp` ist die Definition der Methoden der Klasse Senke zu finden.

Die neue Klasse soll wie folgt funktionieren:
bei der Erstellung der Instanz wird ein Dateiname übergeben
bei der Initialisierung der Instanz wird die Datei geöffnet. Dazu muss ein die Methode

```
open(const char* filename)
```

auf das Objekt `m_file` der Klasse ofstream angewendet werden. Die Funktion erwartetet als Parameter den Typ `const char*`. Diesen können wir aus einem `string` mit der Methode `c_str()` von `string` erzeugen.
Übergebene Werte werden direkt in die Datei geschrieben. Da eine Instanz von ofstream genauso ein Stream ist wie cout, können die Wert in gleicher Weise wie in Senke nach cout in Datei nach ofstream geschreiben werden. Das Gerüst einer Klassen Deklaration befindet sich in `senke.h`. Weiter unten in `senke.h` befindet sich die Deklaration einer Klasse Graphik zur graphischen Anzeige der Simulation auf dem Bildschirm. Diese gehört zwar nicht direkt zur Aufgabe, ist aber vollständig definiert und kann daher von Interessierten gerne ausprobiert werden.

rtr Startseite
Einführung
Inhalt
1. Vorbemerkungen
I. Linux
II. C
III. C++
17. Das Klassenkonzept
18. Basistechniken
19. Templates
20. Die Standardbibliothek STL
21. Klassen II
22. Fortgeschrittene Techniken
23. Vererbung
24. Polymorphie
25. Aufgaben
25.1 Aufgabe 1: STL
25.2 Aufgabe 2: Klassen
25.3 Aufgabe 3: Zur Vererbung von Klassen
25.4 Aufgabe 4: Zum Überladen von Methode
25.5 Loesungen

25.4 Aufgabe 4: Zum Überladen von Methoden

Anleitung:

1. Laden Sie die Files zur Aufgabe:
   .tar.gz oder .zip
2. Entpacken Sie die Aufgabe:

   ```
   > gunzip ueberladen.tar.gz
   > tar xf ueberladen.tar
   ```

3. In dem Projekt ist ein Makefile enthalten, mit dem das Projekt compiliert werden kann:

   ```
   > make
   ```

Beschreibung:

In dieser Aufgabe soll im wesentlichen das in der letzten Übung entstandene Programm verfeinert werden. Deshalb gibt im Verzeichnis zunächst nur dieses Hauptfile. Desweiteren werden aber

- `ueglied.h`
- `ueglied.cpp`
- `senke.h`
- `senke.cpp`

aus der letzten Übung benötigt (alternativ tun's auch die entsprechenden Files)

Als erstes soll die ursprünglich als Konstante definierte Variable T0 (ueglied.cc) als statische Variable in die Oberklasse UeGlied. Da die Variable üblicherweise private ist, muss natürlich auch eine statische Methode `SetT0()` eingeführt werden. Auf die Vermeidung von möglichen Inkonsistenzen, die auftreten, wenn man erst ein UeGlied erstellt und dann T0 ändert, soll hier nicht weiter eingegangen werden. Damit die Simulation tatsächlich zu den gleichen Ergebnissen führt, muß auch noch eine Methode `GetT0()` existieren, mit der die Simulations-Funktion seine Zeitachse skalieren kann.
Als nächstes soll die Möglichkeit geschaffen werden mittels

```
cout << pglied;
```

d.h. also mit dem Anwenden des <<-Operators eine Beschreibung (Typ, Verstärkungsfaktor, ...) des Übertragungsgliedes ausgeben zu können. dazu muß der Operator (als friend; nicht als Memberfunktion!) entsprechend überladen werden.

Schließlich (wenn die Zeit reicht) soll ein nichtlineares Übertragungsglied mit Hysterese programmiert werden. Dazu muß einer der Methoden `integrate()` oder `calc()` der Oberklasse UeGlied als virtuell deklariert und überladen werden. Wie und welche bleibt Dir überlassen! Die Neue Klasse sollte auf den Namen `ZPHRegler` hören.

Am Ende sollten alle mit //####### markierten Zeilen funktionstüchtig sein!

rtr Startseite
Einführung
Inhalt
1. Vorbemerkungen
I. Linux
II. C
III. C++
17. Das Klassenkonzept
18. Basistechniken
19. Templates
20. Die Standardbibliothek STL
21. Klassen II
22. Fortgeschrittene Techniken
23. Vererbung
24. Polymorphie
25. Aufgaben
25.1 Aufgabe 1: STL
25.2 Aufgabe 2: Klassen
25.3 Aufgabe 3: Zur Vererbung von Klassen
25.4 Aufgabe 4: Zum Überladen von Methode
25.5 Loesungen

25.5 Loesungen

Übung 2: Antworten

1. Den Eingangswert kann man auch direkt als Parameter an `calc()` übergeben.
2. Es gibt keine direkte Verbindung zwischen den Klassen, die Klasse `PGlied` bzw. `Pt1Glied` ist lediglich Paramter einer Funktion.
3. Man könnte eine Aggregationsbeziehung zwischen der Klasse `SprungSimulator` und `PGlied` erstellen, d.h. `SprungSimulator` würde einen Pointer auf `PGlied` enthalten, der z.B. über eine Methode `setSystem(PGlied* pg)` gesetzt wird. Dieser Zeiger wird im Konstruktor mit 0 initialisiert. In der `start()` Methode wird geprüft, ob der Zeiger nicht 0 ist, bevor das System, auf das der Zeiger zeigt, simuliert wird.
4. Das zu simulierende System ist hart codiert (`PGlied` oder `Pt1Glied`)

Wer möchte, kann die vorgeschlagenen Änderungen durchführen.
Die Lösungen für die Übungen sind hier zu finden:

1. STL: .tar.gz oder .zip
2. Klassen: .tar.gz oder .zip
3. Vererbung: .tar.gz oder .zip
4. Überladen .tar.gz oder .zip