

Version 4.01 - 12. April 2012

---

**Hydroinformatik I**  
**”C++ und Visual C++”**  
**Olaf Kolditz & Bastian Kolditz**

---

**TU Dresden / UFZ Leipzig**  
**Angewandte Umweltsystemanalyse**  
**Umweltinformatik**  
**SoSe 2012**

© OGS Teaching 2012

## Vorlesungskonzept

Ein wesentlicher Bestandteil der Hydroinformatik ist die Informationsverarbeitung. Typische Hydrodaten sind z.B: digitale Geländemodelle (DGM), Niederschlagsmessungen, Wasserpegel, Bodentemperaturen, Bodenkarten (Bodentypen), geologische Schnitte u.v.a. (s. Abb. 1 und 2)

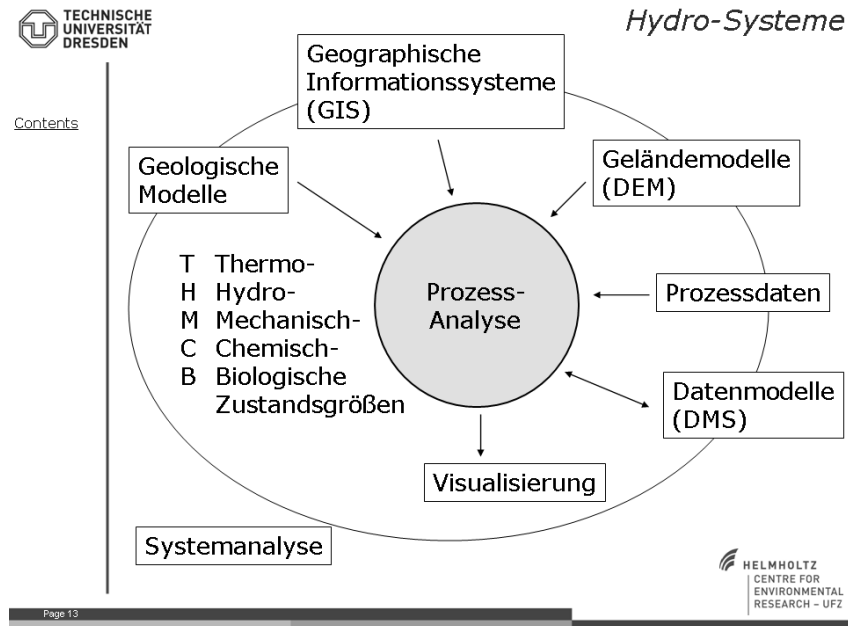


Abbildung 1: Informationsverarbeitung in der Hydrologie - Datentypen

Eine grundlegende Methodik für die Informationsverarbeitung ist die wissenschaftliche Programmierung. Daher beschäftigen wir uns im ersten Semester der Veranstaltung 'Hydroinformatik' mit der objekt-orientierten Programmierung in C++.

Es gibt zwei Möglichkeiten eine Programmiersprache zu erlernen: mehr theoretisch oder mehr praktisch. Wir beschreiten den zweiten Weg. Das heißt anhand von dem, was wir für das Programmieren benötigen, erarbeiten wir uns die Theorie. Unser Motto ist 'Learning by doing'.

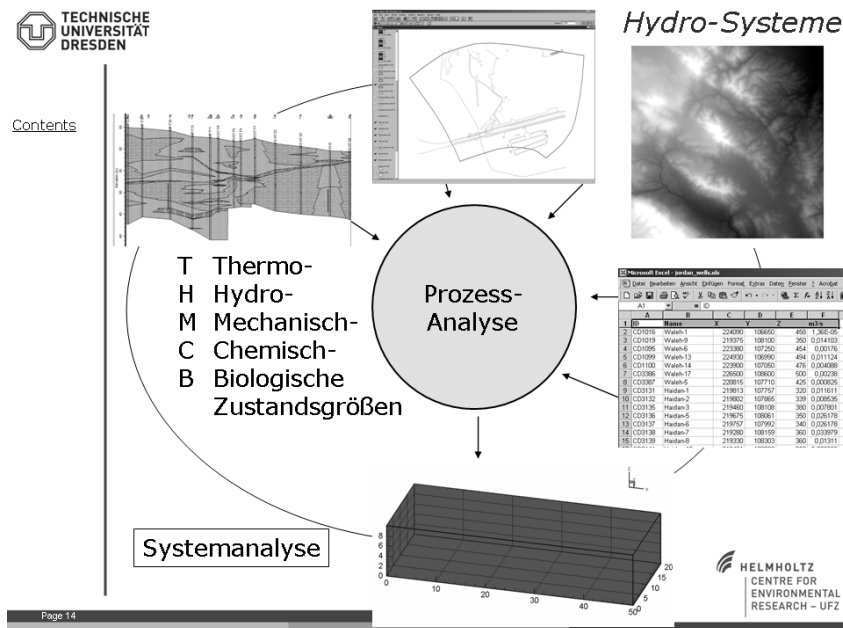


Abbildung 2: Informationsverarbeitung in der Hydrologie - Beispiel aus der Hydrogeologie

## Organisatorisches

Die Prüfung 'Hydroinformatik I' (erstes Semester) ist eine Klausur. Die Prüfung 'Hydroinformatik II' (zweites Semester) besteht aus zwei Teilen, einer Klausur und einer Programmier-Hausarbeit. Bei den Klausuren geht es um die Beantwortung von Verständnisfragen (siehe Testfragen zu jeder Vorlesung).


Sprache: normalerweise in Deutsch. Da die Syntax von Computersprachen (wie C++) immer in Englisch ist, würde ich auch mal eine Vorlesung in Englisch anbieten.

Konsultationen: immer nach der Vorlesung (Freitag ab 11 Uhr).

Kontakt: jederzeit by e-mail (olaf.kolditz@ufz.de), in dringenden Fällen auch per Handy (0151 52739034), ich empfehle auch eine mailing Liste der Studenten anzulegen, wenn alle damit einverstanden sind.

Vorlesungsunterlagen: lege ich erstmal auf meinem UFZ Server ab (<http://www.ufz.de/index.php?de=11877>).

Es gibt viele und sehr gute C++ Bücher. Bei Amazon finden sie locker über 50 C++ Bücher (Abb. 3). Sehr gut finde das Buch von Prinz & Prinz "C++ - Lernen und professionell anwenden", da es eine gute Kombination aus Theorie und Programm-Beispielen ist. Zu diesem Lehrbuch gibt es auch ein extra Übungsbuch. Unsere Vorlesung lehnt sich an dieses Buch an.






TECHNISCHE  
UNIVERSITÄT  
DRESDEN




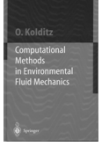
### 50++ C++ Bücher bei Amazon ...

*Literature*


Contents

C++ - Lernen und professionell anwenden von Ulla Kirch-Prinz und Peter Prinz  
 Die C++-Programmiersprache. Deutsche Übersetzung der Special Edition von Bjarne Stroustrup  
 C/C++ Kompendium: Das komplette Programmierwissen für Studium und Job von Dirk Louis

Einstieg in Visual C++ 2008 von André Willms  
 C/C++: Von den Grundlagen zur professionellen Programmierung von Ulrich Kaiser und Christoph Kecher  
 C++ für Spieleprogrammierer von Heiko Kalista



HELMHOLTZ  
CENTRE FOR  
ENVIRONMENTAL  
RESEARCH - UFZ

Page 9

Abbildung 3: C++ Literatur

Das heisst, etwas anders wird unsere Vorgehensweise sein. Im ersten Semester wollen wir zielgerichtet die C++ Grundlagen anhand der Entwicklung einer einfachen 'Datenbank'-Anwendung erlernen - learning-by-doing - und die Daten schließlich in einer graphischen Benutzerschnittstelle (GUI) sichtbar machen. Somit beschäftigen wir uns auch zielgerichtet mit den Grundlagen von Visual C++.

Das Vorlesungskonzept:

- Zielgerichtetes Erlernen der C++ Grundlagen,
- Entwicklung einer 'Datenbank'-Anwendung,
- Umsetzung in eine graphische Benutzerschnittstelle (Visual C++, C++ GUI), siehe Abb. 4.

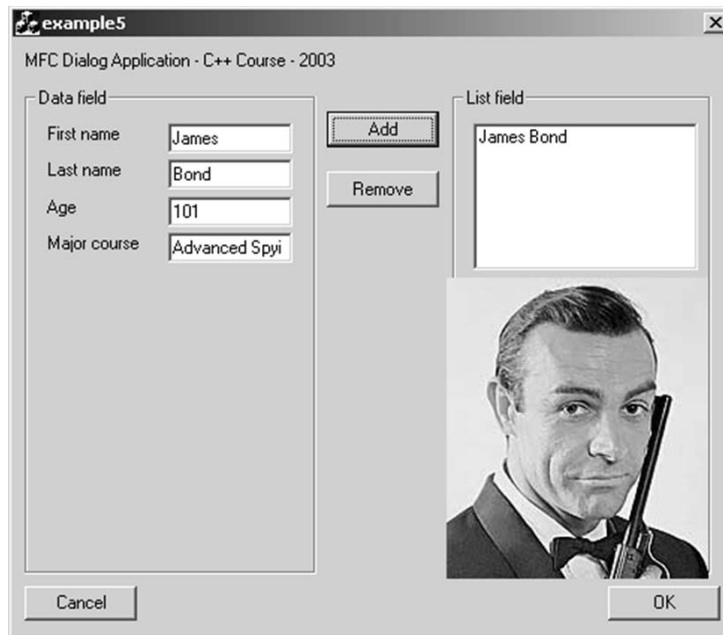


Abbildung 4: Vorlesungs-Konzept

Die Implementierung des Dialogs in Abb. 4 ist das erste greifbare Ziel der Veranstaltung Hydroinformatik I. In dieser GUI-Anwendung können sie (fast) alles, was wir lernen werden, wiederfinden: Datentypen, Ein- und Ausgabe mit verschiedenen Geräten (Tastatur, Bildschirm, Dateien), Klassen, Vererbung, Container (Listen, Vektoren) und graphische Programmierung unter Windows. Es lohnt sich also dranzubleiben ...

**Part I**

**C ++ Basics**

# Kapitel 1

## Einführung

## 1.1 Historisches

Die Programmiersprache C++ wurde von dem dänischen Informatiker Bjarne Stroustrup in den Bell Labs des nordamerikanischen Telekommunikationskonzerns AT&T (American Telephone & Telegraph Corporation) ab 1979/80 entwickelt. C++ stellt eine Weiterentwicklung der ebenfalls von AT&T entwickelten, Programmiersprache C dar. Die Programmiersprache C wurde als Grundlage für C++ aufgrund seiner effizienten Codeverarbeitung und einfachen Übertragbarkeit auf anderen Plattformen verwendet. C wurde in C++ nach dem Vorbild der Programmiersprache Simula-67 (1967), der ersten objektorientierten Programmiersprache, im Wesentlichen um ein Klassenkonzept, d.h. die Abbildung von Objekten in verschiedenen Klassen mit unterschiedlichen Attributen, erweitert. Daher wurde C++ zunächst unter dem Namen 'C mit Klassen' (C with classes) entwickelt. Der Name C++ wurde 1983 von Rick Mascitti erstmals angewandt. Dabei ist das ++ eine Andeutung an den Inkrement-Operator (Inkrement: Erhöhung einer Variable um 1), der die Verbesserungen gegenüber dem Vorgänger C zum Ausdruck bringen soll. Im Jahr 1985 erschien die erste Version von C++, die eine wichtige Grundlage für die weitere Entwicklung und Standardisierung von C++ darstellte, da die Sprache zu dieser Zeit noch nicht akzeptiert war. Im Jahre 1989 erschien die erweiterte Version 2.0 von C++. In den 90er Jahren begann nun der Standardisierungs- und Vereinheitlichungsprozess von C++, der eingeleitet wurde durch das Buch 'The Annotated C++ Reference Manual' (1990) von Margaret Ellis und Bjarne Stroustrup. Die genormte Fassung von C++ erschien 1998 (ISO/IEC 14882:1998), die 2003 mit einer neuen Version (ISO/IEC 14882:2003) nochmals nachgebessert wurde.



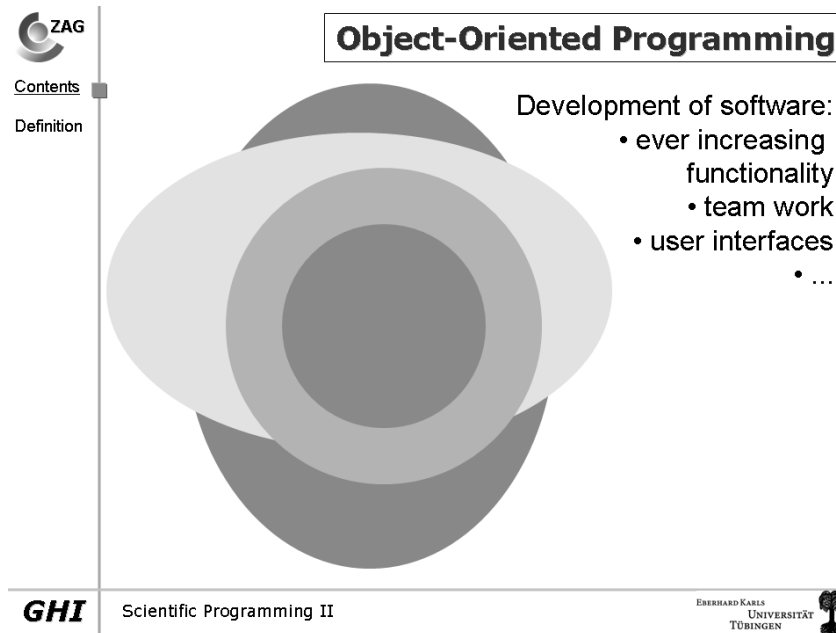
Abbildung 1.1: Bjarne Stroustrup (<http://www.research.att.com/bs/>)



## 1.2 Paradigmen

Programmier-Paradigmen sind sogenannte Leitlinien / Konzepte für die Gestaltung von Programmen, diese können durch entsprechende Sprachmittel unterstützt werden. Allerdings bedeutet die Benutzung einer objekt-orientierten Sprache nicht automatisch ein objekt-orientiertes Programm (OOP).

- Prozedurales Programmieren: Welche Prozeduren benötigen Sie für Ihr Programm ? Verwenden Sie die optimalen Algorithmen.
- Modulares Programmieren: Welche Module benötigen Sie für Ihr Programm ? Strukturieren Sie Ihr Programm entsprechend modular. Die Daten sollten in den Modulen gekapselt sein.
- Objekt-orientiertes Programmieren: Welche Klassen benötigen Sie für Ihr Programm ? Überlegen Sie, welche Gemeinsamkeiten die Klassen haben, benutzen Sie das Konzept der Vererbung.
- Datenabstraktion: Welche Datentypen benötigen Sie für Ihr Programm ? Erstellen Sie für jeden benutzerdefinierten Typen möglichst viele Operationen.



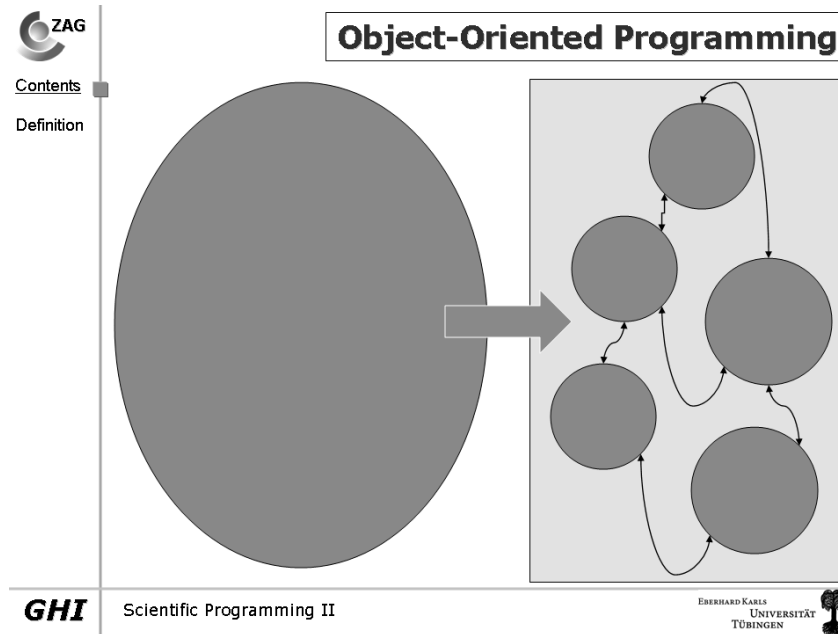


Abbildung 1.2: Objekt-orientiertes Programmieren - OOP

Die Abb. 1.2 soll das Konzept des Objekt-Orientierten Programmierens (OOP) mal aus einer anderen Sichtweise erläutern. Normalerweise bedeutet die Programmentwicklung (prozedural und modular) eine stetige Erweiterung des Codes (Abb. oben). Irgendwann ist das Programm dann so unübersichtlich geworden, dass eigentlich niemand mehr durchblickt. Die grundlegenden Idee von OOP dagegen sind, dass, erstens Objekte (logische Einheiten des Programms) streng gekapselt sind (sich also nicht gegenseitig beeinflussen) und, zweitens, dass es ganz schmale Schnittstellen zwischen diesen Objekten gibt (z.B. nur über eine Adresse (sogenannte pointer)) (Abb. unten). Natürlich wachsen auch diese Objekte weiter und stoßen irgendwann an Grenzen ...

Ein Beispiel aus der Praxis unserer eigenen Programmentwicklung OpenGeoSys. Die Abb. 1.3 zeigt das objekt-orientierte Konzept der Kernels von OpenGeoSys zur Lösung von partiellen Differentialgleichungen. Letztlich führen alle numerischen Methoden zur Lösung von linearen Gleichungssystemen  $Ax = b$ . Daher haben wir ein Objekt konstruiert (PCS von ProCesS), das die Aufgabe übernimmt, alle Bausteine für die Assemblierung und Lösung der Gleichungssysteme einzusammeln. Dabei geht es um geometrische (Punkte, Polylinien, Flächen, Volumen), topologische (Element-Netze), numerische und physikalische Daten (z.B. Materialeigenschaften, Anfangs- und Randbedingungen). Entscheidend für die Machbarkeit eines objekt-orientierten Konzepts ist, dass der **Algorithmus** zum Aufstellen und Lösen der Gleichungssysteme vollkommen unabhängig von dem spezifischen physikalischen Problem ist. Die Basisklasse

für das PCS-Objekt ist CProcess, davon abgeleitet werden die speziellen Klassen für die spezifischen Probleme (z.B. Strömung, Stoff- und Wärmetransport, Deformation). Wenn Sie diesen Abschnitt noch mal durchlesen, sollten sie allerdings stutzig werden, da neben dem erwarteten Begriff der Objekt-Orientierung auf der des **Algorithmus** auftaucht. Richtig, OpenGeoSys ist kein reines OOP sondern eine Kombination von objekt-orientierten und prozeduralen Konzepten. Anyway, der Schritt von C zu C++ (3. zur 4. Version, Abb. 1.3) bedeutete eine Reduzierung des Quellcodes von 10MB auf 3MB, also um 70% !!!

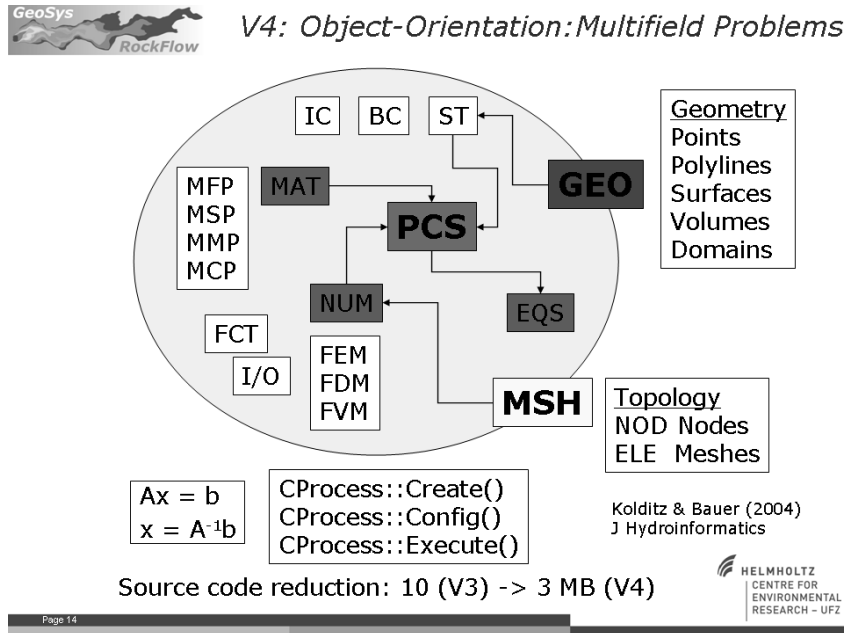


Abbildung 1.3: Objekt-orientiertes Konzept des Kerns von OpenGeoSys

## 1.3 Compiler

Bevor wir irgendetwas programmieren können, benötigen wir zwei Werkzeuge: einen Editor und einen Compiler. In der Abb. 1.4 sind die wichtigsten Schritte der Programmerstellung zu sehen.

In der Computer-Welt (und deshalb auch in der Programmier-Welt) gibt es zwei verschiedene Philosophien: Microsoft (MS) und 'den Rest'. Um diesem Problem 'auszuweichen' und fair zu sein, benutzen wir Werkzeuge aus beiden Welten. Wir beleuchten dieses 'Problem' später noch mal, wenn wir über Open Source Projekte (z.B. OpenGeoSys, unsere eigene Software-Entwicklung) sprechen. Wir installieren uns zunächst den GNU Compiler (Abschn. 1.3.1) und später den MS Compiler (Abschn. 1.3.2) für das visuelle C++ Programmieren.

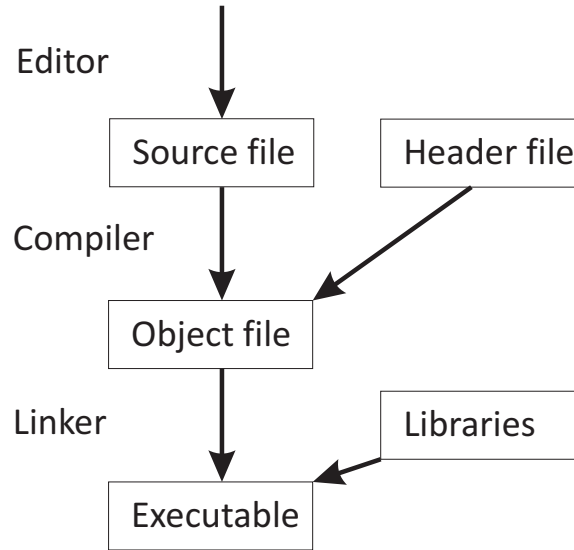


Abbildung 1.4: Quell-Code Kompilation

### 1.3.1 GNU

Das GNU-Projekt wurde von Richard Stallman mit dem Ziel gegründet, ein vollständig freies Betriebssystem, genannt GNU ("GNU is not Unix"), zu entwickeln. Bekannt geworden ist das Projekt vor allen Dingen auch durch die von ihm eingeführte GNU General Public License (GPL), unter der viele bekannte Softwareprojekte veröffentlicht werden (Quelle: Wikipedia).

Wenn ihr mehr über das GNU Projekt erfahren wollt, schaut nach unter <http://de.wikipedia.org/wiki/GNU-Projekt>

Als Kompromiss benutzen wir cygwin (Abb. 1.5). Das ermöglicht uns, auch unter Windows mit GNU Software arbeiten zu können. Die Linux / Ubuntu 'Freaks' haben die GNU Software automatisch mit dem Betriebssystem auf ihrem Rechner. Die Anleitung zur Installation finden sie im Anhang, Abschn. 11.3.2.2 (und wird natürlich in der Vorlesung besprochen)

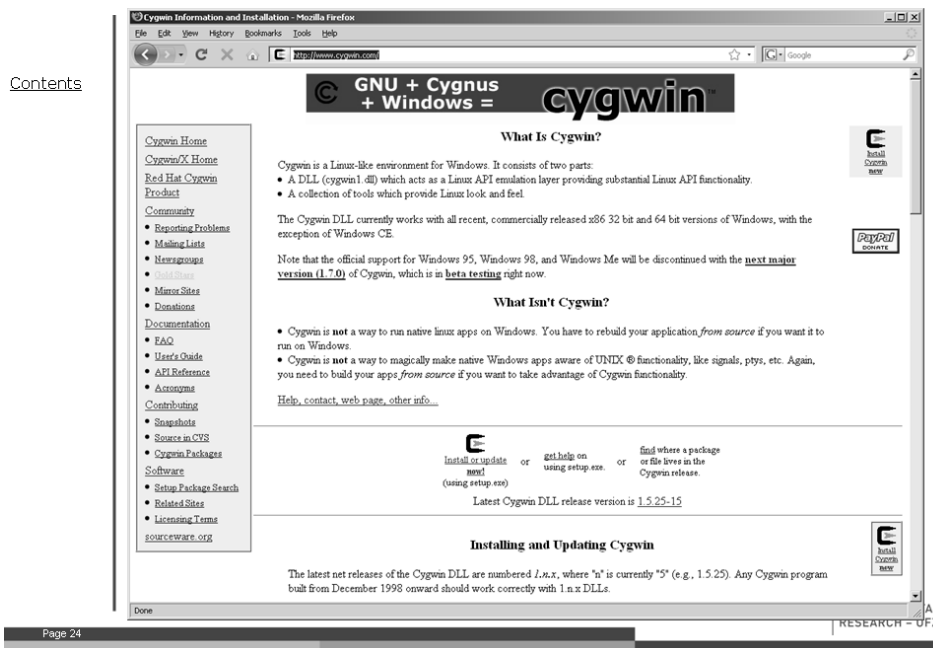


Abbildung 1.5: Ein Kompromiss: cygwin

### 1.3.2 MS Visual C++

Auch die nicht-MS-Fans müssen zugeben, das MS Visual C++ Studio ist eine tolle Entwicklungsumgebung. Die wird uns enorm helfen, wenn wir uns später mit GUI Programmierung beschäftigen. Die Express-Version von VC++ kann kostenlos direkt bei Microsoft unter <http://www.microsoft.com/germany/express/download/webdownload.aspx> heruntergeladen werden.

### 1.3.3 Qt

Als plattformunabhängiges visuelles C++ empfehle ich Qt (siehe Abschn. 10 und 11.4)

## 1.4 "Hello World"

Unser allererstes C++ Programm.

### 1.4.1 Grundlagen

Jedes C/C++ Programm benötigt eine `main()` Funktion

```
int main()
{
    return 0;
}
```

Die `main()` Funktion ist der 'Startpunkt' eines jeden C/C++ Programms. Die wichtigsten Bausteine einer Funktion sind:

- Typ (Rückgabewert): `int` (`return 0`)
- Name: `main`
- Argumente: `()`, man kann die `main` Funktion auch mit Argumenten aufrufen, z.B. der Name der Eingabe-Datei,
- Funktionsrumpf: ...

Die Struktur der `main()` Funktion ist in der Abb. 1.6 dargestellt.

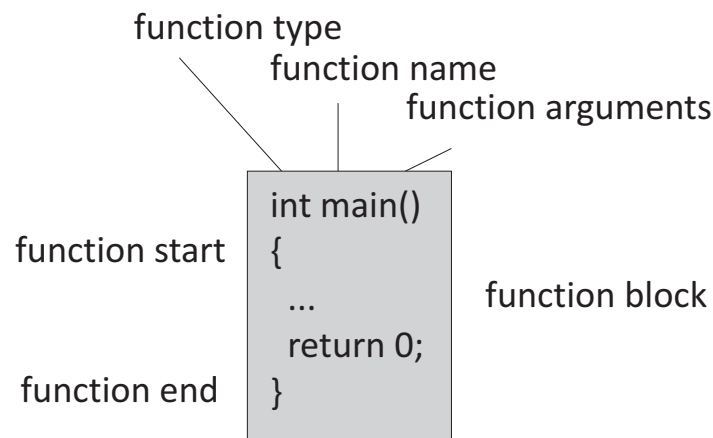


Abbildung 1.6: Struktur der Funktion `main()`

## 1.4.2 Exercise E1



Abbildung 1.7: cygwin aufrufen



Abbildung 1.8: cygwin drive ist /cygwin/home/user, wechseln zum Arbeitsverzeichnis cd c:/myworkdirectory (oder copy und paste)



Abbildung 1.9: slash (/) ist nicht gleich slash (\)

Die einzige C Übung:

```
#include <stdio.h>
int main()
{
    printf("Hello World");
    return 0;
}
```



Abbildung 1.10: Kommandozeile: gcc main.c, das Ergebnis ist a.exe, Aufruf des Programms: ./a

Jetzt endlich zu C++ ...

```
#include <iostream>
using namespace std;
int main()
{
    cout << "Hello World" << endl;
    return 0;
}
```



```
~/cygdrive/c/User/TEACHING/C++/EXERCISES/E01
adm_lokal@envinf1071 ~
$ cd C:/User/TEACHING/C++/EXERCISES/E01
adm_lokal@envinf1071 /cygdrive/c/User/TEACHING/C++/EXERCISES/E01
$ gcc main.c
adm_lokal@envinf1071 /cygdrive/c/User/TEACHING/C++/EXERCISES/E01
$ g++ main.cpp
adm_lokal@envinf1071 /cygdrive/c/User/TEACHING/C++/EXERCISES/E01
$ ls
a.exe clean.bat main.c main.cpp
adm_lokal@envinf1071 /cygdrive/c/User/TEACHING/C++/EXERCISES/E01
$ ./a
Hallo C++ World
adm_lokal@envinf1071 /cygdrive/c/User/TEACHING/C++/EXERCISES/E01
$
```

Abbildung 1.11: Kommandozeile: g++ main.cpp, mit ls den Inhalt des Verzeichnisses ansehen, das Ergebnis ist a.exe, Aufruf des Programms: ./a

## 1.5 Students Forum

Vielen Dank für Ihre Fragen zur Vorlesung!

**JB** Der C compiler funktioniert, der C++ aber nicht: Wahrscheinlich haben sie nur den C Compiler (GCC) installiert. Die können jederzeit cygwin Komponenten nachinstallieren, siehe Abschn. 11.3.2.2.

**DB** Ich habe den MS VC++ installiert und kann die Übungen nachvollziehen. Brauche ich den GNU Compiler von cygwin wirklich noch?: Eigentlich nicht, mit MS VC++ haben sie einen kompletten Compiler mit allem drum und dran, dennoch kann ein Ausflug in die Konsolenwelt ja nicht schaden ... und sie können später in ihre Bewerbungen schreiben, dass sie auch unter Linux Programme entwickelt haben.

**MF** Die aktuelle cygwin Installation sieht anders aus als im Skript beschrieben: [OK] Im Anhang (Abschn. 11.3.2.2) finden sie eine bessere Beschreibung der cygwin Installation.

**C++ und Mac** Hier die Adresse <http://developer.apple.com/TOOLS/xcode/> wo sie den MAC C++ compiler herunterladen und installieren können.



**Studentin, 07.05.2010** "Das ist zwar alles sehr schön - aber sehr verwirrend. Wie bekomme ich mein Programm zum Laufen?" Ich gebe zu, ich bombardiere sie mit neuen Begriffen (IT slang). Schauen sie sich bitte noch mal die Abb. 1.4 an. Dort sehen sie die einzelnen Schritte zur Erstellung eines Programms. Ich erkläre es mal step-by-step.

1. Quell-Datei: Zunächst müssen sie eine Quell-Datei erstellen (oder eine aus den Übungen nehmen). Hierfür können sie einen beliebigen Editor nehmen (z.B. Notepad von Windows). Diese Datei hat die Endung (extension) `.cpp` und enthält die `main()` Funktion.
2. Kompilation: Im nächsten Schritt muss diese Datei kompiliert, d.h. in Maschinencode übersetzt werden. Hierfür muss ein C++ Kompiler (z.B. `g++` von cygwin) installiert sein. Die cygwin Anweisung (also auf der Kommandozeile im cygwin Fenster) ist: `g++ main.cpp`. Im Ergebnis (wenn alles gut gegangen ist, also keine Übersetzungsfehler aufgetreten sind) erhalten sie eine ausführbare Datei (executable), normalerweise: `a.exe`.
3. Jetzt können wir das Programm starten mit `./a.exe` von der Kommandozeile. Das `./` ist wichtig, um dem Betriebssystem mitzuteilen, dass wir das Programm vom aktuellen Verzeichnis (wo `a.exe` liegt) starten wollen. Viel Glück.

## 1.6 Testfragen

1. Was bedeutet das `++` im Namen der Programmiersprache C++ ?
2. Ist C++ eine standardisierte Programmiersprache ?
3. Was ist der Unterschied zwischen C und C++ ?
4. Was sind sogenannte Programmier-Paradigmen ?
5. Welche verschiedenen Programmier-Paradigmen kennen sie und worin unterscheiden sie sich ?
6. Welches Paradigma verfolgen wir in der Vorlesung ?
7. Was ist objekt-orientierte Programmierung ?
8. Was ist ein Kompiler ?
9. Erklären sie die einzelnen Schritte bei einer Kompilierung (s. Abb. 1.4).
10. Welchen Kompiler benutzen sie für die C++ Übungen ?
11. Welche Funktion benötigt jedes C oder C++ Programm ?

12. Was ist der Typ einer Funktion ?
13. Was ist eine Parameterliste einer Funktion ?
14. Was ist der Rückgabewert einer Funktion ?

# Kapitel 2

## Datentypen

Jede Programmiersprache bietet die Verwendung elementarer Datentypen an, wie z.B. Ganzzahlen oder Gleitkommazahlen, logische Ausdrücke oder Zeichenketten. Die Besonderheit von objekt-orientierten (OO) Sprachen ist die Erstellung benutzerdefinierter Datentypen. Dies war schon in der Sprache C mit sog. 'typedef struct' Typen möglich. OO Sprachen, wie C++, bieten aber noch mehr an, die Verknüpfung von benutzerdefinierten Datentypen mit den entsprechenden Methoden. Diese Konstrukte nennt man Klassen, aber darum geht es erst im nächsten Kapitel. Zunächst müssen wir uns (kurz) mit den handelsüblichen Datentypen herumschlagen.

### 2.1 Elementare Datentypen

Die nachfolgende Tabelle zeigt eine Übersicht der elementaren Datentypen.

| Typ              | Bezeichner  | Erläuterung                              |
|------------------|-------------|--|
| Wahrheitswerte   | bool        | Wert kann true (=1) oder false (=0) sein |
| Zeichen          | char        | Zeichen, statische Zeichenketten         |
|                  | wchar_t     | 'wide character type' ( $\geq 2$ Byte)   |
|                  | string      | Dynamische Zeichenketten                 |
| Ganzzahlen       | short       | Ganze Zahlen mit geringer Genauigkeit    |
|                  | int         | Ganze Zahlen mit hoher Genauigkeit       |
|                  | long        | Ganze Zahlen mit sehr hoher Genauigkeit  |
| Gleitpunktzahlen | float       | Reelle Zahlen mit geringer Genauigkeit   |
|                  | double      | Reelle Zahlen mit hoher Genauigkeit      |
|                  | long double | Reelle Zahlen mit sehr hoher Genauigkeit |

Tabelle 2.1: Übersicht elementarer Datentypen

## 2.2 Speicherbedarf

Die unterschiedlichen Datentypen werden durch den Compiler verschieden verarbeitet und gespeichert (Größe des benötigten Speicherplatzes).

| Typ         | Speicherplatz | Wertebereich   |
|-------------|---------------|--|
| bool        | 1 Byte        | (weniger geht leider nicht)                            |
| char        | 1 Byte        | -128 bis + 127 bzw. 0 bis 255                          |
| int         | 2/4 Byte      | -32768 bis +32767 bzw.<br>-2147483648 bis + 2147483647 |
| short       | 2 Byte        | -32768 bis +32767                                      |
| long        | 4 Byte        | -2147483648 bis + 2147483647                           |
| float       | 4 Byte        | $\pm 3.4E+38$ , 6 Stellen Genauigkeit                  |
| double      | 8 Byte        | $\pm 1.7E+308$ , 15 Stellen Genauigkeit                |
| long double | 10 Byte       | $\pm 1.7E+4932$ , 19 Stellen Genauigkeit               |

Tabelle 2.2: Speicherbedarf elementarer Datentypen

Der sizeof Operator ermittelt die benötigte Speichergröße für einen bestimmten Datentypen, z.B. `sizeof(float) = 4`: eine einfach genaue Gleitkommazahl benötigt 4 Byte Speicher. Wir verwenden den sizeof Operator in der Übung E34.

## 2.3 Escape-Sequenzen

Escape-Sequenzen sind nützliche Helfer, z.B. zum Formatieren von Ausgaben oder für die Benutzung von Sonderzeichen in der Ausgabe.

| Zeichen         | Bedeutung            | Wirkung                |
|-----------------|----------------------|------------------------|
| <code>\a</code> | alert (BEL)          | Ausprobieren           |
| <code>\b</code> | backspace (BS)       | eine Position zurück   |
| <code>\t</code> | horizontal tab (HT)  | horizontaler Tabulator |
| <code>\n</code> | line feed (LF)       | Zeilenumbruch          |
| <code>\v</code> | vertical tab (VT)    | vertikaler Tabulator   |
| <code>\r</code> | carriage return (CR) | Zeilenumbruch          |
| <code>\0</code> | end of string        | Zeilenende             |
| <code>\"</code> | "                    | Zeichen                |
| <code>\'</code> | '                    | Zeichen                |
| <code>\?</code> | ?                    | Zeichen                |
| <code>\\</code> | \                    | Zeichen                |

Tabelle 2.3: Nützliche Escape-Sequenzen

Anmerkung:

Die Größe eines Datentyps wird vom C++ Standard eigentlich nicht vorgegeben und kann Rechnerarchitektur und Compiler abhängig sein. In der Regel ist

der Speicherbedarf elementarer Datentypen aber gleich, z.B. `double` benötigt i.d.R. 8 Byte. Es gibt allerdings auch Ausnahmen, z.B. für `string`. Die Klasse `std::string` gehört zur Standard-Bibliothek. Die Größe eines Strings ist abhängig von Rechnerart, Betriebssystem und Compiler. Ein 'leerer' String kann zwischen 4 und 8 Byte lang sein.

## 2.4 Testfragen

1. Was ist der genaueste Datentyp in C++ ?
2. Wie groß ist der Speicherbedarf von einem `string` Datentyp ?
3. Mit welcher Anweisung können wir den Speicherbedarf von elementaren Datentypen ermitteln ?
4. Was sind Escape-Sequenzen ?
5. Was ist der Unterschied zwischen den Escape-Sequenzen `\n` und `\r` ?
6. Was ist die C++ Entsprechung der Klasse `cout` für einen Zeilenumbruch ?

## Kapitel 3

# Ein- und Ausgabe

Ein Programm (ob prozedural, modular, objekt-orientiert) ist eigentlich nichts weiter als eine Datenverarbeitung zwischen einer Eingabe (Input) und einer Ausgabe (Output). I und O können mehr oder weniger schick gemacht sein:

1. I/O Standardgeräte,
2. I/O Dateien,
3. Datenbanken (I) und Visualisierung (O).

Aus didaktischen Gründen müssen wir leider mit dem Langweiligsten - I/O Standardgeräte - anfangen. Spannend wird's, wenn Daten durch die Visualisierung 'lebendig' werden. Die Abb. 3.1 zeigt eine professionelle Datenaufbereitung eines Porenraummodells in unserem Labor für wissenschaftliche Visualisierung (TESSIN-VISLab) am UFZ in Leipzig.

### Die `iostream` Klasse

Die Ein- und Ausgabe in C++ erfolgt mit sogenannten Strömen (**streams**). Die I/O-Stream-Klassen (Abb. 3.2) bieten vielfältige Möglichkeiten, die wir uns in den nachfolgenden Übungen näher anschauen.



Abbildung 3.1: Wissenschaftliche Visualisierung

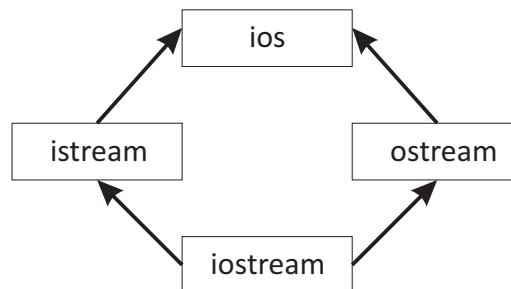


Abbildung 3.2: I/O stream Klassen

Die Klasse `iostream` geht durch Mehrfachvererbung aus den Klassen `istream` und `ostream` hervor. `iostream` stellt damit die Funktionalität beider I/O Klassen zu Verfügung.

## 3.1 Die Standard-Streams

Es gibt vier Standard-Streams:

- `cin`: Standard-Eingabe über die Tastatur, Objekt der Klasse `istream`
- `cout`: Standard-Ausgabe auf dem Bildschirm, Objekt der Klasse `ostream`
- `cerr` und `clog`: zwei Objekte der Klasse `ostream` für die Fehlerausgabe.

Exercise E3.1:

```
#include <iostream>
using namespace std;
int main()
{
    int zahl;
    cout << "Bitte eine ganze Zahl eingeben: ";
    cin >> zahl;
    cout << zahl << endl;
    return 0;
}
```

Die Ein- `>>` und Ausgabeoperatoren `<<` transportieren die Ströme von und zu den Eingabe- bzw. Ausgabegeräten. Dabei formatieren sie die Datentypen (z.B. `int` in der Übung E3.1) entsprechend den Einstellungen der Klasse `ios`. Diese Einstellungen können durch Flags verändert werden (siehe nächsten Abschnitt).

## 3.2 Formatierte Ausgaben

Wir beschäftigen uns mit der Gestaltung, d.h. Formatierung, von Ausgaben, wir wollen die Bildschirmausgabe schick machen, z.B. in Tabellenform, dass alles schön untereinander steht. Der zweite Aspekt der Formatierung ist die Genauigkeit von ausgegebenen Zahlenwerten.

### 3.2.1 Formatierte Ausgabe von Ganzzahlen

In der nachfolgenden Übung beschäftigen wir uns mit den verschiedenen Ausgabemöglichkeiten von ganzen Zahlen.

Exercise E3.2.1:

```
#include <iostream>
using namespace std;
int main()
{
```



```

int zahl;
cout << "Bitte eine ganze Zahl eingeben: ";
cin >> zahl;

cout << uppercase // für Hex-Ziffern
    << " oktall \t\t dezimal \t hexadezimal \n "
    << oct << zahl << " \t\t "
    << dec << zahl << " \t\t "
    << hex << zahl << endl;
return 0;
}

```

### 3.2.2 Formatierte Ausgabe von Gleitpunktzahlen

In der nachfolgenden Übung beschäftigen wir uns mit den verschiedenen Ausgabemöglichkeiten von realen Zahlen.

| Methoden                          | Wirkung                        |
|-----------------------------------|--------------------------------|
| <code>int precision(int n)</code> | Genauigkeit wird auf n gesetzt |

Exercise E3.2.2:

```

#include <iostream>
using namespace std;
int main()
{
    double zahl;
    cout << "Bitte eine Gleitkommazahl eingeben: ";
    cin >> zahl;

    cout.precision(7); // auf sieben Stellen genau

    cout << "Standard: \t" << zahl << endl;
    cout << "showpoint: \t" << showpoint << zahl << endl;
    cout << "fixed: \t\t" << fixed << zahl << endl;
    cout << "scientific: \t" << scientific << zahl << endl;
    return 0;
}

```

### 3.2.3 Ausgabe von Speicherbedarf

In dieser Übung benutzen wir den `sizeof` Operator, um den Speicherbedarf von Standard Daten-Typen zu bestimmen.

Exercise E3.2.3:

```

#include <iostream>

```

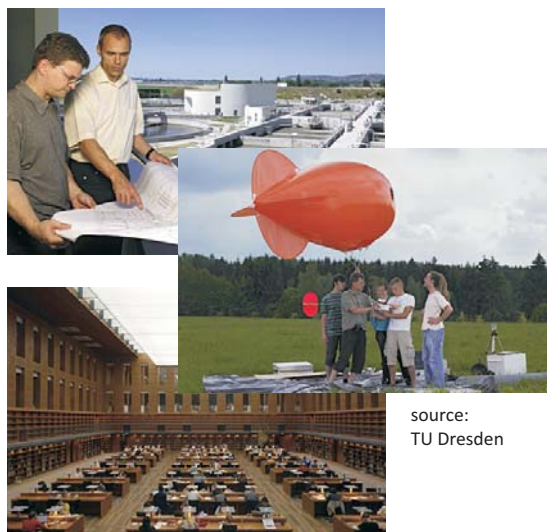
```
using namespace std;
int main()
{
    cout << "Type\tNumber of bytes\n";
    cout << "-----\n";
    cout << "bool\t\t" << sizeof(bool) << endl;
    cout << "char\t\t" << sizeof(char) << endl;
    cout << "short\t\t" << sizeof(short) << endl;
    cout << "int\t\t" << sizeof(int) << endl;
    cout << "long\t\t" << sizeof(long) << endl;
    cout << "float\t\t" << sizeof(float) << endl;
    cout << "double\t\t" << sizeof(double) << endl;
    cout << "long double\t" << sizeof(long double) << endl;
    return 0;
}
```

### 3.3 Testfragen

1. Sind sie mit der Tab. 2.2 einverstanden ?
2. Welche Ein- und Ausgabegeräte kennen sie ?
3. Welche Klasse benötigen wir für die Standard-Ein- und Ausgabe ?
4. Was ist die Basis-Klasse für alle Ein- und Ausgaben in C++ ?
5. Mit welcher Klasse können wir sowohl Eingabe- als auch Ausgabeströme benutzen ?
6. Welche Include-Datei ist notwendig, um mit I/O-Strömen arbeiten zu können ?
7. Wozu dient der Zusatz `using namespace std;` nach dem Include von Standard-Klassen, wie I/O Streams ?
8. Was bewirken die Stream-Operatoren `<<` und `>>` ?
9. Was bewirken die Flags `oct`, `dec`, `hex` für die formatierte Ausgabe von Ganzzahlen ? (ToDo)
10. Wie kann ich die Genauigkeit der Ausgabe von Gleitkomma-Zahlen festlegen ?
11. Wie kann man den Speicherbedarf einer Variable ermitteln ? Schreiben sie die C++ Anweisung für die Berechnung des Speicherbedarfs für eine doppelt-genaue Gleitkomma-Zahl.

# Kapitel 4

## Klassen



Data abstraction ↓

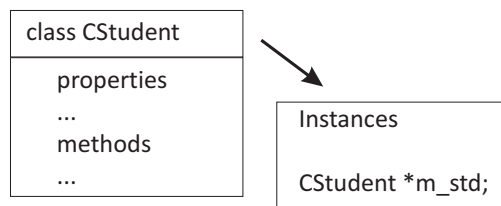


Abbildung 4.1: Das Klassen-Konzept - CStudent

Das Sprachelement der Klassen sind das entscheidende Kriterium von objekt-orientierten Konzepten und die objekt-orientierte Programmierung (OOP). Klassen sind eine Art Schablone für einen benutzerdefinierten Datentypen. Darüber hinaus enthält die Klasse neben den Daten auch alle Methoden (Funktionen), um mit den Daten der Klasse operieren zu können. Unser Beispiel für Klassen, das uns im Verlaufe der Vorlesung beschäftigen wird, ist - wie könnte es anders sein - CStudent (Abb. 4.1). Für die Konzipierung von Klassen spielt die Abstraktion der Daten einer Klasse eine besonders wichtige Rolle.

## 4.1 Daten-Abstraktion

Die Abb. 4.1 illustriert uns, dass eine Abstraktion von Daten (d.h. Eigenschaften) der Klasse Studenten eine durchaus vielschichtige Angelegenheit sein kann. Eine Aufstellung von Daten / Eigenschaften, die es aus ihrer Sicht zu berücksichtigen gilt, ist ihre nächste Hausaufgabe.

Der nächste Block zeigt ihnen das Schema der Syntax der Klassen-Definition CStudent. Das Schlüsselwort für die Klassen-Definition ist `class`, der Name ist CStudent. Der Klassen-Rumpf ist in geschweifte Klammer eingebettet. Wichtig ist der Abschluss mit einem Semikolon. Wie bereits erwähnt, eine Klasse enthält Daten (Eigenschaften) und Methoden (Funktionen) auf den Daten. Prinzipiell geht diese Datenabstraktion auch mit anderen Sprachen wie C. A data construct `typedef struct {...}` in C can be seen as an analogue to a C++ class.

Die C Version:

```
typedef struct
{
    char* name_first;
    char* name_last;
    long matrikel_number;
} TDStudent;
TDStudent *student = NULL;
```

Die C++ Version:

```
class CStudent
{
    data:
    ...
    methods:
    ...
};
```

Bei der Namensgebung von Klassen wird oft ein 'C' (CStudent) vorangestellt, um die Bedeutung als Klasse (wesentlicher Baustein des Programm-Konzepts) herauszustellen.

Ein weiterer Vorzug von OO-Sprachen ist z.B. die Sichtbarkeit / Zugreifbarkeit von Daten zu regeln. Der nachfolgende Block zeigt das Datenschutz-Konzept von C++ (Sicherheitsstufen): Daten können öffentlich sein (public) oder gezielt für 'Freunde' verfügbar gemacht werden (protected) oder nur exklusiv für die eigene Klasse sichtbar zu sein (private).

```
class CStudent
{
    private:
    ...
    protected:
    ...
    public:
    ...
};
```

## 4.2 Klassen-Deklaration

Im vorangegangenen Abschnitt haben wir uns mit der Datenabstraktion mittels Klassen beschäftigt. So sollte konsequenterweise jede Klasse auch ihre eigenen - sorry - eigenen Quelldateien besitzen. Die Deklaration von Klassen erfolgt üblicherweise in einer sogenannten Header-Datei \*.h. Für die Methoden / Funktionen der Klasse ist eine \*.cpp Datei reserviert. Für uns bedeutet dies, zwei Dateien anlegen:

- student.h - die Deklaration der Klasse CStudent
- student.cpp - die Methoden der Klasse CStudent

Um mit der Klasse arbeiten zu können, müssen wir das entsprechende Header-File inkludieren. Dies erfolgt mit der Anweisung `#include "student.h"` am Anfang unseres Main-Files.

```
#include "student.h"
int main
{
    return 0;
}
```

## 4.3 Instanzen einer Klasse

An dieser Stelle möchten wir unsere Eingangsgaphik (Abb. 4.1) erinnern. Instanzen sind Kopien einer Klasse mit denen wir arbeiten können, dass heißt

diese bekommen echten Speicher für ihre Daten (die natürlich für jede Instanz einer Klasse unterschiedlich sein können).

Es gibt zwei Möglichkeiten, Instanzen einer Klasse zu erzeugen:

```
#include "student.h"
void main()
{
    // Creating an instances of a class - 1
    CStudent m_std_A;
    // Creating an instances of a class - 2
    CStudent *m_std_B;
}
```

Der direkte und der mittels eines sogenannten Zeigers (hierfür gibt ein Extra-Kapitel). Wir werden sehen, dass der zweite Weg oft der bessere ist, da wir z.B. die Initialisierung und das Speichermanagement für unsere Daten selber in die Hand nehmen können. Dies können wir mittels sogenannter Konstruktoren und Destruktoren erledigen. Damit beschäftigen wir uns im nächsten Abschnitt.

Exercise E4.3:

```
#include "student.h"
#include <iostream>
using namespace std;
int main()
{
    CStudent *m_std_cpp; // pointer to an instance
    cout << "E41: Instances of classes" << endl;
    cout << "What have we created?\t : " << m_std << endl;
    cout << "What size has it?\t : " << sizeof(m_std) << endl;
    TDStudent *m_std_c; // pointer to TD
    return 0;
}

-> student.h
// Class definition
class CStudent
{
    public:
    protected:
    private:
};
// Type definition
typedef struct
{
} TDStudent;
```

## 4.4 Konstruktor und Destruktor

Wohl die wichtigsten Methoden von Klassen sind deren Konstruktoren `CStudent()` und Destrukturen `~CStudent()`. Diese Funktionen haben den gleichen Namen wie die Klasse selbst. Der nachfolgende Block zeigt die Deklaration unserer ersten beiden Klassen-Funktionen in der Header-Datei `students.h`.

```
class CStudent
{
    public:
        CStudent(); // constructor
        ~CStudent(); // destructor
};
```

Wie sehen die Konstruktor/Destruktor-Funktionen aus.

```
#include "student.h"

CStudent::CStudent()
{
}

CStudent::~CStudent()
{
}
```

Die Konstruktor/Destruktor-Funktionen werden 'automatisch' aufgerufen beim Erzeugen (über die Zeiger-Version) und Zerstören der Klassen-Instanz:

- `new` ruft den dazugehörigen Konstruktor auf,
- `delete` ruft den dazugehörigen Destruktor auf.

Die Anwendung der Konstruktor/Destruktor-Funktionen ist im folgenden Quellcode-Abschnitt dargestellt.

Exercise E4.4:

```
#include <iostream>
using namespace std;
#include "student.h"
int main()
{
    CStudent* m_std = new CStudent(); // instance using constructor
    cout << "E44: Constructor of class CStudent" << endl;
    cout << "What have we created? m_std\t : " << m_std << endl;
    cout << "What size has it?\t : " << sizeof(m_std) << endl;
    cout << "What have we created? &m_std\t : " << &m_std << endl;
    cout << "What size has it?\t : " << sizeof(&m_std) << endl;
    return 0;
}
```

## 4.5 Dateninitialisierung mit dem Konstruktor

Der Klassen-Konstruktor kann darüber hinaus auch zur Initialisierung von Daten für die Klassen-Instanz benutzt werden. Ganz nebenbei schreiben wir unsere erste Klassen-Funktion in der Quell-Datei der Klasse `students.cpp`. Wir initialisieren den Namen unseres ersten Studenten.

Exercise E4.5:

```
CStudent::CStudent()
{
    // Initializations
    name_first = "James";
    name_last = "Bond";
}
```

Dafür müssen wir die notwendigen Klassen-Variablen (member variables) für den Namen bereitstellen. Praktischerweise ist es ratsam, die Daten und Methoden der Klasse zu trennen. Unsere Klassen-Deklaration nimmt langsam Gestalt an ...

```
#include <string>
using namespace std;
class CStudent
{
    // Data / Properties
public:
    string name_first;
    string name_last;
    // Methods / Functions
public:
    CStudent(); // constructor
    ~CStudent(); // destructor
};
```

Für die Namen benutzen wir die `string` Klasse von C++ (auch hierzu gibt's ein Extra-Kapitel). Um den Daten-Typ `string` aus dieser Klasse benutzen zu können, müssen wir die Klasse wie folgt inkludieren.

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

Exercise E4.5:

```
#include <iostream>
using namespace std;
#include "student.h"
```



```
int main()
{
    CStudent* m_std = new CStudent(); // instance using constructor
    cout << "E45: Data initialisation by constructor of class CStudent" << endl;
    cout << "m_std->name_first\t : " << m_std->name_first << endl;
    cout << "m_std->name_last\t : " << m_std->name_last << endl;
    delete(m_std);
    return 0;
}
```

## 4.6 Datenschutz

Eine wichtige Eigenschaft der Sprache C++ ist die Möglichkeit, Daten unterschiedlich sichtbar zu machen. Eingangs des Kapitels haben wir gesehen, dass es verschiedene Schlüsselwörter gibt, um Variablen zu klassifizieren: `public`, `protected` und `private`. Daten des Typs 'public' sind überall sichtbar, aus sie kann von überall im Programm zugegriffen werden - also auch verändert werden. Dies kann sehr schnell problematisch werden, wenn das Programm sehr groß ist oder mehrere Programmierer gleichzeitig entwickeln. Daten des Typs 'private' sind nur für die eigene Klasse sichtbar, dass auf sie können nur durch Klassen-eigene Methoden zugegriffen werden. Private data sind für andere Klassen verborgen. Natürlich gibt es noch einen Kompromiss: Mit dem Schlüsselwort 'protected' kann man gewisse Daten für 'befreundete' Klassen öffnen. Dies ist im Wesentlichen das Datenschutz-Konzept von C++.

In der folgenden Übung sehen wir, wie man mit 'private' data (Kontonummer) umgehen kann. Die Variable `bank_account` ist als `private` Typ deklariert.

```
class CStudent
{
public:
    long GetBankAccount();
    void SetBankAccount(long);
private:
    long bank_account;
};
```

Die Zugriffsfunktion `GetBankAccount()` der Klasse `CStudent` gibt lediglich den Wert der Variable zurück, ihr Typ ist natürlich `public`. Die Kontonummer kann also zunächst nicht von 'außen' geändert werden.

```
long CStudent::GetBankAccount()
{
    return bank_account;
}
```

```
void CStudent::SetBankAccount(long new_bank_account)
{
    bank_account = new_bank_account;
}
```

Natürlich lässt sich auch eine Funktion schreiben, mit der man private Daten von außen ändern kann: `SetBankAccount(long)`. Letztlich entscheidet der Programmierer, ob dieser Zugriff zulässig sein soll, wenn nicht gewünscht, werden die Get und Set Zugriffsfunktionen einfach nicht bereitgestellt.

Die nachfolgende Übung zeigt zusammenfassend den Zugriff auf private Variablen von Außen.

Exercise E4.6:

```
int main()
{
    CStudent* m_std = new CStudent(); // instance using constructor
    cout << "E46: Data security - using private data" << endl;
    cout << "m_std->GetBankAccount()\t : " << m_std->GetBankAccount() << endl;
    m_std->SetBankAccount(987654321); // changing private data
    cout << "m_std->GetBankAccount()\t : " << m_std->GetBankAccount() << endl;
    delete(m_std);
    return 0;
}
```

## 4.7 Testfragen

1. Geben sie eine Definition von C++ Klassen mit eigenen Worten (max 5 Sätze).
2. Was ist ein benutzerdefinierter Datentyp ?
3. Welches Datennschutz-Konzept gibt es für Klassen ?
4. Wozu brauchen wir zwei Dateien für Klassen, eine H (Header) Datei und eine CPP (Quelltext) Datei ?
5. Was ist ein Inklude / Include ?
6. Was ist eine Instanz einer Klasse ?
7. Worin besteht der Unterschied zwischen den Anweisungen: `CStudent m_std_1` und `CStudent* m_std_2` ?
8. Was ist ein Konstruktor einer Klasse ?
9. Was ist das Gegenstück zum Klassen-Konstruktor ?
10. Wie können Daten / Variablen einer Klasse initialisiert werden ?
11. Schreiben sie den Quelltext für den Klassen-Konstruktor und weisen sie den Variablen `name_first` und `name_last` ihren eigenen Namen zu.
12. Was verbirgt sich hinter der Anweisung: `CStudent* m_std = new CStudent()` ?
13. Was ist der Unterschied zwischen `CStudent` und `CStudent()` ?
14. Wie kann ich meine Daten gegen einen externen Zugriff schützen ?

# Kapitel 5

## Strings

Wir haben schon in mehreren Übungen den Datentyp `string` benutzt, ohne diesen Datentyp etwas näher zu beleuchten. Dieses Versäumnis soll in diesem Kapitel nachgeholt werden.

### 5.1 Die Standardklasse `string`

`String` (in Deutsch Zeichenkette) ist vielmehr als nur ein Datentyp, `string` ist eine Standard-Klasse in C++. `String` ist eine der genialsten Weiterentwicklungen des C-Datentyps `char`, die das Hantieren mit Zeichen und Zeichenketten zu einem Kinderspiel macht, na sagen wir mal - uns das Programmierleben erheblich vereinfachen wird, wenn wir mit Zeichenketten operieren werden. So wird z.B. der erforderliche Speicherplatz für Zeichenketten automatisch reserviert und bei Veränderungen angepasst.

Wenn wir `strings` benutzen wollen, müssen den Header der `string`-Klasse wie folgt inkludieren.

```
#include <string> // for using strings
using namespace std; // for using standard names
```

Wichtig ist auch die zweite Zeile `using namespace std`. Diese Anweisung besagt, dass wir im folgenden Standard-Namen benutzen. Sonst müssten wir vor jeder `string`-Operation noch den Zusatz `std::` setzen. Zur Benutzung von `namespace` für die Strukturierung und Kapselung von Programmteilen gibt es später ein Extrakapitel (9.2.2). An dieser Stelle zeigen wir die Bedeutung von `using namespace std`; in bewährter Weise an einem Beispiel.

Sie haben sicher schon bemerkt, dass wir bei `#include` manchmal eckige Klammern `<>` und manchmal Gänsefüßchen `”` benutzen. Wir schauen uns dies, wie

gesagt, genauer im Kapitel 9.2.2 an. Hier nur soviel, dass mit eckigen Klammern `<>` an Header-Dateien im Systemverzeichnis von C++ gesucht wird, während `"` benutzerdefinierte, eigene Header (z.B. `student.h`) kennzeichnet und somit im aktuellen Arbeitsverzeichnis (also `./`) nachgeschaut wird.

## 5.2 Operationen mit strings

Die nachfolgende Tabelle zeigt eine Übersicht der wichtigsten string Operationen, die wir in diesem Kapitel verwenden.

| Methode                 | Erläuterung   |
|-------------------------|---|
| <code>.append()</code>  | verlängert den String   |
| <code>.c_str()</code>   | erzeugt Zeichenfeld mit dem Inhalt des Strings                              |
| <code>.clear()</code>   | löscht den Inhalt des Strings   |
| <code>.compare()</code> | vergleicht Strings  |
| <code>.erase()</code>   | löscht Zeichen im String  |
| <code>.find()</code>    | sucht Zeichen in einem String   |
| <code>.insert()</code>  | fügt in den String ein  |
| <code>.length()</code>  | ermittelt die Länge des Strings   |
| <code>.replace()</code> | ersetzt Zeichen im String   |
|                         | wichtig für die Konvertierung von <code>string</code> zu <code>char*</code> |
| <code>.resize()</code>  | ändert Länge des Strings  |
| <code>.substr()</code>  | gibt einen Substring zurück   |
| <code>&gt;&gt;</code>   | Eingabeoperator für Strings   |
| <code>&lt;&lt;</code>   | Ausgabeoperator für Strings   |
| <code>getline()</code>  | liest Zeichen aus der Eingabe   |

Tabelle 5.1: string Methoden

### 5.2.1 Initialisieren von strings

Eine Möglichkeit für die Initialisierung von strings haben wir uns bereits in der Exercise E4.5 angesehen bei der Verwendung von Klassen-Konstruktoren. Der Standard-Konstruktor `string()` erzeugt einen leeren String. Eine zweite Möglichkeit besteht direkt bei der Deklaration des strings, wie folgt:

Exercise E5.2.1:

```
string exercise("Exercise: string initialisation");
cout << exercise.length() << endl;
```

## 5.2.2 Zuweisen von strings

In der folgenden Übung schauen wir uns an, wie wir mittels Tastatureingabe (Standard-Eingabe-Gerät) strings zuweisen und Teile von strings in andere kopieren zu können.

Exercise E5.2.2:

```
#include ... // Bitte fügen sie die notwendigen Header selbst ein
main()
{...
string eingabe;
string eingabe_anfang;
string message("Bitte geben Sie eine Zeile mit der Tastatur ein.
               Schliessen Sie die Eingabe mit Enter ab");
//-----
cout << message << endl; // Ausgabe der Eingabeaufforderung
getline(cin, eingabe); // Eingabe einer Zeile über Tastatur
eingabe_anfang(eingabe, 0, 10); // die ersten 10 Zeichen von eingabe werden
                               nach eingabe_anfang kopiert
cout << "Ihr Eingabetext: " << eingabe << endl;
cout << "Die ersten 10 Zeichen des Eingabetextes: " << eingabe_anfang << endl;
...}
```

## 5.2.3 Verketteten von strings

Mit dem Operator + können Strings miteinander verknüpft werden und in einem neuen `string name` kopiert. In der folgenden Übung produzieren wird aus Vor- und Nachnamen den ganzen Namen.

Exercise E5.2.3:

```
string name;
CStudent* m_std = new CStudent();
name = m_std->name_first + m_std->name_last;
// oder
name = m_std->name_first;
name += m_std->name_last;
```

Wie bekommen wir einen Zwischenraum (Leerzeichen) zwischen Vor- und Nachnamen ?

## 5.2.4 Vergleichen von strings

Oft sind Abfragen notwendig, ob gewisse Zeichenketten gefunden wurden, um dann gewisse Operationen durchzuführen. Hierfür bietet die String-Klasse mehrere Möglichkeiten an, z.B. den exakten Vergleich (`string::compare`) oder einen

Teil von Zeichenketten (`string::find`). Die nachfolgende Übung zeigt, wenn der Nachname BOND gefunden wurde, dann wird der Vorname auf JAMES gesetzt (und natürlich der russische Geheimdienst informiert).

Exercise E5.2.4:

```
if(m_std->name_last.compare("BOND")==0)
{
    m_std->name_first = "JAMES";
}
```

Die nachfolgende Übung zeigt, wie ein Programm beendet werden kann, wenn eine bestimmte Taste gedrückt wird. Hierfür werden einzelne Zeichen mit dem Operator `==` verglichen.

```
string Taste("N");
while(Taste == "J")
{
    cout << "Soll dieses Programm endlich beendet werden? (J/N)" << endl;
    getline(cin,Taste);
}
cout << "Programm-Ende" << endl;
```

### 5.2.5 Suchen in strings

Diese Übung zeigt ihnen, wie nach Zeichenketten in string suchen können. Sie sehen, je nach Sorgfalt des Programmierers haben sie eventuell schlechte Karten, wenn ihr Vorname die Zeichenkette 'BON' enthält.

Exercise E5.2.5:

```
if(m_std->name_last.find("BON")!=string::npos)
{
    m_std->name_first = "JAMES";
}
```

### 5.2.6 Einfügen in strings

Nun benutzen wir die Einfüge-Funktion von strings (`string::insert`), um Vor- und Nachnamen zusammensetzen. Dabei ermitteln wir zunächst die Länge des Vornamen mit `string::length`, setzen dann den Positionszähler `pos` um Eins hoch (Leerzeichen zwischen Vor- und Nachnamen) und fügen dann den Nachnamen mit `string::insert` ein.

Exercise E5.2.6:

```

string name;
int pos;
if(m_std->name_first.find("JAMES")!=string::npos)
{
    pos = m_std->name_first.length();
    name.insert(pos+1,"BOND");
}

```

### 5.2.7 Ersetzen in strings

Eine weitere nützliche String-Funktion ist das Ersetzen von Zeichen. In der nachfolgenden Übung ändern wir den Nachnamen. Dazu wird zunächst wieder die Länge des Vornamens mit `string::length` ermittelt und dann der neue Nachname eingefügt. So können sie ihre Spuren verwischen ... ist auch praktisch bei Namensänderungen z.B. infolge Heiraten (Beachten sie, dass Triple-Namen wie Müller-Graf-Kleditsch nicht mehr zulässig sind).

Exercise E5.2.7:

```

string name;
int pos;
name = "JAMES" + " " + "CHRISTIE"
if(m_std->name_first.find("JAMES")!=string::npos)
{
    pos = m_std->name_first.length();
    name.replace(pos+1,"BOND");
}

```

Was passiert, wenn der neue Nachname länger ist als der alte ?

### 5.2.8 Löschen in strings

Natürlich können auch Zeichen in einem string gelöscht werden. Diese Funktion passt den Speicherbedarf des gekürzten Strings automatisch an.

Exercise E5.2.8:

```

string name;
int pos;
name = "JAMES" + " " + "CHRISTIE"
if(m_std->name_first.find("JAMES")!=string::npos)
{
    pos = m_std->name_first.length();
    name.erase(pos);
    name.replace(pos+1,"BOND");
}

```

Abschliessend kommen wir zu zwei weiteren Punkten, (1) das Umwandeln von C++ strings in C char und (2) das Auslesen von string Teilen.



## 5.2.9 Umwandeln von strings in char

Manchmal ist es notwendig C++ strings in C char umzuwandeln (wir benötigen dies später, wenn wir mit der MFC Klasse CStrings für grafische Benutzeroberflächen arbeiten). Die String-Methoden `c_str()` und `data()` wandeln strings in char um. Aufgepasst, ab char müssen wir uns selber um das Speichermanagement kümmern.

```
fprintf(f, " %s\n", name.c_str());
name.clear();
const char *char_string;
char_string = name.data();
```

## 5.2.10 Auswerten von Strings: Stringstreams

Stringstreams ... lässt sich kaum aussprechen .. es handelt sich aber definitiv nicht um stringstreams ... (sie erinnern sich an die Eingangsfolie der Vorlesung). Stringstreams sind eine sehr nützliche Sache, damit lassen sich Eingabedaten (von Tastatur und Datei) bequem als Stream auswerten. Um Stringstreams nutzen zu können, muss die Klasse `sstream` inkludiert werden. Die Übung zeigt, wie man eine eingegebene Zeile (Vor- und Nachname) elegant zerlegen kann. Dabei wird die eingegebene Zeile zunächst in den `stringstream` kopiert, danach wird `input_line` wie ein normaler stream ausgelesen.

Exercise E5.2.10:

```
#include <iostream>
#include <string>
#include <sstream>
using namespace std;
int main()
{
    string name;
    string name_first;
    string name_last;
    stringstream input_line;
    cout << "Geben Sie bitte Ihren Namen (Vor- und Nachnamen) ein: ";
    getline(cin, name);
    input_line.str(name);
    // Der Name wird nun zerlegt
    input_line >> name_first;
    cout << "Vorname:\t" << name_first << endl;
    input_line >> name_last;
    cout << "Nachname:\t" << name_last << endl;
    input_line.clear();
    return 0;
}
```

## 5.3 Testfragen

1. Welche Klasse bietet uns C++ zur Verarbeitung von Zeichenketten an ?
2. Welchen Include benötigen wir für die Arbeit mit Zeichenketten ?
3. Wofür ist die Anweisung `using namespace std` nützlich ?
4. Müssen wir selber Speicherplatz für C++ Zeichenketten (Strings) reservieren ?
5. Wie können wir einen String, sagen wir mit der Zeichenkette "Das ist eine gute Frage", initialisieren ?
6. Wie können wir zwei Strings, ihren Vor- und Nachnahmen, miteinander verbinden ?
7. Mit welcher string Funktion kann ich Zeichenketten miteinander vergleichen ?
8. Schreiben sie eine Anweisung zum Vergleich der Klassen-Variable `m_std->name_last` mit ihrem Nachnamen ?
9. Mit welcher string Funktion kann ich Zeichenketten suchen ?
10. Schreiben sie eine Anweisung zum Suchen ihres Vornamens in der Klassen-Variable `m_std->name_first` ?
11. Mit welcher string Funktion kann ich Teile in Zeichenketten erstetzen ?
12. Wie können wir die Länge von Strings ermitteln ?
13. Schreiben sie die Anweisungen, um ihren Nachnamen in die Zeichenkette "JAMES BOND" nach "JAMES" einzufügen ?
14. Was passiert, wenn ihr Nachname länger als "BOND" ist ?
15. Mit welcher string Funktion können wir Zeichen in einem String löschen ?
16. Wir haben gesehen, dass es verschiedene Daten-Typen für Zeichenketten in C, C++, MFC, .NET und Qt gibt. Zeichenketten gehören zu den wichtigsten Daten-Typen bei der Programmierung. Wie können wir einen C++ string in eine C Zeichenkette (char) umwandeln ?
17. Können wir eine .NET Zeichenkette (String $\wedge$ ) in eine C++ Zeichenkette (string) umwandeln ?
18. Was ist ein `stringstream` ?
19. Welchen Include benötigen wir für die Arbeit mit Stringstreams ?

# Kapitel 6

## Ein- und Ausgabe - II

Nachdem wir uns im Kapitel 3 bereits mit der Ein- und Ausgabe über die Standard-Geräte (Tastatur und Bildschirm) beschäftigt haben, geht es in diesem Teil um die Dateiverarbeitung.

### 6.1 Die fstream Klassen

Abb. 6.1 zeigt die Hierarchie der fstream Klassen.

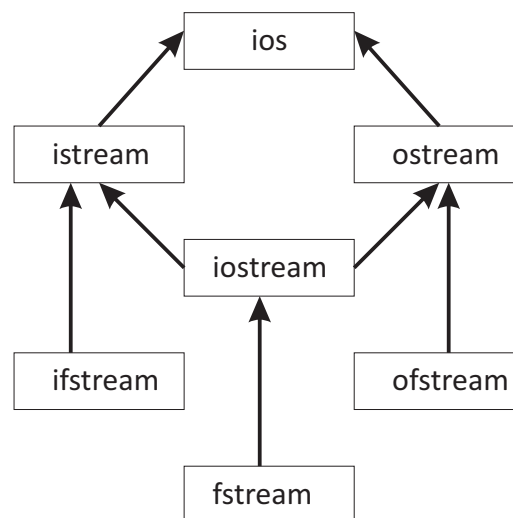


Abbildung 6.1: Die fstream Klassen

Diese sind von den bereits bekannten ios stream Klassen abgeleitet.

- `ifstream`: ist von `istream` abgeleitet für das Lesen von Dateien.
- `ofstream`: ist von `ostream` abgeleitet für das Schreiben von Dateien.
- `fstream`: ist von `iostream` abgeleitet für das Lesen und Schreiben von Dateien.

## 6.2 Arbeiten mit File-Streams

### 6.2.1 File-Streams anlegen

Eröffnungsmodus: Um eine Datei benutzen zu können, muss diese zunächst geöffnet werden. Dafür gibt es verschiedene Möglichkeiten (siehe Tab. 6.1).

| Flag                     | Funktion  |
|--------------------------|---|
| <code>ios::in</code>     | Eine (existierende) Datei wird zum Lesen geöffnet.                                    |
| <code>ios::out</code>    | Eine Datei wird zum Schreiben geöffnet.<br>Existierende Inhalte werden überschrieben. |
| <code>ios::app</code>    | Die neuen Inhalte werden an die existierenden angehängt.                              |
| <code>ios::trunc</code>  | Eine bestehende Datei wird beim Öffnen auf die Länge 0 gekürzt.                       |
| <code>ios::ate</code>    | Schreib- und Leseoperation werden auf das Dateiende gesetzt.                          |
| <code>ios::binary</code> | Schreib- und Leseoperationen werden im Binärmodus ausgeführt.                         |

Tabelle 6.1: Eröffnungsmodi für Dateien

Die default Werte sind:

- `ios::in` für `ifstream`
- `ios::out` | `ios::trunc` für `ofstream`

### 6.2.2 File-Streams schließen

Wir wissen schon, dass es bei objekt-orientierten Sprachen immer zwei passende Dinge gibt, z.B. Klassen-Konstruktoren und -Destruktoren. So ist zu erwarten, dass es zu einer Methode 'Datei öffnen' (`open()`) auch eine Methode 'Datei schließen' gibt (`close()`) (Tab. 6.4)

### 6.2.3 Übung: Eine einfache Kopierfunktion

In unserer ersten Übung zur Dateiverarbeitung schreiben wir eine einfache Kopierfunktion.

Exercise E6.2.3: Eine einfache file copy Funktion

```

#include <iostream> // for using cout
#include <fstream> // for using ifstream / ofstream
#include <string> // for using string
using namespace std; // namespace for std functions

int main()
{
    //-----
    ifstream input_file; // Instance of class ifstream
    input_file.open("input_file.txt"); // Open file "text_file.txt"
    string my_string; // Instance of class string
    input_file >> my_string; // Reading a string from file
    cout << my_string.data() << endl; // Output of string to screen
    //-----
    ofstream output_file; // Instance of class ifstream
    output_file.open("output_file.txt"); // Open file "text_file.txt"
    output_file << my_string; // Writing a string to a file
    //-----
    return 0;
}

```

Die Ein- >> und Ausgabeoperatoren << formatieren die Datentypen (z.B. int in der Übung E6.2.3) entsprechend den Einstellungen der fstream Klasse. Diese Einstellungen können durch Flags verändert werden (siehe nächsten Abschnitt).

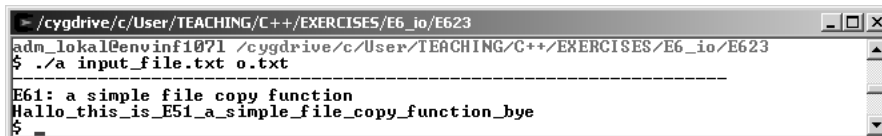
Die main() Funktion kann auch mit einer Parameterliste (int argc, char \*argv[]) versehen werden. Die Anzahl der Parameter (argc) wird automatisch erkannt. Mit jedem Leerzeichen in der Tastatureingabe entsteht ein neuer Eingabeparameter (Abb. 6.2).

```

int main(int argc, char *argv[])
{
    ifstream input_file; // Instance of class ifstream
    input_file.open(argv[1]); // Open file, name from cin
    ofstream output_file; // Instance of class ifstream
    output_file.open(argv[2]); // Open file, name from cin
    return 0;
}

```

Die Benutzung der main Funktion mit Eingabeparametern ist in der folgenden Abbildung zu sehen.



```

/cygdrive/c/User/TEACHING/C++/EXERCISES/E6_io/E623
adm_lokal@envinf1071 /cygdrive/c/User/TEACHING/C++/EXERCISES/E6_io/E623
$ ./a input_file.txt o.txt
-----
E61: a simple file copy function
Hallo_this_is_E51_a_simple_file_copy_function_bye
$

```

Abbildung 6.2: Die main Funktion mit Parametern

## 6.2.4 Übung: Ein einfacher Konverter

Ihre Frage nach dem Sinn der Übung 6.2.3 ist vollkommen berechtigt, wozu ein Programm schreiben, um eine Datei zu kopieren. Das kann ich doch auch mit dem Windows-Explorer oder mit `cp file1 file2` machen. Richtig, aber genauso funktionieren Kopierprogramme, Windows-Explorer ruft 'unser' Kopierprogramm auf. Wenn wir auch nur kleine Änderungen in unserer Datei vornehmen wollen (z.B. eine laufende Nummer in jede Zeile einfügen), kann uns der Windows-Explorer nicht mehr weiter helfen. Dies ist insbesondere dann ärgerlich, wenn die Datei ziemlich groß ist ... Auch hier sagen sie zu Recht, eine Nummerierung in eine größere Datei einfügen, das kann ich z.B. auch mit EXCEL machen. In der nächsten Übung schreiben wir einen kleinen Konverter, also genau was EXCEL im Hintergrund macht, wenn wir eine neue Spalte einfügen.

Exercise E6.2.4: Ein einfacher Konverter

```
int main()
{
    //-----
    ifstream input_file;           // Instance of class ifstream
    input_file.open("input.txt");   // Open file "text_file.txt"
    ofstream output_file;         // Instance of class ifstream
    output_file.open("output.txt"); // Open file "text_file.txt"
    //-----
    char line[80];
    int i=0;
    while(input_file.getline(line,80)) // Loop condition
    {
        output_file << i << " " << line << endl;
        i++;                          // Incrementor (+1)
    }
    //-----
    return 0;
}
```

Unser Konverter macht nichts anderes, als die Datei `input.txt` zu öffnen, nacheinander alle Zeilen lesen und am Anfang der Zeile eine Nummer (`i`) einzufügen und dann in die neue Datei `output.txt` zu schreiben.

Was ist neu bei dieser Übung.

| C++ Ding             | Was tut's   |
|----------------------|---|
| <code>while()</code> | eine Kontrollstruktur für Schleifen (solange der Ausdruck in () wahr (true) ist wird die Schleife ausgeführt) |
| <code>i++</code>     | der Inkremetor (zählt Eins hoch)  |

Tabelle 6.2: C++ news

Alle benutzten fstream Methoden finden sie in Tab. 6.4.

### 6.2.5 Einschub: Erstes Arbeiten mit MS VC++

Wir haben den Editor von MS VC++ ja schon mehrfach benutzt, da die Syntax des Quelltextes sehr schön angezeigt wird (Abb. 6.3). Überhaupt ist MS VC++ ein sehr schönes Entwicklungssystem für Programmentwicklung ... Wir führen die Übung E6.2.4 jetzt mal mit MS VC++ aus.

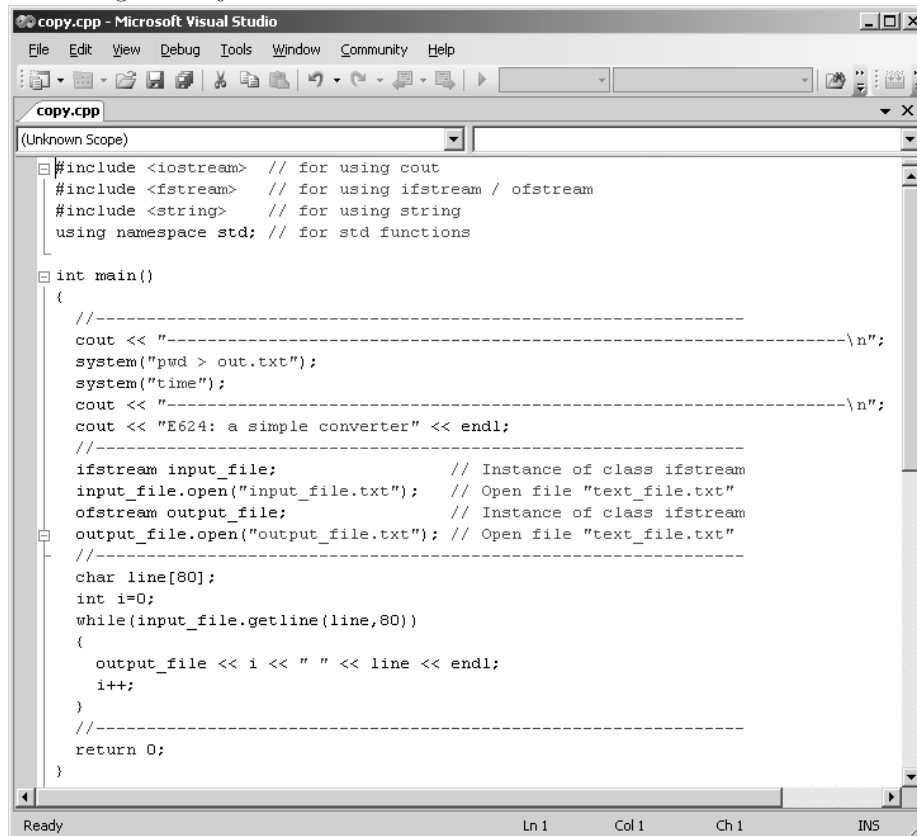


Abbildung 6.3: Der MS VC++ Editor

Standardmäßig werden folgende Einfärbungen benutzt für

- Keywords: Blau
- Kommentare: Grün
- Normaler Quelltext: Schwarz
- Includes: Rot

- Zeichenketten: Rot

Ich bin mir ziemlich sicher, dass man in einem gut versteckten MS VC++ Menüpunkt alles selber einstellen kann ...

Highlighting of syntax ... ist nur eines der sehr nützlichen features (langsam gleiten wir in IT slang ab). Um MS VC++ richtig nutzen zu können, müssen wir ein sogenanntes Projekt anlegen (mal sehen, ob ihre Begeisterung danach noch anhält). Die Schritte für die Projekterstellung finden sie in der Anlage (Abschn. ??) ... Nachdem das Projekt angelegt ist, können wir damit arbeiten.

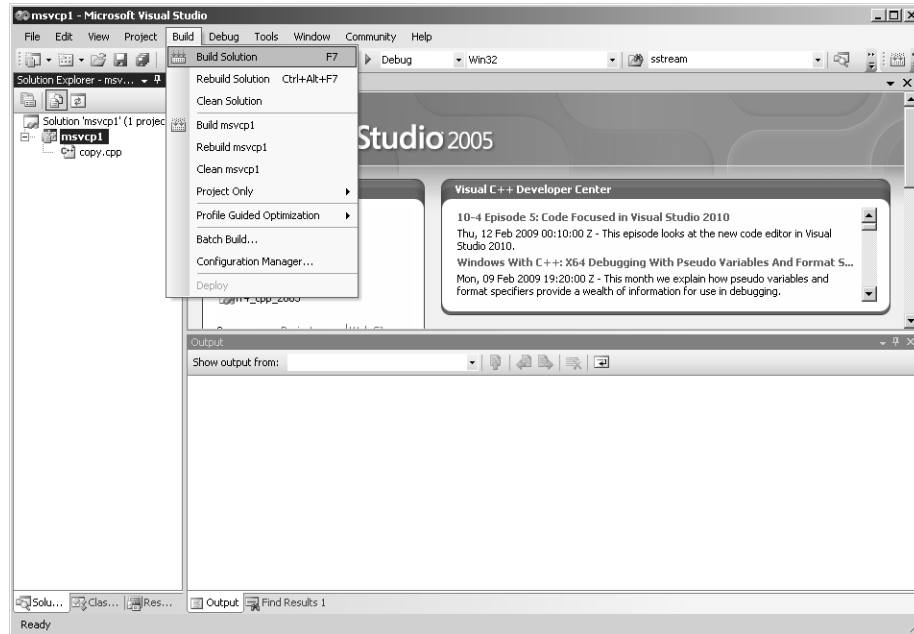


Abbildung 6.4: Übersetzen sie das Projekt mit F7 (build solution)

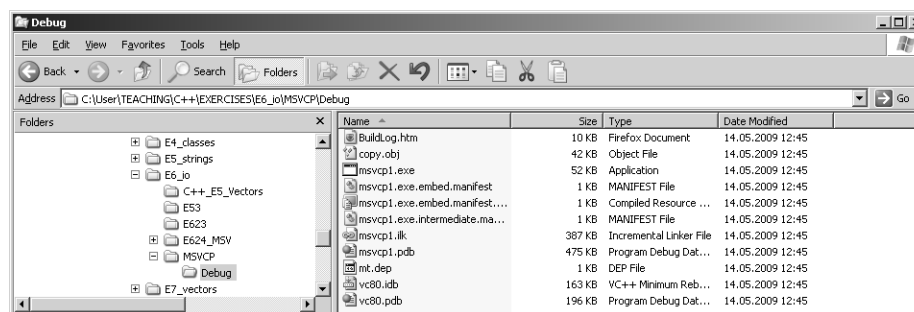


Abbildung 6.5: Das Debug Verzeichnis



MSVS legt allerhand Zeug's an. Das executable finden sie im Unterverzeichnis Debug und kann natürlich mit einem Doppelklick gestartet werden. Wir starten die Konsolenanwendung aus MSVS mit Ctrl-F5 (Abb. 6.6).

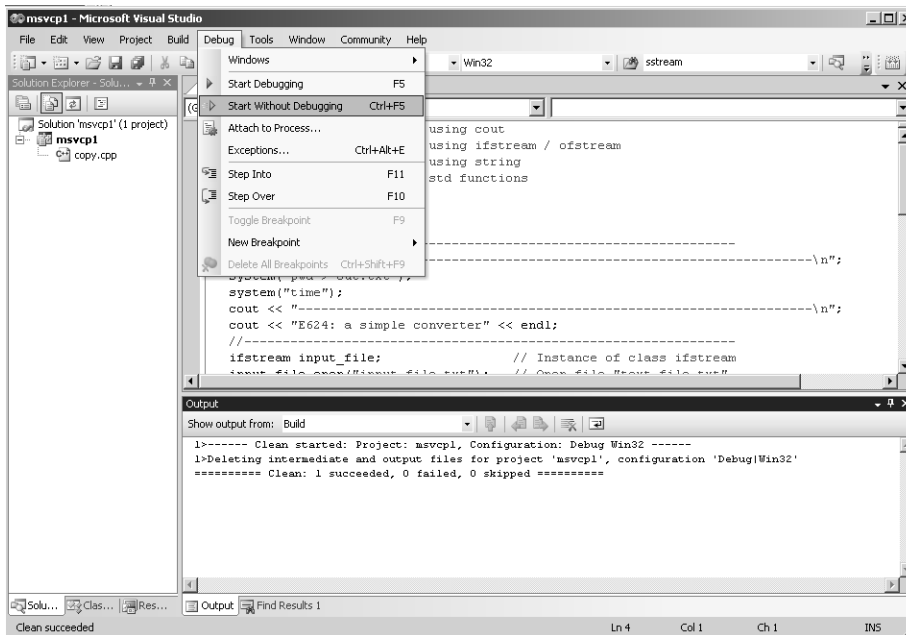


Abbildung 6.6: Starten der Konsolenanwendung mit Ctrl-F5 oder Menü-Auswahl (Start without debugging)

Hier wartet schon die erste Überraschung auf uns: `pwd` wird nicht erkannt und wir werden aufgefordert eine Zeit einzugeben (Abb. 6.7).

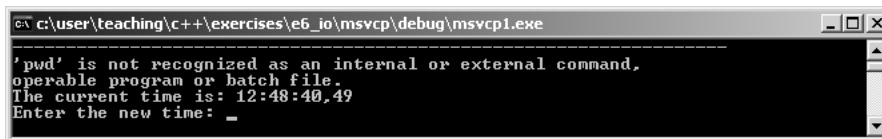


Abbildung 6.7: Das Debug Verzeichnis

Was ist passiert ?

- `pwd` ist ein Linux-Befehl, den kennt der Windows-Compiler nicht.
- `time` gibt es auch als DOS-Befehl, hat aber eine ganz andere Bedeutung: nicht Ausgabe der Zeit sondern Zeit ändern.

Wir sehen also, dass unser Quellcode von verschiedenen Compilern unterschiedlich interpretiert wird.

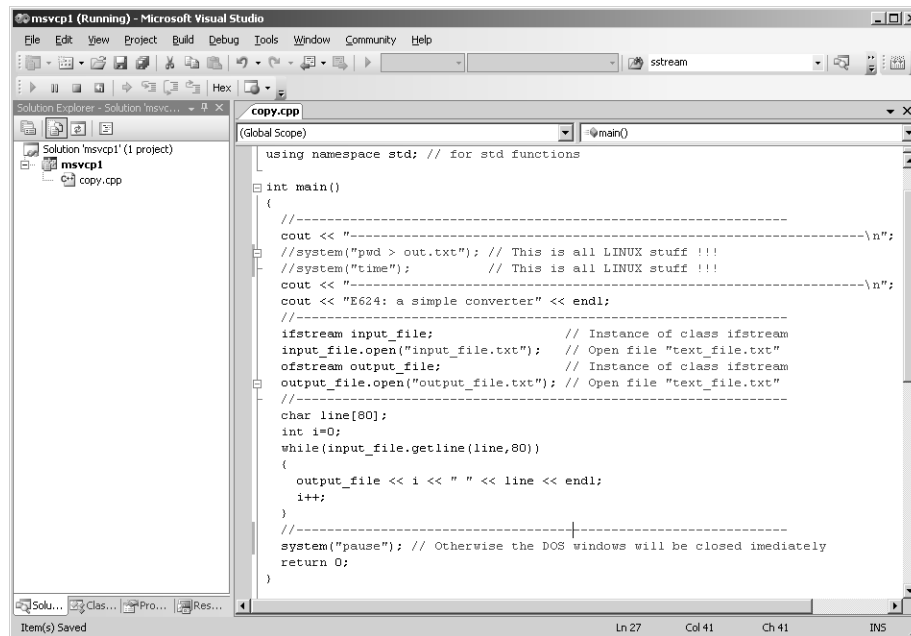


Abbildung 6.8: Der Windows bereinigte Quelltext

Sie sehen 'Schönheit' hat auch ihren Preis ... dennoch die Funktionalität der MS VC++ Entwicklungsumgebung, das muss man neidlos anerkennen, ist einfach Klasse.

### 6.3 File-Streams und Klassen

Wir wollen eine Lesefunktion für die Klasse `CStudent` schreiben. Bevor wir damit beginnen, müssen wir uns Gedanken über eine geeignete Struktur für die Darstellung eines `CStudent` Datensatzes in einer Datei machen. Der Vorschlag für die Strukturierung von Datensätzen ist die Benutzung von Schlüsselwörtern zur Trennung von Datenblöcken, z.B.

```

#STUDENT
$NAME_FIRST
    James
$NAME_LAST
    Bond
...
#STOP

```

Wir benutzen zwei verschiedene Symbole für Schlüsselwörter:

- keyword # : zur Trennung von Datensätzen für eine Instanz von CStudent,
- subkeyword \$ : zur Identifizierung der einzelnen Daten für die CStudent Instanz.
- #STOP zeigt das Ende der Datenbasis an. (Eigentlich wäre dies nicht nötig, da das Dateende auch mit dem Parameter eof (end-of-file) abgefragt werden kann. Wir werden aber sehen, dass mit #STOP einiges einfacher zu programmieren ist.)

### Exercise E6.3: Implementierung der CStudent Lesefunktion

```

ios::pos_type CStudent::Read(istream& input_file)
{
    //-----
    string input_line;
    char buffer[256]; // MAX_LINE
    ios::pos_type position;
    //-----
    while(true)
    {
        position = input_file.tellg();
        input_file.getline(buffer,256);
        input_line = buffer;
        if(input_line.size()<1) // empty line
            continue;
        // Dealing with keywords
        if(input_line.find('#')!=string::npos) // keyword found
            return position;
        // Dealing with subkeywords
        if(input_line.find("$NAME_FIRST")!=string::npos)
        {
            input_file.getline(buffer,256);
            input_line = buffer;
            name_first = input_line;
        }
        if(input_line.find("$NAME_LAST")!=string::npos)
        {
            input_file >> name_last;
        }
        if(input_line.find("$subkeyword")!=string::npos)
        {
            input_file >> member;
        }
    }
    //-----
    return position;
}

```

Der Lesealgorithmus funktioniert wie folgt: Nach dem Auffinden eines Schlüsselworts `#STUDENT`, wird in einer `while` Schleife nach subkeywords (\$) gesucht, wenn gefunden, werden der `CStudent` Instanz die Werte aus der Eingabedatei zugewiesen (also die Namen). Wenn das nächste Schlüsselwort (`#STUDENT` oder `#STOP`) gefunden wird, geht's mit der vorletzten Position raus aus der Funktion. Die vorletzte `position` ist wichtig, damit die letzte Zeile später nochmal gelesen werden kann, um das Schlüsselwort auswerten zu können ...

Die Implementierung der `main` Funktion, in der die `CStudent::Read()` Funktion aufgerufen wird, ist im folgenden Block dargestellt.

```
#include <iostream> // for using cout
#include <fstream> // for using ifstream / ofstream
#include <string> // for using string
#include "student.h" // for using CStudents
using namespace std; // for std functions

int main()
{
    //-----
    // File handling
    ifstream input_file; // ifstream instance
    input_file.open("data_set.txt");
    if(!input_file.good()) // Check is file existing
    {
        cout << "! Error in STD::Read: file could not be opened" << endl;
        return 0;
    }
    input_file.seekg(OL,ios::beg); // Rewind file
    //-----
    CStudent* m_std = new CStudent(); // CStudent instance
    m_std->Read(input_file);
    //-----
    input_file.close();
    return 0;
}
```

Die `main` Funktion besteht aus zwei Teilen, dem File-Handling und dem Aufruf der Lesefunktion. Beim File-Handling wird der stream aus der Datei `data_set.txt` geöffnet, anschließend erfolgt der Test, ob das File erfolgreich geöffnet werden konnte; wenn nicht, wird die `main` Funktion sofort beendet.

Was ist neu bei dieser Übung.

Wir benutzen den Referenz-Operator `&` (Kapitel 7) als Parameter der Funktion `CStudent::Read()`. Eine Referenz ist eigentlich nicht anderes als ein anderer Bezeichner (Alias) für eine bereits existierende Instanz eines Objekts. Es kann also mehrere solcher Referenzen geben und vor Benutzung einer Referenz muss die Instanz des Objekts physikalisch (also speichermäßig) vorhanden sein, sonst 'crashed' unser Programm.

|                                       |  |
|---------------------------------------|--|
| C++ Ding                              | Was tut's  |
| <code>ifstream&amp; input_file</code> | eine Reference auf ein Objekt<br>wird in Kapitel 7 ausführlich abgehandelt |

Tabelle 6.3: C++ news

## 6.4 fstream Methoden

Die bisher benutzten fstream Methoden sind in der Tab. 6.4 zusammengestellt.

| Methode                           | Funktion  |
|-----------------------------------|---|
| <code>open()</code>               | öffnet die Datei  |
| <code>good()</code>               | testet erfolgreiche Öffnung der Datei   |
| <code>seekg(pos, ios::beg)</code> | geht zur Position pos in der Datei  |
| <code>seekg(OL, ios::beg)</code>  | spoolt zum Dateianfang zurück   |
| <code>tellg()</code>              | merkt sich die aktuelle Position im stream  |
| <code>getline(buffer, 256)</code> | holt eine Zeile der Länge 256 (Zeichen) aus dem stream<br>und kopiert diese in buffer |
| <code>close()</code>              | schließt Datei  |
| <code>&gt;&gt;</code>             | Eingabeoperator für Dateien   |
| <code>&lt;&lt;</code>             | Ausgabeoperator für Dateien   |

Tabelle 6.4: fstream Methoden

Die string Auswertung spielt bei der Lesefunktion eine wichtige Rolle, daher haben wir uns wir uns im vorangegangenen Kapitel mit der string Klasse beschäftigt.

## 6.5 Testfragen

1. Was ist die Basis-Klasse für alle Ein- und Ausgaben in C++ ?
2. Was sind die C++ Klassen für das Lesen und Schreiben von Dateien ?
3. Welchen Include benötigen wir für das Arbeiten mit I/O File-Klassen ?
4. Was sind die Standard-Flags für File-Streams (Lesen und Schreiben) ?
5. Mit welchem Flag können wir zu schreibende Daten an eine existierende Datei anhängen ?
6. Was ist der Unterschied zwischen ASCII- und Binär-Formaten ?
7. Mit welchem Flag können wir Daten in einem Binär-Format schreiben ? Mit welcher Anweisung wird ein File geöffnet ? Mit welcher Anweisung wird ein File geschlossen ?
8. Was bewirken die Stream-Operatoren `<<` und `>>` ?
9. Wie können wir mit Dateinamen in unserem Hauptprogramm `main(...)` arbeiten ?
10. Welche Anweisung benötigen wir für die Erzeugung einer Instanz für einen Eingabe-Strom ?
11. Welche Anweisung benötigen wir für die Erzeugung einer Instanz für einen Ausgabe-Strom ?
12. Für die Erstellung einer Datenbank ist es wichtig einzelnen Datensätze zu trennen. Wie können wir soetwas in der Datenbank-Datei bewerkstelligen ?
13. Ist es wichtig das Ende einer Datenbank-Datei, z.B. mit einem Schlüsselwort `#STOP`, zu markieren ?
14. Mit welcher Abfrage könne wir prüfen, ob die Öffnung einer Datei erfolgreich war ?
15. Mit welcher Anweisung können wir die aktuell gelesene Position in einer geöffneten Datei abfragen ?
16. Mit welcher Anweisung können wir zu einer bestimmten Position in einer geöffneten Datei springen ?
17. Mit welcher Anweisung können wir eine komplette Zeile aus geöffneten Datei auslesen ?

# Kapitel 7

## Referenzen und Zeiger

Referenzen und Zeiger (pointer), d.h. die Symbole `&` und `*` sind uns bereits mehrfach begegnet, z.B. in der Parameterliste der `main()` Funktion ... Generell können Zeiger und Referenzen als einfache Variable, Parameter oder Rückgabewert (return) einer Funktion auftauchen.

In der Übung E6.3 haben wir bereits mit Referenzen gearbeitet und festgestellt, dass Referenzen eigentlich nichts anderes als andere Bezeichner (Alias) für eine bereits existierende Instanz eines Objekts sind. Es kann also mehrere solcher Referenzen geben. Bei der Definition eines Zeigers wird also kein neuer Speicher reserviert. Die Instanz des Objekts muss physikalisch (also speichermäßig) natürlich vorhanden sein, sonst 'verabschiedet' sich unser Programm gleich wieder.

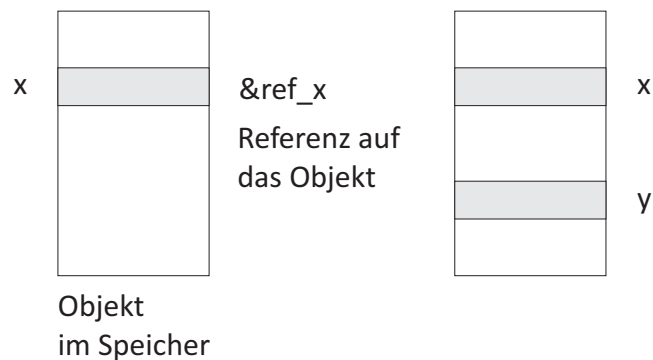


Abbildung 7.1: Referenzen auf Objekte

## 7.1 Referenzen

### 7.1.1 Das Call-by-reference Prinzip - Referenzen als Parameter

Def: Eine Referenz ist nichts anderes als ein anderer Name (Alias) für ein bereits existierendes Objekt. Bei der Definition einer Referenz wird also kein neuer Speicherplatz für das Objekt belegt (außer natürlich die 4 Byte für die Referenz selbst).

Die Abb 7.2 soll das Konzept des Referenzierens noch einmal verdeutlichen.

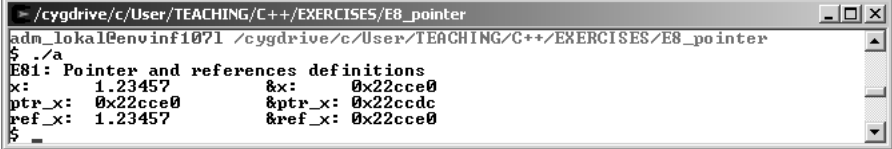
- `x` ist die Instanz eines Datentyps `T`
- `&ref_x` ist eine Referenz (Verweis) auf `x`

In der ersten Übung sehen die Definition von Referenzen und Zeigern im Quelltext

Übung: E7.1.1

```
double x = 1.23456789;
double* ptr_x;
double& ref_x = x; // initialisation is needed
```

Die Bildschirmausgabe der ersten Übung zeigt Folgendes:



```

/cygdrive/c/User/TEACHING/C++/EXERCISES/E8_pointer
adm_lokal@envinf1071 /cygdrive/c/User/TEACHING/C++/EXERCISES/E8_pointer
$ ./a
E81: Pointer and references definitions
x:      1.23457          &x:      0x22cce0
ptr_x:  0x22cce0       &ptr_x:  0x22ccd0
ref_x:  1.23457       &ref_x:  0x22cce0
$ =
```

Abbildung 7.2: Referenzen auf Objekte

Eine Referenz (`ref_x`) muss bei der Definition initialisiert werden (mit `x`), sonst bekommen sie eine Fehlermeldung beim Kompilieren.

Der Referenztyp-Parameter (`&`) ist ein Aliasname für das 'richtige' Argument. Beim Aufruf der Funktion wird der Referenz-Parameter mit dem Objekt initialisiert. Die Funktion arbeitet also direkt mit dem von außen übergebenen Argument. Das 'Call-by-reference' Prinzip ist sozusagen eine Übergabe-von-Außen. Wir schauen uns dies mal an bei der `CStudent-Lesefunktion` aus der letzten Vorlesung.

Übung: E7.1.1



Definition:

```
CStudent::Read(istream& ref_input_file)
```

Aufruf:

```
istream input_file;
CStudent* m_std = new CStudent();    // CStudent instance
m_std->Read(input_file);
```

Was passiert ? Beim Aufruf der Lese-Funktion `Read` wird die Adresse des `istream` Objekts (`input_file`) an die Funktion übergeben. Das heisst die Lese-Funktion arbeitet intern mit dem 'richtigen' Objekt. 'Call-by-reference' Prinzip ermöglicht es also der Funktion einen Zugriff auf Daten des Aufrufers.

Beim sogenannten 'Call-by-value' Prinzip können Funktions-Argumente innerhalb der Funktion nicht geändert werden.

### 7.1.2 Referenzen als Rückgabe-Wert

Referenzen auf Objekte können auch Rückgabe-Werte von Funktionen sein.

Übung: E7.1.2 (aus [5])

Definition:

```
string& message()
{
    static string m_str("Vorsicht Speicher-Falle");
    return m_str;
}
```

Aufruf:

!!! Das Objekt muss nach dem Verlassen der Funktion noch existieren

Was ist neu bei dieser Übung.

| C++ Ding            | Was tut's   |
|---------------------|---|
| <code>static</code> | Speicherklasse: der Speicher für das Objekt bleibt während der gesamten Programmausführung bestehen, wird in Kapitel 7.3 ausführlich abgehandelt. |

Tabelle 7.1: C++ news

## 7.2 Zeiger

Zeiger sind eine der Grundideen moderner Programmiersprachen. Dabei arbeitet man nicht direkt mit den Daten-Blöcken im Speicher sondern mit deren Adressen. Dies macht insbesondere Sinn, wenn es um große Daten-Objekte geht

oder bei Objekten, die erst zur Laufzeit angelegt werden (z.B. Vektoren, Listen, Strings). Das Klassen-Prinzip unterstützt die Zeigertechnik optimal, da die Klasse ihre Daten ja selber verwaltet (siehe Klassenkonstruktor). Der Zeiger (pointer) auf ein Objekt repräsentiert dessen Adresse und Daten-Typ.

### 7.2.1 Definition von Zeigern

Die nachfolgende Übung zeigt, wie wir Zeiger definieren und mit ihnen arbeiten können. Für den Zugriff auf Zeiger gibt es den sogenannten Verweisoperator '\*'.  
Übung: E7.2 (aus [5])

```
int main()
{
    int i, *ptr_i; // Definitionen
    i = 100;      // Zuweisung
    ptr_i = &i;  // Dem Zeiger wird die Adresse der integer Variable i
                // zugewiesen
    *ptr_i += 1; // Die integer Variable i wird um Eins erhöht (i += 1);
}
```

⇒Geben sie Werte und Adressen der Integer-Variable i und den Zeiger auf i (i\_ptr) aus.

Unabhängig vom Daten-Typ ist der benötigte Speicher für eine Zeigervariable immer gleich groß - der zur Speicherung benötigte Platz für die Adresse des Objekts (4 Byte auf einer 32-Bit-Rechner-Architektur, siehe auch Abb. 7.2).

Was ist neu bei dieser Übung.

| C++ Ding | Was tut's                                       |
|----------|---|
| *ptr     | Verweisoperator (unär: ein Operand)             |
| i*i      | Multiplikations-Operator (biär: zwei Operanden) |

Tabelle 7.2: C++ news: Operatoren

### 7.2.2 NULL Zeiger

Wir haben gesehen, dass Referenzen auf Daten-Objekte zwingenderweise initialisiert werden müssen (sonst streikt der Compiler). Zeiger müssen eigentlich nicht initialisiert werden, ihnen wird bei der Definition eine 'vernünftige' Adresse zu gewiesen. Dies bedeutet noch lange nicht, dass sich hinter dieser Adresse etwas Vernünftiges verbirgt. Daher ist es ratsam, auch Zeiger zu initialisieren, um Nachfragen zu können, ob sie tatsächlich auf existierende Objekte zeigen. Zum Initialisieren von Pointern gibt es den NULL-Zeiger.

```
double* ptr_x = NULL;
ptr_x = &x;
if(!ptr_x) cout << "Warning: ptr_x does not have a meaningful adress";
```

Der linke Operand der Operators = muss immer auf eine 'sinnvolle' Speicherstelle verweisen. Dieser wird daher auch als so genannter L-Wert (L(ef)t value) bezeichnet. Wenn eine Fehlermeldung 'wrong L-value' erscheint, wissen sie, dass keine 'sinnvolle' Adresse vergeben wurde, dies spricht wiederum für eine Initialisierung von Zeigern mit Zero-Pointer.

### 7.2.3 Zeiger als Parameter

Der Verweisoperator ist uns unlängst schon mal begegnet bei der erweiterten main() Funktion mit Eingabeparametern: `int main(int argc, char* argv[])`

## 7.3 Zeiger und Arrays

Wie bereits gesagt, die Verzeigerungs-Technik ist eine der Besonderheiten von objekt-orientierten Sprachen, wie z.B. von C++. Mit solchen Pointern können wir auch größere Daten-Objekte wie Vektoren verwalten. Die zentrale Idee ist: Wenn Startpunkt (*id* Adresse) und Länge des Vektors bekannt sind, wissen wir eigentlich alles und können auf alle Vektor-Daten zugreifen.

### 7.3.1 Statische Objekte

Bei der Einführung der String-Klasse hatten wir bereits mit Zeichenketten zu tun. Aus der Sicht des Speichers ist eine Zeichenkette ein Vektor der eine bestimmte Anzahl von Zeichen enthält. Vektoren können natürlich für (fast) alle Datentypen definiert werden. Im nächsten Kapitel (8) beschäftigen wir uns mit sogenannten Containern, damit können z.B. Vektoren, Listen etc. für ganze Klassen generiert werden.

```
char Zeichenkette[80];
int iVektor[50];
double dVektor[50];
```

### 7.3.2 Dynamische Objekte

Bisher haben wir uns fast ausschließlich mit Datentypen beschäftigt, deren Speicherbedarf bereits während der Kompilierung fest steht und reserviert wird (bis auf Strings-Objekte). Es gibt aber auch die Möglichkeit, den Speicherbedarf während der Programmausführung zu ändern - dies geht mit dynamischen Daten-Objekten. Die werden wir folgt deklariert.

```
char* Zeichenkette;
int* iVektor;
double* dVektor;
```

Nun müssen wir uns allerdings selber um die Speicherreservierung kümmern. Dies erfolgt mit dem `new`-Operator und wird in der nächsten Übung gezeigt. Wir ahnen schon, wenn es etwas zum Vereinbaren von Speicher gibt, muss es wohl auch das Gegenstück - freigeben von Speicher - geben. Natürlich habe sie Recht. Dies erledigt der `delete`-Operator.

```
double* dVektor;
dVektor = new double[1000];
delete [] dVektor;
```

Die Anwendung beider Operatoren zum Speichermanagement sowie eine Möglichkeit, den verbrauchten Speicher zu messen, werden in der folgenden Übung gezeigt.

Übung: E7.3.2

```
#include <iostream> // for using cout
#include <malloc.h> // for using malloc_usable_size
using namespace std;

int main()
{
    char* Zeichenkette; // Definitions
    long size_memory; // Auxiliary variable for memory size
    Zeichenkette = new char[1000]; // Memory allocation
    size_memory = malloc_usable_size(Zeichenkette); // calculation of memory
    delete [] Zeichenkette; // Memory release
    return 0;
}
```

HomeWork: HW7.3.2

*Schreiben sie ein kleines Programm, mit der sie den benötigten Speicher ermitteln können und prüfen sie, ob die Freigabe des Speichers funktioniert hat.*

Zeiger können auch verschachtelt werden: Zeiger auf Zeiger ... Damit lassen sich mehrdimensionale Datenstrukturen schaffen, z.B. Matrizen.

## 7.4 Summary

Das Verstehen und Arbeiten mit Zeigern und Referenzen ist nicht ganz einfach und braucht eine ganze Zeit der praktischen Anwendung. Der `*`Operator

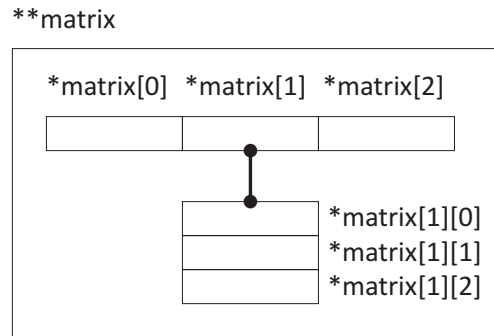


Abbildung 7.3: Die Definition einer Matrix mittels Pointer

kann beispielsweise als Zeiger auf etwas und gleichzeitig für dynamische Vektoren benutzt werden. Dies scheint zunächst absolut verwirrend zu sein, beim genaueren Überlegen aber: \* ist lediglich die Startadresse für den Vektor. Merken sie sich erst einmal nur, dass es geschickter (und schneller) ist, nicht mit den Daten-Objekten direkt sondern mit Verweisen darauf (also deren Adressen) zu arbeiten. Die nachfolgende Tabelle und Abbildung sollen das Thema 'Referenzen und Zeiger' noch einmal illustrieren.

| Syntax                       | Bedeutung   |
|------------------------------|---|
| <code>double x</code>        | Definition einer doppelt-genauen Gleitkommazahl (Speicherbedarf 8 Byte) |
| <code>double* ptr</code>     | Zeiger auf eine doppelt-genauen Gleitkommazahl (Speicherbedarf 4 Byte)  |
| <code>double&amp; ref</code> | Referenz auf eine doppelt-genaue Gleitkommazahl (Speicherbedarf 4 Byte) |
| <code>double* dVektor</code> | Zeiger auf einen Gleitkommazahl-Vektor                                  |

Tabelle 7.3: Zeiger und Referenzen

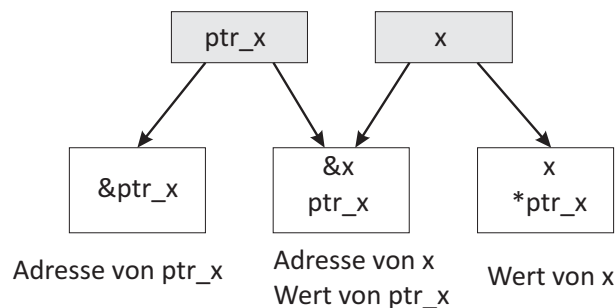


Abbildung 7.4: Referenzen und Zeiger (nach [5])

## 7.5 Testfragen

1. Was ist `&x` ? (`x` ist ein beliebiger Datenobjekt `T`, z.B. eine Gleitkommazahl: `double x`)
2. Was ist eine Referenz `&ref`?
3. Was bedeutet die Anweisung: `&ref = x`?
4. Erklären sie kurz (mit eigenen Worten) das Zeiger-Konzept in C++?
5. Was bewirkt der NULL Zeiger, d.h. `*ptr = NULL` ?
6. Was bedeutet die Anweisung: `ptr = &x`?
7. Müssen Referenzen (`&`) und Zeiger (`*`) auf Daten-Objekte initialisiert werden ?
8. Was bewirkt die Definition `long l[1000]` speichertechnisch ?
9. Wie groß ist der Speicherbedarf des statischen Datenobjekts `long l[1000]` ?
10. Wie groß ist der Speicherbedarf des dynamische Datenobjekts `long* ptr = new long[1000]` ?
11. Woher kommt der Unterschied (4 Byte auf einem 32Bit Rechner) im Speicherbedarf zwischen statischen und dynamischen Datenobjekten.
12. Zusatzfrage: Was bedeutet die Definition `**ptrptr` (Zeiger auf Zeiger), was für ein Datenkonstrukt entsteht ?

# Kapitel 8

## Container

Wir haben bereits mit der String-Klasse gesehen, dass C++ neben der Basisfunktionalität einer Programmiersprache auch sehr nützliche Erweiterungen, wie das Arbeiten mit Zeichenketten, anbietet. Diese Erweiterungen gehören zum Standard von C++, gehören also zum 'Lieferumfang' des Compilers dazu und heißen daher auch Standard-Klassen. Diese Klassen bilden die sogenannte Standard-Bibliothek (STL - STandard Library).

In diesem Kapitel beschäftigen wir uns mit sogenannten Containern mit denen Daten organisiert, gespeichert und verwaltet werden können, z.B. Listen und Vektoren. In der Abb. 8.1 sind die wichtigsten Elemente der Container dargestellt. Der Begriff Container soll verdeutlichen, dass Objekte des gleichen Typs z.B. in einer Liste gespeichert werden. Und wie immer bei Klassen, werden natürlich auch die Methoden für den Zugriff oder die Bearbeitung von diesen Objekten bereitgestellt. Die Speicherverwaltung für Container-Elemente erfolgt dynamisch zur Laufzeit (siehe Kapitel Speicherverwaltung). Die Abb. 8.1 zeigt auch, welche verschiedenen Typen von Containern es gibt:

**Sequentielle Container:** sind z.B. Vektoren, Listen, Queues und Stacks. Vektoren sind Felder (arrays) in denen die einzelnen Elemente nacheinander angeordnet sind. Bei einer Liste kennt jedes Element nur seine Nachbarn (Vorgänger- und Nachfolgeelement). Queues (wir kennen dies von Druckern) sind Warteschlangen, die nach dem FIFO-Prinzip (first in, first out) funktionieren. Das heißt, dass zuerst eingefügte Element wird auch zuerst verarbeitet (z.B. Druckjobs). Stacks sind sogenannte Kellerstapel, die nach dem LIFO-Prinzip (last in, first out) funktionieren. Das bedeutet, das zuletzt eingefügte Element wird zuerst verarbeitet.

**Assoziative Container:** sind z.B. Maps (Karten) und Bitsets. Die Elemente sind nicht wie bei sequentiellen Container linear angeordnet sondern in bestimmtem Strukturen (z.B. Baumstrukturen). Dies ermöglicht einen besonders schnellen Datenzugriff, der über eine Verschlüsselung erfolgt.

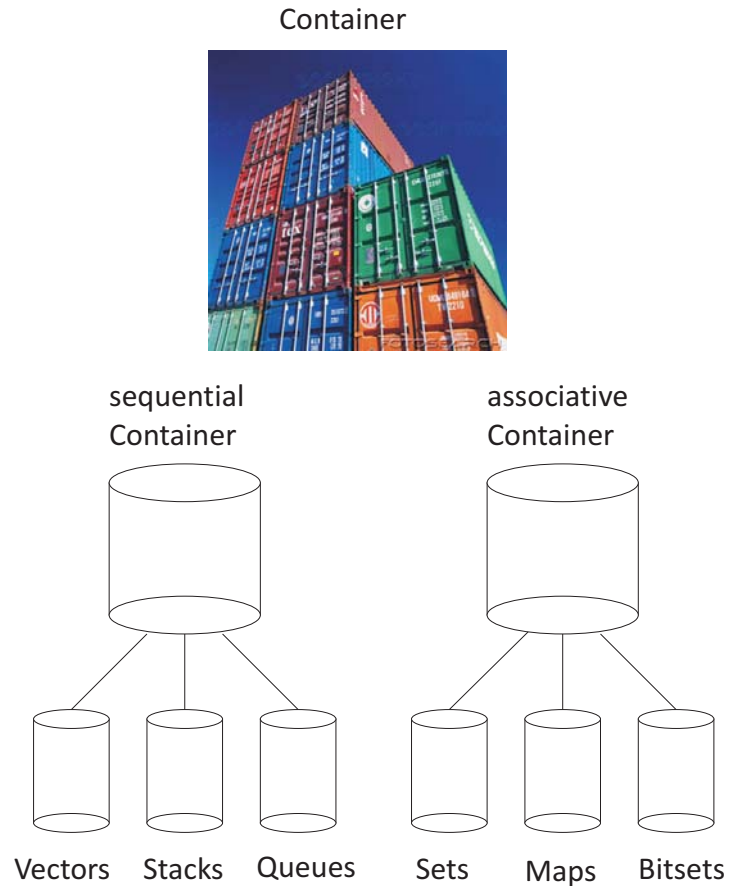


Abbildung 8.1: C++ Container (Bildquellen: [www.prachensky.com](http://www.prachensky.com))

Die Besonderheit der C++ Container ist die dynamische Speicherverwaltung. Das heisst, Einfügen und Entfernen von Container-Elementen ist sehr einfach, wir brauchen uns nicht selbst um das Speichermanagement zu kümmern. Dies übernehmen die Konstruktoren und Destruktoren (siehe Abschnitt 4.4) der jeweiligen Container-Klasse.

## 8.1 Sequentielle Container

Ein weiterer Grund für die Bereitstellung von Containern ist, dass die Elemente gemeinsame Datentypen und Methoden haben. Der Datentyp `T` eines Containers ist natürlich eine (beliebige) Klasse. Die Tab. 8.1 listet die Klassentemplates für einige sequentielle Container sowie die benötigten Include-Files auf.

Ein Klassentemplate ist nichts weiter als eine formale Beschreibung der Syntax.



| Klassen-Template                       | Include-Files               |
|--|-----------------------------|
| <code>vector&lt;T,Allocator&gt;</code> | <code>&lt;vector&gt;</code> |
| <code>list&lt;T,Allocator&gt;</code>   | <code>&lt;list&gt;</code>   |
| <code>stack&lt;T,Container&gt;</code>  | <code>&lt;stack&gt;</code>  |
| <code>queue&lt;T,Container&gt;</code>  | <code>&lt;queue&gt;</code>  |

Tabelle 8.1: Klassentemplates für sequentielle Container

- `vector`: Schlüsselwort
- `< ... >`: Parameterliste
- `T`: erster Parameter, Datentyp des Vektors
- `Allocator`: zweiter Parameter, Speichermodell des Vektors

Die Methoden sequentieller Container sind sehr einfach, intuitiv zu bedienen und sie sind gleich für Vektoren, Listen etc. Eine Auswahl der wichtigsten gemeinsamen Methoden finden sie in der nachfolgenden Tabelle.

| Methode                   | Bedeutung                                |
|---------------------------|--|
| <code>size()</code>       | Anzahl der Elemente, Länge de Containers |
| <code>push_back(T)</code> | Einfügen eines Elements am Ende          |
| <code>pop_back()</code>   | Löschen des letzten Elements             |
| <code>insert(p,T)</code>  | Einfügen eines Elements vor p            |
| <code>erase(p)</code>     | Löschen des p-Elements (an der Stelle p) |
| <code>clear()</code>      | Löscht alle Elemente                     |
| <code>resize(n)</code>    | Ändern der Containerlänge                |

Tabelle 8.2: Methoden für sequentielle Container

## 8.2 Vectors

The standard vector is a template defined in namespace `std` and presented in `<vector>`. The vector container is introduced in stages: member types, iterators, element access, constructors, list operations etc. The structure of a vector is illustrated in Fig. 8.2.

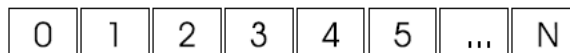


Abbildung 8.2: Structure of vectors

Following lines are necessary in order to use the vector container in your sources code.

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

Types: Very different types can be used in vectors. A vector e.g. of integers can be declared in this way.

```
// Declaration and construction
vector<int> v;
```

Constructors: With the above source code a vector of integer types is constructed as well.

Stack operators: The functions `push_back` and `pop_back` treat the vector as a stack by adding and removing elements from its end.

```
// Input elements to vector
for(i=0;i<9;i++)
{
  v.push_back(i+1);
}
```

Iterators: Iterators can be used to navigate containers without the programmer having to know the actual type to identify the elements, e.g. for inserting and deleting elements. A few number of key functions allow to step through the elements, such as `begin()` pointing to first element and `end()` pointing to the one-past-last element. An example of a loop through all vector elements using the iterator is given below.

```
// Navigating vector elements
for(vector<int>::iterator it=v.begin(); it!=v.end(); ++it)
{
  cout << *it << '\n';
}
```

Vector elements can be addressed directly by element number. Fast data access is an advantage of vectors.

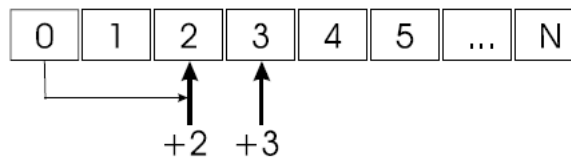


Abbildung 8.3: Access to vector elements

In contrast, vectors are not as flexible as lists. Deleting and adding of elements within the vector is complicated (Fig. 8.4).

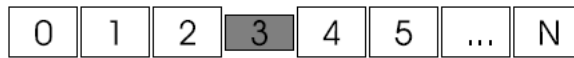


Abbildung 8.4: Deleting elements

## Programming exercises

### 8.2.1 Defining vectors

In the first exercise we just define a vector and insert some elements into it in order to show the concept.

#### Exercise 8.2.1

```
#include <iostream>    // for using cout
#include <vector>      // for using vectors
using namespace std;  // for std functions
void main()
{
    //-----
    cout << "E81 - Working with vectors" << endl;
    //-----
    vector<int>my_vector;    // Declaration of vector
    my_vector.push_back(1); // Inserting an element to vector
    my_vector.push_back(3);
    my_vector.push_back(5);
    for(int i=0;i<(int)my_vector.size();i++)
    {
        cout << my_vector[i] << endl; // Access to vector elements
    }
    //-----
}
```

### 8.2.2 Vectors and data base

In der nächsten Übung werden einige Dinge, die wir bisher gelernt haben, zusammenbringen. Sie werden sehen, dass 'gut' programmierte Dinge einfach 'zusammengebaut' werden können: Objekte (Kapitel 4), IO-Methoden (Kapitel 6), Stringverarbeitung (Kapitel 5).

## Exercise 8.2.2

```

#include <iostream> // for using cout
#include <fstream> // for using ifstream / ofstream
#include <string> // for using string
#include <vector> // for using vectors
#include "student.h" // for using CStudents
using namespace std; // for std functions

#define MAX_ZEILE 256
bool STDRead(ifstream&);

int main()
{
    //-----
    cout << "E82: an object-oriented DB read function" << endl;
    //-----
    // 1 File handling
    ifstream input_file; // ifstream instance
    input_file.open("data_base.txt");
    if(!input_file.good()) // Check is file existing
    {
        cout << "! Error: input file could not be opened" << endl;
        return 0;
    }
    //-----
    // 2 Read data base
    STDRead(input_file);
    //-----
    return 0;
}

```

Unser Programm besteht aus drei gekapselten Teilen:

- `main()`: Das Hauptprogramm: die Steuerzentrale, hier wir i.W. das Dateimanagement erledigt.
- `STDRead(ifstream&)`: Die übergeordnete Lesefunktion für die komplette Datenbank, hier befindet sich eigentlich nur eine Schleife, welche die Objekt-Lesefunktion aufruft, solange Daten in der Datei vorhanden sind.
- `CStudent::Read(ifstream)`: Die Objekt-Lesefunktion, die liest die eigentlichen Daten für jeden Datensatz aus der Datenbankdatei.

Der einzige Übergabeparameter ist die Datenbank-Datei: `std_file`.

Exercise 8.2.2 continued

```

/*****
STDLib-Method:
Task: Reading all STD objects
06/2009 OK Implementation
*****/
bool STDRead(istream& std_file)
{
    //-1-----
    char line[MAX_ZEILE];
    string line_string;
    ios::pos_type position;
    vector<CStudent*>std_vector;
    CStudent* m_std = NULL;
    //-----
//OK  STDelete();
    //-2-----
    // rewind the file
    std_file.seekg(0,ios::beg);
    //-3-----
    // OBJ reading, keyword loop
    cout << "STDRead" << endl;
    while (!std_file.eof())
    {
        std_file.getline(line,MAX_ZEILE);
        line_string = line;
        if(line_string.find("#STOP")!=string::npos)
            break;
        if(line_string.find("#STUDENT")!=string::npos)
        {
            m_std = new CStudent();
            position = m_std->Read(std_file); // from E63
            std_vector.push_back(m_std);
            std_file.seekg(position,ios::beg);
        }
    } // eof
    //-----
    cout << "Number of data sets: " << std_vector.size() << endl;
    return true;
}

```

### 8.2.3 Updating your data base entry

Homework 8.2.3a: Please update your data base entries and send it by mail to me (olaf.kolditz@ufz.de).

Homework 8.2.3b: Please write a print function for the CStudent class.

### 8.3 Lists

A list is a sequence optimized for insertion and deletion of elements. They allow a very flexible organization of elements. But the price for flexibility is a relatively slow access to data.

List provides bidirectional iterators (Fig. 8.3). This implies that a STL list will typically be implemented using some form of a doubly-linked list.

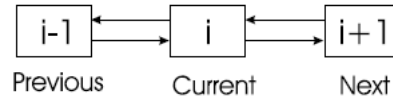


Abbildung 8.5: Structure of lists

Following lines are necessary in order to use the list container in your sources code.

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

Types: All data types are possible for lists, e.g. a class.

```
// Declaration
list<CStudent*> std_list;
list<int> int_list;
```

Constructors: With the above source code a list for class instances is constructed as well.

Stack operators: Lists provide same typical member functions as for vectors.

```
// Insert list elements at end
CStudent* m_std;
my_list.push_back(m_std);
```

Iterators: Again iterators can be used to navigate lists. We show an example for deleting a specific element from the list. Identification of the list element is by name (Fig. 8.6).

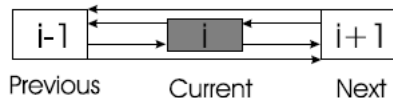


Abbildung 8.6: Deleting list elements

The following exercise shows how conveniently we can deal with lists, i.e. in order to sort, reverse, merge, and unify lists. The unifying function removes

multiple entries from lists, which is a very nice tool in order to clean a data base.

### Exercise 8.3

```
#include <iostream> // for using cout
#include <list>      // for using lists
using namespace std; // for std functions
typedef list<int>INTLIST;
void display(list<int>);

int main()
{
    int i;
    list<int>list1;
    INTLIST list2;
    // Filling the list with random numbers
    for(i=0;i<4;i++)
    {
        list1.push_back(rand()%10);
        list2.push_back(rand()%10);
    }
    display(list1); // Display the list
    display(list2);
    // Putting first element to the end as well
    list1.push_back(list1.front());
    list1.reverse(); // Reverse list
    list1.sort();    // Sort list
    list1.merge(list2); // Merge two lists
    list1.sort();    // Remove multiple entries #1
    list1.unique();  // Remove multiple entries #2
    return 0;
}

void display(list<int>my_list)
{
    int i;
    list<int>::const_iterator iterator;
    iterator = my_list.begin();
    while(iterator != my_list.end())
    {
        i = *iterator;
        cout << i << endl;
        ++iterator;
    }
}
```

The concept of the data base read function is illustrated in Fig. 8.7.

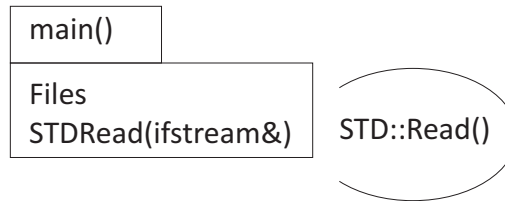


Abbildung 8.7: Data base read concept

The last exercise shows how we can delete entries from the students data base (i.e. people we don't like)

```

// Deleting list elements
CStudent* m_std = NULL;

list<CStudent*>::const_iterator p = std_list.begin();
while(p!=std_list.end())
{
    m_std = *p;
    if(m_std->GetStudentName()=="James Bond")
    {
        my_list.remove(*p);
        break;
    }
    ++p;
}
// Deleting repeated entries
std_list.unique();
  
```

## 8.4 Testfragen

1. Was sind C++ Container ?
2. Welche Typen von C++ Containern kennen sie ?
3. Worin besteht der Unterschied zwischen sequentiellen und assoziativen Containern ?
4. Welche sequentiellen Container kennen sie ?
5. Erklären sie die Syntax des Vektor-Containers: `vector<int>my_vector` .
6. Was ist der Unterschied zwischen Vektoren und Listen ?
7. Was sind die Gemeinsamkeiten von Vektoren und Listen ?
8. Welcher Include ist notwendig für das Arbeiten mit Vektoren ?
9. Welcher Include ist notwendig für das Arbeiten mit Listen ?



10. Benötigen wir den Zusatz (Namensraum) `using namespace std`, wenn ja warum ?
11. Mit welcher Instruktion können Elemente in Vektoren und Listen einfügen ?
12. Wo werden sequentielle Container-Elemente mit der Instruktion `push_back(T)` eingefügt ?
13. Mit welcher Anweisung können wir die Länge von sequentiellen Container-Elementen bestimmen ?
14. Mit welcher Anweisung können wir einen Vektor platt machen (d.h. alle Elemente löschen) ?
15. Wie können wir auf ein Vektor-Element, sagen wir das 17te Element des Vektors `vector<int>my_vector`, direkt zugreifen ?
16. Quellcode verstehen: Erklären sie die Struktur (1,2,3) der DB-Lese-Funktion `STDRead(ifstream& std_file)` in der Übung 8.2.2. Beginnen sie mit der Parameterliste.
17. Wie können wir unsere Studenten-Klasse `CStudent` in die DB-Anwendung (Übung 8.2.1) einbinden ?
18. Was ist eigentliche Lesefunktion für unsere Studenten-Datensätze (Übung 8.2.2) ?
19. Mit welchem Befehl können wir die Reihenfolge von Listen-Elementen umkehren ?
20. Können wir Listen-Elemente sortieren, wenn ja wie, mit welcher Instruktion ?
21. Mit welchem Befehl können wir mehrere Listen zusammenführen ?
22. Können wir doppelte Elemente aus einer Liste entfernen sortieren, wenn ja wie, mit welcher Instruktion ?
23. Was ist ein Iterator ?
24. Quellcode verstehen: Erklären sie die Funktion `void display(list<int>my_list)` der Übung 8.3. Beginnen sie mit der Parameterliste.
25. Wie können wir Elemente aus einer Liste entfernen ?

# Kapitel 9

## Andere Sprachelemente

Wir haben das Ende der C++ Grundvorlesung erreicht (hoffentlich ein Grund zur Freude). Das Ziel der Vorlesung 'Hydroinformatik' ist allerdings bei Weitem noch nicht erreicht. Sie erinnern sich, wir wollen eine Dialog-geführte Datenbank-Anwendung (Fig. 4) mit C++ entwickeln. Sie werden sehen, dass wir fast alle Bausteine beisammen haben (außer die Grafik-Programmierung mit Visual C++) und die Realisierung unserer Applikation (nach der langen Vorbereitung) dann sehr schnell geht.

Bevor wir zum graphischen (Visual) C++ kommen, wollen wir Sprachelemente (C++ news), die wir zuvor schon benutzt haben, aber noch nicht explizit besprochen haben, hier kurz einführen.

### 9.1 Kontrollstrukturen

Bei Kontrollstrukturen handelt es sich um Sprachelemente für bedingte Anweisungen und Schleifen mit Zählern und Bedingungen.

| Control                  | Bedeutung                   |
|--------------------------|-----------------------------|
| <code>if-else</code>     | Anweisungen mit Bedingungen |
| <code>switch-case</code> | Fallunterscheidungen        |
| <code>for</code>         | Schleifen mit Zählern       |
| <code>while</code>       | Schleifen mit Bedingungen   |

Tabelle 9.1: Kontrollstrukturen

Verschachtelungen von Kontrollstrukturen sind natürlich möglich.

### 9.1.1 if-else

Es geht um die Ausführung von Anweisungen und Alternativen unter bestimmten Bedingungen. Ausführung einer Anweisung unter einer Bedingung (siehe, z.B. Übung 8.2.2):

```
if(logische Bedingung)
{...} // Ausführung, wenn Bedingung erfüllt ist
```

Ausführung einer Anweisung unter einer Bedingung mit Alternative (**else**) (siehe, z.B. Übung 8.2.2):

```
if(logische Bedingung)
{...} // Ausführung, wenn Bedingung erfüllt ist
else
{...} // Ausführung, wenn Bedingung nicht erfüllt ist
```

### 9.1.2 switch-case

Die Kontrollstruktur **switch-case** bietet die Möglichkeit von Fallunterscheidungen durch Ausdrücke (z.B. Aufzählungen durch Zahlen). Der Schalter **switch** springt entsprechend dem Zahlenausdruck in einen Fall **case**.

```
switch(Ausdruck)
{
    case 0: // wenn Ausdruck == 0
        ... // Anweisungen
        break; // Verlassen
    default: // wenn kein Ausdruck zutrifft
}

```

Es gibt eine weitere Variante, wo der Ausdruck ein Zeichen sein kann.

```
switch(Ausdruck)
{
    case 'A': // wenn Ausdruck == A
        ... // Anweisung
        break; // Verlassen
}

```

Eine weitere Möglichkeit diese Kontrollstruktur elegant einzusetzen ist die Benutzung von **enum** - einem Konstrukt für eigene Zähler.

```
enum Wochentag {Montag,Dienstag,Mittwoch,Donnerstag,Freitag,Samstag,Sonntag}
```

Die einzelnen Wochentage bekommen dann aufsteigende Werte (von 0 beginnend) zugewiesen. Dann können wir unsere Fallunterscheidung (z.B. für das Mensa-Menü) wie folgt schreiben:

```
switch(Wochentag)
{
  case Montag: // wenn Wochentag == 0
  case Dienstag: // wenn Wochentag == 1
  ...
  case Sonntag: // wenn Wochentag == 6
}
```

### 9.1.3 for

Die `for` Schleife ist eine Schleife mit einem Zähler.

```
for(int i=0;i<stop;i++)
{
  ... // Anweisungen
}
```

Die Anweisungen im Funktionsrumpf `{...}` werden solange ausgeführt, bis der Zähler den Wert `stop` erreicht (das heißt, für den Wert `stop` wird die Schleife nicht mehr ausgeführt). Bei jedem Schleifendurchlauf wird der Zähler `i` um Eins (`i++`) hochgezählt. Ich habe gerade mal nachgesehen. In unserem Programmsystem `OpenGeoSys` wird die `for`-Schleife über 4000 mal verwendet! Wir haben sie z.B. in der Übung 8.2.1 zum Generieren von Vektor-Elementen benutzt.

### 9.1.4 while

Die `while` Schleife ist eine Endlos-Schleife mit einer Bedingung.

```
while(Bedingung)
{
  ... // Anweisungen
}
```

Die Anweisungen im Funktionsrumpf `{...}` werden ausgeführt, solange die Bedingung erfüllt (das heißt, der numerische Wert `Bedingung` muss größer als 0 sein). Um aus der Schleife herauszukommen, muss also der Wert der Bedingung geändert werden - oder 'hart' abgebrochen werden (siehe nächster Abschn. 9.1.5) In unserem Programmsystem `OpenGeoSys` wird die `while`-Schleife fast 500 mal verwendet! Wir haben sie für unsere Lesefunktion (Übung 8.2.1) und beim Iterieren in Listen (Übung 8.2.1) verwendet.

### 9.1.5 continue, break, return

Es gibt mehrere Möglichkeiten, um Kontrollstrukturen (vorzeitig) verlassen zu können.

## 9.2 Gültigkeitsbereiche

Für alles, was wir deklarieren, Funktionen, Klassen, Variablen gibt es Gültigkeitsbereiche. Wenn irgendwo etwas definieren, ist es also nicht überall sichtbar. Wir hatten uns mit diesem Thema schon bei der Definition von 'privaten' Klassen-Elementen beschäftigt (Abschn. 4.6). Die Abb. ?? zeigt das Prinzip von lokalen und globalen Gültigkeitsbereichen. Wir sollen immer versuchen soviel wie möglich lokal zu definieren, das unterstützt die Daten-Kapselung in unserem Programm. Klassen sind immer nur dort sichtbar, wo ihre Header-Dateien inkludiert worden sind.

### 9.2.1 Namensbereiche

Sind uns auch schon mehrfach begegnet bei der Verwendung von Strings, Vektoren und Listen. Dabei wurde nach dem Includieren der entsprechenden Header-Dateien die Anweisung `using namespace std` benutzt. Strings, Vektoren und Listen gehören zum Namensraum der Standard-Bibliothek STL.

```
#include <strings>
std::my_string("Beispiel ohne Namensbereich");
using namespace std;
my_string("Beispiel mit Namensbereich");
```

Wir können auch eigene Namensbereiche definieren.

```
namespace Meins
{ // hier sind Funktionen des Namensbereichs
  class CStudent
}
using namespace Meins;
```

Wozu ? Wenn sie ihren Programmcode mit einem anderen Entwickler zusammenführen wollen und es gibt mehrere Klassen CStudent, dann können sie sicher auf ihre Funktion zugreifen, wenn sie ihren Namensraum gesetzt haben.

### 9.2.2 Compiler-Direktiven – #

Alles was mit # anfängt, versetzt den Compiler in den Alarmzustand - es sind sogenannte Compiler-Direktiven, das heißt beim Kompilieren werden gewisse Weichen gestellt.

| Direktive             | Wirkung                     |
|-----------------------|-----------------------------|
| <code>#include</code> | Einfügen einer Header-Datei |
| <code>#define</code>  | Definition einer Direktive  |
| <code>#ifdef</code>   | Wenn die Direktive gilt     |
| <code>#ifndef</code>  | Wenn sie nicht gilt         |
| <code>#endif</code>   | Ende der Direktive          |
| <code>#define</code>  | Definition eines Makros     |

Tabelle 9.2: Compiler-Direktiven

### 9.2.2.1 Includes

Ohne Includes geht gar nichts, sie erinnern sich, selbst um einfache Bildschirm-ausgaben mit `cout` machen zu können mussten wir Klasse `iostream` einfügen. An dieser Stelle nochmal zur Erinnerung. Es gibt Arten von Includes:

<**Header**> Die eckigen Klammern bedeuten, es geht um einen System-Inklude. Es wird im Systemverzeichnis (dort wo C++ installiert ist) gesucht.

”**Header**” Die Gänsefüßchen bedeuten, es geht um einen eigenen Inklude. Es wird lokalem Verzeichnis gesucht.

Wenn der Programm-Code verzweigt ist, also eine komplexere Verzeichnisstruktur hat, müssen Pfade der Inklude-Dateien mit angegeben werden, sonst werden sie nicht gefunden. Keine Angst, in diesem Falle bekommen sie sofort eine eindeutige Fehler-Meldung beim Versuch zu Kompilieren.

### 9.2.2.2 Bedingte Kompilation

Eine sehr nützliche Sache sind Direktiven für die Kompilation von Test-Versionen. Gerade während der Entwicklung möchte man neue Programmteile einfach an- und ausschalten können. Dies geht mit der Compiler-Direktive (KD) `#ifdef`.

```
#define TEST // Anschalten der Direktive TEST
#ifdef TEST
...
#endif
```

Wenn die Compiler-Direktive `TEST` an ist, wird der Programmtext von der Stelle `#ifdef` bis `#endif` beim Kompilieren übersetzt, ansonsten einfach ignoriert. Compiler-Direktiven sind immer aus, wenn sie nicht explizit eingeschaltet werden. Das Gegenstück von `#ifdef` ist `#ifndef`, also wenn die Compiler-Direktive aus ist. Oft müssen Funktionen überarbeitet werden. Mit Compiler-Direktiven kann man sehr schön eine neue Variante der Funktion entwickeln und die alte

erstmal zu Vergleichszwecken beibehalten. Erst wenn alles richtig funktioniert, schmeißt man das alte Zeug weg.

Übung 9.2.2.2:

```
#define NEW_VERSION // Anschalten der Direktive NEW_VERSION
#ifndef NEW_VERSION
... // wenn NEW_VERSION an ist
#else
... // wenn NEW_VERSION aus ist
#endif
```

Ich empfehle ihnen, das mal in irgendeiner Übung auszuprobieren. Wichtig: Denken sie immer an das `#endif` zum Ausschalten, sonst gilt die Kompiler-Direktive bis zum Ende des gesamten Programms.

### 9.2.2.3 Makros

Die Kompiler-Direktive `#define` kann auch zur Definition sogenannter Makros verwendet werden, z.B. für Konstanten, die man oft benutzt. Es geht aber auch für einfache Funktionen oder sogar Print-Anweisungen.

```
#define PI 3.1416 // Definition der Zahl PI
#define MULT(a,b) ((a)*(b))
#define PRINT(x) cout << (#x) << ": " << (x);
```

Der Makro-Einsatz sollte aber nicht übertrieben werden, eigentlich gibt es dafür richtige Funktionen. Bei Makros werden z.B. keine Prüfungen durch den Compiler gemacht, daher sind sie fehleranfällig.

## 9.3 Testfragen

1. Was sind Kontrollstrukturen, welche kennen sie ?
2. Bei welcher logischen Bedingung wird eine `if`-Anweisung ausgeführt ?
3. Lassen sich Kontroll-Strukturen verschachteln ?
4. Mit welcher Kontrollstruktur können wir Fallunterscheidungen programmieren ?
5. Welche Ausdrücke können wir bei der `switch-case` Kontrollstruktur benutzen ?
6. Wie kann ich eine Kompiler-Direktive an- und ausschalten ?

7. Schreiben sie die Übung 9.2.2.2 für die Benutzung von `#ifndef` anstelle von `#ifdef` um.
8. Erläutern sie den Kopf der `for`-Schleife: `for(int i=0;i<stop;i++)`, welchen Daten-Typ hat `stop` ?
9. Was ist eine Endlos-Schleife, wie kommen wir daraus ?
10. Was sind Namensbereiche ?
11. Was müssen wir tun, um die Arbeit mit C++ Standard-Klassen zu vereinfachen, d.h anstelle von `std::my_string()` direkt `my_string()` benutzen zu können ?
12. Was sind Compiler-Direktiven, was bewirken diese ?
13. Wie können wir Compiler-Direktiven an- und ausschalten ?
14. Worin besteht der Unterschied bei Includes mit eckigen Klammern `< ... >` bzw. mit Gänsefüßchen `"..."` ?
15. Schreiben sie die Übung 9.2.2.2 für die Benutzung von `#ifndef` anstelle von `#ifdef` um ?
16. Was sind Makros ?
17. Welche beiden Einsatzmöglichkeiten von Makros gibt es ?
18. Worin besteht die Gefahr, Makros als Funktionsersatz zu benutzen ?



**Part II**

**Visual C++**

# Kapitel 10

## Qt

Die Geschichte: Bei der ersten HI-I Vorlesung gabe es ziemlich peinliche eine Panne. Die (mit viel Mühe vorbereiteten Übungen) liefen zwar mit meiner MSVC++ (professional) Version aber (nicht ohne Weiteres) mit freien MSVC++ Express Version, wo die MFC standartmäßig nicht mehr dabei ist. Dies zeigt uns aber, auch beim Programmieren niemals nur auf ein Pferd setzen. Wichtig ist, unser eigener Code muss C++ Standard sein, den können wir dann problemlos in verschiedene GUI Frameworks, wie MSVC++, Qt etc. einbauen. Warum jetzt noch Qt ? Wir werden in den Übungen sehen, dass Qt sehr dicht an C++ Standards dran ist und Qt ist a cross-platform GUI. Qt läuft auf Windows, Linux, Mac ... In der Anlage 5.1 finden sie eine Anleitung zur Installation von Qt.

### 10.1 Hello Qt

Wie es sich für ein ordentliches Programmier-Framework gehört, ist die erste Übung "Hello World" auf graphisch.

Übung: 10.1

```
#include <QApplication>
#include <QLabel>

int main(int argc, char *argv[])
{
    QApplication app(argc,argv);
    QLabel *label = new QLabel("Hello Qt!");
    label->show();
    return app.exec();
}
```

Sehr erfreulich ist die Nähe von Qt zum Standard C++, es gibt eine klassische `main()` Funktion. Die Komposition von "Hello Qt" ist sehr einfach:

- Es wird eine Qt Application angelegt (ähnlich wie bei Windows, aber doch irgendwie einfacher),
- Es wird ein Label angelegt, das mit der Methode `→show()` angezeigt wird.

Qt bietet auch die Möglichkeit, einfach HTML code zu integrieren. Ersetzen sie die QLabel Zeile wie folgt.

```
QLabel *label = new QLabel("<h2><i>Hello</i>""<font color=red>Qt!</font></h2>");
```



Abbildung 10.1: Hallo Qt

## 10.2 Executable von der Konsole starten

Um ein ausführbares Qt Programm zu erzeugen benötigen wir 3+1 Schritte:

- Gehen sie in ihre Start - Programme Menü (Windows) und starten sie die den Qt command prompt (Fig. 10.2).

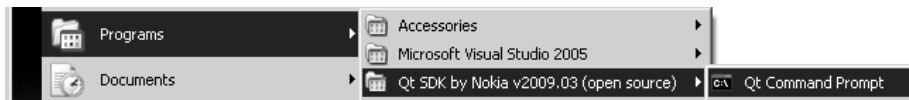


Abbildung 10.2: Qt Kommando Zeile

- Mit `qmake -project` erzeugen sie ein Qt Projekt.
- Mit `qmake` legen sie ein makefile an.
- Mit `mingw32-make` (kein Leerzeichen) kompilieren sie das Projekt und generieren ein ausführbares Programm.
- Mit `E10.1_HelloQt.exe` starten sie das Programm.

```

c:\ Qt Command Prompt
Setting up a MinGW/Qt only environment...
-- QTDIR set to C:\Qt\2009.03\qt
-- PATH set to C:\Qt\2009.03\qt\bin
-- Adding C:\Qt\2009.03\bin to PATH
-- Adding C:\WINDOWS\System32 to PATH
-- QMAKESPEC set to win32-g++

C:\Qt\2009.03\qt>cd C:\User\TEACHING\C++\EXERCISES\E10_Qt\E10_1_HelloQt
C:\User\TEACHING\C++\EXERCISES\E10_Qt\E10_1_HelloQt>qmake -project
C:\User\TEACHING\C++\EXERCISES\E10_Qt\E10_1_HelloQt>qmake

C:\User\TEACHING\C++\EXERCISES\E10_Qt\E10_1_HelloQt>mingw32-make
mingw32-make -f Makefile.Debug
mingw32-make[1]: Entering directory `C:/User/TEACHING/C++/EXERCISES/E10_Qt/E10_1_HelloQt`
g++ -c -g -frtti -fexceptions -mthreads -Wall -DUNICODE -DQT_LARGEFILE_SUPPORT -DQT_DLL -DQT_GUI_LIB -DQT_CORE_LIB -DQT_THREAD_SUPPORT -DQT_NEEDS_QMAIN -I\"c:\Qt\2009.03\qt\include\QtCore\" -I\"c:\Qt\2009.03\qt\include\QtGui\" -I\"c:\Qt\2009.03\qt\include\" -I\" -I\"c:\Qt\2009.03\qt\include\ActiveQt\" -I\"debug\" -I\"c:\Qt\2009.03\qt\mkspecs\win32-g++\" -o debug\helloQt.o helloQt.cpp
g++ -enable-stdcall-fixup -Wl,-enable-auto-import -Wl,-enable-runtime-pseudo-reloc -mthreads -Wl -Wl,-subsystem,windows -o debug\E10_1_HelloQt.exe debug\helloQt.o -L\"c:\Qt\2009.03\qt\lib\" -lmingw32 -lqtmaind -lQtGui4 -lQtCore4
mingw32-make[1]: Leaving directory `C:/User/TEACHING/C++/EXERCISES/E10_Qt/E10_1_HelloQt`

C:\User\TEACHING\C++\EXERCISES\E10_Qt\E10_1_HelloQt>cd debug
C:\User\TEACHING\C++\EXERCISES\E10_Qt\E10_1_HelloQt\debug>E10_1_HelloQt.exe
C:\User\TEACHING\C++\EXERCISES\E10_Qt\E10_1_HelloQt\debug>_

```

Abbildung 10.3: Die Schritte in Bildern ...

## 10.3 Quit - Schaltflächen

Die nächste Übung zeigt uns, wie einfach mit Schaltflächen (Push Button) gearbeitet werden kann.

```

#include <QApplication>
#include <QPushButton>

int main(int argc, char *argv[])
{
    QApplication app(argc,argv);
    QPushButton *button = new QPushButton("Quit");
    QObject::connect(button, SIGNAL(clicked()),&app, SLOT(quit()));
    button->show();
    return app.exec();
}

```

Im Unterschied zu "Hello Qt" ist eine Kommunikation mit dem Anwender (drücken der Schaltfläche) notwendig. Die Signalverarbeitung erfolgt über die Object-Methode connect.

```

QObject::connect(button, SIGNAL(clicked()),&app, SLOT(quit()));

```

Dabei wird die Nachricht `SIGNAL(clicked())` mit der Schaltfläche `Quit` mit einem sogenannten `SLOT(quit())` verbunden.

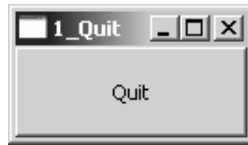


Abbildung 10.4: Hallo Qt

## 10.4 Dialoge

Zum Abschluss unsere Dialog-Anwendung. Dabei wird in der `main` Funktion eine Instanz der Klasse `QDialog` erzeugt.

```
#include <QtGui/QApplication>
#include "dialog.h"

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    Dialog w;
    w.show();
    return a.exec();
}
```

Der Header des Dialogs enthält im Wesentlichen die Ressourcen-Elemente, z.B. die Schaltfläche `pushButtonReadDB` zum Lesen der Datenbank sowie die dazugehörigen Funktionen `on_pushButtonReadDB_clicked()`.

```
class Dialog : public QDialog
{
    Q_OBJECT

public:
    Dialog(QWidget *parent = 0);
    ~Dialog();

private:
    Ui::Dialog *ui;

    QListWidget *listWidget;
    QPushButton *pushButtonReadDB;
    QPushButton *pushButtonAdd;
    QPushButton *pushButtonDelete;
```

```

    QLineEdit *lineEditNameFirst;
    QLineEdit *lineEditNameLast;

private slots:
    void on_pushButtonReadDB_clicked();
    void on_pushButtonAdd_clicked();
    void on_pushButtonDelete_clicked();
    void on_listWidget_clicked();
};

```

Die Implementierung des Dialogs besteht aus drei Teilen:

1. Konstruktion der grafischen Ressourcen-Elemente: pushButton (Schaltflächen), listWidget (Liste), lineEdit (Editierfelder),
2. Signalverarbeitung: Wenn pushButtonReadDB gedrückt wurde, führe die Funktion on\_pushButtonReadDB\_clicked aus,
3. Layout: Es werden horizontale und vertikale Boxen angelegt.

```

Dialog::Dialog(QWidget *parent)
    : QDialog(parent), ui(new Ui::Dialog)
{
    ui->setupUi(this);

    // 1.1 - pushButton
    pushButtonReadDB = new QPushButton(tr("&Read DB"));
    pushButtonAdd = new QPushButton(tr("&Add DS"));
    pushButtonDelete = new QPushButton(tr("&Delete DS"));
    // 1.2 - listWidget
    QString name = "Tom Hanks";
    QStringList names;
    names << name << "Alice Wonderland" << "Bob Dylan" << "Carol Crow"
        << "Donald Dug" << "Emma Thomson";
    listWidget = new QListWidget();
    for (int row = 0; row < 5; ++row) {
        listWidget->addItem(names[row]);
    }
    //1.3 - lineEdit
    lineEditNameFirst = new QLineEdit();
    lineEditNameFirst->insert("Tom");
    lineEditNameLast = new QLineEdit();
    lineEditNameLast->insert("Hanks");
    // 2 - connect
    connect(pushButtonReadDB, SIGNAL(clicked()), this,
        SLOT(on_pushButtonReadDB_clicked()));
    connect(pushButtonAdd, SIGNAL(clicked()), this,
        SLOT(on_pushButtonAdd_clicked()));
    connect(pushButtonDelete, SIGNAL(clicked()), this,

```

```

        SLOT(on_pushButtonDelete_clicked()));
connect(listWidget, SIGNAL(itemSelectionChanged()), this,
        SLOT(on_listWidget_clicked()));
//3 - Layout
QHBoxLayout *leftTopLayout = new QHBoxLayout;
leftTopLayout->addWidget(lineEditNameFirst);
leftTopLayout->addWidget(lineEditNameLast);
QVBoxLayout *leftLayout = new QVBoxLayout;
leftLayout->addLayout(leftTopLayout);
leftLayout->addWidget(listWidget);
QVBoxLayout *rightLayout = new QVBoxLayout;
rightLayout->addWidget(pushButton);
rightLayout->addWidget(pushButtonAdd);
rightLayout->addWidget(pushButtonDelete);
rightLayout->addStretch();
QHBoxLayout *mainLayout = new QHBoxLayout;
mainLayout->addLayout(leftLayout);
mainLayout->addLayout(rightLayout);
setLayout(mainLayout);
}

```

Das Ergebnis der Dialog-Implementierung sehen wir in Abb. 10.5.



Abbildung 10.5: Qt Dialog

Um zu prüfen, ob die Signalverarbeitung hinter den Knöpfchen funktioniert, rufen wir eine MessageBox auf.

```

void Dialog::on_pushButtonAdd_clicked()
{
    QMessageBox msgBox;
    msgBox.setText("pushButtonAdd clicked");
}

```

```
    msgBox.exec();  
}
```



## 10.5 Qt Projekt

Es ist ihnen hoffentlich gelungen, Qt für ihr Betriebssystem erfolgreich zu installieren. Eine kurze Installationsanleitung war im Skript [?] (Abschn. 12.8) zu finden. Hier noch mal die Web-Seite für den Download <http://qt.nokia.com/products>. Der große Vorteil von Qt ist die *Plattform-unabhängigkeit* sowie die freie Verfügbarkeit unter der GPL (Gnu Public License).

Wir starten Qt mit einem Doppelklick auf das Desktop-Symbol (Abb. 11.4).



Abbildung 10.6: Qt Start

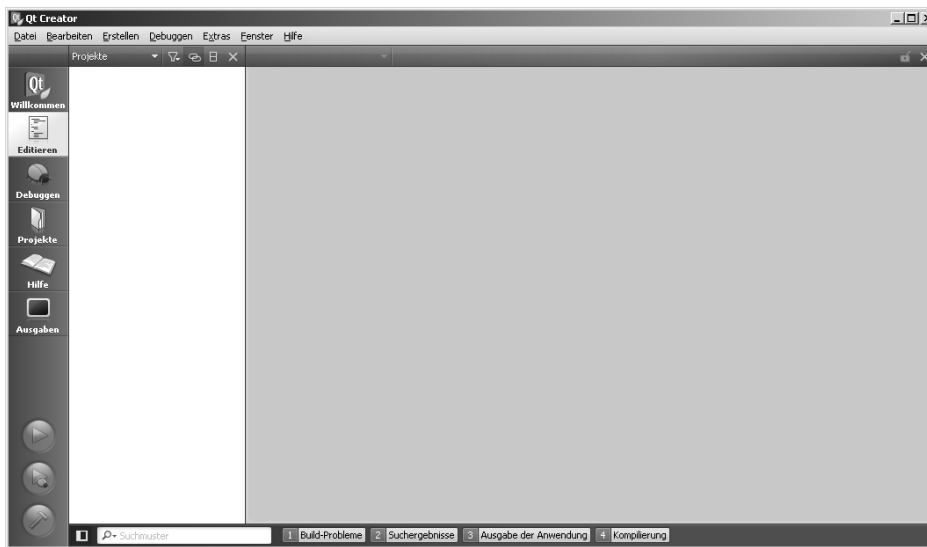


Abbildung 10.7: Qt Creator

Zunächst müssen wir ein neues (leeres) Qt Projekt anlegen (Abb. 11.4-10.11).

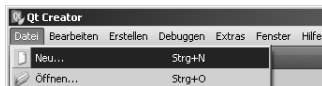


Abbildung 10.8: Neues Qt Projekt anlegen - Schritt 1

Schritte:

- pro Datei

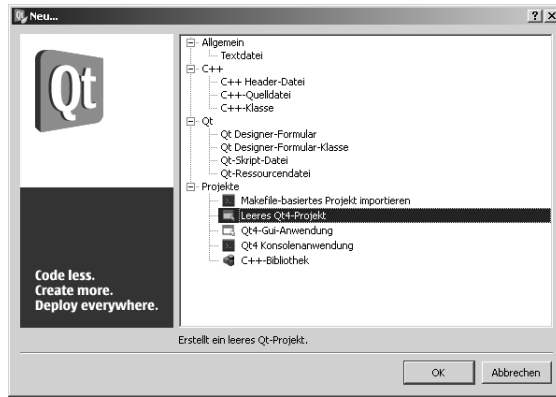


Abbildung 10.9: Neues Qt Projekt anlegen - Schritt 2

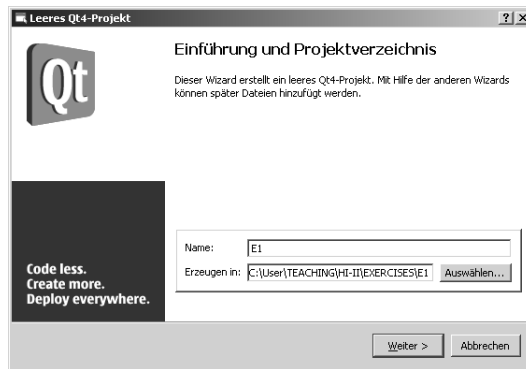


Abbildung 10.10: Neues Qt Projekt anlegen - Schritt 3

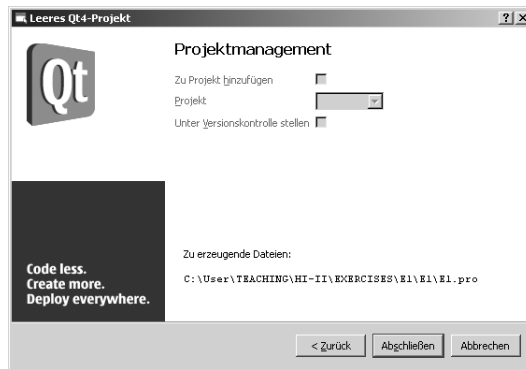


Abbildung 10.11: Neues Qt Projekt anlegen - Schritt 4

Nachdem ein leeres Qt Projekt angelegt ist, fügen wir zunächst eine existierende Quell-Datei mit der main Funktion hinzu (Abb. 10.12).

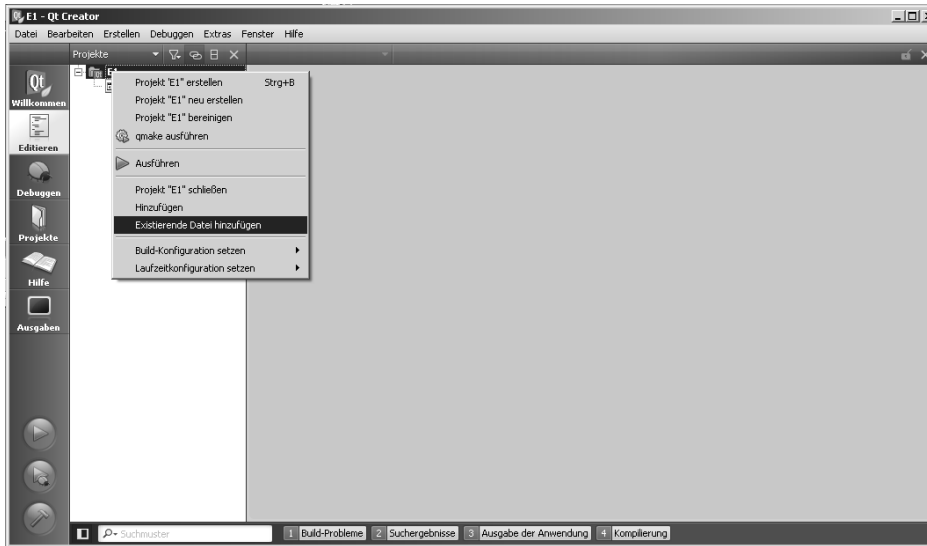


Abbildung 10.12: Hinzufügen einer Quell-Datei

Es können weitere Quell-Dateien hinzugefügt werden, wie z.B. der Qt Plotter (Abb. 10.13).

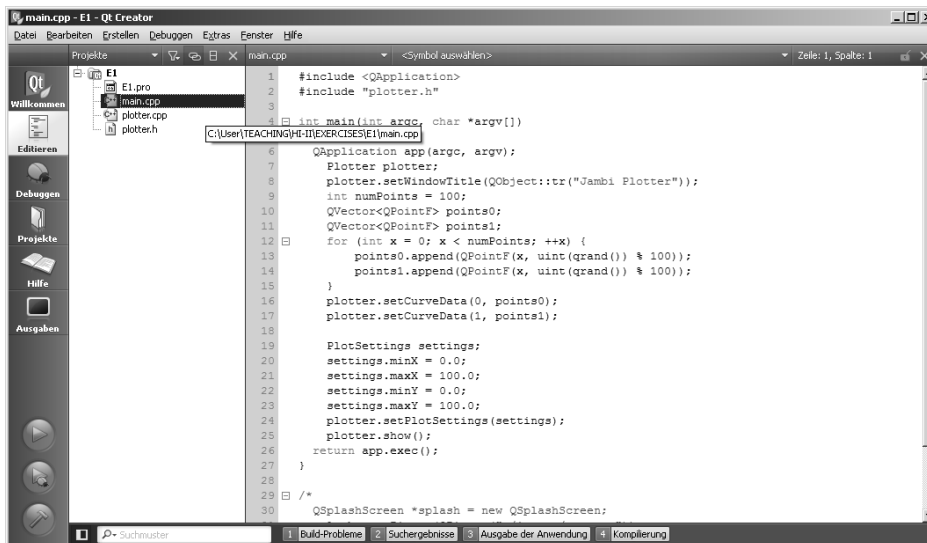


Abbildung 10.13: Qt Editor

Unser Projekt ist schon kompilierfähig. Mit einem Druck auf das "grüne Knöpfchen" (Abb. 10.14) wird kompiliert und das Programm ausgeführt.



Abbildung 10.14: Kompilation und Programmstart

Im Ergebnis haben wir unseren ersten Qt Plot (Abb. 10.15), unheimlich - nicht wahr.

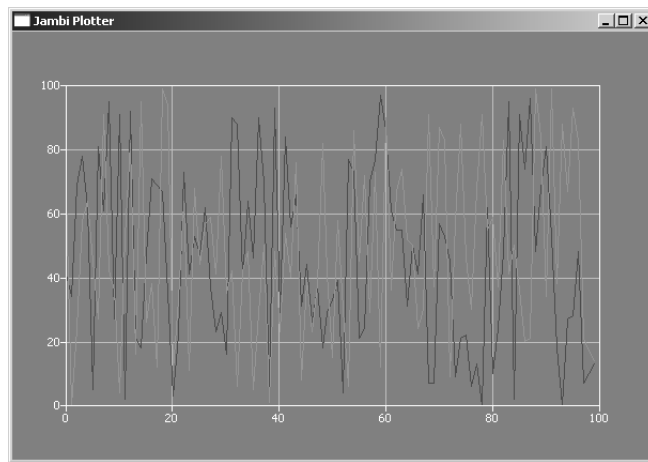


Abbildung 10.15: Qt Plotter

Jetzt schauen wir uns die Sache mal etwas genauer an.

Übung: E10.5

```
#include <QApplication>
#include "plotter.h"

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    // Data
    int numPoints = 100;
    QVector<QPointF> points0;
    QVector<QPointF> points1;
    for (int x = 0; x < numPoints; ++x) {
        points0.append(QPointF(x, uint(qrand()) % 100));
        points1.append(QPointF(x, uint(qrand()) % 100));
    }
}
```

```
}  
// Plotter  
Plotter plotter;  
plotter.setWindowTitle(QObject::tr("Jambi Plotter"));  
plotter.setCurveData(0, points0);  
plotter.setCurveData(1, points1);  
PlotSettings settings;  
settings.minX = 0.0;  
settings.maxX = 100.0;  
settings.minY = 0.0;  
settings.maxY = 100.0;  
plotter.setPlotSettings(settings);  
plotter.show();  
//  
return app.exec();  
}
```

Dies soll als kurze Einführung in Qt erstmal genügen, es geht weiter im nächsten Semester.

# Part III

## Anlagen / Software

## Kapitel 11

# Software-Installation

## 11.1 Glossar

**Compiler** ist ein Programm (z.B. `g++`), das eine Quelldatei in Maschinencode übersetzt.

**Quelldatei** ist ein File (z.B. `main.cpp`), das Quellcode enthält.

**Quellcode** sind sämtliche C++ Anweisungen.

**Maschinencode** ist ein ausführbares Program, z.B. `a.exe`.



## 11.2 cygwin Installation

In dieser Anlage finden sie eine Anleitung zur Installation von cygwin - dies ist eine Linux-Umgebung für Windows. In dieser Umgebung können sie die GNU C und C++ Compiler verwenden.

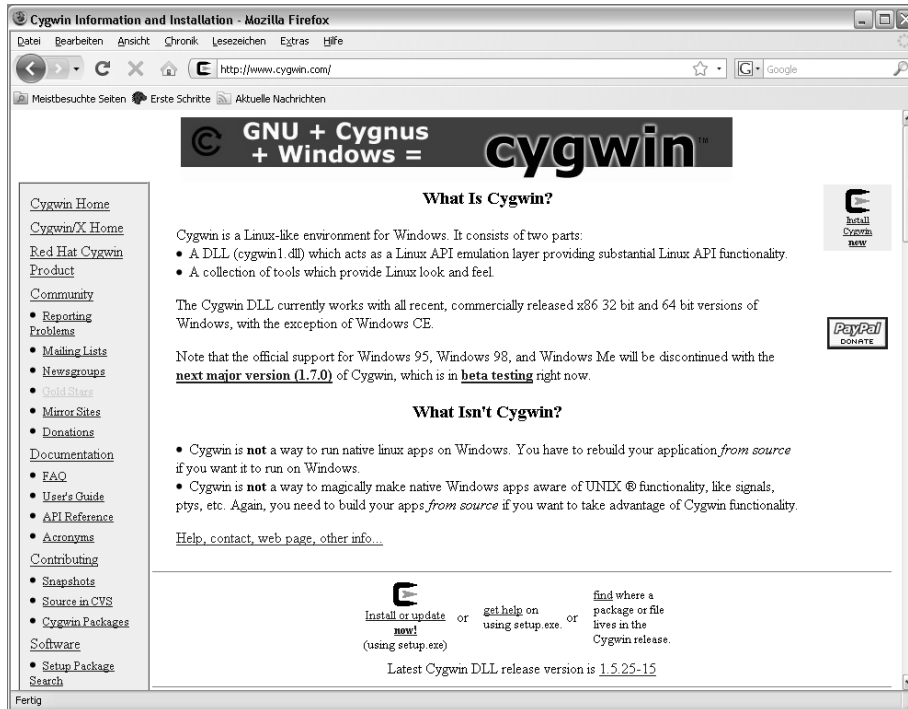


Abbildung 11.1: Schritt 1: Gehen sie auf cygwin Web Seite - www.cygwin.com. Klicken sie auf das cygwin Symbol (Install or update now!).



Abbildung 11.2: Schritt 2: Laden sie die setup.exe Datei auf ihren Rechner herunter und führen sie die setup.exe aus.



Abbildung 11.3: Schritt 3: Das cygwin setup Programm wird gestartet. Hier sehen sie auch die aktuelle Versionsnummer (2.573.2.3). Bestätigen sie mit Weiter.

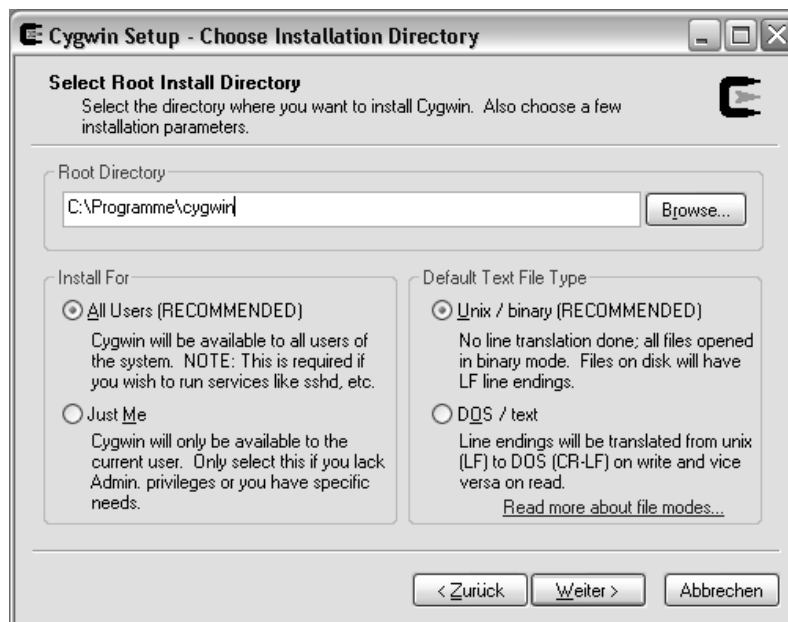


Abbildung 11.4: Schritt 4: Wählen sie das Verzeichnis zur Installation von cygwin aus, z.B. das Verzeichnis, in dem die Anwendungsprogramme sind (C:/Programme). Behalten sie die Voreinstellungen. Bestätigen sie mit Weiter.

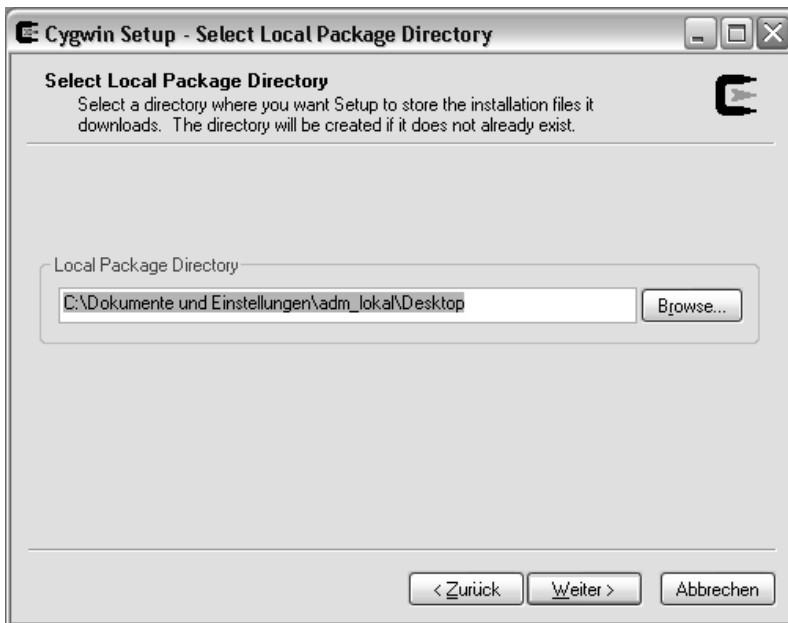


Abbildung 11.5: Schritt 5: cygwin benötigt ein temporäres Verzeichnis für die Installation, das später gelöscht werden kann. Behalten sie die Voreinstellungen. Bestätigen sie mit Weiter.

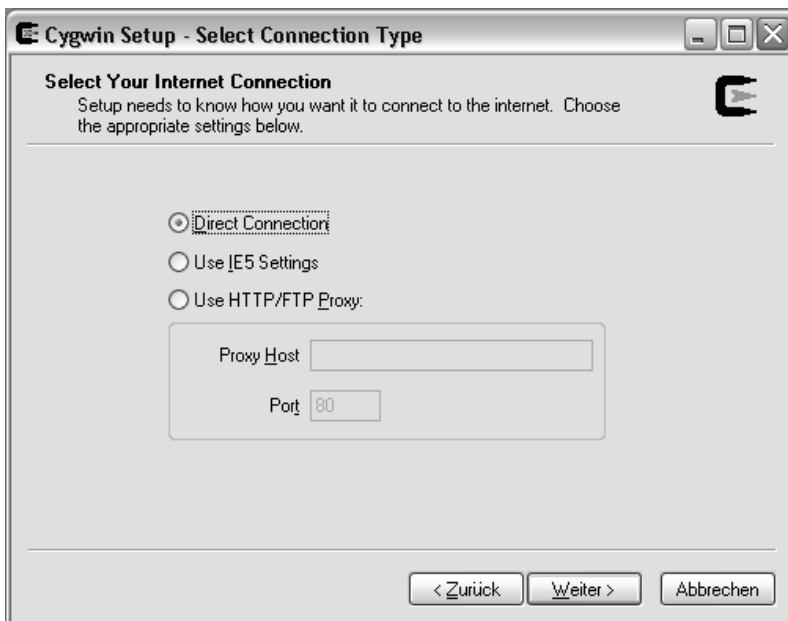


Abbildung 11.6: Schritt 6: Benutzen sie die voreingestellte direkte Internetverbindung. Bestätigen sie mit Weiter.

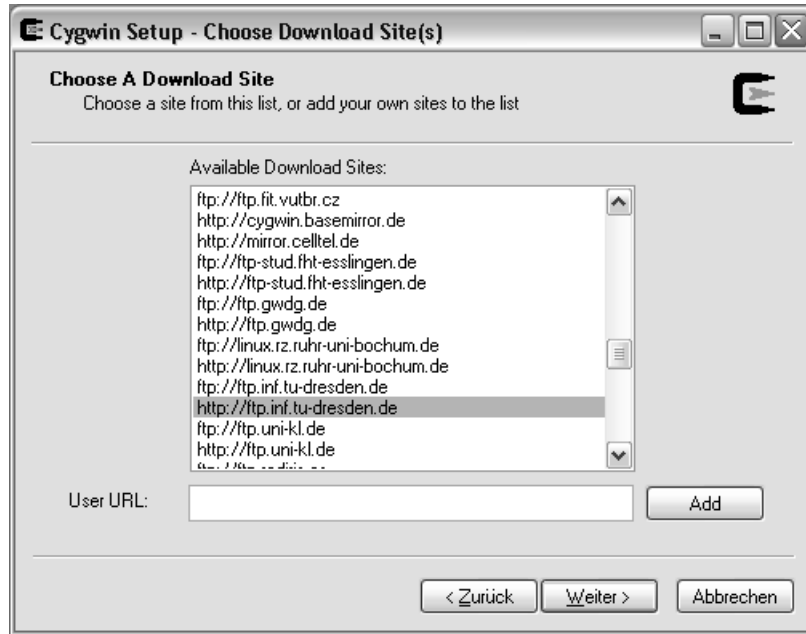


Abbildung 11.7: Schritt 7: Wählen sie die download site, am besten die Dresdner: `http://ftp.inf.tu-dresden.de`; Bestätigen sie mit Weiter.

Liste der 6 zu installierenden Pakete aus der Kategorie 'Devel':

- `binutils`: The GNU assembler linker and binary utilities (release 20080624-2) [6MB]
- `gcc-core`: C compiler (release 3.4.4.999) [3.6MB]
- `gcc-g++`: C++ compiler (release 3.4.4.999) [7.8MB]
- `gcc-mingw-core`: support headers and libraries for GCC (release 20050522-1) [69kB]
- `gcc-mingw-g++`: support headers and libraries for GCC C++ (release 20050522-1) [1.9MB]
- `mingw-runtime` library (release 3.15.2-1) [372kB]

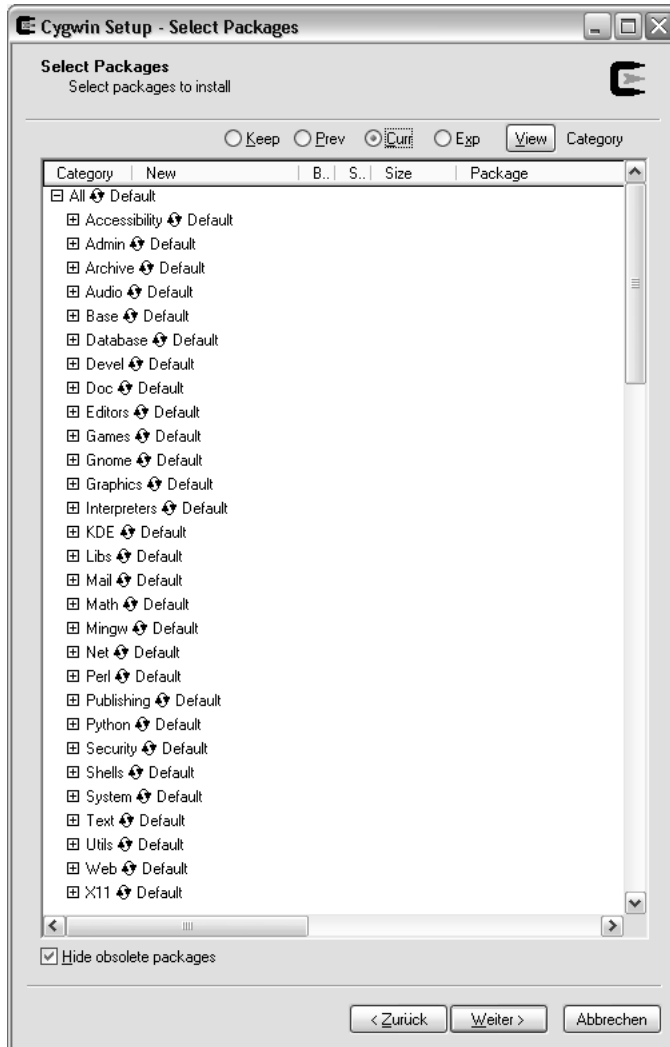


Abbildung 11.8: Schritt 8: Nun erscheint die Liste der packages, die installiert werden können. Öffnen sie dem Baum Devel (klicken Sie auf das Plus-Symbol). Wählen sie die oben aufgeführten Pakete aus, welche für die Installation von C und C++ benötigt werden. Bestätigen sie mit Weiter.

Die nachfolgenden Abbildungen zeigen ihnen, welche Markierungen aktiviert sein müssen, um den C (GCC) und den C++ (G++) zu installieren. Eigentlich reicht es, die `binutils`, `gcc-core` und `gcc-g++` auszuwählen, die restlichen Pakete (`mingw`) müssten automatisch aktiviert werden. Egal, überprüfen sie, ob alles wie in den nachfolgenden Abbildungen markiert ist. Viel Erfolg!

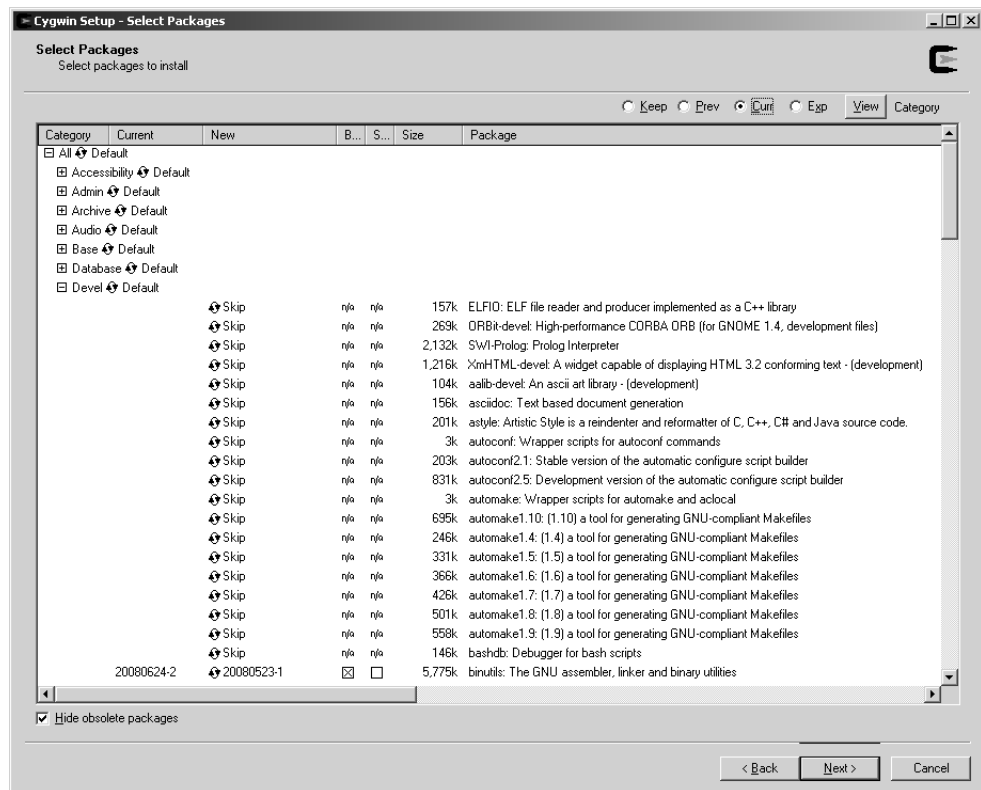


Abbildung 11.9: Schritt 8a: (aktive) Auswahl der binutils

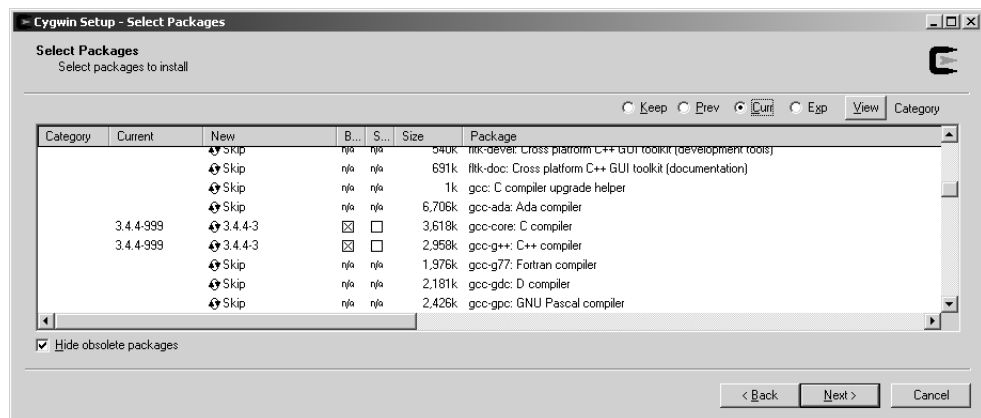


Abbildung 11.10: Schritt 8b: (aktive) Auswahl der C und C++ Compiler

Die nachfolgenden Pakete sollte automatisch markiert werden, bitte überprüfen sie dies sicherheitshalber.

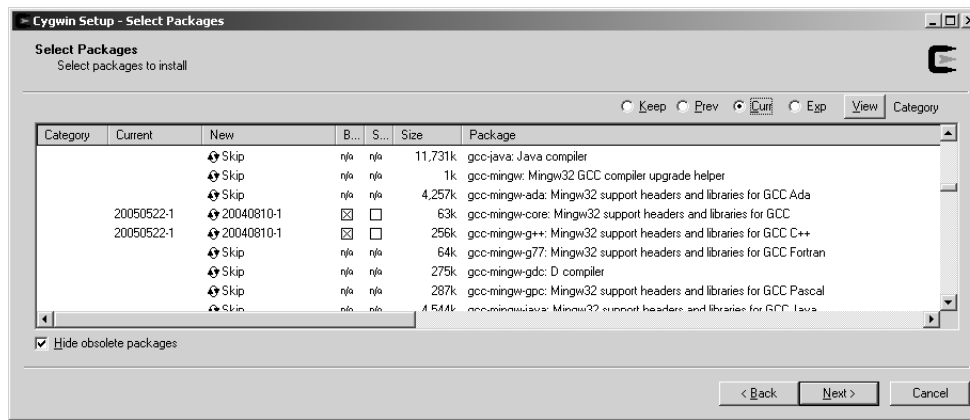


Abbildung 11.11: Schritt 8c: (automatische) Auswahl von Hilfspaketeten

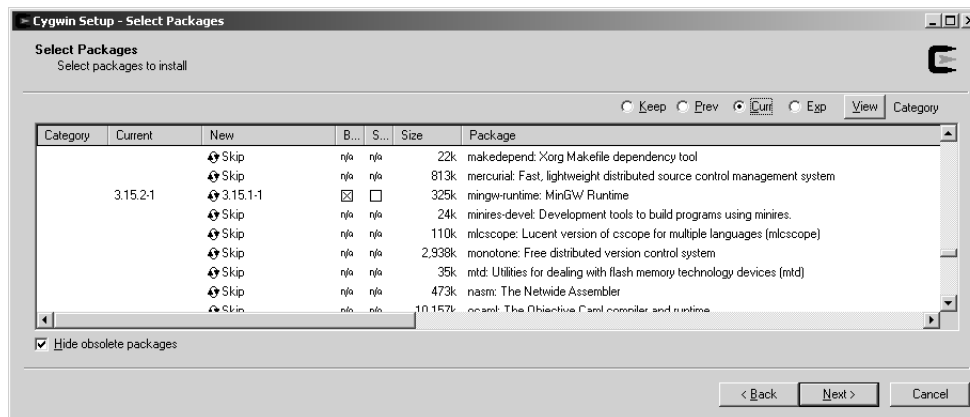


Abbildung 11.12: Schritt 8d: (automatische) Auswahl von Hilfspaketeten



Abbildung 11.13: Schritt 9: Behalten sie die Voreinstellungen, um eine cygwin Verknüpfung auf ihrem desktop zu erstellen. Schließen sie die Installation ab.

Im Ergebnis der Installation erhalten sie eine desktop Verknüpfung mit cygwin, das nun mit einem Doppelclick gestartet werden kann.

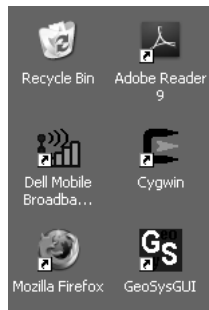


Abbildung 11.14: Schritt 8d: cygwin desktop Verknüpfung

Starten sie cygwin mit einem Doppelclick auf die desktop Verknüpfung.

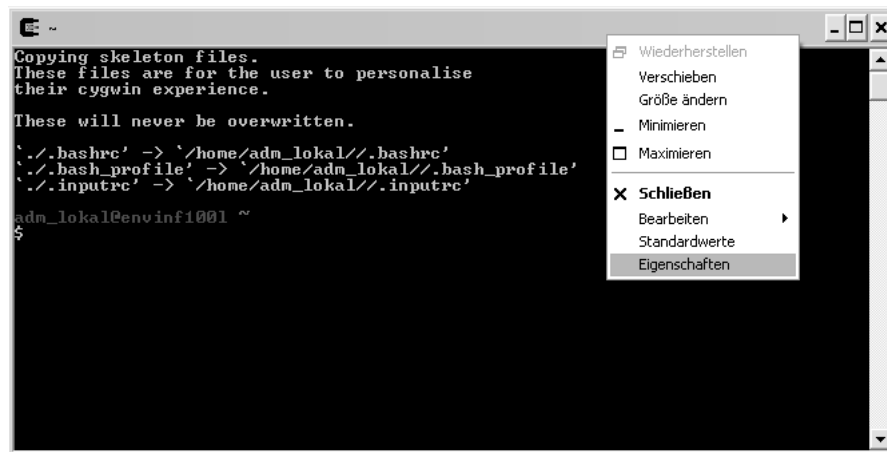


Abbildung 11.15: Start von cygwin

Mit einem Rechtsklick auf den oberen Balken gelangen sie in das Eigenschaften-Menu und können Einstellungen (z.B. Farben) nach ihrem Geschmack ändern.



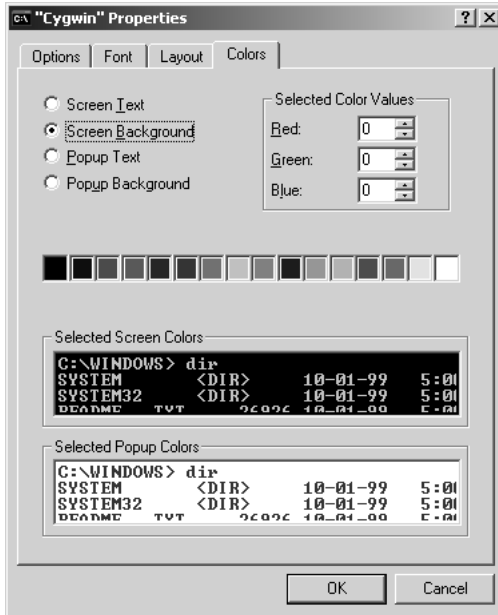


Abbildung 11.16: Das Eigenschafts-Menü von cygwin

Testen sie, ob die Compiler einsetzbar sind, indem sie diese einfach aufrufen:

- gcc - Aufruf des C Compilers
- g++ - Aufruf des C++ Compilers

Da sie die Compiler ohne etwas zu tun aufrufen, erhalten sie die Meldung: keine Eigabedateien.

```

adm_lokal@envinf1071 ~
$ gcc
gcc: no input files
adm_lokal@envinf1071 ~
$ g++
g++: no input files
adm_lokal@envinf1071 ~
$ -

```

Abbildung 11.17: Aufruf der Compiler

Nun kann es also endlich losgehen mit der ersten Übung ⇒ Abschnitt 1.4.

## 11.3 LINUX

### 11.3.1 Übersicht einiger grundlegenden Linux-Befehle

| Befehl / command                                     | Bedeutung  |
|--|--|
| <b>Hilfe</b>   |  |
| <code>man</code>                                     | Klassische Hilfe (verlassen mit Q)                   |
| <code>info</code>                                    | Online Hilfe von GNU (verlassen mit Q)               |
| <code>apropos</code>                                 | Schlüsselwörtersuche bei <code>man</code> -Seiten    |
| <code>whatis</code>                                  | Kurzbeschreibung zu einem Kommando / Schlüsselwort   |
| <b>Dateien</b>                                       |  |
| <code>ls</code>                                      | Dateien auflisten                                    |
| <code>ls -attr</code>                                | Dateiattribute auflisten                             |
| <code>file</code>                                    | Dateityp ermitteln                                   |
| <code>cp Quelle Ziel</code>                          | Kopiert Quelle nach Ziel                             |
| <code>mv Datei1 Datei2</code>                        | Datei1 umbenennen in Datei2                          |
| <code>mv Datei V1</code>                             | Verschiebt Datei in das Verzeichnis V1               |
| <code>rm Dateien</code>                              | Dateien löschen                                      |
| <code>rmdir Verzeichnis</code>                       | Verzeichnis löschen                                  |
| <code>find</code>                                    | Suche nach Dateien                                   |
| <code>find/</code>                                   | Suche startet im Root-Verzeichnis / im ganzen System |
| <b>Verzeichnisse</b>                                 |  |
| <code>pwd</code>                                     | gibt aktuelles Arbeitsverzeichnis an                 |
| <code>./</code>                                      | aktuelles Verzeichnis                                |
| <code>cd</code>                                      | Wechsel des aktuellen Verzeichnisses                 |
| <code>mkdir</code>                                   | ein Verzeichnis anlegen                              |
| <code>rm Verzeichnis</code>                          | ein Verzeichnis löschen                              |
| <code>rmdir</code>                                   | ein leeres Verzeichnis löschen                       |
| <code>ls</code>                                      | Verzeichnisinhalt auflisten                          |
| <b>Systembefehle</b>                                 |  |
| <code>shutdown -h</code>                             | Herunterfahren des Systems                           |
| <code>shutdown Uhrzeit</code>                        | Herunterfahren des Systems bei Uhrzeit (z.B.: 14.00) |
| <code>shutdown -t Sekunden</code>                    | Herunterfahren des Systems nach Anzahl der Sekunden  |
| <code>reboot</code> oder<br><code>shutdown -r</code> | Neustart des Systems                                 |
| <code>uname</code>                                   | Systeminformationen ausgeben                         |

Diese Übersicht basiert auf 'Linux auf einem Blatt' von Christian Helmbold ([www.helmbold.de](http://www.helmbold.de)) (siehe nächste Seite)

### weitere Informationen und Erklärungen

deutsche Übersichten:

- Pdf-Übersicht 'Linux auf einem Blatt' (auch zum Download)  
[www.helmbold.de/linux](http://www.helmbold.de/linux)

- Linux-Befehle für Einsteiger  
[http://www.linux-fuer-alle.de/doc\\_show.php?docid=33](http://www.linux-fuer-alle.de/doc_show.php?docid=33)

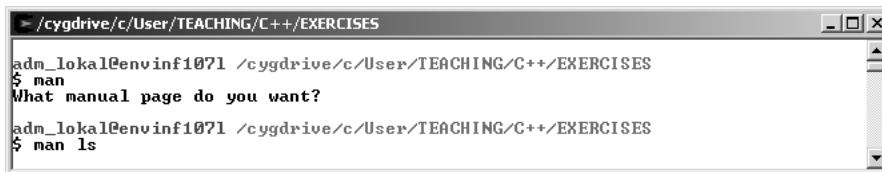
- Übersicht für wichtige Linux-Befehle und Programme  
<http://www.tnt-computer.de/yanip/lbefehle.html>

englische Übersichten

- umfangreiche Übersicht mit Parametern und Beispielen  
<http://www.computerhope.com/unix.htm#04>

- Vergleich zwischen MS-DOS- und Linux-Befehlen  
[http://www.yolinux.com/TUTORIALS/unix\\_for\\_dos\\_users.html](http://www.yolinux.com/TUTORIALS/unix_for_dos_users.html)

Wir arbeiten uns jetzt mal mit der online-Hilfe 'man' (steht für Manual) weiter vor.



```
adm_lokal@envinf1071 /cygdrive/c/User/TEACHING/C++/EXERCISES
$ man
What manual page do you want?
adm_lokal@envinf1071 /cygdrive/c/User/TEACHING/C++/EXERCISES
$ man ls
```

Abbildung 11.18: Benutzen der online Hilfe

Mit dem Befehl `man ls` können wir uns die Optionen von `ls` (Dateien auflisten) anzeigen lassen.

```

- /cygdrive/c/User/TEACHING/C++/EXERCISES
LS(1) User Commands LS(1)
NAME
  ls - list directory contents
SYNOPSIS
  ls [OPTION]... [FILE]...
DESCRIPTION
  List information about the FILES (the current directory by default).
  Sort entries alphabetically if none of -cftuwSUX nor --sort.
  Mandatory arguments to long options are mandatory for short options
  too.
  -a, --all
    do not ignore entries starting with .
  -A, --almost-all
    do not list implied . and ..
  --author
    with -l, print the author of each file
  -b, --escape
    print octal escapes for nongraphic characters
  --block-size=SIZE
    use SIZE-byte blocks
  -B, --ignore-backups
    do not list implied entries ending with ~
  -c
    with -lt: sort by, and show, ctime (time of last modification of
    file status information) with -l: show ctime and sort by name
    otherwise: sort by ctime
  -C
    list entries by columns
  --color[=WHEN]
    control whether color is used to distinguish file types. WHEN
    may be 'never', 'always', or 'auto'
  -d, --directory
    list directory entries instead of contents, and do not derefer-
    ence symbolic links
  -D, --dired
    generate output designed for Emacs' dired mode
  -f
    do not sort, enable -aU, disable -ls --color
  -F, --classify
    append indicator (one of */=>@!) to entries
  --file-type
    likewise, except do not append '*'
  --format=WORD
    across -x, commas -m, horizontal -x, long -l, single-column -l,
    verbose -l, vertical -C
  --full-time
    like -l --time-style=full-iso
  -g
    like -l, but do not list owner
  --group-directories-first
    group directories before files
  -G, --no-group
    in a long listing, don't print group names
  -h, --human-readable
    with -l, print sizes in human readable format (e.g., 1K 234M 2G)
  --si
    likewise, but use powers of 1000 not 1024
  -H, --dereference-command-line
    follow symbolic links listed on the command line
  --dereference-command-line-symlink-to-dir
  :

```

Abbildung 11.19: Optionen von ls (Dateien auflisten)

## 11.3.2 Profile

Wir nähern uns der Lösung, einfach in das Verzeichnis zu gelangen, in dem sich unsere Übungen befinden. Ähnlich wie in DOS die Programme autoexec.bat und config.sys beim Start des Betriebssystems automatisch ausgeführt werden, ist es bei LINUX ein so genanntes Profile: `.bash_profile`. In diese Datei können eigene Befehle eingetragen werden.

Eigentlich ist es ganz einfach ....

```
cd C:/User/TEACHING/C++/EXERCISES
```

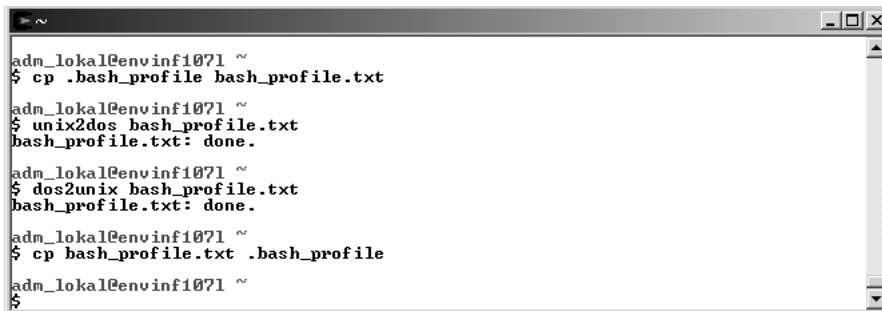
Durch diese Instruktion in der `.bash_profile` wechseln wir direkt in das Verzeichnis, in dem sich unsere Übungen befinden.

Nachdem wir verschiedene Editoren ausprobiert haben (und erhebliche Unterschiede in der Behandlung von Zeilenenden (CR) gesehen haben), ließ sich zu allem Übel unsere mühsam editierte `.bash_profile` mit dem Windows-Explorer nicht speichern (es liegt an dem Punkt am Anfang des Datei-Namens). Ich kann ihre Enttäuschung gut verstehen, nicht umsonst gibt es Windows und Linux-Anhänger. Dennoch müssen wir eine Lösung finden oder ?

### 11.3.2.1 Lösung 1: unix2dos

Nach unserem Schiffbruch mit dem Speichern einer `.bash_profile` unter Windows ('denkt' vor dem Punkt kommt der Datei-Name und nach dem Punkt die File-Extension, also nix vor dem Punkt heißt für Windows, kein Datei-Name ... so ist das, wenn Programme zu viel denken).

Die zweite Lösung ist eine Dateikonvertierung. Linux bietet zwei Programme an: `dos2unix` und `unix2dos`, mit denen man Dateien plattformgerecht konvertieren kann (Fig. 11.20)



```
adm_lokal@envinf1071 ~
$ cp .bash_profile bash_profile.txt
adm_lokal@envinf1071 ~
$ unix2dos bash_profile.txt
bash_profile.txt: done.
adm_lokal@envinf1071 ~
$ dos2unix bash_profile.txt
bash_profile.txt: done.
adm_lokal@envinf1071 ~
$ cp bash_profile.txt .bash_profile
adm_lokal@envinf1071 ~
$
```

Abbildung 11.20: Wandeln zwischen Linux und Dos-Dateien

### 11.3.2.2 Lösung 2: Cygwin (the Linux point of view)

Wir brauchen einen 'einfachen' Editor für cygwin, d.h. wir müssen nachinstallieren (siehe Abschn. )

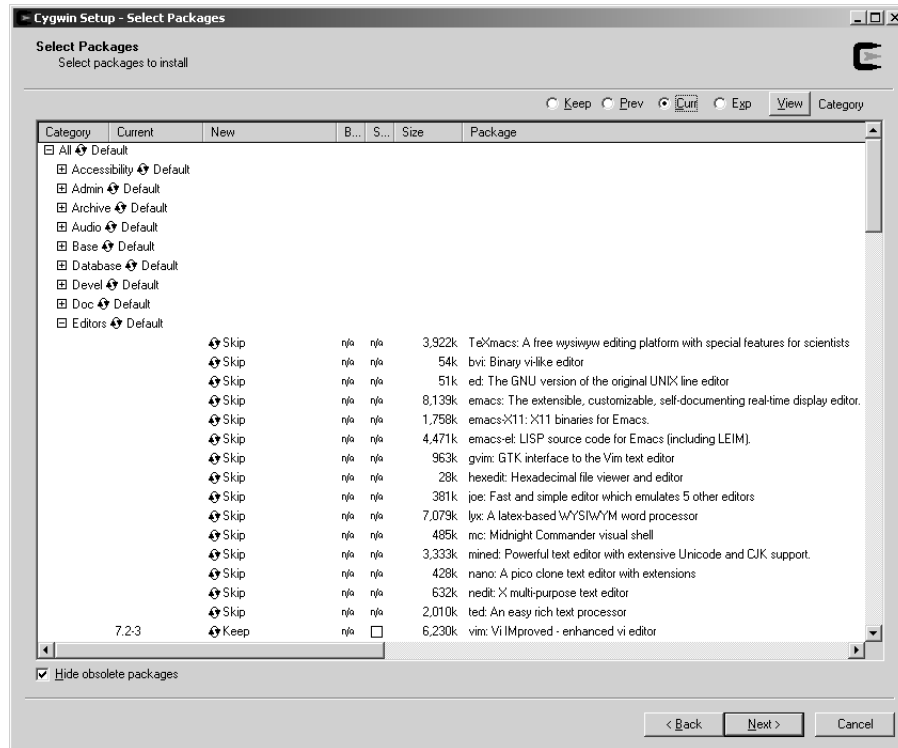


Abbildung 11.21: Nachinstallieren des vi Editors

Nach der vi Installation müssen wir in das Verzeichnis (home directory) gelangen, wo sich die `.bash_profile` Datei befindet.

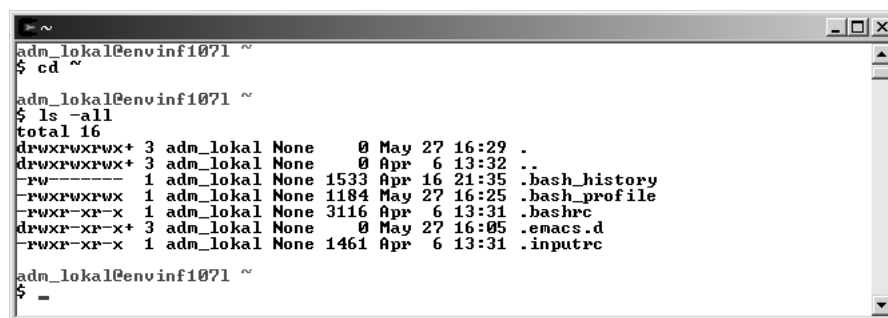


Abbildung 11.22: home directory und dessen Inhalt

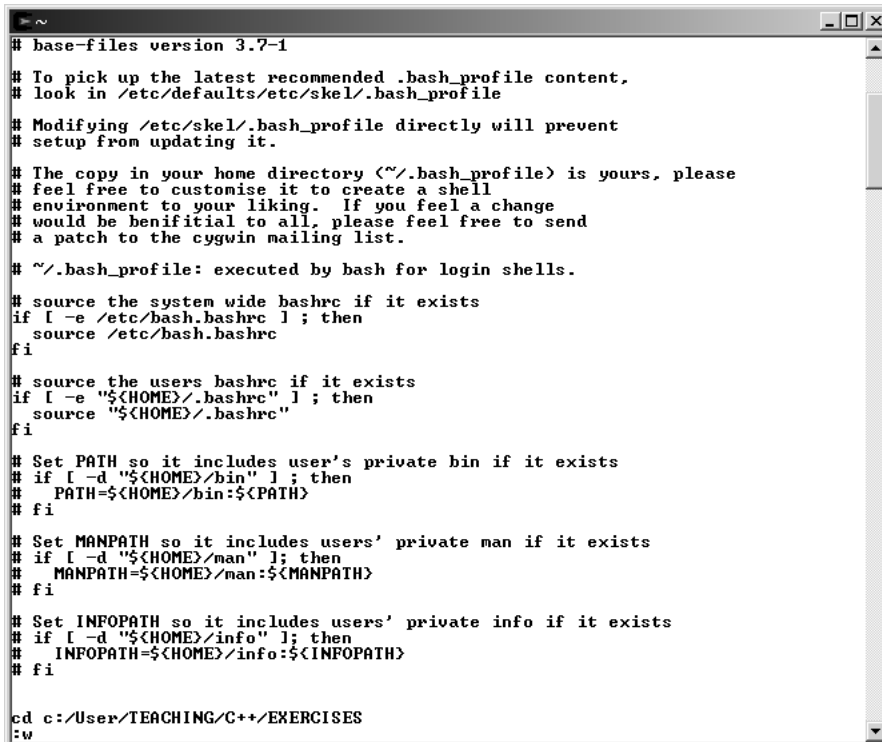
Wir öffnen die `.bash_profile` Datei mit dem vi Editor wie folgt:



```
adm_lokal@envinf1071 ~
$ vi .bash_profile
```

Abbildung 11.23: Starten des vi Editors

Jetzt können wir mal einen Linux-Editor live erleben (danach werden sie Windows noch mehr 'lieben' ...). Nach dem Öffnen der Datei gehen sie mit dem Cursor dorthin, wo sie Text einfügen möchten. Dann drücken sie die 'i' Taste, der insert Modus wird gestartet. Nun können sie den Befehl zum Wechseln in ihr Übungsverzeichnis eingeben `cd ...` Um die Änderungen zu speichern, drücken sie 'Esc' ':' 'w'. vi wechselt nun in den File-Modus 'w' steht für write ... so einfach ist das.



```
# base-files version 3.7-1
# To pick up the latest recommended .bash_profile content,
# look in /etc/defaults/etc/skel/.bash_profile
# Modifying /etc/skel/.bash_profile directly will prevent
# setup from updating it.
# The copy in your home directory (~/.bash_profile) is yours, please
# feel free to customise it to create a shell
# environment to your liking. If you feel a change
# would be beneficial to all, please feel free to send
# a patch to the cygwin mailing list.
# ~/.bash_profile: executed by bash for login shells.
# source the system wide bashrc if it exists
if [ -e /etc/bash.bashrc ] ; then
  source /etc/bash.bashrc
fi
# source the users bashrc if it exists
if [ -e "${HOME}/.bashrc" ] ; then
  source "${HOME}/.bashrc"
fi
# Set PATH so it includes user's private bin if it exists
if [ -d "${HOME}/bin" ] ; then
  PATH=${HOME}/bin:${PATH}
fi
# Set MANPATH so it includes users' private man if it exists
if [ -d "${HOME}/man" ] ; then
  MANPATH=${HOME}/man:${MANPATH}
fi
# Set INFOPATH so it includes users' private info if it exists
if [ -d "${HOME}/info" ] ; then
  INFOPATH=${HOME}/info:${INFOPATH}
fi
cd c:/User/TEACHING/C++/EXERCISES
:w
```

Abbildung 11.24: Editieren und speichern

| Befehl | Wirkung   |
|--------|---|
| i      | Wechseln in den Insert/Einfüge-Modus                    |
| a      | Wechseln in den Append/Anhängen-Modus                   |
| Esc:   | Wechseln in den Kommando-Modus                          |
| w      | Write: Datei speichern (im Kommando-Modus)              |
| q      | Quit: Datei ohne Änderung verlassen (im Kommando-Modus) |
| q!     | Quit: Datei ohne Änderung verlassen (im Kommando-Modus) |

### 11.3.3 Make

Mit den Make utilities können ganze Projekte kompiliert werden, die aus vielen Quelltext-Dateien bestehen. Nachfolgend sehen sie die Instruktionen des makefile für die Kompilierung unseres Projekts. Dabei werden die Quell-Dateien main.cpp und student.cpp kompiliert, gelinkt und die ausführbare Datei a.exe erzeugt.

```
#
OPTFLAG = -O3
C++      = g++ $(OPTFLAG) -Wall
CFile = main.cpp student.cpp
OFile = main.o student.o
.SUFFIXES: .o .cpp .h
.cpp.o:
$(C++) -c -o $*.o  $<
main: $(OFile)
$(C++) -o main $(OFile)
clean:
rm -f *.o
```



## 11.4 Qt Installation

The image shows two screenshots of the Qt Developer website in a Mozilla Firefox browser window.

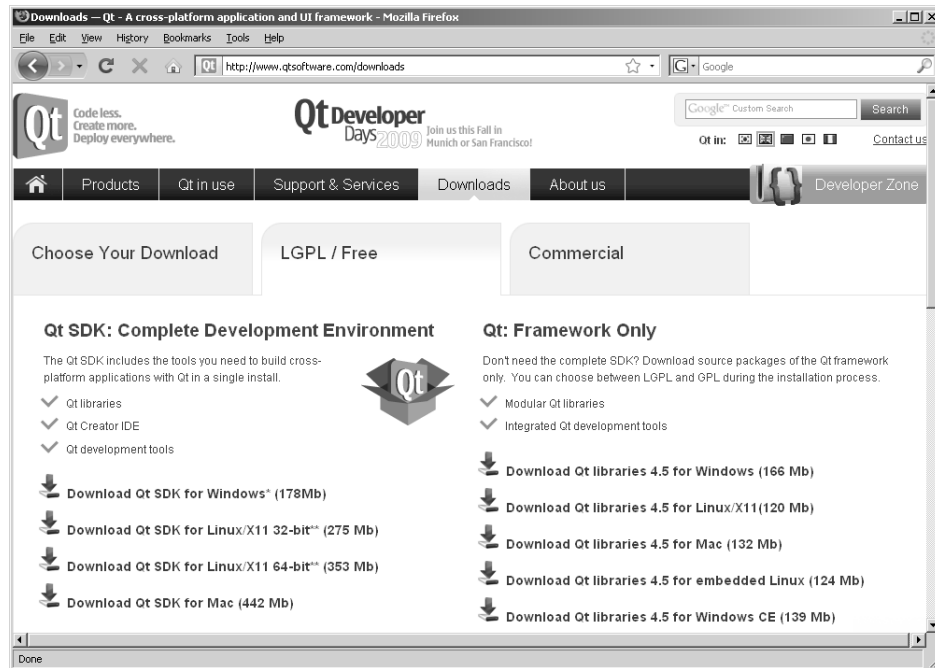
**Top Screenshot: Products Page**  
 The browser address bar shows `http://www.qtsoftware.com/products/`. The page features the Qt logo with the tagline "Code less. Create more. Deploy everywhere." and the "Qt Developer Days 2009" logo. A navigation menu includes "Products", "Qt in use", "Support & Services", "Downloads", and "About us". The main content area is titled "Products" and states: "Qt enables UI and application developers to create a better user experience. Qt is a cross-platform application and UI framework. Using Qt, you can write applications once and deploy them across many desktop and embedded operating systems without rewriting the source code." It lists features such as "Intuitive C++ class library", "Portability across desktop and embedded operating systems", "Integrated development tools with cross-platform IDE", and "High runtime performance and small footprint on embedded". A central box highlights that the Qt SDK includes tools for building cross-platform applications, listing "Qt Creator cross-platform IDE", "Qt libraries", and "Qt Linguist translation & internationalization tools". Below this are two download buttons: "Download Free Qt SDK" and "Download Qt Sources Only". A right-hand sidebar contains a "Navigation" menu with links like "Class Library", "Developer Tools", "Platforms", and "Programming Language Support", along with a "Resources" section for whitepapers.

**Bottom Screenshot: Downloads Page**  
 The browser address bar shows `http://www.qtsoftware.com/downloads`. The page is titled "Choose Your Download" and offers two options: "LGPL / Free" and "Commercial". A summary table compares the two licenses:

|  | LGPL / Free | Commercial |
|--|-------------|------------|
| Charge for development licenses                            | ✗           | ✓          |
| Changes to Qt source code must be shared                   | ✓           | ✗          |
| Can create proprietary application                         | ✓           | ✓          |
| Technical support available ( <a href="#">Learn more</a> ) | ✓           | ✓          |

Below the table are two buttons: "Choose LGPL / Free" and "Choose Commercial". A note at the bottom states: "The above table is a summary only. For details on licensing, visit the Qt licensing page."

Zur Qt Installation gehen sie auf folgende Web-Seite und folgen den Schritten gemäß den Abbildungen: <http://www.qtsoftware.com/products/>. Wählen die freie LGPL Lizenz.



Dabei wird die Installationsdatei für das entsprechende Betriebssystem heruntergeladen (Abb. 11.25).



Abbildung 11.25: Qt Installationsdatei für Windows

Nach der Installation haben sie den Qt Creator zur Verfügung (Abb. 11.26).



Abbildung 11.26: Qt Creator

# Kapitel 12

## Questions

### 12.1 Testfragen - Kapitel 1 - Einführung

1. Was bedeutet das ++ im Namen der Programmiersprache C++ ?
2. Ist C++ eine standardisierte Programmiersprache ?
3. Was ist der Unterschied zwischen C und C++ ?
4. Was sind sogenannte Programmier-Paradigmen ?
5. Welche verschiedenen Programmier-Paradigmen kennen sie und worin unterscheiden sie sich ?
6. Welches Paradigma verfolgen wir in der Vorlesung ?
7. Was ist objekt-orientierte Programmierung ?
8. Was ist ein Kompiler ?
9. Erklären sie die einzelnen Schritte bei einer Kompilierung (s. Abb. 1.4).
10. Welchen Kompiler benutzen sie für die C++ Übungen ?
11. Welche Funktion benötigt jedes C oder C++ Programm ?
12. Was ist der Typ einer Funktion ?
13. Was ist eine Parameterliste einer Funktion ?
14. Was ist der Rückgabewert einer Funktion ?

## 12.2 Testfragen - Kapitel 2 - Datentypen

1. Was ist der genaueste Datentyp in C++ ?
2. Wie groß ist der Speicherbedarf von einem string Datentyp ?
3. Mit welcher Anweisung können wir den Speicherbedarf von elementaren Datentypen ermitteln ?
4. Was sind Escape-Sequenzen ?
5. Was ist der Unterschied zwischen den Escape-Sequenzen `\n` und `\r` ?
6. Was ist die C++ Entsprechung der Klasse `cout` für einen Zeilenumbruch ?

## 12.3 Testfragen - Kapitel 3 - Ein- und Ausgabe I

1. Sind sie mit der Tab. 2.2 einverstanden ?
2. Welche Ein- und Ausgabegeräte kennen sie ?
3. Welche Klasse benötigen wir für die Standard-Ein- und Ausgabe ?
4. Was ist die Basis-Klasse für alle Ein- und Ausgaben in C++ ?
5. Mit welcher Klasse können wir sowohl Eingabe- als auch Ausgabeströme benutzen ?
6. Welche Include-Datei ist notwendig, um mit I/O-Strömen arbeiten zu können ?
7. Wozu dient der Zusatz `using namespace std;` nach dem Include von Standard-Klassen, wie I/O Streams ?
8. Was bewirken die Stream-Operatoren `<<` und `>>` ?
9. Was bewirken die Flags `oct`, `dec`, `hex` für die formatierte Ausgabe von Ganzzahlen ? (ToDo)
10. Wie kann ich die Genauigkeit der Ausgabe von Gleitkomma-Zahlen festlegen ?
11. Wie kann man den Speicherbedarf einer Variable ermitteln ? Schreiben sie die C++ Anweisung für die Berechnung des Speicherbedarfs für eine doppelt-genaue Gleitkomma-Zahl.

## 12.4 Testfragen - Kapitel 4 - Klassen

1. Geben sie eine Definition von C++ Klassen mit eigenen Worten (max 5 Sätze).
2. Was ist ein benutzerdefinierter Datentyp ?
3. Welches Datennschutz-Konzept gibt es für Klassen ?
4. Wozu brauchen wir zwei Dateien für Klassen, eine H (Header) Datei und eine CPP (Quelltext) Datei ?
5. Was ist ein Inklude / Include ?
6. Was ist eine Instanz einer Klasse ?
7. Worin besteht der Unterschied zwischen den Anweisungen: `CStudent m_std_1` und `CStudent* m_std_2` ?
8. Was ist ein Konstruktor einer Klasse ?
9. Was ist das Gegenstück zum Klassen-Konstruktor ?
10. Wie können Daten / Variablen einer Klasse initialisiert werden ?
11. Schreiben sie den Quelltext für den Klassen-Konstruktor und weisen sie den Variablen `name_first` und `name_last` ihren eigenen Namen zu.
12. Was verbirgt sich hinter der Anweisung: `CStudent* m_std = new CStudent()` ?
13. Was ist der Unterschied zwischen `CStudent` und `CStudent()` ?
14. Wie kann ich meine Daten gegen einen externen Zugriff schützen ?

## 12.5 Testfragen - Kapitel 5 - Strings

1. Welche Klasse bietet uns C++ zur Verarbeitung von Zeichenketten an ?
2. Welchen Include benötigen wir für die Arbeit mit Zeichenketten ?
3. Wofür ist die Anweisung `using namespace std` nützlich ?
4. Müssen wir selber Speicherplatz für C++ Zeichenketten (Strings) reservieren ?
5. Wie können wir einen String, sagen wir mit der Zeichenkette "Das ist eine gute Frage", initialisieren ?
6. Wie können wir zwei Strings, ihren Vor- und Nachnahmen, miteinander verbinden ?

7. Mit welcher string Funktion kann ich Zeichenketten miteinander vergleichen ?
8. Schreiben sie eine Anweisung zum Vergleich der Klassen-Variable `m_std->name_last` mit ihrem Nachnamen ?
9. Mit welcher string Funktion kann ich Zeichenketten suchen ?
10. Schreiben sie eine Anweisung zum Suchen ihres Vornamens in der Klassen-Variable `m_std->name_first` ?
11. Mit welcher string Funktion kann ich Teile in Zeichenketten erstetzen ?
12. Wie können wir die Länge der in einem Objekt der Klasse `std::string` verwalteteten Zeichenkette ermitteln ?
13. Schreiben sie die Anweisungen, um ihren Nachnamen in die Zeichenkette "JAMES BOND" nach "JAMES" einzufügen ?
14. Was passiert, wenn ihr Nachname länger als "BOND" ist ?
15. Mit welcher string Funktion können wir Zeichen in einem String löschen ?
16. Wir haben gesehen, dass es verschiedene Daten-Typen für Zeichenketten in C, C++, MFC, .NET und Qt gibt. Zeichenketten gehören zu den wichtigsten Daten-Typen bei der Programmierung. Wie können wir einen C++ string in eine C Zeichenkette (char) umwandeln ?
17. Können wir eine .NET Zeichenkette (String^) in eine C++ Zeichenkette (string) umwandeln ?
18. Was ist ein `stringstream` ?
19. Welchen Include benötigen wir für die Arbeit mit Stringstreams ?

## 12.6 Testfragen - Kapitel 6 - Ein- und Ausgabe II

1. Was ist die Basis-Klasse für alle Ein- und Ausgaben in C++ ?
2. Was sind die C++ Klassen für das Lesen und Schreiben von Dateien ?
3. Welchen Include benötigen wir für das Arbeiten mit I/O File-Klassen ?
4. Was sind die Standard-Flags für File-Streams (Lesen und Schreiben) ?
5. Mit welchem Flag können wir zu schreibende Daten an eine existierende Datei anhängen ?
6. Was ist der Unterschied zwischen ASCII- und Binär-Formaten ?

7. Mit welchem Flag können wir Daten in einem Binär-Format schreiben ?  
Mit welcher Anweisung wird ein File geöffnet ? Mit welcher Anweisung wird ein File geschlossen ?
8. Was bewirken die Stream-Operatoren << und >> ?
9. Wie können wir mit Dateinamen in unserem Hauptprogramm `main(...)` arbeiten ?
10. Welche Anweisung benötigen wir für die Erzeugung einer Instanz für einen Eingabe-Strom ?
11. Welche Anweisung benötigen wir für die Erzeugung einer Instanz für einen Ausgabe-Strom ?
12. Für die Erstellung einer Datenbank ist es wichtig einzelnen Datensätze zu trennen. Wie können wir soetwas in der Datenbank-Datei bewerkstelligen ?
13. Ist es wichtig das Ende einer Datenbank-Datei, z.B. mit einem Schlüsselwort `#STOP`, zu markieren ?
14. Mit welcher Abfrage könne wir prüfen, ob die Öffnung einer Datei erfolgreich war ?
15. Mit welcher Anweisung können wir die aktuell gelesene Position in einer geöffneten Datei abfragen ?
16. Mit welcher Anweisung können wir zu einer bestimmten Position in einer geöffneten Datei springen ?
17. Mit welcher Anweisung können wir eine komplette Zeile aus geöffneten Datei auslesen ?

## 12.7 Testfragen - Kapitel 7 - Referenzen und Zeiger

1. Was ist `&x` ? (`x` ist ein beliebiger Datenobjekt `T`, z.B. eine Gleitkommazahl: `double x`)
2. Was ist eine Referenz `&ref`?
3. Was bedeutet die Anweisung: `&ref = x`?
4. Erklären sie kurz (mit eigenen Worten) das Zeiger-Konzept in C++?
5. Was bewirkt der NULL Zeiger, d.h. `*ptr = NULL` ?
6. Was bedeutet die Anweisung: `ptr = x`?

7. Müssen Referenzen (&) und Zeiger (\*) auf Daten-Objekte initialisiert werden ?
8. Was bewirkt die Definition `long l[1000]` speichertechnisch ?
9. Wie groß ist der Speicherbedarf des statischen Datenobjekts `long l[1000]` ?
10. Wie groß ist der Speicherbedarf des dynamische Datenobjekts `long* ptr_l = new long[1000]` ?
11. Woher kommt der Unterschied (4 Byte auf einem 32Bit Rechner) im Speicherbedarf zwischen statischen und dynamischen Datenobjekten.
12. Zusatzfrage: Was bedeutet die Definition `**ptrptr` (Zeiger auf Zeiger), was für ein Datenkonstrukt entsteht ?

## 12.8 Testfragen - Kapitel 8 - Container

1. Was sind C++ Container ?
2. Welche Typen von C++ Containern kennen sie ?
3. Worin besteht der Unterschied zwischen sequentiellen und assoziativen Containern ?
4. Welche sequentiellen Container kennen sie ?
5. Erklären sie die Syntax des Vektor-Containers: `vector<int>my_vector` .
6. Was ist der Unterschied zwischen Vektoren und Listen ?
7. Was sind die Gemeinsamkeiten von Vektoren und Listen ?
8. Welcher Include ist notwendig für das Arbeiten mit Vektoren ?
9. Welcher Include ist notwendig für das Arbeiten mit Listen ?
10. Benötigen wir den Zusatz (Namensraum) `using namespace std`, wenn ja warum ?
11. Mit welcher Instruktion können Elemente in Vektoren und Listen einfügen ? Nennen sie mindestens zwei Möglichkeiten.
12. Wo werden sequentielle Container-Elemente mit der Instruktion `push_back(T)` eingefügt ?
13. Mit welcher Anweisung können wir die Länge von sequentiellen Container-Elementen bestimmen ?



14. Mit welcher Anweisung können wir einen Vektor platt machen (d.h. alle Elemente löschen) ?
15. Wie können wir auf ein Vektor-Element, sagen wir das 17te Element des Vektors `vector<int>my_vector`, direkt zugreifen ?
16. Quellcode verstehen: Erklären sie die Struktur (1,2,3) der DB-Lese-Funktion `STDRead(ifstream& std_file)` in der Übung 8.2.2. Beginnen sie mit der Parameterliste.
17. Wie können wir unsere Studenten-Klasse `CStudent` in die DB-Anwendung (Übung 8.2.1) einbinden ?
18. Was ist eigentliche Lesefunktion für unsere Studenten-Datensätze (Übung 8.2.2) ?
19. Mit welchem Befehl können wir die Reihenfolge von Listen-Elementen umkehren ?
20. Können wir Listen-Elemente sortieren, wenn ja wie, mit welcher Instruktion ?
21. Mit welchem Befehl können wir mehrere Listen zusammenführen ?
22. Können wir doppelte Elemente aus einer Liste entfernen sortieren, wenn ja wie, mit welcher Instruktion ?
23. Was ist ein Iterator ?
24. Quellcode verstehen: Erklären sie die Funktion `void display(list<int>my_list)` der Übung 8.3. Beginnen sie mit der Parameterliste.
25. Wie können wir Elemente aus einer Liste entfernen ? Nennen sie mindestens zwei Möglichkeiten.

## 12.9 Testfragen - Kapitel 9 - Andere Sprachelemente

1. Was sind Kontrollstrukturen, welche kennen sie ?
2. Bei welcher logischen Bedingung wird eine `if(...)`-Anweisung ausgeführt ?
3. Lassen sich Kontroll-Strukturen verschachteln ?
4. Mit welcher Kontrollstruktur können wir Fallunterscheidungen programmieren ?

5. Welche Ausdrücke können wir bei der `switch-case` Kontrollstruktur benutzen ?
6. Wie kann ich eine Compiler-Direktive an- und ausschalten ?
7. Schreiben sie die Übung 9.2.2.2 für die Benutzung von `#ifndef` anstelle von `#ifdef` um.
8. Erläutern sie den Kopf der for-Schleife: `for(int i=0;i<stop;i++)`, welchen Daten-Typ hat `stop` ?
9. Was ist eine Endlos-Schleife, wie kommen wir daraus ?
10. Was sind Namensbereiche ?
11. Was müssen wir tun, um die Arbeit mit C++ Standard-Klassen zu vereinfachen, d.h anstelle von `std::my_string()` direkt `my_string()` benutzen zu können ?
12. Was sind Compiler-Direktiven, was bewirken diese ?
13. Wie können wir Compiler-Direktiven an- und ausschalten ?
14. Worin besteht der Unterschied bei Includes mit eckigen Klammern `< ... >` bzw. mit Gänsefüßchen `"..."` ?
15. Schreiben sie die Übung 9.2.2.2 für die Benutzung von `#ifndef` anstelle von `#ifdef` um ?
16. Was sind Makros ?
17. Welche beiden Einsatzmöglichkeiten von Makros gibt es ?
18. Worin besteht die Gefahr, Makros als Funktionsersatz zu benutzen ?

# Literaturverzeichnis

- [1] Stroustrup B. *The programming languages C++*. Addison-Wesley, Reading, 1991.
- [2] Blanchette J and Summerfield M. *C++ GUI Programming with Qt 4*. Prentice Hall, Boston, 2006.
- [3] Kolditz O. *Computational methods in environmental fluid mechanics*. Springer, Berlin-Heidelberg, 2002.
- [4] Breymann U. *C++ Einführung und professionelle Programmierung*. Hanser, München-Wien, 2001.
- [5] Kirch-Prinz U and Prinz P. *C++ Lernen und professionell anwenden*. mitp, Heidelberg, 2007.

# Inhaltsverzeichnis

|  |           |
|--|-----------|
| <b>Part I - C++ Basics</b>                         | <b>6</b>  |
| <b>1 Einführung</b>                                | <b>7</b>  |
| 1.1 Historisches . . . . .                         | 8         |
| 1.2 Paradigmen . . . . .                           | 9         |
| 1.3 Compiler . . . . .                             | 11        |
| 1.3.1 GNU . . . . .                                | 12        |
| 1.3.2 MS Visual C++ . . . . .                      | 13        |
| 1.3.3 Qt . . . . .                                 | 13        |
| 1.4 "Hello World" . . . . .                        | 14        |
| 1.4.1 Grundlagen . . . . .                         | 14        |
| 1.4.2 Exercise E1 . . . . .                        | 15        |
| 1.5 Students Forum . . . . .                       | 16        |
| 1.6 Testfragen . . . . .                           | 17        |
| <b>2 Datentypen</b>                                | <b>19</b> |
| 2.1 Elementare Datentypen . . . . .                | 19        |
| 2.2 Speicherbedarf . . . . .                       | 20        |
| 2.3 Escape-Sequenzen . . . . .                     | 20        |
| 2.4 Testfragen . . . . .                           | 21        |
| <b>3 Ein- und Ausgabe</b>                          | <b>22</b> |
| 3.1 Die Standard-Streams . . . . .                 | 24        |
| 3.2 Formatierte Ausgaben . . . . .                 | 24        |
| 3.2.1 Formatierte Ausgabe von Ganzzahlen . . . . . | 24        |

---

|          |  |           |
|----------|--|-----------|
| 3.2.2    | Formatierte Ausgabe von Gleitpunktzahlen . . . . . | 25        |
| 3.2.3    | Ausgabe von Speicherbedarf . . . . .               | 25        |
| 3.3      | Testfragen . . . . .                               | 26        |
| <b>4</b> | <b>Klassen</b>                                     | <b>27</b> |
| 4.1      | Daten-Abstraktion . . . . .                        | 28        |
| 4.2      | Klassen-Deklaration . . . . .                      | 29        |
| 4.3      | Instanzen einer Klasse . . . . .                   | 29        |
| 4.4      | Konstruktor und Destruktor . . . . .               | 31        |
| 4.5      | Dateninitialisierung mit dem Konstruktor . . . . . | 32        |
| 4.6      | Datenschutz . . . . .                              | 33        |
| 4.7      | Testfragen . . . . .                               | 35        |
| <b>5</b> | <b>Strings</b>                                     | <b>36</b> |
| 5.1      | Die Standardklasse string . . . . .                | 36        |
| 5.2      | Operationen mit strings . . . . .                  | 37        |
| 5.2.1    | Initialisieren von strings . . . . .               | 37        |
| 5.2.2    | Zuweisen von strings . . . . .                     | 38        |
| 5.2.3    | Verketteten von strings . . . . .                  | 38        |
| 5.2.4    | Vergleichen von strings . . . . .                  | 38        |
| 5.2.5    | Suchen in strings . . . . .                        | 39        |
| 5.2.6    | Einfügen in strings . . . . .                      | 39        |
| 5.2.7    | Ersetzen in strings . . . . .                      | 40        |
| 5.2.8    | Löschen in strings . . . . .                       | 40        |
| 5.2.9    | Umwandeln von strings in char . . . . .            | 41        |
| 5.2.10   | Auswerten von Strings: Stringstreams . . . . .     | 41        |
| 5.3      | Testfragen . . . . .                               | 42        |
| <b>6</b> | <b>Ein- und Ausgabe - II</b>                       | <b>43</b> |
| 6.1      | Die fstream Klassen . . . . .                      | 43        |
| 6.2      | Arbeiten mit File-Streams . . . . .                | 44        |
| 6.2.1    | File-Streams anlegen . . . . .                     | 44        |
| 6.2.2    | File-Streams schließen . . . . .                   | 44        |
| 6.2.3    | Übung: Eine einfache Kopierfunktion . . . . .      | 44        |

|          |  |           |
|----------|--|-----------|
| 6.2.4    | Übung: Ein einfacher Konverter . . . . .                 | 46        |
| 6.2.5    | Einschub: Erstes Arbeiten mit MS VC++ . . . . .          | 47        |
| 6.3      | File-Streams und Klassen . . . . .                       | 50        |
| 6.4      | fstream Methoden . . . . .                               | 53        |
| 6.5      | Testfragen . . . . .                                     | 54        |
| <b>7</b> | <b>Referenzen und Zeiger</b>                             | <b>55</b> |
| 7.1      | Referenzen . . . . .                                     | 56        |
| 7.1.1    | Das Call-by-reference Prinzip - Referenzen als Parameter | 56        |
| 7.1.2    | Referenzen als Rückgabe-Wert . . . . .                   | 57        |
| 7.2      | Zeiger . . . . .   | 57        |
| 7.2.1    | Definition von Zeigern . . . . .                         | 58        |
| 7.2.2    | NULL Zeiger . . . . .                                    | 58        |
| 7.2.3    | Zeiger als Parameter . . . . .                           | 59        |
| 7.3      | Zeiger und Arrays . . . . .                              | 59        |
| 7.3.1    | Statische Objekte . . . . .                              | 59        |
| 7.3.2    | Dynamische Objekte . . . . .                             | 59        |
| 7.4      | Summary . . . . .  | 60        |
| 7.5      | Testfragen . . . . .                                     | 62        |
| <b>8</b> | <b>Container</b>   | <b>63</b> |
| 8.1      | Sequentielle Container . . . . .                         | 64        |
| 8.2      | Vectors . . . . .  | 65        |
| 8.2.1    | Defining vectors . . . . .                               | 67        |
| 8.2.2    | Vectors and data base . . . . .                          | 67        |
| 8.2.3    | Updating your data base entry . . . . .                  | 69        |
| 8.3      | Lists . . . . .  | 70        |
| 8.4      | Testfragen . . . . .                                     | 72        |
| <b>9</b> | <b>Andere Sprachelemente</b>                             | <b>74</b> |
| 9.1      | Kontrollstrukturen . . . . .                             | 74        |
| 9.1.1    | if-else . . . . .  | 75        |
| 9.1.2    | switch-case . . . . .                                    | 75        |
| 9.1.3    | for . . . . .  | 76        |

|                                      |  |            |
|--------------------------------------|--|------------|
| 9.1.4                                | while  | 76         |
| 9.1.5                                | continue, break, return                        | 76         |
| 9.2                                  | Gültigkeitsbereiche                            | 77         |
| 9.2.1                                | Namensbereiche                                 | 77         |
| 9.2.2                                | Compiler-Direktiven - #                        | 77         |
| 9.3                                  | Testfragen                                     | 79         |
| <b>Part II - Visual C++ mit Qt</b>   |  | <b>81</b>  |
| <b>10 Qt</b>                         |  | <b>82</b>  |
| 10.1                                 | Hello Qt                                       | 82         |
| 10.2                                 | Executable von der Konsole starten             | 83         |
| 10.3                                 | Quit - Schaltflächen                           | 84         |
| 10.4                                 | Dialoge  | 85         |
| 10.5                                 | Qt Projekt                                     | 89         |
| <b>Part III - Anlagen / Software</b> |  | <b>94</b>  |
| <b>11 Software-Installation</b>      |  | <b>95</b>  |
| 11.1                                 | Glossar  | 96         |
| 11.2                                 | cygwin Installation                            | 97         |
| 11.3                                 | LINUX  | 106        |
| 11.3.1                               | Übersicht einiger grundlegenden Linux-Befehle  | 106        |
| 11.3.2                               | Profile  | 109        |
| 11.3.3                               | Make   | 112        |
| 11.4                                 | Qt Installation                                | 113        |
| <b>12 Questions</b>                  |  | <b>115</b> |
| 12.1                                 | Testfragen - Kapitel 1 - Einführung            | 115        |
| 12.2                                 | Testfragen - Kapitel 2 - Datentypen            | 116        |
| 12.3                                 | Testfragen - Kapitel 3 - Ein- und Ausgabe I    | 116        |
| 12.4                                 | Testfragen - Kapitel 4 - Klassen               | 117        |
| 12.5                                 | Testfragen - Kapitel 5 - Strings               | 117        |
| 12.6                                 | Testfragen - Kapitel 6 - Ein- und Ausgabe II   | 118        |
| 12.7                                 | Testfragen - Kapitel 7 - Referenzen und Zeiger | 119        |

|   |     |
|---|-----|
| 12.8 Testfragen - Kapitel 8 - Container . . . . .             | 120 |
| 12.9 Testfragen - Kapitel 9 - Andere Sprachelemente . . . . . | 121 |



