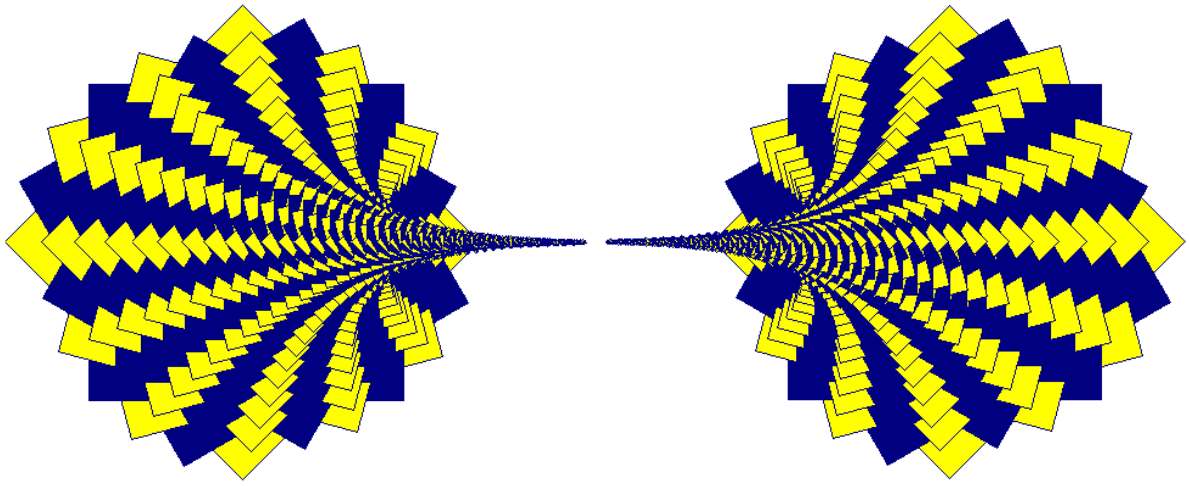
Einführung in die Programmiersprache Borland C++ / Champ

Peter Zingg

Mathematisch-Naturwissenschaftliches
Gymnasium Bern Neufeld

2. Auflage

Juni 1998



Vorwort

Am Mathematisch-Naturwissenschaftlichen Gymnasium Bern-Neufeld gehört die Einführung in eine höhere Programmiersprache schon lange zum Stoffplan des Faches Angewandte Mathematik (bzw. Anwendungen der Mathematik). Waren es zuerst die Sprachen BASIC und COMAL, wird seit einiger Zeit die Sprache C++ der Firma Borland unterrichtet. Die Gründe für den Wechsel waren die folgenden:

1. C++ ist eine Sprache, die weit verbreitet ist und später auch an Universitäten verwendet wird.
2. C++ erlaubt die Entwicklung von strukturierten Programmen.
3. An der Universität Bern wurde die Graphikbibliothek *Unigraph* entwickelt, die es erlaubt, in C++ mit einfachen Befehlen (ohne Umrechnung in Bildschirmkoordinaten) Graphiken zu zeichnen.

Die ersten Versionen von Borland C++ liefen unter dem Betriebssystem MS-DOS. Mit dem Aufkommen von Windows 3.x gab dann Borland Versionen auf den Markt, die nur noch unter Windows laufen. *Unigraph* wurde deshalb weiter entwickelt und zudem in *Champ* umgetauft.

Dieses Skript soll eine Einführung in die Programmiersprache C++, unter Verwendung von Champ, liefern. Es ist wie folgt aufgebaut:

- Die ersten zwei Kapitel liefern eine Einführung in die Begriffe der Informatik, in das Betriebssystem Windows 95 und die Programmierumgebung von Borland C++.
- Die Kapitel 3 und 4 zeigen, welche Elemente ein Programm haben muss und wie dieses in Unterprogramme (genauer: Funktionen) unterteilt werden kann.
- Kontrollstrukturen sind zentrale Elemente von höheren Programmiersprachen, sie werden in den Kapiteln 5 und 7 behandelt. Dazwischen wird gezeigt, wie Daten via Tastatur eingegeben und auf dem Bildschirm ausgegeben werden können (Kapitel 6).
- Behandelt werden weiter Ein- und Ausgabe sowie Verarbeitung von Strings und Arrays (Kapitel 8) und Zeichnen von Graphiken (Kapitel 9).
- Die letzten zwei Kapitel geben einige Ergänzungen und sowie einen Ausblick auf die Objektorientierte Programmierung.

Ich möchte allen danken, die mir in irgendeiner Form geholfen haben, dieses Skript zu realisieren, vor allem aber den Herren Aegidius Plüss und Hans Salvisberg für ihre Auseinandersetzung und prompte Erledigung meiner Fragen und Anregungen.

Bern, im Juli 1997 Peter Zingg

Vorwort zur 2. Auflage

Die vorliegende zweite Auflage dieses Skripts baut auf die erste Auflage auf, die im letzten Schuljahr am MNG Bern-Neufeld zum ersten Mal verwendet wurden. Nebst der üblichen Ausmerzung von Druckfehlern wurden folgende Kapitel entscheidend abgeändert:

- Alle Abbildungen und Vorgehensweisen wurden an die aktualisierte, erstmals auf dem Markt erschienene Version von Champ und deren Konfiguration an den Gymnasien Bern-Neufeld angepasst. Dies betrifft insbesondere das Kapitel 9.1.2 über die Arbeit mit der Champ-Hilfe.
- Im Kapitel 6.2 über die formatierte Ausgabe wurde ein separates Beispiel für links- oder rechtsbündige Ausgabe geschrieben.
- Neu geschrieben wurden ergänzende Kapitel über die Messagebox (Kapitel 10.3), über selbstdefinierte Datentypen (Kapitel 10.5) und über Header-Dateien (Anhang Anhang D:).

Danken möchte ich dem Kollegen Hansulrich Hubschmid und den Schülerinnen und Schülern der Klasse MN3b (Schuljahr 1997/98), die mich auf Schwächen und Druckfehler in der ersten Auflage hinwiesen.

Bern, im Juni 1998 Peter Zingg

Inhaltsverzeichnis

1 Einführung in die Arbeit am Computer

1.1 Elemente und Arbeitsweise des Computers

1.1.1 Das EVA-Prinzip

Die **Informatik** ist die Lehre von der systematischen Verarbeitung von Informationen mit Automaten (insbesondere mit Computern). Das Wort *Informatik* setzt sich zusammen aus den Wörtern **Infor**mation und Auto**matik**. Im englischen Sprachraum spricht man von *computer science*.

Die Verarbeitung von Informationen aller Art ist erst mit dem Aufkommen der Computer zum Gegenstand wissenschaftlicher Betrachtungen geworden. Weit verbreitet ist die Annahme, der Computer könne die anfallenden Probleme alleine lösen, wobei der Mensch nur noch als Kontrollkraft benötigt wird. Das ist aber nicht richtig! Der Computer ist im Grunde genommen sehr dumm. Insbesondere kann er keine selbständigen Entscheide treffen und weiss nicht einmal, ob sein Tun sinnvoll ist oder nicht. Alles muss ihm vorher gesagt werden. Dies geschieht über die sogenannte **Software**, d.h. über Programme, die zuerst in den Computer geladen werden müssen.

Der Computer arbeitet, wie jede andere Maschine, nach dem sogenannten **EVA-Prinzip:**

Eingabe → **Verarbeitung** → **Ausgabe**

Beispiele:

Eingabe	Maschine	Ausgabe
Tastaturanschläge	Schreibmaschine	Typenanschläge
Elektrische Signale	Lautsprecher	Töne
Kaffeebohnen, Wasser	Kaffeemaschine	Kaffee

Ein Computer ist aber keine gewöhnliche Maschine mit einem festen Arbeitsablauf; die Verarbeitung geschieht gemäss den Anweisungen eines Programms:

Programm

↓

Eingabe → **Verarbeitung** → **Ausgabe**

Deshalb versteht man unter einem **Computer** (gemäss Informatik-Duden) ein universell einsetzbares Gerät zur automatischen Verarbeitung von Daten.

1.1.2 Information

In der Umgangssprache wird eine Nachricht, die einen Sachverhalt ausdrückt und erklärt, als Information bezeichnet. Eine Nachricht wird aber erst dank der Interpretation eines Empfängers zur Information. Genauer: Werden Daten von einem Sender zu einem Empfänger übermittelt, so spricht man von einer **Nachricht**. Unter **Information** verstehen wir die Bedeutung oder den Inhalt einer Nachricht. Ein Beispiel soll diesen Sachverhalt veranschaulichen:

Am 5. Juni 1944, dem Vortag der Invasion der alliierten Streitkräfte in der Normandie, sendete die BBC London innert weniger Minuten folgende Nachrichten: "Die Würfel liegen auf dem Teppich." und "Es ist heiss in Suez". Für die deutschen Abhörspezialisten war die Bedeutung dieser Nachrichten (also die Information!) unklar. Für den französischen Widerstand war sie aber klar: Sprengen von Telefon- und Eisenbahneinrichtungen im Norden Frankreichs.

Den "Rohstoff" für Nachrichten liefern **Daten**. Daten sind Angaben über Sachverhalte in Form von Wörtern, Zahlen oder Bildern. Daten bestehen aus einer Folge von Zeichen, die alphabetische, numerische und graphische Zeichen sein können. Diese Zeichen sind die Träger der Informationen und ermöglichen deren Austausch (Kommunikation). Damit die Kommunikation gelingt, müssen Sender und Empfänger im Besitz desselben Schlüssels sein, der die Bedeutung der Zeichen festlegt. Einen solchen Schlüssel nennt man **Code**. Das Wort "Code" hat aber noch eine zweite Bedeutung: Ein Code ist eine Vorschrift für die eindeutige Zuordnung der

Zeichen eines Zeichenvorrats (**Alphabets**) zu den Zeichen eines anderen Zeichenvorrats; er ist nichts anderes als eine Übersetzungs- oder Abbildungsvorschrift.

Die kleinste Einheit für die Darstellung von Informationen, die ein Computer verstehen kann, ist **1 Bit**, sie kennt nur zwei Zustände ("Ein" / "Aus" bzw. 1 / 0). Die ganze Verarbeitung von Daten am Computer (Eingabe, Verarbeitung, Ausgabe) erfolgt in Form von **binären Signalen**, d.h. in einer Folge von 1 und 0. Das bedeutet, dass die Zeichen mit Hilfe eines **Binärcodes** in eine "Sprache" übersetzt werden müssen, die nur zwei Zeichen (meistens 1 und 0) kennt. Umgekehrt übersetzt der Binärcode auch 0-1-Folgen in "normale" Zeichen. Der für uns wichtigste Code ist der **ASCII-Code** (American Standard Code for Information Interchange; vgl. Anhang Anhang B:).

Eine Folge von **8 Bit** wird in der Informatik als **1 Byte** bezeichnet. In der Regel wird mit einem Byte ein Zeichen (Buchstabe, Ziffer, Sonderzeichen) dargestellt. Unter einem **Kilobyte** (1 KB) versteht man $2^{10} = 1024$ Byte, ein **Megabyte** (1 MB) entspricht $2^{20} = 1'048'576$ Bytes.

1.1.3 Hardware und Software

Unter **Hardware** versteht man den Computer und sein Zubehör. Dazu zählt man Drucker, Tastatur, Maus, Bildschirm, Diskettenlaufwerk, Festplatte, CD-ROM, etc.

Die wichtigste Komponente der Hardware ist natürlich der Computer. Die Computer in den meisten Büros und auch die meisten Heimcomputer sind sogenannte **Personal Computer** (**PC**), d.h. es sind auf den einzelnen Arbeitsplatz und auf die persönliche Anwendung zugeschnittene Computer.

Das Herz eines jeden Computers ist die **Zentraleinheit** (engl. **Central Processing Unit**, **CPU**). Grob gesagt, besteht die Zentraleinheit aus dem (**Mikro**-)**Prozessor** und den Verbindungen des Prozessors zu den übrigen Hardwarekomponenten wie Speicher, Ein- und Ausgabegeräte (diese Verbindungen werden auch **Busse** genannt). Einige Autoren zählen aber auch noch den Hauptspeicher zur Zentraleinheit.

Die **Eingabegeräte**, die die meisten Anwender brauchen, sind die **Tastatur** (**Keyboard**) und die **Maus**.

Jeder Tastendruck auf der Tastatur löst ein spezielles elektrisches Signal aus, das bewirkt, dass ein spezielles Codewort in einem kleinen Speicher, dem **Tastatur-Buffer**, zwischengespeichert wird, bis das System dieses Codewort abruft. Die Tastatur eines Computers ist normalerweise unterteilt in

- den **alphanumerischen Tastenblock** mit Buchstaben, Ziffern, Sonder- und Steuerzeichen;
- den **numerischen Tastenblock** zur Eingabe von Zahlen und Rechenoperationszeichen;
- die **Cursor-Steuertasten** zur Steuerung des Positionsanzeigers (engl. **Cursor**) auf dem Bildschirm;
- die **Funktionstasten**, mit denen in vielen Programmen Befehle aufgerufen werden.

Die Maus ist ein kleines Kästchen, das beim Abrollen auf einer Unterlage elektrische Impulse erzeugt, durch welche eine Markierung über einen Bildschirm bewegt werden kann. Über diese Markierung kann dann z.B. ein bestimmter Befehl aus einem Menü ausgewählt werden, indem man auf einen entsprechenden, an der Maus angebrachten Knopf drückt (Fachjargon: "klickt").

Da die Daten im Arbeitsspeicher grösstenteils verloren gehen, wenn der Computer ausgeschaltet wird, ist ihre Speicherung auf einem **externen Speicher** unabkömmlich. Darunter versteht man Speichermedien wie **Disketten** oder **Harddisks** (**Festplatten**). Noch eine geringe Rolle spielt heute (Sommer 1997) im Heimanwenderbereich die beschreibbare **CD-ROM**.

Die Computer an unserer Schule besitzen ein 3½-Zoll-Laufwerk (Laufwerk `A:`) sowie eine in zwei logische Laufwerke (**Partitionen**) `C:` und `D:` unterteilte Festplatte. Ausserdem befinden sich weitere Laufwerke auf dem Netzserver, d.h. auf dem Computer, der das Netzwerk kontrolliert.

Unter dem Sammelbegriff **Peripheriegeräte** werden alle Geräte zusammengefasst, welche an den Computer angeschlossen werden (Drucker, Bildschirm, Tastatur etc.)

Alle Programme, die auf einem Computer laufen, fasst man unter dem Begriff **Software** zusammen. Dazu gehören Textverarbeitungsprogramme, Tabellenkalkulationen, Programmiersprachen (genauer gesagt deren *Compiler*), Graphikprogramme (z.B. CAD; Computer Aided Design), Spiele, etc.

1.1.4 Das Betriebssystem

Alle Aktionen und Operationen eines Computers müssen koordiniert und die inneren Abläufe müssen gesteuert werden. Diese Aufgaben übernimmt das **Betriebssystem**. Es besteht aus vielen einzelnen Programmen.

Zum Betriebssystem gehören die **Gerätetreiber**. Das sind elementare Programme, die z.B. ein Zeichen aus dem Tastatur-Buffer holen oder ein Zeichen auf dem Bildschirm bringen. Treiber sorgen auch dafür, dass die Ausgabe auf einem Drucker richtig erfolgt oder die Impulse der Maus richtig umgesetzt werden.

Eine weitere wichtige Aufgabe eines Betriebssystems ist die **Speicherverwaltung**; der Anwender muss, anders als bei einem programmierbaren Taschenrechner, nicht selbst dafür sorgen, dass alles am richtigen Platz abgelegt wird.

Es kann sein, dass die Arbeit des Prozessors für einen Moment unterbrochen werden muss, z.B. beim unerwarteten Eintreffen neuer Daten aus einem Peripheriegerät. Bei solchen **Interrupts** (Unterbrechungen) sorgt das Betriebssystem dafür, dass die gerade bearbeiteten Daten nicht verloren gehen.

Nicht zuletzt kontrolliert ein Betriebssystem die Verbindung zur Aussenwelt. Es stellt dem Benutzer Befehle oder Methoden zur Verfügung, die es ihm erlauben, elementare und direkte Operationen zu veranlassen (Starten eines Programms, Speichern oder Kopieren von Daten etc.).

Zusammenfassung:

Die Aufgaben eines Betriebssystems sind:

- Abarbeiten von direkten Befehlen, welche dem Benutzer vom Betriebssystem zur Verfügung gestellt werden;
- Verwalten von Daten auf der Diskette, insbesondere die Steuerung des Schreib-/Lesekopfes der Disketten- und Festplattenlaufwerke;
- Laden und Starten von bestehenden Programmen;
- Verwaltung des Arbeitsspeichers;
- Verwaltung der Ein- und Ausgabe, insbesondere der Tastatur, der Maus und des Bildschirms;
- Kommunikation mit externen Geräten (Drucker, Scanner etc.);
- Störroutinen, Fehlermeldungen;
- Behandlung von Unterbrechungen.

1.2 Windows 95

1.2.1 Grundlegendes

Die Computer an unserer Schule verwenden das Betriebssystem **Windows 95**, das von der Firma *Microsoft* entwickelt wurde. Windows 95 besitzt eine **graphische Benutzeroberfläche** (engl. **Graphic User Interface**, GUI) und unterstützt die Ablage von Dateien in verschiedenen **Ordnern** (unter DOS und Windows 3.1 nannte man die Ordner noch **Verzeichnisse**). Man kann sich die Ordner wie eine Ablage in einem Aktenschrank vorstellen. Ein Laufwerk übernimmt die Rolle des Schrankes, die Schubladen entsprechen den Ordnern. Wichtig ist, dass ein Ordner wiederum mehrere Ordner enthalten kann (eine Schublade im Aktenschrank enthält eventuell auch mehrere Aktenmappen).

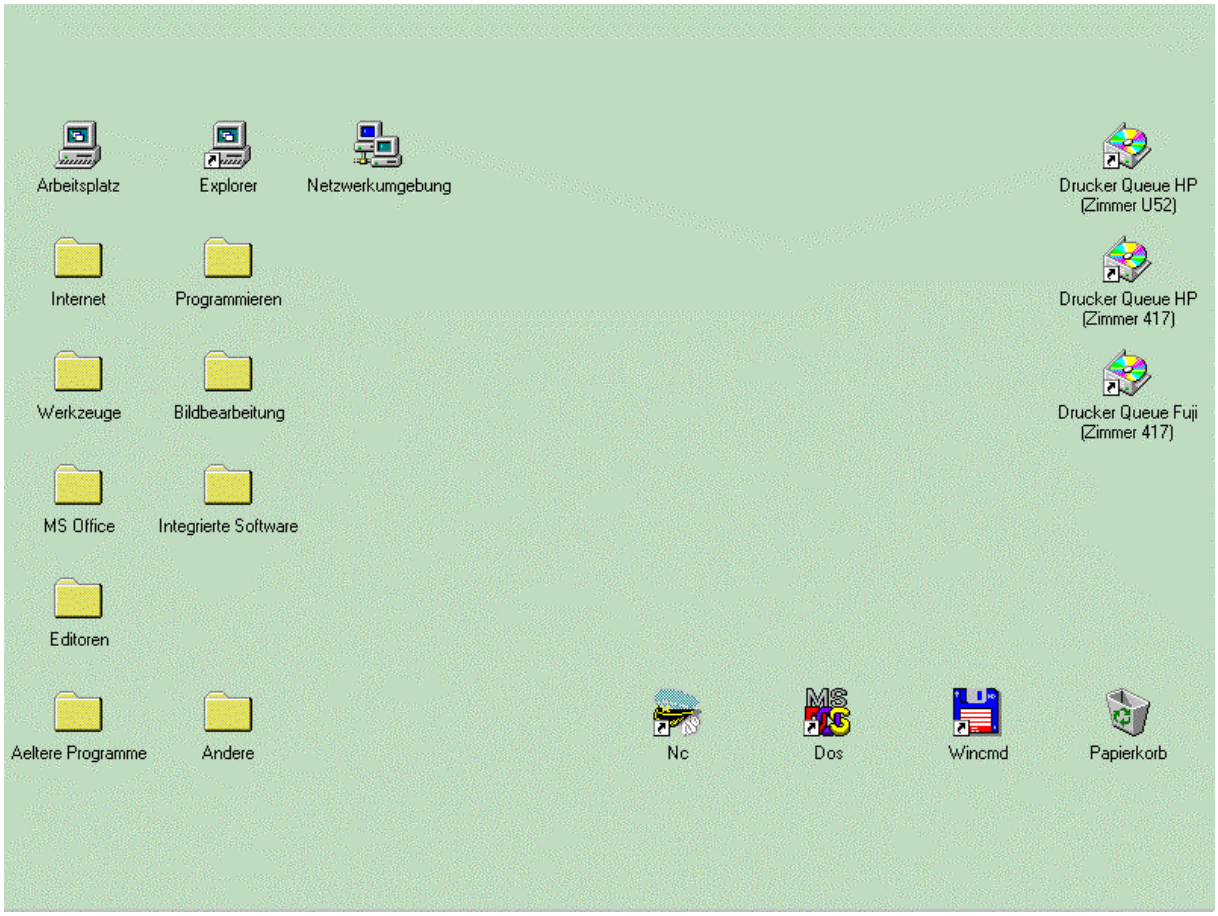
Dateien erhalten einen **Dateinamen**, der in Windows 95 insgesamt 256 Zeichen (inkl. Leerzeichen) lang sein darf. Jede Datei sollte noch eine **Erweiterung** (normalerweise 3 Zeichen) erhalten. Bei der Angabe der Erweiterung sollte man gewisse Abmachungen einhalten, da sie Informationen über den Inhalt der Datei geben kann (vgl. Tabelle rechts).

Erweiterung	Dateiinhalt	Erweiterung	Dateiinhalt
BAK	Sicherungsdatei	HLP	Hilfedatei
CPP	C++-Programmdatei	IDE	C++-Projekt
DAT	Datendatei	TMP	Temporärdatei
DOC	Word-Dokumentdatei	TXT	Textdatei
EXE	Ausführbare Datei	XLS	Exceldatei

Will man angeben, in welchem Ordner sich eine Datei befindet, so muss man deren **Pfad** angeben. Die Datei `s:\users\stud\champ97\treppe1.cpp` hat den Namen `treppe1.cpp` (d.h. es ist eine C++-Programm-

datei) und befindet sich im Ordner `champ97`, der ein Unterordner des Ordners `stud` ist, der sich wiederum im Ordner `users` befindet, der ein Ordner des Laufwerks `s:` ist.

Beim Start eines Computers erscheint der sogenannte **Desktop**, der vom Anwender für seine Bedürfnisse angepasst werden kann. Unser Desktop an der Schule ist wie folgt aufgebaut (Stand Juni 1998):



Auf dem Desktop sind mehrere Ordner abgelegt. Nach Öffnen des Ordners `MS Office` (durch Doppelklick) erscheint ein neues Fenster, aus dem die Anwendungen des Programmpakets *MS Office* (Word für Windows, Excel etc.) gestartet werden können. Im Ordner `Internet` befinden sich verschiedene Anwendungen zu Internet (Netscape Navigator, Eudora ...).

Mit einem Doppelklick auf `Dos` wird die DOS-Oberfläche gestartet.

Weitere Ordner können geöffnet werden, in dem man auf `Arbeitsplatz` klickt. Via Laufwerk, Ordner, Unterordner etc. wird der gewünschte Ordner geöffnet. Mit Doppelklicks auf (der Reihe nach) `Arbeitsplatz`, `Vol3 auf 'Nf1' (S:)`, `Users`, `Stud`, `Champ97` wird der Ordner `Champ97` geöffnet.

Sind zwei verschiedene Ordner geöffnet (z.B. zweimal via Arbeitsplatz), so können Dateien mit der Maus von einem Ordner zum anderen verschoben werden. Man fährt dazu mit der Maus auf die zu verschiebende Datei, drückt die linke Maustaste, fährt mit gedrückter Maustaste in das andere Fenster und lässt dann die Maustaste los. Dieses Verfahren wird auch **drag and drop** genannt. Befinden sich die Ordner in verschiedenen Laufwerken, wird die "verschobene" Datei sogar kopiert. Will man die Datei innerhalb eines Laufwerks kopieren, so muss man die <Ctrl>-Taste gedrückt halten, wenn die Maustaste losgelassen wird.

Sollen mehrere Dateien verschoben oder kopiert, so können diese auf zwei Arten markiert werden:

1. Stehen diese Dateien im Fenster alle direkt untereinander, so wird mit der Maus zuerst die oberste, dann, bei gedrückter <Shift>-Taste, die unterste Datei markiert. Alle dazwischen liegenden Dateien werden ebenfalls markiert.
2. Mit gedrückter <Ctrl>-Taste werden alle gewünschten Dateien markiert.

Achten Sie beim Verschieben bzw. Kopieren mit drag and drop, dass Sie auf eine markierte Datei klicken.

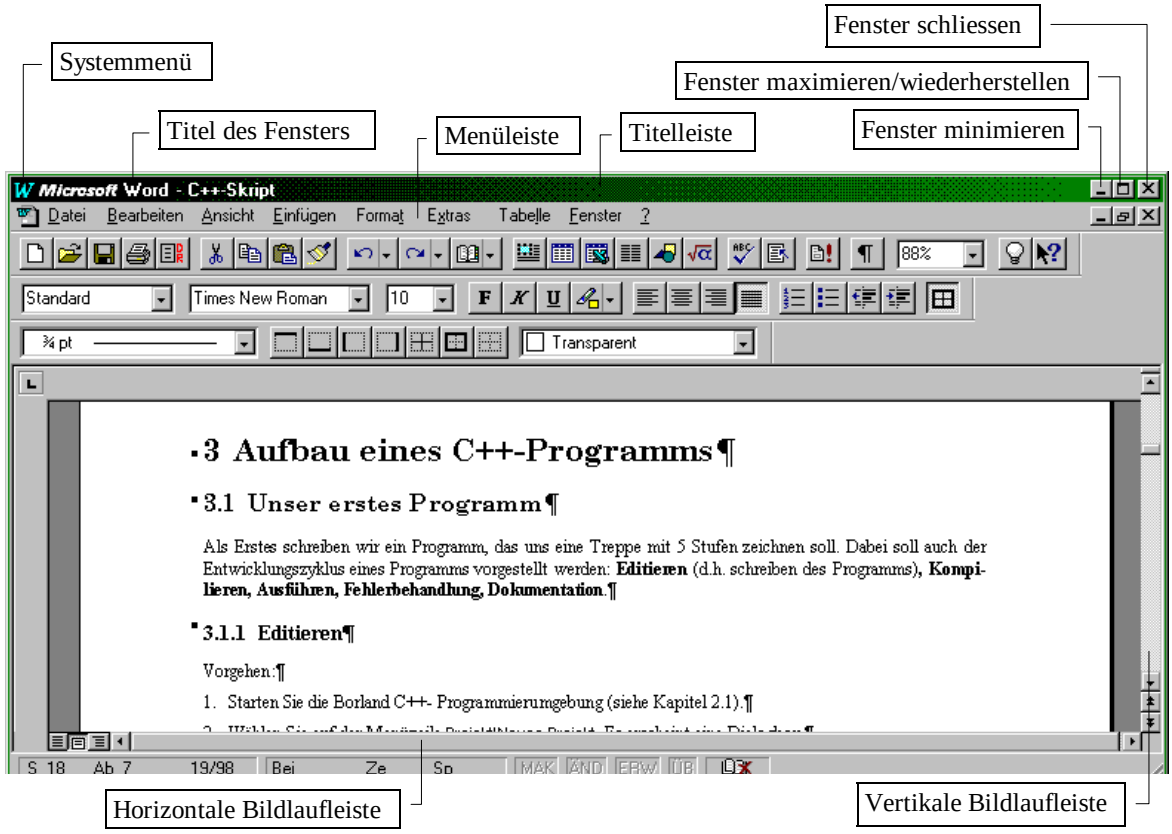
`Wincmd`, Norton Commander (`Nc`) und Windows-Explorer (`Explorer`) sind weitere Hilfsprogramme zur Organisation (löschen, verschieben, kopieren) von Dateien und Ordnern.

Verschiebt man eine Datei mit drag and drop in den `Papierkorb`, so wird die Datei gelöscht. Wenn man mit einem Doppelklick den Papierkorb öffnet, so kann (bei entsprechender Einstellung) eine versehentlich gelöschte Datei noch eine Zeitlang gerettet werden. An unserer Schule ist aber eine gelöschte Datei verloren!

Eine wichtige Rolle spielt in den Anwendungen unter Windows 95 die **rechte Maustaste**. Mit ihr werden sogenannte **Kontextmenüs** geöffnet, das sind Menüs, die sich auf das geklickte Objekt (Fenster, Wort, Mauspalette, Desktop etc.) beziehen.

1.2.2 Aufbau eines Fensters

Die folgende Abbildung zeigt ein typisches Fenster:



Zu jedem Fenster gehört eine Titelleiste. Mit einem Mausklick auf das Symbol links in der Titelleiste wird das **Systemmenü** aufgerufen. Das Systemmenü enthält Befehle, die die Grösse des Fensters auf dem Bildschirm betreffen. Ausserdem kann die Anwendung mit dem Punkt Schliessen beendet werden.

Die Grösse eines Fensters kann (sofern die Anwendung das zulässt) mit der Maus verändert werden. Fahren Sie mit der Maus auf die rechte untere Ecke des Fensters und drücken Sie die linke Taste. Nachdem Sie die Grösse des Fensters nach Ihren Wünschen verändert haben, lassen Sie die Taste wieder los.

Sie können das Fenster auch an eine andere Stelle des Bildschirms verschieben, indem Sie mit der Maus auf die Titelleiste fahren. Das Fenster wird dann mit gedrückter Maustaste verschoben.

Ein Fenster **minimieren** bedeutet faktisch, das Fenster vom Bildschirm "verschwinden" zu lassen. Die Anwendung wird aber nicht geschlossen, mit Hilfe der Task-Leiste kann das Fenster wieder hergestellt werden.

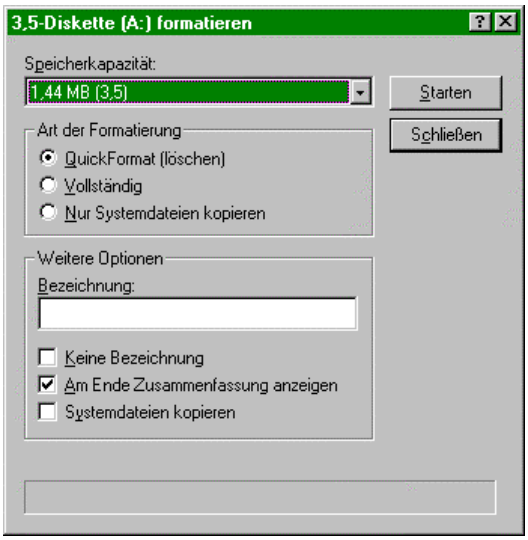
Soll ein Fenster den ganzen Bildschirm einnehmen, so kann man es **maximieren**. Umgekehrt kann man ein Fenster **wiederherstellen**, d.h. wieder auf die zuletzt definierte Grösse "zurücksetzen". Beides geschieht mit einem Mausklick auf das zweite Symbol von rechts in der Titelleiste.

Klickt man auf das Symbol rechts aussen, so wird die Anwendung (also auch das Fenster) geschlossen.

Mit Hilfe der Bildlaufleisten kann der Inhalt eines Fensters im Fenster bewegt werden.

Windows 95 erlaubt einem, mehrere Fenster geöffnet zu haben. Allerdings kann gleichzeitig (natürlich) nur in einem Fenster, dem **aktiven** Fenster, gearbeitet werden, alle anderen Fenster sind **inaktiv**. Ein Fenster wird entweder mit Hilfe der Task-Leiste oder mit einem Mausklick auf das Fenster aktiviert.

1.2.3 Formatieren einer Diskette



Heute sind die Disketten, die im Handel erhältlich sind, bereits formatiert, d.h. sie sind für ihre Verwendung unter dem Betriebssystem Windows 95 vorbereitet. (Andere Betriebssysteme verlangen unter Umständen andere Formatierungen.) Es kommt aber vor, dass eine alte Diskette neu formatiert werden soll. In diesem Fall geht man wie folgt vor:

1. Öffnen sie (auf dem Desktop) mit einem Doppelklick den Ordner `Arbeitsplatz`.
2. Klicken Sie dann **einmal** auf das Symbol für den Datenträger, den Sie formatieren möchten. Im Normalfall betrifft dies das Laufwerk `A:` (`3,5-Diskette (A:)`).
3. Wählen Sie den Menüpunkt Datei|Formatieren. Es erscheint die Dialogbox rechts.
4. Wählen Sie in der Dialogbox die gewünschten Optionen.
5. Mit einem Mausklick auf Starten wird die Formatierung gestartet.

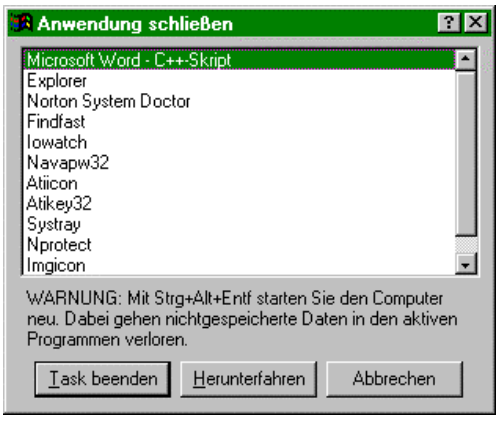
Bemerkung:

Wird eine Diskette neu formatiert, so gehen die auf der Diskette gespeicherten Informationen verloren!

1.2.4 Multitasking, die Task-Leiste

Windows 95, ist in der Lage, mehrere Anwendungen (scheinbar) parallel arbeiten zu lassen, man spricht in diesem Fall von **Multitasking**. Man kann sich die Kontrolle des Multitaskings wie einen Stafettenlauf vorstellen. Das Betriebsystem übergibt den Stab an eine laufende Anwendung, diese beansprucht den Prozessor für eine kurze Zeit, übergibt den Stab wieder an den Prozessor, der den Stab an die nächste Anwendung übergibt, usw.

Manchmal kommt es vor, dass eine Anwendung "den Stab" nicht mehr zurückgeben kann. Das ist zum Beispiel dann der Fall, wenn sich die Anwendung in einer Endlosschleife befindet (dies kann bei selbst programmierten Anwendungen in der Testphase oft passieren) oder aus anderen Gründen "abgestürzt" ist. Daher muss der Programmierer daran denken, in seinem Programm an bestimmten Stellen die Kontrolle an das Betriebssystem zurück zu geben. In den Programmen, die wir schreiben werden, ist das automatisch der Fall, z.B. dann, wenn ein Programm vom Anwender eine Eingabe erwartet. Eine Ausnahme bildet die Ausgabe im Konsolenfenster in einer Endlosschleife (vgl. Seite 34).



Drücken Sie, falls ein Programm "abgestürzt" ist (dies ist dann der Fall, wenn der Computer auf keine Eingaben mit der Tastatur oder der Maus mehr reagiert), gleichzeitig die drei Tasten <Ctrl> + <Alt> + <Del>. Es erscheint das Fenster rechts. Mit Task beenden können Sie die markierte Anwendung (normalerweise also das "hängende" Programm) beenden, die Informationen gehen aber verloren, falls Sie vorher nicht gespeichert wurden.

Klicken Sie auf Herunterfahren, so wird Windows 95 beendet.

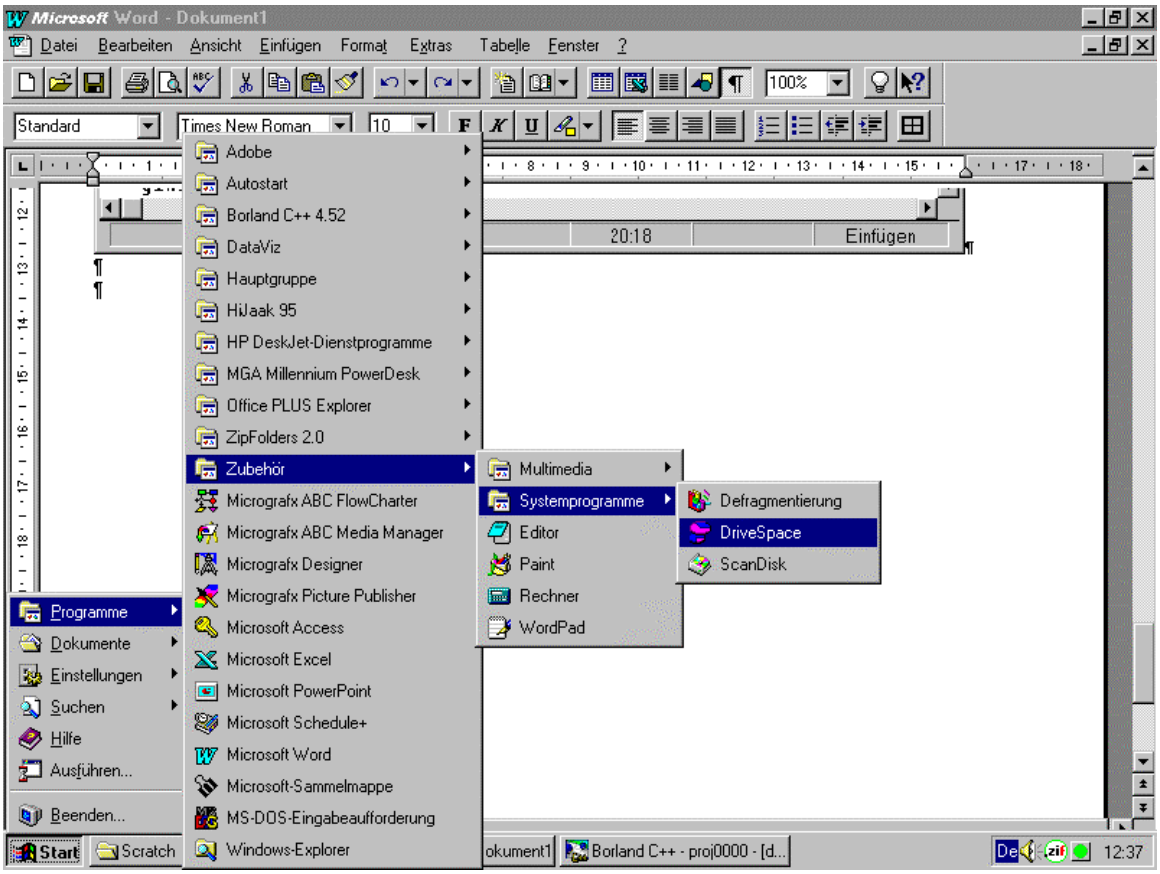
Es kann vorkommen, dass ein Programm auch mit der Tastenkombination <Ctrl> + <Alt> + <Del> nicht beendet werden kann. In diesem Fall hilft nur noch ein Neustart des Computers.

Fährt man mit der Maus an den unteren Bildschirmrand, so erscheint die **Task-Leiste**:

Die Task-Leiste enthält eine Liste aller geöffneter Anwendungen und Ordner. Ein Mausklick auf eine Schaltfläche aktiviert das entsprechende Fenster.

Soll eine neue Anwendung gestartet werden, klickt man mit der Maus die Schaltfläche Start. Das Startmenü erscheint. Fährt man mit der Maus auf einen Menüpunkt, der einen Pfeil enthält, wird ein weiteres Menü geöffnet. Durch einen einzigen Mausklick auf den entsprechenden Menüpunkt wird eine Anwendung gestartet.

Beispiel:



1.2.5 Dialogboxen

Viele Anwendungen von Windows 95 arbeiten mit **Dialogboxen**. Die folgende Abbildung zeigt eine typische Dialogbox.

Diese Dialogbox enthält zwei **Eingabefelder** (Suchen nach und Ersetzen durch), mehrere **Auswahlfelder** und vier **Aktionsschalter** (bzw. **Schaltflächen**; OK, Alles ersetzen, Abbruch und Hilfe).

Von den Auswahlfeldern unter Optionen sind die Optionen Gross-/Kleinschreibung und Mit Bestätigung aktiviert, dies erkennt man an den Haken in den entsprechenden Feldern. Die Auswahlfeldern unter den Stichworten Bereich, Richtung und Beginn enthalten können jeweils nicht beide gewählt werden, es sind Auswahlen vom Typ entweder/oder.

Sollen die Eingaben in der Dialogbox verworfen werden, so kann sie mit der Schaltfläche Abbruch oder mit der <Esc>-Taste geschlossen werden.

2 Der Editor von Borland C++ und Champ

Die integrierte Entwicklungsumgebung (engl. Integrated Development Environment, kurz **IDE**) von Borland stellt alle Befehle zum Schreiben, Editieren, Kompilieren und Testen von Programmen zur Verfügung. In den folgenden Abschnitten werden die wichtigsten Befehle und die Arbeit mit der IDE vorgestellt.

2.1 Starten der IDE

Auf den Computern unserer Schule wird die Entwicklungsumgebung am besten wie folgt gestartet.

1. Öffnen Sie (mit einem Doppelklick) auf dem Desktop von Windows 95 den Ordner `Programmieren`.
2. Klicken Sie zweimal auf den Eintrag "Champ"; die IDE wird gestartet.

Nach dem Start erscheint das Fenster rechts:

In der Titelleiste erscheint neben den üblichen Symbolen auch die Champ-Schaltfläche.

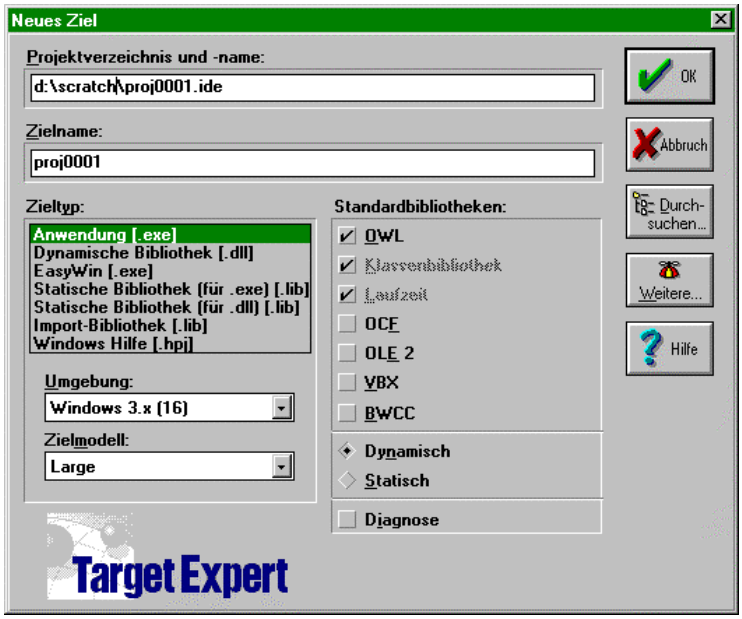
Die Zeile mit den Stichworten "Datei Bearbeiten Suchen . . ." nennt man die **Menüzeile**. Die einzelnen Menüs werden im Kapitel 2.4 vorgestellt.

Die Zeile mit den Symbolen wird in der IDE von Borland **Mauspalette** genannt, in anderen Anwendungen (z.B. Word für Windows) nennt man diese Zeile auch **Symbolleiste**. Die einzelnen Symbole werden im Kapitel 2.3 erklärt.

Zuunterst im Fenster ist die **Statuszeile**. In ihr werden Information wie aktueller Standort des Cursors, Überschreibe-/Einfüge-Modus, Informationen zu den einzelnen Symbolen der Mauspalette usw. angezeigt.

2.2 Öffnen eines neuen Projekts

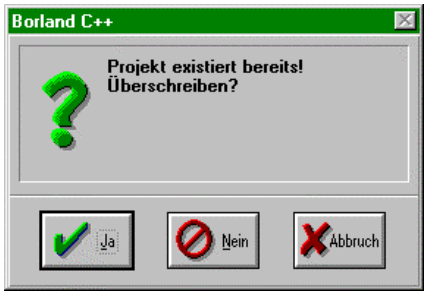
In Borland C++ werden nicht Programme, sondern **Projekte** bearbeitet. Alle Dateien (**Module**), die zu einem lauffähigen Programm nötig sind, werden zu einem Projekt zusammengefasst; ein Projekt kann aber mehrere lauffähige Programme umfassen. Die IDE kann gleichzeitig nur ein Projekt geöffnet haben.



Nachdem Sie die IDE gestartet haben, gehen Sie wie folgt vor:

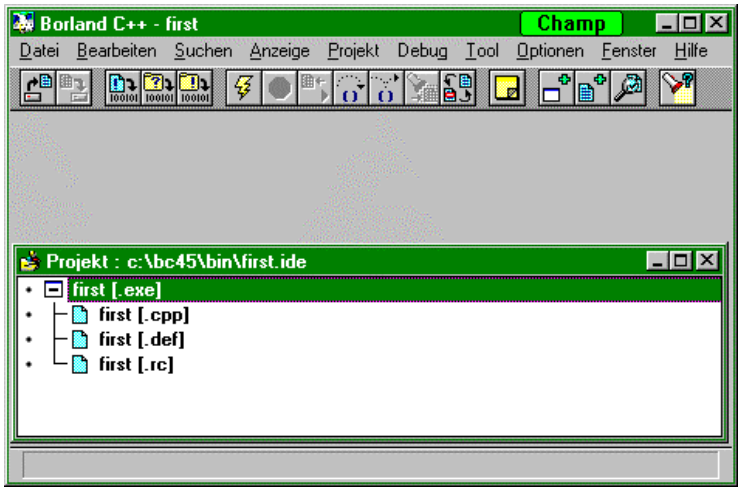
1. Wählen Sie auf der Menüzeile den Befehl Projekt|Neues Projekt oder klicken Sie auf das entsprechende Symbol. Es erscheint die Dialogbox in der Abbildung rechts:
2. Geben Sie in der ersten Zeile den gewünschten Namen des Projekts ohne Angabe eines Pfades oder einer Erweiterung an (z.B. `first`). Bestätigen Sie die Eingabe mit der Return-Taste oder mit einem Mausklick auf OK. **Hinweis!** Ausser der Eingabe des Projektnamens braucht in diesem Fenster nichts verändert zu werden. Beachten Sie, dass der Projektname, dies im Unterschied zu Dateinamen in Windows 95, höchstens acht Zeichen enthalten darf.
3. Falls Sie einen Namen eines bereits bestehenden Projekts eingeben, erscheint die folgende Warnung:

1. Klicken Sie in diesem Fall, je nach Ihrem Wunsch, auf "Ja" oder "Nein". Im letzteren Fall wird der Vorgang abgebrochen, d.h. es wird kein neues Projekt erzeugt. Dasselbe geschieht auch, wenn Sie auf "Abbruch" klicken.



Haben Sie einen neuen Projektnamen eingegeben oder auf "Ja" geklickt, so wird das Projekt erzeugt bzw. überschrieben. Zusätzlich erscheint innerhalb des Fensters der IDE ein weiteres Fenster, so dass die ganze IDE etwa wie folgt aussieht:

Alle Symbole bis auf eine Ausnahme werden auch in der Mauspalette gezeigt, wenn das **Editierfenster** aktiv ist. Diese Palette wird im Kapitel 2.3 beschrieben.



Die Ausnahme betrifft das fünfte Symbol von rechts. Wird auf dieses Symbol geklickt, wird das sogenannte **Meldungsfenster** geöffnet (vgl. Kapitel 3.1.2).

Um ein Programm schreiben zu können, öffnen Sie mit einem Mausklick auf die hellgrüne Champ-Schaltfläche in der Titelleiste den **Champ Project Manager**. Existiert bereits eine Datei mit demselben Namen (z.B. wenn ein Projekt überschrieben wurde), so erscheint das Fenster links (mit dem Hinweis <reuse existing file>), andernfalls erscheint das Fenster rechts (mit dem Hinweis <create>): Im ersten Fall wird die bestehende Datei

`first.cpp` (unser Projektname des Beispiels) wiederverwendet, im zweiten Fall wird eine Datei mit diesem Namen erzeugt. Die Endung **CPP** bedeutet, dass die Datei ein C++-Programm (genauer gesagt den Quellcode eines C++-Programms, vgl. Kapitel 3.1.1) enthält.

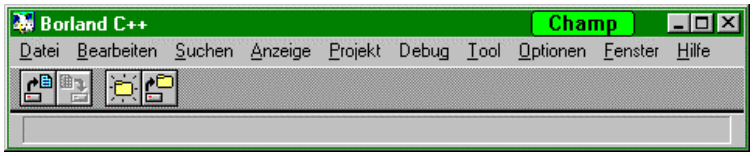
Normalerweise können die Angaben mit Mausklick auf OK bestätigt werden. Es erscheint ein neues Fenster, das sogenannte **Editierfenster** oder **Editor**, in dem nun das Programm geschrieben wird (näheres dazu erfahren Sie im Kapitel 2).

Wird ein geschriebenes Programm gestartet, so wird der Quellcode automatisch gespeichert, sofern die entsprechende Option eingeschaltet ist (dies ist an unserer Schule der Fall). Arbeitet man an umfangreichen Programmen, so empfiehlt es sich, die Datei auch zwischendurch zu speichern.

2.3 Die Mauspalette

2.3.1 IDE-Desktop

Ist noch kein Projekt geöffnet (d.h. ist nur der IDE-**Desktop** vorhanden), so sieht die Mauspalette wie folgt aus:

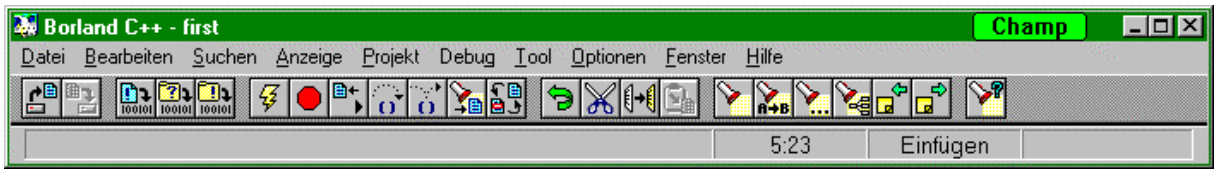


Die einzelnen Symbole bedeuten (von links nach rechts):

- Datei öffnen oder erzeugen.
- Datei im aktiven (d.h. aktuellen) Fenster speichern. Dieser Befehl kann im Moment nicht aufgerufen werden, weil nach dem Start der Umgebung keine Datei geöffnet ist.
- Neues Projekt erzeugen.
- Ein bestehendes Projekt öffnen.

2.3.2 Editor

Die Symbole in der Mauspalette haben folgende Bedeutung (von links nach rechts):



- Datei öffnen/erzeugen.
- Datei im aktiven Fenster speichern (nur aktiviert, falls die geöffnete Datei `first.cpp` verändert wird).
- Vollständige Kompilation der Datei im aktiven Fenster (vgl. Kapitel 3.1.2).
- Kompilation des ganzen Projekts im aktiven Fenster, es werden nur die geänderten Programme aktualisiert.
- Das gesamte Projekt wird neu kompiliert. (Ein Projekt kann mehrere Programme enthalten!)
- Ausführen (Starten) des Programms im aktuellen Fenster. Falls nötig, wird das Programm vorher kompiliert (vgl. Kapitel 3.1.3).
- Beenden des im Moment laufenden Programms (wird dann aktiviert, wenn ein Programm gestartet wird).

Die nächsten fünf Symbole dienen zur schrittweisen Ausführung (**Debuggen**) des aktuellen Programms. Dieses Vorgehen ist vor allem dann sinnvoll, wenn das Programm nicht wunschgemäss läuft. An dieser Stelle wird aber nicht näher darauf eingegangen.

- Letzte Aktion rückgängig machen.
- Markierter Block ausschneiden (nur aktiviert, falls ein Block markiert ist).
- Markierter Block in die **Zwischenablage** kopieren (nur aktiviert, falls ein Block markiert ist).
- Inhalt der Zwischenablage einfügen (nur aktiviert, wenn die Zwischenablage nicht leer ist) .
- Suchen nach einem bestimmten Text.
- Suchen nach einem bestimmten Text und diesen durch einen anderen ersetzen.
- Gehe zu der Zeile, auf welche sich die vorherige Meldung (im Meldungsfenster) bezieht.
- Gehe zu der Zeile, auf welche sich die nächste Meldung (im Meldungsfenster) bezieht.
- Aufruf der Hilfe: Angezeigt werden Informationen über das Wort, auf welchem sich der Cursor befindet.

2.4 Die Menüzeile und die einzelnen Menüs

Allgemeine Bemerkungen:

- Alle Punkte in den Menüs können auch ohne Maus ausgewählt werden: Drücken Sie gleichzeitig die Taste <Alt> und denjenigen Buchstaben, der im gewünschten Menü unterstrichen ist. Sobald das Menü auf dem Bildschirm erscheint, geben Sie den Buchstaben ein, der im gewünschten Menüpunkt unterstrichen ist.

  Beispiel: Den Punkt Suchen|Ersetzen erreichen Sie, indem Sie zuerst gleichzeitig die Tasten <Alt> und <S> und dann die Taste <E> drücken.

  Dies gilt auch für andere Anwendungen unter Windows 95.

- Wo vorhanden, werden auch die Tastenkombinationen (engl. **Short-Cuts**) der einzelnen Menüpunkte angegeben. (Die Umschalttaste wird mit <Shift> bezeichnet.)

- Wählt man einen Menüpunkt, der zusätzlich drei Punkte (...) enthält, wird eine Dialogbox geöffnet.

2.4.1 Das Menü "Datei"

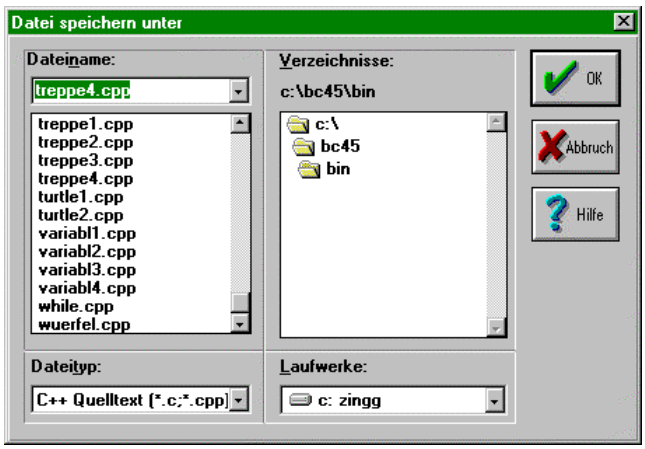
Das Menü Datei stellt Befehle zur Verfügung, die die Bearbeitung von Dateien betreffen:

- Datei|Neu: Öffnen einer neuen Datei. Dieser Befehl wird in Borland C++ selten gebraucht, da eine neue Datei auch dann erzeugt wird, wenn ein neues Projekt geöffnet wird (vgl. Kapitel 2.2).

- Datei|Öffnen: Öffnen einer bestehenden Datei. Dieser Befehl wird etwas häufiger verwendet, doch wird beim Öffnen eines bestehenden Projekts normalerweise auch die Datei mit dem Quellcode geöffnet. Wird durch ein Programm eine neue Datei erzeugt (vgl. Kapitel 10.3), so kann diese aber mit Datei|Öffnen geöffnet und betrachtet werden.

- Datei|Speichern: Speichert die Datei im aktuellen Fenster. Dieser Befehl ist vor allem dann sinnvoll, wenn man an einem grösseren Programm arbeitet. Wird ein Programm ausgeführt, so wird die Datei mit dem Quellcode (vgl. Kapitel 3.1.1) automatisch gespeichert. Wird eine Datei das erste Mal gespeichert, so muss in einem speziellen Fenster noch der Dateiname eingegeben werden (vgl. Datei|Speichern unter...). Short-Cut: <Alt> + K, dann die Taste <S> drücken.

- Datei|Speichern unter...: Speichert eine Datei unter einem neuen Namen: Nach der Wahl dieses Punktes erscheint das Fenster rechts. Im Feld oben links können Sie den gewünschten Namen eingeben, allenfalls können Sie das Laufwerk (Feld rechts unten, klicken Sie dazu auf das Feld mit dem Pfeil) und den Ordner wechseln. Die Endung cpp wird automatisch hinzugefügt, sie kann also bei der Eingabe des Namens weggelassen werden. Beachten Sie, dass der Name maximal acht Zeichen lang sein darf.



- Datei|Alles speichern: Speichert alle geänderten Dateien.

- Datei|Drucken: Druckt die aktuelle Datei.

- Datei|Druckerinstallation...: Dieser Punkt wird z.B. dann gewählt, wenn man angeben will, auf welchem Drucker eine Datei gedruckt werden soll.

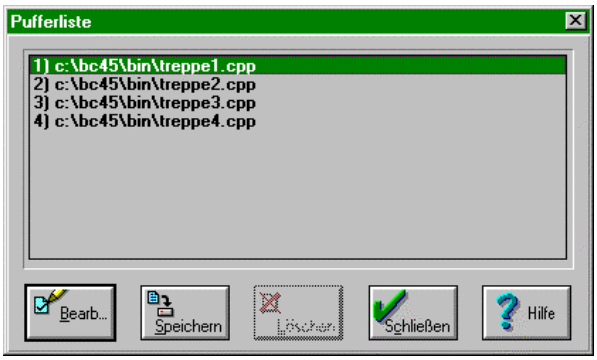
- Datei|Beenden: **Vorsicht:** Die ganze IDE wird geschlossen, d.h. Borland C++ wird beendet!

Unter Umständen sind im Menü noch die letzten Dateien angegeben, die zuletzt geöffnet waren. Wählt man eine dieser Dateien, so wird sie geöffnet.

2.4.2 Das Menü "Bearbeiten"

Dieses Menü enthält Befehle, die das Editieren eines Programms erleichtern (vgl. Kapitel 2.5.2).

- Bearbeiten|Rückgängig: Die letzte Aktion wird rückgängig gemacht, wird z.B. dann verwendet, wenn versehentlich ein Block gelöscht wurde. Short-Cut: <Ctrl> + <Z>.
- Bearbeiten|Widerrufen: Widerruft das letzte "Rückgängig". Short-Cut: <Shift> + <Ctrl> + <Z>.
- Bearbeiten|Ausschneiden: Verschiebt markierten Block in die Zwischenablage. Short-Cut: <Ctrl> + <X>.
- Bearbeiten|Kopieren: Kopiert einen markierten Block in die Zwischenablage. Short-Cut: <Ctrl> + <C>.
- Bearbeiten|Einfügen: Fügt den Inhalt der Zwischenablage an der aktuellen Cursorposition ein. Short-Cut: <Ctrl> + <V>.
- Bearbeiten|Löschen: Löscht den markierten Block. Short-Cut: <Ctrl> + <Del>.
- Bearbeiten|Alles markieren: Markiert die ganze Datei.
- Bearbeiten|Pufferliste: Zeigt in einem Fenster die Liste aller geöffneter Dateien an (Fenster rechts). Aus dieser Liste kann dann z.B. eine andere Datei aktiviert und bearbeitet werden.



2.4.3 Das Menü "Suchen"

Die Befehle dieses Menüs helfen beim Suchen (und eventuell Ersetzen) einer bestimmten Stelle bzw. eines bestimmten Wortes im Programm (vgl. Kapitel 2.5.3). Die wichtigsten Befehle sind:

- Suchen|Suchen nach...: Suche nach einem bestimmten Text. Short Cut: <Ctrl> + <Q>, dann <F>.
- Suchen|Ersetzen...: Suchen und ersetzen eines bestimmten Textes. Short Cut: <Ctrl> + <Q>, dann <A>.
- Suchen|Suche wiederholen...: Suche wird wiederholt. Short Cut: Taste <F3>.
- Suchen|Vorherige Meldung...: Zeigt die Zeile an, auf welche sich die vorherige Meldung (im Meldungsfenster) bezieht. Short Cut: <Alt> + <F7>.
- Suchen|Nächste Meldung...: Zeigt die Zeile an, auf welche sich die nächste Meldung (im Meldungsfenster) bezieht. Short Cut: <Alt> + <F8>.

2.4.4 Das Menü "Anzeige"

Mit dem Menü Anzeige können verschiedene Informationen zum Projekt, zum Programm usw. abgerufen werden. Auf diese Möglichkeiten wird aber an dieser Stelle nicht weiter eingegangen.

2.4.5 Das Menü "Projekt"

Dieses Menü enthält Befehle zur Bearbeitung eines Projekts. Die wichtigsten Befehle sind:

- Projekt|Neues Projekt...: Erzeugen eines neuen Projekts.
- Projekt|Projekt öffnen...: Öffnen eines bestehenden Projekts.
- Projekt|Projekt schliessen: Schliessen eines Projekts.
- Projekt|Neue Zieldatei...: Einem bestehenden Projekt wird eine neue Zieldatei hinzugefügt. In den meisten Fällen ist das ein weiteres Programm (vgl. Kapitel 2.5.1)
- Projekt|Compilieren: Vollständige Kompilation der Datei im aktiven Fenster (vgl. Kapitel 3.1.2). Short-Cut: <Alt> + <F9>.
- Projekt|Projekt aktualisieren: Kompilation des ganzen Projekts im aktiven Fenster, es werden nur die geänderten Programme aktualisiert.
- Projekt|Projekt neu compilieren: Das gesamte Projekt wird neu kompiliert. (Ein Projekt kann mehrere Programme enthalten!)

2.4.6 Das Menü "Debug"

Dieses Menü enthält verschiedene Befehle, die das Ausführen und das Testen eines Programms betreffen. Für uns ist im Moment nur ein Befehl wichtig:

- Debug|Ausführen: Ausführen eines Programms. Short Cut: <Ctrl> + <F9>.

2.4.7 Die Menüs "Tools" und "Optionen"

Aus dem Menü Tools können verschiedene Dienstprogramme (z.B. auch der Champ Project Manager) aufgerufen werden, die für uns unwichtig sind. Im Menü Optionen können Bildschirmeinstellungen, Vorgaben für den Compiler etc. eingestellt werden.

2.4.8 Das Menü "Fenster"

In diesem Menü können Sie wählen, wie Sie die verschiedenen Fenster eines geöffneten Projekts (Programmdateien, Meldungsfenster, etc.) auf dem Bildschirm anordnen wollen. Ausserdem enthält das Menü eine Liste aller geöffneter Fenster, mit Hilfe derer Sie ein bestimmtes Fenster aktivieren können.

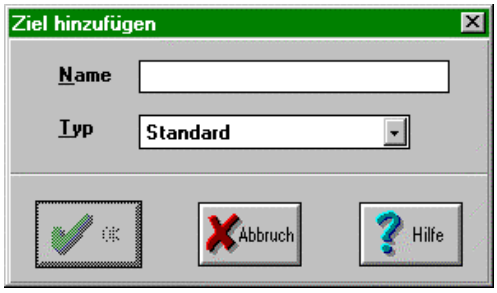
2.4.9 Das Menü "Hilfe"

Suchen Sie einen Befehl oder Informationen zu einem bestimmten Befehl, so kann mit dem Menü Hilfe die Borland-C++-Hilfe oder die Champ-Hilfe aufgerufen werden. Mehr zur Hilfe erfahren Sie im Kapitel 9.1.2.

2.5 Arbeiten mit dem Editor

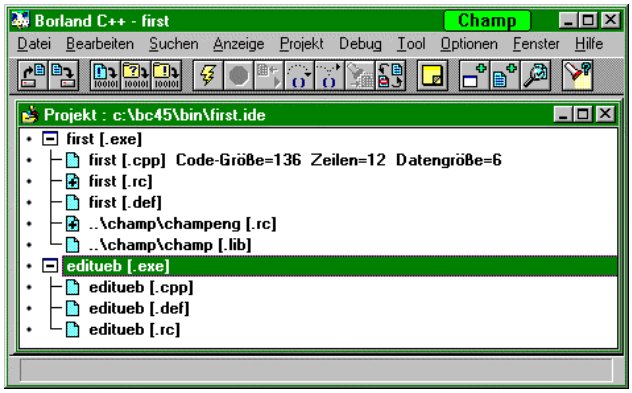
In diesem Kapitel soll gezeigt werden, wie man mit dem Editor arbeitet. Zu diesem Zweck wird eine bestehende Datei bearbeitet. Auf unserem Server steht eine spezielle Datei, `editueb.cpp`, zur Verfügung.

2.5.1 Öffnen einer neuen Zieldatei



Die Datei soll als neue Zieldatei in unser Projekt `first` eingebunden werden:

1. Falls Sie das Projekt nicht geöffnet haben, können Sie es mit Projekt|Projekt öffnen... öffnen. Wählen Sie dazu im Fenster, das erscheint, das Projekt `first.ide`.
2. Sobald das Projekt geöffnet ist, wählen Sie den Menüpunkt Projekt|Neue Zieldatei..., es erscheint das Fenster rechts.



1. Geben Sie im ersten Feld einen Namen ein (z.B. `editueb`, ohne Erweiterung und mit höchstens acht Zeichen) und bestätigen Sie diesen mit OK. Ein neue Dialogbox erscheint, die Sie ebenfalls ohne etwas zu ändern mit OK schliessen können. Im Projektfenster erscheint ein neuer Knoten.
2. Klicken Sie mit der Maus einmal auf diesen Knoten (also auf `editueb.exe`), so dass diese Zeile markiert ist, und klicken Sie dann auf die Champ-Schaltfläche in der Titelleiste. Im **Champ Project Manager**, d.h. in der neuen Dialogbox, klicken Sie auf die Schaltfläche mit der Aufschrift COPY. (Beachten Sie, dass das Fenster noch vier Knöpfe mit der Aufschrift Copy hat, wählen Sie also die richtige Schaltfläche!)
3. Wählen Sie die Datei `s:\users\stud\champ98\editueb.cpp`. Der Champ Project Manager warnt Sie mit einer roten Markierung links des Feldes mit dem Dateinamen, wenn eine bereits bestehende Datei überschrieben werden soll. Bestätigen Sie die Eingabe mit OK. Auf dem Bildschirm erscheint in einem Fenster der Inhalt der Datei.

Bemerkung: Natürlich kann auch ein neues Projekt mit dem Namen `editueb` erzeugt werden. Die Punkte 4 und 5 zum Kopieren einer bestehenden Datei bleiben aber dieselben.

2.5.2 Blöcke

Die Datei `editueb.cpp` hat folgenden Inhalt:

```
// EDITUEB.CPP
// Uebungsdatei zum BC++ Editor.

 5  Markieren Sie diese zwei Zeilen mit der Maus.
    Diese markierten Zeilen nennt man einen Block.

    Verschieben Sie diesen Block an die Stelle, die mit &&& markiert ist:

10  &&&

    Markieren Sie hinterher den gleichen Block und kopieren Sie ihn an die
    Stelle, die mit dem @@@ markiert ist:

15  @@@

    Loeschen Sie diesen Satz, indem Sie ihn als Block markieren und
    anschliessend den Menuepunkt Bearbeiten|Loeschen anklicken.

20  Sie finden hier eine Schlange, die Gemuese enthaelt:

    BlumenkohlWirzBuschbohnenTomatenKarottenKabisErbsenSchnittmangoldSellerie
    LauchBroccoliAuberginenRettichRandenKopfsalatNuesslerSpinatKartoffelnZwiebeln

25  Ersetzen Sie mit Hilfe des Menuepunktes Suchen|Ersetzen...
    das Wort 'Kabis' durch 'Kohl'.

    Entnehmen Sie dann aus der Liste Ihr Liebliengsgemuese
    und setzen Sie es an die folgende Stelle:
30
    Lieblingsgemuese:



35  Setzen die den Cursor auf diese Zeile und loeschen Sie die Zeile mit <Ctrl> + <Y>.
```

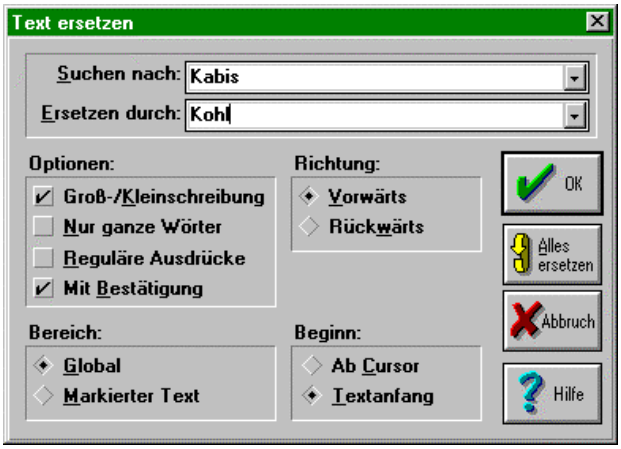
Wenn Sie die Datei wie angegeben bearbeiten wollen, gehen Sie wie folgt vor:

1. Markieren Sie mit der Maus die Zeilen 5 und 6, in dem Sie den Cursor auf den Beginn der Zeile 5 bewegen, die linke Taste drücken, mit gedrückter Taste den Mauszeiger auf das Ende der Zeile 6 bewegen und dann die Taste loslassen. Die markierten Zeilen nennt man einen **markierten Block** oder kurz **Block**.
2. Mit Bearbeiten|Ausschneiden **verschieben** Sie den Block in die Zwischenablage.
3. Bewegen Sie den Cursor mit der Maus auf die Zeile 10.
4. Mit Bearbeiten|Einfügen wird der zuletzt ausgeschnittene Block an der entsprechenden Stelle eingefügt.
5. Markieren Sie den Block erneut. Mit Bearbeiten|Kopieren wird der markierte Block in die Zwischenablage **kopiert**.
6. Bewegen Sie den Cursor auf die Zeile 15 und fügen Sie dort den Block wieder mit Bearbeiten|Einfügen ein.
7. Markieren Sie die Zeilen 17 und 18. Mit Bearbeiten|Löschen können Sie den markierten Block löschen.
8. Suchen Sie aus den Zeilen 22 und 23 Ihr Lieblingsgemüse aus und kopieren Sie es (mit kopieren und einfügen) auf die Zeile 33.
9. Die letzte Zeile kann gelöscht werden, wenn man den Cursor auf die Zeile setzt und diese dann mit dem Short-Cut <Ctrl> + <Y> löscht.

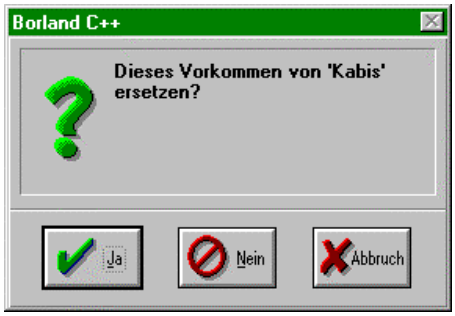
Bemerkung: Anstelle der Menüpunkte können auch, wo vorhanden, die entsprechenden Symbole der Mauspalette verwendet werden!

2.5.3 Suchen und Ersetzen

Will man das Wort "Kabis" durch "Kohl" ersetzen, geht man wie folgt vor:



1. Wählen Sie den Menüpunkt Suchen|Ersetzen..., es erscheint das Fenster rechts.
2. Im ersten Feld schreiben Sie "Kabis", im zweiten Feld "Kohl". Bestätigen Sie die Eingabe entweder mit OK oder mit Alles ersetzen.
1. Falls vorher die Option Mit Bestätigung gewählt wurde, erscheint das nächste Fenster (rechts unten), sobald der gesuchte Text gefunden wird. Er wird dann im Textfenster markiert. Mit Ja wird die Ersetzung bestätigt, mit Nein wird der Text nicht ersetzt. Falls Sie im Punkt 2 die Eingabe mit Alles ersetzen bestätigt haben, wird in beiden Fällen die Suche fortgesetzt. Wählen Sie den Punkt Abbruch, wird die Suche abgebrochen.

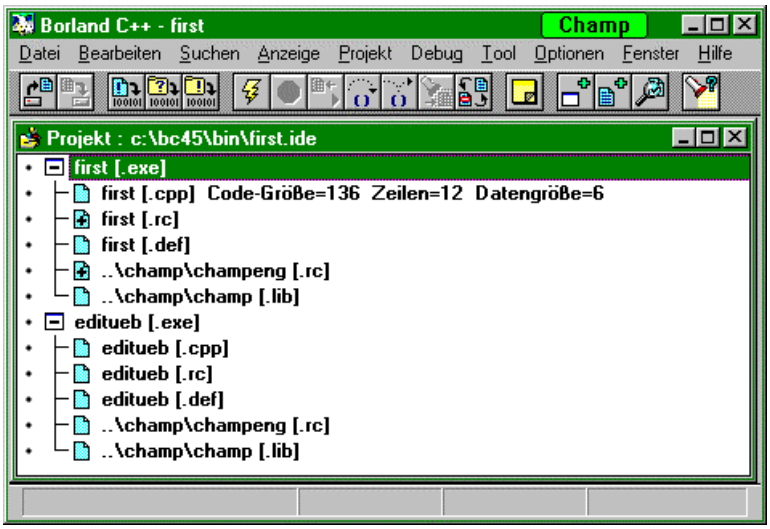


Haben Sie die Datei `editueb.cpp` korrekt bearbeitet, hat sie nun den folgenden Inhalt:

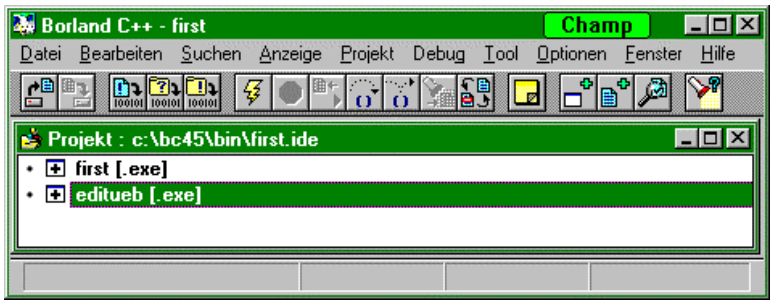
```
// EDITUEB.CPP
// Uebungsdatei zum BC++ Editor.


Verschieben Sie diesen Block an die Stelle, die
mit &&& markiert ist:

Markieren Sie diese zwei Zeilen mit der Maus.
Diese markierten Zeilen nennt man einen Block.

Markieren Sie hinterher den gleichen Block und kopieren Sie ihn an die
Stelle, die mit dem @@@ markiert ist:

Markieren Sie diese zwei Zeilen mit der Maus.
Diese markierten Zeilen nennt man einen Block.


Sie finden hier eine Schlange, die Gemuese enthaelt:

BlumenkohlWirzBuschbohnenTomatenKarottenKohlErbsenSchnittmangoldSellerie
LauchBroccoliAuberginenRettichRandenKopfsalatNuesslerSpinatKartoffelnZwiebeln

Ersetzen Sie mit Hilfe des Menuepunktes Suchen|Ersetzen...
das Wort 'Kabis' durch 'Kohl'.

Entnehmen Sie dann aus der Liste Ihr Liebliengsgemuese
und setzen Sie es an die folgende Stelle:

Lieblingsgemuese:

Karotten
```

2.5.4 Öffnen einer bestehenden Datei

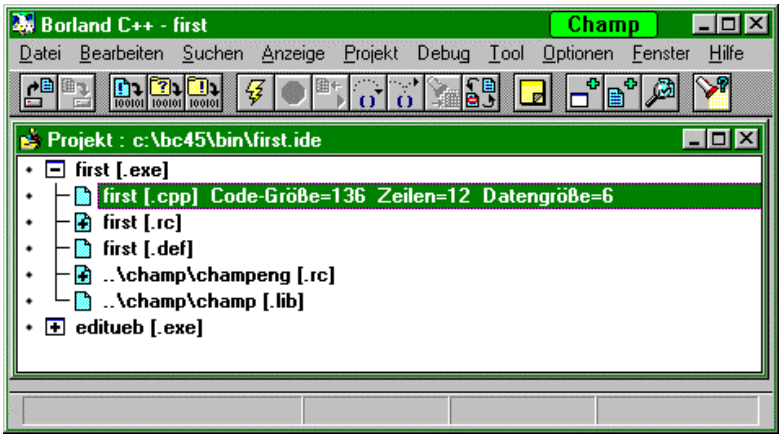
Soll eine bestehende Datei eines Projekts weiter bearbeitet werden, so muss das entsprechende Editierfenster unter Umständen wieder geöffnet werden. Am einfachsten geschieht dies mit Hilfe des Projektfensters:



Das Projekt `first` hat zwei **Knoten** (Zieldateien), `first` und `editueb`, erkennbar an dem Zeichen - im Projektfenster. Klickt man auf diese Zeichen so werden die **Unterknoten** nicht mehr gezeigt:



Das Zeichen + vor den Knoten zeigt an, dass diese Knoten (immer noch) Unterknoten besitzen, die mit einem Mausklick auf + wieder angezeigt werden können:



Mit einem Doppelklick auf die Zeile `first[.cpp] ...` kann die Datei `first.cpp` geöffnet und bearbeitet werden.

Mit einem Doppelklick auf `first[.exe]` wird das Programm kompiliert und ausgeführt (vgl. Kapitel 3.1.3).

3 Aufbau eines C++-Programms

3.1 Unser erstes Programm

Als Erstes schreiben wir ein Programm, das uns eine Treppe mit 5 Stufen zeichnen soll. Dabei soll auch der Entwicklungszyklus eines Programms vorgestellt werden: **Editieren** (d.h. Schreiben des Programms)**, Kompilieren, Ausführen, Fehlerbehandlung, Dokumentation**.

3.1.1 Editieren

Vorgehen:

1. Starten Sie die Borland C++- Programmierumgebung (siehe Kapitel 2.1).
2. Wählen Sie auf der Menüzeile Projekt|Neues Projekt. Es erscheint eine Dialogbox.
3. Geben Sie in der Dialogbox unter "Projektverzeichnis und -name" den gewünschten Namen des Projekts ohne Angabe eines Pfades oder einer Erweiterung an (z.B. `treppe1`). Bestätigen Sie mit der Return-Taste oder mit einem Mausklick auf OK. Die Dialogbox verschwindet.
4. Klicken Sie einmal mit der Maus auf die hellgrüne Champ-Schaltfläche. Eine neue Dialogbox erscheint. Es genügt für unser erstes Programm, wenn man in dieser Dialogbox mit der Maus auf OK klickt. Die Dialogbox verschwindet, statt dessen erscheint ein neues Fenster (das **Editierfenster** oder **Editor**) mit folgendem Programmtext:

```
/*************************************************************************
  PROGRAM  : TREPPE1.CPP
  COPYRIGHT:
  CODED BY :
  PURPOSE  :
  REMARKS  :
  COMPILE  : Borland C++, Version 4.5, large memory model
 *************************************************************************/

/*************************** Revision history ***************************
  REV   DATE         NAME  PURPOSE
  ----  ------------ ----  ------------------------------------------------
  1.00   2-JAN-1997
 *************************************************************************/

#include <champ.h>
#include "treppe1.rh"


void gmain ()
{
   ginit( "Champ" );




}
```

5. Geben Sie nun nach der letzten Zeile mit den Sternen folgende Zeile ein.

```
#define GLOBAL_TURTLE
```

6. Geben Sie nun das Hauptprogramm ein. Die *kursiv gedruckten Zeilen* sind schon geschrieben und müssen deshalb nicht noch einmal eingegeben werden!

```
void gmain ()
{
   ginit( "Champ" );
   speed( 100 );
   forward( 20 );
   right( 90 );
   forward( 20 );
   left( 90 );

   forward( 20 );
```

```
   right( 90 );
   forward( 20 );
   left( 90 );

   forward( 20 );
   right( 90 );
   forward( 20 );
   left( 90 );

   forward( 20 );
   right( 90 );
   forward( 20 );
   left( 90 );

   forward( 20 );
   right( 90 );
   forward( 20 );
   left( 90 );

   getch();
   gend();
}
```

Achten Sie darauf, dass Sie jedes Zeichen richtig eingeben (insbesondere die Semikolons und die Gross- und Kleinschreibung der einzelnen Zeichen).

Die Befehle (`forward`, `right`, `left` etc.) nennt man auch **Anweisungen**; am Schluss jeder Anweisung steht ein Semikolon. Eine Abfolge von Anweisungen nennt man **Anweisungsblock**, der immer zwischen geschweiften Klammern stehen muss.

Nun haben wir den **Quellcode** (auch **Quelltext, Quellprogramm** oder **Programmcode** genannt) eingegeben.

3.1.2 Kompilieren

Ist das Programm einmal geschrieben, will man es natürlich laufen lassen. Der Computer versteht aber das Programm nicht in der Form des Quelltextes. Deshalb müssen wir das Programm kompilieren, d.h. es muss durch den **Compiler** in die Maschinensprache (d.h. Sprache des Prozessors) übersetzt werden! Dies geschieht mit dem Befehl Projekt|Compilieren oder mit der Tastenkombination <Alt> + <F9>. Ein Fenster zeigt uns den aktuellen Stand der Kompilation an. Der Compiler erzeugt ein lauffähiges Programm, das den gleichen Namen wie das Quellprogramm hat, aber die Erweiterung **EXE** besitzt. In unserem Fall hat der Compiler die Datei `treppe1.exe` erzeugt.

Bei Beendigung oder Abbruch der Kompilation erscheint im Fenster eine OK-Schaltfläche. Mit einem Mausklick auf diese Schaltfläche oder mit der Return-Taste wird das Fenster geschlossen.

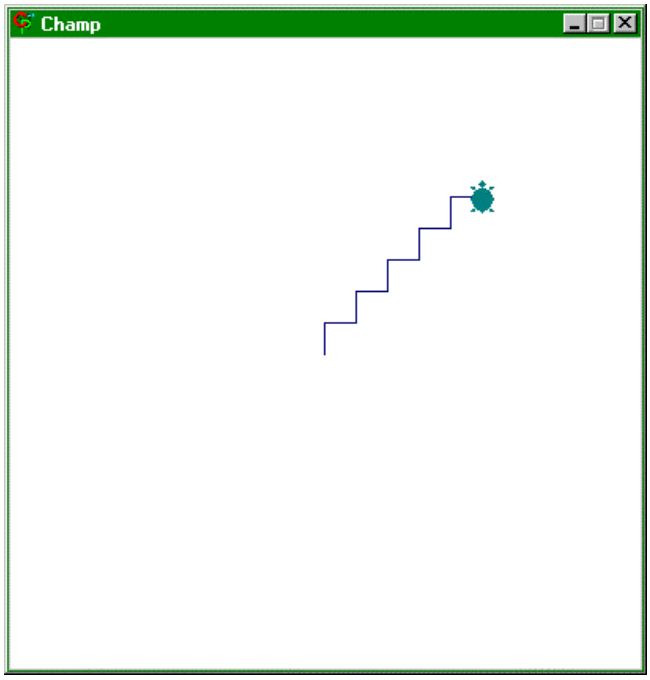
Stellt der Compiler einen syntaktischen Fehler (`Error`) oder eine programmiertechnische Unsauberkeit (`Warning`) fest, so wird uns in einem **Meldungsfenster** mitgeteilt, von welcher Art und wo der Fehler aufgetreten ist.

Ist das Programm korrekt, so erscheint zwar ebenfalls ein Meldungsfenster, aber die Mitteilungen sind für uns ohne Bedeutung.

Merke: Ein Programm ist erst dann korrekt, wenn nach der Kompilation keine Fehlermeldungen und auch keine Warnungen mehr angegeben werden!

Bemerkung: Im deutschen Sprachraum schreibt man (auch nach dem neuen Duden) "kompilieren" statt "compilieren". Die zweite Schreibweise wird aber in der Entwicklungsumgebung von Borland C++ verwendet.

3.1.3 Ausführen



Ist die Kompilation erfolgreich abgeschlossen, kann das Programm mit Debug|Ausführen gestartet werden. Auf dem Bildschirm erscheint ein neues Fenster, in dem eine Treppe gezeichnet wird. Durch Drücken einer beliebigen Taste kann das Programm beendet werden, das Fenster wird wieder geschlossen.

Ein Programm kann aber noch auf zwei weitere Arten gestartet werden:

1. Durch die Tastenkombination <Ctrl> + <F9>.
2. Durch einen Mausklick auf das entsprechende Symbol in der Mauspalette (vgl. Kapitel 2.3.2).

In beiden Fällen wird das Programm auch kompiliert, falls das vorher noch nicht geschehen ist. Mit andern Worten: Mit diesen zwei Möglichkeiten können die Schritte "Kompilation" und "Ausführen" zusammen ausgeführt werden. Die Ausführung des Programms erfolgt aber nur, wenn der Compiler keine Fehler festgestellt hat. Die Warnungen werden aber ignoriert!

Ein Programm kann durch Anklicken des entsprechenden Symbols in der Mauspalette beendet werden. Dies ist aber nur dann ratsam, wenn das Programm "hängt", d.h. wenn es auf Grund eines logischen Programmierfehlers nicht mehr anders beendet werden kann (vgl. Anhang Anhang E:).

3.1.4 Fehlerbehandlung

Treten beim Ausführen des Programms Fehler auf (sogenannte **Laufzeitfehler**), so müssen die Fehler gefunden und korrigiert werden. Manchmal liegt ein Fehler in der Angabe einer falschen Zahl oder einer falschen Variablen, oft aber auch in der fehlerhaften Programmierung eines Programms oder einer (mathematischen) Formel.

3.1.5 Dokumentation

Zu einem guten Programm gehört eine gute Dokumentation. Im Programmkopf, der in unserer Champ-Umgebung beim Erzeugen eines neuen Programms automatisch geschrieben wird, muss zumindest der Name des Programms und der Zweck des Programms angegeben werden. Gewöhnen Sie sich an, ein Programm zu dokumentieren. Es hilft Ihnen (und auch andern), Ihre Programme besser zu verstehen.

Das folgende Beispiel zeigt noch einmal das Programm `treppe1.cpp`, diesmal aber mit einer ausführlichen (schon bald zu ausführlichen) Dokumentation. Sie soll Ihnen aber helfen, die einzelnen Befehle zu verstehen.

```
// TREPPE1.CPP
// Zeichnet eine Treppe mit fuenf Stufen.

#define GLOBAL_TURTLE  // Sagt dem Compiler, dass Turtle-Graphik verwendet wird.

#include <champ.h>     // Die Dateien champ.h und treppe1.rh sollen in das
#include "treppe1.rh"  // Programm eingebunden werden.


void gmain ()          // Beginn des Hauptprogramms, gehoert zu jedem Programm.
{
   ginit( "Champ" );   // Erzeugt ein Graphikfenster mit dem Titel "Champ".
   speed( 100 );       // Gibt die Geschwindigkeit der Turtle an.

   forward( 20 );      // Turtle bewegt sich um 20 Einheiten vorwaerts.
   right( 90 );        // Turtle dreht sich um 90 Grad nach rechts.
   forward( 20 );
   left( 90 );         // Turtle dreht sich um 90 Grad nach links.

   forward( 20 );
```

```
   right( 90 );
   forward( 20 );
   left( 90 );

   forward( 20 );
   right( 90 );        // Allgemeine Bemerkung: Zu jeder Anweisung (d.h. zu
   forward( 20 );      // jedem Befehl) gehoert ein Semikolon.
   left( 90 );

   forward( 20 );
   right( 90 );
   forward( 20 );
   left( 90 );

   forward( 20 );
   right( 90 );
   forward( 20 );
   left( 90 );

   getch();            // Programm wartet auf einen Tastendruck.
   gend();             // Das Graphikfenster wird geschlossen.
}
```

Bemerkungen:

- Bei der Initialisierung des Graphikfensters steht zwischen den Klammern ein **String** (**Zeichenkette**), der den Titel des Graphikfensters enthält. Beachten Sie, dass Strings **zwischen Anführungszeichen** stehen müssen!
- Kommentar, der am Schluss einer Zeile angefügt wird, wird mit zwei Schrägstrichen `//` eingeleitet. Soll der Kommentarblock mehrere Zeilen lang sein, so schreibt man zu Beginn des Blocks die Zeichen `/*` und am Ende des Blocks die Zeichen `*/`. Es ist aber natürlich auch hier möglich, jede Kommentarzeilen mit den zwei Schrägstrichen (`//`) zu beginnen.
- Die Klammern bei den Befehlen `getch();` und `gend();` sind notwendig. Diese zwei Befehle sind, wie auch die Turtle-Befehle, Funktionsaufrufe und benötigen deshalb, wie alle Funktionsaufrufe, Klammern. Oft werden in die Klammern die Werte (**Parameter**) geschrieben, die an die Funktion übergeben werden.

- Werden bei einem Funktionsaufruf die Klammern vergessen, so führt dies nicht immer zu einer Fehlermeldung des Compilers, sondern zu einem Laufzeitfehler und einem Absturz des Programms!

3.2 Modularisierung

Die Aufgabe im ersten Programm lautete, eine Treppe mit fünf Stufen zu zeichnen. Diese Aufgabe kann man wie folgt angehen: Die Treppe besteht aus fünf Stufen. Wenn man also weiss, wie eine Stufe gezeichnet wird, dann kann man auch die Treppe zeichnen, indem fünfmal eine Stufe gezeichnet wird.

Mit andern Worten: Das Problem wird in **Teilprobleme** (**Module**) zerlegt, die normalerweise einfacher und übersichtlicher zu lösen sind als die ursprüngliche Aufgabe. Diesen Lösungsprozess nennt man **Modularisierung**. Unser Beispiel kennt nur ein Teilproblem des Problems "Treppe", nämlich das Problem "Stufe".

In C++ nennt man die Module **Funktionen**.

Beispiel: Wir lassen eine Treppe mit fünf Stufen zeichnen, indem wir eine Funktion `stufe` definieren:

```
// TREPPE2.CPP
// Verwendet die Funktion stufe, um eine Treppe mit 5 Stufen zu zeichnen.

#define GLOBAL_TURTLE

#include <champ.h>
#include "treppe2.rh"

void stufe ();              // Deklaration der Funktion stufe, achten Sie auf
                            // das Semikolon!
void gmain ()
{
   ginit( "Champ" );

   stufe();                 // Aufruf der Funktion stufe, achten Sie auf
   stufe();                 // die Klammern!
   stufe();
```

```
   stufe();
   stufe();

   CP::msgBoxD();           // Es erscheint ein Dialogfenster mit einer OK-Taste.
   gend();
}

void stufe ()               // Definition der Funktion stufe, diesmal ohne
{                           // Semikolon!
   forward( 20 );
   right( 90 );
   forward( 20 );
   left( 90 );
}
```

Bemerkungen:

- In der Mathematik kann man sich eine Funktion $y = f(x)$ als eine Maschine vorstellen, die einen Wert x aus der **Definitionsmenge** als Eingabe erhält und uns einen Wert y aus der **Wertemenge** zurückgibt. Eine Funktion in einem C++-Programm erfüllt im Prinzip denselben Zweck: In unserem speziellen Fall ist `stufe()` eine Funktion, die keine Eingabe erhält (deshalb die leeren Klammern) und auch keinen Wert (genauer: den Wert `void`[1]) zurückgibt.
- Eine Funktion muss vor dem Hauptprogramm `gmain` **deklariert** werden, damit der Compiler den Befehl `stufe()` als Funktionsaufruf erkennt und nicht eine Fehlermeldung zurückgibt. Die **Definition** (d.h. Ausprogrammierung der Funktion) erfolgt nach dem Hauptprogramm.
- Die Programmiersprache C++ lebt von Funktionsaufrufen. Es gibt nur wenige reservierte Wörter, aber eine grosse Anzahl von Bibliotheken (libraries) mit sauber definierten Funktionen. Auch unsere bis jetzt bekannten Befehle wie `forward` oder `right` sind Funktionsaufrufe.
- Eine Alternative zur Funktion `getch();` ist die Funktion `CP::msgBoxD();`. Sobald die Treppe gezeichnet ist, erscheint eine kleine Dialogbox (rechts), die einen OK-Knopf enthält. Das Programm hält an und fährt erst dann weiter, wenn mit der Maus auf OK geklickt oder die Return-Taste gedrückt wird.

3.3 Parameterübergabe

In der Mathematik nennt man x das **Argument** der Funktion $y = f(x)$. Es ist möglich, in C++ den Funktionen ebenfalls Werte zu übergeben, die man nicht Argumente, sondern **Parameter** (genauer: **Werteparameter**) der Funktion nennt.

Beispiel: Gezeichnet wird eine Stufe mit variablen Stufenhöhen.

```
// TREPPE3.CPP
// Uebergibt der Funktion stufe Parameterwerte, um eine Treppe mit 5 verschieden
// hohen Stufen zeichnen zu koennen.

#define GLOBAL_TURTLE

#include <champ.h>
#include "treppe3.rh"

void stufe ( int hoehe );   // Deklaration der Funktion stufe

void gmain ()
{
   ginit( "Champ" );

   stufe( 20 );             // Fuenfmaliger Aufruf der Funktion stufe, an die
   stufe( 30 );             // der Reihe nach die Werte 20, 30, 40, 10, 30
   stufe( 40 );             // uebergeben wird.
   stufe( 10 );
   stufe( 30 );

   CP::msgBoxD();
   gend();
}
```

[1] void (engl.): leer

```
void stufe ( int hoehe )    // Definition der Funktion stufe. "hoehe" nimmt
{                           // jeweils den Wert des uebergebenen Parameters an.
   forward( hoehe );
   right( 90 );
   forward( 20 );
   left( 90 );
}
```

Bemerkung:

Parameter werden deklariert, indem man ihren Namen und ihren (Daten-)Typ in der Funktionsdeklaration und auch im Kopf der Funktion angibt. In unserem Programm wird der Funktion `stufe` ein Parameter `hoehe` vom Typ **int** (Integer; d.h. ganze Zahl) übergeben. Mit andern Worten: `hoehe` ist ein Platzhalter für den übergebenen Wert; beim ersten Aufruf hat `hoehe` den Wert 20, beim zweiten Aufruf den Wert 30, etc.

3.4 Die Ausgabe auf dem Drucker

3.4.1 Ausdruck einer Graphik aus dem Graphikfenster

Die Ausgabe einer Graphik auf einen Drucker ist sehr einfach:

1. Starten Sie das gewünschte Programm (die Graphik wird gezeichnet).
2. Klicken Sie im Graphikfenster mit der Maus auf das Symbol oben links.
3. Wählen Sie im erscheinenden Menü (dem **Systemmenü**) den Punkt Print. Drücken Sie jeweils in den nun erscheinenden Fenstern OK. (Diese Fenster dienen zur Bestätigung des gewählten Druckers bzw. der Bestätigung der langen Wartezeit.)

3.4.2 Ausgabe einer Graphik direkt auf einem Drucker

Es ist möglich, die Graphik nur auf dem Drucker auszugeben: Ersetzt man im Programm `ginit();` und `gend();` durch die Befehle `pinit();` und `pend();`, so wird eine Graphik direkt auf dem Drucker ausgegeben. Der Druck erfolgt wesentlich schneller als mit der ersten Methode!

Beispiel: Das folgende Programm gibt eine Graphik direkt auf dem Drucker aus.

```
// TREPPE4.CPP
// Uebergibt der Funktion stufe Parameterwerte, um eine Treppe mit 5 verschieden
// hohen Stufen zu zeichnen. Die Treppe wird direkt auf dem Drucker ausgegeben!

#define GLOBAL_TURTLE

#include <champ.h>
#include "treppe4.rh"

void stufe ( int hoehe );   // Deklaration der Funktion stufe


void gmain ()
{
   pinit();                 // Graphik soll fuer einen Drucker vorbereitet werden.

   stufe( 20 );
   stufe( 30 );
   stufe( 40 );
   stufe( 10 );
   stufe( 30 );

   pend();                  // Abschluss der Vorbereitung --> Drucken!
}

void stufe ( int hoehe )
{
   forward( hoehe );
   right( 90 );
   forward( 20 );
   left( 90 );
}
```

Bemerkungen:

- Mit `pinit();` wird eine Graphik (vereinfacht gesagt) vom Bildschirm auf den Drucker umgeleitet.
- Mit `pend();` weist man das Programm an, die Graphik auszudrucken. **Wird im Programm der Befehl** `pend();` **vergessen, so wird nichts ausgedruckt!**
- In diesem Programm sind `getch();` bzw. `CP::msgBoxD();` nicht nötig, da auf dem Bildschirm vor dem Abschluss keine Ausgabe betrachtet werden soll (und auch nicht betrachtet werden kann).

3.4.3 Ausdrucken des Quellcodes

Noch einfacher ist die Ausgabe des eingegeben Programms: Wählen Sie in der Entwicklungsumgebung den Punkt Datei|Drucken und bestätigen sie in der nun erschienenen Dialogbox die gewählten Optionen mit OK.

3.4.4 Kopieren einer Graphik in ein anderes Dokument

Es kommt oft vor, dass man eine Graphik über die **Zwischenablage** in ein anderes Dokument (z.B. in eine Textverarbeitung) kopieren möchte:

1. Öffnen Sie in einem Textverarbeitungsprogramm (z.B. WordPad oder Word für Windows) ein neues Dokument.
2. Starten Sie in der Entwicklungsumgebung von Borland das gewünschte Graphikprogramm.
3. Öffnen Sie das Systemmenü des Graphikfensters (vgl. Kapitel 1.2.2)
4. Wählen Sie im Menü den Punkt Copy|Content. Die Graphik wird in die Zwischenablage gespeichert.
5. Wählen Sie in der Textverarbeitung den Punkt Bearbeiten|Einfügen. Die Graphik wird im neuen Dokument eingefügt.
6. Das Dokument kann jetzt weiter bearbeitet werden.

4 Variablen und wichtige Datentypen

4.1 Globale Variablen

Beispiel: Gezeichnet wird eine Treppe mit fünf verschieden hohen Stufen. Verwendet wird dabei eine **globale Variable** `hoehe`.

```
// VARIABL1.CPP
// Zeichnet eine Treppe, verwendet wird die globale Variable "hoehe".

#define GLOBAL_TURTLE

#include <champ.h>
#include "variabl1.rh"

void stufe ();              // Deklaration der Funktion stufe

int hoehe = 0;              // Definition und Initialisierung der Variablen hoehe

void gmain ()
{
   ginit( "Champ" );

   hoehe = 20;
   stufe();
   hoehe = 30;
   stufe();
   hoehe = 40;
   stufe();
   hoehe = 10;
   stufe();
   hoehe = 20;
   stufe();

   CP::msgBoxD();
   gend();
}

void stufe ()               // Definition der Funktion stufe. Der Wert von
{                           // "hoehe" wird im Hauptprogramm definiert und ist
   forward( hoehe );        // in der Funktion stufe bekannt.
   right( 90 );
   forward( 20 );
   left( 90 );
}
```

Bemerkungen:

- Die Variable `hoehe` wird ausserhalb des Hauptprogramms definiert und ist deshalb überall, also auch in allen Funktionen bekannt. Solche Variablen nennt man aus diesem Grund **globale Variablen**.
- Bei der **Definition** wird durch den Compiler ein Speicherplatz für die Variable reserviert.
- Ist der Wert einer Variablen schon zum Zeitpunkt ihrer Definition bekannt, so erfolgt ihre **Initialisierung** (d.h. erste Zuweisung eines Wertes) am besten ebenfalls in der Definition. Diese Initialisierung muss erfolgen, da sonst das Programm mit einem zufälligen Wert rechnet, der am reservierten Speicherplatz "gespeichert" ist.
- Die Verwendung von globalen Variablen ist schwerfällig, da die Variable im Hauptprogramm verändert werden muss, damit sie dann in der Funktion `stufe` verwendet werden kann. Das Programm `treppe3`, in dem Parameter an die Funktion übergeben werden, ist in dieser Hinsicht wesentlich eleganter.
- Die Verwendung von globalen Variablen ist nicht nur schwerfällig, sondern auch gefährlich, weil ihr Wert auch in Funktionen verändert werden kann. Solche Veränderungen, die oft auch unbeabsichtigt sind, nennt man **Seiteneffekte**. Die Wirkung solcher Seiteneffekte wird im folgenden Programm gezeigt.

Beispiel: Gezeichnet wird eine Treppe, wieder mit Hilfe der globalen Variablen `hoehe`, die aber durch einen Seiteneffekt verändert wird.

```
// VARIABL2.CPP
```

```
// Zeichnet eine Treppe, verwendet wird die globale Variable "hoehe". Zudem
// wird ein Seiteneffekt erzeugt.

#define GLOBAL_TURTLE

#include <champ.h>
#include "variabl2.rh"

void stufe ();              // Deklaration der Funktion stufe
void seiteneffekt ();

int hoehe = 0;              // Definition und Initialisierung der Variablen hoehe

void gmain ()
{
   ginit( "Champ" );

   hoehe = 20;
   stufe();
   hoehe = 30;
   stufe();
   hoehe = 40;
   stufe();
   hoehe = 10;
   stufe();
   hoehe = 20;
   stufe();

   CP::msgBoxD();
   gend();
}

void stufe ()               // Definition der Funktion stufe. Der Wert von
{                           // "hoehe" wird im Hauptprogramm definiert und ist
                            // in der Funktion stufe bekannt.
   seiteneffekt();          // Aufruf der Funktion seiteneffekt. Dort wird der
   forward( hoehe );        // Wert von hoehe veraendert (Seiteneffekt)!
   right( 90 );
   forward( 20 );
   left( 90 );
}

void seiteneffekt ()
{
   hoehe = 10;
}
```

4.2 Lokale Variablen

Beispiel: Es soll eine Treppe gezeichnet werden, deren Stufen doppelt so breit wie hoch sind. Dazu wird eine **lokale Variable** `breite` verwendet.

```
// VARIABL3.CPP
// Zeichnet eine Treppe, deren Stufen doppelt so breit wie hoch sind.

#define GLOBAL_TURTLE
#include <champ.h>
#include "variabl3.rh"

void stufe ( int hoehe );   // Deklaration der Funktion stufe

void gmain ()
{
   ginit( "Champ" );

   setPos( -100 , -100 );   // Positioniert Turtle an der Position (-100/-100)

   stufe( 20 );             // Fuenfmaliger Aufruf der Funktion stufe, an die
   stufe( 30 );             // der Reihe nach die Werte 20, 30, 40, 10, 30
   stufe( 40 );             // uebergeben wird.
   stufe( 10 );
   stufe( 30 );
```

```
   hideTurtle();            // Die Turtle wird "versteckt"
   CP::msgBoxD();
   gend();
}

void stufe ( int hoehe )    // Definition der Funktion stufe. "hoehe" nimmt
{                           // jeweils den Wert des uebergebenen Parameters an.
   int breite = 2 * hoehe;  // Definition und Initialisierung der lokalen
                             // Variable "breite".
   forward( hoehe );
   right( 90 );
   forward( breite );
   left( 90 );
}
```

Bemerkungen:

- Die Variable `breite` ist eine **lokale Variable**, d.h. sie ist nur innerhalb der Funktion `stufe` gültig.
- Es ist auch möglich, in einem Hauptprogramm lokale Variablen zu definieren. Diese sind nur dort gültig, d.h. sie sind in den Funktionen nicht bekannt.
- Der Befehl `setPos( -100 , -100 );` setzt die Turtle auf den Punkt (–100/–100). Wird wie in den bisherigen Programmen dieser Befehl weggelassen, so wird die Turtle auf den Ursprung gesetzt, der sich im Mittelpunkt des Graphikfensters befindet. Die Koordinaten des Fensters gehen in der *x*-Richtung und in der *y*-Richtung von –200 bis 200.
- Der Befehl `hideTurtle();` bewirkt, dass die Turtle im Graphikfenster nicht gezeichnet wird. Soll sie nach einem "Verstecken" wieder gezeigt werden, muss der Befehl `showTurtle();` eingegeben werden.

Verwendet man nur lokale Variablen, so ist auch die Gefahr von Seiteneffekten gebannt.

Beispiel:

```
// VARIABL4.CPP
// Zeichnet eine Treppe, deren Stufen doppelt so breit wie hoch sind.

#define GLOBAL_TURTLE

#include <champ.h>
#include "variabl4.rh"

void stufe ( int hoehe );   // Deklaration der Funktion stufe

void seiteneffekt ();

void gmain ()
{
   ginit( "Champ" );
   setPos( -100,-100 );

   stufe( 20 );
   stufe( 30 );
   stufe( 40 );
   stufe( 10 );
   stufe( 30 );

   CP::msgBoxD();
   gend();
}

void stufe ( int hoehe )
{
   int breite = 2 * hoehe;

   seiteneffekt();              // Aufruf der Funktion "seiteneffekt".
   forward( hoehe );
   right( 90 );
   forward( breite );
   left( 90 );
}

void seiteneffekt ()            // Funktion "seiteneffekt", mit einer lokalen
{                               // Variablen "breite", die einen andern
```

```
   int breite = 10;             // Speicherplatz belegt als die gleichnamige
}                               // Variable in der Funktion "stufe".
```

Bemerkung:

Die Funktionen `stufe` und `seiteneffekt` kennen beide eine lokale Variable `breite`. Da sie aber durch den Compiler verschiedene Speicherplätze zugeordnet erhalten, haben sie aufeinander keinen Einfluss. Mit andern Worten:

Wird in der Funktion `seiteneffekt` die lokale Variable `breite` verändert, so hat dies auf die gleichnamige lokale Variable in der Funktion `stufe` keinen Einfluss. Der Wert dieser Variablen bleibt unverändert.

4.3 Datentypen

Damit bei der Kompilation des Programms der Speicherplatz korrekt einer Variablen zugeordnet werden kann, ist bei ihrer Definition die Angabe des **Datentyps** nötig. Die wichtigsten Datentypen sind:

- `int` (Abkürzung für **integer**): Eine Integer-Zahl ist ein ganze Zahl, die Werte zwischen –32’768 und 32'767 annehmen kann. Ist klar, dass nur positive ganze Zahlen gebraucht werden, so kann eine Variable vom Typ `unsigned int` verwendet werden, sie kann Werte zwischen 0 und 65'535 annehmen.
- `long`: Oft ist der Zahlenbereich für Integer-Zahlen zu klein. In diesem Fall definiert man ein Variable vom Typ `long`, sie kann Werte zwischen -2'147'483’648 und 2'147'483'647 annehmen. Analog zum Datentyp `unsigned int` gibt es einen Typ `unsigned long` (Werte von 0 bis 4'294'967'295).
- `char` (Abkürzung für **character**): Eine Variable vom Typ `char` enthält als Werte ASCII-Zeichen (z.B. Buchstaben), die aber auch als ganze Zahlen zwischen –128 und 127 interpretiert werden können. (Näheres dazu im Kapitel 6.3.)
- `float`: Eine Zahl von diesem Typ ist eine reelle Zahl zwischen $\pm 3.4 \cdot 10^{-38}$ und $\pm 3.4 \cdot 10^{38}$. Sie ist aber nur auf sieben wesentliche Stellen genau (wesentliche Stellen: ohne führende Nullen).
- `double`: Mit diesem Datentyp erreicht man eine genauere Angabe (auf 17 Stellen genau) einer reellen Zahl (von $\pm 1.7 \cdot 10^{-308}$ bis $\pm 1.7 \cdot 10^{308}$).
- `complex`: Komplexe Zahlen können durch eine Variable vom Typ `complex` (z.B. `z`) dargestellt werden.
- `bool`: Variablen vom Typ `bool` sind logische Variablen und können nur die zwei Werte `true` (wahr) und `false` (falsch) annehmen.

5 Iterationen

Der Ablauf eines **strukturierten Programms** besteht im Wesentlichen aus drei Elementen:

1. **Sequenz** (Abfolge): Die Befehle werden der Reihe nach ausgeführt.
2. **Iteration** (Wiederholung): Die Befehle werden wiederholt.
3. **Selektion** (Verzweigung): Je nach Bedingung werden Programmteile ausgeführt oder nicht.

Ein strukturiertes Programm ist nur aus diesen drei Elementen aufgebaut. Allerdings können sie ineinander verschachtelt werden, d.h. ein Programm kann eine Sequenz von mehreren Iterationen, eine Iteration selbst kann eine Sequenz von Befehlen enthalten. Ältere Programmiersprachen (z.B. alte BASIC-Dialekte, FORTRAN, aber auch Assemblersprachen) kennen zusätzlich noch Sprungbefehle, die aber unsaubere Programmiergewohnheiten zuliess (sogenannte Spaghetticodes). Auch strukturierte Programmiersprachen wie C++ und Pascal kennen Sprungbefehle, sie sollten aber sehr zurückhaltend und nur in bestimmten Situationen verwendet werden (z.B. Programmabbruch wegen eines Fehlers).

In diesem Kapitel sollen die Möglichkeiten vorgestellt werden, wie eine Iteration programmiert werden kann.

5.1 Die repeat-Anweisung

Die einfachste Möglichkeit, Wiederholungen zu programmieren, ist die **repeat**-Anweisung. Sie wird dann verwendet, wenn wir wissen, wie oft eine Anweisung oder ein Anweisungsblock ausgeführt werden soll. Das Programm `treppe2.cpp` (auf Seite 24) wird dadurch einfacher und übersichtlicher:

Beispiel:

```
// REPEAT1.CPP
// Zeichnet eine fuenfstufige Treppe, mit Hilfe der repeat-Anweisung.

#define GLOBAL_TURTLE

#include <champ.h>
#include "repeat1.rh"

void stufe ( int hoehe );

void gmain ()
{
   ginit( "Champ" );

   repeat ( 5 )
      stufe( 20 );

   CP::msgBoxD();
   gend();
}

void stufe ( int hoehe )
{
   forward( hoehe );
   right( 90 );
   forward( 20 );
   left( 90 );
}
```

Bemerkungen:

- Die *repeat*-Anweisung ist keine in C oder C++ definierte Anweisung, sondern sie wird uns von Champ zur Verfügung gestellt. Dazu ist entweder die Zeile `#define GLOBAL_TURTLE` oder, wenn keine Turtle-Graphik verwendet werden soll, die Zeile `#include <cprepeat.h>` nötig (vgl. Anhang Anhang D:).

- In diesem Programm wird der Befehl `stufe( 20 );` fünfmal durchgeführt, gezeichnet wird also eine Treppe mit fünf (gleich hohen) Stufen.

Sollen mehrere Befehle wiederholt werden, so sind geschweifte Klammern nötig:

Beispiel:

```
// REPEAT2.CPP
// Zeichnet eine Treppe mit abwechselnder Stufenhoehe (20 - 30 - 20 - 30 etc.).

#define GLOBAL_TURTLE

#include <champ.h>
#include "repeat2.rh"

void stufe ( int hoehe );

void gmain ()
{
   ginit( "Champ" );

   setPos( -100, -100 );        // TURTLE wird so positioniert, damit die
   repeat ( 5 )                 // ganze Treppe im Fenster Platz hat.
   {
      stufe( 20 );              // Der Anweisungsblock, der zur
      stufe( 30 );              // repeat-Anweisung gehoert, wird zwischen
   }                            // geschweifte Klammern gesetzt.

   CP::msgBoxD();
   gend();
}

void stufe ( int hoehe )
{
   forward( hoehe );
   right( 90 );
   forward( 20 );
   left( 90 );
}
```

Die geschweiften Klammern, die den Anweisungsblock, der wiederholt werden soll, begrenzen, sind in diesem Programm notwendig. Werden sie vergessen, so wird nur die erste Anweisung nach `repeat`, in diesem Fall die Zeile `stufe( 20 );`, wiederholt. Es ist auch möglich, im ersten Programm die (einzige) Anweisung, die wiederholt werden soll, zwischen geschweifte Klammern zu setzen.

5.2 Die while-Anweisung

Häufig wird eine Schleife (Iteration) solange wiederholt, wie eine bestimmte Bedingung (die **Laufbedingung**) erfüllt ist. Die Laufbedingung kann zu Beginn der Schleife (**while**-Anweisung) oder am Schluss der Schleife (**do while**-Anweisung) geprüft werden.

Die while-Schleife wird auch **vorprüfende Schleife** genannt und hat folgende Form:

`while (` *Laufbedingung* `)` *Anweisung*`;`

Bemerkungen:

- Die Laufbedingung wird in Klammern gesetzt.
- Statt der Anweisung kann auch ein ganzer Anweisungsblock stehen, der wiederum zwischen geschweifte Klammern gesetzt werden muss.

Beispiel: Es soll eine Treppe mit aufsteigender Stufenhöhe (10, 20, 30, 40, 50) gezeichnet werden.

```
// WHILE.CPP
// Zeichnet eine fuenfstufige Treppe, mit aufsteigender Stufenhoehe

#define GLOBAL_TURTLE

#include <champ.h>
#include "while.rh"

void stufe ( int hoehe );

void gmain ()
{
   int h = 10;                  // Definition und Initialisierung von h

   ginit( "Champ" );
```

```
   while ( h <= 50 )
   {
      stufe( h );
      h = h + 10;               // Die Variable h wird um 10 erhoeht.
   }

   CP::msgBoxD();
   gend();
}

void stufe ( int hoehe )
{
   forward( hoehe );
   right( 90 );
   forward( 20 );
   left( 90 );
}
```

Bemerkung:

Es ist wichtig, dass in der while-Schleife Variablenwerte so verändert werden können, dass die Laufbedingung irgendwann nicht mehr erfüllt sein wird. Ist das nicht der Fall, so läuft die Schleife endlos weiter (**Endlosschleife**). Das Programm kann aber noch mit <Ctrl> + <Alt> + <Del> abgebrochen werden (vgl. Seite 10).

In unserem Programm wird zu diesem Zweck die Variable `h` erhöht.

Warnung: Wird aus Versehen trotzdem eine Endlosschleife programmiert, so kann das Programm unter Umständen nicht mehr angehalten werden (auch nicht durch Schliessen des Fensters mit der Maus); das ganze System stürzt ab! Um das zu vermeiden, sollte in jeder Schleife sicherheitshalber der Befehl `CP::yield();` stehen!

`CP::yield();` unterbricht für einen Moment den Programmablauf, damit das Betriebssystem weiterhin andere Abläufe des Computers kontrollieren kann (vgl. Kapitel 1.2.3). Da auch diese Abläufe die Kontrolle dem Betriebssystem zurückgeben, kann unser Programm meist schon nach sehr kurzer Zeit (d.h. nach Bruchteilen von Sekunden) weiterfahren.

`CP::yield();` ist in der Praxis aber nur dann nötig, wenn die Schleife nur Ausgaben in der Textkonsole enthält (vgl. Kapitel 6.1). Wird in einem Graphikfenster gezeichnet oder wird in einer Schleife eine Eingabe verlangt, dann ist `CP::yield();` nicht nötig.

5.3 Die do while-Anweisung

Im Unterschied zur vorabprüfenden while-Schleife ist die **do while**-Schleife eine **nachprüfende Schleife**, d.h. die Laufbedingung wird am Schluss der Schleife geprüft. Das hat zur Folge, dass eine *do while*-Schleife mindestens einmal durchlaufen wird. (Eine *while*-Schleife wird unter Umständen gar nie durchlaufen.)

Die *do while*-Schleife hat folgende Form:

`do` *Anweisung* `while (` *Laufbedingung* `);`

```
// DOWHILE.CPP
// Zeichnet eine fuenfstufige Treppe, mit aufsteigender Stufenhoehe.

#define GLOBAL_TURTLE
#include <champ.h>
#include "dowhile.rh"

void stufe ( int hoehe );

void gmain ()
{
   int h = 10;                  // Definition und Initialisierung von h

   ginit( "Champ" );

   do
   {
      stufe( h );
      h = h + 10;               // Die Variable h wird um 10 erhoeht.
   } while ( h <= 50 );
```

```
   CP::msgBoxD();
   gend();
}

void stufe ( int hoehe )
{
   forward( hoehe );
   right( 90 );
   forward( 20 );
   left( 90 );
}
```

Bemerkungen:

- An Stelle einer Anweisung kann (wie bei der *while*-Schleife) ein ganzer Anweisungsblock stehen, der natürlich wieder zwischen geschweiften Klammern stehen muss.
- Achten Sie auf das Semikolon nach der Laufbedingung, die ihrerseits zwischen Klammern stehen muss.

5.4 Die for-Anweisung

In der Mathematik werden häufig **Zählschleifen** gebraucht (z.B. Berechnung von Summen). Mit einer **for**-Schleife können wir diese Zählschleifen gut programmieren. Die *for*-Anweisung hat folgende Form:

`for (` *Initialisierung*`;` *Bedingung*`;` *Inkrementierung* `)` *Anweisung*`;`

Beispiel:

```
// FOR.CPP
// Zeichnet eine fuenfstufige Treppe, mit aufsteigender Stufenhoehe.

#define GLOBAL_TURTLE

#include <champ.h>
#include "for.rh"

void stufe ( int hoehe );

void gmain ()
{
   ginit( "Champ" );

   for ( int h = 10; h <= 50; h = h + 10 )
      stufe( h );

   CP::msgBoxD();
   gend();
}

void stufe ( int hoehe )
{
   forward( hoehe );
   right( 90 );
   forward( 20 );
   left( 90 );
}
```

Ablauf der *for*-Anweisung und **Bemerkungen**:

1. Zuerst wird die **Initialisierung** (hier: `int h = 10`) (falls vorhanden) ausgeführt. Dabei wird normalerweise der **Schleifenzähler** (hier: `h`) definiert und initialisiert.
2. Als nächstes wird die **Bedingung** (hier: `h <= 50`), die eigentlich nichts anderes als eine Laufbedingung ist, ausgewertet.
3. Ist die Laufbedingung erfüllt, so wird die Schleifenanweisung ausgeführt, andernfalls endet die *for*-Schleife. Die Schleifenanweisung kann wieder ein ganzer Anweisungsblock sein. In diesem Fall darf man die geschweiften Klammern nicht vergessen.
4. Die **Inkrementierung** (hier: `h = h + 10`) gibt an, um welchen Wert der Schleifenzähler erhöht (oder auch erniedrigt) werden soll.

5. Die Steuerung kehrt zu Schritt 2 zurück.

Zusätzliche Bemerkungen:

- Initialisierung, Bedingung und Inkrementierung werden je durch ein Semikolon abgetrennt.
- Der Schleifenzähler kann auch ausserhalb der *for*-Schleife definiert werden. Dies ist vor allem dann sinnvoll, wenn das Programm mehrere *for*-Schleifen enthalten soll. Ein solches Programm ist das Programm `ausgabe2.cpp` im Kapitel 6.2.2.
- Wie bei der *while*-Schleife und der *do while*-Schleife muss man aufpassen, dass keine Endlosschleife programmiert wird.
- Im Unterschied zu anderen Programmiersprachen (BASIC, Pascal) wird die Laufbedingung und nicht die Abbruchbedingung angegeben (Pascal: `for i := 1 to 50`).

5.5 Einige Bemerkungen zur Darstellung eines Programms

Theoretisch ist es möglich, das Programm `dowhile.cpp` auch so einzugeben:

```
// DOWHILE.CPP Zeichnet eine fuenfstufige Treppe, mit aufsteigender Stufenhoehe.
#define GLOBAL_TURTLE
#include <champ.h>
#include "dowhile.rh"
void stufe(int hoehe);void gmain(){int h=10;ginit("Champ");do{stufe(h);h=h+10;}
while(h<=50);CP::msgBoxD();gend();}void stufe(int hoehe){forward(hoehe);right(90);
forward( 20 );left( 90 );}
```

Dieses Programm ist lauffähig und zeichnet genau dieselbe Treppe wie das Programm auf der Seite 34, aber es ist unübersichtlich. Deshalb halten sich die Programmierer an einige Darstellungsregeln, die hier vorgestellt werden.

- Schreiben Sie pro Zeile nur eine Anweisung!
- Die geschweiften Klammern, die den Anfang und das Ende eines Anweisungsblocks anzeigen, werden untereinander, d.h. in derselben Spalte geschrieben, die Anweisungen werden eingerückt (2-3 Leerschläge):

```
{
   Anweisung1;
   Anweisung2;
   .
   .
   .
   Anweisungn;
}
```

- Anweisungen und Anweisungsblöcke, die z.B. zu einer Schleife gehören, werden noch mehr eingerückt:

```
{
   Anweisung1;
   for ( ... )
      Anweisung2;
   Anweisung3;
   do
   {
      Anweisung4;
      Anweisung5;
   } while ( ... );
   Anweisung6
}
```

- Gehen Sie nicht zu geizig mit Leerschlägen um: `for ( int i = 10; i <= 50; i = i + 10 )` ist besser als `for(int i=10; i<=50; i=i+10)`.
- Auch mit Leerzeilen kann ein Programm optisch gegliedert werden.
- In der Regel werden lokale Variablen zu Beginn einer Funktion, manchmal aber auch irgendwo innerhalb der Funktion definiert. Die **Scope-Regeln** (d.h. den Regeln über die Gültigkeit und Lebensdauer von Variablen) sagen aus, dass Variablen bis zum Ende des Anweisungsblocks gültig sind, in dem sie definiert werden.

- Die Namen von Funktionen und Variablen sollten mit Kleinbuchstaben geschrieben werden. Wird ein Name aus mehreren Wörtern zusammengesetzt (z.B. "`anzahlStufen`") so werden diese in einem Wort geschrieben, wobei der erste Buchstabe des zweiten Wortes gross geschrieben wird. Veraltet sind die Schreibweisen wie `anzahl_stufen` oder `AnzahlStufen`.

- Bei Funktionsdeklarationen und Funktionsköpfen ist es üblich, zwischen dem Funktionsnamen und der Klammer mit der Parameterliste einen Leerschlag zu schreiben:

  ```
  void stufe ( int hoehe );
  ```

  Beim Aufruf der Funktion wird aber kein Leerschlag eingegeben:

  ```
  stufe( 30 );
  ```

  Der Grund dieser Unterscheidung liegt in einer effizienteren Suche nach den Aufrufen einer Funktion bzw. der Deklaration einer Funktion bei grösseren Projekten (Suche nach "`stufe(`" bzw. nach "`stufe (`").

6 Tastatureingabe und Bildschirmausgabe

6.1 Einfache Ein- und Ausgabe

Beispiel:

```
// EINGABE1.CPP
// Zeichnet eine Treppe, die Anzahl der Stufen wird vom Anwender eingegeben.

#define GLOBAL_TURTLE

#include <champ.h>
#include "eingabe1.rh"

void stufe ( int hoehe );

void gmain ()
{
   int anzahl;                          // Anzahl der Stufen
                                        // Initialisierung des Textfensters,
   cinit( "Textfenster" );              // mit der Ueberschrift "Textfenster".

   cout << "Dieses Programm zeichnet eine Treppe.\n";
   cout << "Wie viele Stufen soll die Treppe haben? ";
   cin >> anzahl;

   ginit( "Champ" );
   repeat ( anzahl )                    // Wiederhole 'anzahl'-mal
      stufe( 20 );

   cout << "Die Treppe ist jetzt gezeichnet!\n";
   cout << "Druecken Sie eine Taste, um das Graphikfenster zu loeschen! ";
   cgetch();                            // Aktiviert das Textfenster und
   gend();                              // wartet auf einen Tastendruck

   cout << "\n\nDruecken Sie nun eine Taste, um das Textfenster zu loeschen\n";
   cout << "und das Programm zu beenden!";
   cgetch();
   cend();                              // Loescht das Textfenster.
}

void stufe ( int hoehe )
{
   forward( hoehe );
   right( 90 );
   forward( 20 );
   left( 90 );
}
```

Bemerkungen:

- Mit dem Befehl `cinit( "Textfenster" );` wird ein **Konsolenfenster** (d.h. ein **Textfenster**) geöffnet, das den Titel "Textfenster" hat. Wird der Titel des Fensters weggelassen (mit: `cinit();`), wählt das Programm den vordefinierten Titel "Champ Console Window". Dasselbe geschieht auch, wenn bei `ginit();` der Titel weggelassen wird ("Champ Graphics Window"). Das Textfenster wird mit `cend();` geschlossen.
- Im Konsolenfenster sollte statt `getch();` der Befehl `cgetch();` verwendet werden. `cgetch();` hat im Wesentlichen die selbe Wirkung wie `getch();`, aktiviert aber vorher das Konsolenfenster. Noch besser ist aber auch hier die Verwendung von `CP::msgBoxD();`.
- Die Ausgabe eines Textes geschieht mit dem sogenannten **Streamobjekt** (kurz **Stream**) `cout`. Zu diesem Stream gehört der Operator `<<` (**Einfügeoperator**, **Inserter**, dem dann der auszugebende Text, den man auch "Stream" ("Strom") nennt, **zwischen Anführungszeichen** angefügt wird.

- Zusätzlich zum gewünschten Ausgabetext können zwischen den Anführungszeichen auch **Bildschirmsteuerzeichen** angegeben werden. Eines dieser Steuerzeichen, die übrigens mit einem **Backslash** (`\`) beginnen, haben wir in unserem Programm verwendet: `\n` bewirkt, dass der nachfolgende Text am Beginn der nächsten Zeile beginnt. Weitere wichtige Steuerzeichen sind in der Tabelle rechts angegeben.
- Die Eingabe eines Variablenwertes geschieht mit dem Streamobjekt `cin >>`. Den Operator `>>` nennt man auch **Extractor**.
- Mit der Hilfe des Systemmenüs des Konsolenfensters kann der Text entweder gedruckt oder markiert und via Zwischenablage in eine andere Anwendung (z.B. Word für Windows) kopiert werden.

Bildschirmsteuerzeichen:	
`\n`	Neue Zeile
`\t`	Gehe zum nächsten horizontalen Tabulator
`\b`	Backspace (lösche das letzte Zeichen)
`\f`	Form feed (Drucker: gehe zur nächsten Seite)
`\a`	Gibt ein akustisches Zeichen
`\\`	Schreibt einen Backslash
`\'`	Schreibt ein einfaches Anführungszeichen
`\"`	Schreibt ein doppeltes Anführungszeichen
`\0`	Schreibt einen **Null-Character** (vgl. Kapitel 8.5)

Es ist natürlich auch möglich, Variablenwerte auf dem Bildschirm auszugeben:

Beispiel: Das folgende Programm berechnet die Hypotenuse eines rechtwinkligen Dreiecks.

```
// PYTHAG1.CPP
// Berechnet die Hypotenuse eines rechtwinkligen Dreiecks aus seinen Katheten.

#include <champ.h>
#include "pythag1.rh"

void gmain ()
{
   double a, b;                          // Definition der Katheten a und b.
   double c;                             // Definition der Hypotenuse.

   cinit( "Pythagoras" );
   cout << "Dieses Programm berechnet die Hypotenuse c eines rechtwinkligen\n";
   cout << "Dreiecks aus seinen Katheten a und b.\n\n";
   cout << "Geben Sie die Laengen der Katheten ein:\n";
   cout << "a = ";
   cin >> a;
   cout << "b = ";
   cin >> b;

   c = sqrt( pow( a, 2 ) + pow( b, 2 ) );    // Berechnung der Hypotenuse

   cout << "\nLaenge der Hypotenuse: c = " << c;
   CP::msgBoxD();
   cend();
}
```

Bemerkungen:

- Der Befehl `#define GLOBAL_TURTLE` ist in diesem Programm nicht nötig, da keine Turtle-Graphik verwendet wird.
- Die Variablen `a`, `b` und `c` sollen reelle Zahlen von doppelter Genauigkeit, d.h. vom Typ *double* sein.
- Zur Berechnung der Hypotenuse sind zwei neue Befehle nötig:
  - `sqrt( ... )`: Berechnung einer Quadratwurzel (engl.: "square root"). In der Klammer steht der Ausdruck, dessen Wurzel berechnet werden soll.
  - `pow( x, y )`: Berechnet die Potenz x^y. In unserem Programm werden a^2 (`pow( a, 2 )`) und b^2 (`pow( b, 2 )`) berechnet.
- Es ist ohne Probleme möglich, nach der Ausgabe der Variablen noch Text anzufügen: (Zum Beispiel mit `cout << "c = " << c << " cm ";`) oder die Variable sogar ohne begleitenden Text auszugeben (`cout << c;`). Die letzte Möglichkeit entspricht aber nicht dem Stil eines guten Programms!
- Es ist möglich, aber nicht "mit Stil", beim Befehl `cin` mehrere Variablen einzugeben.

Beispiel:

```
cout << "Geben Sie die Laengen der Katheten ein:\n";
cin >> a >> b;
```

Der erste eingegebene Wert wird der Variablen `a` zugeordnet, der zweite Wert der Variablen `b`.

- Falls die Ausgabe in einer Schleife programmiert wird, so muss die Schleife mit `CP::yield();` ergänzt werden, damit das Programm unterbrochen werden kann, falls die Schleife aus Versehen als eine Endlosschleife programmiert worden ist (vgl. Kapitel 5.2).

6.2 Formatierte Ausgabe

Oft kommt es vor, dass man die Bildschirmausgabe "schön" darstellen will; die Ausgabe der Werte in Tabellenform, rechtsbündige Ausgabe bei einer unterschiedlichen Anzahl Stellen oder eine feste Anzahl Nachkommastellen sind einige Beispiele dafür. Für diese **Formatierung** der Ausgabe gibt es spezielle Befehle.

6.2.1 Gewünschte Genauigkeit

Beispiel:

```
// AUSGABE1.CPP
// Dieses Programm zeigt verschiedene Formen von formatierter Ausgabe.

#include <champ.h>
#include "ausgabe1.rh"

void ausgabe ( double piDouble, float piFloat, double a, double b,
               double c, double d, int e );

void gmain ()
{
   double piDouble = 3.141592653589793;          // "genaues" PI
   float piFloat = 3.141592;                     // "ungenaues" PI
   double a = 2.0 / 3.0;
   double b = 8.0 / 3.0;
   double c = 9.0 / 3.0;
   double d = 7.0 / 4.0;
   int e = 6 / 3;

   cinit( "Formatierte Ausgabe" );

   cout << "Unformatierte Ausgabe: \n";
   ausgabe( piDouble, piFloat, a, b, c, d, e );

   cout << "Auf 4 wesentliche Stellen genau (ohne fixed/showpoint): \n";
   cout << setprecision( 4 );
   ausgabe( piDouble, piFloat, a, b, c, d, e );

   cout << "Auf 4 wesentliche Stellen genau (mit showpoint): \n";
   cout << setiosflags( ios::showpoint ) << setprecision( 4 );
   ausgabe( piDouble, piFloat, a, b, c, d, e );
   cout << resetiosflags( ios::showpoint );

   cout << "Auf 4 Nachkommastellen genau (mit fixed): \n";
   cout << setiosflags( ios::fixed );
   ausgabe( piDouble, piFloat, a, b, c, d, e );

   cout << "Auf 10 Nachkommastellen genau: \n";
   cout << setprecision( 10 );
   ausgabe( piDouble, piFloat, a, b, c, d, e );

   cout << "Auf 10 wesentliche Stellen genau: \n";
   cout << resetiosflags( ios::fixed );
   ausgabe( piDouble, piFloat, a, b, c, d, e );

   cout << "Wissenschaftliche Schreibweise: \n";
   cout << setiosflags( ios::scientific );
   ausgabe( piDouble, piFloat, a, b, c, d, e );

   cout << "Zurueckschalten in Normalschreibweise: \n";
```

```
   cout << resetiosflags( ios::scientific );
   ausgabe( piDouble, piFloat, a, b, c, d, e );

   CP::msgBoxD();
   cend();
}

void ausgabe ( double piDouble, float piFloat, double a, double b,
               double c, double d, int e )
{
   cout << "PI1 = " << piDouble << "\n";
   cout << "PI2 = " << piFloat << "\n";
   cout << "2/3 = " << a << "\n";
   cout << "8/3 = " << b << "\n";
   cout << "9/3 = " << c << "\n";
   cout << "7/4 = " << d << "\n";
   cout << "6/3 = " << e << "\n\n";
}
```

Bemerkungen:

- Die erste Ausgabe der Werte ist eine Ausgabe ohne Formatbefehle. Das bedeutet, dass alle Werte (mit Ausnahme der *int*-Variablen `e`) auf sechs wesentliche Stellen (d.h. ohne führende Nullen) ausgegeben werden, wobei "angehängte Nullen" ebenfalls nicht auf den Bildschirm geschrieben werden:

```
Unformatierte Ausgabe:
PI1 = 3.14159
PI2 = 3.14159
2/3 = 0.666667
8/3 = 2.66667
9/3 = 3
7/4 = 1.75
6/3 = 2
```

- Mit dem Befehl `setprecision( n )` können alle folgenden Werte (bis auf "Gegenbefehl") auf *n* wesentliche Stellen angegeben werden, nachführende Nullen werden nicht ausgegeben. Ohne Angabe der Anzahl wesentlicher Stellen wird diese vom Compiler auf sechs festgelegt.

```
Auf 4 wesentliche Stellen genau (ohne fixed/showpoint):
PI1 = 3.142
PI2 = 3.142
2/3 = 0.6667
8/3 = 2.667
9/3 = 3
7/4 = 1.75
6/3 = 2
```

- Der Befehl `setiosflags( ios::showpoint )` bewirkt, dass nachführende Nullen ebenfalls ausgegeben werden (z.B. `Bob Beamon sprang an der Olympiade 8.90 m weit.` statt `... 8.9 m weit.`). Der Befehl `resetiosflags( ios::showpoint )` schaltet diese Möglichkeit wieder aus. Ausgabe:

```
Auf 4 wesentliche Stellen genau (mit showpoint):
PI1 = 3.142
PI2 = 3.142
2/3 = 0.6667
8/3 = 2.667
9/3 = 3.000
7/4 = 1.750
6/3 = 2
```

- Mit dem Befehl `setiosflags( ios::fixed )`werden die Werte auf diejenige Anzahl Nachkommastellen genau ausgegeben, die mit `setprecision( n )` definiert worden ist. "Angehängte" Nullen werden dabei ausgegeben. Mit `resetiosflags( ios::fixed )`wird diese Option abgeschaltet.

```
Auf 4 Nachkommastellen genau (mit fixed):
PI1 = 3.1416
PI2 = 3.1416
2/3 = 0.6667
```

```
8/3 = 2.6667
9/3 = 3.0000
7/4 = 1.7500
6/3 = 2
```

- Beim nächsten Block wird nur die Anzahl der Nachkommastellen geändert. Es ist aber nicht sinnvoll, eine *float*-Variable auf mehr als sechs Nachkommastellen auszugeben (beachten Sie die Ausgabe von `PI2`):

```
Auf 10 Nachkommastellen genau:
PI1 = 3.1415926536
PI2 = 3.1415920258
2/3 = 0.6666666667
8/3 = 2.6666666667
9/3 = 3.0000000000
7/4 = 1.7500000000
6/3 = 2
```

- Wie oben bereits erwähnt, schaltet `resetiosflags( ios::fixed )` die Ausgabe mit einer festen Anzahl Nachkommastellen aus. Es werden wieder die ersten *n* wesentlichen Stellen ausgegeben (*n* ist abhängig von der letzten Definition mit `setprecision( n )`). Falls nicht noch die Option zur Anzeige von nachführenden Nullen eingeschaltet ist (`setiosflags( ios::showpoint )`), werden diese nicht mehr ausgegeben.

```
Auf 10 wesentliche Stellen genau:
PI1 = 3.141592654
PI2 = 3.141592026
2/3 = 0.6666666667
8/3 = 2.666666667
9/3 = 3
7/4 = 1.75
6/3 = 2
```

- Mit `setiosflags( ios::scientific )` wird auf die wissenschaftliche Schreibweise umgeschaltet:

```
Wissenschaftliche Schreibweise:
PI1 = 3.141593e+00
PI2 = 3.141592e+00
2/3 = 6.666667e-01
8/3 = 2.666667e+00
9/3 = 3.000000e+00
7/4 = 1.750000e+00
6/3 = 2
```

- Mit `resetiosflags( ios::scientific )` wieder auf die normale Schreibweise zurückgeschaltet:

```
Zurueckschalten in Normalschreibweise:
PI1 = 3.141592654
PI2 = 3.141592026
2/3 = 0.6666666667
8/3 = 2.666666667
9/3 = 3
7/4 = 1.75
6/3 = 2
```

- **Wichtig:** Alle angegebenen Befehle müssen in Zusammenhang mit `cout <<` angegeben werden. Sie haben zudem keinen Einfluss auf *Integer*-Variablen. Beachten Sie jeweils die Ausgabe von 6/3, deren Wert einer *Integer*-Variable zugeordnet wurde.

6.2.2 Rechtsbündige und linksbündige Ausgabe

Beispiel:

```
// AUSGABE2.CPP
// Dieses Programm zeigt Beispiele von links- und rechtsbuendiger Ausgabe.

#include <champ.h>
#include "ausgabe2.rh"

void gmain ()
{
   int i;
```

```
   cinit( "Formatierte Ausgabe" );

   cout << "Unformatierte Ausgabe: \n";
   for ( i = 0; i <= 4; i = i + 1 )
      cout << pow10( i ) << "\n";
   cout << "\n";

   cout << "Rechtsbuendig, mit Leerschlaegen als Fuellzeichen: \n";
   for ( i = 0; i <= 4; i = i + 1 )
      cout << setw( 5 ) << pow10( i ) << "\n";
   cout << "\n";

   cout << "Rechtsbuendig, mit Fuellzeichen: \n";
   cout << setfill( '.' );
   for ( i = 0; i <= 4; i = i + 1 )
      cout << setw( 5 ) << pow10( i ) << "\n";
   cout << "\n";

   cout << "Linksbuendig, mit Fuellzeichen: \n";
   cout << setiosflags( ios::left );
   for ( i = 0; i <= 4; i = i + 1 )
      cout << setw( 5 ) << pow10( i ) << "\n";
   cout << "\n";

   cout << "Linksbuendig, mit Leerschlaegen als Fuellzeichen: \n";
   cout << setfill( ' ' );
   for ( i = 0; i <= 4; i = i + 1 )
      cout << setw( 5 ) << pow10( i ) << "\n";
   cout << "\n";

   cout << "Rechtsbuendig, mit einer zu langen Zahl: \n";
   cout << setiosflags( ios::right );
   for ( i = 0; i <= 5; i = i + 1 )
      cout << setw( 5 ) << pow10( i ) << "\n";
   cout << "\n";
   CP::msgBoxD();
   cend();
}
```

Bemerkungen zur formatierten Ausgabe:

- Wird nichts anderes angegeben, so ist die Ausgabe (scheinbar) linksbündig:

```
Unformatierte Ausgabe:
1
10
100
1000
10000
```

- Mit dem Befehl `setw ( n )` wird die sogenannte **Feldweite** der Ausgabe definiert. Das bedeutet, dass für die Ausgabe des folgenden Wertes (**und nur des folgenden Wertes**) *n* Stellen reserviert werden. Die Ausgabe erfolgt ohne anderweitige Angabe rechtsbündig.

```
Rechtsbuendig, mit Leerschlaegen als Fuellzeichen:
    1
   10
  100
 1000
10000
```

- Fehlende Stellen werden (sofern nichts anderes definiert wird) mit Leerzeichen gefüllt. Soll ein anderes Füllzeichen verwendet werden, so muss dieses mit `setfill( c )` angegeben werden (in unserem Beispiel mit `setfill ( '.' );` c ist ein Character). Ausgabe:

```
Rechtsbuendig, mit Fuellzeichen:
....1
...10
..100
.1000
10000
```

- Nach `setiosflags( ios::left );` erfolgt die Ausgabe linksbündig:

```
Linksbuendig, mit Fuellzeichen:
1....
10...
100..
1000.
10000
```

- Mit `setfill( ' ' )` wird wieder der Leerschlag als Füllzeichen definiert:

```
Linksbuendig, mit Leerschlaegen als Fuellzeichen:
1
10
100
1000
10000
```

- Nach `setiosflags( ios::right )` erfolgt die Ausgabe wieder rechtsbündig. Beachten Sie, dass die Zahl 100'000 eine Stelle mehr hat als die angegebene Feldweite, deshalb wird sie rechts über das Feld hinaus geschrieben:

```
Rechtsbuendig, mit einer zu langen Zahl:
    1
   10
  100
 1000
10000
100000
```

- Die Angabe der Feldweite mit `setw( n )` gilt nur für die Ausgabe der nächsten Zahl, d.h. sollen mehrere Zahlen ausgegeben werden, muss die Feldweite jedesmal angegeben werden:
  `cout << "i = " << setw( 7 ) << i << "j = " << setw( 7 ) << j << "\n";`
- Wird die Angabe der Feldweite weggelassen, so wird für die Ausgabe der Zahl nur eine Stelle reserviert. Deshalb erscheint die erste, unformatierte Ausgabe unseres Programms linksbündig.

Weitere Bemerkungen zum Programm:

- Der Schleifenzähler `i` wird zu Beginn des Programms definiert. Es ist nicht möglich, mehrere Male die *for*-Schleife mit `for ( int i = 0; ... )` zu initialisieren, weil die Variable `i` nicht mehrmals definiert werden darf. Es wäre auch möglich, in jeder *for*-Schleife eine andere Variable als Schleifenzähler zu verwenden.
- Mit dem Befehl `pow10( i )` wird die Zehnerpotenz 10^i berechnet.
- Es ist möglich, das Resultat von 10^i direkt mit `cout << pow10( i );` auszugeben, ohne es vorher in einer Variablen (z.B. `res`) zu speichern und dieses dann mit `cout << res;` auszugeben.

6.2.3 Gewünschtes Zahlsystem

Zahlen vom Typ *integer* oder *long* können in drei verschiedenen Zahlsystemen ausgegeben werden: Im gewohnten Dezimalsystem, im Hexadezimalsystem und im Achtersystem (Oktalsystem). Es gibt zwei Möglichkeiten, das gewünschte System anzugeben.

1. Mit den Befehlen `dec` (Dezimalsystem), `hex` (Hexadezimalsystem) und `oct` (Achtersystem).
2. Mit dem Befehl `setbase( n )`, wobei *n* die gewünschte Basis des Zahlsystems ist (10, 16 oder 8). Wird für *n* eine andere Zahl angegeben, wird die Zahl im Dezimalsystem ausgegeben.

Beide Möglichkeiten werden wie die Formatierungsbefehle des letzten Kapitels mit dem Befehl `cout <<` verwendet.

Wird keine Formatierung angegeben, so erfolgt die Ausgabe der Zahl im Dezimalsystem. Die Befehle `dec` bzw. `setbase( 10 )` sind also nur nötig, um ins Dezimalsystem zurück zu schalten.

Beispiel:

```
// AUSGABE3.CPP
// Gibt eine Integerzahl in verschiedenen Zahlsystemen aus.

#include <champ.h>
#include "ausgabe3.rh"
```

```
void gmain ()
{
   int zahl = 211;

   cinit( "Ausgabe von Integerzahlen" );
   cout << "Zahl im Dezimalsystem:     " << zahl << "\n\n";

   cout << "Ausgabe mit oct, hex und dec:\n";
   cout << "Zahl im Achtersystem:      " << oct << zahl << "\n";
   cout << "Zahl im Hexadezimalsystem: " << hex << zahl << "\n";
   cout << "Zahl im Dezimalsystem:     " << dec << zahl << "\n\n";

   cout << "Ausgabe mit setbase:\n";
   cout << "Zahl im Achtersystem:      " << setbase( 8 ) << zahl << "\n";
   cout << "Zahl im Hexadezimalsystem: " << setbase( 16 ) << zahl << "\n";
   cout << "Zahl im Dezimalsystem:     " << setbase( 10 ) << zahl << "\n";

   CP::msgBoxD();
   cend();
}
```

Das Programm liefert folgende Ausgabe auf dem Bildschirm:

```
Zahl im Dezimalsystem:     211

Ausgabe mit oct, hex und dec:
Zahl im Achtersystem:      323
Zahl im Hexadezimalsystem: d3
Zahl im Dezimalsystem:     211

Ausgabe mit setbase:
Zahl im Achtersystem:      323
Zahl im Hexadezimalsystem: d3
Zahl im Dezimalsystem:     211
```

6.3 Variablen vom Typ char

Will man über die Tastatur ein Zeichen (genauer gesagt: den Wert einer Variablen vom Typ *char*) eingeben, so steht ausser der Eingabe mit `cin >>` noch der Befehl `cgetch();` zur Verfügung. Im Unterschied zur Eingabe mit `cin >>` ist nach der Eingabe des Zeichens mit `cgetch();` die Bestätigung mit der Return-Taste überflüssig.

Beispiel:

```
// EINGABE2.CPP
// Eingabe einzelner Zeichen mit cgetch();.

#include <champ.h>
#include "eingabe2.rh"

void gmain ()
{
   char c;

   cinit( "Textfenster" );
   cout << "Geben Sie irgendwelche Zeichen ein; X = Ende des Programms.\n";

   do
   {
      c = cgetch();             // Eingabe eines Zeichens via Tastatur
      cout << c;                // Ausgabe des Zeichens auf dem Bildschirm bis
   } while ( c != 'X' );        // 'X' gedrueckt wird (!= bedeutet "ungleich").
   cout << "\nEnde des Programms.\n";

   CP::msgBoxD();
   cend();
}
```

Bemerkung:

Zur Eingabe eines Zeichens kann auch der Befehl `getch();` verwendet werden. Es ist aber (wie bereits schon einmal erwähnt; Seite 38) besser, im Konsolenfenster `cgetch();` zu verwenden.

Lässt man das Programm laufen, so stellt man fest, dass nach der Betätigung der Return-Taste kein Zeilenvorschub (engl. **Line Feed**) gemacht wird, d.h. die **eingegebene Zeile wird überschrieben**. Im Kapitel 7.1 wird gezeigt, wie man diesen Mangel beheben kann.

Wie bereits im Kapitel 4.3 erwähnt, kann eine Variable vom Typ *char* auch als eine ganze Zahl zwischen -127 und 128 aufgefasst werden. Diese Zahl entspricht dem ASCII-Code des entsprechenden Zeichens (vgl. Anhang Anhang B:). Das folgende Programm zeigt, wie der Zahlenwert einer *char*-Variablen ausgegeben wird.

Beispiel:

```
// EINGABE3.CPP
// Ausgabe von Variablen vom Typ char.

#include <champ.h>
#include "eingabe3.rh"

void gmain ()
{
   char c;

   cinit( "Textfenster" );
   cout << "Geben Sie irgendein Zeichen ein: ";

   c = cgetch();
   cout << "\n\nSie haben das Zeichen " << c << " eingegeben.\n";
   cout << "Das Zeichen " << c << " hat den ASCII-Code " << ( int ) c << ".";

   CP::msgBoxD();
   cend();
}
```

Die Angabe `( int ) c` hat zur Folge, dass nicht das Zeichen selbst, sondern der ASCII-Code des eingegebenen Zeichens ausgegeben wird. Die Ausgabe lautet also:

```
Geben Sie irgendein Zeichen ein:

Sie haben das Zeichen E eingegeben.
Das Zeichen E hat den ASCII-Code 69.
```

Umgekehrt ist es auch möglich, einer *char*-Variablen einen Zahlenwert zuzuordnen.

```
// EINGABE4.CPP
// Ausgabe von Variablen vom Typ char.

#include <champ.h>
#include "eingabe4.rh"

void gmain ()
{
   char c1 = 98;     // Der Zahlenwert 98 (ASCII-Code von 'b') wird c1 zugeordnet.
   char c2 = 'p';    // Das Zeichen 'p' wird c2 zugeordnet.

   cinit( "Textfenster" );

   cout << ( int ) c1 << " ist der ASCII-Code des Zeichens " << c1 << ".\n";
   cout << "Das Zeichen " << c2 << " hat den ASCII-Code " << ( int ) c2 << ".";

   CP::msgBoxD();
   cend();
}
```

Ausgabe des Programms:

```
98 ist der ASCII-Code des Zeichens b.
Das Zeichen p hat den ASCII-Code 112.
```

Bemerkung:

Soll einer *char*-Variablen ein bestimmtes Zeichen zugeordnet werden, so muss dieses Zeichen zwischen einfachen Anführungszeichen (') stehen. Die Zuordnung `char c = p;` ist also nicht richtig.

6.4 CPInput

Die Eingabe von Variablenwerten mit `cin >> a;` hat verschiedene Tücken:

1. Ist a eine *int*-Variable, so ist es trotzdem möglich, Eingaben wie `3.14` oder `b3` zu tätigen. Im ersten Fall wird zwar eine Treppe mit 3 Stufen gezeichnet, im zweiten Fall hat die Treppe aber bedeutend mehr Stufen, da a einen zufälligen Wert annimmt. Es ist möglich, aber aufwendig, solche Fehler im Programm abzufangen.
2. Die Verwendung von `cgetch();` und `cin` im selben Programm ist problematisch, vor allem, wenn **nach** `cgetch();` noch eine Eingabe mit `cin` folgt.
3. Beim Testen eines Programms ist es mühsam, bei jedem Programmstart immer dieselben Variablenwerte eingeben zu müssen.

Champ stellt uns ein potentes Hilfsmittel zur Verfügung, das diese Schwierigkeiten umgeht; eine Dialogbox, die nur die Eingabe von erlaubten Werten oder Zeichen ermöglicht. Diese Dialogbox wird mit `CPInput...` aufgerufen. Das folgende Programm zeigt, wie Werte von Variablen verschiedener Datentypen eingegeben werden können.

Beispiel:

```
// CPINPUT.CPP
// Beispiel zur Eingabe von Werten mit CPInput.

#include <champ.h>
#include "cpinput.rh"

void gmain ()
{
   int i1, i2;
   long l1, l2;
   float f1, f2;
   double d1, d2;
   char c1, c2, c3;

   cinit();                // Initialisierung des Ausgabefensters

   CPInputInt( "Integer", "Geben Sie eine Integer-Zahl ein:", i1 ).showModal();
   CPInputInt( "", "Geben Sie noch einen Integer ein:", i2, 312 ).showModal();

   cout << "Sie haben die Integer-Zahlen " << i1 << " und ";
   cout << i2 << " eingegeben.\n";

   CP::msgBoxD();

   CPInputLong( "Long", "Geben Sie eine Long-Zahl ein:", l1 ).showModal();
   CPInputLong( "", "Geben Sie noch einen Long ein:", l2, 145000L ).showModal();

   cout << "Sie haben die Long-Zahlen " << l1 << " und ";
   cout << l2 << " eingegeben.\n";

   CP::msgBoxD();

   CPInputFloat( "Float", "Geben Sie eine Float-Zahl ein:", f1 ).showModal();
   CPInputFloat( "", "Geben Sie noch einen Float ein:", f2, 3.14 ).showModal();
   cout << "Sie haben die Float-Zahlen " << f1 << " und ";
   cout << f2 << " eingegeben.\n";

   CP::msgBoxD();

   CPInputDouble( "Double", "Geben Sie eine Double-Zahl ein:", d1 ).showModal();
```

```
   CPInputDouble( "", "Geben Sie noch einen Double ein:", d2, 4.83 ).showModal();


   cout << "Sie haben die Double-Zahlen " << d1 << " und ";
   cout << d2 << " eingegeben.\n";

   CP::msgBoxD();

   CPInputChar( "Char", "Geben Sie einen Character ein:", c1 ).showModal();
   CPInputChar( "", "Geben Sie noch einen Character ein:", c2, 104 ).showModal();
   CPInputChar( "", "Geben Sie den dritten Character ein:", c3, 'T' ).showModal();

   cout << "Sie haben die Characters " << c1 << ", " << c2 << " und ";
   cout << c3 << " eingegeben.\n";

   CP::msgBoxD();
   cend();
}
```

Ausgabe des Programms, die Default-Werte wurden mit OK bestätigt:

```
Sie haben die Integer-Zahlen 4 und 312 eingegeben.
Sie haben die Long-Zahlen 23 und 145000 eingegeben.
Sie haben die Float-Zahlen 4.5644 und 3.14 eingegeben.
Sie haben die Double-Zahlen 3.34525 und 4.83 eingegeben.
Sie haben die Characters J, h und T eingegeben.
```

Bemerkungen:

- Will man mit `CPInput` eine Dialogbox öffnen, so lautet der Befehl wie folgt:

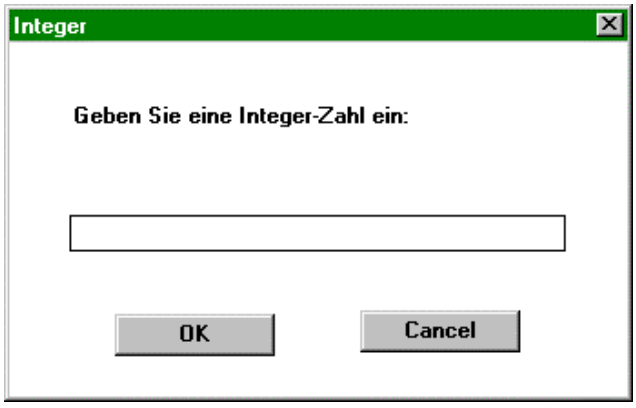
  `CPInput`*Typ*`(` *Titel*`,` *Aufforderung*`,` *Variable*`,` *[Default]* `).showModal();`

  Direkt nach `CPInput` muss angegeben werden, von welchem Datentyp der Wert ist, der eingelesen werden soll: `CPInputChar` liest eine Variable vom Typ *char* ein, `CPInputDouble` eine Variable vom Typ *double*, usw.. Es ist sogar möglich, Werte vom Typ *unsigned int* und *unsigned long* (vgl. Seite 31) einlesen zu lassen (`CPInputUInt` bzw. `CPInputULong`).

- Die Parameter *Titel* und *Aufforderung* sind zwei Strings (vgl. Kapitel 7), die einerseits den Titel des Fensters und andererseits die Aufforderung zur Eingabe enthalten. Wird kein Titel angegeben (mit `""`, er kann aber nicht ganz weg gelassen werden), so hat das Fenster den Titel "*Title*". *Variable* ist der Name der Variablen, der der eingegebene Wert zugeordnet werden soll. Da `CPInput` kein Befehl, sondern eine Klasse ist (vgl. Kapitel 11.1), muss das Eingabefenster auf eine für Klassen typische Art geöffnet werden: Der Befehl `showModal()` (genauer gesagt die **Methode** `showModal()`; vgl. Kapitel 11) muss, durch einen Punkt abgetrennt, direkt hinter der Klammer angefügt werden. Nach dem Befehl

  ```
  CPInputInt( "Integer", "Geben Sie eine Integer-Zahl ein:", i1 ).showModal();
  ```

  erscheint die folgende Dialogbox:

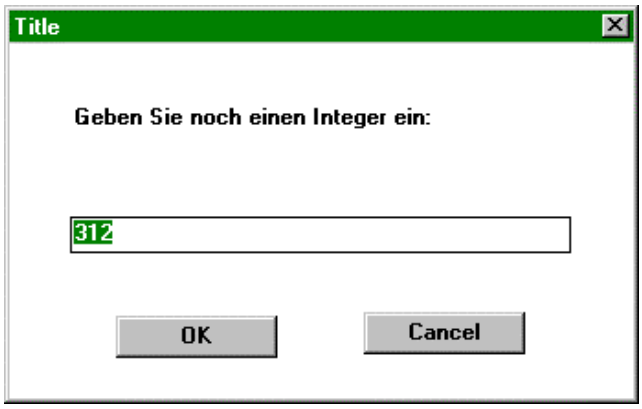


- Der Anwender kann nun eine Integer-Zahl eingeben. Das Programm prüft während der Eingabe, ob nicht fälschlicherweise ein Zeichen (wird gar nicht geschrieben) oder eine zu grosse Zahl eingegeben wird (auch hier ist die Eingabe unmöglich, versuchen Sie den Wert 50'000 einzugeben). Die Eingabe wird mit der Return-Taste oder einem Mausklick auf OK bestätigt. Wird auf Cancel geklickt, so wird der bisherige Wert der Variablen beibehalten.

- Vor allem beim Testen eines Programms ist es manchmal von Vorteil, dass nicht alle Werte jedesmal eingegeben werden müssen. Bei `CPInput` ist es möglich, bereits einen Wert vorzuschlagen, der dann nur noch mit Return oder OK bestätigt werden muss:

```
CPInputInt( "", "Geben Sie noch einen Integer ein:", i2, 312 ).showModal();
```

  liefert folgendes Fenster:



- Natürlich kann die Eingabe durch den Anwender verändert werden. Wird aber bloss die Return-Taste gedrückt oder mit OK bestätigt, dann erhält in diesem Fall `i2` den vorgeschlagenen Wert 312. Ist der vorgeschlagene Wert eine Zahl vom Typ *long*, so muss dies durch anfügen eines grossen `L` an den Wert angegeben werden (`145000L`).
- Bei der Eingabe eines Characters kann das vorgeschlagene Zeichen entweder im ASCII-Code (siehe Eingabe von `c2`) oder direkt (zwischen Hochkommas; siehe Eingabe von `c3`) angegeben werden.

7 Selektionen

Ein wesentliches Merkmal höherer Programmiersprachen sind nicht nur die Schleifen (Iterationen) sondern auch das Ausführen von Programmteilen unter bestimmten Bedingungen (**Selektionen)**. Diese Bedingungen lassen sich umgangssprachlich mit "Wenn-dann" umschreiben. Das wichtigste Hilfsmittel für Selektionen ist die **if-Anweisung**.

7.1 Die if-Anweisung

Im Programm `eingabe2.cpp` stellten wir fest, dass nach der Betätigung der Return-Taste die neuen Zeichen zwar wieder am Zeilenanfang, aber auf der gleichen Zeile geschrieben werden. Mit andern Worten: Nach einem Wagenrücklauf (engl. "Carriage Return") wird kein Zeilenvorschub (Line Feed) gemacht. Deshalb soll das Programm wie folgt geändert werden: **Wenn** die Return-Taste gedrückt wird, **dann** soll zusätzlich zum Wagenrücklauf noch ein Zeilenvorschub gemacht werden.

Beispiel:

```
// IF1.CPP
// Eingabe einzelner Zeichen mit cgetch();. Nach der Eingabe der Return-Taste
// wird noch ein Zeilenvorschub gemacht.

#include <champ.h>
#include "if1.rh"

const char CR = 13;             // ASCII-Code fuer "Carriage Return" (CR)
const char LF = 10;             // ASCII-Code fuer "Line Feed" (LF)

void gmain ()
{
   char c;

   cinit( "Textfenster" );
   cout << "Geben Sie irgendwelche Zeichen ein; X = Ende des Programms.\n";
   do
   {
      c = cgetch();
      cout << c;
      if ( c == CR )            // Falls die Return-Taste gedrueckt wird,
         cout << LF;            // dann gehe eine Zeile weiter!
   } while ( c != 'X' );
   cout << "\nEnde des Programms.\n";

   CP::msgBoxD();
   cend();
}
```

Bemerkungen:

- `CR` und `LF` sind zwei **Konstanten** vom Typ `char`, d.h. sie können weder im Hauptprogramm noch in einer allfälligen Funktion geändert werden. Ihre Werte sind die ASCII-Codes (vgl. Anhang Anhang B:) für den Wagenrücklauf (CR = 13) und für den Zeilenvorschub (LF = 10).
- Die vorliegende if-Anweisung ist ein Beispiel einer **einseitigen Auswahl**: Falls die Bedingung erfüllt ist, dann wird die Anweisung ausgeführt, andernfalls wird sie übersprungen. Form einer einseitigen Auswahl:

  `if (` *Bedingung* `)` *Anweisung*`;`

  Wie gewohnt kann statt einer Anweisung auch ein ganzer Anweisungsblock (natürlich zwischen geschweiften Klammern) stehen. **Beachten Sie**: Im Unterschied zu anderen Programmiersprachen (BASIC, Pascal) kennt C++ das Wort *then* nicht!

- Will man zwei Ausdrücke auf Gleichheit testen, so sind statt einem sogar zwei Gleichheitszeichen nötig:
  - `c = 13` bedeutet: Der Variablen `c` wird der Wert 13 zugeordnet.
  - `c == 13` bedeutet: Es wird getestet, ob die Variable `c` den Wert 13 hat.

- **Vorsicht!** Schreibt man statt `if ( c == CR )` nur `if ( c = CR )`, so gibt der Compiler keine Fehlermeldung an, sondern nur eine Warnung. Das bedeutet: Das Programm ist lauffähig, wird aber nicht so laufen, wie wir es beabsichtigt haben.

Eine **zweiseitige Auswahl** sieht wie folgt aus:

`if (` *Bedingung* `)` *Anweisung 1* `else` *Anweisung 2*`;`

Falls die Bedingung erfüllt ist, wird die Anweisung 1 (genauer: der Anweisungsblock nach *if*) ausgeführt, andernfalls wird die Anweisung 2 (bzw. der Anweisungsblock nach *else*) abgearbeitet.

Beispiel: Es soll eine Treppe mit unterschiedlichen Stufenhöhen gezeichnet werden: Die dritte Stufe hat Stufenhöhe 40, die restlichen haben Stufenhöhe 20.

```
// IF2.CPP
// Zeichnet eine fuenfstufige Treppe, die dritte Stufe ist hoeher als die andern.

#define GLOBAL_TURTLE

#include <champ.h>
#include "if2.rh"

void stufe ( int hoehe );

void gmain ()
{
   int i = 1;

   ginit( "Champ" );

   while ( i <= 5 )
   {
      if ( i == 3 )            // Falls i = 3 ist, so wird eine Stufe der
         stufe( 40 );          // Hoehe 40 gezeichnet;
      else                     // andernfalls eine Stufe der Hoehe 20.
         stufe( 20 );
      i++;                     // Die Variable i wird um 1 erhoeht;
   }                           // andere Schreibweise fuer "i = i + 1;".

   CP::msgBoxD();
   gend();
}

void stufe ( int hoehe )
{
   forward( hoehe );
   right( 90 );
   forward( 20 );
   left( 90 );
}
```

Bemerkung:

Der Befehl `i++;` bewirkt, dass der Wert der Variablen `i` um 1 erhöht wird. `i++` ist eine Abkürzung für den Befehl `i = i + 1;`.

Natürlich ist es möglich, mehrere *if-else*-Anweisungen zu verschachteln. Oft stellt sich dann aber die Frage, zu welchem *if*-Teil ein *else*-Teil gehört. Scheinbare Mehrdeutigkeiten werden gelöst, indem ein *else* dem letzten auf derselben Schachtelungstiefe aufgetretenen *if* ohne *else* zugeordnet wird. Beachten Sie den Unterschied der beiden Programme auf der nächsten Seite:

```
// IF3.CPP

#define GLOBAL_TURTLE

#include <champ.h>
#include "if3.rh"

void stufe ( int hoehe );

void gmain ()
{
   int i = 1;

   ginit( "Champ" );

   while ( i <= 5 )
   {
      if ( i <= 2 )
         stufe( 10 );
      if ( i == 3 )
         stufe( 40 );
      else
         stufe( 20 );
      i++;
   }

   CP::msgBoxD();
   gend();
}

void stufe ( int hoehe )
{
   forward( hoehe );
   right( 90 );
   forward( 20 );
   left( 90 );
}
```

```
// IF4.CPP

#define GLOBAL_TURTLE

#include <champ.h>
#include "if4.rh"

void stufe ( int hoehe );

void gmain ()
{
   int i = 1;

   ginit( "Champ" );

   while ( i <= 5 )
   {
      if ( i <= 2 )
      {
         stufe( 10 );
         if ( i == 3 )
            stufe( 40 );
      }
      else
         stufe( 20 );
      i++;
   }

   CP::msgBoxD();
   gend();
}

void stufe ( int hoehe )
{
   forward( hoehe );
   right( 90 );
   forward( 20 );
   left( 90 );
}
```

Das Programm `if3.cpp` zeichnet eine Treppe mit Stufenhöhen 10 - 20 - 10 - 20 - 40 - 20 - 20. Hier gehört der *else*-Teil zu `if ( i == 3 )`. Im Programm `if4.cpp` wird eine Treppe mit den Stufenhöhen 10 - 10 - 20 - 20 - 20 gezeichnet, das *else* wird dem `if ( i <= 2 )` zugeordnet. Beachten Sie, dass im zweiten Programm nie eine Treppe mit Stufenhöhe 40 gezeichnet wird, da der Test `if ( i == 3 )` nur dann erfolgt, wenn `i` entweder gleich 1 oder gleich 2 ist.

Bemerkung: Durch optische Darstellung (also Einrücken) kann man die Zuordnung einer *else*-Anweisung zu einem *if* nicht beeinflussen. Die Zeilen

```
   while ( i <= 5 )
   {
      if ( i <= 2 )
         stufe( 10 );
         if ( i == 3 )
            stufe( 40 );
      else
         stufe( 20 );
      i++;
   }
```

führen zur selben Treppe, die auch im Programm `if3.cpp` gezeichnet wird.

Aus Gründen der Übersichtlichkeit ist es nicht zu empfehlen, allzu komplizierte *if-else*-Verschachtelungen zu programmieren. In einem solchen Falle ist es besser, die **switch-Anweisung** zu verwenden.

7.2 Die switch-Anweisung

Als Alternative zu verschachtelten *if*-Anweisungen bietet sich die sogenannte **Mehrfachauswahl** mit der **switch-Anweisung** an.

Beispiel: Zu zeichnen ist eine Treppe mit zehn Stufen. Die Stufen 2 und 7 sollen eine Höhe von 10, Stufe 5 soll eine Höhe von 30, die übrigen sollen eine Höhe von 20 haben:

```
// SWITCH.CPP
// Zeichnet eine Treppe mit unterschiedlichen Stufenhoehen.

#define GLOBAL_TURTLE

#include <champ.h>
#include "switch1.rh"

void stufe ( int hoehe );

void gmain ()
{
   int i = 1;

   ginit( "Champ" );

   while ( i <= 10 )
   {
      switch ( i )
      {
         case 2 :
            stufe( 10 );            // Ist i = 2, so ist die Stufenhoehe = 10.
            break;
         case 5 :
            stufe( 30 );            // Ist i = 5, so ist die Stufenhoehe = 30.
            break;
         case 7 :
            stufe( 10 );            // Ist i = 7, so ist die Stufenhoehe = 10.
            break;
         default :                  // In den andern Faellen ist
            stufe( 20 );            // die Stufenhoehe = 20.
      }
      i++;
   }

   CP::msgBoxD();
   gend();
}

void stufe ( int hoehe )
{
   forward( hoehe );
   right( 90 );
   forward( 20 );
   left( 90 );
}
```

Bemerkungen:

- Die *switch*-Anweisung wird wie folgt abgearbeitet: Zuerst wird der Ausdruck in der Klammer (der **switch-Ausdruck**) nach dem Wort `switch` ausgewertet, d.h. in unserem Beispiel wird der Wert der Variablen `i` ermittelt. Dieser Wert wird mit den Konstanten hinter den **case**-Marken verglichen. Stimmt eine dieser Marken mit dem Switch-Ausdruck überein, springt das Programm zu dieser Marke und fährt dort fort, andernfalls springt das Programm zur Marke `default` und führt die Anweisungen aus, die dieser Marke folgen.
- Nachdem das Programm zu einer Marke gesprungen ist, werden sämtliche Befehle abgearbeitet, bis entweder die *switch*-Anweisung beendet ist oder eine **break**-Anweisung erreicht wird. Die *break*-Anweisung kann nur innerhalb von Schleifen oder einer *switch*-Anweisung verwendet werden. Sie dient zur Beendigung der Schleife bzw. der Anweisung. In unserem Fall springt das Programm nach einem `break;` zur Zeile `i++;`. Das Verwenden der *break*-Anweisungen scheint auf den ersten Blick recht mühsam zu sein, hat aber auch seine Vorteile, wie die folgende Änderung der *while*-Schleife des Beispiels zeigt:

```
while ( i <= 10 )
{
   switch ( i )
   {
      case 2 :
      case 7 :
```

```
            stufe( 10 );
            break;
         case 5 :
            stufe( 30 );
            break;
         default:
            stufe( 20 );
      }
      i++;
   }
```

- Ist `i == 2`, so springt das Programm zur Marke `case 2`. Weil dort keine Anweisung, aber auch keine *break*-Anweisung ist, wird die Anweisung `stufe( 10 );`, die sich bereits bei der Marke `case 7` befindet, durchgeführt. Es ist also nicht nötig, identische Abfolgen bei *switch*-Anweisungen mehrmals zu schreiben.

7.3 Operatoren

In der Informatik bezeichnet man mit dem Begriff **Operator** meist die Funktionszeichen für arithmetische und logische Operationen sowie relationale Operatoren. Es sind aber noch weitere Arten von Operatoren denkbar (vgl. Hilfe für Borland C++), z.B. der **Zuweisugsoperator** (=).

7.3.1 Arithmetische Operatoren

Die arithmetischen Operatoren sind die Funktionszeichen für die Addition (+), die Subtraktion (-), die Multiplikation (*) und die Division (/), sowie der **Modulo-Operator** (%).

Die Hierarchie der einzelnen Operatoren ist dabei dieselbe wie in der Mathematik (also Punktrechnung vor Strichrechnung).

Vorsicht: Stehen vor und nach dem Divisionszeichen zwei Zahlen vom Typ *int*, so wird eine Division *ohne Rest* durchgeführt (sogenannte **ganzzahlige Division**).

Wird statt dessen das exakte Resultat erwünscht, so muss eine Zahl vom Typ *float* bzw. *double* sein oder man schreibt vor der Division in Klammern den gewünschten Datentyp des Quotienten hin (z.B. `( float )` oder `( double )`). Die Typenkonvertierung (engl. Typecasting), d.h. die letzte Möglichkeit, wählt man dann, wenn zwei Variablen vom Typ *int* (bzw. *long*) dividiert werden sollen.

Mit dem Modulo-Operator berechnet man den Rest einer ganzzahligen Division.

Beispiel:

```
// DIVISION.CPP
// Dieses Programm zeigt Beispiele verschiedener Divisionsarten.

#include <champ.h>
#include "division.rh"

void gmain ()
{
   double x1, x2, x3;
   int rest;
   cinit( "Champ" );

   x1 = 5 / 3;
   x2 = 5.0 / 3;
   x3 = ( double ) 5 / 3;
   rest = 5 % 3;

   cout << " 5 / 3 \t\t= " << x1 << "\n";
   cout << " 5.0 / 3 \t= " << x2 << "\n";
   cout << " ( double ) 5 / 3 = " << x3 << "\n";
   cout << " 5 % 3 \t\t= " << rest << "\n";

   CP::msgBoxD();
   cend();
}
```

Ausgabe des Programms:

```
5 / 3           = 1
5.0 / 3         = 1.66667
```

```
( double ) 5 / 3 = 1.66667
5 % 3            = 2
```

7.3.2 Relationale Operatoren

Unter den relationalen Operatoren versteht man diejenigen Operatoren, die zwei arithmetische Ausdrücke vergleichen:

- `==` : Testet zwei Ausdrücke auf Gleichheit:
```
if ( x == 3 )
   stufe( 40 );
```
- `!=` : Testet zwei Ausdrücke auf Ungleichheit:
```
if ( x != 3 )
   stufe( 40 );
```
- `>` : "grösser als":
```
if ( x > 3 )
   stufe( 40 );
```
- `>=` : "grösser gleich":
```
if ( x >= 3 )
   stufe( 40 );
```
- `<=` : "kleiner gleich":
```
if ( x <= 3 )
   stufe( 40 );
```
- `<` : "kleiner als":
```
if ( x < 3 )
   stufe( 40 );
```

7.3.3 Logische Operatoren

In den Köpfen der Schleifen und in den *if*-Anweisungen werden logische Ausdrücke ausgewertet. Diese Ausdrücke können unter Umständen kompliziert sein. Die wichtigsten logischen Operatoren sind `&&` (logisches UND, $\wedge$) und `||` (logisches ODER, $\vee$). Das Zeichen `|` wird durch die Tastenkombination <Alt Gr> + <7> erzeugt. Ein logischer Ausdruck ist bekanntlich entweder "falsch" oder "wahr", diese Wahrheitswerte entsprechen in C++ den Werten 0 bzw. 1 und können deshalb sogar Variablen vom Typ *int* zugeordnet werden.

Beispiele:

- `x = ( 2 < 3 );` `// x erhaelt den Wert 1`
- `x = ( 2 > 3 );` `// x erhaelt den Wert 0`
- `x = ( ( 2 < 3 ) || ( 2 > 3 ) );` `// x erhaelt den Wert 1`
- `x = ( ( 2 < 3 ) && ( 2 > 3 ) );` `// x erhaelt den Wert 0`

Der dritte logische Operator ist das Zeichen `!` für die Negation:

- `x = !( 2 > 3 );` `// x erhaelt den Wert 1`

Die hier vorgestellten (Aussage-)logischen Operatoren sollten nicht mit den Bitweisen Operatoren verwechselt werden: `|` (bitweises ODER) und `&` (bitweises UND).

8 String-Objekte, Strings und Arrays

Ein **String** (**Zeichenkette**) ist eine Kette von Characters, meistens Buchstaben des Alphabets. Wir haben bereits mit Strings gearbeitet: Im Ausgabebefehl `cout << "Hallo Welt"` wird ein String mit dem Inhalt `Hallo Welt`, der zwischen Anführungszeichen stehen muss, auf den Bildschirm ausgegeben. Es ist aber auch notwendig, einen String (z.B. einen Vornamen) während des Programmablaufs eingeben oder verarbeiten zu können. Wir wollen dabei zwei Arten von Zeichenketten unterscheiden: den "normalen" String (siehe Kapitel 8.5) und das **String-Objekt**.

8.1 Ein- und Ausgabe von String-Objekten

Zur einfachen Behandlung von Strings stellt uns C++ eine **Klasse string** zur Verfügung:

Vereinfacht gesagt definiert eine Klasse einen Datentyp, der Variablen (**Attribute**) und Funktionen (**Methoden**) zusammenfasst.

Beispiel: Im folgenden Programm soll ein String-Objekt (`name`) eingegeben werden. Ausgegeben wird zuerst der ganze Name, dann der zweite Buchstabe des Namens. Zuletzt wird der Name noch rückwärts geschrieben.

```
// STRING1.CPP
// Eingabe und Verarbeitung eines String-Objekts.

#include <champ.h>
#include <cstring.h>     // Noetig, damit die verschiedenen String-Befehle
#include "string1.rh"    // erkannt werden.

void gmain ()
{
   string name;          // Instanziierung (Definition) eines String-Objekts

   cinit( "String-Objekte" );
   cout << "Geben Sie einen Namen ein: ";
   cin >> name;
   cout << "\nSie haben den Namen " << name << " eingegeben.\n";
   cout << "Der zweite Buchstabe des Namens ist ein " << name[ 1 ] << ".\n";

   cout << "Der Name rueckwaerts geschrieben lautet: ";
   for ( int i = name.length() - 1; i >= 0; i-- )
      cout << name[ i ];
   cout << ".\n\n";

   cout << "Der erste Buchstabe soll geaendert werden; ";
   cout << "geben Sie einen neuen ein: ";
   cin >> name[ 0 ];
   cout << "\nDer Name lautet nun " << name << ". \n\n";

   CP::msgBoxD();
   cend();
}
```

Ausgabe auf dem Bildschirm:

```
Geben Sie einen Namen ein: Andreas

Sie haben den Namen Andreas eingegeben.
Der zweite Buchstabe des Namens ist ein n.
Der Name rueckwaerts geschrieben lautet: saerdnA.

Der erste Buchstabe soll geaendert werden; geben Sie einen neuen ein: E

Der Name lautet nun Endreas.
```

Bemerkungen:

- `Damit der Compiler die Befehle, mit denen die String-Objekte bearbeitet werden können, und die String-Objekte selbst erkennen kann, muss die Datei cstring.h mit #include <cstring.h>` eingebunden werden (vgl. Anhang Anhang D:).

- `name` ist ein **Objekt** (eine **Instanz**) der Klasse *string*. Die Instanziierung eines String-Objekts (`string name;`) geht wie die Definition einer Variablen. Es ist auch möglich, bei seiner Instanziierung dem Objekt bereits einen String zu übergeben (beachten Sie dazu das Programm `string3.cpp`).
- Der String eines String-Objekts kann wie eine Variable mit dem Befehl `cin >>` eingegeben und mit dem Befehl `cout <<` ausgegeben werden.
- Es ist möglich, einzelne Zeichen eines String-Objekts zu betrachten und diese wie Variablen vom Typ *char* zu behandeln (z.B. `name[ 1 ] = 'e';`). Die Zeichen werden dabei wie folgt numeriert: `name[ 0 ]`, `name[ 1 ]`, `name[ 2 ]` usw. Beachten Sie, dass die Numerierung bei 0 beginnt und dass die einzelnen Indizes zwischen eckigen Klammern stehen.
- Neu und ungewohnt für uns ist die Bestimmung der Länge eines String-Objekts. Naheliegend wäre eigentlich der Befehl `i = length( name );` (die Stringlänge soll also der Variablen `i` zugeordnet werden). Da aber `name` eben ein *Objekt* der *Klasse* **string** ist, erfolgt der Aufruf der Methode `length()` auf eine für die Klassen typische Art und Weise: *NameDesObjekts*`.`*NameDerMethode*. Hier: `i = name.length();`. Beachten Sie, dass der Objektname und der Name der Methode durch einen Punkt getrennt werden.
- Hat ein String-Objekt die Länge *n*, so sind die einzelnen Zeichen von 0 bis *n* – 1 numeriert, deshalb wird im Initialisierungsausdruck der *for*-Schleife der Variablen `i`, die rückwärts von *n* – 1 bis 0 laufen soll, mit dem Befehl `i = name.length() - 1;` der Wert *n* – 1 zugeordnet.
- Die maximal mögliche Länge eines Strings in einem String-Objekts ist praktisch unbeschränkt, sie wird nur durch die Computerressourcen beschränkt.

Das Programm hat ein grosses Manko: Gibt man statt dem Namen "Andreas" etwa den Namen "Anna Maria" ein, so sieht die Ausgabe wie folgt aus:

```
Geben Sie einen Namen ein: Anna Maria

Sie haben den Namen Anna eingegeben.
Der zweite Buchstabe des Namens ist ein n.
Der Name rueckwaerts geschrieben lautet: annA.

Der erste Buchstabe soll geaendert werden; geben Sie einen neuen ein:
Der Name lautet nun Mnna.
```

Man stellt fest, dass nur der Name "Anna" im String-Objekt gespeichert wird. Der Grund liegt in der Arbeitsweise des Befehls `cin >>`: Wird nach der Eingabe einer Zeile die Return-Taste gedrückt, so wird die ganze Zeile im **Tastatur-Buffer** zwischengespeichert. Kommt das Programm zur Zeile `cin >> name;`, so prüft es zuerst, ob sich im Buffer noch Zeichen befinden. Ist dies der Fall, so werden diese (bis zum ersten Leerschlag) dem String-Objekt zugewiesen. Andernfalls kann nun der Anwender den Namen eingeben und diesen nach Drücken der Return-Taste im Tastatur-Buffer ablegen, bevor die Eingabe bis zum ersten Leerschlag dem String-Objekt zugewiesen wird. In beiden Fällen verbleiben die restlichen Zeichen ("Maria") im Buffer. Aus diesem Grund wird bei der Eingabe des neuen ersten Buchstabens automatisch der Buchstabe `M` eingelesen.

Das folgende Beispiel zeigt, wie man mit `getline` diesen Fehler beheben kann:

Beispiel:

```
// STRING2.CPP
// Eingabe eines String-Objekts mit getline.
```

```
#include <champ.h>
#include <cstring.h>
#include "string2.rh"

void gmain ()
{
   string name;          // Instanziierung (Definition) eines String-Objekts

   cinit( "String-Objekte" );
   cout << "Geben Sie einen Namen ein: ";
   getline( cin, name );
   cout << "\nSie haben den Namen " << name << " eingegeben.\n";
   cout << "Der zweite Buchstabe des Namens ist ein " << name[ 1 ] << ".\n";

   cout << "Der Name rueckwaerts geschrieben lautet: ";
   for ( int i = name.length() - 1; i >= 0; i-- )
      cout << name[ i ];
   cout << ".\n\n";

   cout << "Der erste Buchstabe soll geaendert werden; ";
   cout << "geben Sie einen neuen ein: ";
   cin >> name[ 0 ];
   cout << "\nDer Name lautet nun " << name << ". \n\n";

   CP::msgBoxD();
   cend();
}
```

Ausgabe:

```
Geben Sie einen Namen ein: Anna Maria

Sie haben den Namen Anna Maria eingegeben.
Der zweite Buchstabe des Namens ist ein n.
Der Name rueckwaerts geschrieben lautet: airaM annA.

Der erste Buchstabe soll geaendert werden; geben Sie einen neuen ein: E

Der Name lautet nun Enna Maria.
```

Bemerkungen:

- Im Vergleich zum Programm `string1.cpp` wird nur eine Zeile geändert: Statt `cin >> name;` wird für die Eingabe des Strings der Befehl `getline( cin, name );` aus der Klasse *string* verwendet. Die Parameter von `getline` sind ein Objekt der Klasse *istream* (in unserem Fall das Objekt `cin`, das mit der Tastatur verknüpft ist) und ein Objekt der Klasse *string*. (Näheres zum Objekt `cin` und zur Klasse *istream* erfahren Sie im Kapitel 10.3.)
- Der Befehl `i--;` hat zur Folge, dass der Wert der Variablen `i` um 1 verkleinert wird. `i--;` ist also eine Abkürzung für `i = i - 1;`.

8.2 Verarbeitung von String-Objekten

Beispiel:

```
// STRING3.CPP
// Beispiele zur Verarbeitung von String-Objekten.

#include <champ.h>
#include <cstring.h>
#include "string3.rh"

void gmain ()
{
   string vorname1, vorname2, name1, name2;
```

```
   string nachname1 = "Muster";         // Instanziierung und Initialisierung
   string nachname2( "Meier" );         // der Nachnamen auf zwei Arten.
   string teilstring1, teilstring2;

   cinit( "String-Objekte" );

   cout << "Geben Sie einen Vornamen ein: ";
   getline( cin, vorname1 );
   cout << "Geben Sie einen zweiten Vornamen ein: ";
   getline( cin, vorname2 );

   if ( vorname1 > vorname2 )           // Vergleich der beiden Strings
      name1 = vorname1;
   else
      name1 = vorname2;                 // Zuweisung eines String-Objekts
                                        // zu einem anderen.
   cout << "\nDer alphabetisch groessere Vorname ist " << name1 << ".\n\n";

   name1 = vorname1 + " " + nachname1;  // Die Strings werden 'zusammengesetzt'
   name2 = vorname2 + " " + nachname2;
   cout << "Ein erster vollstaendiger Name ist " << name1 << ".\n";
   cout << "Ein zweiter vollstaendiger Name ist " << name2 << ".\n\n";

   teilstring1 = name1.substr( 2 );     // Erzeugen von Teilstrings mit
   teilstring2 = name2.substr( 2, 4 );  // der Methode substr

   cout << "Ein Teil des ersten Namens ist " << teilstring1 << ".\n";
   cout << "Ein Teil des zweiten Namens ist " << teilstring2 << ".\n\n";

   string teilstring3( name1, 1 );      // Erzeugen von Teilstrings bei
   string teilstring4( name2, 1, 5 );   // der Instanziierung.

   cout << "Ein anderer Teil des ersten Namens ist " << teilstring3 << ".\n";
   cout << "Ein anderer Teil des zweiten Namens ist " << teilstring4 << ".\n\n";

   CP::msgBoxD();
   cend();
}
```

Die Ausgabe des Programms ist:

```
Geben Sie einen Vornamen ein: Felix
Geben Sie einen zweiten Vornamen ein: Hans

Der alphabetisch groessere Vorname ist Hans.

Ein erster vollstaendiger Name ist Felix Muster.
Ein zweiter vollstaendiger Name ist Hans Meier.

Ein Teil des ersten Namens ist lix Muster.
Ein Teil des zweiten Namens ist ns M.

Ein anderer Teil des ersten Namens ist elix Muster.
Ein anderer Teil des zweiten Namens ist ans M.
```

Bemerkungen:

- Strings können den String-Objekten bereits bei ihrer Instanziierung auf mehrere Arten zugeordnet werden: Mit (z.B. `string nachname1 = "Muster";` bzw. `string nachname2( "Meier" );` werden die String-Objekte bereits initialisiert.

- String-Objekte können wie Variablen miteinander verglichen werden: `string1` ist kleiner als `string2`, wenn er im Alphabet vor `string2` erscheint (z.B. `Felix < Hans`). Allerdings richtet sich dieses Alphabet nach dem ASCII-Code. Das bedeutet, dass alle String-Objekte, die mit Kleinbuchstaben beginnen, kleiner sind als die String-Objekte, die mit einem Grossbuchstaben beginnen (also ist `hans < Felix`).
- String-Objekte können zueinander zugewiesen werden; z.B. `name1 = vorname1;` oder `name1 = "Hans";`.
- Mit der Methode `substr` wird ein Teilstring (engl. **substring**) des im String-Objekt enthaltenen Strings erzeugt: `name1.substr( 2 );` bestimmt den Teilstring von `name1` ab dem dritten Zeichen an, ein allfälliger zweiter Parameter gibt die gewünschte Länge des Teilstrings an: `name2.substr( 2, 4 );`. Beachten Sie, dass die Numerierung der Zeichen eines Strings bei 0 beginnt!
- Teilstrings können schon bei der Instanziierung dem String-Objekt zugewiesen werden, wie die Instanziierung von `teilstring3` und `teilstring4` zeigt: In unserem Programm erhält `teilstring3` einen Teil des Strings aus dem String-Objekt `name1` zugeordnet.
- String-Objekte können addiert (**konkateniert**) werden: `name1 = vorname1 + " " + nachname1;`. Achten Sie darauf, dass ohne die "Addition" des Leerzeichens die zwei String-Objekte direkt aneinander gehängt werden (`FelixMuster`).

8.3 Arrays

Ein **Array** ist eine Aneinanderreihung von Variablen vom gleichen Typ, wobei auf die einzelnen Variablen mit Hilfe eines Indexes zugegriffen wird.

Beispiel: Im folgenden Programm wird ein Würfel simuliert. Er soll hundert mal geworfen werden, anschliessend soll im Textfenster ausgegeben werden, welche Augenzahl wie oft erschienen ist (absolute Häufigkeit, relative Häufigkeit).

```
// WUERFEL.CPP
// Simuliert einen Wuerfel

#include <champ.h>
#include <stdlib.h>                        // Noetig, damit die Befehle random()
#include "wuerfel.rh"                      // und randomize() verstanden werden.

void relativeHaeufigkeiten ( const int anzahl[], double rel[] );

const int ANZAHLWUERFE = 100;             // Anzahl der Wuerfe



void gmain ()
{
   int anzahl[ 6 ] = { 0, 0, 0, 0, 0, 0 }; // Array fuer die Augenzahlen
   double rel[ 6 ];                        // Array fuer die relative Haeufigkeit
   int wurf;                               // Augenzahl des aktuellen Wurfes
   int i;                                  // Zaehlvariable

   cinit( "Wuerfelsimulator" );
   randomize();

   for ( i = 1; i <= ANZAHLWUERFE; i++)    // Simulation der Wuerfe
   {
      wurf = random( 6 );
      anzahl[ wurf ]++;                    // entspricht dem Befehl
   }                                       // anzahl[ wurf ] = anzahl[ wurf ] + 1;

   relativeHaeufigkeiten( anzahl, rel );   // Berechnung der relativen
                                           // Haeufigkeiten

   cout << "Augenzahl:       abs. Haeufigkeit:    rel. Haeufigkeit: \n";
   for ( i = 0; i <= 5; i++)
      cout << ( i + 1 ) << "\t\t" << anzahl[ i ] << "\t\t   " << rel[ i ] << "\n";

   CP::msgBoxD();
   cend();
}
```

```
void relativeHaeufigkeiten ( const int anzahl[], double rel[] )
{
   for ( int i = 0; i <= 5; i++ )
      rel[ i ] = ( double ) anzahl[ i ] / ANZAHLWUERFE;
}
```

Ausgabe des Programms:

```
Augenzahl:  abs. Haeufigkeit:  rel. Haeufigkeit:
1               14                 0.14
2               17                 0.17
3               19                 0.19
4               21                 0.21
5               18                 0.18
6               11                 0.11
```

Bemerkungen:

- Mit `const int ANZAHLWUERFE = 100;` wird eine globale (d.h. im ganzen Programm gültige) Konstante definiert und initialisiert, deren Wert die Anzahl der Würfe beinhalten soll. Die Definition dieser Konstanten vereinfacht eine allfällige Änderung der Anzahl der Würfe. Statt überall im Programm den alten Wert 100 zu suchen und durch einen neuen Wert (z.B. 1000) zu ersetzen, erreichen wir dasselbe mit der einfachen Änderung `const int ANZAHLWUERFE = 1000;`.

  Beachten Sie, dass Konstanten normalerweise mit Grossbuchstaben geschrieben werden, damit sie auch optisch von den Variablen unterscheidbar sind. Konstanten können im Programm nicht verändert werden!

- Mit `int anzahl[ 6 ] = { 0, 0, 0, 0, 0, 0 };` wird ein Array definiert, das aus sechs Feldern (numeriert von 0 bis 5) besteht, die jeweils eine Variable vom Typ *int* enthalten. Die Zahlen in den geschweiften Klammern geben die Werte an, die den entsprechenden Feldern zugewiesen werden. Ein anderes Beispiel ist die Zuordnung von Koordinaten zu einem Punkt:

  ```
  double punkt[ 2 ] = { 3.0, 2.5 };
  ```

  `punkt` ist ein Array mit zwei Feldern, die Werte vom Typ *double* enthalten: `punkt[ 0 ]` erhält den Wert 3.0, `punkt[ 1 ]` den Wert 2.5. Bei grösseren Arrays ist es nicht mehr angebracht, deren Inhalte wie soeben beschrieben zu definieren, sondern mit Hilfe einer *for*-Schleife:

  ```
  int anzahl[ 100 ];

  for ( int i = 0; i < 100; i++ )
     anzahl[ i ] = 0;
  ```

- Da die Zählvariable `i` in der Funktion `gmain()` mehrmals verwendet wird, ist es besser, sie bereits zu Beginn der Funktion zu definieren (dies hat nichts mit Arrays, wohl aber mit guter Programmierung zu tun).

- Der Befehl `random( n );` generiert eine ganzzahlige Zufallszahl zwischen 0 und $n - 1$, in unserem Beispiel also eine Zahl zwischen 0 und 5. Damit bei mehreren Programmdurchläufen nicht immer dieselben Zufallszahlen erzeugt werden (was beim Testen eines Programms durchaus sinnvoll sein kann), ist zu Beginn des Programms der Befehl `randomize();` nötig. Er setzt den Zufallszahlengenerator auf einen zufälligen (von der internen Uhr abhängigen) Wert.

- Damit der Compiler die Befehle `random( 6 );` und `randomize();` erkennt, muss zu Beginn des Programms die Datei `stdlib.h` mit `#include <stdlib.h>` eingebunden werden (vgl. Anhang Anhang D:).

- Die einzelnen Elemente des Arrays `anzahl` können wie Variablen vom Typ *int* bearbeitet werden. Man kann sie u.a. addieren, multiplizieren oder (wie in unserem Beispiel) um 1 erhöhen.

- Eine Besonderheit von Arrays ist deren Übergabe als Parameter an Funktionen. Ohne spezielle Angaben werden Arrays nicht als Werteparameter (vgl. Kapitel 3.3), sondern als **Referenzparameter** (Kapitel 10.2) übergeben werden. Die Eigenschaft von Referenzparametern ist, dass sie in einer Funktion geändert werden können und diese geänderten Werte an die aufrufende Funktion zurückgegeben werden. In unserem Fall heisst das: Werden in der Funktion `relativeHaeufigkeiten` die einzelnen Werte des Arrays `rel` geändert, so werden sie automatisch auch in der aufrufenden Funktion (also in `gmain()`) geändert. Will man sicherstellen, dass die Werte eines Arrays in einer Funktion nicht verändert werden können, übergibt man das Array als einen konstanten Parameter. In unserem Programm hat die Funktion `relativeHaeufigkeiten` folgenden Kopf:

  ```
  relativeHaeufigkeiten ( const int anzahl[], double rel[] )
  ```

Das Array `rel` kann also geändert werden, währenddem der Versuch einer Änderung im Array `anzahl` zu einer Fehlermeldung des Compilers führt.

Beachten Sie auch das Kapitel 10.1.

- Die Grösse des Arrays wird im Funktionskopf nicht angegeben. Es genügt also, statt `rel[ 6 ]` nur `rel[]` zu schreiben.
- Die Ausgabe der Werte wird in einer tabellarischen Form (mit Hilfe des Steuerzeichen `\t` für den Tabulator) realisiert. Da `anzahl` von 0 bis 5, die Augenzahlen eines Würfels aber von 1 bis 6 numeriert sind, ist es nötig, zuerst (statt `i`) die Zahl `i + 1` schreiben zu lassen. Es ist nötig, diese Zahl in Klammern zu schreiben, da sonst der Compiler eine Fehlermeldung herausgibt: `cout << ( i + 1 ) << ...`
- Da `anzahl[ i ]` und `ANZAHLWUERFE` Werte vom Typ *int* sind, ist es nötig vor der Division der beiden Werte `( double )` zu schreiben (vgl. Seite 54).

8.4 Zweidimensionale Arrays

In manchen Anwendungen (z.B. Erstellen einer Multiplikationstafel) ist es sinnvoll, **mehrdimensionale Arrays** zu verwenden.

Beispiel: Das folgende Programm erstellt eine Tafel des kleinen Einmaleins.

```
// MULTITAF.CPP
// Erstellt eine Multiplikationstafel des kleinen Einmaleins.

#include <champ.h>
#include "multitaf.rh"

void gmain ()
{
   int tafel[ 10 ][ 10 ];       // Zweidimensionaler Array mit Groesse 10 * 10
   int i, j;                    // Zaehlvariablen

   cinit( "Das kleine Einmaleins" );

   for ( i = 0; i <= 9; i++ )
      for ( j = 0; j <= 9; j++ )
         tafel[ i ][ j ] = ( i + 1 ) * ( j + 1 );

   cout << "\n\t";              // Ausgabe der ersten Zeile
   for ( i = 0; i <= 9; i++)
      cout << setw( 7 ) << ( i + 1 );

   cout << "\n\n";              // Zwei Leerzeilen zur optischen Verschoenerung
   for ( j = 0; j <= 9; j++ )   // Ausgabe der uebrigen Zeilen
   {
      cout << "\n\n  " << setw( 2 ) << ( j + 1 ) << "\t";
      for ( i = 0; i <= 9; i++ )
         cout << setw( 7 ) << tafel[ i ][ j ];
   }

   cgetch();
   cend();
}
```

Ausgabe:

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


   1         1      2      3      4      5      6      7      8      9     10

   2         2      4      6      8     10     12     14     16     18     20

   3         3      6      9     12     15     18     21     24     27     30

   4         4      8     12     16     20     24     28     32     36     40

   5         5     10     15     20     25     30     35     40     45     50
```

```
   6          6     12     18     24     30     36     42     48     54     60
   7          7     14     21     28     35     42     49     56     63     70
   8          8     16     24     32     40     48     56     64     72     80
   9          9     18     27     36     45     54     63     72     81     90
  10         10     20     30     40     50     60     70     80     90    100
```

Bemerkungen:

- Die Ausgabe im Konsolenfenster zeigt, dass man sich ein zweidimensionales Array wie eine Tabelle vorstellen kann: Der erste Index gibt die Nummer der Spalte an, der zweite Index die Nummer der Zeile.
- Wie üblich beginnt die Numerierung der Zeilen und Spalten bei 0, deshalb wird z.B. das Produkt `2*2` dem Feld `tafel[ 1 ][ 1 ]` bzw. das Produkt `( i + 1) * (j + 1)`dem Feld `tafel[ i ][ j ]` zugeordnet.
- Es ist auch möglich drei-, (vier-, fünf- usw.) -dimensionale Arrays zu definieren.

8.5 Strings als char-Arrays

Arrays, deren Felder Variablen vom Typ *char* enthalten, werden auch als Strings betrachtet:

Beispiel:

```
// STRING4.CPP
// Beispielprogramm zur Arbeit mit char-Arrays als Strings.

#include <champ.h>
#include <string.h>                    // Noetig fuer die string-Befehle
#include "string4.rh"

void gmain ()
{
   char name1[] = "Anna Maria";        // Definition, Initialisierung von name1
   char name2[ 8 ];
   char name3[ 21 ];
   char name4[ 21 ];
   char name5[ 21 ];

   int i;

   cinit( "Strings" );

   name2[ 0 ] = 'A';                   // Initialisierung des Strings name2
   name2[ 1 ] = 'n';
   name2[ 2 ] = 'd';
   name2[ 3 ] = 'r';
   name2[ 4 ] = 'e';
   name2[ 5 ] = 'a';
   name2[ 6 ] = 's';
   name2[ 7 ] = 0;

   CPInputString( "", "Geben Sie einen Namen ein:", name3, 20, "DJ" ).showModal();
   CPInputString( "", "Geben Sie noch einen Namen ein:", name4, 20 ).showModal();

   strcpy( name5, name4 );             // Statt name5 = name4!

   cout << "Erster Name:   " << name1 << "\n";
   cout << "Zweiter Name:  " << name2 << "\n";
   cout << "Dritter Name:  " << name3 << "\n";
   cout << "Vierter Name:  ";

   i = 0;
```

```
   while ( name4[ i ] != 0 )
   {
      cout << name4[ i ];
      i++;
   }
   cout << "\n";

   cout << "Fuenfter Name: " << name5 << "\n";

   CP::msgBoxD();
   cend();
}
```

Ausgabe des Programms:

```
Erster Name:   Anna Maria
Zweiter Name:  Andreas
Dritter Name:  DJ
Vierter Name:  Anna Barbara
Fuenfter Name: Anna Barbara
```

Bemerkungen:

- Ist der Inhalt eines Strings schon bei der Definition bekannt, so muss bei der Definition von `name1` die Grösse des *char*-Arrays zwischen den eckigen Klammern nicht angegeben werden. Der Compiler bestimmt und reserviert den benötigten Speicherplatz.
- Die Felder des Strings/Arrays können auch einzeln eingegeben werden, wie die Initialisierung von `name2` zeigt. Wichtig ist in diesem Zusammenhang, dass dem Feld nach dem letzten Zeichen des Strings noch der sogenannte **Null-Character** (ASCII-Code 0) zugewiesen wird. Der Null-Character zeigt das Ende eines Strings an. Deshalb wird dem String `name1` bei seiner Initialisierung automatisch ein Null-Character angefügt (d.h. `name1[ 10 ]` hat den Wert 0). Es ist deshalb auch darauf zu achten, dass bei Definitionen von Strings genügend Platz reserviert wird.
- **Wichtig:** Wird die Zuweisung des Null-Characters vergessen, so kann das zu Programmabstürzen führen.
- Der String `name3` wird mit einer CPInput-Dialogbox eingegeben. Der Aufruf hat die folgende Form:

  `CPInputString(` *Titel*`,` *Aufforderung*`,` V*ariable*`,` *Länge*`,` *[Default]* `).showModal();`

  Titel ist ein String und wird der Titel der Dialogbox sein; dieser String kann leer sein (`""`). *Aufforderung* ist ebenfalls ein String, in dem die Eingabeaufforderung (z.B. `"Geben Sie einen Namen ein:"`) steht. *Variable* ist der Name des Strings, dem die Eingabe zugeordnet werden soll. *Länge* gibt die maximal mögliche Anzahl der Zeichen an, die der eingegebene String haben darf. (**Vorsicht:** Die erlaubte Länge der Eingabe kann grösser sein als der in der Definition reservierte Platz. Das Programm ist in diesem Fall ebenfalls lauffähig, es kann aber sein, dass andere Variablen überschrieben werden!). Freiwillig ist die Angabe eines Strings (*Default*), der direkt übernommen werden kann (vgl. Eingabe von `name3[]`).
- Weil `name4` und `name5` zwei *char*-Arrays sind, ergibt die Zuordnung `name5 = name4;` eine Fehlermeldung des Compilers. Statt dessen wird die Funktion `strcpy` verwendet: `strcpy( name5, name4 );`. Damit `strcpy` erkannt wird, muss die Datei `string.h` mit dem Befehl `#include <string.h>` eingebunden werden (vgl. Anhang Anhang D:).
- Die Ausgabe der Strings `name1` und `name2` erfolgt wie gewohnt mit `cout`. Der String `name4` wird mit Hilfe einer *while*-Schleife Zeichenweise ausgegeben. Solange das aktuelle Zeichen nicht dem abschliessenden Null-Character entspricht (d.h. ist `name4[ i ] != 0`), wird das Zeichen ausgegeben.

Wie gewohnt ist es mit dem Zuweisungsoperator (=) möglich, einen als *char*-Arrays definierten String einem String-Objekt zuzuordnen:

Beispiel:

```
// STRING5.CPP
// Zuordnung von Strings an String-Objekte.

#include <champ.h>
#include <cstring.h>
#include "string5.rh"

void gmain ()
{
```

```
   char vorname[ 20 ];
   char nachname[ 20 ];
   string ganzerName;

   cinit( "Strings" );

   cout << "Geben Sie einen Vornamen ein: ";
   cin.getline( vorname, sizeof( vorname ) );
   cout << "Geben Sie den Nachnamen ein: ";
   cin.getline( nachname, sizeof( nachname ) );

   ganzerName = vorname;
   ganzerName = ganzerName + " " + nachname;


   cout << "Sie haben den folgenden Namen eingegeben: " << ganzerName << "\n";
   CP::msgBoxD();
   cend();
}
```

Die Ausgabe im Konsolenfenster ist (bei Eingabe von "Hans Jakob" bzw. "Meier"):

```
Sie haben den folgenden Namen eingegeben: Hans Jakob Meier
```

Bemerkungen:

- Ein String mit Leerzeichen kann nicht nur mit `CPInputString` eingegeben werden, sondern auch mit die Methode `getline` der Klasse *istream* (vgl. Kapitel 10.3):

  ```
  cin.getline( nachname, sizeof( nachname ) );
  ```

  Die Parameter der Methode `getline` sind der Name des Strings und die maximal mögliche Grösse des Strings, die am besten mit der Funktion `sizeof` bestimmt wird. Beachten Sie den Unterschied zur Eingabe eines String-Objekts mit `getline`!

- `sizeof` kann auch mit `CPInputString` verwendet werden:

  ```
  CPInputString( "", "name = ", name, sizeof( name ), "DJ" ).showmodal();
  ```

- Natürlich ist es auch möglich, mit `ganzerName = vorname + " " + nachname;` den Namen mit nur einer Anweisung zu definieren.

Manche Funktionen verlangen als Parameter einen String (z.B. `ginit( "Champ" )`). Soll der Anwender den Titel des Graphikfensters selbst eingeben können, so stehen dem Programmierer zwei Möglichkeiten offen:

1. Er definiert einen *char*-Array (z.B. `char titel[ 80 ];`) und lässt dann den Titel vom Anwender (mit `cin.getline( titel, sizeof( titel );` oder) eingeben. Mit `ginit( titel );` erhält das Graphikfenster den gewünschten Titel.
2. Er instanziiert ein String-Objekt (`string titel;`) und lässt wiederum den gewünschten Namen durch den Anwender eingeben (mit `getline`). Die Initialisierung des Graphikfensters mit `ginit( titel );` erzeugt aber eine Fehlermeldung des Compilers; der Parameter in der Klammer darf kein String-Objekt sein. Mit der Methode `c_str`, die aus dem String des String-Objekts ein *char*-Array (samt terminierendem Null-Character) erzeugt, ist aber trotzdem eine Initialisierung möglich, wie das folgende Beispiel zeigt:

Beispiel:

```
// STRING6.CPP
// Zeigt, wie man Strings aus String-Objekten in char-Arrays "verwandelt".

#include <champ.h>
#include <cstring.h>
#include "string6.rh"

void gmain ()
{
   string titel;

   cinit();
   cout << "Dieses Programm generiert ein Graphikfenster mit einem\n";
   cout << "von Ihnen gewaehlten Titel.\n\n";
   cout << "Geben Sie den Titel des Graphikfensters ein: ";
   getline( cin, titel );
```

```
   cend();

   ginit( titel.c_str() );
   gline( 0, 0, 1, 1 );                // Zeichnet Diagonale im Graphikfenster.

   CP::msgBoxD();
   gend();
}
```

9 Graphik

9.1 Koordinatengraphik

9.1.1 Grundlagen

Für gewisse Graphiken ist die Turtle-Graphik zu aufwendig, z.B. wenn eine Linie vom Punkt (10/20) zum Punkt (30/-45) gezogen werden soll: Setzen der Turtle auf den ersten Punkt, Ausrichten auf den zweiten Punkt, Bestimmen der Distanz, vorwärts fahren um diese Distanz. Das Programm `turtle1.cpp` auf der Seite 75 zeigt ein solches Beispiel.

Man kann aber auch die Graphiken mit der **Koordinatengraphik**, d.h. ohne die Turtle-Graphik zeichnen.

Beispiel: Das folgende Programm zeichnet ein Dreieck mit der Koordinatengraphik.

```
// GRAPHIK1.CPP
// Zeichnet ein Dreieck mit Koordinatengraphik.

#include <champ.h>
#include "graphik1.rh"

void gmain ()
{
   ginit( "Graphikfenster" );           // Initialisiere Graphikfenster

   gline( 0.1, 0.1, 0.5, 0.9 );         // Zeichne Linie von Punkt ( 0.1, 0.1 )
   gline( 0.5, 0.9, 0.9, 0.1 );         // zum Punkt ( 0.5, 0.9 )
   gline( 0.9, 0.1, 0.1, 0.1 );

   CP::msgBoxD();
   gend();
}
```

Bemerkungen:

- Beachten Sie, dass bei der Verwendung der Koordinatengraphik der Befehl `#define GLOBAL_TURTLE` nicht verwendet wird.
- `ginit( "Graphikfenster" );` initialisiert, genau wie bei der Turtle-Graphik, ein Graphikfenster mit dem Titel "Graphikfenster". Die Koordinaten des Graphikfensters sind so definiert, dass links unten der Punkt (0/0) und oben rechts der Punkt (1/1) ist.

Soll ein Linienzug gezeichnet werden, kann man statt `gline` auch die Befehle `gpos` und `gdraw` verwenden.

Beispiel: Das folgende Programm zeichnet ein Dreieck mit den Befehlen `gpos` und `gdraw`.

```
// GRAPHIK2.CPP
// Zeichnet ein Dreieck mit Koordinatengraphik

#include <champ.h>
#include "graphik2.rh"

void gmain ()
{
   ginit( "Graphikfenster" );   // Initialisiere Graphikfenster

   gpos( 0.1, 0.1 );            // Positioniert den Stift im Punkt ( 0.1, 0.1 )
   gdraw( 0.5, 0.9 );           // Zieht den Stift zum Punkt ( 0.5, 0.9 )
   gdraw( 0.9, 0.1 );
   gdraw( 0.1, 0.1 );

   CP::msgBoxD();
   gend();
}
```

Bemerkungen:

- Der Befehl `gpos( 0.1, 0.1 );` positioniert einen Zeichenstift *ohne zu zeichnen* auf den angegebenen Koordinaten.

- Der Befehl `gdraw( 0.5, 0.9 );` zeichnet eine Linie vom Standort des Zeichenstifts zum Punkt mir den angegeben Koordinaten. Anschliessend wird der Stift auf die neuen Koordinaten gesetzt.

Es ist möglich, im Graphikfenster eigene Koordinaten (sogenannte **User-Koordinaten**) zu definieren:

Beispiel: Das folgende Programm zeichnet ein Dreieck in einem selbst gewählten Koordinatensystem:

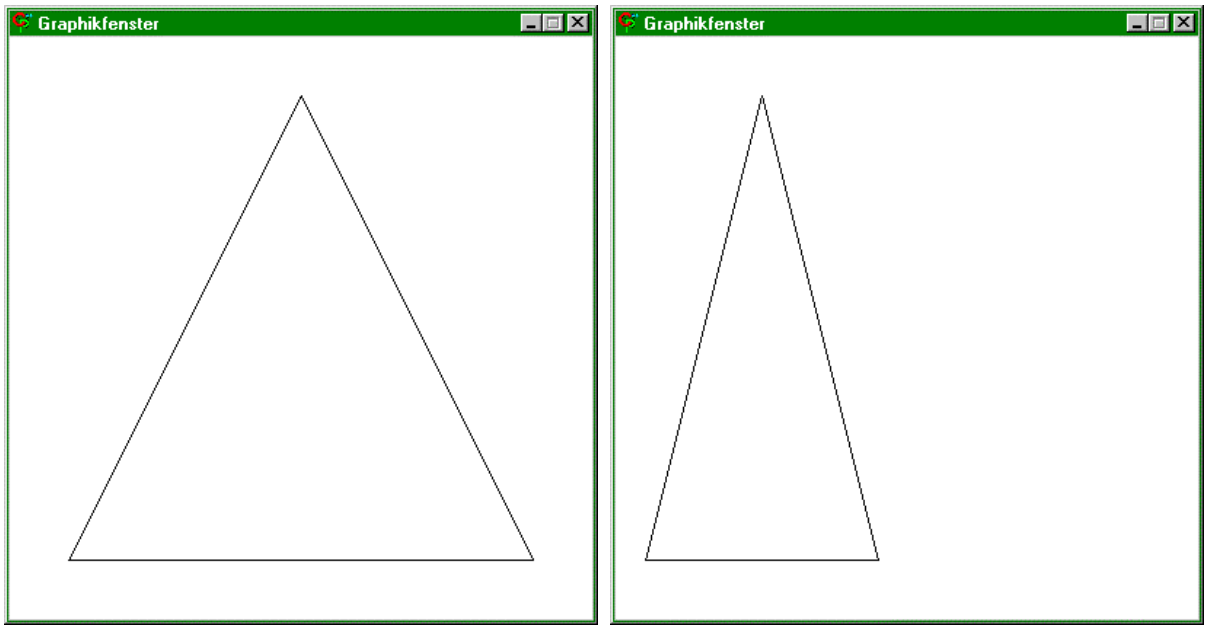
```
// GRAPHIK3.CPP
// Zeichnet ein Dreieck in einem selbstdefinierten Koordinatensystem.

#include <champ.h>
#include "graphik3.rh"

void gmain ()
{
   ginit( "Graphikfenster" );   // Initialisiere Graphikfenster
   gwindow( 0, 100, 0, 100 );   // Definiert ein eigenes Koordinatensystem
                                // Linke untere Ecke: ( 0, 0 )
   gpos( 10, 10 );              // Rechte obere Ecke: ( 100, 100 )
   gdraw( 50, 90 );
   gdraw( 90, 10 );
   gdraw( 10, 10 );

   CP::msgBoxD();
   gend();
}
```

Die Koordinaten werden mit dem Befehl `gwindow( ... );` definiert, der folgende Form hat:

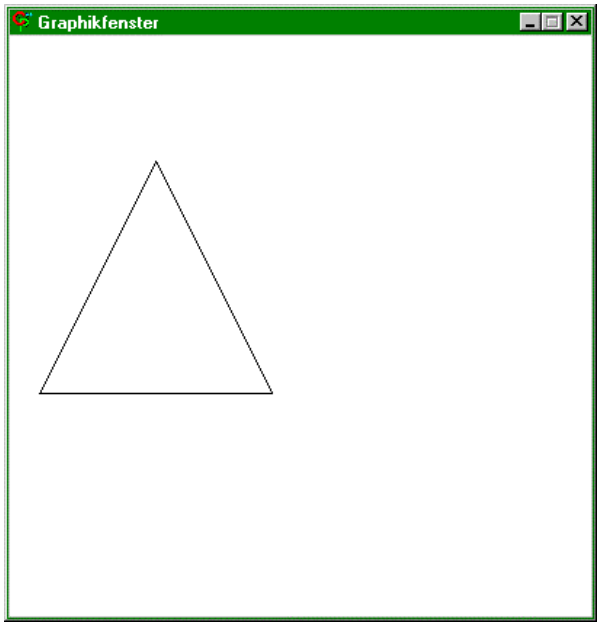
`gwindow(` *xleft, xright, ybottom, ytop* `);`

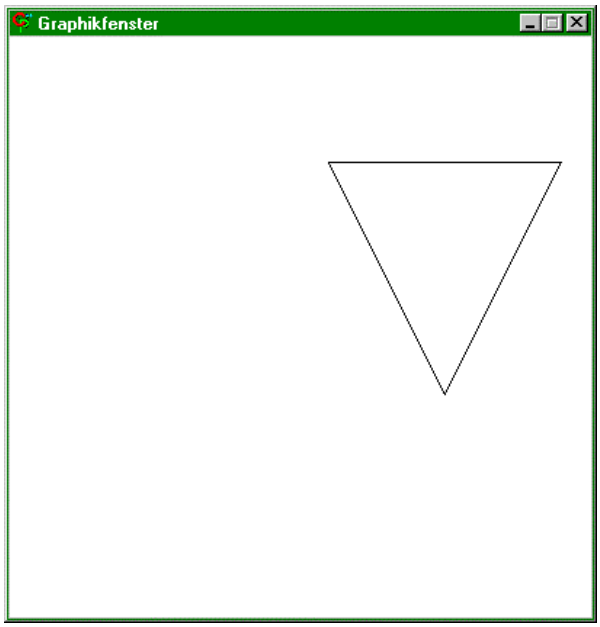
Die einzelnen Werte sind vom Typ *double* und haben folgende Bedeutung: der erste Parameter (*xleft*) gibt die *x*-Koordinate des linken Randes, der zweite Parameter die *x*-Koordinate des rechten Randes (*xright*), der dritte die *y*-Koordinate des unteren Randes (*ybottom*) und die vierte die *y*-Koordinate des oberen Randes (*ytop*) an. In unserem Fall hat die linke untere Ecke die Koordinaten (0/0), die rechte obere Ecke die Koordinaten (100/100).

Das Dreieck, das gezeichnet wird, ist dasselbe wie in den ersten beiden Programmen (Graphik links). Gibt man aber andere Koordinaten ein (z.B. `gwindow( 0, 200, 0, 100 );`), so wird das Dreieck unter Umständen verzerrt (Graphik rechts).



Oft sind die gewählten Bereiche für die *x*-Richtung und die *y*-Richtung verschieden gross. Gibt man dann den Befehl `gwindowEven( 0, 200, 0, 100 );` (achten Sie auf das grosse ‘E’ mitten im Befehl) ein, so erreicht man, dass die Graphik (insbesondere die Kreise) nicht verzerrt wird. (Graphik unten links).





Unüblich (aber nicht unmöglich) ist es, die Koordinaten so zu setzen, dass sich oben rechts der Punkt (0/0) und unten links der Punkt (200/100) befinden. Mit `gwindowEven( 200, 0, 100, 0 );` erhält man die Graphik oben rechts.

Im Befehl `ginit` kann u.a. noch die Grösse des Graphikfensters beeinflusst werden:

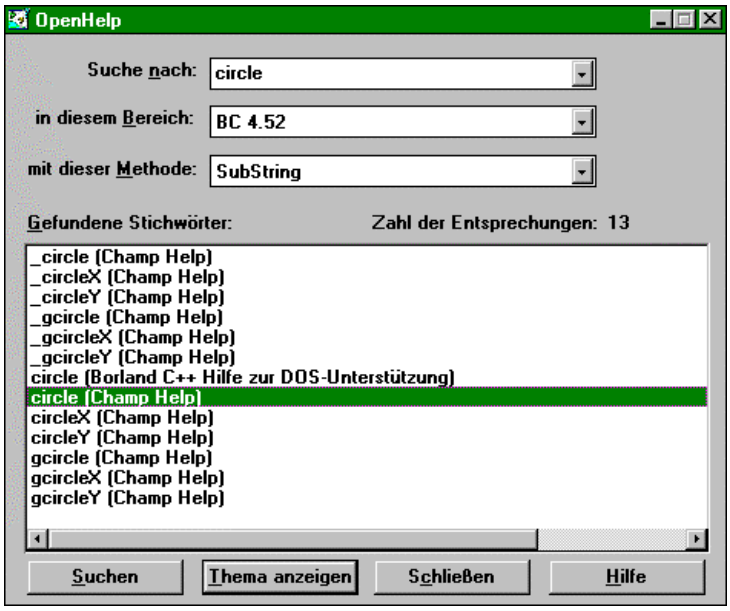
Befehl:	**Fenstergrösse:**	**vordefinierte Koordinaten:**
`ginit();`	Standardgrösse, quadratisch.	Die x-Koordinaten und y-Koordinaten liegen in allen drei Fällen zwischen 0 bis 1.
`ginit( "Champ" );`	Standardgrösse (400×400 Pixel).	
`ginit( "Champ", CP::WinBigSq );`	So gross als möglich, vom Bildschirm abhängig; quadratisch.	
`ginit( "Champ", CP::WinBig4by3 );`	So gross als möglich; für das Fenster gilt: Länge : Breite = 4 : 3.	x-Koordinaten: zwischen $-\frac{1}{3}$ und 1, y-Koordinaten: zwischen 0 und 1.
`ginit( "Champ", CP::WinMax );`	Ganzer Bildschirm.	Vom Bildschirm abhängig.

9.1.2 Die Champ-Hilfe

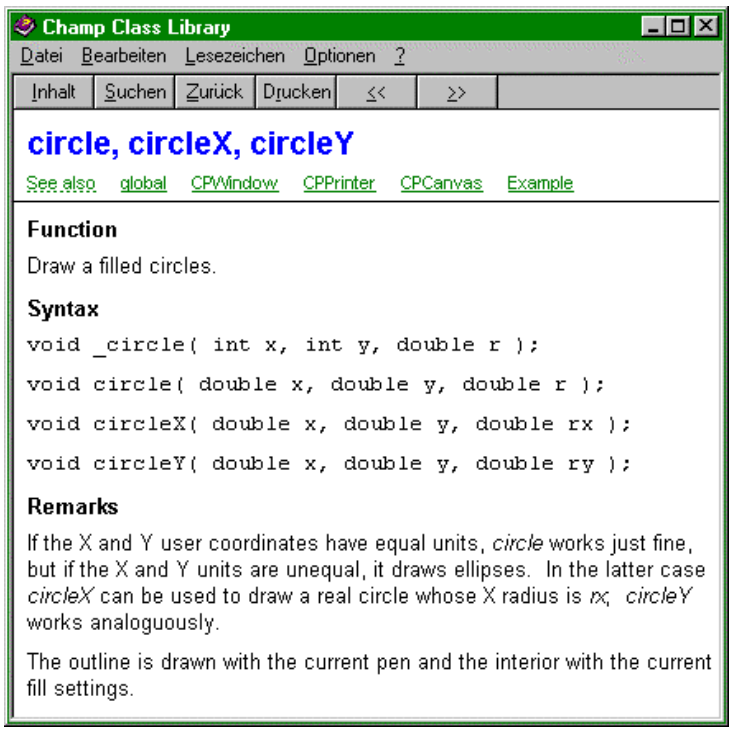
Problem: Im Graphikfenster soll auf schwarzem Grund ein weisser Kreis gezeichnet werden.

Anhand dieses Problems soll gezeigt werden, wie man sich mit der (Champ-)Hilfe Informationen zu den Befehlen beschaffen kann. Ein typisches Vorgehen ist etwa das folgende:

Zeichnen des Kreises:



1. Erstellen Sie wie gewohnt ein neues Projekt.
2. Das englische Wort für "Kreis" ist "circle". Geben Sie im eröffneten Projekt das Wort `circle` ein. Verschieben Sie den Cursor auf dieses Wort und drücken Sie die Taste <F1>. (Statt der Taste <F1> können Sie auch auf das entsprechende Symbol in der Mauspalette klicken oder mit der rechten Maustaste das entsprechende Kontextmenü öffnen und dort den Punkt Hilfethema aufrufen wählen.) Es erscheint das Fenster rechts.
1. Es ist wichtig, dass sich der Cursor wirklich auf dem gewünschten Wort `circle` befindet, da sich <F1> bzw. die Maus auf den aktuellen Standort des Cursors beziehen. (In Textverarbeitungen ist es zum Teil anders: Da beziehen sich die sogenannten Kontextmenüs (vgl. Seite 9) auf die Position des Mauszeigers.)
2. Klicken Sie mit der Maus im neuen Fenster auf circle (Champ Help) und dann auf den Punkt Thema anzeigen. Ein Fenster mit den Informationen zur Funktion `circle(...);` erscheint.
3. Aus den Informationen entnehmen wir, dass der Befehl `circle(...);` drei Parameter hat: *x*- und *y*-Koordinate des Mittelpunkts sowie den Radius des Kreises.
4. **VORSICHT:** Allen Graphikbefehlen, die in der Champ-Hilfe angegeben sind, muss noch ein ‘g’ voran gestellt werden. Der Befehl zum Zeichnen eines Kreises lautet also `gcircle` statt `circle`.
5. Geben Sie im Programm den Befehl `gcircle ( 0.5, 0.5, 0.4 );` sowie `CP::msgBoxD();` und `gend();` ein. Starten Sie das Programm.



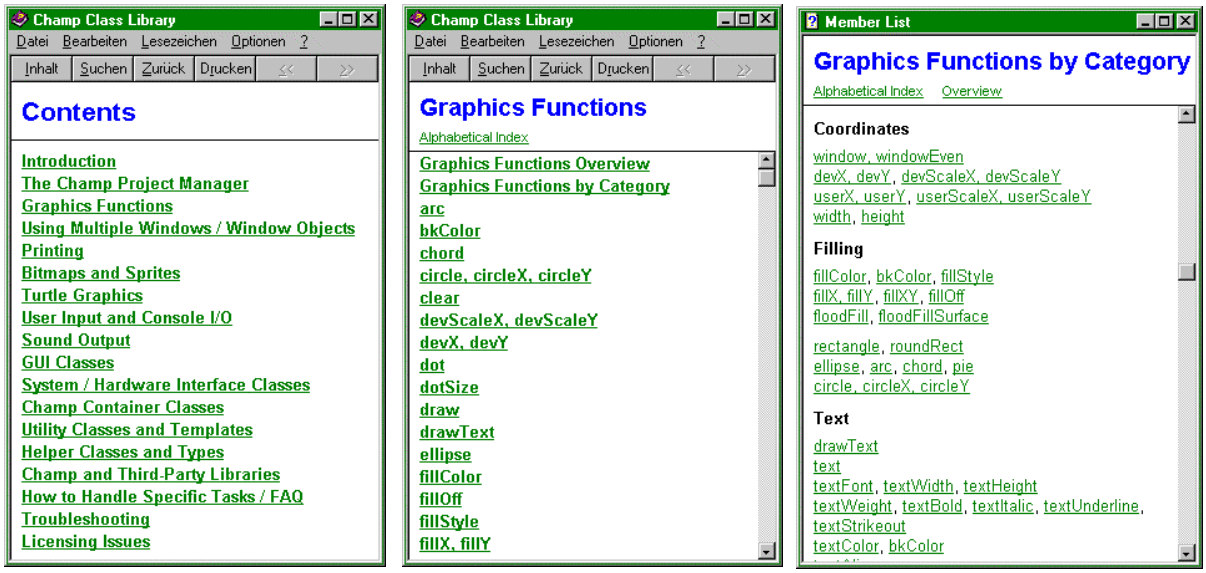
Gezeichnet wird (im Moment eher überraschend) eine graue Kreisscheibe mit schwarzem Rand. Den Grund erklärt uns der letzte Satz im Hilfe-Fenster: "The outline is drawn with the current pen and the interior with the current fill settings." ("Die Grenzlinie wird mit dem aktuellen Stift und das Innere mit dem aktuellen Füllmodus gezeichnet.") Es ist deshalb sinnvoll, uns mit den Füllbefehlen zu beschäftigen.

Zeichnen des Kreises ohne "Füllung":

1. Um eine Übersicht über die Champ-Hilfethemen zu erhalten, klicken Sie mit der *rechten* Maustaste auf die Champ-Schaltfläche in der Titelleiste. Es erscheint das linke Fenster.

2. Weil wir eine Graphik zeichnen, brauchen wir natürlich Informationen über Graphikfunktionen. Klicken Sie im Fenster auf die entsprechende Zeile (Graphics Functions). Es erscheint eine alphabetische Liste aller Graphikbefehle (vgl. mittleres Fenster).

1. Um eine nach Themen geordnete Liste der Graphikbefehle zu erhalten, klicken Sie auf Graphics Functions by Category. Die Liste erscheint in einem neuen Fenster (rechts).



1. "Filling" ist das englische Wort für "Füllung", also wird die gesuchte Funktion in diesem Abschnitt zu finden sein. Nach einem Mausklick auf das Stichwort fillStyle erscheinen die Informationen zum gleichnamigen Befehl `fillStyle`. Darin steht unter anderem, dass der konstante Parameter `EMPTY_FILL` bewirkt, dass die Kreisfläche nicht gefüllt wird. Ergänzen Sie das Programm (vor dem Befehl `gcircle` (!)) mit dem Befehl `gfillStyle( EMPTY_FILL );` (achten Sie wiederum auf das vorangehende ‘g’!).

Ein zweiter Start des Programms zeigt uns, dass nun eine schwarze Kreislinie auf einem immer noch weissen Hintergrund gezeichnet wird.

Setzen der Farben:

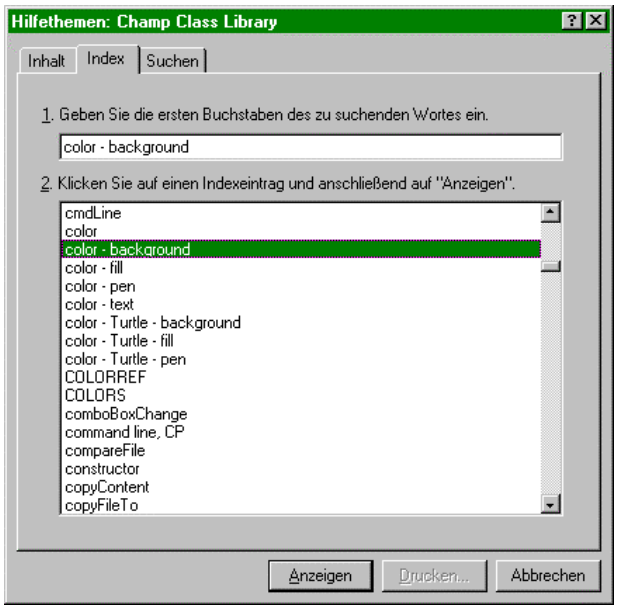
1. Aktivieren Sie wiederum das Hilfe-Fenster.

2. Wählen Sie den Punkt Suchen.

3. Geben Sie im ersten Feld des neu erschienenen Fensters die Buchstaben `color` ein.

4. Wählen Sie color - background und klicken Sie dann auf Anzeigen. Ein neues Fenster erscheint, in dem Sie die Zeile bkColor anklicken und nochmals mit Anzeigen bestätigen müssen. Nun erscheint das Fenster mit den Informationen zum Befehl `bkColor`.

5. Die Funktion `bkColor` verlangt einen Parameter, nämlich die Hintergrundfarbe, die verwendet werden soll. Damit diese wirklich auch erscheint, muss nachher der Befehl `clear` verwendet werden. Wenn Sie mit der Maus auf Example klicken erscheint im Hilfe-Fenster das folgende Programm, das uns zeigt, wie die Hintergrundfarbe gesetzt wird:




```
// bkColor Example
```

```
#include <champ.h>

void gmain()
{
   ginit();
   gbkColor( GREEN );
   gclear();
}
```

6. Mit der Wahl Zurück kehren Sie zurück zum letzten Fenster (`bkColor`). Klicken Sie in diesem Fenster auf CPColor (wegen "See the CPColor class for information on how to specify colors."). Wählen Sie im neuen Fenster mit der Maus den Punkt enum COLORS, es erscheint eine Liste der Farben, bzw. der vordefinierten (*int*-)Konstanten, welche diesen Farben entsprechen (vgl. Tabelle).

Name:	**Wert:**	**Bedeutung:**	**Name:**	**Wert:**	**Bedeutung:**
BLACK	0	schwarz	DARKGRAY	8	dunkelgrau
BLUE	1	blau	LIGHTBLUE	9	hellblau
GREEN	2	grün	LIGHTGREEN	10	hellgrün
CYAN	3	türkis	LIGHTCYAN	11	helltürkis
RED	4	rot	LIGHTRED	12	hellrot
MAGENTA	5	fuchsinrot	LIGHTMAGENTA	13	hell-fuchsinrot
BROWN	6	braun	YELLOW	14	gelb
LIGHTGRAY	7	hellgrau	WHITE	15	weiss

7. Ergänzen Sie das Programm an passender Stelle mit den beiden Befehlen `gbkColor( BLACK );` und `gclear();`. Wird das Programm jetzt gestartet, so wird ein schwarzer Kreis auf schwarzem Hintergrund gezeichnet!

8. Sucht man mit eine der nun vorgestellten Arten weiter in der Champ-Hilfe, so erfährt man, dass die gewünschte Zeichenfarbe mit dem Befehl `gpenColor` gesetzt wird. Ergänzen Sie also das Programm mit `gpenColor( WHITE );`.

Das ganze Programm, ergänzt mit Kommentar, sieht so aus:

```
// GRAPHIK4.CPP
// Zeichnet einen weisse Kreislinie auf schwarzem Hintergrund.

#include <champ.h>
#include "graphik4.rh"

void gmain ()
{
   ginit( "Champ" );

   gfillStyle( EMPTY_FILL );    // Flaechen sollen nicht gefuellt werden
   gbkColor( BLACK );           // Hintergrundfarbe schwarz
   gpenColor( WHITE );          // Zeichenfarbe weiss
   gclear();                    // Loeschen des Graphikbildschirms, zugleich
                                // "zeichnen" des Hintergrunds
   gcircle( 0.5, 0.5, 0.4 );    // Zeichne Kreis (M = ( 0.5, 0.5 ), r = 0.4)

   CP::msgBoxD();
   gend();
}
```

Es gibt noch weitere Möglichkeiten, zu den gewünschten Informationen zu kommen: Wenn man z.B. im Fenster des Befehls `bkColor` auf See also klickt, so erscheint das folgende Fenster, das auf verwandte Befehle hinweist. Mit einem Mausklick auf das gewünschte Stichwort wird das entsprechende Fenster geöffnet.

9.1.3 Textausgabe in der Graphik

Es ist möglich, mit dem Befehl `gtext` in Graphiken auch Text auszugeben: Dies kann auf zwei Arten geschehen:

1. `gtext( "Hallo Welt!" );` bzw. `gtext( x, y, "Hallo, Welt!" );`. Im ersten Fall wird der Text dorthin geschrieben, wo sich der Zeichenstift im Moment befindet (vgl. Seite 67), im zweiten Fall kann man die Koordinaten selbst angeben.
2. `gtext() << "Hallo Welt";` bzw. `gtext( x, y ) << "Hallo Welt!";`. Der Unterschied zwischen diesen beiden Befehlen ist derselbe wie zwischen den zwei Befehlen des ersten Beispiels.
3. Es können auch Variablenwerte ausgegeben werden (z.B. `gtext() << "i = " << i;`).

Beispiel:

```
// GRAPHIK5.CPP
// Zeichnet ein Dreieck und schreibt zusaetzlichen Text im Graphikfenster.

#include <champ.h>
#include "graphik5.rh"

void gmain ()
{
   ginit( "Graphikfenster" );
   gwindow( 0, 100, 0, 100 );

   gpos( 10, 10 );
   gdraw( 50, 90 );
   gdraw( 90, 10 );
   gdraw( 10, 10 );

   gpos( 40, 40 );                                  // Positionieren des Stifts
   gtext() << "Hallo Welt!";                        // Textausgabe
   gtext( 30, 5 ) << "Druecken Sie eine Taste!"; // Textausgabe an selbst-
                                                    // definierter Stelle
   getch();
   gend();
}
```

Bemerkung:

Man kann das Schriftbild, die Grösse, die Farbe, die Ausrichtung usw. des Textes beeinflussen. In der Champ-Hilfe können Sie selbst nachschlagen, wie die exakten Befehle lauten.

9.1.4 Zeichnen eines Funktionsgraphen

Beispiel: Es soll der Graph der Funktion $f(x) = \sin x$ gezeichnet werden.

```
// SINUS.CPP
// Zeichnet den Graphen der Sinusfunktion

#include <champ.h>
#include "sinus.rh"

const float XMIN = -0.1;
const float XMAX = 6.5;
const float YMIN = -1.1;
const float YMAX = 1.1;

void zeichneKoordinatensystem ();

void gmain ()
{
   float dx;                                  // Schrittweite in for-Schleife
   float y;                                   // y = sin( x );
   ginit( "Sinusfunktion", CP::WinBigSq );    // Graphik in einem grossen
   gwindowEven( XMIN, XMAX, YMIN, YMAX );     // Graphikfenster
   dx = ( XMAX - XMIN ) / gwidth();

   zeichneKoordinatensystem();

   y = sin( XMIN );
   gpos( XMIN, y );
   gputPixel( XMIN, y, BLACK );               // Setze einen schwarzen Punkt

   for ( float x = XMIN; x <= XMAX; x = x + dx )
   {
```

```
      y = sin( x );
      gdraw( x, y );
   }

   CP::msgBoxD();
   gend();
}

void zeichneKoordinatensystem ()             // Zeichnet Koordinatensystem
{
   gline( XMIN, 0, XMAX, 0 );
   gline( 0, YMIN, 0, YMAX );
   for ( int x = ceil( XMIN ); x <= XMAX; x++ )
   {
      gline( x, -0.05, x, 0.05 );            // Beschriften der x-Achse
      gtext( x, -0.2 ) << x;
   }
   for ( int y = ceil( YMIN ); y <= YMAX; y++ )
   {
      gline( -0.05, y, 0.05, y );            // Beschriften der y-Achse
      if ( y != 0 )                          // Der Ursprung soll nicht zweimal
         gtext( 0.05, y ) << y;              // angeschrieben werden!
   }

   gpos( 3, 2 );
   gtextFont( "Times New Roman", 0.5 );      // Schriftart und -hoehe
   gtextItalic();                            // Umschalten auf Kursivschrift
   gtext() << "f(x) = ";
   gtextItalic( false );                     // Umschalten auf Normalschrift
   gtext() << "sin ";
   gtextItalic();
   gtext() << "x";
}
```

Bemerkungen:

- Zur Definition der Koordinaten im Graphikfenster werden die Konstanten XMIN, XMAX, YMIN und YMAX verwendet. Dies hat den Vorteil, dass bei einer Änderung der Koordinaten nur diese vier Konstanten und nicht alle Werte im Programm geändert werden müssen.
- Die Idee der Variablen `dx` ist die folgende: Damit die Schrittweite in der *for*-Schleife des Hauptprogramms nicht zu weit oder zu eng ist, wird mit `gwidth();` die Anzahl der Bildschirmpixel in der *x*-Richtung des Graphikfensters bestimmt. Zur Berechnung der richtigen Schrittweite `dx` wird die Breite des Graphikfensters (`XMAX - XMIN`) durch die Anzahl der Pixel in *x*-Richtung dividiert. (Der entsprechende Befehl zur Bestimmung der Anzahl der Pixel in *y*-Richtung lautet `gheight();`).
- Es ist möglich, alle berechneten Punkte des Graphen mit `gputPixel( XMIN, y, BLACK );` zu setzen. Dies ist aber nicht bei allen Funktionen zu empfehlen. Vor allem bei Funktionen, die "zu steil" sind, erscheinen dann statt eines "schön gezeichneten" Graphen nur einzelne Punkte. Deshalb wird in diesem Programm (mit `gpos` und `gputPixel`) nur der erste Punkt gesetzt (was eigentlich auch überflüssig wäre). Die weiteren Punkte werden mit `gdraw` miteinander verbunden.
- Der Befehl `x = ceil( XMIN )` berechnet die kleinste ganze Zahl `x`, die grösser oder gleich `XMIN` ist. Entsprechend berechnet der Befehl `x = floor( XMIN )`die grösste ganze Zahl `x`, welche kleiner oder gleich `XMIN` ist.
- Mit den Befehlen und `gtextFont( "Times New Roman", 0.5 );` und `gtextItalic();` werden die Schriftart ("Times New Roman" *kursiv* und normal) und Schrifthöhe (0.5 Einheiten) gesetzt. Will man einen Text im Graphikfenster in Kursivschrift ausgeben, so muss man den Befehl `gtextItalic();` (oder auch `gtextItalic( true );` eingeben, mit `gtextItalic( false );` stellt man wieder auf die Normalschrift zurück. Zu den weiteren Möglichkeiten dieser Befehle beachten Sie bitte die Champ-Hilfe.
- Beachten Sie, dass die Winkel den trigonometrischen Funktionen im Bogenmass übergeben werden.

9.2 Ergänzungen zur Turtle-Graphik

Will man in einer Graphik die Turtle verwenden, so ist **vor** dem Befehl `#include <champ.h>` noch der Befehl `#define GLOBAL_TURTLE` nötig, andernfalls erkennt der Compiler die Turtle-Befehle nicht, was zu Fehlermeldungen führt. Das folgende Programm verwendet einige neue Befehle der Turtle-Graphik:

Beispiel:

```
// TURTLE1.CPP
// Zeichnet ein rechtwinkliges Dreieck mit den Katheten 60 und 80.

#define GLOBAL_TURTLE

#include <champ.h>
#include "turtle.rh"

void gmain ()
{
   ginit( "Champ" );
   gwindow( -100, 100, -100, 100 );  // Definition der User-Koordinaten

   turtleColor( LIGHTRED );         // Setze die Farbe der Turtle und die
   penColor( LIGHTRED );            // Farbe des Zeichenstiftes auf hellrot

   forward( 60 );
   penUp();                          // Anheben des Zeichenstifts
   back( 60 );                       // Gehe 60 Schritte rueckwaerts
   penDown();                        // Absetzen des Zeichenstifts
   right( 90 );
   forward( 80 );
   setHeading( towards( 0, 60 ) );   // Drehe die Turtle Richtung Punkt ( 0, 60 )
   forward( distance( 0, 60 ) );     // Fahre bis zum Punkt ( 0, 60 )

   CP::msgBoxD();
   gend();
}
```

Bemerkungen:

- Wenn nichts anderes definiert ist, gehen die Koordinaten im Graphikfenster in beiden Richtungen von -200 bis 200, die Turtle wird auf den Ursprung gesetzt und blickt nach oben. Mit `gwindow` können aber die Koordinaten wie gewohnt geändert werden.
- Mit den Befehl `turtleColor( LIGHTRED );` und `penColor( LIGHTRED );` werden die Farbe der Turtle sowie die Zeichenfarbe der Turtle auf hellrot gesetzt.
- `penUp();` bewirkt, dass der Zeichenstift angehoben wird. Bei Bewegungen der Turtle wird nichts gezeichnet. Mit `penDown();` wird der Stift wieder aufgesetzt.
- `back( 60 );` bewegt die Turtle 60 Schritte **rückwärts**.

- Mit `setHeading` wird die Turtle in eine bestimmte Richtung gedreht; z.B. bewirkt `setHeading( 30 );`, dass die Turtle 30° gegenüber der Vertikalen gedreht ist. Will man erreichen, dass die Turtle in die Richtung eines bestimmten Punktes zeigt, verwendet man die Funktion `towards( x, y )`, die den Winkel zwischen der aktuellen Position der Turtle und dem Punkt (*x* / *y*) bestimmt. Dieser Winkel wird von der Vertikalen aus im Uhrzeigersinn gemessen.
- Ist die Länge einer zu zeichnenden Strecke nicht bekannt, so kann sie mit `distance( x, y );`, die den Abtand zwischen der aktuellen Position und dem Punkt (*x* / *y*) bestimmt, ermittelt werden. Zusammen mit den zwei Befehlen `setHeading` und `towards` führt das zu einer (allerdings etwas umständlichen) Möglichkeit, eine Linie zu einem bestimmten Punkt zu zeichnen:

  ```
  setHeading( towards( 0, 60 ) );
  forward( distance( 0, 60 ) );
  ```

Es ist möglich, in einem Programm Turtle-Graphik **und** Koordinatengraphik zu verwenden, wie das folgende Beispiel zeigt:

Beispiel: Gezeichnet wird ein rechtwinkliges Dreieck mit hellroten Katheten und einer grünen Hypotenuse.

```
// TURTLE2.CPP
// Zeichnet ein rechtwinkliges Dreieck mit den Katheten 60 und 80.
// Befehle der Turtle-Graphik und der Koordinatengraphik werden vermischt.
```

```
#define GLOBAL_TURTLE

#include <champ.h>
#include "turtle2.rh"

void gmain ()
{
   ginit( "Champ" );
   gwindow( -100, 100, -100, 100 );

   penColor( LIGHTRED );             // Setze Zeichenfarbe der Turtle
   gpenColor( GREEN );               // Setze Zeichenfarbe des Stiftes

   forward( 60 );                    // Zeichnen der Katheten mit der Turtle
   home();                           // Setze Turtle auf den Ursprung
   right( 90 );
   forward( 80 );
   hideTurtle();
   gline( 80, 0, 0, 60 );            // Zeichnen der Hypotenuse
                                     // mit Koordinatengraphik
   CP::msgBoxD();
   gend();
}
```

Bemerkungen:

- Die Befehle `penColor` und `gpenColor` beeinflussen sich nicht, da der erste Befehl die Zeichenfarbe der Turtle, der zweite Befehl aber die Zeichenfarbe des "normalen" Zeichenstifts setzt.
- Mit dem Befehl `home();` wird die Turtle in die Ausgangslage zurück gesetzt, sie befindet sich also auf dem Ursprung und blickt nach oben.
- Da `gline( 80, 0, 0, 60 );` ein Befehl der Koordinatengraphik ist, wird die Linie, die gezeichnet werden soll, mit den Einstellungen der Koordinatengraphik (in unserem Beispiel insbesondere also der grünen Zeichenfarbe) gezeichnet. Da die Katheten aber mit den Einstellungen der Turtle-Graphik (insbesondere Zeichenfarbe hellrot) gezeichnet werden, hat das Dreieck am Schluss eben hellrote Katheten und eine grüne Hypotenuse.

Für die Verwendung von weiteren Befehlen der Turtle-Graphik wird auf die Champ-Hilfe verwiesen (z.B. unter dem Stichwort "Turtle Graphics" in der Übersicht über die Hilfethemen (vgl. Seite 71).

10 Ergänzungen

10.1 Rückgabewerte von Funktionen

In vielen mathematischen Funktionen berechnet man aus "eingegebenen" Werten (Argumenten) einen Funktionswert, der zu einer bestimmten Bildmenge gehört. Auch in C++ können Funktionen Werte zurückgeben, die einen bestimmten Datentyp haben. Wir haben bereits einige dieser Funktionen verwendet; z.B. die Funktion `sqrt`, die einen Wert vom Typ *double* an die aufrufende Funktion zurückgibt. Dieser Wert kann einer Variablen zugeordnet werden: `y = sqrt( x );`. Natürlich können auch neu definierte Funktionen einen Wert an die aufrufende Funktion zurückgeben.

Beispiel: Das folgende Programm berechnet die Länge der Hypotenuse eines rechtwinkligen Dreiecks.

```
// PYTHAG2.CPP
// Berechnet die Hypotenuse eines rechtwinkligen Dreiecks aus seinen Katheten.

#include <champ.h>
#include "pythag2.rh"

double berechneHypotenuse ( double a, double b );

void gmain ()
{
   double a, b;
   double c;

   cinit( "Pythagoras" );
   cout << "Dieses Programm berechnet die Hypotenuse c eines rechtwinkligen\n";
   cout << "Dreiecks aus seinen Katheten a und b.\n\n";
   cout << "Geben Sie die Laengen der Katheten ein:\n";
   cout << "a = ";
   cin >> a;
   cout << "b = ";
   cin >> b;

   c = berechneHypotenuse( a, b );

   cout << "\nLaenge der Hypotenuse: c = " << c;

   CP::msgBoxD();
   cend();
}

double berechneHypotenuse ( double a, double b )
{
   double c;

   c = sqrt( pow( a, 2 ) + pow( b, 2 ) );    // Berechnung der Hypotenuse
   return c;                                 // Der Wert von c wird an die auf-
}                                            // rufende Funktion zurueckgegeben
```

Bemerkungen:

- Wegen der Angabe `double ( ... )` gibt die Funktion `berechneHypotenuse` einen Wert vom Typ *double* an das Hauptprogramm zurück. Dieser Wert wird der Variablen `c` zugeordnet. (Beachten Sie, dass `c` in der Funktion `berechneHypotenuse` ein lokale Variable ist, die nichts mit der Variablen `c` des Hauptprogramms zu tun hat; vgl. Kapitel 4.2.)
- Der Befehl `return` hat zur Folge, dass die Funktion sofort beendet wird und der Ausdruck, der hinter dem Befehl steht, an die aufrufende Funktion zurückgegeben wird.

- Es ist möglich, die Funktion `berechneHypotenuse` auch so zu schreiben:

```
double berechneHypotenuse ( double a, double b )
{
   return sqrt( pow( a, 2 ) + pow( b, 2 ) );
```

```
   }
```

10.2 Übergabe von Referenzparametern

Soll eine Funktion mehrere in ihr berechneten Werte an die aufrufende Funktion zurückgeben (z.B. die zwei Lösungen einer quadratischen Gleichung), so ist das nicht mit `return` möglich. Man kann aber Parameter als **Referenzparameter** an die Funktion übergeben, d.h. als Parameter, deren Wert in der Funktion verändert werden darf.

Beispiel: Das folgende Programm berechnet die Lösungen einer quadratischen Gleichung (Grundmenge: **R**).

```
// QUADRGL1.CPP
// Berechnet die Loesungen einer quadratischen Gleichung in der Grundmenge R.

#include <conio.h>              // Noetig fuer den Befehl clrscr();
#include <champ.h>
#include "quadrgl1.rh"

int berechneLoesungen ( double a, double b, double c, double & x1, double & x2 );

const char ESC = 27;            // ASCII-Code der ESC-Taste

void gmain ()
{
   double a, b, c;              // Koeffizienten
   double loesung1, loesung2;   // Loesungen
   int i;                       // Ist i = 2, dann hat die Gleichung 2 Loesungen
                                // Ist i = 1, dann hat die Gleichung 1 Loesung
                                // Ist i = 0, dann hat die Gleichung keine Loesung

   cinit( "Quadratische Gleichungen" );
   cout << "Dieses Programm loest die quadratische Gleichung ax^2 + bx + c = 0.";

   do
   {
      clrscr();                 // Loescht den Inhalt des Konsolenfensters

      CPInputDouble( "", "Geben Sie den Koeffizienten a ein", a, 1 ).showModal();
      CPInputDouble( "", "Geben Sie den Koeffizienten b ein", b, 3 ).showModal();
      CPInputDouble( "", "Geben Sie den Koeffizienten c ein", c, 2 ).showModal();

      cout << "Sie haben die Gleichung " << a << "x^2 + " << b << "x + "
           << c << " = 0 eingegeben.\n\n";

      i = berechneLoesungen( a, b, c, loesung1, loesung2 );

      switch ( i )
      {
         case 0 :
            cout << "Diese Gleichung hat keine reellen Loesungen!";
            break;
         case 1 :
            cout << "Diese Gleichung hat die Loesung: x = " << loesung1;
            break;
         default :
            cout << "Diese Gleichung hat die Loesungen x1 = " << loesung1
                 << " und x2 = " << loesung2 << ".";
      }

      cout << "\n\nESC-Taste: Programmabbruch";
      cout << "\nUebrige Tasten: Neue Berechnung";
   } while ( cgetch() != ESC );
   cend();
}

int berechneLoesungen ( double a, double b, double c, double & x1, double & x2 )
{
   double diskr = b * b - 4 * a * c;    // Definition und Initialisierung
                                        // der Diskriminante
   if ( diskr < 0)
      return 0;                         // Die Gleichung hat keine Loesung,
                                        // also wird der Wert 0 zurueckgegeben.
   if ( diskr == 0 )
```

```
   {                                    // Die Gleichung hat eine Loesung, also
      x1 = ( -b ) / (2 * a);            // wird nur x1 berechnet. Zurueckgegeben
      return 1;                         // wird der Wert 1.
   }

   x1 = ( -b + sqrt( diskr ) ) / ( 2 * a );
   x2 = ( -b - sqrt( diskr ) ) / ( 2 * a );
   return 2;                            // Die Gleichung hat zwei Loesungen, also
}                                       // wird der Wert 2 zurueckgegeben.
```

Soll ein Parameter als Referenzparameter übergeben werden, so ist es nötig, bei der Deklaration zwischen Datentyp und Namen des Parameters ein kaufmännisches "Und" (&; engl. "ampersand") zu schreiben. In unserem Programm sind also `x1` und `x2` Referenzparameter (`..., double & x1, ...`).

Der Unterschied zwischen der Übergabe eines Werteparameters (engl. "Call by value") und derjenigen eines Referenzparameters (engl. "Call by reference") ist der folgende:

Wird ein Werteparameter übergeben, wird dessen Wert auf einen eigens dafür reservierten Speicherplatz im sogenannten **Stack** kopiert. In unserem Programm trifft das für die Werte der Parameter `a`, `b` und `c` zu, im Speicher ist also z.B. der Wert von `a` an zwei verschiedenen Adressen gespeichert, einmal für die im Hauptprogramm lokale Variable `a`, einmal für die in der Funktion `berechneLoesungen` lokale Variable, die ebenfalls `a` heisst (vgl. Seite 31).

Ein Referenzparameter greift auf dieselbe Speicheradresse zu wie die übergebene Variable. In unserem Programm belegen z.B. die (lokalen) Variablen `x1` und `loesung1` dieselbe Speicheradresse. `x1` und `loesung1` sind also verschiedene Namen für dieselbe Variable. Das hat zur Folge, dass die Änderung von `x1` in der Funktion `berechneLoesungen` automatisch eine Änderung der Variablen `loesung1` im Hauptprogramm zur Folge hat. Natürlich ist es möglich, die Variablen in der beiden Funktionen gleich zu nennen (`loesung1` kann also auch `x1` heissen, vgl. Programm `quadrgl2.cpp` im Kapitel 10.3).

Denken Sie daran, dass Arrays immer als Referenzparameter übergeben werden (vgl. Seite 61). Deshalb ist in diesem Fall in der Deklaration kein kaufmännisches "Und" nötig.

Weitere Bemerkungen zum Programm:

- Mit dem Befehl `clrscr();` wird der Inhalt des Konsolenfensters gelöscht. Damit der Compiler den Befehl erkennt, wird zu Beginn des Programms die Datei `conio.h` mit `#include <conio.h>` eingebunden (vgl. Anhang Anhang D:).
- Das Programm soll mit der <Esc>-Taste beendet werden. Deshalb wird die Konstante `ESC` vom Typ *char* definiert, die den ASCII-Code der <Esc>-Taste (27) enthält. Diese Konstante dient zur besseren Transparenz des Programms, denn es ist natürlich möglich, die Laufbedingung der *do while*-Schleife wie folgt zu schreiben: `while ( cgetch() != 27 );`. Eine elegantere Variante wird im Kapitel 10.3 vorgestellt.
- In unserem Beispiel gibt die Funktion `berechneLoesungen` die Anzahl der Lösungen, also einen der drei Werte 0, 1 oder 2 (vom Typ *int*), an die aufrufende Funktion zurück. Dieser Wert wird wegen `i = berechneLoesungen( ... )` der Variablen `i` zugeordnet. Das erlaubt uns dann, je nach Anzahl der Lösungen in der *switch*-Anweisung verschiedene Texte auszugeben.
- Beachten Sie in der *switch*-Anweisung, wie eine Ausgabe mit einem `cout` über mehrere Programmzeilen verteilt werden kann.

10.3 Die "Messagebox"

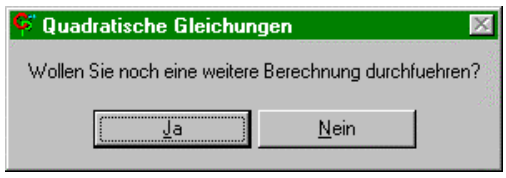
Das letzte Programm wurde mit einer *do while*-Schleife geschrieben, damit mehrere quadratische Gleichungen hintereinander gelöst werden können, ohne jedesmal das Programm neu starten zu müssen. Das Programm wird mit der <Esc>-Taste abgebrochen. Dieser Lösungsansatz ist aber für ein Programm, das unter Windows laufen soll, veraltet. Zeitgemässer ist eine Abfrage mit einer Dialogbox, die mit dem Befehl `CP::msgBoxD()` aktiviert werden kann. Diese Dialogbox nennen wir auch **Messagebox**, weil sie oft Mitteilungen enthalten.

Beispiel: Das Programm `quadrgl1.cpp` wird so abgeändert, dass die Abfrage, ob noch eine Berechnung durchgeführt werden soll, mittels einer Dialogbox beantwortet werden kann.

```
// QUADRGL2.CPP
// Berechnet die Loesungen einer quadratischen Gleichung in der Grundmenge R.
// Beispiel fuer CP::msgBoxD

#include <conio.h>                          // Noetig fuer den Befehl clrscr();
```

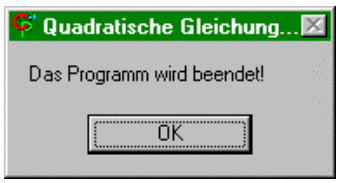
```
#include <champ.h>
#include "quadrgl2.rh"

int berechneLoesungen ( double a, double b, double c, double & x1, double & x2 );

void gmain ()
{
   double a, b, c;                              // Koeffizienten
   double x1, x2;                               // Loesungen
   int i;                                       // vgl. Programm quadrgl1.cpp
   int antwort;                                 // Variable, in der die Antwort
                                                // der Dialogbox gespeichert wird.
   char titel[] = "Quadratische Gleichungen";   // Titel der verschiedenen Fenster.

   cinit( titel );
   cout << "Dieses Programm loest die quadratische Gleichung ax^2 + bx + c = 0.";

   do
   {
      clrscr();           // Loescht den Inhalt des Konsolenfensters

      CPInputDouble( titel, "Koeffizient a:", a, 1 ).showModal();
      CPInputDouble( titel, "Koeffizient b:", b, 3 ).showModal();
      CPInputDouble( titel, "Koeffizient c:", c, 2 ).showModal();

      cout << "Sie haben die Gleichung " << a << "x^2 + " << b << "x + "
           << c << " = 0 eingegeben.\n\n";

      i = berechneLoesungen( a, b, c, x1, x2 );

      switch ( i )
      {
         case 0  :
            cout << "Diese Gleichung hat keine reellen Loesungen!";
            break;
         case 1  :
            cout << "Diese Gleichung hat die Loesung: x = " << x1;
            break;
         default :
            cout << "Diese Gleichung hat die Loesungen x1 = " << x1
                 << " und x2 = " << x2 << ".";
      }
      CP::msgBoxD( titel, MB_YESNO, antwort ) << "Wollen Sie noch eine weitere "
                                              << "Berechnung durchfuehren?";
   } while ( antwort == IDYES );
   CP::msgBoxD( titel ) << "Das Programm wird beendet!";
   cend();
}
int berechneLoesungen ( double a, double b, double c, double & x1, double & x2 )
{
   double diskr = b * b - 4 * a * c;    // Definition und Initialisierung
                                        // der Diskriminante
   if ( diskr < 0)
      return 0;                         // Die Gleichung hat keine Loesung,
                                        // also wird der Wert 0 zurueckgegeben.
   if ( diskr == 0 )
   {                                    // Die Gleichung hat eine Loesung, also
      x1 = ( -b ) / (2 * a);            // wird nur x1 berechnet. Zurueckgegeben
      return 1;                         // wird der Wert 1.
   }

   x1 = ( -b + sqrt( diskr ) ) / ( 2 * a );
   x2 = ( -b - sqrt( diskr ) ) / ( 2 * a );
   return 2;                            // Die Gleichung hat zwei Loesungen, also
}                                       // wird der Wert 2 zurueckgegeben.
```

Nachdem das Programm gestartet und die erste Berechnung durchgeführt worden ist, erscheint die Messagebox rechts.



Mit einem Mausklick auf Ja wird das Programm wiederholt, mit einem Mausklick auf Nein wird das Programm beendet, wobei das durch eine weitere Messagebox bestätigt wird:

Bemerkungen:



- Alle Fenster und Dialogboxen, die während der Laufzeit des Programms auf dem Bildschirm erscheinen, sollen in der Titelleiste alle denselben Titel haben. Damit im Quellcode nicht sechsmal "Quadratische Gleichungen" geschrieben werden muss (was vor allem dann lästig ist, wenn der Titel geändert werden soll), wird ein String `titel` definiert, dessen Inhalt eben "Quadratische Gleichungen" ist (vgl. Kapitel 8.5).
- Die Messagebox erscheint nach dem Befehl

```
CP::msgBoxD( titel, MB_YESNO, antwort ) << "Wollen Sie noch eine weitere "
                                        << "Berechnung durchfuehren?";
```

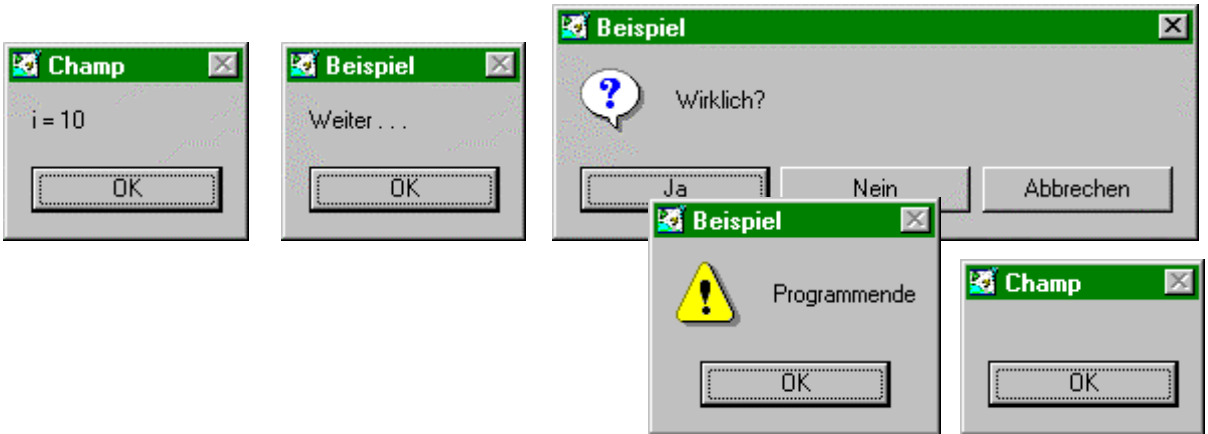
- Bis jetzt haben wir immer nur den Befehl `CP::msgBoxD();` ohne Angabe eines Parameters gebraucht. Hier sind nun aber drei Parameter nötig: Der erste Parameter muss ein String sein (kein String-Objekt!), der den Titel der Messagebox beinhaltet. Der zweite Parameter gibt an, welche (optischen) Eigenschaften die Messagebox haben soll. Mit `MB_YESNO` geben wir an, dass die Messagebox zwei Schaltflächen mit den Inhalten Ja bzw. Nein haben soll. Der dritte Parameter gibt an, welcher Variablen die Antwort zugeordnet werden soll. Wichtig ist, dass dies Variable vom Datentyp *Integer* sein muss. In unserem Fall hat die Variable den Namen `antwort`. Klickt man in der Messagebox auf Ja, so wird der Variablen `antwort` ein Wert zugeordnet, der durch die Konstante `IDYES` gegeben ist, bei Nein wird `antwort` ein Wert zugeordnet, der durch die Konstante `IDNO` gegeben ist.
- Die *do while*-Schleife wird dann wiederholt, wenn der Wert der Variablen `antwort` mit dem Wert der Konstanten `IDYES` übereinstimmt.
- Zur Ausgabe des Textes, der in die Messagebox geschrieben werden soll, wird genauso wie bei `cout` der Operator << verwendet. Auch hier kann wie bei `cout` die Ausgabe über mehrere Zeilen verteilt werden.

Das folgende Beispielprogramm zeigt weitere Möglichkeiten, wie Aussehen und Inhalt der Messagebox beeinflusst werden kann.

Beispiel:

```
// MSGBOX.CPP
// Beispiele zur Verwendung der Messagebox.

#include <champ.h>
#include "msgbox.rh"

void gmain ()
{
   int i = 10;
   int antwort;
   char titel[] = "Beispiel";

   cinit( "Champ" );
   CP::msgBoxD() << "i = " << i;
   CP::msgBoxD( "Beispiel" ) <<  "Weiter . . .";
   CP::msgBoxD( titel, MB_YESNOCANCEL | MB_ICONQUESTION, antwort ) << "Wirklich?";
   CP::msgBoxD( "Beispiel" , MB_ICONEXCLAMATION ) << "Programmende";
   CP::msgBoxD();
   cend();
}
```

Wird das Programm gestartet, so erscheinen der Reihe nach folgende Dialogboxen:



Bemerkungen:

- In der ersten Box wird der Inhalt der Variablen `i` angegeben.
- Die zweite Box erhält einen eigenen Titel.
- Die dritte Box erhält einen eigenen Titel, die drei Schaltflächen Ja, Nein und Abbrechen (wegen `MB_YESNOCANCEL`) und das Symbol mit dem Fragezeichen (wegen `MB_ICONQUESTION`). Diese zwei Konstanten werden mit einem Bitweisen ODER (`|`) verknüpft.
- Die vierte Box hat wiederum einen eigenen Titel und zusätzlich ein Ausrufezeichen, das häufig bei Warnmeldungen gebraucht wird.
- Die letzte Box ist die Box, die in den meisten bisherigen Programmen erschienen ist.

In der Champ-Hilfe werden weitere Möglichkeiten gezeigt, wie die Messagebox verwendet und gestaltet werden kann.

10.4 Ausgabe von Daten in eine Datei

Manchmal ist es wünschenswert, einige berechnete Werte (Daten) in einer anderen Anwendung weiter zu bearbeiten. Diese Daten können Messwerte sein, die dem Computer eingegeben werden und die z.B. in einer EXCEL-Tabelle verarbeitet werden.

Das folgende Beispiel zeigt, wie berechnete Daten in eine Datei (engl. File) statt auf den Bildschirm geschrieben werden.

Beispiel: Die Wurfweiten bei verschiedenen Abwurfwinkeln und bei gegebener Anfangsgeschwindigkeit sollen in die Datei `d:\scratch\schwurf.dat` geschrieben werden.

```
// FILE.CPP
// Ausgabe von Daten in einem File.

#include <champ.h>
#include <fstream.h>
#include "file.rh"

const double g = 9.807;                 // Fallbeschleunigung (Normwert)

void gmain ()
{
   ofstream ausgabe( "d:\\scratch\\schwurf.dat" );
   double v0;                          // Abwurfgeschwindigkeit
   double weite;

   cinit( "Schiefer Wurf" );
   cout << "Dieses Programm berechnet die Wurfweite in Abhaengigkeit des\n"
        << "Abwurfwinkels und der Anfangsgeschwindigkeit. Die Daten\n"
        << "werden in die Datei ""d:\\scratch\\schwurf.dat"" geschrieben.";
   cout << "\n\nGeben Sie die Anfangsgeschwindigkeit im m/s ein: ";
   cin >> v0;

   ausgabe << "Wurfweiten bei einer Anfangsgeschwindigkeit von " << v0 << " m/s."
           << "\n\nAbwurfwinkel\tWeite: \n\n";

   ausgabe << setprecision( 3 );    // Angabe auf drei wesentliche Stellen genau.
   for ( int winkel = 0; winkel <= 90; winkel++ )
   {
      weite = pow ( v0, 2 ) * sin ( 2 * M_PI * winkel / 180.0 ) / g;
      ausgabe << winkel << "\t" << weite << " m \n";
   }

   CP::msgBoxD() << "Die Berechnung ist erfolgreich abgeschlossen worden.";
   cend();
}
```

Bemerkungen:

- Sollen ein Text oder andere Daten in ein File geschrieben werden, so muss ein Objekt der Klasse `ofstream` erzeugt werden. In unserem Programm hat dieses Objekt den Namen `ausgabe`. Bei der Instanziierung dieses Objekts kann in den Klammern der Pfad der Datei als String eingegeben werden. In unserem Fall ist es der

Pfad `d:\scratch\schwurf.dat`. Kennt Ihr Computer kein Laufwerk `D:`, so ist es nötig, dass Sie, um das Programm laufen lassen zu können, in den Klammern einen anderen Pfad angeben.

- Weil ein Backslash in einem String ein Steuerzeichen anzeigt (wie `\n` oder `\t` für "Neue Zeile" bzw. "Gehe zum nächsten Tabulator"), wird das Zeichen `\` nicht als solches erkannt. Deshalb muss an der Stelle, an der ein String einen Backslash enthalten soll, das Steuerzeichen für einen Backslash (`\\`) eingegeben werden. Analog muss anstelle eines (einfachen) Anführungszeichens dessen Steuerzeichen (`\'`) eingegeben werden. Eine Liste der Steuerzeichen befindet sich auf der Seite 39. Die Ausgabe des Streams `... in die Datei \'d:\\scratch\\schwurf.dat\' geschrieben ...` lautet im Konsolenfenster: `... in die Datei 'd:\scratch\schwurf.dat' geschrieben` (vgl. Seite 39).
- Das Schreiben von Text in die Datei geschieht mit dem Operator `<<`, der dieselbe Bedeutung hat wie beim Streamobjekt `cout`: Mit `ausgabe <<` werden die Daten in die Datei geschrieben.
- Mit Datei|Öffnen kann der Inhalt der erzeugten Datei betrachtet werden.

Die zweitletzte Bemerkung rückt auch die längst bekannte Bildschirmausgabe mit `cout` in ein neues Licht. Der Stream `cout` ist ein Objekt der Klasse `ostream`, das mit dem Bildschirm "verknüpft" wird, so wie in unserem Programm die Datei `d:\scratch\schwurf.dat` mit dem Objekt `ausgabe` verknüpft wird. Auch `cin` ist ein Objekt (der Klasse `istream`), das mit der Tastatur verknüpft wird. Weil `cout` und `cin` so häufig verwendet werden, werden sie bereits in der eingebundenen Datei `champ.h` instanziiert.

10.5 Selbstdefinierte Datentypen

Will man die Koordinaten eines Punktes in einer Variablen speichern, so verwendet man dazu normalerweise ein Array mit zwei Feldern (vgl. Seite 61). Arbeitet man aber in einem Programm mit mehreren Punkten, so empfiehlt es sich, einen neuen, eigenen Datentyp zu definieren. Dies geschieht mit dem Befehl `typedef`.

Beispiel: Das folgende Programm berechnet den Mittelpunkt zweier gegebener Punkte: Verwendet wird ein selbstdefinierter Datentyp `punkt`.

```
// TYPEDEF.CPP
// Berechnet den Mittelpunkt M zweier Punkte P und Q, gerechnet wird mit Hilfe
// eines neuen Datentyps.

#include <champ.h>
#include "typedef.rh"

typedef double punkt[ 2 ];      // Definition des neuen Datentyps punkt.

void mittelpunkt ( punkt p, punkt q, punkt m );

void gmain ()
{
   punkt p = { 1, 3 };          // Erste Moeglichkeit, Koordinaten zu definieren.
   punkt q;
   punkt m;

   cinit( "Champ" );
   q[ 0 ] = 2;                  // Zweite Moeglichkeit, Koordinaten zu definieren.
   q[ 1 ] = 5;

   mittelpunkt( p, q , m );

   cout << "Koordinaten des Mittelpunkts:\n";
   cout << "x-Koordinate: " << m[ 0 ] << "\n";
   cout << "y-Koordinate: " << m[ 1 ] << "\n";

   CP::msgBoxD() << "Programmende";
```

```
   cend();
}

void mittelpunkt ( punkt p, punkt q, punkt m )
{
   m[ 0 ] = ( p[ 0 ] + q[ 0 ] ) / 2;
   m[ 1 ] = ( p[ 1 ] + q[ 1 ] ) / 2;
}
```

Ausgabe auf dem Bildschirm:

```
Koordinaten des Mittelpunkts:
x-Koordinate: 1.5
y-Koordinate: 4
```

Bemerkungen:

- In der Zeile `typedef double punkt[ 2 ];` wird ein neuer Datentyp `punkt` definiert: Die Definition gleicht derjenigen eines Arrays mit zwei Feldern mit dem Namen `punkt`. Erst durch das reservierte Wort `typedef` macht aus einer Definition einer (globalen) Variablen eine Definition eines neuen Datentyps.
- Jede Variable vom Typ `punkt` ist ein Array mit zwei Feldern, die je einen Wert vom Typ *double* enthalten. Deshalb ist es möglich, die Variablen `p` und `q` wie ein Array zu initialisieren (vgl. Kapitel 8.3).
- Auch Variablen von selbstdefinierten Datentypen können als Parameter an Funktionen übergeben werden. Weil sich in unserem Programm die Variablen `p`, `q` und `m` wie Arrays verhalten, werden sie als Referenzparameter übergeben (vgl. Kapitel 8.3 und 10.2).

10.6 Öffnen von mehreren Graphikfenstern

Im Programm `eingabe1.cpp` auf Seite 38 arbeiteten wir mit mehreren Fenstern, nämlich dem Graphikfenster und dem Konsolenfenster. Champ ermöglicht uns aber mit der Klasse `CPWindow`, auch mehrere Graphikfenster zu öffnen, z.B. um Graphen mehrerer Funktionen zu zeichnen:

Beispiel: Das folgende Programm zeichnet die Graphen der Sinus- und der Cosinusfunktion in zwei verschiedene Graphikfenster.

```
// SINCOSIN.CPP
// Zeichnet die Graphen der Sinus- und der Cosinusfunktion
// in zwei verschiedene Graphikfenster.

#include <champ.h>
#include "sincosin.rh"

const double XMIN = -0.1;
const double XMAX = 6.5;
const double YMIN = -1.1;
const double YMAX = 1.1;

void gmain ()
{
   double dx;                    // Schrittweite in for-Schleife

   CPWindow sinus( "Sinusfunktion", CP::WinBigSq );      // Instanziiert zwei
   CPWindow cosinus( "Cosinusfunktion", CP::WinBigSq );  // grosse Graphikfenster
   dx = ( XMAX - XMIN ) / sinus.width();

   sinus.windowEven( XMIN, XMAX, YMIN, YMAX );        // Setzt das Koordinatensy-
   cosinus.windowEven( XMIN, XMAX, YMIN, YMAX );      // stem in beiden Fenstern

   sinus.line( XMIN, 0, XMAX, 0 );                    // Koordinatensystem im
   sinus.line( 0, YMIN, 0, YMAX );                    // ersten Fenster
   cosinus.line( XMIN, 0, XMAX, 0 );                  // Koordinatensystem im
   cosinus.line( 0, YMIN, 0, YMAX );                  // zweiten Fenster
   for ( int ix = ceil( XMIN ); ix <= XMAX; ix++ )
   {
      sinus.line( ix, -0.05, ix, 0.05 );              // Beschriften der
      sinus.text( ix, -0.2 ) << ix;                   // x-Achsen
      cosinus.line( ix, -0.05, ix, 0.05 );
      cosinus.text( ix, -0.2 ) << ix;
   }
```

```
   for ( int iy = ceil( YMIN ); iy <= YMAX; iy++ )
   {
      sinus.line( -0.05, iy, 0.05, iy );              // Beschriften der
      cosinus.line( -0.05, iy, 0.05, iy );            // y-Achsen
      if ( iy != 0 )
      {
         sinus.text( 0.05, iy ) << iy;
         cosinus.text( 0.05, iy ) << iy;
      }
   }
   sinus.pos( XMIN, sin( XMIN ) );                    // Positioniert je einen
   cosinus.pos( XMIN, cos( XMIN ) );                  // Zeichenstift in beiden
                                                      // Fenstern.
   for ( double x = XMIN; x <= XMAX; x = x + dx )
   {
      sinus.draw( x, sin( x ) );
      cosinus.draw( x, cos( x ) );
   }


   sinus.activate();                                  // Aktiviert das Fenster
   CP::msgBoxD();                                     // 'sinus'
   sinus.close();                                     // Schliesst das Fenster
   cosinus.activate();                                // (Objekt) 'sinus'
   CP::msgBoxD();
}
```

Bemerkungen:

- Mit `CPWindow sinus( "Sinusfunktion", CP::WinBigSq );` wird ein Objekt `sinus` der Klasse `CPWindow` instanziiert, d.h. `sinus` ist ein Graphikfenster mit dem Titel "Sinusfunktion" und mit maximal möglicher Grösse eines quadratischen Fensters. Beachten Sie, dass die Parameter dieselben sind wie beim Befehl `ginit`, das bekanntlich ein (und nur ein) Graphikfenster erzeugt. Analog wird ein zweites Objekt (`cosinus`) erzeugt.
- In beiden Fenstern können jetzt (fast) wie gewohnt Graphiken gezeichnet werden, alle Befehle die uns im "normalen" Graphikfenster zur Verfügung stehen, sind in der Klasse `CPWindow` als **Methoden** bekannt. Allerdings gibt es einen kleinen, aber sehr wichtigen Unterschied: Bei der Einführung (Seite 69) in die Champ-Hilfe wurde erwähnt, dass allen Graphikbefehlen ein `g` vorangestellt werden soll. Dies ist aber nur richtig, wenn in ein mit `ginit` erzeugtes Graphikfenster gezeichnet werden soll. Wird aber in ein mit `CPWindow` "erzeugtes" Fenster gezeichnet, **muss** das vorangestellte `g` weg gelassen werden.
- Ein erstes Beispiel einer Methode der Klasse `CPWindow` ist die Methode `windowEven`. Die Wirkung ist dieselbe wie bei `gwindowEven` in einem "normalen" Graphikfenster, der Befehl kann aber nicht alleine für sich stehen, weil es eben eine Methode ist und als solche durch einen Punkt an das Objekt angefügt wird, dessen Koordinaten neu gesetzt werden soll: `sinus.windowEven( XMIN, XMAX, YMIN, YMAX );` setzt also die Koordinaten im Fenster des Objekts.
- Mit `sinus.activate();` wird das Fenster `sinus` aktiviert, d.h. in den Vordergrund gestellt.
- Das Fenster `sinus` wird mit `sinus.close();` geschlossen; das Fenster `cosinus` wird am Ende des Programms automatisch geschlossen, weil die Lebensdauer des Objekts genau definiert ist: Ein Objekt ist nur in dem Anweisungsblock bekannt, in dem es instanziiert wurde, verlässt das Programm den Anweisungsblock, so wird das Objekt **vernichtet**; in unserem Fall wird also das Fenster geschlossen.

Das Programm (`sincosin.cpp`) hat noch einen Schönheitsfehler: Da in beiden Fenstern ein Koordinatensystem gezeichnet werden soll, werden in diesem Programm alle nötigen Methoden zweimal aufgerufen; einmal für das Objekt `sinus`, das zweite Mal für das Objekt `cosinus`. Eine Möglichkeit, dieses Manko zu beheben, ist die Definition einer Funktion `zeichneKoordinatensystem`, der ein Objekt der Klasse `CPWindow` als Referenzparameter übergeben wird. Die Deklaration dieser Funktion könnte etwa so aussehen:

```
void zeichneKoordinatensystem( CPWindow & fenster );
```

In der **objektorientierten Programmierung** wird das Problem aber anders gelöst: Man definiert eine neue Klasse, die eine Methode enthalten soll, die das Koordinatensystem zeichnen lässt. Dieses Verfahren soll im folgenden Kapitel näher erläutert werden.

11 Ausblick: Objektorientierte Programmierung

In diesem Kapitel soll anhand des letzten Programms (`sincosin.cpp`) exemplarisch gezeigt werden, wie man eine eigene Klasse definieren kann. Dazu müssen aber zuerst einige Begriffe erläutert werden:

11.1 Klassen, Objekte

Im Zentrum der objektorientierten Programmierung stehen die **Objekte**. Ein Objekt kann man definieren als Verbund von Variablen (Attribute) und dazugehörigen Methoden. Wir kennen bereits String-Objekte, Streamobjekte (`cin`, `cout`), Objekte der Klasse `CPWindow`. Jedes Objekt hat gewisse Eigenschaften und kennt gewisse Verhalten. Die Eigenschaften werden durch **Attribute** oder **Variablen** (engl. **Data member**), die verschiedenen Verhalten durch **Methoden** (engl. **Member function**) beschrieben. Eine **Klasse** ist eine Beschreibung von gleichartigen Objekten, ein bestimmtes Element einer Klasse nennt auch eine **Instanz** oder **Inkarnation** der Klasse.

Beispiele:

Klasse:	**Instanz:**	**mögliche Attribute:**	**mögliche Methoden**
Katzen	Murrli	Fellfarbe Geschlecht Grösse	"komm" "friss"
Autos	Das Auto von Fritz	Lackfarbe Anzahl der Gänge Anzahl der Türen momentaner Gang momentane Fahrgeschwindigkeit	beschleunigen bremsen schalten Fenster öffnen
Klasse `string` (Kap 1)	`name`	Länge des String-Objekts (`length`) Inhalt des String-Objekts (z.B. "Fritz")	Eingabe (mit `getline`) Erzeugen eines Teilstrings (mit `substr`)

Klasse `CPWindow` (Kap 10.5)	`sinus`	Grösse des Graphikfensters
		Position auf dem Bildschirm
		Titel des Graphikfensters
			`pos`
			`draw`
			`line`

Die Attribute werden normalerweise vom Zugriff von der Aussenwelt geschützt, eine Änderung kann nur über den Aufruf einer passenden Methode erfolgen. Dieses Konzept des Schutzes der Attribute nennt man **Kapselung** (engl. **Encapsulation**).

11.2 Vererbung

Wird eine Klasse von einer anderen **abgeleitet**, so spricht man von **Vererbung** (engl. **Inheritance**), die Eigenschaften der ursprünglichen Klasse (der **Basisklasse**) werden auf die abgeleitete Klasse vererbt. Von der Klasse "Katzen" kann man z.B. die Klasse "Löwen" ableiten: ein Löwe hat dieselben Attribute und Methoden wie eine Katze, kann aber noch zusätzliche Eigenschaften haben. Ebenso kann man von der Klasse "Autos" z.B. eine Klasse "Lastwagen" ableiten.

Beispiel: Das Programm `sincosin.cpp` wird wie folgt abgeändert: Definiert wird eine Klasse `Fenster`, die von der Klasse `CPWindow` abgeleitet wird. Als neue Methode soll die Methode `zeichneKoordinatensystem` definiert werden.

```
// KLASSEN1.CPP
// Erzeugt wird eine Klasse Fenster, abgeleitet von der Klasse CPWindow.

#include <champ.h>
#include "klassen1.rh"


const double XMIN = -0.1;
const double XMAX = 6.5;
const double YMIN = -1.1;
```

```
const double YMAX = 1.1;
                                       // Definition der Klasse Fenster, ein
class Fenster : public CPWindow        // Objekt dieser Klasse soll dieselben
{                                      // Eigenschaften haben wie ein Objekt der
   public :                            // Klasse CPWindow (Vererbung). Die
      void zeichneKoordinatensystem (); // Methode zeichneKoordinatensystem wird
};                                     // zusaetzlich definiert.
                                       // Semikolon nicht vergessen!!!

void Fenster::zeichneKoordinatensystem ()
{
   line( XMIN, 0, XMAX, 0 );           // Die Methode 'zeichneKoordinatensystem'
   line( 0, YMIN, 0, YMAX );           // wird definiert.
   for ( int x = ceil( XMIN ); x <= XMAX; x++ )
   {
      line( x, -0.05, x, 0.05 );
      text( x, -0.2 ) << x;
   }
   for ( int y = ceil( YMIN ); y <= YMAX; y++ )
   {
      line( -0.05, y, 0.05, y );
      if ( y != 0 )
         text( 0.05, y ) << y;
   }
}

void gmain ()
{
   double dx;

   Fenster sinus;                      // Instanziierung der Objekte sinus und
   Fenster cosinus;                    // cosinus der Klasse Fenster.
   dx = ( XMAX - XMIN ) / sinus.width();

   sinus.windowEven( XMIN, XMAX, YMIN, YMAX );
   cosinus.windowEven( XMIN, XMAX, YMIN, YMAX );

   sinus.zeichneKoordinatensystem();
   cosinus.zeichneKoordinatensystem();

   sinus.pos( XMIN, sin( XMIN ) );
   cosinus.pos( XMIN, cos( XMIN ) );

   for ( double x = XMIN; x <= XMAX; x = x + dx )
   {
      sinus.draw( x, sin( x ) );
      cosinus.draw( x, cos( x ) );
   }

   sinus.activate();
   CP::msgBoxD();
   sinus.close();
   cosinus.activate();
   CP::msgBoxD();
}
```

Bemerkungen:

- Die Klasse `Fenster` wird mit `class Fenster : public CPWindow` definiert, `public CPWindow` (hinter einem Doppelpunkt) bedeutet, dass die Klasse `Fenster` von der Klasse `CPWindow` abgeleitet wird, bzw. dass `CPWindow` eine Basisklasse der Klasse `Fenster` ist. Das bedeutet, dass alle Methoden der Klasse `CPWindow` auch für Objekte der Klasse `Fenster` zur Verfügung stehen.
- Die neu zu definierenden Methoden und Attribute werden zwischen den geschweiften Klammern der Klassendefinition (ähnlich wie Variablen und Funktionsdeklarationen) definiert bzw. deklariert. Dabei wird zwischen den "öffentlich zugänglichen" (**public**; d.h. aus einer Funktion aufrufbaren) und "privaten" (**private)** Methoden und Attributen unterschieden. In unserem Beispiel wird nur die (öffentliche) Methode `zeichneKoordinatensystem`. deklariert.
- Nach der Klassendefinition muss ein Semikolon stehen!
- Der Kopf der Definition einer Methode sieht wie folgt aus:

NameDerKlasse::*NameDerMethode* (*Parameterliste*)

In unserem Fall lautet der Kopf also `void Fenster::zeichneKoordinatensystem ()`, die zwei Doppelpunkte deuten an, dass `zeichneKoordinatensystem` eine Methode der Klasse `Fenster` ist. Da keine Parameter übergeben werden, stehen (wie bei Funktionsdefinitionen) am Ende des Kopfes leere Klammern. Die Definition der Methoden erfolgen ausserhalb der Definition der Klassen!

- Zur Erinnerung: Soll eine Methode auf ein bestimmtes Objekt angewendet werden, so erfolgt ihr Aufruf, durch einen Punkt abgetrennt, nach dem Objektnamen (z.B. `sinus.zeichneKoordinatensystem();`).
- In der Methode `zeichneKoordinatensystem` werden andere (abgeleitete) Methoden aus der Basisklasse `CPWindow` aufgerufen. Der Aufruf dieser Methoden erfolgt ohne Angabe eines Objekts, da sie sich nur auf ein mögliches Objekt beziehen kann: Beim Aufruf durch `sinus.zeichneKoordinatensystem();` beziehen sie sich auf das Objekt `sinus`, beim Aufruf durch `cosinus.zeichneKoordinatensystem();` auf das Objekt `cosinus`.
- Das Objekt `sinus` der Klasse `Fenster` werden mit `Fenster sinus;` instanziiert.

11.3 Der Konstruktor

Im Programm `klassen1.cpp` ist es nicht möglich, den erzeugten Graphikfenstern einen eigenen Titel (wie im Programm `sincosin.cpp`) zu geben. Um dies zu ermöglichen, müssen wir einen eigenen **Konstruktor** der Klasse `Fenster` definieren. Ein Konstruktor (engl. **Constructor**) einer Klasse ist eine Methode dieser Klasse, die bei der Instanziierung eines Objekts automatisch aufgerufen wird.

Wichtig: Der Name des Konstruktors entspricht dem Namen der Klasse.

Die Definition eines Konstruktors hat folgenden Kopf:

NameDerKlasse::*NameDerKlasse* (*Parameterliste*)

Beispiel: Ausgehend vom letzten Programm wird ein Konstruktor der Klasse `Fenster` definiert, dem ein String (Titel des Graphikfensters) als Parameter übergeben wird.

```
// KLASSEN2.CPP
// Erzeugt wird eine Klasse Fenster, abgeleitet von der Klasse CPWindow.
// Die Klasse erhaelt einen eigenen Konstruktor.

#include <champ.h>
#include "klassen2.rh"

const double XMIN = -0.1;
const double XMAX = 6.5;
const double YMIN = -1.1;
const double YMAX = 1.1;

class Fenster : public CPWindow
{
   public :
      Fenster ( char titel[] );          // Deklaration des Konstruktors
      void zeichneKoordinatensystem ();
};



Fenster::Fenster ( char titel[] ) : CPWindow( titel )
{                                  // Definition des konstruktors, initialisiert
}                                  // wird der Konstruktor der Klasse CPWindow.

void Fenster::zeichneKoordinatensystem ()
{
   line( XMIN, 0, XMAX, 0 );
   line( 0, YMIN, 0, YMAX );
   for ( int x = ceil( XMIN ); x <= XMAX; x++ )
   {
      line( x, -0.05, x, 0.05 );
      text( x, -0.2 ) << x;
   }
   for ( int y = ceil( YMIN ); y <= YMAX; y++ )
   {
      line( -0.05, y, 0.05, y );
```

```
      if ( y != 0 )
         text( 0.05, y ) << y;
   }
}

void gmain ()
{
   double dx;

   Fenster sinus( "Sinusfunktion" );      // Instanziierung der Objekte, an den
   Fenster cosinus( "Cosinusfunktion" );  // Konstruktor wird ein String ueber-
   dx = ( XMAX - XMIN ) / sinus.width();  // geben.

   sinus.windowEven( XMIN, XMAX, YMIN, YMAX );
   cosinus.windowEven( XMIN, XMAX, YMIN, YMAX );

   sinus.zeichneKoordinatensystem();
   cosinus.zeichneKoordinatensystem();

   sinus.pos( XMIN, sin( XMIN ) );
   cosinus.pos( XMIN, cos( XMIN ) );

   for ( double x = XMIN; x <= XMAX; x = x + dx )
   {
      sinus.draw( x, sin( x ) );
      cosinus.draw( x, cos( x ) );
   }

   sinus.activate();
   CP::msgBoxD();
   sinus.close();
   cosinus.activate();
   CP::msgBoxD();
}
```

Bemerkungen:

- Der Konstruktor muss wie die anderen Methoden in der Definition der Klasse deklariert werden.
- Wird eine Klasse von anderen Klassen abgeleitet, so werden bei der Instanziierung eines Objekts die Konstruktoren der übergeordneten Klassen ebenfalls aufgerufen. In unserem Beispiel enthält die Definition des Konstruktors nur eine Angabe: Beim Aufruf des Konstruktors der Klasse `CPWindow` soll das erzeugte Objekt (das Graphikfenster) initialisiert werden, das Fenster erhält den gewünschten Titel. Werden Parameter an einen übergeordneten Konstruktor übergeben, so geschieht das im Kopf der Definition des (abgeleiteten) Konstruktors, durch einen Doppelpunkt abgetrennt.
- Da unser Konstruktor ausser der Initialisierung eines übergeordneten Konstruktors keine Befehle enthält, werden in seiner Definition nur die geschweiften Klammern gesetzt.

- Im speziellen Fall des Titels eines Graphikfensters gibt es noch eine Alternative für unseren Konstruktor:

```
Fenster::Fenster ( char titel[] )
{
   setTitle( titel );                  // Setzt den Titel des Graphikfensters.
}
```

Die Programme `klassen1.cpp` und `klassen2.cpp` haben immer noch einen gravierenden Schönheitsfehler: die Methode `zeichneKoordinatensystem` greift auf die globalen Konstanten `XMIN`, `XMAX`, etc. zurück. Es ist besser, diese Konstanten entweder als Parameter direkt der Methode zu übergeben (mit dem entsprechenden Aufruf `sinus.zeichenKoordinatensystem( XMIN, XMAX, YMIN, YMAX );`) oder sie im Konstruktor **privaten Attributen** der Klasse `Fenster` zuzuordnen. Privat bedeutet, dass diese Attribute nur von Methoden der Klasse, denen sie angehören, gelesen und verändert werden können. (Auch in diesem Fall werden die Konstanten `XMIN`, `XMAX`, etc. als Parameter übergeben, jedoch als Parameter des Konstruktors.)

Beispiel: Die Klasse Fenster erhält private Datenelemente (Attribute), die im Konstruktor initialisiert werden.

```
// KLASSEN3.CPP
// Erzeugt wird eine Klasse Fenster, abgeleitet von der Klasse CPWindow.
// Die Klasse erhaelt einen eigenen Konstruktor und private Variablen (Attribute).
```

```
#include <champ.h>
#include "klassen3.rh"

const double XMIN = -0.1;
const double XMAX = 6.5;
const double YMIN = -1.1;
const double YMAX = 1.1;

class Fenster : public CPWindow
{
   public :
      Fenster ( char titel[],
                double xmin,
                double xmax,
                double ymin,
                double ymax );

      void zeichneKoordinatensystem ();

   private :
      double _xmin, _xmax, _ymin, _ymax; // Private Variablen (Attribute) der
};                                       // Klasse Fenster

Fenster::Fenster ( char titel[],
                   double xmin,
                   double xmax,
                   double ymin,
                   double ymax  ) : CPWindow( titel )
{
   _xmin = xmin;                         // Initialisierung der Attribute
   _xmax = xmax;
   _ymin = ymin;
   _ymax = ymax;
   windowEven( _xmin, _xmax, _ymin, _ymax );
}                                        // Definition des Konstruktors

void Fenster::zeichneKoordinatensystem ()

{
   line( _xmin, 0, _xmax, 0 );
   line( 0, _ymin, 0, _ymax );



   for ( int x = ceil( _xmin ); x <= _xmax; x++ )
   {
      line( x, -0.05, x, 0.05 );
      text( x, -0.2 ) << x;
   }
   for ( int y = ceil( _ymin ); y <= _ymax; y++ )
   {
      line( -0.05, y, 0.05, y );
      if ( y != 0 )
         text( 0.05, y ) << y;
   }
}

void gmain ()
{
   double dx;

   Fenster sinus( "Sinusfunktion", XMIN, XMAX, YMIN, YMAX );
   Fenster cosinus( "Cosinusfunktion", XMIN, XMAX, YMIN, YMAX );
   dx = ( XMAX - XMIN ) / sinus.width();

   sinus.zeichneKoordinatensystem();
   cosinus.zeichneKoordinatensystem();

   sinus.pos( XMIN, sin( XMIN ) );
   cosinus.pos( XMIN, cos( XMIN ) );

   for ( double x = XMIN; x <= XMAX; x = x + dx )
   {
      sinus.draw( x, sin( x ) );
```

```
      cosinus.draw( x, cos( x ) );
   }

   sinus.activate();
   CP::msgBoxD();
   sinus.close();
   cosinus.activate();
   CP::msgBoxD();
}
```

Bemerkungen:

- Private Attribute werden in der Klassendefinition nach dem reservierten Wort `private` wie Variablen in einer Variablendefinition definiert.
- Sollen Parameter, die dem Konstruktor übergeben werden, den Attributen zugeordnet werden, so werden diese Attribute im allgemeinen gleich genannt wie die Parameter, aber mit einem vorne angefügten "underline" (_). Mit dem Befehl `_xmin = xmin;` wird der übergebene Parameter `xmin` dem Attribut `_xmin` zugeordnet. Dieses Verfahren ist zwar nicht zwingend, aber zweckmässig.
- Die Attribute `_xmin, _xmax` etc. sind der Methode `zeichneKoordinatensystem` bekannt und können von dieser verwendet werden.

11.4 Überladen

Wir haben bereits gesehen, dass bei der Initialisierung eines Graphikfensters mit `ginit` verschiedene Parameter angegeben werden können: Titel des Graphikfensters, Grösse des Graphikfensters, sogar die Position des Graphikfensters können in den Klammern hinter `ginit` angegeben werden (vgl. Seite 69 und Hilfe zu Champ). Die selben Parameter können bei einer Instanziierung eines Objekts der Klasse `CPWindow` angegeben werden. Will man diese verschiedenen Parameterlisten bei einer Instanziierung eines Objekts einer abgeleiteten Klasse (wie in unserem Beispiel die Klasse `Fenster`) angeben können, so muss man für jede Form der Liste, die man erlauben will, einen eigenen Konstruktor definieren. Mit dem Konzept des **Überladens** (engl. **Overloading**) ist es möglich, all diesen Konstruktoren denselben Namen zu geben. Anhand der Parameterliste erkennt der Compiler bei der Instanziierung eines Objekts, welcher Konstruktor aufgerufen werden muss.

Beispiel: Im folgenden Programm werden für die Klasse `Fenster` drei Konstruktoren definiert, d.h. der Konstruktor wird überladen.

```
// KLASSEN4.CPP
// Erzeugt wird eine Klasse Fenster, abgeleitet von der Klasse CPWindow.
// Der Konstruktor wird ueberladen!

#include <champ.h>
#include "klassen4.rh"

const double XMIN = -0.1;
const double XMAX = 6.5;
const double YMIN = -1.1;
const double YMAX = 1.1;

class Fenster : public CPWindow
{
   public :
      Fenster ( char titel[],             // Erste Version des Konstruktors
                double xmin,
                double xmax,
                double ymin,
                double ymax );
      Fenster ( char titel[],             // Zweite Version des Konstruktors,
                CP::CanvasType groesse,   // als zusaetzlicher Parameter kann
                double xmin,              // die Groesse des Graphikfensters
                double xmax,              // angegeben werden.
                double ymin,
                double ymax );
      void zeichneKoordinatensystem ();

   private :
      double _xmin, _xmax, _ymin, _ymax;
};

Fenster::Fenster ( char titel[],          // Definition des ersten Konstruktors
```

```
                    double xmin,
                    double xmax,
                    double ymin,
                    double ymax  ) : CPWindow( titel )
{
   _xmin = xmin;
   _xmax = xmax;
   _ymin = ymin;
   _ymax = ymax;
   windowEven( _xmin, _xmax, _ymin, _ymax );
}

Fenster::Fenster ( char titel[],          // Definition des zweiten Konstruktors
                    CP::CanvasType groesse,
                    double xmin,
                    double xmax,
                    double ymin,
                    double ymax  ) : CPWindow( titel, groesse )
{
   _xmin = xmin;
   _xmax = xmax;
   _ymin = ymin;
   _ymax = ymax;
   windowEven( _xmin, _xmax, _ymin, _ymax );
}



void Fenster::zeichneKoordinatensystem ()
{
   line( _xmin, 0, _xmax, 0 );
   line( 0, _ymin, 0, _ymax );
   for ( int x = ceil( _xmin ); x <= _xmax; x++ )
   {
      line( x, -0.05, x, 0.05 );
      text( x, -0.2 ) << x;
   }
   for ( int y = ceil( _ymin ); y <= _ymax; y++ )
   {
      line( -0.05, y, 0.05, y );
      if ( y != 0 )
         text( 0.05, y ) << y;
   }
}

void gmain ()
{
   double dx;

   Fenster sinus( "Sinusfunktion", XMIN, XMAX, YMIN, YMAX );
   Fenster cosinus( "Cosinusfunktion", CP::WinBigSq, XMIN, XMAX, YMIN, YMAX );
   dx = ( XMAX - XMIN ) / sinus.width();

   sinus.zeichneKoordinatensystem();
   cosinus.zeichneKoordinatensystem();

   sinus.pos( XMIN, sin( XMIN ) );
   cosinus.pos( XMIN, cos( XMIN ) );

   for ( double x = XMIN; x <= XMAX; x = x + dx )
   {
      sinus.draw( x, sin( x ) );
      cosinus.draw( x, cos( x ) );
   }

   sinus.activate();
   CP::msgBoxD();
   sinus.close();
   cosinus.activate();
   CP::msgBoxD();
}
```

Bemerkungen:

- Beachten Sie, dass beide Konstruktoren denselben Namen haben! Anhand der Parameterlisten erkennt der Compiler, dass bei der Instanziierung des Objekts `sinus` der erste und bei `cosinus` der zweite Konstruktor aufgerufen werden muss.
- Im ersten Konstruktor hat die Initialisierung des Konstruktors von `CPWindow` nur einen Parameter (Titel des Fensters), im zweiten Konstruktor aber zwei Parameter (Titel und Grösse). Das bedeutet, dass auch der Konstruktor von `CPWindow` (den wir nicht näher zu kennen brauchen) überladen ist.
- Der Parameter, mit dem die Grösse des Fensters angegeben wird, ist vom Typ `CP::CanvasType`, der im Champ definiert wird. Die Werte, die eine Variable von diesem Typ annehmen kann, sind in der Tabelle auf der Seite 69 angegeben.
- Es ist auch möglich, eine dritten Konstruktor zu schreiben, bei dem die Reihenfolge der Parameter vertauscht wird.
- Nebst Konstruktoren können auch andere Methoden oder sogar Operatoren (z.B. Additionszeichen) überladen werden.

Anhang A: Installation von C++ und Champ

Zur Installation von Champ ist auch die Installation von Borland C++, Version 4.5 oder später nötig. Eine CD-ROM mit dieser Version kann bei folgender Adresse bezogen werden:

SFIB/Schweizerische Fachstelle
für Informationstechnologien

Erlachstrasse 21
3000 Bern 9

Tel. 031 / 301 20 91, Fax 031 / 301 01 04

Die Schulversion kostet inkl. Verpackung und Versand ca. 160 Franken. Der Bestellung sollte eine Kopie des Schülerausweises oder (für Lehrkräfte) eine Bestätigung der Schule beigelegt werden.

Auf der CD-ROM befinden sich die Versionen 4.52 und 5.0 von Borland C++, jeweils in Deutsch und in Englisch (Stand 1996). Wollen Sie die Version 4.52 installieren, so müssen Sie wie folgt vorgehen:

1. Legen Sie die CD ein.
2. Öffnen Sie den Ordner `X:\setup\bcw452ge\` (`X:` ist die Bezeichnung des CD-Laufwerks, in den meisten Fällen steht statt `X:` die Bezeichnung `D:` oder `E:`.
3. Starten Sie (z.B. mit einem Doppelklick auf die Entsprechende Zeile im Fenster) die Anwendung `setup`.
4. Folgen Sie den Anweisungen auf dem Bildschirm.

Als Alternative zu Borland C++ kann im Fachhandel relativ günstig eine Version von Turbo C++ (ebenfalls von der Firma Borland) bezogen werden (Versionsnummer: 4.5 oder später).

Die Gymnasien Bern-Neufeld besitzen eine Schullizenz von Champ. Das bedeutet, dass die Schülerinnen und Schüler der Gymnasien kostenlos eine Version beziehen können, mit der sie zu Hause arbeiten dürfen.

Vorgehen:

1. Nehmen Sie zwei leere 3½-Zoll-Disketten mit.
2. Starten Sie an der Schule einen Computer und wechseln Sie in die MS-DOS-Eingabeaufforderung.
3. Geben Sie in der Eingabeaufforderung den Befehl `champ a:` ein.
4. Folgen Sie den weiteren Anweisungen des Computers.

Zur genaueren Installation von Champ lesen Sie bitte die Dateien `readme.txt` und `readnf.txt`. Achten Sie darauf, dass Sie bei der Installation der Heimversion den Namen und den Schlüsselcode (beide stehen in der Datei `readnf.txt`) exakt eingeben.

Anhang B: ASCII-Tabelle

Da der Computer bekanntlich einen Binärcode verwenden muss (vgl. Kapitel 1.1.2), müssen alle Zeichen (Buchstaben, Steuerzeichen) in einen Binärcode verwandelt werden. Der bei uns bekannteste Binärcode ist der **ASCII-Code** ASCII: American Standard Code for Information Interchange). Der ASCII-Code ist eigentlich ein 7-Bit Code, d.h. mit dem ASCII-Code können $2^7 = 128$ Zeichen kodiert werden. In einem Byte, nach dem Bit die zweit kleinste Speichereinheit im Computer (1 Byte = 8 Bit), können aber $2^8 = 256$ verschiedene Zeichen gespeichert werden. Deshalb wird das achte Bit meist zur Erweiterung des ASCII-Codes, manchmal aber auch als **Prüfbit** verwendet.

Der Code wird meistens nicht im Dualsystem, sondern im Dezimal- oder Hexadezimalsystem angegeben:

Beispiel: Zeichen 'M'; dezimal: 77; hexadezimal: 4D; dual: 0100'1101.

Die ersten 32 Zeichen sind Steuerzeichen, die wichtigsten sind mit grauer Farbe unterlegt:

Dez:	Hex:	Abk.:	Bedeutung:	Dez:	Hex:	Abk.:	Bedeutung:
0	00	NUL	Null-Character:	16	10	DLE	Datenübertragungsumschaltung (Data Link Escape)
1	01	SOH	Anfang des Kopfes (Start of Heading)	17	11	DC1	Gerätesteuerung (Device Control)
2	02	STX	Anfang des Textes (Start of Text)	18	12	DC2	
3	03	ETX	Ende des Textes (End of Text)	19	13	DC3	
4	04	EOT	Ende der Übertragung (End of Transmission)	20	14	DC4	
5	05	ENQ	Stationsaufforderung (Enquiry)	21	15	NAK	Negative Rückmeldung (Negative Acknowledge)
6	06	ACK	Positive Rückmeldung (Acknowledge)	22	16	SYN	Synchronisierung (Synchronous Idle)
7	07	BEL	Klingel (Bell)	23	17	ETB	Ende des Datenübertragungsblocks (End of Transmission Block)
8	08	BS	Rückwärtsschritt (Backspace)	24	18	CAN	Ungültig (Cancel)
9	09	HAT	Horizontaltabulator (Horizontal Tabulation)	25	19	EM	Ende der Übertragung (End of Medium)
10	0A	LF	Zeilenvorschub (Line Feed)	26	1A	SUB	Substitution (Substitute Character)
11	0B	VT	Vertikaltabulator (Vertical Tabulation)	27	1B	ESC	Umschaltung (Escape)
12	0C	FF	Formularvorschub (Form Feed)	28	1C	FS	Hauptgruppen-Trennung (File Separator)
13	0D	CR	Wagenrücklauf (Carriage Return)	29	1D	GS	Gruppen-Trennung (Group Separator)
14	0E	SO	Dauerumschaltung (Shift-Out)	30	1E	RS	Untergruppen-Trennung (Record Separator)
15	0F	SI	Rückschaltung (Shift-In)	31	1F	US	Teilgruppen-Trennung (Unit Separator)

Die nächsten Zeichen sind (bis auf eine Ausnahme) Buchstaben oder Spezialzeichen:

Dez:	Hex:	Zeichen:	Dez:	Hex:	Zeichen:	Dez:	Hex:	Zeichen:
32	20	Leerschlag (Space)	64	40	@	96	60	`
33	21	!	65	41	A	97	61	a
34	22	"	66	42	B	98	62	b
35	23	#	67	43	C	99	63	c
36	24	$	68	44	D	100	64	d
37	25	%	69	45	E	101	65	e
38	26	&	70	46	F	102	66	f
39	27	'	71	47	G	103	67	g
40	28	(	72	48	H	104	68	h
41	29	)	73	49	I	105	69	i
42	2A	*	74	4A	J	106	6A	j
43	2B	+	75	4B	K	107	6B	k
44	2C	,	76	4C	L	108	6C	l
45	2D	-	77	4D	M	109	6D	m
46	2E	.	78	4E	N	110	6E	n
47	2F	/	79	4F	O	111	6F	o
48	30	0	80	50	P	112	70	p
49	31	1	81	51	Q	113	71	q
50	32	2	82	52	R	114	72	r
51	33	3	83	53	S	115	73	s
52	34	4	84	54	T	116	74	t
53	35	5	85	55	U	117	75	u
54	36	6	86	56	V	118	76	v
55	37	7	87	57	W	119	77	w
56	38	8	88	58	X	120	78	x
57	39	9	89	59	Y	121	79	y
58	3A	:	90	5A	Z	122	7A	z
59	3B	;	91	5B	[	123	7B	{
60	3C	<	92	5C	\	124	7C	|
61	3D	=	93	5D	]	125	7D	}
62	3E	>	94	5E	^	126	7E	~
63	3F	?	95	5F	_	127	7F	DEL

Das Zeichen mit der Nummer 127 ist nochmals ein Steuerzeichen: DEL steht für Löschen (Delete).

Die restlichen Zeichen gehören nicht mehr zum standardisierten ASCII-Code, sondern hängen vom u.a. vom Betriebssystem und vom gewählten Zeichensatz ab. Eine mögliche Erweiterung ist die folgende: (Beachten Sie, dass nicht jeder Zahl ein Zeichen zugeordnet wird.)

Dez:	Hex:	Zeichen:	Dez:	Hex:	Zeichen:	Dez:	Hex:	Zeichen:	Dez:	Hex:	Zeichen:
128	80		160	A0		192	C0	À	224	E0	à
129	81		161	A1	¡	193	C1	Á	225	E1	á
130	82	‚	162	A2	¢	194	C2	Â	226	E2	â
131	83	*f*	163	A3	£	195	C3	Ã	227	E3	ã
132	84	„	164	A4	¤	196	C4	Ä	228	E4	ä
133	85	…	165	A5	¥	197	C5	Å	229	E5	å
134	86	†	166	A6	¦	198	C6	Æ	230	E6	æ
135	87	‡	167	A7	§	199	C7	Ç	231	E7	ç
136	88	ˆ	168	A8	¨	200	C8	È	232	E8	è
137	89	‰	169	A9	©	201	C9	É	233	E9	é
138	8A	Š	170	AA	ª	202	CA	Ê	234	EA	ê
139	8B	‹	171	AB	«	203	CB	Ë	235	EB	ë
140	8C	Œ	172	AC	¬	204	CC	Ì	236	EC	ì
141	8D		173	AD	-	205	CD	Í	237	ED	í
142	8E		174	AE	®	206	CE	Î	238	EE	î
143	8F		175	AF	¯	207	CF	Ï	239	EF	ï
144	90		176	B0	°	208	D0	Ð	240	F0	ð
145	91	‘	177	B1	±	209	D1	Ñ	241	F1	ñ
146	92	’	178	B2	²	210	D2	Ò	242	F2	ò
147	93	“	179	B3	³	211	D3	Ó	243	F3	ó
148	94	”	180	B4	´	212	D4	Ô	244	F4	ô
149	95	•	181	B5	µ	213	D5	Õ	245	F5	õ
150	96	–	182	B6	¶	214	D6	Ö	246	F6	ö
151	97	—	183	B7	·	215	D7	×	247	F7	÷
152	98	˜	184	B8	¸	216	D8	Ø	248	F8	ø
153	99	™	185	B9	¹	217	D9	Ù	249	F9	ù
154	9A	š	186	BA	º	218	DA	Ú	250	FA	ú
155	9B	›	187	BB	»	219	DB	Û	251	FB	û
156	9C	œ	188	BC	¼	220	DC	Ü	252	FC	ü
157	9D		189	BD	½	221	DD	Ý	253	FD	ý
158	9E		190	BE	¾	222	DE	Þ	254	FE	þ
159	9F	Ÿ	191	BF	¿	223	DF	ß	255	FF	ÿ

Anhang C: Häufige Fehlermeldungen

Das folgende Programm erzeugt bei der Kompilation acht Fehlermeldungen und zwei Warnungen:

```
// FEHLER.CPP
// Dieser Quellcode erzeugt bei der Kompilation einige typische Fehlermeldungen.

#include <champ.h>
#include "fehler.rh"

void stufe ( int hoehe );

void gmain ()
{
   int x1 = 20;

   ginit( "Champ" );
   randomize();
   if ( random( 2 ) == 0 )
      stufe( x2 );
   CP::msgBoxD()
   gend();
}

void stufe ( int hoehe )
{
   forward( Hoehe );
   right( 90 );
   forward( 20 );
   left( 90 );
}
```

Die Meldungen lauten der Reihe nach:

- `Error FEHLER.CPP 15: Call to undefined function 'randomize' in function gmain()` Damit der Compiler den Befehl `randomize()` erkennt, muss die Datei `stdlib.h` mit dem Befehl `#include <stdlib.h>` eingebunden werden (vgl. Seite 61 und Anhang Anhang D:).
- `Error FEHLER.CPP 16: Call to undefined function 'random' in function gmain()` Auch dieser Fehler lässt sich auf das fehlende `#include <stdlib.h>` zurückführen.
- `Error FEHLER.CPP 17: Undefined symbol 'x2' in function gmain()` Die Variable `x2` ist unbekannt, mögliche Ursachen sind: ein Tippfehler oder vergessene Definition der Variablen.
- `Error FEHLER.CPP 19: Statement missing ; in function gmain()` Der Compiler hat in der Zeile 19 einen Fehler gefunden, er vermisst bei einer Anweisung (statement) ein Semikolon. Normalerweise fehlt das Semikolon **in einer Zeile weiter oben** (hier in Zeile 18)!
- `Warning FEHLER.CPP 20: 'x1' is assigned a value that is never used in function gmain()` Sie haben in der Funktion eine Variable definiert, die Sie nie verwenden. Steht in direktem Zusammenhang mit dem Fehler in der Zeile 17!
- `Error FEHLER.CPP 24: Call to undefined function 'forward' in function stufe(int)` Die Funktion `forward` ist nicht definiert, weil der Befehl `#define GLOBAL_TURTLE`, ohne den keine Turtle-Graphik gezeichnet werden kann, im Kopf des Programms fehlt.
- `Error FEHLER.CPP 24: Undefined symbol 'Hoehe' in function stufe(int)` Die Variable `Hoehe` ist wegen eines Schreibfehlers (Gross-/Kleinschreibung) unbekannt!
- `Error FEHLER.CPP 25: Call to undefined function 'right' in function stufe(int)`
- `Error FEHLER.CPP 27: Call to undefined function 'left' in function stufe(int)` Diese Fehler lassen sich wiederum auf das fehlende `#define GLOBAL_TURTLE` zurückführen.
- `Warning FEHLER.CPP 28: Parameter 'hoehe' is never used in function stufe(int)` Diese Warnung ist eine direkte Folge des Schreibfehlers in der Zeile 24.

Anhang D: Header-Dateien

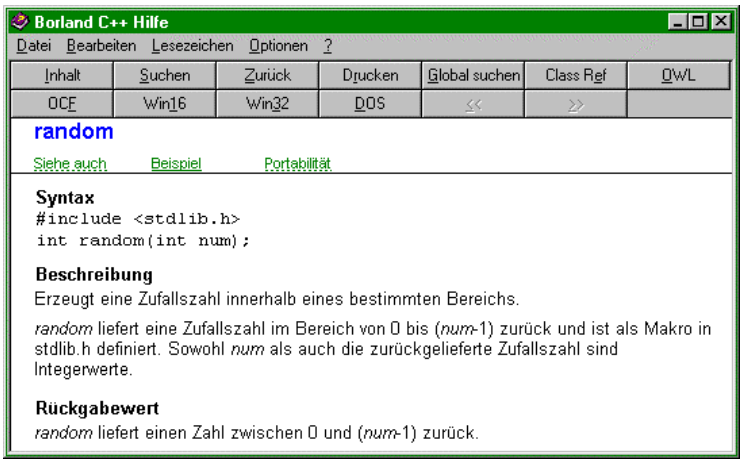
Schon im Kapitel 3.2 (Programm `treppe2.cpp`) haben wir gesehen, dass jede Funktion im Programmkopf deklariert werden muss. Trotzdem haben wir bereits in diesem Programm Funktionen verwendet, die wir nicht deklariert haben: z.B. `forward`, `right` oder `left`. Wo aber sind die Deklarationen (und Definitionen) dieser Funktionen geblieben?

Der ursprüngliche Befehlssatz des Sprache C++ besteht aus relativ wenigen Befehlen und Funktionen. Damit wir unsere Programme trotzdem mit einem komfortablen Befehlssatz schreiben können, wurden mit der Zeit unzählige Funktions- und Klassenbibliotheken geschrieben. Dabei werden die neuen Funktionsdeklarationen und Klassendefinitionen in sogenannten **Header-Dateien** geschrieben, die dann bei Bedarf mit dem Befehl `#include <...>` eingebunden werden. Header-Dateien erkannt man an der Erweiterung `.h`.

Mit anderen Worten: Die Deklarationen aller nicht vom Programmierer geschriebenen Funktionen (und Klassen) stehen in den Header-Dateien.

In all unseren Programmen haben wir mit `#include <champ.h>` die Header-Datei `champ.h` eingefügt. Sie enthält alle wichtigen Definitionen und Deklarationen zur Graphik sowie zur Ein- und Ausgabe auf dem Bildschirm, ruft aber auch weitere Header-Dateien auf. Jedoch stiessen wir schon in unseren relativ einfachen Programmen auf Funktionen, die nicht in der Datei `champ.h` deklariert worden sind: `random` und `randomize` (Kapitel 8.3), `clrscr` (Kapitel 10.2), Befehle zur Bearbeitung von String-Objekten und Strings (Kapitel 8.1 und 8.5) sowie Befehle zur Ausgabe von Daten in eine Datei (Kapitel 10.4). In allen Fällen banden wir mit `#include <...>` die nötige Header-Datei ein.

Wird das Einbinden vergessen, so erscheint beim Kompilieren z.B. die Fehlermeldung `Call to undefined function 'random' in function gmain()`. Ruft man dann die Hilfe zu `random` auf (z.B. mit der Taste `<F1>`), so erscheint das folgende Fenster der Borland C++ - Hilfe:



Aus der Angabe der Syntax des Befehls `random` entnehmen wir, dass die Datei `stdlib.h` eingebunden werden muss.

Neue Programme, die mit dem Champ Project Manager geöffnet werden (vgl. Kapitel 2.5.1), enthalten unter anderem auch die Zeile `#include "prog.rh"` (`prog` steht für den Namen des Programms). Die Dateien mit der Erweiterung `.rh` sind Header-Dateien, die die sogenannten **Ressourcen** enthalten. Ressourcen sind Daten, die die sichtbaren Teile einer Windows-Anwendung (Menüs, Dialogboxen usw.) definieren.

Der Name der Header-Datei wird entweder zwischen eckigen Klammern (`<...>`) oder zwischen Anführungszeichen (`"..."`) geschrieben. Im ersten Fall muss sich die Datei in einem Ordner befinden, der in den Optionen des Projekts (Optionen|Projekt) definiert werden kann (ansonsten erhält man eine Fehlermeldung des Compilers). Im zweiten Fall wird zuerst im aktuellen Ordner (meistens ist das der Ordner, in dem sich das kompilierte Programm befindet), erst dann wie im ersten Fall nach der Header-Datei gesucht.

Anhang E: Nützliche Tastenkombinationen

Viele Befehle des Editors können statt mit der Mauspalette auch mit Tastenkombinationen (Short-Cuts) aufgerufen werden. Eine Auswahl davon wurde bereits in den Kapiteln 2.4 und 2.5 angegeben. Die nützlichsten Short-Cuts werden in diesem Anhang zusammengestellt:

Bewegung des Cursors

<End>	Cursor zum Zeilenende
<Home>	Cursor zum Zeilenanfang
<Ctrl> + <←>	Cursor ein Wort nach links bewegen
<Ctrl> + <→>	Cursor ein Wort nach rechts bewegen
<Ctrl> + <Home>	Cursor zum Dateianfang bewegen
<Ctrl> + <End>	Cursor zum Dateiende bewegen

Blockbefehle

<Ctrl> + <Del>	Den momentan markierten Block löschen
<Ctrl> + <C> oder <Ctrl> + <Ins>	Bearbeiten|Kopieren
<Ctrl> + <X> oder <Shift> + <Del>	Bearbeiten|Ausschneiden
<Ctrl> + <V> oder <Shift> + <Ins>	Bearbeiten|Einfügen
<Ctrl> + <Shift> + <I>	Block einrücken
<Ctrl> + <Shift> + <U>	Block ausrücken
<Ctrl> + <Shift> + <Links>	Wort links vom Cursor markieren
<Ctrl> + <Shift> + <Rechts>	Wort rechts vom Cursor markieren
<Ctrl> + <Shift> + <Home>	Von der Cursorposition bis zum Dateianfang markieren
<Ctrl> + <Shift> + <Ende>	Von der Cursorposition bis zum Dateiende markieren

Löschbefehle

<Del>	Zeichen rechts vom Cursor löschen
<Ctrl> + <Del>	Den momentan markierten Block löschen
<Ctrl> + <T>	Wort löschen
<Ctrl> + <Y>	Zeile löschen
<Ctrl> + <Shift> + <Y>	Löschen bis zum Zeilenende

Compilerbefehle:

<F9>	Projekt/Projekt aktualisieren
<Alt> + <F9>	Projekt/Compilieren
<Ctrl> + <F9>	Debug/Ausführen, (Starten eines Programms)

Weitere Befehle:

<F1>	Hilfe/Suche über Schlüsselwort
<Ins>	Einfügemodus ein|aus

Anhang F: Programmabsturz! Was nun?

Es kommt häufig vor, dass ein Programm nicht normal beendet werden kann, sondern irgendwo "hängt", beispielsweise wegen einer Endlosschleife oder eines Laufzeitfehlers. Das Programm kann aber trotzdem noch auf verschiedene Arten abgebrochen werden:

1. Klicken Sie in der Mauspalette auf das Symbol "Aktuellen Prozess beenden", (das ist das Symbol mit dem roten Punkt)!
2. Meistens öffnet das Programm ein Graphikfenster oder die Textkonsole. Schliessen Sie das Fenster mit dem entsprechenden Symbol in der Titelleiste oben rechts!
3. Kann das Programm immer noch nicht beendet werden, dann öffnen Sie mit <Ctrl> + <Alt> + <Del> die Liste mit den laufenden Programmen, mit Hilfe derer Sie die Anwendung schliessen können (vgl. Seite 10).
4. Hilft auch das nichts mehr, so ist ein Neustart des Computers nötig.

Hinweis: Schreiben Sie in Schleifen, in denen Sie nur Text in das Konsolenfenster ausgeben und/oder eine Berechnung ausführen, den Befehl `CP::yield();` hin, damit Sie das Programm mit "Aktuellen Prozess beenden" oder mit dem Schliessen des Fensters beenden können , falls Sie unabsichtlich eine Endlosschleife programmiert haben (vgl. Seite 34).

Stichwortverzeichnis

Literaturhinweise

Die folgende Literaturliste soll eine Auswahl an Büchern zu den Themen Informatik, Programmierung in C und C++ sowie Objektorientierte Programmierung angeben.

Informatik:

- Les Goldschlager, *Informatik: eine moderne Einführung*, 3. Auflage, Carl Hanser Verlag, München, Wien
- *Duden Informatik*, 2. Auflage, Dudenverlag Mannheim, Leipzig, Wien, Zürich
- René Hugelshofer (Hrsg.), *Informatik, Anwendungen - Algorithmen - Computer - Gesellschaft*, 2. Auflage, Verlag Moritz Diesterweg, Frankfurt am Main, Verlag Sauerländer, Aarau

C++ und Champ

- Burkhard Sachs, *Borland C++ und Turbo C++, Eine Einführung*, Carl Hanser Verlag, München, Wien
- Petra Nortz, Franz Morick, *C / C++ Referenz*, Franzis Verlag
- Bjarne Stroustrup, *Die C++ Programmiersprache*, 2. Auflage, Addison-Wesley Verlag
- Nicolai Josuttis, *Objektorientiertes Programmieren in C++*, Addison-Wesley Verlag
- Joel Adams, Sanford Leestma, Larry Nyhoff, *Turbo C++, An Introduction to Computing*, Prentice-Hall
- Aegidius Plüss, *Introduction to object-oriented programming with C++ and the MS-Windows framework Champ*, Version 1.02, Universität Bern, Konferenz der Lehrerbildungsinstitutionen
- Hans Salvisberg *Champ Documentation*, Salvisberg Software & Consulting