Einführung in das Programmieren mit Visual C++ 6.0

2003.05.16

Herausgeber: Universitäts-Rechenzentrum Trier
Universitätsring 15
D-54286 Trier
WWW: http://urt.uni-trier.de/index.php?id=518
E-Mail: urt@uni-trier.de
Tel.: (0651) 201-3417
Leiter: Prof. Dr.-Ing. Manfred Paul
Autor: Bernhard Baltes-Götz (E-Mail: baltes@uni-trier.de)
Druck: Druckerei der Universität Trier

Vorwort

Dieses Manuskript entstand als Begleitlektüre zum C++ - Einführungskurs, den das Universitäts-Rechenzentrum Trier (URT) im WS 2002/2003 angeboten hat.

Die wesentlichen **Lernziele** des Textes sind:

- Einführung in die objektorientierte Programmierung mit C++
  Das Manuskript beginnt allerdings mit elementaren Sprachelementen, gewissermaßen mit einer Einführung in die prozedurale Programmiersprache C.
- Einführung in die Windows-Programmierung unter Verwendung der Microsoft Foundation Classes (MFC) als Klassenbibliothek
- Elementare Fertigkeiten im Umgang mit der Entwicklungsumgebung Microsoft Visual C++

Von den Leser(innen) wird erwartet

- EDV-Allgemeinbildung
  Dass die Leser(innen) wenigstens durchschnittliche Erfahrungen bei der *Anwendung* von Computerprogrammen haben sollten, versteht sich von selbst.
  Im Text wird zwar die Entwicklungsumgebung Visual C++ unter MS-Windows benutzt, doch sind die Kursinhalte mit Ausnahme des Abschnitts über Windows-Programmierung relativ betriebssystem-unabhängig.
- Programmierkenntnisse werden *nicht* vorausgesetzt.
  Leser(innen) *mit* Programmiererfahrung werden sich bei den ersten Abschnitten eventuell etwas langweilen.
- Motivation
  Generell ist mit einem erheblichen Zeitaufwand bei der Lektüre und bei der aktiven Auseinandersetzung mit dem Stoff (z.B. durch das Lösen von Übungsaufgaben) zu rechnen.

Dieses Kursmanuskript ist im PDF-Format zusammen mit den behandelten Beispielen und Lösungsvorschlägen zu vielen Übungsaufgaben auf dem Webserver der Universität Trier von der Startseite (http://www.uni-trier.de/) ausgehend folgendermaßen zu finden:

Rechenzentrum > Studierende > EDV-Dokumentationen >
Programmierung > Einführung in das Programmieren mit Visual C++ 6.0

Trier, im April 2003 Bernhard Baltes-Götz

Inhaltsverzeichnis

1 Einleitung

1.1 Was ist ein Computerprogramm?

Eine erschöpfende Antwort auf diese Frage wäre recht aufwändig und an dieser Stelle nicht sonderlich gut platziert. Es geht um einen ersten Eindruck vom Ziel unserer Bemühungen:
Ein Computerprogramm ist eine Folge von Anweisungen, um ein bestimmtes Problem zu lösen
In folgendem Beispiel wird mit einem C++ - Programm die Fakultät einer vom Benutzer gewählten Zahl ausgerechnet:

```
// Einleitungsbeispiel: Berechnung der Fakultät

#include <iostream.h>

void main()
{
  int argument = 1;
  double fakul;

  while (argument > 0)
  {
    cout << "\nArgument (0 = Ende): ";
    cin >> argument;
    if (argument > 0)
    {
      fakul = 1.0;
      for (int i = 1; i <= argument; i++)
        fakul *= i;
      cout << "Fakultaet: " << fakul << endl;
    }
  }
}
```

1.2 Hinweise zu Grundprinzipien des Softwaredesigns

Jedes seriöse Lehrbuch der Programmierung legt größten Wert darauf, dass der konkreten Kodierung eine ausführliche Planungsphase vorauszugehen hat, und stellt entsprechende (heutzutage meist objektorientierte) Techniken vor (siehe z.B. Lee & Phillips, 1997, S. 63ff oder RRZN, 1999, S. 16ff). Dass wir uns (zumindest vorerst) wenig mit Überlegungen zum Entwurf von Softwaresystemen aufhalten, kann man folgendermaßen begründen:
Der Entwurf muss zumindest *auch* die verfügbaren Konzepte einer Programmiersprache in Rechnung stellen. Folglich kann es nicht völlig sinnlos sein, zunächst die von C++ angebotenen Konzepte kennen zu lernen.
Spätestens im Zusammenhang mit der objektorientierten Programmierung werden wir uns mit Fragen des Softwaredesigns etwas näher beschäftigen.

1.3 Argumente für die Programmiersprache C++

C++ ist die objektorientierte Weiterentwicklung der prozeduralen Programmiersprache C, die im Zusammenhang mit der Entwicklung des Betriebssystems UNIX entstand. Auf eine Einordnung von C++ im Ensemble der verschiedenen Programmiersprachen bzw. Softwaretechnologien soll hier verzichtet werden (siehe z.B. RRZN 1999).
Einige Vor- und Nachteile von C++ sollen jedoch erwähnt werden, weil sie eventuell relevant sind für die Motivation zur Kursteilnahme:

- C/C++ ist außerordentlich verbreitet und auf praktisch jeder Rechner-Plattform anzutreffen. U.a. sind wesentliche Teile der aktuellen Betriebssysteme in C/C++ programmiert. Als Folge der hohen Verbreitung finden C/C++ - Programmierer für sehr viele Aufgabenstellungen Unterprogramm-Bibliotheken bzw. -Klassenhierarchien vor, die in eigenen Projekten verwandt werden können.
- C++ ist auf dem Arbeitsmarkt nach wie vor gefragt und kann als Industriestandard gelten. Ein großer Teil der heutigen Standardsoftware ist in C++ erstellt worden.
  Unter Windows wird allerdings die Microsoft-Kreation C# vermutlich mittelfristig die Nachfolge von C++ antreten. Hier handelt es sich um eine von Java inspirierte, rein objektorientierte Sprache, auf die erfahrene C++ - Programmierer relativ leicht umsteigen können.
- C++ gilt nicht unbedingt als ideale Programmiersprache für Anfänger:
  - Sie enthält einige Tücken für unerfahrene Programmierer.
  - Es ist eine hybride Sprache mit prozeduralen und objektorientierten Elementen.

Aufgrund des hohen Verbreitungsgrades und der Bedeutung auf dem Arbeitsmarkt ist es trotz gewisser didaktischer Erschwernisse attraktiv, C++ zu lernen.

1.4 Vom Quellcode zum ausführbaren Programm

Die von uns erstellten Computerprogramme (siehe obiges Beispiel) sind in der **problemorientierten** Programmiersprache C++ verfasst. Von diesem **Quellcode**, der grundsätzlich mit einem beliebigen Texteditor erstellt werden kann, bis zum ausführbaren Programm sind noch einige Verarbeitungsschritte erforderlich, denn jede CPU versteht nur einen speziellen Satz von Befehlen, die als Folge von Nullen und Einsen (**Maschinencode**) formuliert werden müssen. Die ebenfalls CPU-spezifische Assemblersprache stellt eine lesbare Form des Maschinencodes dar.
Die CPU holt sich in einer „Endlosschleife“ einen Maschinenbefehl nach dem anderen aus dem Hauptspeicher und führt ihn aus, heutzutage immerhin mehrere hundert Millionen Befehle pro Sekunde.

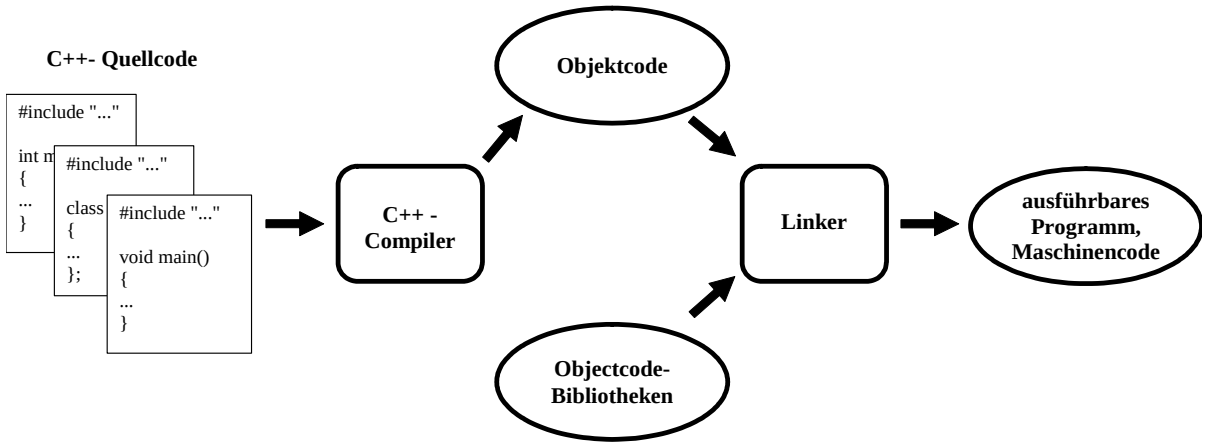
Der Quellcode muss also erst übersetzt werden, was der so genannte **Compiler** erledigt. Hier handelt es sich um ein bereits vorhandenes Programm, das wir zum Erzeugen neuer Programme benutzen. Das Übersetzungsergebnis wird als **Objektcode** bezeichnet.[1]

Er kann jedoch noch nicht ausgeführt werden, weil in der Regel externe Referenzen enthalten sind, also Aufrufe von **Bibliotheksroutinen**. In obigem Beispielprogramm tauchen z.B. die Objekte `cin` und `cout` auf, die nicht zum eigentlichen Sprachumfang von C++ gehören und erst über die Bibliothek **iostream** verfügbar werden.
Man informiert den Compiler über die benötigten Bibliotheksroutinen, indem entsprechende Header-Dateien am Anfang des Quelltextes über **#include**-Anweisungen eingebunden werden (siehe Beispiel). In einer Header-Datei sind die Schnittstellen der Bibliotheksroutinen deklariert, so dass der Compiler deren korrekten Gebrauch überprüfen kann.

Nach dem Compiler tritt der so genannte **Linker** (Verbinder) in Aktion. Er ist u.a. dafür zuständig, die externen Referenzen aufzulösen, indem Objektcode aus Bibliotheken eingefügt wird.

Wie der Compiler und der Linker aus dem Quellcode sowie den benötigten Objektcode-Bibliotheken ein ausführbares Programm (Maschinencode) erstellen, zeigt die folgende Abbildung nach Lee & Phillips (1997, S. 55):

[1] Zwischen dem Objektcode und der (bisher nur andeutungsweise erwähnten) objektorientierten Programmierung gibt es übrigens *keine* besondere Beziehung.



Beim Einsatz einer Entwicklungsumgebung werden Compiler und Linker automatisch im Hintergrund ausgeführt, wenn der Benutzer das Erstellen seines Programms verlangt. Bei Visual C++ 6.0 handelt es sich beim Compiler um das Programm

…\Microsoft Visual Studio\VC98\Bin\cl.exe

und als Linker kommt das Programm

…\Microsoft Visual Studio\VC98\Bin\link.exe

zum Einsatz. Man könnte sie auch direkt aufrufen, doch ihre Verwendung über die graphische Benutzerschnittstelle der Entwicklungsumgebung ist deutlich bequemer (siehe Abschnitt 2).

In folgender Passage unseres Beispielprogramms wird die Fakultät des übergebenen Argumentes definitionsgemäß durch wiederholte Multiplikation ermittelt:

```
for (int i = 1; i <= argument; i++)
  fakul *= i;
```

Daraus erstellt der Compiler folgenden Objektcode (hexadezimal notiert), der mangels externer Aufrufe bereits fertigen Maschinencode darstellt:

```
C7 45 F0 01 00 00 00
EB 09
8B 4D F0
83 C1 01
89 4D F0
8B 55 F0
3B 55 FC
7F 0B
DB 45 F0
DC 4D F4
DD 5D F4
EB E4
```

Der zugehörigen Assemblercode wirkt schon etwas zugänglicher. Hier tauchen Befehlskennungen, Register und Speicheradressen auf:

```
mov         dword ptr [i],1
jmp         main+6Ch (004010ac)
mov         ecx,dword ptr [i]
add         ecx,1
mov         dword ptr [i],ecx
mov         edx,dword ptr [i]
cmp         edx,dword ptr [ebp-4]
jg          main+7Fh (004010bf)
fild        dword ptr [i]
fmul        qword ptr [ebp-0Ch]
fstp        qword ptr [ebp-0Ch]
jmp         main+63h (004010a3)
```

Während in der betrachteten Programmpassage keine Bibliotheksroutinen auftreten, nehmen diese im ausführbaren Programm **Fakul.exe**, das wir in Abschnitt 2 erstellen werden, einen sehr großen Raum ein.

Nach diesem Ausflug in die Welt des Maschinencodes, mit der wir im Programmier-Alltag kaum in Berührung kommen, soll noch einmal zusammengefasst werden, welche Werkzeuge bzw. Hilfsmittel zum Erstellen von ausführbareren C++ - Programmen erforderlich sind:

- ein Texteditor zum Verfassen des Quellcodes
- Objektcode-Bibliotheken mit Standardroutinen
- ein Compiler
  Er übersetzt den Quellcode in Objektcode.
- ein Linker
  Er verbindet den Objektcode mit den darin aufgerufenen Routinen aus Objektcode-Bibliotheken zum ausführbaren Programm.

2 Einstieg in Visual C++ 6

Microsofts Visual C++ 6 ist eine integrierte Umgebung zum Entwickeln von C++ - Programmen für Windows. Aus dem Quellcode kann Visual C++ (VC++) Maschinencode in verschiedenen Dateiformaten erzeugen, z.B.:

- Selbständige Anwendungen (vom Konsolen- oder Windows-Typ)
- Dynamische Link-Bibliotheken (DLLs)
- ActiveX-Komponenten

Beim Umgang mit VC++ ist eine außerordentliche Fülle von Informationen zur Programmiersprache und zur Betriebssystemumgebung potentiell relevant. Damit wird die als Online-Hilfe verfügbare **MSDN Library Visual Studio 6.0** zum unverzichtbaren Werkzeug (für Anfänger und Profis).

In diesem Abschnitt werden wir aus dem obigen Beispiel-Quellcode ein ausführbares Programm erzeugen und dabei einige elementare Konzepte der VC++ - Entwicklungsumgebung behandeln. Wesentliche Funktionen (z.B. Debugger, Klassenassistent) werden wir später kennen lernen.

2.1 Projekt

Um ein Programm, eine Bibliothek oder eine Komponente zu entwickeln, wird ein *Projekt* angelegt. Im Projektordner sind alle zugehörigen Dateien enthalten (z.B. Quelltexte, Kompilate, Ressourcen).

2.2 Arbeitsbereich

Ein Arbeitsbereich enthält im Normalfall *ein* Projekt. Eine Ausnahme kann z.B. gemacht werden, um Dateien zwischen zwei Projekten zu kopieren.
Alle zu einem Arbeitsbereich gehörigen Projekte werden im Arbeitsbereichsfenster der Entwicklungsumgebung angezeigt (s.u.).
Um die Arbeit an einem vorhandenen Projekt fortzusetzen, öffnet man am besten den Arbeitsbereich, zu dem es gehört.

2.3 Assistenten

Visual C++ bietet Assistenten für zahlreiche Standardaufgaben. Natürlich dürfen auch Studierende sich dieser Assistenten bedienen, um sich das Leben zu erleichtern. Ein ehrlicher Professor meinte kürzlich zum Uni-Betrieb:

- Studierende müssen Alles wissen.
- Ein Assistent muss wissen, wo es steht.
- Ein Professor muss wissen, wo der Assistent ist.

Welch ein Segen, dass VC++ nun auch den Studierenden zahlreiche Assistenten zur Verfügung stellt, z.B.:

- **Anwendungsassistent**
  Eine Windows-Anwendung muss die Mechanismen eines recht komplexen Betriebssystems beherrschen (, z.B. Botschaften verarbeiten können,) und außerdem seinen verwöhnten Benutzern zahlreiche Komfortmerkmale bieten (z.B. Menüs, Symbolleisten, Statuszeile, Drucken mit Seitenansicht, Datenbankzugriff, Hilfesystem). Daher sind für eine Windows-Anwendung einige Tausend Programmzeilen erforderlich, bevor Ihre eigentliche Programm-Idee überhaupt ins Spiel kommt.
  Der Anwendungsassistent kann eine vollständig funktionstüchtige Windows-Anwendung erstellen, der Sie dann anschließend die gewünschte Funktionalität einverleiben können.

- **Klassenassistent**
  Er wird uns später bei der objektorientierten Programmierung unterstützen.

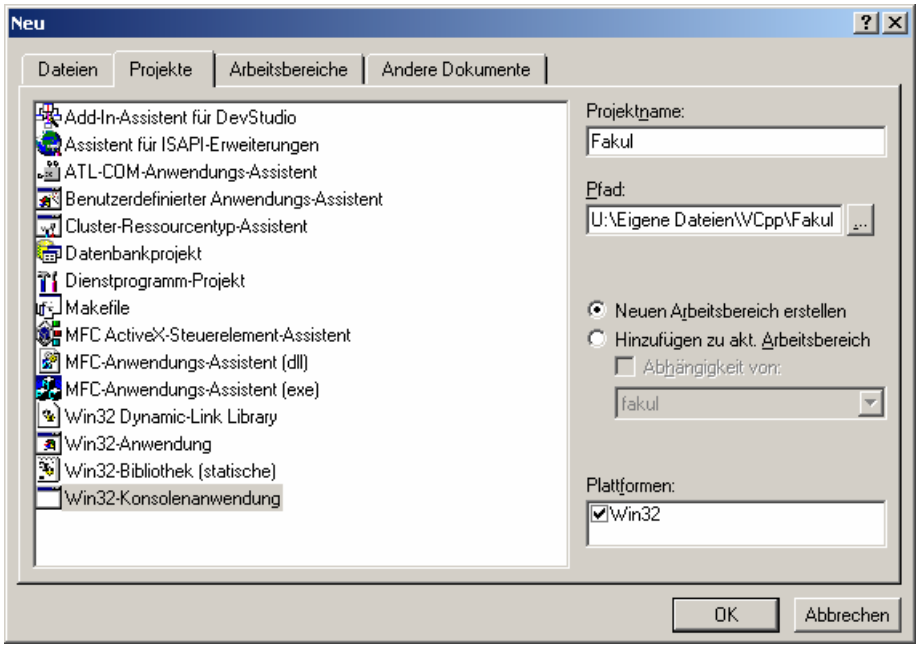
Wo in VC++ die Assistenten sind, kann man sich leicht merken. Nach dem VC++ - Start, der auf den aktuellen URT-Pool-PCs mit

Start > Programme > Programmentwicklung > Microsoft Visual Studio 6.0

angefordert wird, erreicht man über

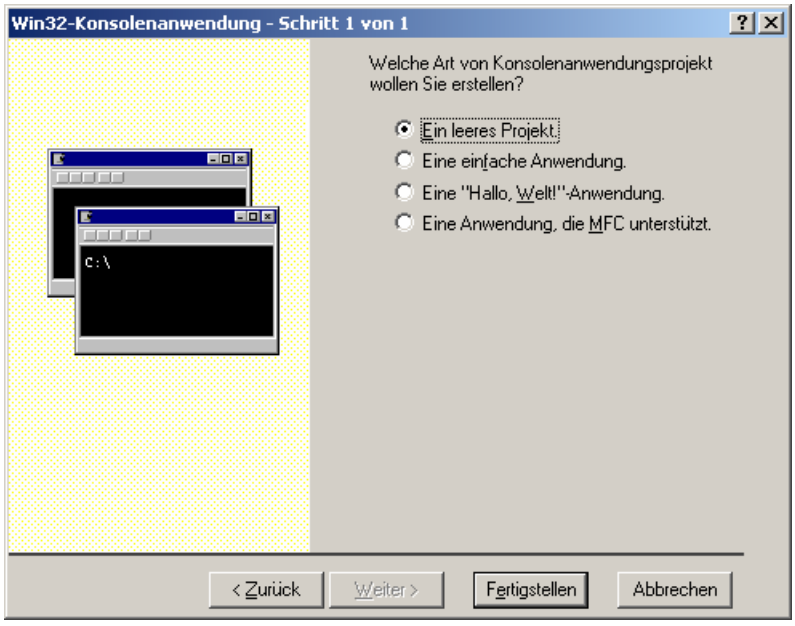
Datei > Neu > Projekte

ein Menü mit Assistenten für diverse Projekttypen.
Wir legen für das Einleitungsbeispiel ein **Win32-Konsolenanwendungs**-Projekt an:

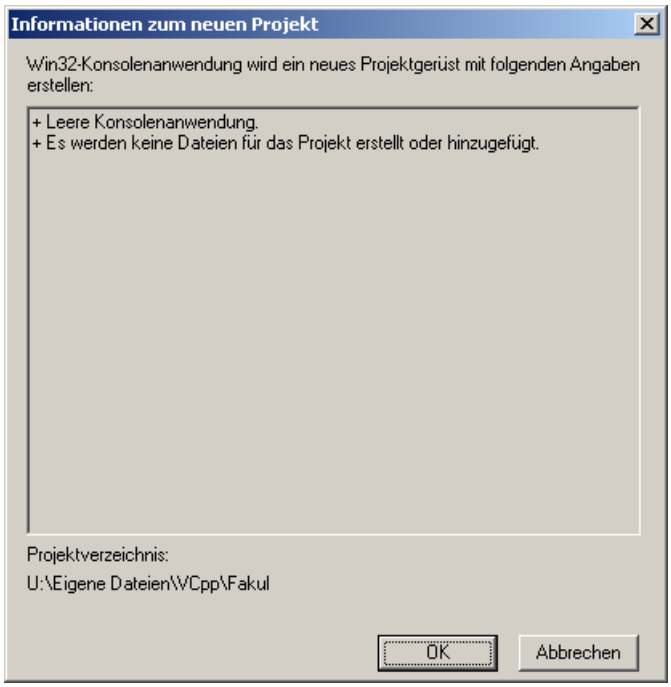


Geben Sie einen geeigneten **Projektnamen** an. Dieser wird automatisch in das Feld **Path** als Name für den **Projektordner** übernommen, wobei jedoch wegen des voreingestellten Wurzelordners oft kein sinnvoller Vorschlag resultiert. Tragen Sie einen Ordner ein, für den Sie Schreibrechte besitzen, z.B. auf Ihrem Benutzerlaufwerk U.

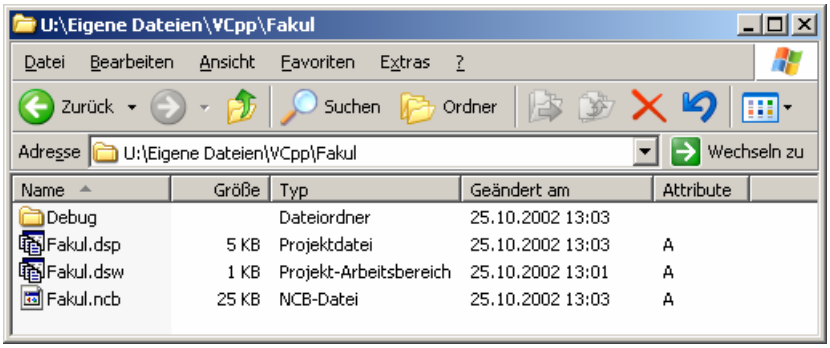
Nun startet der Assistent für Konsolenanwendungen mit seinem ersten und letzten Schritt:



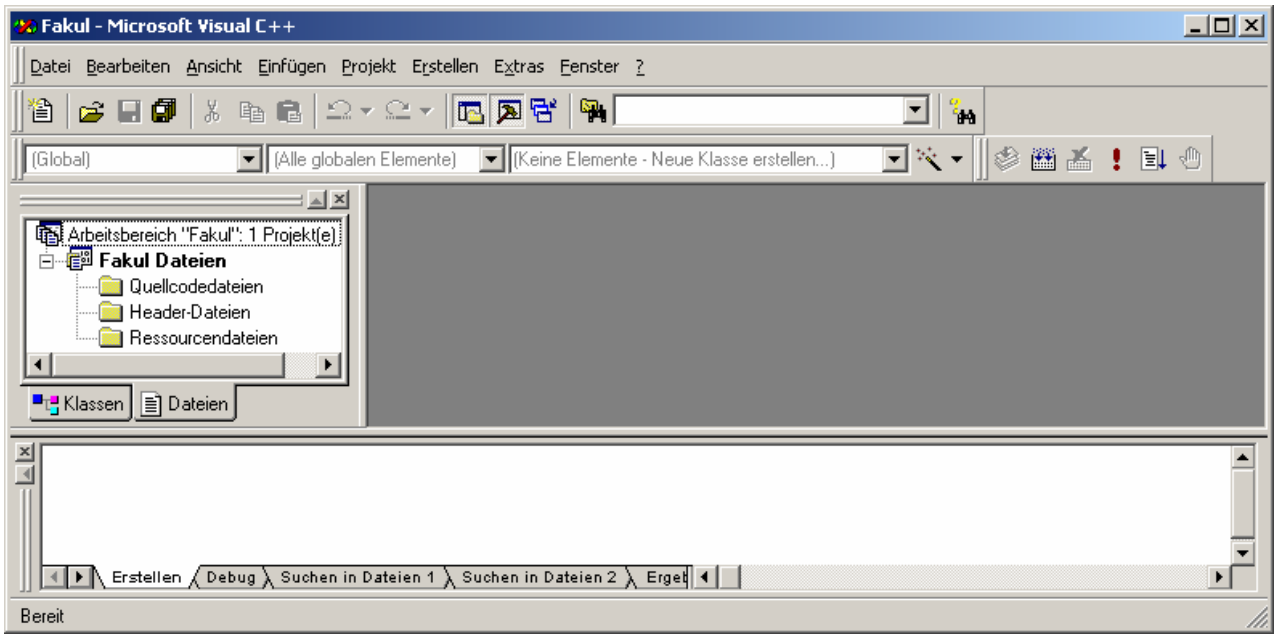
Übernehmen Sie die Voreinstellung **leeres Projekt** mit einem Klick auf den Schalter **Fertigstellen**. Daraufhin erscheint eine Zusammenfassung wichtiger Information zum neuen Projekt:



Wenn Sie diese mit **OK** quittieren, erstellt VC++ den Projektordner mit einigen Hilfsdateien:



Außerdem taucht das neue Projekt im Arbeitsbereichsfenster der Entwicklungsumgebung auf:



Das Arbeitsbereichsfenster kann über den Schalter oder den Menübefehl

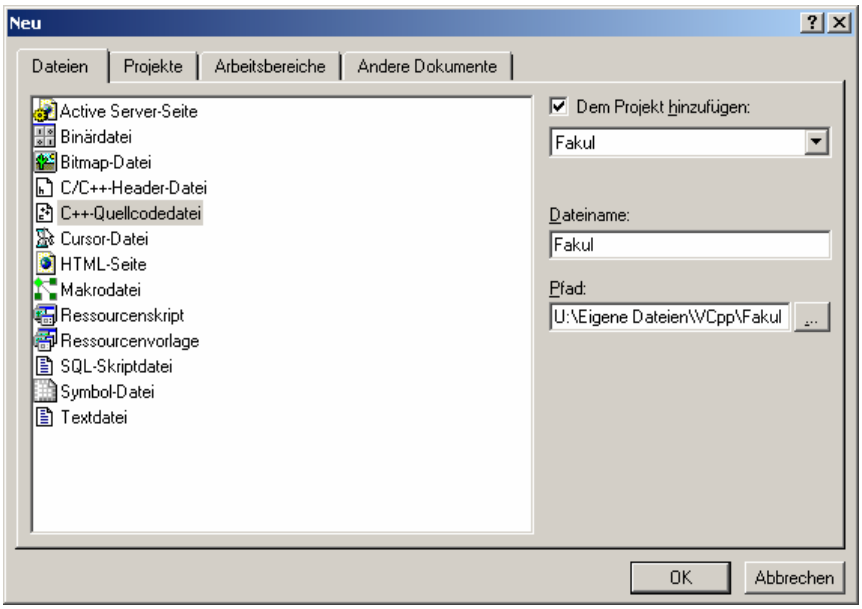
Ansicht > Arbeitsbereich

ein- bzw. ausgeschaltet werden.

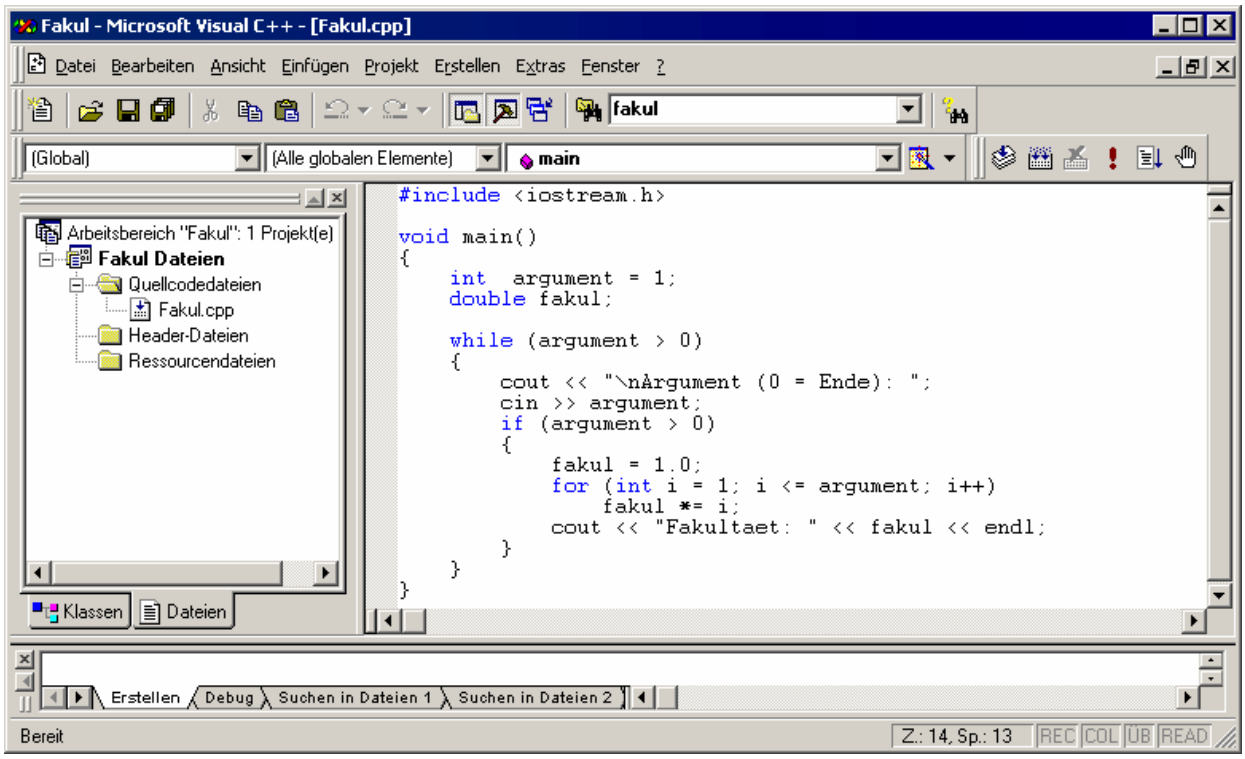
Derzeit enthält unser Projekt weder Dateien (siehe Registerkarte **Dateien**) noch Klassendefinitionen (siehe Registerkarte **Klassen**). Für das in der Einleitung gezeigte Beispielprogramm benötigen wir eine C++-Quellcodedatei, die nach:

Datei > Neu > Datei

mit dem Namen **Fakul.cpp** angelegt wird:



Anschließend taucht auf unserem Arbeitsplatz das für eine Entwicklungsumgebung so elementare Editorfenster auf, und wir können das Beispielprogramm eintragen:

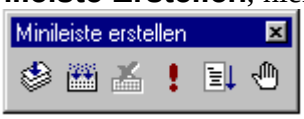


Seine Bestandteile werden später ausführlich erläutert.

2.4 Das ausführbare Programm erstellen

Um das ausführbare Programm **Fakul.exe** aus der Quelltextdatei mit Hilfe von Compiler und Linker (siehe oben) erstellen zu lassen, haben Sie folgende Möglichkeiten:

- Menübefehl **Erstellen > Fakul.exe erstellen**
- Schalter **Erstellen** auf der **Minileiste Erstellen**, hier im „abgerissenen“ Zustand:

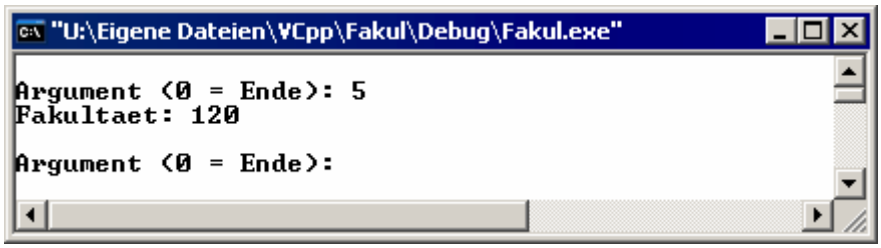


- Tastaturbefehl **<F7>**

Auf der **Minileiste Erstellen** oder mit äquivalenten Optionen im Menü **Erstellen** sind noch einige verwandte Aktionen möglich:

- **Kompilieren (Strg+F7)**
  Kompilieren ohne anschließendes Binden zum ausführbaren Programm
- **Programm ausführen (Strg+F5)**
  Das Programm wird ohne Kontrolle durch den Debugger ausgeführt.
- **Ausführen (F5)**
  Das Programm wird unter Kontrolle des Debuggers ausgeführt.

Wenn Sie das Beispielprogramm fehlerfrei eingetippt haben, können Sie es **Strg+F5** starten und anschließend ausprobieren, z.B.:



Wie Sie der Titelzeile der Konsole mit dem Testlauf entnehmen können, hat VC++ das Programm **Fakul.exe** im **Debug**-Unterverzeichnis zum Projektordner erzeugt.

Hier findet sich in der Datei **Fakul.obj** auch der vom Compiler aus unserer Quelle erzeugte Objektcode (vgl. Abschnitt 1.4). Der enorme Größenunterschied zwischen **Fakul.obj** (6 KB) und **Fakul.exe** (205 KB) geht u.a. auf die vom Linker ergänzten Bibliotheksroutinen zurück.

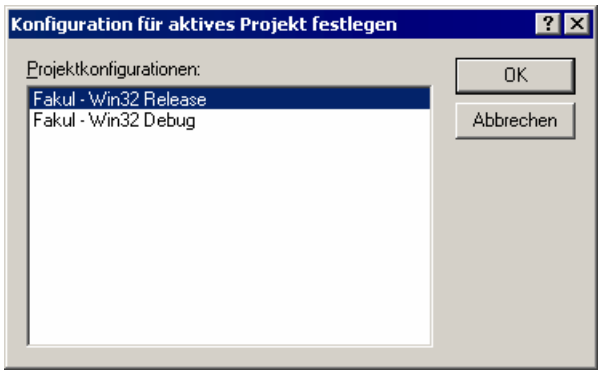
Das Programm **Fakul.exe** sollte auf jedem Rechner mit Windows-32-Bit-Betriebssystem lauffähig sein. Auf anderen Plattformen ist die EXE-Datei (ohne Emulator) nicht zu gebrauchen. Der C++ - Quellcode ist jedoch hochgradig portabel und kann in unveränderter Form unter jedem Betriebssystem von einem aktuellen C++ - Compiler in die dortige Maschinensprache übersetzt werden. Die später zu erstellenden Windows-Programme mit graphischer Benutzerschnittstelle sind hingegen auch auf Quellcode-Ebene so stark mit dem Windows-API „verheiratet“, dass eine Portabilität auf andere Betriebssysteme kaum gegeben ist.

2.5 Debug- und Release-Konfiguration

Die nach obiger Beschreibung im **Debug**-Unterverzeichnis des Projektordners entstandene Programmversion enthält in großem Umfang Code zur Unterstützung von Debug-Methoden, mit denen wir uns später noch beschäftigen werden. Ist dies nach einer ausführlichen Testphase nicht mehr erforderlich, kann die voreingestellte Projekt-Einstellungskonfiguration **Debug** über den Menübefehl

Erstellen > Aktive Konfiguration festlegen

durch die laufzeitoptimierte **Release**-Variante ersetzt werden:



Beim nächsten Erstellen entsteht in unserem Beispiel im Projektordner ein Unterverzeichnis **Release** mit einer ca. 50 KB kleinen Produktionsversion von **Fakul.exe**.

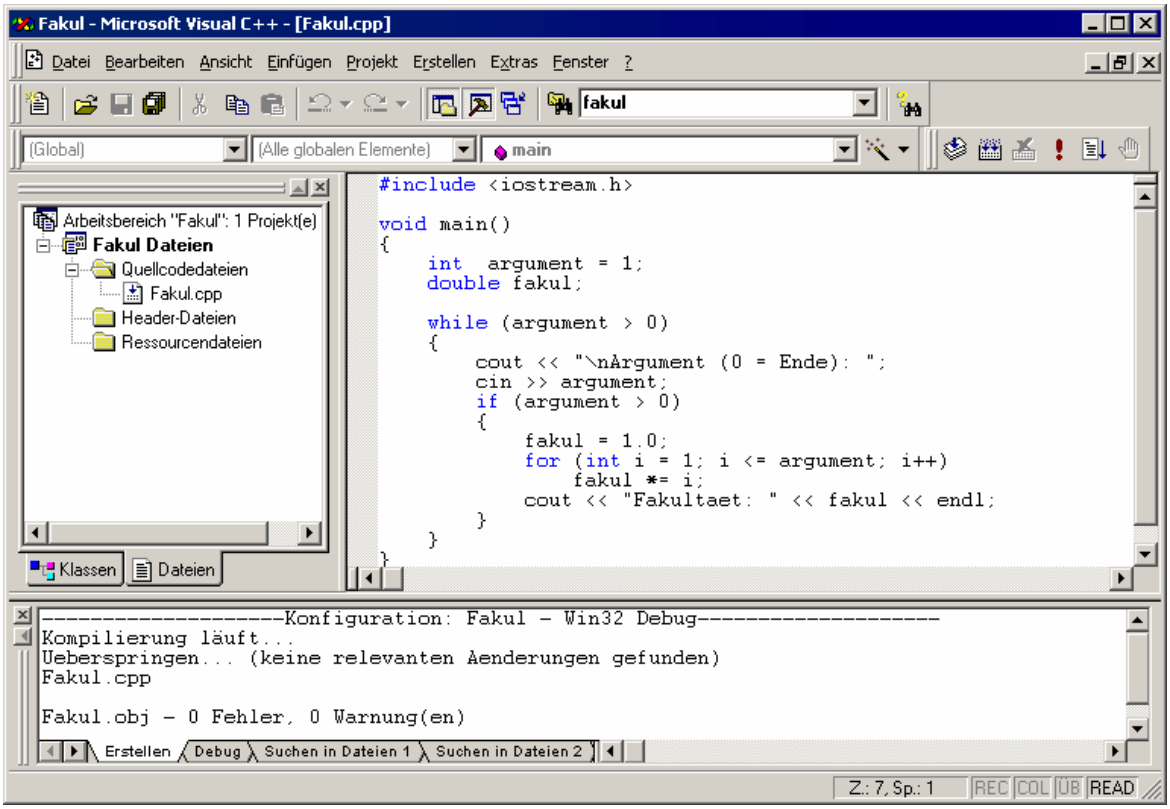
Für Debug- und Release-Modus können über

Projekt > Projekteinstellungen

zahlreiche Einstellungen (z.B. zum Compiler, Linker oder Debugger) getrennt festgelegt werden.

2.6 Das Ausgabefenster

Beim Erstellen des ausführbaren Programms öffnet VC++ nötigenfalls das Ausgabefenster und protokolliert dort das Geschehen. Wenn Sie den Quelltext ohne Fehler eingetragen haben, sollten Sie ungefähr folgendes Ergebnis erhalten:



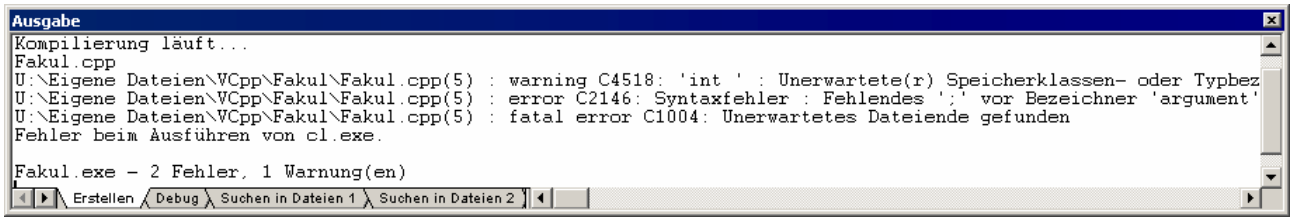
Das Ausgabefenster kann über den Schalter  oder den Menübefehl

Ansicht > Ausgabe

ein- bzw. ausgeschaltet werden.

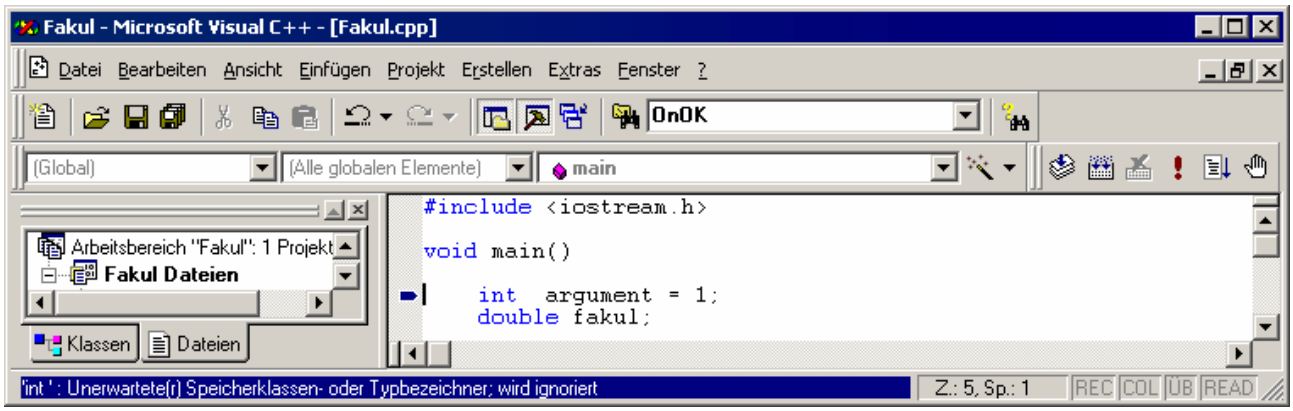
2.7 Hinweise zur Fehlersuche

Auf Dauer werden wir Fehler wohl kaum vermeiden können. Um die Reaktion des Compilers auf Syntaxfehler kennen zu lernen, beseitigen wir die öffnende geschweifte Klammer zum Block der Funktion `main()` und lassen den traurigen Rest kompilieren. Daraufhin erscheinen im Ausgabefenster (hier aus seiner „Verankerung“ gelöst) folgende Fehlermeldungen:



VC++ versucht zu jedem Fehler die betroffene Quelltextzeile zu lokalisieren, jedoch kann es passieren, dass der Compiler erst einige Zeilen hinter dem eigentlichen Fehler Alarm schlägt. Im Beispiel wird die eigentlich fehlerfreie Zeile 5 moniert, die der Compiler durch das Streichen der öffnenden Klammer zum Block der Funktion `main()` nun nicht mehr sinnvoll interpretieren kann.

Nach einem Doppelklick auf eine Ausgabefenster-Fehlermeldung wird die betroffene Zeile im Quelltext-Editor hervorgehoben, z.B.:

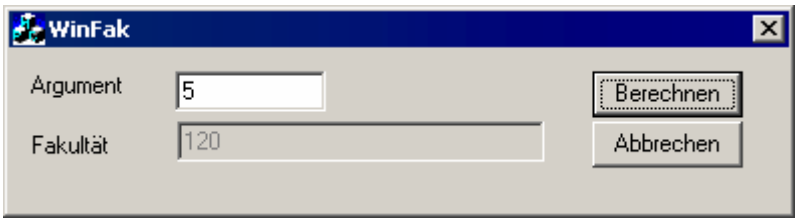


Außerdem wird bei aktivem Quelltext-Editor in der VC++ - Statuszeile angezeigt, in welcher Zeile und Spalte sich die Schreibmarke gerade befindet.

2.8 Ausblick auf die Windows-Programmierung mit MFC

Wer möglichst schnell ein Windows-Programm mit aktuellem „Look-And-Feel“ schreiben möchte, muss in den ersten Abschnitten dieses Kurses eine Durststrecke überstehen, weil erst die Grundlagen der objektorientierten Programmierung mit C++ erarbeitet werden sollen.
Damit Sie dabei nicht den Glauben an Ihre Chance, attraktive Windows-Programme zu erstellen, verlieren, wird in diesem Abschnitt gezeigt, wie mit Hilfe der Assistenten von VC++ ein Windows-Programm zur Fakultätsberechnung relativ flott erstellt werden kann. Lassen Sie sich jedoch nicht von den vielen technischen Details beunruhigen, die Ihnen beim Erstkontakt mit der Windows-Programmierung noch fremd sind. Wir werden uns später ausführlich damit beschäftigen. Sie können den Abschnitt 2.8 als eine Besichtigungstour durch die „höheren Abteilungen“ Ihres neuen Arbeitsplatzes betrachten. Sie sollen sich neugierig umsehen und Einiges ausprobieren, ohne alle Details verstehen und abspeichern zu müssen.

Wir wollen als Alternative zur Konsolenanwendung **Fakul.exe** das folgende Windowsprogramm **WinFak.exe** erstellen:

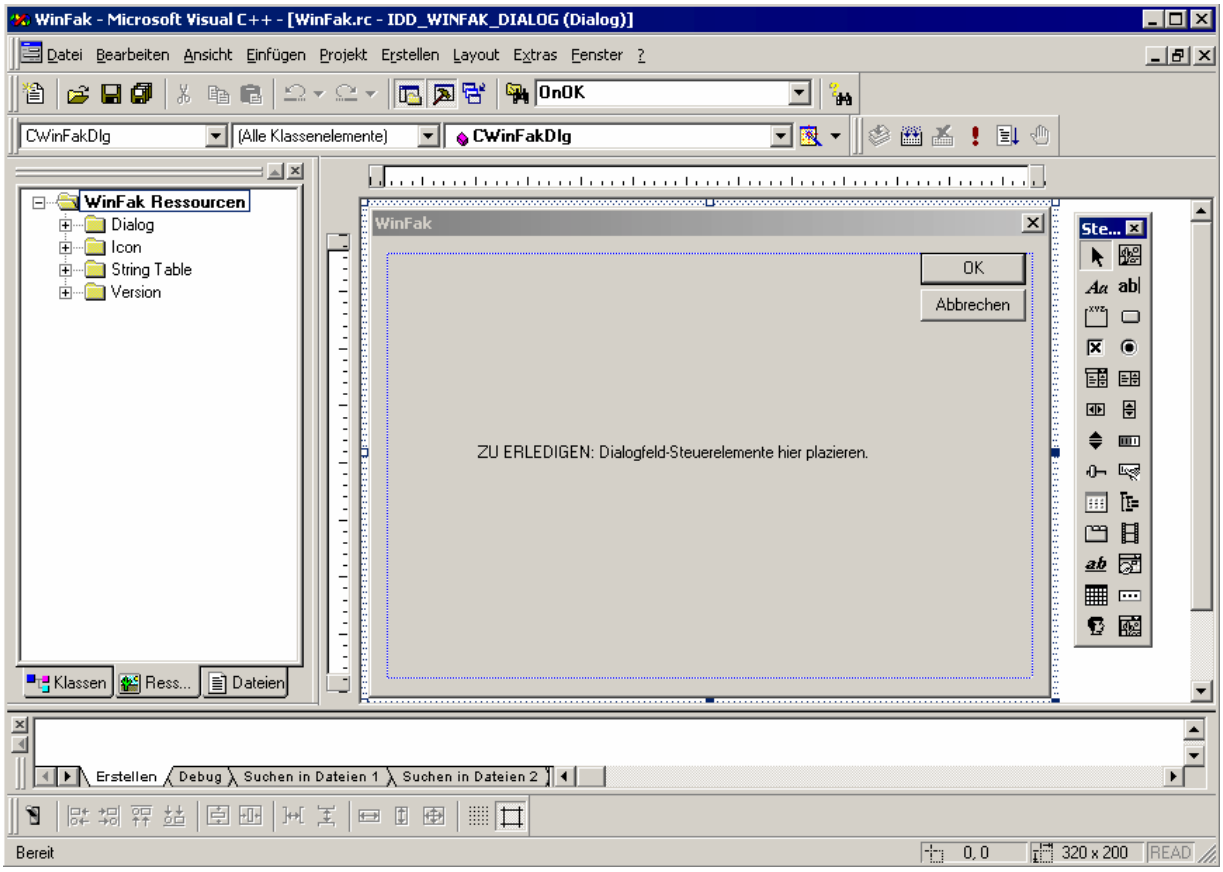


2.8.1 Der Beitrag des Anwendungsassistenten

Wir beauftragen den Anwendungsassistenten, das Gerüst unseres Programms zu erstellen:

- Öffnen Sie mit **Datei > Neu > Projekt** die Dialogbox für neue Projekte und nehmen Sie hier folgende Einstellungen vor:
  - Markieren: **MFC-Anwendungs-Assistent (exe)**
  - Projektname: **WinFak**
  - Pfad: **U:\Eigene Dateien\VCpp\WinFak**
- Wählen Sie im 1. Assistenten-Schritt die Anwendungsart **Dialogfeldbasierend**, und klicken Sie anschließend auf **Fertigstellen**. Mit den zahlreichen Konfigurationsoptionen der weiteren Assistentenschritte wollen wir uns zum jetzigen Zeitpunkt nicht beschäftigen.

Nachdem der Anwendungsassistent seine Arbeit verrichtet hat, präsentiert VC++ u.a. eine in Bearbeitung befindliche Dialogbox, bereits mit zwei Standardschaltflächen (**OK**, **Abbrechen**) bestückt:

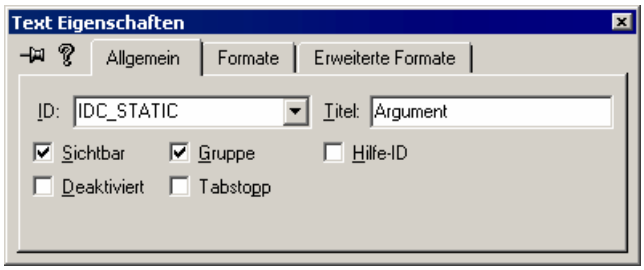


Das Programm kann sogar schon kompiliert, gelinkt und ausgeführt werden (z.B. per **Ctrl+F5**), wobei das Berechnen von Fakultäten allerdings noch nicht gelingt.

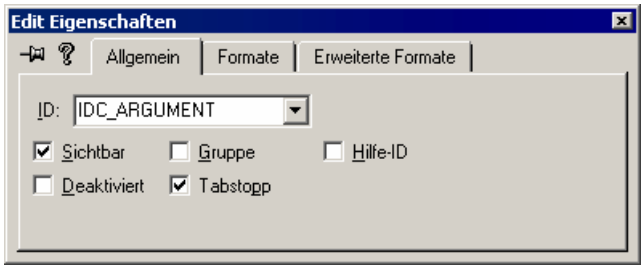
2.8.2 Der Dialogboxdesigner

Nun erstellen wir mit der Steuerelemente-Palette und den Eigenschaftsdialogen zu den einzelnen Steuerelementen die Dialogbox unserer Anwendung, z.B. so

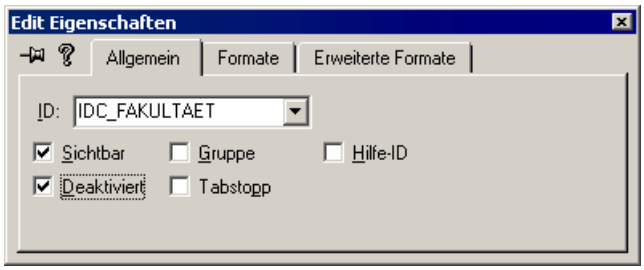
- Schieben Sie das vorhandene Label (ZU ERLEDIGEN: Dialogfeld-Steuerelemente hier plazieren.) in die linke oberer Ecke, wählen Sie **Eigenschaften** aus seinem Kontextmenü, und ändern Sie seinen **Titel**:



  Schließen Sie den Eigenschaftsdialog, und korrigieren Sie per Maus die Größe des Rechtecks zum Label.
- Klicken Sie in der Steuerelemente-Palette auf das **Text**-Symbol (Aa), ziehen Sie in der Dialogbox am passendem Ort in passender Größe ein Rechteck für das Label zur Beschriftung der Ergebnisanzeige auf, und tragen Sie in seinem Eigenschaftsdialog einen passenden Titel ein.
- Klicken Sie in der Steuerelemente-Palette auf das **Eingabefeld**-Symbol (ab), ziehen Sie in unserer Dialogbox am passendem Ort in passender Größe ein Rechteck für die Argumenteingabe auf, und vereinbaren Sie im Eigenschaftsdialog eine leicht zu merkende Identifikation (**ID**) für das Steuerelement:

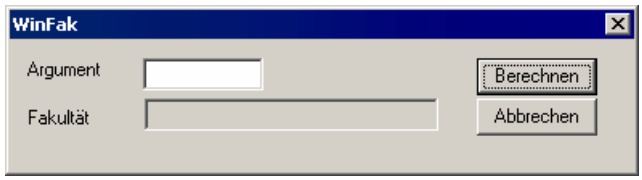


- Verfahren Sie analog mit dem Feld für die Fakultätsanzeige, das vielleicht etwas mehr Platz benötigt. Hier sollen nur Ergebnisse angezeigt werden, so dass wir in der **Eigenschaften**-Dialogbox auf dem Registerblatt **Allgemein** das Steuerelement **deaktivieren** und auch den **Tabstopp** abschalten:



- Die beiden vorgegebenen Schalter können wir verwenden, wobei die Eigenschaften des oberen Schalters angepasst werden sollten (neue **ID** und neuer **Titel**):

- Nun wird die Dialogbox noch auf das rechte Maß gebracht:

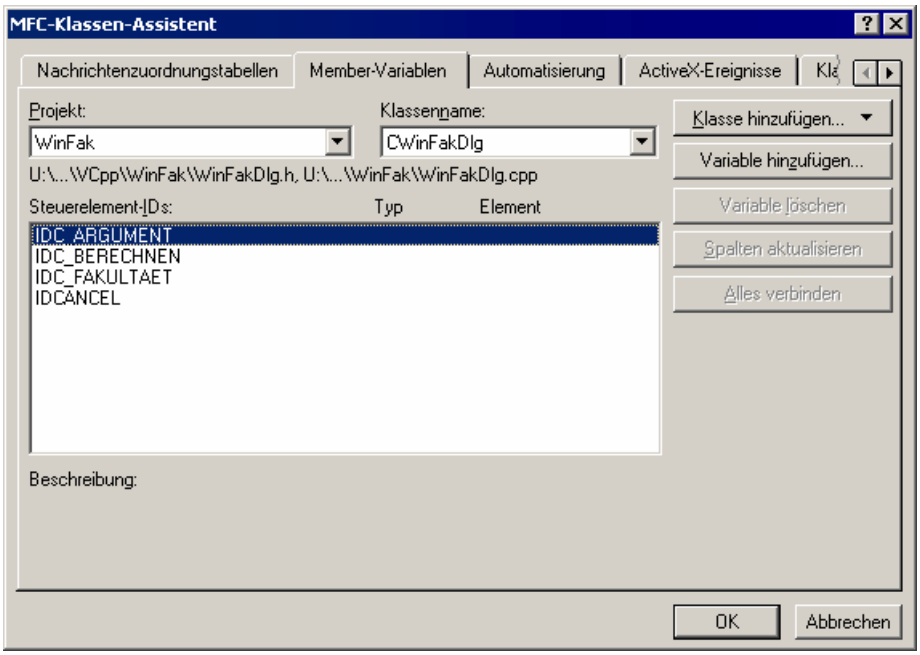


2.8.3 Der Beitrag des Klassenassistenten

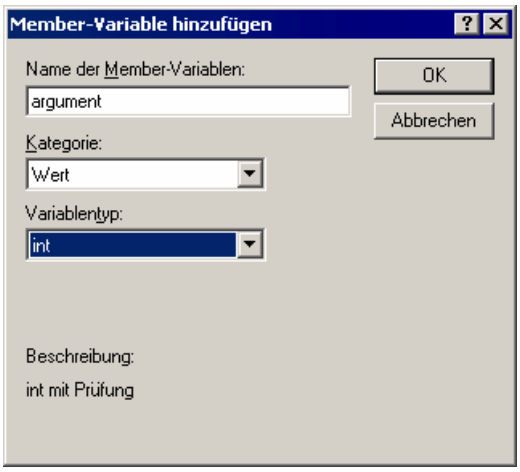
Beim Mausklick auf **Berechnen** soll aus dem Inhalt des Textfeldes `IDC_ARGUMENT` (als ganze Zahl interpretiert) die Fakultät berechnet und im Textfeld `IDC_FAKULTAET` angezeigt werden.
Dazu werden geeignete Variablen benötigt, die mit den beiden Eingabefelder verknüpft werden können. Diese Variablen erzeugen wir mit dem Klassenassistenten, dessen Name erst später voll plausibel sein wird. Er erscheint nach **Strg+W** bzw.

Ansicht > Klassenassistent

Wir wählen das Registerblatt **Member-Variablen**, markieren die Steuelement-ID `IDC_ARGUMENT` und klicken auf **Variable hinzufügen**.

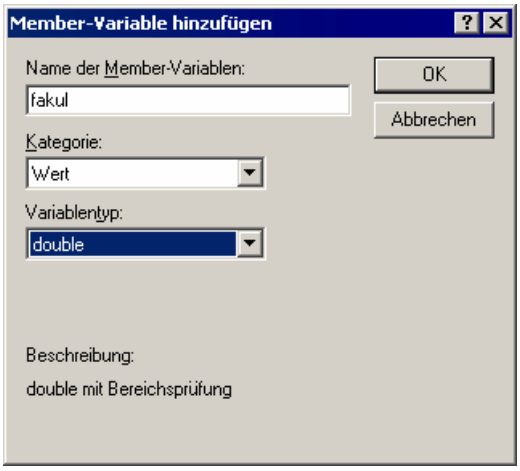


Weil als Argumente nur ganze Zahlen in Frage kommen, stellen wir **int** als Variablentyp ein. Zwar gäbe es bessere Alternativen, doch sollen technische Details momentan soweit als möglich vermieden werden. Auch kümmern wir uns noch nicht um die Konventionen, die C++ - Programmierer unter Windows bei der Wahl von Variablennamen beachten, sondern übernehmen den Namen aus der Konsolen-Version unseres Beispielprogrammes:

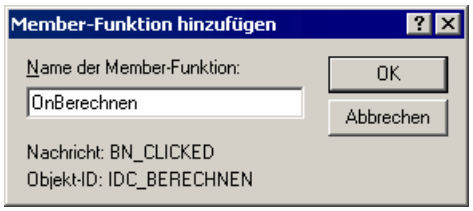


Für die Ganzzahlvariable dürfen wir anschließend im Klassen-Assistenten noch einen Wertebereich vereinbaren, der bei der Übernahme von Werten aus dem Eingabefeld `IDC_ARGUMENT` automatisch überwacht wird (Komfort!). Im Hinblick auf den Wertebereich der geplanten **double**-Ergebnisvariablen eignen sich die Grenzen 0 und 170.

Fügen Sie zum Ergebnis-Steuerelement eine reellwertige Variable vom Typ **double**- hinzu, für die wir keine Bereichskontrolle benötigen:



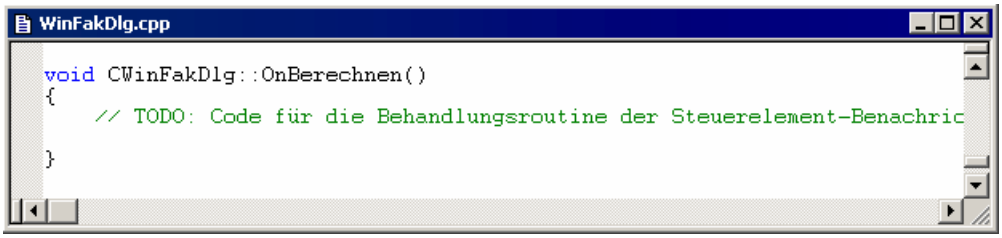
Wechseln Sie nun zum Registerblatt **Nachrichtenzuordnungstabellen**, markieren Sie die **Objekt-ID** `IDC_BERECHNEN` sowie die Nachricht `BN_CLICKED`. Wir wollen nämlich nun kodieren, was passieren soll, wenn der Benutzer auf den Schalter **Berechnen** klickt.
Nach **Funktion hinzufügen** können wir den Namen einer Routine wählen, die in diesem Fall ausgeführt werden soll, z.B.:



Beachten Sie, dass die Groß-/Kleinschreibung in C++ bei allen Bezeichnern signifikant ist!
Um zum Quellcode für die neue Routine zu springen, können Sie z.B. einen Doppelklick auf den neuen Eintrag unter **Member-Funktion** setzen.

2.8.4 Unser Beitrag: Eine Ereignisbehandlungsroutine

Zur Vervollständigung unserer Anwendung sind nur noch einige Zeilen C++ von Nöten:



Wir ersetzen den vom Anwendungsassistenten hinterlassenen Kommentar durch Anweisungen, die stark an die Konsolenvariante des Programms erinnern:

```
void CWinFakDlg::OnBerechnen()
{
  if (UpdateData(TRUE))
  {
    fakul = 1.0;
    for (int i = 1; i <= argument; i++)
      fakul *= i;
    UpdateData(FALSE);
  }
}
```

Neu ist nur die Anweisung **UpdateData()**. Sie dient zum Datenaustausch zwischen den Eingabefeld-Steuerelementen und den korrespondierenden Variablen, wobei der zwischen runden Klammern angegebene Parameter die Richtung vorgibt. Im Kern der Routine steckt die selbe **for**-Schleife wie in der Konsolenanwendung.

Nun sollte sich Ihr **WinFak**-Programm erstellen lassen und auch brauchbare Arbeit leisten. Mathematiker werden bemängeln, dass initial zum Argument 0 das Ergebnis 0 angezeigt wird, und erst ein Klick auf **Berechnen** die korrekte Fakultät 1 produziert. Um das Problem zu lösen, müssten wir im Quellcode, den die Assistenten erzeugt haben, verändern. Mit dem jetzigen Kenntnisstand ist dies wenig sinnvoll, am Ende des Kurses wird es Ihnen jedoch keinerlei Schwierigkeiten mehr bereiten.

2.9 Hinweise zu den Dateien eines VC++ - Projektes

Falls Sie schon jetzt einige der zahlreichen Dateien kennen lernen möchten, die Visual C++ 6 im Arbeitsbereichs- bzw. Projektordner anlegt, hilft Ihnen vielleicht die folgende Tabelle weiter:

Extension	Bedeutung	Erläuterung
cpp	C++ - Quelltextdatei	
h	Header-Datei	In Header-Dateien werden vor allem Definitionen von Datentypen, Funktionen, Konstanten usw. abgelegt, vor allem dann, wenn diese von mehreren Programmen verwendet werden sollen.
dsp	Developer Studio Project File Projekterstellungsdatei	Enthält Informationen über ein Projekt. Das Projekt kann über die DSP-Datei geöffnet werden. Es ist allerdings sinnvoller, ein Visual C++ - Projekt über den zugehörigen Arbeitsbereich zu öffnen.

Extension	Bedeutung	Erläuterung
dsw	Developer Studio Workspace File Projektarbeitsbereichsdatei	Enthält Informationen über einen Arbeitsbereich, der aus mehreren Projekten bestehen kann (, meist aber nur ein Projekt enthält). Ein Arbeitsbereich wird über seine DSW-Datei geöffnet.
obj	Kompilate, die noch gebunden werden müssen	
rc	Ressourcen-Skriptdatei (im Textformat) für den Ressourcen-Compiler	Diese Datei wird von den Ressourcen-Editoren erstellt und sollte nicht manuell editiert werden. Im **WinFak**-Beispiel enthält die Datei **WinFak.rc** z.B. das Ergebnis unserer Bemühungen im Dialogboxdesigner.
res	Vom Ressourcen-Compiler erstellte binäre Ressourcen-Datei, die vom Linker verarbeitet wird	

Außerdem erstellt die Entwicklungsumgebung noch etliche temporäre Dateien.

Auf der Suche nach freiem Festplatten-Speicherplatz dürfen Sie die oft sehr voluminösen Debug- bzw. Release-Unterverzeichnisse zu VC++ - Projekten bedenkenlos löschen, weil diese bei Bedarf aus den übrigen Dateien erstellt werden können.

In der klassischen C-Programmierung existiert zu jedem Projekt ein **Makefile** mit allen Compiler- und Linker-Aufrufen zum Erstellen des Endproduktes.
Diese oft recht umfangreiche Datei enthält z.B. alle Abhängigkeiten zwischen den einzelnen Quelltextdateien des Projektes.
In einem Visual C++ - Projekt wird ihre Aufgabe von der DSP-Datei übernommen. Man kann jedoch mit

Projekt > Makefile exportieren

ein klassisches Makefile erstellen und mit dem Hilfsprogramm **Nmake** ausführen lassen.

2.10 Übungsaufgaben zu Abschnitt 2

1) Erstellen Sie in VC++ ein neues Projekt für das weltberühmte „Hallo, world“ - Programm von Kernighan & Ritchie (1988):

```
#include <stdio.h>

void main()
{
   printf("Hallo, world!\n");
}
```

2) Entfernen Sie im Konsolen-Fakultätsprogramm die **#include**-Anweisung, und lassen Sie anschließend den Rest kompilieren. Welche „externen“ Bestandteile sind dem Compiler nun unbekannt?

3) Wie groß ist die Release-Version der Datei **WinFak.exe**?
4) Welche Aussage ist richtig?

a) Um Objektcode zu erzeugen, benötigt man eine objektorientierte Programmiersprache.
b) Der Quellcode wird vom Compiler in Objektcode übersetzt.

3 Elementare Bestandteile von C++

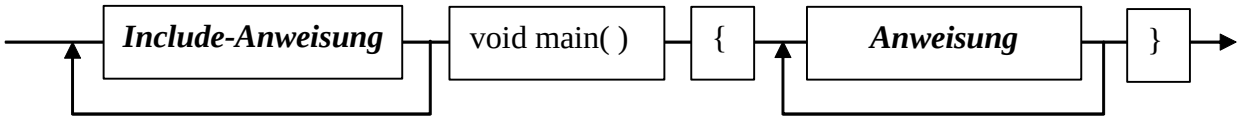
3.1 Einstieg

Dieser Abschnitt liefert Informationen, die von Anfang erforderlich oder nützlich beim Erstellen der Beispiel- bzw. Übungsprogramme sind, mit denen wir die grundlegenden Bestandteile von C++ kennen leinen wollen.

3.1.1 Struktur eines einfachen C++ - Programms

Das folgende **Syntaxdiagramm** beschreibt die Struktur eines einfachen C++ - Programms, das zum Üben der grundlegenden Sprachelemente einen geeigneten Rahmen bietet. Der Begriff „einfaches C++ - Programm“ ist nicht offiziell definiert, sondern besitzt nur manuskript-interne Gültigkeit.

Einfaches C++ - Programm



Per Syntaxdiagramm lässt sich für Bestandteile von Programmiersprachen auf anschauliche und präzise Weise beschreiben, wie zulässige Realisationen gebildet werden. In den Rechtecken stehen konstante oder variable Elemente, die Pfeile definieren die erlaubten Übergänge.
Für die Syntaxdiagramme in diesem Manuskript sollen folgende Vereinbarungen gelten:

- Für Sprachbestandteile, die exakt in der angegebenen Form geschrieben werden müssen, wird der Standard-Zeichensatz verwendet.
- Platzhalter sind durch ***fett-kursive*** Schrift gekennzeichnet.

Ausführlichere Erläuterungen zu Syntaxdiagrammen sparen wir uns in der Erwartung, dass sich Beispiele und Syntaxdiagramme gegenseitig abstützen und gemeinsam eine klare Vorstellung vermitteln werden.

Mit der gleich näher zu erläuternden Include-Anweisung kann eine andere Quellcode-Datei eingefügt werden. Man verwendet sie meist dazu, Header-Datei zu Bibliotheken einzubinden. Je nach Aufgabenstellung des Programms werden eventuell auch mehrere Header-Dateien benötigt.

Ein C++ - Programm besteht im wesentlichen aus **Funktionen**, die jeweils eine Teilaufgabe übernehmen. Es muss auf jeden Fall eine Funktion mit dem Namen **main()** vorhanden sein, die beim Programmstart automatisch aufgerufen wird. Unsere ersten Programme beschränken sich der Einfachheit halber auf diese **Hauptfunktion**.
Jede Funktion besteht aus

- **Kopf**
  Im Kopf wird zunächst der Typ des Rückgabewertes angegeben. Bei dem in obigem Syntaxdiagramm vorgeschlagenen Einfachprogramm wird zur Hauptfunktion der Pseudo-Typ **void** angegeben. Damit wird festgestellt, dass die Funktion *keinen* Rückgabewert liefert. Dem Funktionsnamen folgt ein Paar runder Klammern, die optional Parameter zur Steuerung der Funktion enthalten dürfen, wovon bei der Hauptfunktion des Einfachprogramms kein Gebrauch gemacht wird.
- **Rumpf**
  Im Rumpf der Funktion ist durch eine (beliebig lange) Serie von **Anweisungen** der Algorithmus zur Lösung ihrer Aufgabe realisiert. Der Anweisungsblock einer Funktion wird durch geschweifte Klammern begrenzt.

Eine **Anweisung** ist die kleinste ausführbare Einheit eines Programms. In C++ sind bis auf wenige Ausnahmen alle Anweisungen mit einem Semikolon abzuschließen.

Zur **Formatierung** von C++ - Programmen haben sich Konventionen entwickelt, die wir bei passender Gelegenheit besprechen werden. Der Compiler ist hinsichtlich der Formatierung sehr tolerant und beschränkt sich auf folgende Regeln:

- Die einzelnen Bestandteile einer Deklaration oder Anweisung müssen in der richtigen Reihenfolge stehen.
- Zwischen zwei **Sprachbestandteilen** muss im Prinzip ein Trennzeichen stehen, wobei das Leerzeichen, das Tabulatorzeichen und der Zeilenumbruch erlaubt sind. Diese Trennzeichen dürfen sogar in beliebigen Anzahlen und Kombinationen auftreten.
  Zeichen mit festgelegter Bedeutung wie z.B. ";", "(", "+", ">" sind *selbstbegrenzend,* d.h. vor und nach ihnen sind keine Trennzeichen nötig (aber erlaubt).

3.1.2 Include-Anweisung, Header-Dateien, Präprozessor

Die **#include** – Anweisung richtet sich *nicht* an den Compiler sondern an den so genannten **Präprozessor**, der angewiesen wird, die in der angegebenen Include-Datei enthaltenen Anweisungen in den aktuellen Quellcode einzufügen.

In der Regel werden auf diese Weise Typ-, Klassen- und Funktionsdefinitionen von **Objektcode-Bibliotheken** verfügbar gemacht, die in so genannten **Header-Dateien** untergebracht sind. Wir werden uns später noch ausführlich mit den in Header-Dateien niedergelegten **Schnittstellen-Definitionen** zu Funktionen, Typen und Klassen befassen.

Die Namen der Header-Dateien zu **Standardbibliotheken**, die allen C/C++ - Implementationen gemeinsam sein sollten und vom Präprozessor in einem speziellen Verzeichnis erwartet werden, gibt man dabei zwischen spitzen Klammern an. Im Einleitungsbeispiel wird auf diese Weise die **iostream**-Standardbibliothek integriert, die für einfache Ein- und Ausgabeoperationen bei Konsolenanwendungen zuständig ist (s.u.):

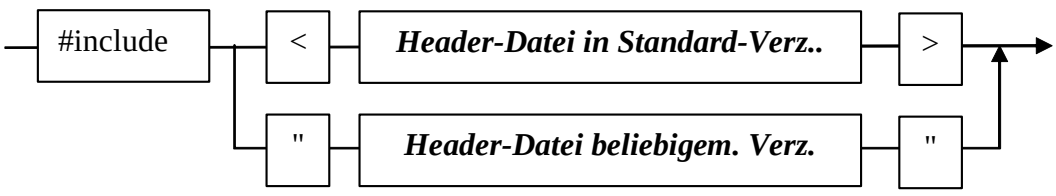
```
#include <iostream.h>
```

Bei Visual C++ 6 finden Sie die Standard-Header-Dateien vom Installations-Verzeichnis (z.B. C:\Programme\Microsoft Visual Studio) ausgehend in:

VC98\Include

Man kann mit der Entwicklungsumgebung noch weitere Standardverzeichnisse vereinbaren, die durchsucht werden sollen, wenn eine Header-Datei wie in obigem **#include**-Beispiel angesprochen wird (siehe unten).
Um eine Header-Datei aus einem *beliebigem* Verzeichnis anzusprechen, muss ihr Name durch Anführungszeichen (doppelt, hochgestellt) begrenzt werden.

Das Syntaxdiagramm für eine Include-Anweisung:



Der Präprozessor versteht übrigens neben **#include** noch andere Anweisungen, die wir später kennen lernen werden. Allen ist gemeinsam, dass sich im Unterschied zu den C++ - Anweisungen **nicht** durch ein Semikolon abgeschlossen werden.
Außerdem filtert der Präprozessor auch noch die Kommentare aus dem Quelltext und übergibt somit dem Compiler eine einerseits komplettierte und andererseits entschlackte Quellcode-Version.

3.1.3 Einfache Ein- und Ausgabe bei Konsolenanwendungen

Im Einleitungsbeispiel wird per **#include**-Befehl die **iostream**-Standardbibliothek integriert. Dort sind folgende Objekte für einfache Ein- und Ausgabeoperationen bei Konsolenanwendungen deklariert[1]:

- Über das Objekt **cin** und den Operator >> hat unser Programm Zugang zur **Standardeingabe**. Dies ist in der Regel die Tastatur.[2]
  Die Anweisung:
  ```
  cin >> argument;
  ```
  bedeutet ungefähr: „Speichere die vom Anwender per Tastatur eingegebene Zahl in der Variablen `argument`."
- Die mit dem Operator << an das Objekt **cout** gesandten Daten landen in der **Standardausgabe**.[2] Dies ist in der Regel der Bildschirm.
  Wie die folgende Zeile
  ```
  cout << "Fakultaet: " << fakul << endl;
  ```
  des Einleitungsbeispiels zeigt, kann man dem Objekt **cout** schicken:
  - Zeichenketten (in doppelte Anführungszeichen eingeschlossen)
  - Variablen
  - über das Schlüsselwort **endl** („end of line") ein Steuerzeichen für Zeilenwechsel

3.1.4 Kommentare

C++ bietet zwei Möglichkeiten, den Quelltext zu kommentieren:

- Alle Zeichen von „//" bis zum Ende der Zeile gelten als Kommentar, wobei kein Terminierungszeichen erforderlich ist. Von dieser Möglichkeit wurde im Einleitungsbeispiel Gebrauch gemacht:
  ```
  // Einleitungsbeispiel: Berechnung der Fakultät
  ```
- Zwischen einer Einleitung durch „/*" und einer Terminierung durch „*/" kann sich ein ausführlicher Kommentar auch über mehrere Zeilen erstrecken. Diese Möglichkeit wird gerne dazu genutzt, größere Programm-Passagen zu deaktivieren (etwa zu Testzwecken), z.B.:
  ```
  /*
  Das folgende "Programm" benötigt keine Bibliotheksroutinen
  und kommt daher ohne #include-Anweisung aus.
  Ein langer Kommentar vergrößert übrigens nicht das
  resultierende Programm, weil er schon vom so genannten
  Präprozessor ausgefiltert wird, den Compiler also gar nicht erreicht.
  */
  ```

---

[1] Wen die Einsprengsel von objektorientierter Terminologie beunruhigen, der kann sich auf die offensichtliche Funktionalität und die einfache Syntax der neuen Sprachelemente konzentrieren.

[2] Erfahrene C-Programmierer werden eventuell einwenden, die Operatoren >> und << seien doch für das bitweise Verschieben zuständig (s.u.). Zu deren Beruhigung soll hier erwähnt werden, dass C++ das so genannte Überladen von Operatoren erlaubt. Ein Operator kann also im Zusammenhang mit einer bestimmten Klasse eine neue Bedeutung erhalten (siehe z.B. Eckel, 2000, S. 99ff).

3.1.5 Bezeichner

Für Konstanten, Variablen, Typen, Objekte, Klassen, Funktionen und sonstige Elemente eines C++ - Programms, die Sie jetzt noch nicht kennen müssen, benötigen wir eindeutige Namen, für die folgende Regeln gelten:

- Das erste Zeichen muss ein Buchstabe ('A' ... 'Z', 'a' ... 'z') oder der Unterstrich sein, danach dürfen außerdem auch Ziffern auftreten.
- Die Groß-/Kleinschreibung ist signifikant.
  Für den C++ - Compiler sind also z.B.

  ```
  Anz  anz ANZ
  ```

  grundverschiedene Bezeichner.
- Bezeichner müssen innerhalb ihres Gültigkeitsbereichs (s.u.) eindeutig sein.
- Für die Eindeutigkeit eines Bezeichners sind nur die ersten *k* Zeichen signifikant, wobei die Anzahl *k* vom Compiler abhängt. Microsofts C++ - Compiler arbeitet mit 247 signifikanten Zeichen. Dabei ist noch zu beachten, dass ein C++ - Compiler viele Bezeichner um Typinformationen erweitert. Insgesamt können bei Visual C++ aber reichlich viele Zeichen vom Programmierer genutzt werden, so dass keine praxisrelevante Längenbeschränkung vorliegt.
- Reservierte Wörter dürfen nicht als Bezeichner verwendet werden, z.B.: **int**, **void**. Wenn Sie versehentlich eines der ca. 70 C++ - Schlüsselwörter als Bezeichner verwenden, wird sich der Compiler beschweren, z.B.:

  ```
  error C2059: Syntaxfehler : 'public'
  ```

  Der Visual C++ - Editor stellt reservierte Wörter übrigens per Voreinstellung in blauer Farbe dar, so dass Sie frühzeitig sofort auf die Kollision aufmerksam werden.
- Außerdem sind reserviert und damit zu vermeiden:
  - Bezeichner im globalen Gültigkeitsbereich (siehe unten), die mit einem Unterstrich beginnen, z.B.

    ```
    _global
    ```
  - Bezeichner, die mit einem Unterstrich (_) beginnen, dem ein Großbuchstabe folgt, z.B.:

    ```
    _BesserNicht
    ```
  - Bezeichner, die eine Sequenz aus zwei Unterstichen enthalten, z.B.

    ```
    lass__das
    ```

  Am besten vermeiden sie grundsätzlich doppelte sowie einleitende Unterstriche.
- Bei Bezeichnern, die aus mehreren Wörtern bestehen, können Sie einen (einzelnen!) Unterstrich oder Groß-/Kleinschreibung verwenden, um die Lesbarkeit zu verbessern, z.B.:

  ```
  minBestand;
  min_bestand;
  ```

3.2 Variablen und Standardtypen

Jedes nicht-triviale Programm muss zahlreiche Daten im Hauptspeicher aufbewahren und verändern, wozu Programmiersprachen so genannte *Variablen* zur Verfügung stellen:
Eine Variable erlaubt über ihren Namen den lesenden und/oder schreibenden Zugriff auf einen gespeicherten Wert im Hauptspeicher. Um die Details bei der Verwaltung der Variablen im Hauptspeicher müssen wir uns nicht kümmern, da wir schließlich mit einer problemorientierten Programmiersprache arbeiten. Dennoch verlangt C++ beim Umgang mit Variablen im Vergleich zu anderen Programmier- oder Skriptsprachen einige Sorgfalt, die entscheidend zur Vermeidung von Fehlern beiträgt:

- Variablen müssen **explizit deklariert** werden.
  Wenn Sie versuchen, eine nicht-deklarierte Variable zu verwenden, beschwert sich der Compiler, z.B.:
  ```
  error C2065: 'argument' : nichtdeklarierter Bezeichner
  ```
  Durch den Deklarationsswang werden z.B. Programmfehler wegen falsch geschriebener Variablennamen verhindert.
- Jede Variable besitzt einen **Datentyp**, der die erlaubten Werte sowie die zulässigen Operationen festlegt. Damit kann der Compiler viele Programmierfehler aufdecken.

Im Einleitungsbeispiel wird z.B. die Variable `argument` vom Typ **int** deklariert, der ganze Zahlen im Bereich von –2147483648 bis 2147483647 unterstützt:

```
int argument = 1;
```

Bei der Definition erhält die Variable `argument` auch gleich einen Initialisierungswert. Auf diese oder andere Weise sollten Sie unbedingt jeder Variablen einen Wert zuweisen, bevor Sie zu ersten Mal lesend darauf zugreifen.

Als relevante Eigenschaften einer C++ - Variablen halten wir noch einmal fest:

- **Name**
- **Typ**
  Damit sind festgelegt: Wertebereich, zulässige Operationen, Speicherplatzbedarf
- **aktueller Wert**
- **Ort im Hauptspeicher**
  Für spezielle Zwecke bietet C++ die Möglichkeit, die Speicheradresse einer Variablen zu ermitteln. Insbesondere beim Einsatz von dynamischen Datenstrukturen von problemadäquater Größe kommt der Verwaltung von Speicheradressen eine erhebliche Bedeutung zu.

3.2.1 Standard-Datentypen

In Ihren Visual C++ - Programmen können Sie zahlreiche **Standard-Datentypen** verwenden, u.a. (siehe Online-Hilfe):

Typ	Beschreibung	Werte	Speicher-bedarf in Bit
int	Speichert ganze Zahlen. Beispiel: `int argument = 1;`	–2147483648 ... 2147483647	32
float	Speichert Fließkommazahlen mit einer Genauigkeit von mindestens 7 Dezimalstellen Beispiel: `float pi = 3.141593f;`	$-3.40282347 \cdot 10^{38}$ … $3.40282347 \cdot 10^{38}$	32 Vorzeichen: 1 Exponent: 8 Mantisse: 2
double	Speichert Fließkommazahlen mit einer Genauigkeit von mindestens 15 Dezimalstellen Beispiel: `double pi = 3.141593;`	$-1.79769313 \cdot 10^{308}$ … $1.79769313 \cdot 10^{308}$	64 Vorzeichen: 1 Exponent: 1 Mantisse: 5

Typ	Beschreibung	Werte	Speicher-bedarf in Bit
char	Speichert ein Textzeichen über seine Ordnungszahl im verwendeten Zeichensatz (ASCII/ANSI in VC++). **char** ist also ein *Ganzzahltyp*! Beispiel: `char zeichen = 'j';`	-128 … 127 Andere Compiler benutzen eventuell den Wertebereich 0 … 255	8
bool	speichert einen Wahrheitswert. Beispiel: `bool cond = false;`	**true**, **false** In der Ausgabe erscheint **true** als 1 und **false** als 0.	8

Zu den beiden Ganzzahl-Typen **int** und **char** gibt es etliche Alternativen, die einen erweiterten Wertebereich bieten oder in speziellen Situationen den Speicherplatz besser nutzen:

Typ	Beschreibung	Werte	Speicher-bedarf in Bit
short int alias **short**	Beim Typ **short int** (oder einfach **short**) werden zum Speichern einer Ganzzahl nur 16 Bit verwendet (**int**: 32 Bit). Beispiel: `short n = -100;`	-32768 ... 32767	16
long int alias **long**	In Visual C++ sind diese Bezeichner synonym zu **int**.		
__int64	Von einigen Compilern (z.B. VC++) wird ein 64-Bit-Ganzzahltyp unterstützt. Er gehört nicht zum ANSI/ISO – Standard.	–9223372036854775808 … 9223372036854775807	64
unsigned char	Viele Ganzzahl-Typen gibt es auch als vorzeichenfreie Varianten, die sich für nicht-negative Werte eignen.	0 … 255	8
unsigned short		0 … 65535	16
unsigned int alias **unsigned long**		0 ... 4294967295	32

Nach obigen Tabellen ist **char** ein Ganzzahltyp mit Werten von -128 bis 127 ist. Die ASCII-Zeichen mit den Nummern 0 bis 127 sind hier gut aufgehoben. Beim so genannten erweiterten ASCII-Code für Umlaute und Sonderzeichen, die Ordnungszahlen zwischen 128 und 255 erhalten, wird es etwas schwierig. Dabei ist der **char**-Wertebereich noch das geringste Problem, weil man leicht zu **unsigned char** wechseln kann. Dummerweise gibt es für die oberen Zeichen konkurrierende ASCII-Code-Erweiterungen: die so genannte OEM-Erweiterung (in mehreren länderspezifischen Varianten) und die ANSI-Erweiterung, die u.a. in Windows verwendet wird.
Wenn wir mit dem Editor der VC++ - Entwicklungsumgebung ein 'Ä' in den Quellcode eintragen, dann sieht der Compiler im ANSI-Sinn das Zeichen mit der Nummer 196. Bei einer Ausgabe in

einer Konsolenanwendung wird dieser Zeichencode aber nicht mehr als 'Ä' interpretiert sondern als waagerechter Strich, weil hier die OEM-Erweiterung des ASCII-Codes in Gebrauch ist. Dieses Phänomen wird in folgendem Beispiel vorgeführt:

Quellcode	Ausgabe

`#include <iostream.h>`
 	
`void main()`
`{`
`  cout << 'Ä' << endl;`
`}`
	–

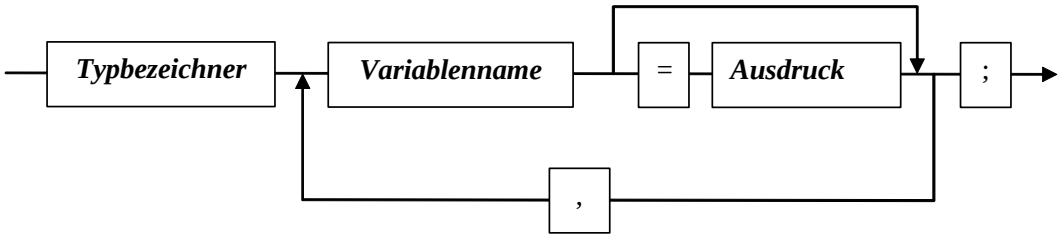
Über geeignete Datentypen bzw. Klassen lässt sich das Problem aber lösen, und spätestens bei der Windows-Programmierung wird es kein Thema mehr sein. Im Rahmen einer Übungsaufgabe wird auch für Konsolenprogramme eine Lösungsmöglichkeit vorgestellt.

Neben den in diesem Abschnitt vorgestellten Standard-Datentypen werden wir u.a. noch zusammengesetzte Typen (z.B. Felder) und so genannte Zeigertypen kennen lernen.
Als besonders wichtig bei der Entwicklung von problemadäquaten Softwaresystemen wird sich die Möglichkeit zur Definition von benutzereigenen Datentypen erweisen. Darauf gehen wir im Zusammenhang mit Strukturen und Klassen intensiv ein.

3.2.2 Variablendeklaration und -definition

Obwohl Sie die Regeln für die **Variablendeklarations-Anweisung** sicher leicht aus den bisherigen Beispielen abstrahieren können, wird ein Syntaxdiagramm dazu angegeben:

Variablendeklaration



In der Regel wird man eine Variable am Anfang des Blockes (s.u.) deklarieren, in dem sie benötigt wird.

Es hat sich eingebürgert, Variablenbezeichner mit einem Kleinbuchstaben beginnen zu lassen, um sie im Quelltext gut von anderen Bezeichnern (z.B. für benutzerdefinierte Typen oder Konstanten) unterscheiden zu können.

Wir haben übrigens gerade eine erste Variante des C++ - Sprachelements **Anweisung** offiziell eingeführt.

Wird aufgrund einer Variablendeklaration auch Speicher belegt, handelt es sich nach der offiziellen Sprachregelung um eine *Variablendefinition*. Die beiden Begriffe Deklaration und Definition werden in der Literatur zu C/C++ oft mehr oder weniger synonym verwendet, obwohl es einen klar festgelegten Bedeutungsunterschied gibt:

- Bei einer **Deklaration** wird lediglich die Bedeutung eines Bezeichners festgelegt. Er wird dem Compiler bekannt gemacht.
- Bei einer **Definition** wird zusätzlich zur Deklaration auch Speicherplatz belegt (bei Variablen) bzw. Objektcode erzeugt (bei Funktionen).

Die meisten Variablendeklarationen auch sind auch Definitionen. Bei *Funktionen* sind reine Deklarationen sehr viel häufiger anzutreffen. Meist werden sie in Header-Dateien konzentriert.
Eine Variable bzw. Funktion darf nur einmal definiert, aber beliebig oft (konsistent) deklariert werden.
Auch für das vorliegende Manuskript kann kein perfekter Gebrauch der Begriffe Definition und Deklaration garantiert werden, weil es gelegentlich zu Konflikten kommt zwischen einer vertrauten, leicht verständlichen und einer ungewohnten, exakten Formulierung. Diese Konflikte werden eventuell unterschiedlich gelöst, wobei aber hoffentlich das Verständnis wesentlicher Sachverhalte nie gefährdet ist.

3.2.3 Initialisierung und Wertzuweisung

Neu deklarierte Variablen kann man optional auch gleich **initialisieren**. Spätestens vor dem ersten lesenden Zugriff müssen Sie jeder Variablen einen Wert zuweisen, wenn Sie überflüssigen Ärger vermeiden wollen.
Der Compiler „löst" das Initialisierungsproblem in Abhängigkeit vom Gültigkeitsbereich einer Variablen (s.u.) folgendermaßen:

- Globale Variablen erhalten als Initialisierungswert:
  - 0 (numerische Variablen)
  - **false** (boolsche Variablen)
  - Leerzeichen (bei Typ **char**)
- Lokale Variablen besitzen einen zufälligen Wert.

Dieses Verhalten wird durch folgendes Programm demonstriert[1]:

```
#include <iostream.h>

int globVar;

void main()
{
   int lokVar;

   cout << "Initial.wert f. die globale Integer-Variable: " << globVar << endl;
   cout << "Initial.wert f. die lokale  Integer-Variable: " << lokVar << endl;
}
```

Seine Ausgabe:

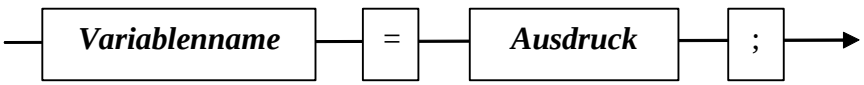
```
Initial.wert f. die globale Integer-Variable: 0
Initial.wert f. die lokale  Integer-Variable: -858993460
```

Der Compiler warnt nur bei *lokalen* Variablen vor dem gefährlichen Lesezugriff ohne vorherige Initialisierung:

```
warning C4700: Lokale Variable 'lokVar' wurde ohne Initialisierung verwendet
```

Daraus ergibt sich die Empfehlung, globale Variablen zu vermeiden. In einem einfachen C++ - Programm im Sinn von Abschnitt 3.1.1 sind sie u.a. aus diesem Grund nicht vorgesehen.

Wertzuweisungen zählen zu den einfachsten und am häufigsten benötigten Anweisungen:



[1] Wegen der Verwendung einer globalen Variablen handelt es sich *nicht* um ein „einfaches C++ - Programm" im Sinne von Abschnitt 3.1.1. Mit der dortigen Definition sollte ja auch nicht die Fülle der Möglichkeiten berücksichtigt, sondern ein möglichst einfacher Rahmen für Übungszwecke vorgeschlagen werden.

Ein Beispiel (aus dem Konsolen-Fakultätsprogramm):

```
fakul = 1.0;
```

Mit Ausdrücken sowie einigen wichtigen Ergänzungen zum Thema *Wertzuweisung* werden wir uns später befassen.

3.2.4 Blöcke und Gültigkeitsbereiche

In C++ werden mit Hilfe der geschweiften Klammern Anweisungsblöcke definiert, die jeweils auch einen Gültigkeitsbereich für Variablen bilden. Eine blockintern deklarierte (**lokale**) Variable verliert ihre Existenz beim Verlassen des Blocks, wobei der belegte Speicher freigegeben wird.

Eine „blockfrei" deklarierte (**globale**) Variable ist an jeder Stelle des Programms verfüg- und veränderbar, was für Probleme bei der Fehlersuche führen kann und ein wesentlicher Grund für den schlechten Ruf der globalen Variablen bei erfahrenen Programmierern ist. In einem einfachen C++ - Programm nach dem Vorschlag aus Abschnitt 3.1.1 sind sie daher nicht vorgesehen. Das folgende Programm hält sich nicht an den Vorschlag:

Quellcode	Ausgabe

`#include <iostream.h>`
 	
`int global = 1;`
 	
`void main()`
`{`
`  int lokal = 2;`
`  global = 2;`
`  cout << global << " " << lokal << endl;`
`}`
	`2 2`

Anweisungsblöcke dürfen hierarchisch geschachtelt werden. Treten dabei auf mehreren Stufen Variablen mit identischem Namen auf, so dominiert bei Zugriffen die „lokalste" Definition, z.B.:

Quellcode	Ausgabe

`#include <iostream.h>`
 	
`void main()`
`{`
`  int wert = 1;`
`  {`
`  int wert = 2;`
`  cout  << "Wert = " << wert << endl;`
`  }`
`}`
	`Wert = 2`

Die im letzten Beispiel innerhalb des Anweisungsblocks zur Funktion **main()** vorgenommene Blockbildung wirkt recht unmotiviert. Wir werden bald im Zusammenhang mit Wiederholungen und bedingten Anweisungen sinnvolle Anlässe für funktionsinterne Blöcke kennen lernen.

Ein Anweisungsblock wird auch als **Verbundanweisung** bezeichnet und darf stets an die Stelle einer einfachen Anweisung gesetzt werden.

Bei der übersichtlichen Gestaltung von C++ - Programmen ist das Einrücken von Anweisungsblöcken (s.u.) besonders wichtig. Im Visual C++ - Quelltexteditor kann ein markierter Block aus mehreren Zeilen mit

<Tab> nach rechts eingerückt

und mit

<Umschalt>+<Tab> nach links ausgerückt

werden.

Sobald Header-Dateien eingebunden werden, ist der globale Namensraum auch dann recht bevölkert, wenn wir keine globalen Variablen verwenden, weil in den Header-Dateien im Allgemeinen zahlreiche Funktionen deklariert werden. Um die bei Verwendung mehrerer Header-Dateien drohenden Namenskollisionen möglichst zu vermeiden, wurden mit C++ so genannte eingeführten Namensräume vorgestellt, mit denen wir uns in Abschnitt 3.4.2 beschäftigen.

3.2.5 Konstanten

In der Regel sollten auch die im Programm benötigten konstanten Werte (z.B. für den Mehrwertsteuersatz) in einer Variablen abgelegt und im Quelltext über ihren Variablennamen angesprochen werden. Vorteile:

- Bei einer späteren Änderung des Wertes ist nur eine einzige Quellcode-Zeile betroffen.
- Bessere Lesbarkeit des Quellcodes

Beispiel:

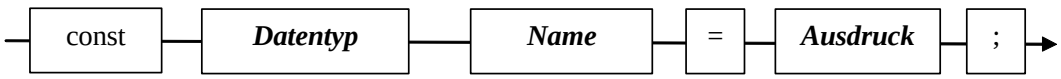
Quellcode	Ausgabe

`#include <iostream.h>`
 	
`void main()`
`{`
`  const double MWST = 1.16;`
`  double netto = 100.0, brutto;`
`  brutto = netto * MWST;`
`  cout << "Brutto: " << brutto << endl;`
`}`
	`Brutto: 116`

Um sicher zu stellen, dass sich der Wert einer so eingesetzten Variablen nach der Initialisierung nicht mehr ändert, setzt man den Typqualifizierer **const** an den Anfang der Deklaration.
Das Syntaxdiagramm zur Deklaration einer Konstanten ergibt sich also aus dem Syntaxdiagramm zur Variablendeklaration durch zwei Veränderungen:

- Es wird das Schlüsselwort **const** an den Anfang gesetzt.
- Die Variable muss natürlich initialisiert werden.

Deklaration einer Konstanten



Viele Programmierer verwenden für die Konstanten-Bezeichner nur Großbuchstaben, um sie im Quelltext gut von Variablennamen unterscheiden zu können.

Bevor der Typqualifizierer **const** in die C++ - Sprachdefinition aufgenommen wurde, nutzte man die Präprozessor-Anweisung **#define**, um Konstanten zu vereinbaren, z.B.:

```
#define MWST 1.16
```

Hier wird der Präprozessor angewiesen, jedes nachfolgende Vorkommen des Bezeichners „MWST“ durch die Zeichenfolge „1.16“ zu ersetzen. Im Unterschied zur Konstantendefinition (siehe Abschnitt 3.2.5) wird lediglich (vor dem Auftritt des eigentlichen Compilers) eine Zeichenfolge durch eine andere ersetzt, so dass auch keine Typdeklaration erfolgen muss.
Man sollte von der moderneren Möglichkeit Gebrauch machen und eine konstante Variable definieren, weil dann eine Typinformation vorhanden ist, und der Debugger den Namen der Konstanten kennt. In vorhandener Software ist die Präprozessor-Anweisung **#define** jedoch außerordentlich häufig anzutreffen, z.B. in den Header-Dateien für die Windows-Programmierung mit C/C++.

Von der Argumentation gegen globale Variablen (siehe z.B. Abschnitt 3.2.4) sind *globale Konstanten* nicht betroffen.

3.2.6 Literale

Die im Programmcode auftauchenden expliziten Werte bezeichnet man als *Literale*. Wie Sie aus dem Abschnitt 3.2.5 wissen, ist es oft sinnvoll, Literale innerhalb von Konstanten-Deklarationen zu verwenden, z.B.:

```
const double MWST = 1.16;
```

Auch die Literale besitzen in C++ stets einen Typ, wobei in Bezug auf die Standardtypen einige Regeln zu beachten sind:

3.2.6.1 Ganzzahl-Literale

Ganzzahlige Literale von –2147483648 bis 2147483647 haben in Visual C++ den Typ **int**, anderenfalls den Typ **__int64**. Sie können im Wesentlichen gemäß mathematischen Gepflogenheiten verwandt werden, wenngleich einige EDV-typische Feinheiten bei der Notation zu beachten sind. So liefern die harmlos wirkenden Anweisungen:

```
const int i = 11, j = 011;
cout << "i = " << i << ", j = " << j << endl;
```

ein aus mathematischer Sicht verblüffendes Ergebnis

```
i = 11, j = 9
```

Durch ein anderes Beispiel:

```
const int i = 8, j = 08;
cout << "i = " << i << ", j = " << j << endl;
```

kommen wir dem Rätsel auf die Spur. Hier erhalten wir im Ausgabefenster eine Fehlermeldung des Compilers:

```
error C2041: Ungueltige Ziffer '8' fuer Basis '8'
```

Die führende Null in „011“ ist keinesfalls irrelevant, sondern kündigt dem Compiler einen Wert im *Oktalsystem* an, er interpretiert „011“ als:

$$11_{\text{Oktal}} = 1 \cdot 8 + 1 \cdot 1 = 9$$

Mit der Präfix „0x“ (oder auch „0X“) werden ganzzahlige Werte im *Hexadezimalsystem* ausgedrückt. Die Anweisungen:

```
const int i = 11, j = 0x11;
cout << "i = " << i << ", j = " << j << endl;
```

liefern die Ausgabe:

```
i = 11, j = 17
```

Soll ein Ganzzahl-Literal explizit den Typ **__int64** erhalten, so ist das Suffix *i64* anzuhängen, z.B.:

```
2i64
```

3.2.6.2 Gleitkomma-Literale

Zahlen mit Dezimalpunkt oder Exponent sind in C++ vom Typ **double**, wenn nicht durch den Suffix **F** oder **f** der Datentyp **float** erzwungen wird.
Die folgende Variablendeklaration mit Initialisierung:

```
float test = 3.142;
```

führt daher zu der Warnung:

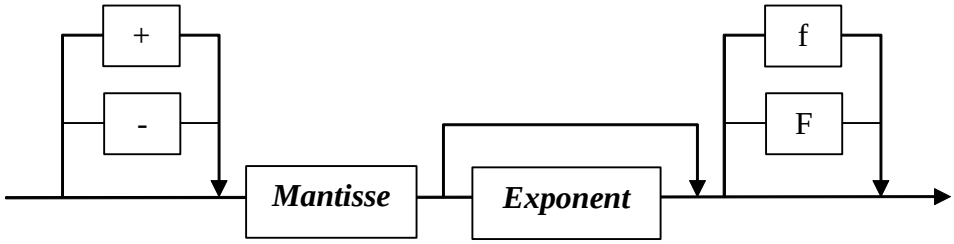
```
warning C4305: 'initializing' : Verkuerzung von 'const double' in 'float'
```

Im Beispiel hätte die Verkürzung allerdings keine nachteiligen Folgen, weil auch der Typ **float** immerhin eine Genauigkeit von mindestens 7 Dezimalstellen bietet. Die Warnung kann aber leicht vermieden werden:

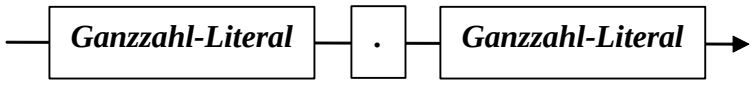
```
float test = 3.142f;
```

Hinsichtlich der Schreibweise von Gleitkomma-Literalen bietet C++ einige Möglichkeiten, von denen die Wichtigsten in folgenden Syntaxdiagrammen dargestellt werden:

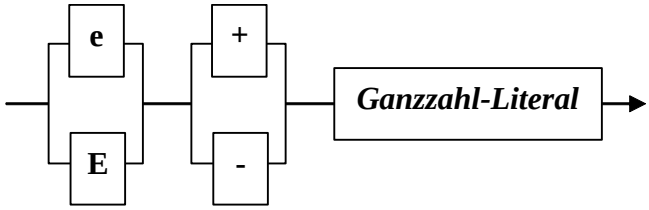
Gleitkomma-Literal



Mantisse



Exponent



Die in Gleitkomma-Literalen auftretenden **int**-Literale werden stets dezimal interpretiert. Die oben berichteten Präfixe für Ganzzahlen im Oktal- bzw. Hexadezimalsystem sind also nicht erlaubt.
Beispiele für gültige **double**-Literale:

```
9.78
-0.45875e-20
```

3.2.6.3 char-Literale

char-Literale werden in C++ mit einfachen Hochkommata eingerahmt. Dabei sind erlaubt:

- **Druckbare Zeichen** (teilweise plattformabhängig)
  Beispiel:
  ```
  'a'
  ```
  Nicht erlaubt: Einfaches Hochkomma und Rückwärts-Schrägstrich („Backslash“)
- **Escape-Sequenzen**
  Hier dürfen einem einleitenden Rückwärts-Schrägstrich folgen:

- Ein Befehls-Zeichen, z.B.:

Alarmton:	`'\a'`
Neue Zeile	`'\n'`
Horizontaler Tabulator	`'\t'`

- Eines der oben aus syntaktischen Gründen verbotenen Zeichen (einfaches Hochkomma und Rückwärts-Schrägstrich), um es auf diese Weise doch angeben zu können:

  ```
  '\\'
  '\''
  ```

- Eine ASCII-Escapesequenz mit hexadezimaler Zeichennummer, z.B.:

  ```
  '\x8e'
  ```

  Wird dieses **char**-Literal in einem Konsolenfenster ausgegeben, so erscheint ein 'Ä', weil dieses Zeichen in der OEM-ASCII-Erweiterung unter der Ordnungszahl 0x8E (hexadezimal!) zu finden ist.

3.2.6.4 Zeichenketten-Literale

Zeichenketten-Literale werden (im Unterschied zu **char**-Literalen) durch *doppelte* Hochkommata begrenzt. Hinsichtlich der erlaubten Zeichen und der Escape-Sequenzen gelten die Regeln für **char**-Literale analog, wobei natürlich das einfache und das doppelte Hochkomma ihre Rollen tauschen.
Beispiel:

```
cout << "Hallo, world!\n";
```

Wie Sie aus Abschnitt 3.1.3 wissen, hätte man im Beispiel den Zeilenwechsel auch über das Schlüsselwort **endl** anfordern können:

```
cout << "Hallo, world!" << endl;
```

Zum Speichern von Zeichenketten kennt C++ keinen eigenständigen Zeichentyp. Man verwendet stattdessen Felder mit dem Basistyp **char**, so dass wir uns mit dem Abspeichern von Zeichenketten bis zur Behandlung von Feldern gedulden müssen.

Im Alltag der MFC-Windows-Programmierung verwendet man allerdings an Stelle der klassischen **char**-Felder eine Klasse namens **CString**, die einen hohen Komfort bietet und auch keine Probleme mit Umlauten hat (vgl. Abschnitt 3.2.1).

3.2.6.5 bool-Literale

Als Literale vom Typ **bool** sollten Sie die beiden reservierten Wörter **true** und **false** verwenden. Allerdings können in der Regel auch **int**-Literale angegeben werden, wobei gilt:

0	≈	**false**
alles Andere	≈	**true**

Beispiel:

```
bool cond = false;
```

3.2.7 Übungsaufgaben zum Abschnitt 3.2

1) Suchen Sie bitte in folgendem Programm die irregulären Literale:

```
#include <iostream.h>

int main()
{
   double d;
   char c;
   bool b;
   d = 0,67;
   c = '\';
   b = 30;

   cout << d << endl << c << endl << b << endl;
   return 0;
}
```

2) Beseitigen Sie bitte alle Fehler in folgendem Programm:

```
#include <iostream.h>

void main()
{
   double umfang
   const double PI = 3,141593;
   cout << "Wie gross ist Ihr Umfang?    ";
   cin >> umfang;

   double radius;
   radius = Umfang/2;
   cout  << "Ihr Flaecheninhalt betraegt:   << PI * radius * radius  << endl;

}
```

3) Was liefert das folgende Programm als Ergebnis A bis D?

```
#include <iostream.h>

void main()
{
   int wert;
   wert = 1;
   cout << "Ergebnis A = " << wert << endl;
   {
     int wert;
     wert = 2;
     cout  << "Ergebnis B = " << wert << endl;
   }
   cout << "Ergebnis C = " << wert << endl;
   {
     int wert;
     wert = 3;
     cout  << "Ergebnis D = " << wert << endl;
   }
}
```

4) In folgendem Programm erfährt die Zahl -100 eine verblüffende Transformation. Welches Wort muss man aus dem Quellcode streichen, um den Unfug zu verhindern?

Quellcode	Ausgabe

`#include <iostream.h>`
 	
`void main()`
`{`
`  const unsigned short MINUS100 = -100;`
`  cout << "MINUS100 = " << MINUS100 << endl;`
`}`
	`MINUS100 = 65436`

5) Schreiben Sie in einem C++ - Konsolenprogramm das Wort „Ärger“ auf den Bildschirm.

3.3 Operatoren und Ausdrücke

Im Zusammenhang mit der Variablendeklaration und der Wertzuweisung haben wir das Sprachelement *Ausdruck* ohne Erklärung benutzt; diese soll nun nachgeliefert werden. Im aktuellen Abschnitt werden wir Ausdrücke als wichtige Bestandteile von C++ - Anweisungen recht detailliert untersuchen. Große Begeisterungsstürme werden dabei wohl nicht aufbrausen, doch im Sinne einer zügigen und fehlerfreien Softwareentwicklung kommen wir an den handwerklichen Grundlagen nicht vorbei.
Schon bei einem Literal oder bei einer Variablen handelt es sich um einen Ausdruck. Mit Hilfe diverser Operatoren entstehen aber noch weit komplexere Ausdrücke.

Die meisten Operatoren verarbeiten *zwei* Operanden und heißen daher **zweistellig** bzw. **binär**.

Beispiel: `a + b`
Der Additionsoperator wird mit dem „+“-Symbol bezeichnet. Er erwartet zwei numerische Argumente.

Manche Operatoren begnügen sich mit einem Argument und heißen daher **einstellig** bzw. **unär**.

Beispiel: `-a`
Die Vorzeichenumkehr wird mit dem „-“-Symbol bezeichnet. Sie erwartet *ein* numerisches Argument.

Wir werden auch noch einen dreistelligen Operator kennen lernen.

Bei jeder gelungenen Operation entsteht ein **Wert** von bestimmtem **Datentyp**.

Beispiel: `2 + 1.5`
Hier resultiert der **double**-Wert 3.5, weil eine Typanpassung „nach oben“ stattfindet.

Als Argumente dürfen auch Funktionsaufrufe und Ausdrücke auftreten, sofern sie einen Wert vom passenden Typ liefern. Damit sind beliebig komplexe Ausdrücke möglich, die zahlreiche Variablen, Literale, Operatoren und Funktionsaufrufe enthalten.

3.3.1 Arithmetische Operatoren

Die arithmetischen Operatoren akzeptieren als Operanden Ganzzahl- oder Gleitkomma-Variablen (bzw. Ausdrücke dieses Typs). Bei binären Operatoren müssen beide Argumente vom selben Typ sein, was der Compiler nötigenfalls durch eine automatische Typanpassung sicherstellt (s.u.).
Dass Computer beim Umgang mit den Grundrechenarten so ihre Eigenarten haben, zeigt das folgende Programm:

Quellcode	Ausgabe

`#include <iostream.h>`
 	
`void main()`
`{`
`  int i = 2, j = 3;`
`  double a = 2.0;`
`  cout << i/j << endl;`
`  cout << a/j << endl;`
`}`
	`0`
	`0.666667`

Es hängt von den Datentypen der Operanden ab, ob die **Ganzzahl**- oder die **Gleitkommarithmetik** zum Einsatz kommt. Der wesentliche Unterschied besteht darin, dass bei der Ganzzahl-Division Nachkommastellen abgeschnitten werden, was gelegentlich durchaus erwünscht ist.

Wie der vom Compiler bei binären Operationen gewählte Arithmetik-Typ und damit auch der Ergebnis-Datentyp von den Datentypen der Argumente abhängen, ist in folgender Tabelle beschrieben:

Datentypen der Operanden		**Datentyp des Ergebniswertes**
Kein Operand hat einen Gleitkommatyp (**float** oder **double**), so dass mit **Ganzzahl-Arithmetik** ausgeführt wird	Kein Operand hat den Typ **__int64**.	**int**
	Mindestens ein Operand hat den Typ **__int64**.	**__int64**
Mindestens ein Operand hat einen Gleitkommatyp, so dass mit **Gleitkomma-Arithmetik** gerechnet wird.	Kein Operand hat den Typ **double**.	**float**
	Mindestens ein Operand hat den Typ **double**.	**double**

Bevor ein ganzzahliges Argument an der Gleitkomma-Arithmetik teilnimmt, wird es automatisch in einen Gleitkommatyp gewandelt.

In der folgenden Tabelle mit allen arithmetische Operatoren stehen ***num***, ***num1*** und ***num2*** für Ausdrücke mit einem numerischen Typ (z.B **char**, **short**, **int**, **long**, **float, double**), ***var*** vertritt eine numerische Variable:

Operator	Bedeutung	Beispiel	
		Programmfragment	**Ausgabe**
-num	**Vorzeichenumkehr**	`int i = -2;` `cout << -i;`	`2`
num1 + ***num2***	**Addition**	`int i = 2, j = 3;` `cout << i+j;`	`5`
num1 – ***num2***	**Subtraktion**	`double a = 2.6, b = 1.1;` `cout << a-b;`	`1.5`
num1 * ***num2***	**Multiplikation**	`int i = 4, j = 5;` `cout << i*j;`	`20`

num1 / ***num2***	**Division**
	Sind beide Argument ganzzahlig, wird eine Ganzzahldivision ausgeführt, ansonsten eine Gleitkommadivision.
		`double a = 4, b = 5;`
		`int i = 4, j = 5;`
		`cout << a/b << endl;`
		`cout << i/j;`
			`0.8`
			`0`

num1 % ***num2***	**Modulo** Das Ergebnis von ***num1*** % ***num2*** ist der Rest der ganzzahligen Division von ***num1*** durch ***num2***. Beide Argumente müssen einen ganzzahligen Typ besitzen.	`int i = 4, j = 5;` `cout << i%j;`	`4`

++*var*
--*var*
	Präinkrement bzw. dekrement
	Als Argumente sind hier nur Variablen erlaubt.
	++*var* liefert *var* + 1
	erhöht *var* um 1
	--*var* liefert *var* - 1
	verringert *var* um 1
		`int i = 4, j;`
		`double a = 0.2;`
		`j = ++i;`
		`cout << j << endl << --a;`
			`5`
			`-0.8`

var++
var--
	Postinkrement bzw. -dekrement
	Als Argumente sind hier nur Variablen erlaubt
	var++ liefert ***var***
	erhöht ***var*** um 1
	var-- liefert ***var***
	verringert ***var*** um 1
		`int i = 4, j;`
		`j = i++;`
		`cout << i << endl << j;`
			`5`
			`4`

Bei den Prä- und Postinkrement- bzw. dekrementoperatoren ist zu beachten, dass sie *zwei* Effekte haben:

- Das Argument wird ausgelesen, um den Wert des Ausdrucks ermitteln zu können.
- Der Wert des Argumentes wird verändert.
  Wegen dieses „Nebeneffektes“ sind Prä- und Postinkrement- bzw. dekrementausdrücke im Unterschied zu den meisten anderen arithmetischen Ausdrücken bereits vollwertige Anweisungen, sobald ein Semikolon dahinter gesetzt wird:

Quellcode	Visual C++ - Ausgabefenster	Ausgabe

`#include <iostream.h>`
 	
`void main()`
`{`
`  int i = 12;`
`  i++;`
`  cout << i;`
`}`
	`Test.exe - 0 Fehler, 0 Warnung(en)`	`13`

`#include <iostream.h>`
 	
`void main()`
`{`
`  int i = 12;`
`  i+1;`
`  cout << i;`
`}`
	`warning C4552: '+' :`
	`Operator hat keine Auswirkungen;`
	`Operator mit Seiteneffekt erwartet`
	
	`Test.exe - 0 Fehler, 1 Warnung(en)`
		`12`

In der Bezeichnung „C++“ wird übrigens durch den Inkrementoperator zum Ausdruck gebracht, dass es sich um eine Erweiterung der prozeduralen Computersprache C handelt.

Mathematisch Interessierte vermissen in obiger Tabelle vermutlich die Potenzfunktion. Sie wird neben vielen anderen mathematischen Funktionen (Logarithmus, Wurzel, trigonometrische und hyperbolische Funktionen) in der Mathematik-Standardbibliothek **math** bereitgestellt und z.B. in folgendem Programm verwendet:

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

void main()
{
  cout << pow(2, 3) << endl;
}
```

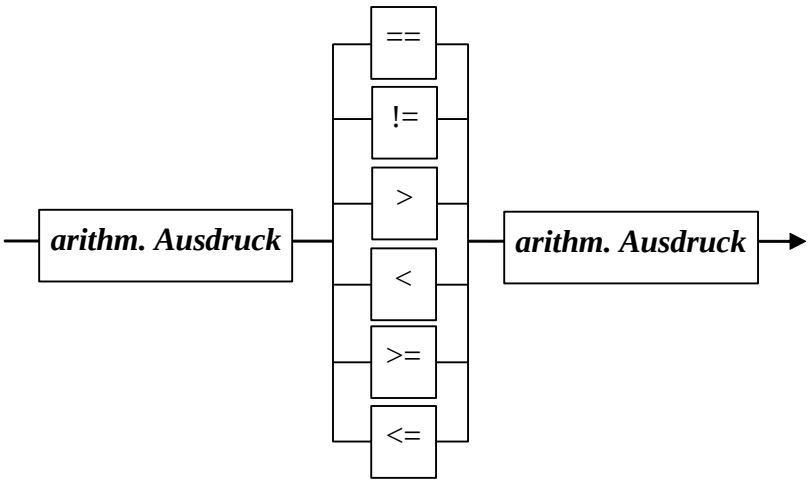
Bald werden wir uns ausführlich mit der Verwendung von Funktionen aus den C++ - Standardbibliotheken befassen.

3.3.2 Vergleichsoperatoren

Während sich mit arithmetischen Ausdrücken mathematische Berechnungen ausführen lassen, dienen **logische Ausdrücke** dazu, Bedingungen zu formulieren.
Sie haben den Typ **bool** und können folgende Werte annehmen:

- 1 (**true**), falls die Bedingung zutrifft
- 0 (**false**), sonst

Ein **Vergleich** ist ein besonders einfach aufgebauter logischer Ausdruck, bestehend aus zwei arithmetischen Ausdrücken und einem Vergleichsoperator:



In der folgenden Tabelle stehen ***ara1*** und ***ara2*** für arithmetische Ausdrücke:

Operator	Bedeutung	Beispiel	
		Programmfragment	**Ausgabe**

ara1 == ***ara2***	**Gleichheit**
	Der C++ - Compiler erwartet für Zuweisungs- und Identitätsoperator verschiedene Symbole!
		`int i = 2, j =3;`
		`bool erg;`
		`erg = i==j;`
		`cout << erg;`
			`0`

ara1 != ***ara2***	**Ungleichheit**	`int i = 2, j =3;`
		`bool erg;`
		`erg = i!=j;`
		`cout << erg;`
			`1`

ara1 > ***ara2***	**größer**	`cout << (3>2);`	`1`
ara1 < ***ara2***	**kleiner**	`cout << (3<2);`	`0`
ara1 >= ***ara2***	**größer gleich**	`cout << (3>=2);`	`1`
ara1 <= ***ara2***	**kleiner gleich**	`cout << (3<=2);`	

Achten Sie unbedingt darauf, dass der Identitätsoperator durch **zwei** „=“-Zeichen ausgedrückt wird. Einer der häufigsten C++ - Programmierfehler besteht darin, beim Identitätsoperator nur *ein* Gleichheitszeichen zu schreiben. Dabei kommt es in der Regel *nicht* zu einer Compiler-Fehlermeldung, sondern zu einem unerwarteten Verhalten des Programms, was im Folgenden durch zwei Varianten eines Beispielprogramms demonstriert werden soll.
Die erste Variante verwendet den korrekten Identitätsoperator und zeigt ein vernünftiges Verhalten:

Quellcode	Ausgabe

`#include <iostream.h>`
 	
`void main()`
`{`
`  int i = 1;`
`  cout << (i == 2) << endl;`
`  cout << i;`
`}`
	`0`
	`1`

Nach dem Entfernen eines Gleichheitszeichens erhält man völlig andere Ausgaben:

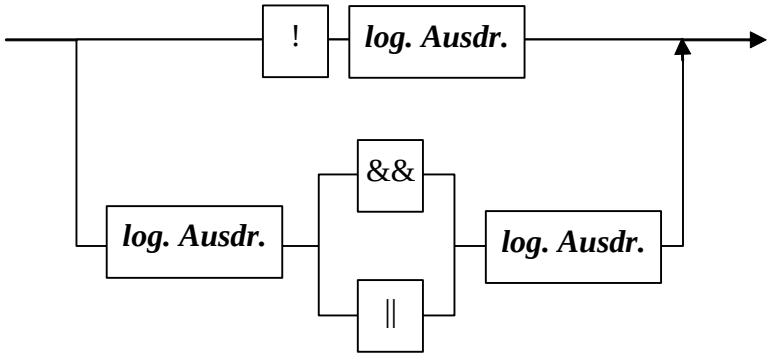
Quellcode	Ausgabe

`#include <iostream.h>`
 	
`void main()`
`{`
`  int i = 1;`
`  cout << (i = 2) << endl;`
`  cout << i;`
`}`
	`2`
	`2`

Die Konstruktion in der zweiten Beispiel-Variante dürfte nur sehr selten sinnvoll sein, ist aber erlaubt, weil ein einzelnes Gleichheitszeichen einen *Zuweisungsoperator* darstellt (siehe Abschnitt 3.3.6), so dass ein Ausdruck resultiert. Dessen Wert ist identisch mit dem Ergebnis der Zuweisung.

3.3.3 Logische Operatoren

Jeder Vergleich ist schon ein vollständiger logischer Ausdruck. Durch Anwendung der logischen Operatoren auf bereits vorhandene logische Ausdrücke kann man neue, komplexere logische Ausdrücke erstellen:



Die Wirkungsweise der logischen Operatoren wird in „Wahrheitstafeln“ beschrieben (***la1*** und ***la2*** sind logische Ausdrücke):

Argument ***la1***	**Negation** !***la1***
true	false
false	true

Argument 1 ***la1***	Argument 2 ***la2***	**Logisches UND** ***la1*** && ***la2***	**Log. ODER** ***la1*** || ***la2***
true	true	true	true
true	false	false	true
false	true	false	true
false	false	false	false

In der folgenden Tabelle gibt es noch einige Erläuterungen und Beispiele:

Operator	Bedeutung	Beispiel	
		Programmfragment	**Ausgabe**[1]
!***la1***	**Negation** Der Wahrheitswert wird umgekehrt.	`bool erg = true;` `cout << !erg;`	`0`

la1 && ***la2***	**Logisches UND**
	la1 && ***la2*** ist genau dann wahr, wenn beide Argumente wahr sind.
		`bool la1 = true, la2 = false, erg;`
		`erg = la1 && la2;`
		`cout << erg;`
			`0`

la1 || ***la2***	**Logisches ODER**
	la1 || ***la2*** ist genau dann wahr, wenn mindestens ein Argument wahr ist.
		`bool la1 = true, la2 = false, erg;`
		`erg = la1 || la2;`
		`cout << erg;`
			`1`

1 steht für true und 0 für false

3.3.4 Bitorientierte Operatoren

Über unseren momentanen Bedarf hinaus gehend bietet C++ Operatoren zur bitweisen Manipulation von Variableninhalten, die vor allem in der Systemprogrammierung von Interesse sein dürften. Statt einer systematischen Darstellung der verschiedenen Operatoren (siehe z.B. RRZN 1999, S. 68) beschränken wir uns daher auf ein Beispielprogramm, das zudem generell nützliche Einblicke in die Speicherung von **char**-Daten im Computerspeicher vermitteln kann. Das auf einer Vorlage von Eckel (2000, S. 173) basierende Programm **CBit** liefert zu einem beliebigen Zeichen die binäre Kodierung:

```
// Programm CBit nach Eckel (2000, S. 173) zur
//
// Anzeige des Bitmusters zu einem Zeichen

#include <iostream.h>

void main()
{
  unsigned char cbit;
  int i;
  cout << "Welches Zeichen moechten Sie als Bitmuster sehen? ";
  cin >> cbit;
  for(i = 7; i >= 0; i--)
  if(cbit & (1 << i))
    cout << "1";
  else
    cout << "0";
  cout << " (" << (int) cbit << ")\n";
}
```

Der **Links-Shift-Operator** << in:

```
(1 << i)
```

verschiebt die Bits in der binären Repräsentation der Ganzzahl 1 um `i` Stellen nach links. Von den insgesamt belegten 32 Bit (s.o.) interessieren nur die letzten (rechten) 8:

```
00000001
```

Im zweiten Schleifendurchgang (`i` = 6) geht dieses Muster z.B. über in:

```
01000000
```

Nach dem Links-Shift kommt der Operator **&** für das **bitweise UND** zum Einsatz:

```
cbit & (1 << i)
```

Er erzeugt ein Bitmuster, das an der Stelle **j** eine 1 enthält, wenn *beide* Argumentmuster an dieser Stelle den Wert 1 haben, und anderenfalls eine 0. Ist z.B. das Zeichen „x“ mit dem Bitmuster:

```
01111000
```

zu bearbeiten, dann liefert `cbit & (1 << i)` bei `i` = 6 das Muster:

```
01000000
```

Nun müssen Sie noch zur Kenntnis nehmen, dass in C++ jeder Ausdruck mit ganzzahligem Wert auch logisch interpretiert werden kann, wobei gilt:

0 ≈ **false**
alles Andere ≈ **true**

Das Bitmuster `01000000` liefert also den Wahrheitswert **true**, so dass

```
cout << "1";
```

ausgeführt wird.

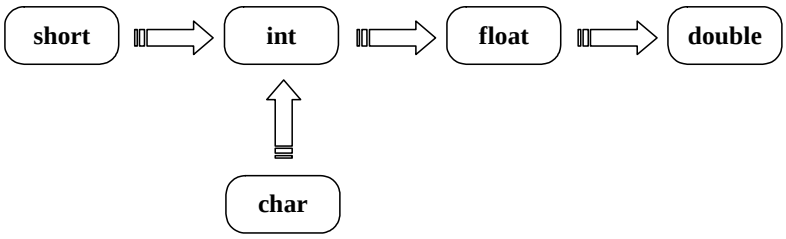
3.3.5 Typumwandlung (Casting)

Beim Auswerten des Ausdrucks

```
2.0/7
```

trifft der Divisions-Operator auf ein **double**- und auf ein **int**-Argument, so dass nach der Tabelle in Abschnitt 3.3.1 die Gleitkomma-Arithmetik zum Einsatz kommt. Dabei wird für das **int**-Argument eine **automatische (implizite) Wandlung** in den Datentyp **double** vorgenommen.

C++ nimmt bei Bedarf für Standard-Datentypen u.a. die folgenden **erweiternden Konvertierungen** zu einem allgemeineren Typ ohne Kommentar automatisch vor:



Außerdem verwendet C++ bei Operationen mit ganzzahligen Werten für Zwischenergebnisse mindestens den Typ **int**, so dass in folgendem Beispiel das richtige Produkt auf dem Bildschirm erscheint, obwohl es nicht im Wertebereich der Faktoren liegt:

Quellcode	Ausgabe

`#include <iostream.h>`
 	
`void main()`
`{`
`  char a = 100, b = 100;`
`  cout << a*b;`
`}`
	`10000`

Bei einer impliziten Typumwandlung mit potentiellem Informationsverlust warnt der Compiler, sofern Gleitkomma-Variablen beteiligt sind, z.B.:

Quellcode	Visual C++ - Ausgabefenster	Ausgabe

`#include <iostream.h>`
 	
`void main()`
`{`
`  float kurz;`
`  double lang = 40000.0;`
`  kurz = lang;`
`  cout << kurz;`
`}`
	`warning C4244: '=' : Konvertierung von 'double' in 'float', moeglicher Datenverlust`
	`Test.exe - 0 Fehler, 1 Warnung(en)`
		`40000`

Findet die implizite Typumwandlung jedoch innerhalb der Ganzzahl-Familie statt, dann wird **nicht** gewarnt, z.B.:

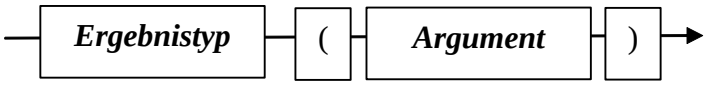
Quellcode	Visual C++ - Ausgabefenster	Ausgabe

`#include <iostream.h>`
 	
`void main()`
`{`
`  short kurz;`
`  int lang = 40000;`
`  kurz = lang;`
`  cout << kurz;`
`}`
	`Test.exe - 0 Fehler, 0 Warnung(en)`	`-25536`

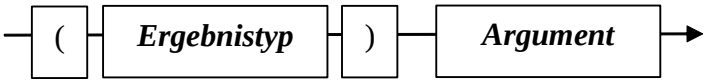
Erfahrene Programmierer halten es generell für riskant, auf implizite Typumwandlungen zu vertrauen (siehe z.B. Lee & Phillips 1997, S. 356) und empfehlen:

- erforderliche Typanpassungen stets explizit vornehmen
- bei Literalen in Ausdrücken stets den richtigen Typ verwenden, z.B.:
  `cout << 4.0/5.0;`

Die Syntax für eine explizite Typumwandlung (engl.: type casting) ist simpel:



In der Sprache C üblich und in machen Situationen nach wie vor erforderlich ist die folgende, weniger intuitive, Syntax zur Typumwandlung:



In folgendem Beispiel wird durch eine explizite Typumwandlung dafür gesorgt, dass der Compiler mit der Gleitkomma-Arithmetik arbeitet:

Quellcode	Ausgabe

`#include <iostream.h>`
 	
`void main()`
`{`
`    int i = 100, j = 8;`
`    cout << i/j << endl;`
`    cout << double(i)/j;`
`}`
	`12`
	`12.5`

Bei einer expliziten Typumwandlung geht der Compiler geht davon aus, dass der Programmierer weiß, was er tut, und warnt in keinem Fall vor einem möglichen Informationsverlust.

3.3.6 Zuweisungsoperatoren

Bei den ersten Erläuterungen zu Wertzuweisungen blieb aus didaktischen Gründen unerwähnt, dass in C++ eine Wertzuweisung als *Ausdruck* aufgefasst wird, dass wir es also mit einem binären (zweistelligen) Operator = zu tun haben:

- Auf der linken Seite muss eine Variable stehen (oder ein anderes Objekt, das einen Wert annehmen kann). Man redet hier von **L-Werten** (*L* für *left*) und erhält bei Missachtung der genannten Regel in Visual C++ folgende Fehlermeldung:

  ```
  error C2106: '=' : Linker Operand muss ein L-Wert sein
  ```

- Auf der rechten Seite muss ein Ausdruck stehen, der einen kompatiblen Wert liefert.
- Der Wert der rechten Seite wird auch dem gesamten Zuweisungsausdruck als Wert zugeordnet.
- Der Zuweisungsausdruck kann als Argument eines anderen Ausdrucks fungieren, z.B.:

Quellcode	Ausgabe

`#include <iostream.h>`
 	
`void main()`
`{`
`  int i = 2, j = 4;`
`  i = j = j * i;`
`  cout << i << " " << j;`
`}`
	`8 8`

  Weil bei mehrfachem Auftreten des Zuweisungsoperators eine Abarbeitung von **rechts nach links** erfolgt (siehe Abschnitt 3.3.8), passiert im letzten Beispiel folgendes:
  - `j * i` liefert das Ergebnis 8, das der Variablen `j` zugewiesen wird und gleichzeitig den Wert des Ausdrucks `j = j * i` darstellt.
  - In der zweiten Zuweisung (bei Betrachtung von Rechts nach Links) wird der Wert des Ausdrucks `j = j * i` an die Variable `i` übergeben.

Wie wir spätestens seit Abschnitt 3.2.3 wissen, stellt ein Zuweisungsausdruck (z.B. `a = 3`) bereits eine vollständige **Anweisung** dar, sobald man ein Semikolon dahinter setzt. Dies gilt auch für die die Prä- und Postinkrementausdrücke (vgl. Abschnitt 3.3.1), jedoch **nicht** für die anderen Ausdrücke, die in Abschnitt 3.3 vorgestellt werden.

Weil häufig Zuweisungen nach dem Muster `j = j * i` benötigt werden (eine Variable erhält ein Berechnungsergebnis, an dem sie selbst mitwirkt), wurden in C++ einige spezielle Zuweisungsoperatoren eingeführt, die in Anlehnung an Lee & Phillips (1997, S. 362) als **Aktualisierungsoperatoren** bezeichnet werden sollen.
In der folgenden Tabelle steht ***var*** für eine numerische Variable (z.B. **char**, **short**, **int**, **float** oder **double**) und ***expr*** für einen Ausdruck mit zulässigem Typ:

Operator	Bedeutung	Beispiel	
		Programmfragment	Neuer Wert[1]
var += ***expr***	***var*** erhält den neuen Wert ***var*** + ***expr***.	`int i = 2;` `i += 3;`	`5`
var -= ***expr***	***var*** erhält den neuen Wert ***var*** - ***expr***.	`int i = 10, j = 3;` `i -= j*j;`	`1`
var *= ***expr*** ***var*** /= ***expr***	***var*** erhält den neuen Wert ***var*** * ***expr*** bzw. ***var*** / ***expr***.	`int i = 10;` `i /= 5;`	`2`
var %= ***expr***	***var*** erhält den neuen Wert ***var*** % ***expr***.	`int i = 10;` `i %= 5;`	`0`

[1] Die Spalte zeigt jeweils den neuen Wert der aktualisierten Variablen `i`.

3.3.7 Konditional- und Kommaoperatoren

In der folgenden Tabelle werden zwei Operatoren vorgestellt, die als gute Beispiele für den C-AbKüFi[1] gelten können (***expr1***, ***expr1*** und ***expr3*** sind beliebige Ausdrücke):

Operator	Bedeutung	Beispiel	
		Programmfragment	Ergebnis[1]
expr1 ? ***expr2*** : ***expr3***	Falls ***expr1*** wahr ist (≠ 0), wird ***expr2*** ausgewertet, anderenfalls ***expr3***. Der ausgewertete Teilausdruck liefert Typ und Wert des gesamten Konditionalausdrucks.	`int i = 12, j = 3, k;` `k = (i > j)?i:j;`	`12`
expr1, ***expr2***	Erst wird ***expr1*** ausgewertet, dann ***expr2***. Typ und Wert des gesamten Ausdrucks liefert ***expr2***.	`int i = 12, k;` `k = (++i, 55);`	`55`

[1] Die Spalte zeigt jeweils den neuen Wert der aktualisierten Variablen `k`.

Die Besonderheit des **Konditionaloperators** besteht darin, dass er *drei* Argumente verarbeitet, welche durch die Zeichen **?** und **:** sortiert werden:

Mit dem **Kommaoperator** kann man einen zwei oder gar mehrere Ausdrücke an einer Stelle unterbringen, an der eigentlich nur einer erlaubt ist, z.B. auf der rechten Seite einer Zuweisung. Weil das Komma außerdem noch als Trennzeichen verwendet wird (z.B. in Parameterlisten), besteht eine enorme Verwechslungsgefahr. Eine weitere Gefahrenquelle sind falsch eingeschätzte Auswertungsprioritäten (siehe Tabelle in Abschnitt 3.3.8), die es nahe legen, den Kommaoperator mit seinen Operanden systematisch einzuklammern.
In der folgenden Tabelle sind weitere Fallen veranschaulicht:

Deklaration und Zuweisung	Ergebnis	Ursache für das unerwartete Ergebnis
`int i = 12, k;` `k = ++i, 55;`	k = 13	Der Zuweisungsoperator hat eine höhere Priorität.
`double x;` `x = 3,013;`	x = 3.0	Das falsche Dezimaltrennzeichen wird (ohne Fehlermeldung) als Kommaoperator mit den beiden Operanden `x = 3` und `013` interpretiert.

[1] C-Abkürzungs-Fimmel

3.3.8 Auswertungsreihenfolge

Bisher haben wir Ausdrücke mit mehreren Operatoren und das damit verbundene Problem der *Auswertungsreihenfolge* nach Möglichkeit gemieden. In diesem Abschnitt werden die Regeln vorgestellt, nach denen der C++ - Compiler komplexe Ausdrücke mit mehreren Operatoren auswertet.

Priorität

Zunächst entscheidet die Priorität der Operatoren (siehe Tabelle unten) darüber, in welcher Reihenfolge die Auswertung vorgenommen wird. Z.B. hält sich C++ bei rein arithmetischen Ausdrücken an die mathematische Regel: Punktrechnung geht vor Strichrechnung.

Auswertungsrichtung

Bei gleicher Priorität wird ein zweites Attribut der beteiligten Operatoren relevant: deren Auswertungsrichtung[1]. In der Regel werden gleichrangige Operatoren von links nach rechts ausgewertet, doch haben wir beim Zuweisungsoperator schon eine Ausnahme kennen gelernt. Um letzte Unklarheiten auszuräumen, ist in C++ für Operatoren gleicher Priorität stets auch dieselbe Richtung festgelegt.

Bei manchen Operationen gilt das mathematische Assoziativitätsgesetz, so dass die Reihenfolge der Auswertung irrelevant ist, z.B.:

$$(3 + 2) + 1 = 6 = 3 + (2 + 1)$$

Anderen Operationen fehlt diese nette Eigenschaft, z.B.:

$$(3 - 2) - 1 = 0 \neq 3 - (2 - 1) = 2$$

Klammern

Wenn die aus obigen Regeln resultierende Auswertungsfolge nicht zum gewünschten Ergebnis führt, kann mit runden Klammern steuernd eingegriffen werden: Die eingeklammerten Teilausdrücke werden „von innen nach außen“ ausgewertet.

Operatorentabelle

In der folgenden Tabelle sind die bisher behandelten Operatoren in absteigender Priorität (von oben nach unten) mit Auswertungsrichtung aufgelistet. Gruppen von Operatoren mit gleicher Priorität sind durch fette horizontale Linien begrenzt. Sie verfügen dann stets auch über dieselbe Auswertungsrichtung.

[1] In der Informatik spricht man auch von Bindungsrichtung bzw. Assoziativität (siehe z.B. Schiedermeier 1996).

	Operator	Bedeutung	Richtung
↓ absteigende Priorität ↓	!	Negation	L ← R
	~	Bitweise Negation	
	++, --	Prä- oder Postinkrement bzw. -dekrement	
	-	Vorzeichenumkehr	
	(*type*)	Typumwandlung	
	*, /	Punktrechnung	L → R
	%	Modulo	
	+, -	Strichrechnung	L → R
	<<, >>	Bitweiser Links- bzw. Rechts-Shift	L → R
	>, <, >=, <=	Vergleichsoperatoren	L → R
	==, !=	Gleichheit, Ungleichheit	L → R
	&	Bitweises UND	L → R
	^	Exklusives bitweises ODER	L → R
	|	Bitweises ODER	L → R
	&&	Logisches UND	L → R
	||	Logisches ODER	L → R
	? :	Konditionaloperator	L ← R
	=, +=, -=, *=, /=, %=	Wertzuweisung, Aktualisierung	L ← R
	,	Kommaoperator	L → R

Im Anhang finden Sie eine erweiterte Version dieser Tabelle, die zusätzlich alle Operatoren enthält, die im Verlauf des Kurses noch behandelt werden.
In der Online-Hilfe zu C++ 6.0 (MSDN) finden Sie weitere Informationen, z.B. bei einer Suche zu den Themen *Precedence and Order of Evaluation* oder *C++ Operators*.

Eine Kenntnis der Operatoren-Prioritäten ist zweifellos wichtig, sollte aber nicht zu einem extrem knappen und dabei schwer nachvollziehbarem Programmierstil eingesetzt werden. Durch den Einsatz von Klammern kann man die Lesbarkeit des Quellcodes verbessern und Fehler durch falsch erinnerte Operatoren-Prioritäten vermeiden.

3.3.9 Über- und Unterlauf

Wie wir inzwischen wissen, haben Variablen, Konstanten und Literale einen durch ihren Typ bestimmten Wertebereich. Treten unzulässige Werte auf (z.B. in einer Berechnung oder Zuweisung), dann kann das betroffene Programm ein mehr oder weniger gravierendes Fehlverhalten zeigen, so dass Sie einen solchen Über- oder Unterlauf unbedingt vermeiden bzw. rechtzeitig diagnostizieren müssen.

Leider dürfen Sie von einem C++ - Programm nicht erwarten, dass es im Falle einer „übergelaufenen“ **Ganzzahlvariablen** mit einem Laufzeitfehler seine Tätigkeit einstellt, um Schlimmeres zu vermeiden. Das folgende Programm

```
#include <iostream.h>

void main()
{
   int i, j, k;
   i = 2147483647;
   j = 5;
   k = i + j;
   cout << i << " + " << j << " = " << k << endl;
}
```

liefert ohne jede Warnung oder Selbstzweifel das fragwürdige Ergebnis:

```
2147483647 + 5 = -2147483644
```

Auch beim Kompilieren fällt Nichts auf, wie das zugehörige Ausgabefenster zeigt:

```
--------------------Konfiguration: Test - Win32 Debug--------------------
Kompilierung läuft...
test.cpp
Linker-Vorgang läuft...

Test.exe - 0 Fehler, 0 Warnung(en)
```

Da es sich bei `i` und `j` um *Variablen* handelt, erregt die kritische Zuweisung `k = i + j` beim Compiler keinen Verdacht. Bei *Konstanten* oder *Literalen*, deren endgültige Werte ja schon beim Kompilieren bekannt sind, ist die Lage weit günstiger. Nach dem Kompilieren der folgenden Programmversion:

```
#include <iostream.h>

void main()
{
   const int i = 2147483647, j = 5;
   int k;
   k = i + j;
   cout << i << " + " << j << " = " << k << endl;
}
```

meldet Visual C++ im Ausgabefenster:

```
--------------------Konfiguration: Test - Win32 Debug--------------------
Kompilierung läuft...
test.cpp
U:\Eigene Dateien\VCpp\Test\test.cpp(7) : warning C4307: '+' : Ueberlauf einer
ganzzahligen Konstanten
Linker-Vorgang läuft...

Test.exe - 0 Fehler, 1 Warnung(en)
```

Ein Überlauf kann auch bei **Gleitkommavariablen** auftreten, wobei aber *keine* sinnlosen Zahlen entstehen. Das Programm

```
#include <iostream.h>

void main()
{
   double bigd = 1.79769313e308, smalld = 1.79769313e-323;
   cout << "bigd =   \t" << bigd << "\t bigd * 10 = \t" << bigd*10 << endl;
   cout << "smalld = \t" <<smalld << "\t smalld / 10 = \t" << smalld/10 << endl;
}
```

liefert die folgende Ausgabe, wobei mit `1.#INF` der Wert +∞ bezeichnet wird:

```
bigd =          1.79769e+308     bigd * 10 =    1.#INF
smalld =        1.97626e-323     small / 10 =   0
```

Obwohl sich C++ bei der Gleitkomma-Arithmetik bemüht, Zahlen mit Beträgen außerhalb des darstellbaren Wertebereiches sinnvoll zu behandeln, sollte man den Grenzbereich besser meiden, weil nach einem Überlauf mit merkwürdigen Ergebnissen zu rechnen ist. Im folgenden Programm wird der Ausdruck `d/10.0e307` (mit der übergelaufenen Variablen `d`) ohne zwischenzeitliche Veränderung zweimal ausgegeben:

Quellcode	Ausgabe

`#include <iostream.h>`
 	
`void main()`
`{`
`  double bigd = 1.79769313e308, d;`
`  d = bigd*10;`
`  cout << d/10.0e307;`
`  cout << "  "  << d/10.0e307 << endl;`
`}`
	`17.9769  1.#INF`

Erstaunlicherweise fallen die Ausgaben sehr verschieden aus:

- `cout << d/10.0e307;` liefert den korrekten Wert `17.9769`, wobei der `d`-Überlauf geschickt rückgängig gemacht wird.
- Wird zusätzlich mit `<< endl` ein Zeilenumbruch ausgegeben, unterbleibt die Rekonstruktion des alten `d`-Wertes, und auf dem Bildschirm erscheint +∞, ein von `17.9769` recht verschiedenes Ergebnis.

Mit einem alternativen C++ - Compiler lassen sich diese Ergebnisse vermutlich nicht reproduzieren.

Bei Gleitkommavariablen ist auch ein **Unterlauf** möglich, wobei eine Zahl mit sehr kleinem Betrags nicht mehr dargestellt werden (z.B. 1.79769313e-324). Im diesem Fall rechnet ein Visual C++ - Programm mit dem Wert 0 weiter, was in der Regel akzeptabel ist.

Bei der Kalkulation des Unter- bzw. Überlaufrisikos darf man sich nicht auf die Endergebnisse von Ausdrücken beschränken, sondern muss auch alle Zwischenschritte berücksichtigen. Im folgenden Beispiel wird ein **double**-Argument durch 10 dividiert und anschließend logarithmiert. Für das Argument $1.5 \cdot 10^{-323}$ wird als Funktionswert erwartet:

$$\log\left(\frac{1.5 \cdot 10^{-323}}{10}\right) = \log(1.5 \cdot 10^{-323}) - \log(10) \approx -743.34 - 2.30 = -745.64$$

Wegen eines Unterlaufs während der Berechnung kommt das Programm aber zum Ergebnis -∞ (`-1.#INF`):

Quellcode	Ausgabe

`#include <iostream.h>`
`#include <math.h>`
 	
`void main()`
`{`
`  double doub = 1.5e-323;`
`  cout << log(doub/10) << endl;`
`  cout << log(doub) - log(10) << endl;`
`}`
	`-1.#INF`
	`-745.644`

3.3.10 Übungsaufgaben zu Abschnitt 3.3

1) Welche Werte und Datentypen besitzen die folgenden Ausdrücke?

```
6/4*2.0
(int)6/4.0*2  3*5+8/3%4*5
```

2) Welche Werte erhalten in folgendem Programm die Integer-Variablen `erg1` und `erg2`?

```
#include <iostream.h>

void main()
{
  int i = 2, j = 3, erg1, erg2;
  erg1 = (i++ == j ? 7 : 8) % 2;
  cout << erg1 << endl;
  erg2 = (++i == j ? 7 : 8) % 2;
  cout << erg2 << endl;
}
```

3) Laut Schulmathematik gilt für $b > 0$:

$$\frac{a}{b} \cdot b = a$$

Warum kommt das folgende Programm zu einem anderen Ergebnis?

Quellcode	Ausgabe

`#include <iostream.h>`
 	
`void main()`
`{`
`   cout << 100/8 * 8;`
`}`
	`96`

4) Wird ein **double**-Wert einer **int**-Variablen zugewiesen, dann gehen die Nachkommastellen verloren, so dass z.B. aus dem **double**-Wert 2.85 der **int**-Wert 2 resultiert. Was kann man tun, damit beim Übergang auf die nächste ganze Zahl gerundet wird?

5) Welche Wahrheitsweite erhalten in folgendem Programm die Variablen `la1`, `la2` und `la3`?

```
#include <iostream.h>

void main()
{
   bool la1, la2, la3;
   const int i = 3;

   la1 = (2 > 3) && (2 == 2) || (1 == 1);
   cout << la1 << endl;

   la2 = (2 > 3) && ((2 == 2) || (1 == 1));
   cout << la2 << endl;

   la3 = i > 0 && i * 4 == 012;
   cout << la3 << endl;
}
```

6) In Abschnitt 3.3.6 wurde versichert, dass bei der Aktualisierung ***var*** += ***expr*** die Variabel ***var*** den neuen Wert ***var*** += ***expr*** erhalte. Liefert das folgende Programm einen Widerspruch zu dieser Behauptung?

Quellcode	Ausgabe

`#include <iostream.h>`
 	
`void main()`
`{`
`  int i = 5;`
`  i += 1.5;`
`  cout << i;`
`}`
	`6`

7) Kann bei Ganzzahlvariablen ein Unterlauf auftreten?

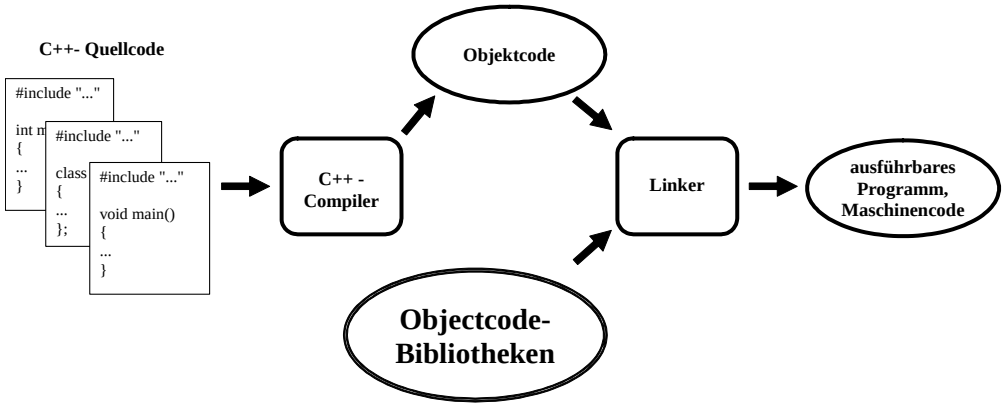
8) Schreiben Sie ein kaufmännisch einsetzbares Programm, das einen Netto-Betrag entgegennimmt und daraus nach folgenden Regeln den zu zahlenden Endbetrag berechnet:

- Zum Netto-Betrag müssen 16% Mehrwertsteuer addiert werden.
- Falls der so ermittelte Zwischenbetrag 100 € übertrifft, wird dem Kunden 3 % Rabatt gewährt.

Das Ergebnis soll natürlich ausgegeben werden.

3.4 Vordefinierte Bibliotheksfunktionen

Mit den bisher beschriebenen Operatoren lassen sich zwar schon viele Probleme lösen, bisweilen aber nur sehr umständlich. Außerdem wäre z.B. sehr viel mathematischer Sachverstand erforderlich, trigonometrische Funktionswerte mit den bisher eingeführten Hilfsmitteln zu berechnen. Statt das Rad und ähnliche Dinge neu zu erfinden, benutzen wir lieber die Funktionen der **C++ - Standardbibliothek**, die (ggf. zusammen mit anderen Bibliotheken) vom Linker in unsere Programme eingebunden werden können:



Im Unterschied zu den Operatoren gehören die Bibliotheksfunktionen *nicht* zur Sprachdefinition, doch wurde in ANSI/ISO - Standards festgelegt, welche Funktionen ein C++ - Entwicklungssystem im Rahmen der Standardbibliothek anbieten muss, und wie deren Schnittstellen (s.u.) beschaffen sein müssen.
Wie wir bereits aus eigener Praxis wissen, muss der Compiler auch bei den vordefinierten Funktionen aus der C++ - Standardbibliothek mit Hilfe einer inkludierten Header-Datei über die geplante Verwendung informiert werden.
Durch einen Funktionsaufruf können beliebig komplexe Verarbeitungen an ein „im Hintergrund tätiges“ Modul delegiert werden, wobei wir als Anwender lediglich gewisse Schnittstellen-Regeln zu beachten haben. Durch die Wiederverwendung von Programmcode aus hochwertigen Bibliotheken erhalten wir auf rationelle Weise ein Programm mit hoher Zuverlässigkeit und übersichtlichem Quellcode.
Bald werden wir lernen, eigene Funktionen zu definieren, um komplexe Probleme mit Hilfe des Modularitätsprinzips in handliche Teilprobleme zu zerlegen.

3.4.1 Migration und Migräne

Mit dem Übergang von C zu C++ und der allmählichen Weiterentwicklung von C++ haben sich einige Altlasten und Inkompatibilitäten zwischen verschiedenen Compiler-Versionen bzw. –Implementationen ergeben, und es war mit *der* einheitlichen Standardbibliothek vorbei.
In Visual C++ 6.0 haben wir es mit folgenden Standardbibliotheksdateien (Single-Thread[1], ohne Debug-Code) zu tun:

[1] Während sich ein klassisches Computerprogramm zu einem Zeitpunkt jeweils nur einer Aufgabe widmet, können moderne Anwendungen auch mehrere Threads („Fäden“ bzw. Ausführungspfade) simultan verfolgen. Die Unterschiede zwischen Single-Thread- und Multi-Thread-Programmen sind so groß, dass in Visual C++ jeweils eine eigene Familie von Standardbibliotheken zu verwenden ist.

Datei	Inhalt	Zugehörige Header-Dateien		
LIBC.LIB	Basis-C-Laufzeitbibliotheks-funktionen	CASSERT, CFLOAT, CLOCALE, CSIGNAL, CSTDIO, CTIME,	CCTYPE, CISO646, CMATH, CSTDARG, CSTDLIB, CWCHAR,	CERRNO, CLIMITS, CSETJMP, CSTDDEF, CSTRING, CWTYPE
LIBCI.LIB	Veraltete C++ - **iostream**-Klassen	FSTREAM.H, IOSTREAM.H, STDIOSTR.H,	IOMANIP.H, ISTREAM.H, STREAMB.H,	IOS.H, OSTREAM.H, STRSTREA.H
LIBCP.LIB	C++ - Erweiterungen mit neuen **iostream**-Klassen	ALGORITHM, DEQUE, FUNCTIONAL, IOSFWD, ITERATOR, LOCALE, NUMERIC, SET, STDEXCEPT, STRSTREAM, VALARRAY,	BITSET, EXCEPTION, IOMANIP, IOSTREAM, LIMITS, MAP, OSTREAM, SSTREAM, STREAMBUF, TYPEINFO, VECTOR	COMPLEX, FSTREAM, IOS, ISTREAM, LIST, MEMORY, QUEUE, STACK, STRING, UTILITY,

Die Bibliotheken LIBCI.LIB und LIBCP.LIB sind **unverträglich**, so dass sie nicht gemeinsam in ein Programm eingebunden werden können. Visual C++ bindet neben den Basis-C-Laufzeit-bibliotheksfunktionen in LIBC.LIB zusätzlich die C++ - Erweiterungen in LIBCP.LIB **oder** die alte **iostream**-Bibliothek LIBCI.LIB ein, wenn eine zugehörige Header-Datei inkludiert wird.
Zunächst zwei Klarstellungen:

- Aus einer Bibliothek werden nur die tatsächlich verwendeten Funktionen bzw. Objekte in den Objektcode eingebunden.
- Zu einer Bibliothek können zahlreiche Header-Dateien gehören.

Um die Header-Dateien zur Bibliothek LIBCP.LIB von den Header-Dateien zur alten **iostream**-Bibliothek unterscheiden zu können, wird gemäß ANSI/ISO – C++ -Standard auf die Namenserweiterung „.h“ verzichtet.
Die neue Notation wird auch auf die C-Header-Dateien angewendet, die (mit leichten Modifikationen) in ANSI/ISO - C++ übernommen wurden, wobei der alten Namenswurzel ein „C“ vorangestellt wird. Aus Kompatibilitätsgründen werden in ANSI/ISO - C++ auch noch die alten Namen der C-Header-Dateien unterstützt, die in folgender Tabelle den neuen Namen gegenübergestellt werden:

ANSI-C++	ANSI-C
<cassert>	<assert.h>
<cctype>	<ctype.h>
<cerrno>	<errno.h>
<cfloat>	<float.h>
<ciso646>	<iso646.h>
<climits>	<limits.h>
<clocale>	<locale.h>
<cmath>	<math.h>
<csetjmp>	<setjmp.h>
<csignal>	<signal.h>
<cstdarg>	<stdarg.h>
<cstddef>	<stddef.h>
<cstdio>	<stdio.h>
<cstdlib>	<stdlib.h>

<cstring>	<string.h>
<ctime>	<time.h>
<cwchar>	<wchar.h>
<cwtype>	<wtype.h>

Bisher wurden in diesem Manuskript, wie auch in den meisten aktuellen Lehrbüchern, die klassischen Header-Dateien verwendet (mit allen damit verbundenen Konsequenzen). Ab jetzt wollen wir uns an die ANSI/ISO - Empfehlungen halten und müssen uns daher mit Namensräumen (siehe Abschnitt 3.4.2) beschäftigen.

Da es noch viele Pre - ANSI/ISO - C++ - Programme gibt, und andererseits viel dafür spricht, die neuen ANSI/ISO - Konventionen zu benutzen, müssen wir uns ganz zu Beginn unserer C++ - Karriere schon mit Migrationsproblemen plagen, hoffentlich ohne allzu starke Migräne-Beschwerden.

Aus Abschnitt 2.5 kennen Sie schon den Unterschied zwischen der Debug- und der Release-Version eines mit dem Visual Studio 6.0 erstellten Programms. Welche Bibliotheksdateien Visual C++ 6.0 für beide Versionen sowie für weitere Varianten benutzt, können Sie bei Bedarf im Anhang nachschlagen. Wer sich näher für die Bibliotheksdateien interessiert, findet sie vom Installations-Verzeichnis (z.B. C:\Programme\Microsoft Visual Studio) ausgehend in:

VC98\Lib

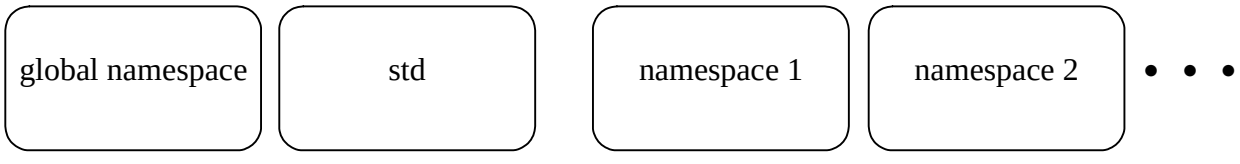
Wir haben im Moment keinen Anlass, uns mit derlei technischen Details zu beschäftigen.

3.4.2 Namensräume

Die bereits in Abschnitt 3.2.4 angesprochenen Gültigkeitsbereiche von Variablen kommen nun in einem generelleren Zusammenhang erneut zur Sprache. In C++ gehört jeder Bezeichner (z.B. für eine Variable oder für eine Funktion) zu einem Namensraum und muss nur innerhalb dieses Kontextes eindeutig sein. Bisher kennen wir:

- **den globalen Namensraum (global namespace, global scope)**
  Er enthält die außerhalb aller Funktionen deklarierten Bezeichner, z.B. die Namen der globalen Variablen und der Funktionen. Deren Gültigkeitsbereich erstreckt sich über das gesamte Programm.
  Man spricht auch von **extern** deklarierten Bezeichnern.
- **den Anweisungsblock**
  Jeder Anweisungsblock (z.B. der Rumpf einer Funktion) stellt einen eigenen Namensraum dar. Die blockintern (lokal) deklarierten Variablen sind gültig vom Deklarationspunkt bis zum Ende des Blockes.
  Anweisungsblöcke dürfen geschachtelt werden, wobei dann bei Namensgleichheit die innerste Definition dominiert. Auf diese Weise dürfen auch globale Bezeichner überdeckt werden. Änderungen lokaler Variablen haben keinen Einfluss auf „globalere“ Variablen selben Namens.

In der Programmiersprache C landen nach dem Inkludieren einer Header-Datei die Namen der nun verfügbaren Funktionen etc. grundsätzlich im globalen Namensraum, so dass es hier leicht zu Kollisionen kommt. Um Probleme durch mehrfach verwendete Bezeichner zu vermeiden, wurden in C++ die **expliziten Namensräume** eingeführt. Nun kann man den globalen Namensraum durch eigene Namensräume ergänzen:



Sofern die beteiligten Programmierer von der Möglichkeit Gebrauch machen, ist die Gefahr von Namenskonflikten erheblich reduziert.
Allerdings steht dieser Flexibilität ein erhöhter Aufwand bei der Notation gegenüber, weil zu einem Bezeichner auch der jeweilige Namensraum angegeben werden muss. Das folgende Beispielprogramm demonstriert die Definition und Verwendung von Namensräumen:

Quellcode	Ausgabe

`#include <iostream>`
 	
`namespace one`
`{`
`  int var = 1;`
`}`
 	
`namespace too`
`{`
`  double var = 3.14159;`
`}`
 	
`void main ()`
`{`
`  std::cout << one::var << std::endl;`
`  std::cout << too::var << std::endl;`
`}`
	`1`
	`3.14159`

Im Beispiel kommen nicht nur die beiden neu definierten Namensräume `one` und `too` zum Einsatz, sondern auch der Namensraum **std**, zu dem in ANSI/ISO - C++ alle Klassen und Funktionen der C++ - Standardbibliothek gehören. Durch Verwendung der Header-Datei **iostream** (ohne Extension „.h“, siehe Abschnitt 3.4.1) befindet sich das Standardausgabeobjekt **cout** nicht mehr (wie bisher gewohnt) im globalen Namensraum, sondern im Namensraum **std**.

Laut ANSI/ISO - C++ - Spezifikation gehören auch die von C übernommenen und in den den **c***-Header-Dateien (z.B. **cstdio**) deklarierten Bezeichner dem Namensraum **std** an. Leider hält sich Visual C++ nicht an diese Regel und platziert die fraglichen Bezeichner stattdessen in den globalen Namensraum, so dass viele ANSI/ISO - konforme Programme nicht übersetzt werden können, z.B:

Quellcode	Fehlermeldung

`#include <cstdio>`
 	
`void main()`
`{`
`  std::printf("Hallo");`
`}`
	`error C2653: 'std' : Keine Klasse oder Namespace`

Es ist generell festzustellen, dass Microsoft den ANSI/ISO - Standard recht zögerlich umsetzt. Immerhin hat der Konzern angekündigt, im Jahr 2003 mit dem Visual Studio .Net den C++ ANSI/ISO – Standard zu 98 % zu erfüllen (Heise-Newsticker-Meldung vom 12.11.2002).
In Visual C++ 6.0 kann man zwar Über **Projekt > Einstellungen > C/C++ > Kategorie = Anpassen > Spracherweiterungen deaktivieren** eine stärkere ANSI/ISO – Konformität erzwingen, doch treten dann Probleme mit zahlreichen mitgelieferten VC++ - Header-Dateien auf. Z.B. produziert das harmlose Programm:

```
#include <iostream>

void main()
{
  std::cout << "Hallo";
}
```

dann immerhin 102 Fehlermeldungen.

Offenbar gibt es zwischen den Normierungs-Gremien und den Compiler-Herstellern noch einige Diskussion darüber, ob die C-Standardbibliotheks-Bezeichner zum globalen Namensraum oder zum Namensraum **std** gehören sollen. Ein möglicher Kompromiss wird eventuell darin bestehen, dass sie in beiden Namensräume bekannt sind.

Wie die Bezeichner aus dem globalen Namensraum sind auch die Bezeichner aus expliziten Namensräumen *überall* (in allen anderen Namensräumen) verfügbar, wobei sie auf unterschiedliche Weise angesprochen werden können:

- **Vollqualifizierte Ansprache mit Namensraum-Bezeichner und Scope-Operator ::**
  Diese Methode wurde in den beiden letzten Beispielen vorgeführt. Der Scope- bzw. Namensraumauflösungsoperator besitzt maximale Priorität und wird von Links nach Rechts ausgewertet. Wir werden ihn vor allem bei der objektorientierten Programmierung oft benötigen.
- **Import eines Bezeichners in den aktuellen (z.B. globalen) Namensraum**
  Zur Vereinfachung der Schreibweise kann ein Bezeichner mit Hilfe der **using** - Direktive in den globalen Namensraum importiert und anschließend direkt angesprochen werden, z.B.:

```
#include <iostream>

using std::cout;

void main ()
{
 cout << "Jetzt klappt's wieder wie frueher.\n";
}
```

- **Import eines kompletten Namensraumes in den aktuellen (z.B. globalen) Namensraum**
  Gerade im Zusammenhang mit der C++ - Standardbibliothek mit ihrem Namensraum **std** können die bisher genannten Methoden lästig werden. Man kann sich das Leben erleichtern, indem man per **using**-Direktive gleich den gesamten Namensraum **std** importiert, z.B.:

```
#include <iostream>

using namespace std;

void main ()
{
 cout << "Jetzt kennt das Programm alle std-Bezeichner." << endl;
}
```

  Durch den Import von kompletten Namensräumen in den globalen Gültigkeitsbereich wird das eigentliche Ziel der Namensraum-Trennung natürlich gefährdet. Solche Importe sollten daher auf Implementierungsdateien (Endung **cpp**) beschränkt bleiben und keinesfalls in Header-Dateien vorgenommen werden.

Wird ein einem lokalen Namensraum ein Bezeichner aus dem globalen Namensraum überdeckt, so steht die globale Bezeichnung mit Hilfe des **einstelligen Scope-Operators** weiterhin zur Verfügung, z.B.:

Quellcode	Ausgabe

`#include <iostream>`
 	
`using namespace std;`
 	
`int i = 1;`
 	
`void main()`
`{`
`  int i = 2;`
`  cout << ::i << endl << i << endl;`
`}`
	`1`
	`2`

Wir werden in diesem Kurs ab jetzt ANSI/ISO - konform die C++ - Standard-Header-Dateien (z.B. **iostream**) verwenden und den **std**-Namensraum mit einer **using**-Direktive in den globalen Namensraum unserer Anwendungen importieren. Das in Abschnitt 3.1.1 vereinbarte *einfache C++ - Programm* muss also um eine **using**-Direktive erweitert werden. Am Ende von Abschnitt 3.5 wird es als einfacher Rahmen zum Üben elementarer Sprachbestandteile ohnehin ausgedient haben.

3.4.3 Freie Auswahl

In der Einleitung zu Abschnitt 3.4 war eigentlich von Möglichkeiten die Rede, sich das Leben zu *erleichtern*. Nach den etwas anstrengenden Erläuterungen der beiden letzten Abschnitte wollen wir uns nun endlich den großen Fundus von Funktionen in der Standardbibliothek erschließen.
In der **MSDN Library** zum Visual Studio 6.0 finden Sie auf dem Registerblatt **Inhalt** über

> **Visual C++ - Dokumentation > Arbeiten mit Visual C++ >**
> **Visual C++ - Programmierhandbuch > Laufzeitbibliothek-Referenz >**
> **Run-Time Routines by Category**

einen brauchbaren Überblick. Auf der Suche nach der Potenzfunktion findet man z.B. in der Abteilung **Floating-Point Support** die Funktion **pow**, zu der nach einem weiteren Mausklick u.a. folgende Informationen bereitstehen:

> **pow**
>
> Calculates *x* raised to the power of *y*.
>
> **double pow(double** *x***, double** *y* **);**

Routine	Required Header	Compatibility
pow	<math.h>	ANSI, Win 95, Win NT

> For additional compatibility information, see Compatibility in the Introduction.
>
> **Return Value**
> **pow** returns the value of x^y. No error message is printed on overflow or underflow.

Values of x and y	Return Value of pow
x < > 0 and y = 0.0	1
x = 0.0 and y = 0.0	1
x = 0.0 and y < 0	INF

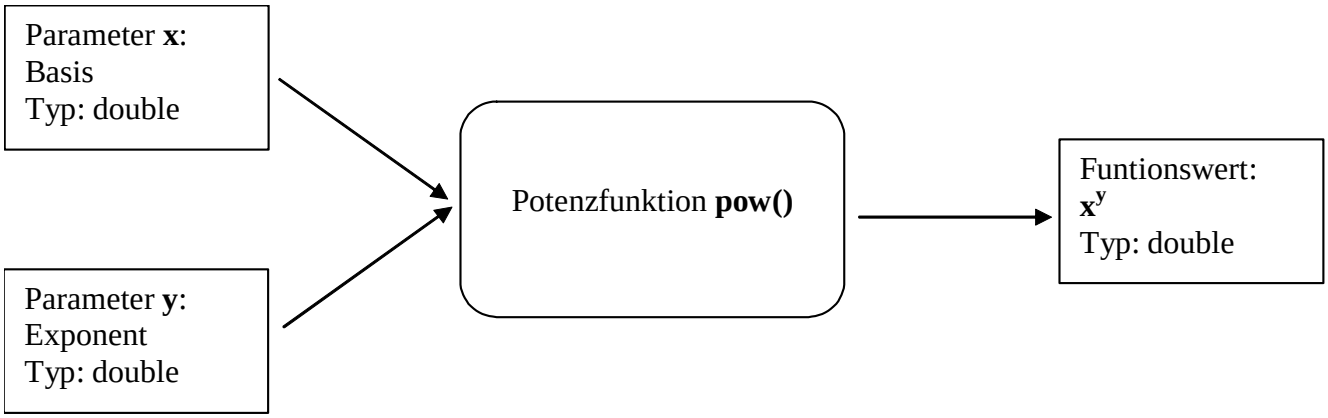
> **Parameters**
> *x*
> Base
> *y*
> Exponent

3.4.4 Funktionsschnittstellen

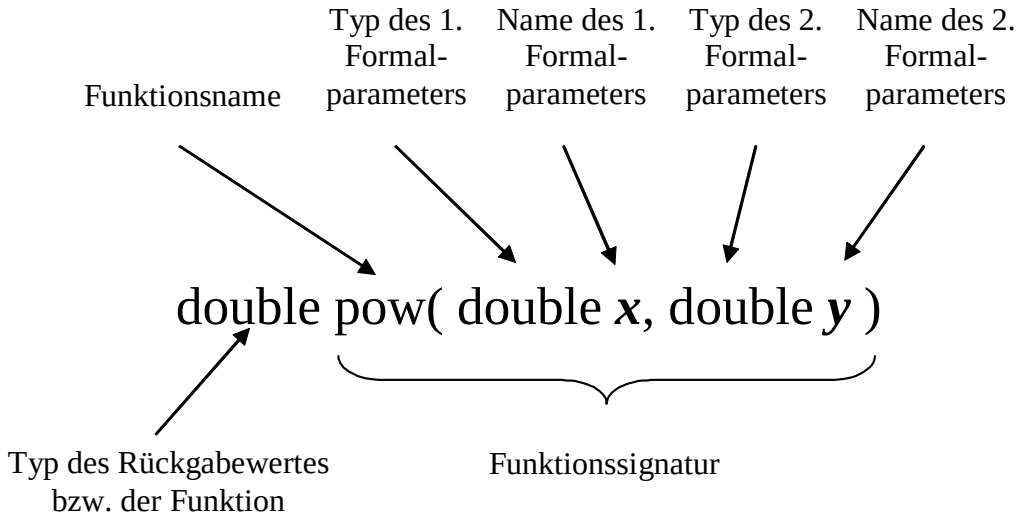
Die in der Standardbibliothek implementierten Funktionen sind Unterprogramme zur Lösung bestimmter Aufgaben, die (analog zu mathematischen Funktionen):

- **Parameter** (Eingabewerte, Argumente) entgegen nehmen,
- einen **Rückgabe- bzw. Funktionswert** berechnen.

Parameter und Rückgabewert sind von genau festgelegtem Typ.
Im Beispiel der bereits erwähnten Potenzfunktion, haben wir folgende Situation:



Die (z.B. in der Online-Hilfe wiedergegebene) **Schnittstellendefinition** der Funktion **pow()** lautet:



Die in diesem Beispiel veranschaulichte Syntax wird für uns an Bedeutung gewinnen, wenn wir *eigene* Funktionen definieren. Vorläufig beschränken wir uns auf die Verwendung vordefinierter Standardfunktionen. Dabei benötigen wir folgende Informationen:

- Name der Funktion
  Im Beispiel: **pow()**
- Erforderliche Header-Datei
  Hier findet der Compiler die für seine Syntaxprüfung erforderlichen Informationen.
  Im Beispiel: **math.h** alias **cmath** (vgl. Abschnitt 3.4.1)
- Eventuell: In welchem Namensraum ist die Funktion definiert?
  Im Beispiel: **global** (vgl. Abschnitt 3.4.2 wegen der ANSI/ISO-Kompatibilität von VC++)
- Typ und Position aller Parameter
  Im Beispiel: Basis und Exponent vom Typ **double**
- Typ des Funktionswertes
  Im Beispiel: **double**

Die in der Schnittstellendefinition enthaltenen **Formalparameter** werden beim Funktionsaufruf durch **Aktualparameter** mit kompatiblem Typ ersetzt. Gegebenenfalls nimmt der Compiler eine automatische (implizite) Typumwandlung vor, was nach Überlegungen aus Abschnitt 3.3.5 aber möglichst vermieden werden sollte.
Als Aktualparameter sind erlaubt:

- Variablen oder Konstanten (mit beliebigem Namen)
  Die Namen der Formalparameter binden uns in keiner Weise. Sie tauchen nur in der Schnittstellendefinition und in dem (für uns irrelevanten) Quellcode zur Funktions-Implementierung auf.
- Literale oder sonstige Ausdrücke

Es versteht sich von selbst, dass im Funktionsaufruf *alle* Aktualparameter in der *richtigen Reihenfolge* auftreten müssen.
Der Funktionsaufruf darf überall stehen, wo ein Wert seines Rückgabetyps zulässig ist.

Nach diesen Vorbereitungen steht einer erfolgreichen Verwendung der Funktion **pow()** nichts mehr im Wege:

Quellcode	Ein- und Ausgabe

`#include <iostream>`
`#include <cmath>`
`using namespace std;`
 	
`void main()`
`{`
`  double via, gra;`
`  cout << "Welche Potenz haetten Sie gern? ";`
`  cin >> via >> gra;`
`  cout << pow(via, gra) << endl;`
`}`
	`Welche Potenz haetten Sie gern? 2 3`
	`8`

Wir müssen uns nicht darum kümmern, wie die Funktion **pow()** ihr Ergebnis ermittelt. Den zugehörigen Objektcode findet der Linker in einer LIB-Datei mit den C-Laufzeitbibliotheksfunktionen (siehe Abschnitt 3.4.1).

Funktionen ohne Parameter

Es gibt durchaus nützliche Funktionen, die keine Parameter benötigen, wobei die leere Parameterliste aber nicht weggelassen werden darf. Ein Beispiel ist die Funktion **rand()** zum Erzeugen von ganzen Pseudozufallszahlen, über die wir in der Online-Hilfe u.a. erfahren:

> **rand**
>
> Generates a pseudorandom number.
>
> **int rand(void);**
>
> | Routine | Required Header | Compatibility |
> |---|---|---|
> | **rand** | <stdlib.h> | ANSI, Win 95, Win NT |
>
> **Remarks**
> The **rand** function returns a pseudorandom integer in the range 0 to **RAND_MAX**. Use the srand function to seed the pseudorandom-number generator before calling **rand**.

Die Parameterliste der Funktion **rand()** ist **void** (= leer). Als maximalen Wert liefert **rand()** die Zahl 32767, für die in der Header-Datei **cstdlib (**alias **stdlib.h**) der Name **RAND_MAX** vereinbart wird.

Mit der folgenden Anweisung

```
cout << (rand() % 49 + 1) << endl;
```

wird eine pseudo-zufällige Lottozahl ermittelt und ausgegeben.
Beim mehrfachen Ausführen der zugehörigen Anwendung werden Sie vermutlich den Eindruck gewinnen, dass selbst die einschränkende Bezeichnung „*Pseudo*zufallszahl“ zuviel versprochen hat: Das Programm liefert stets die selbe Pseudozufallszahl 42. Ursache ist der stets gleiche Startwert 1 des Pseudozufallszahlengenerators. Sicherlich haben Sie aus den oben wiedergegebenen Online-Informationen zur **rand()**-Funktion noch den Querverweis auf die verwandte Funktion **srand()** in Erinnerung. Damit können Sie die per **rand()** abrufbare Pseudozufallssequenz mit einem alternativen Wert starten lassen. Nähere Informationen zu **srand()** folgen sofort:

Funktionen ohne Rückgabewert

Manche Programmiersprachen (z.B. Delphi) kennen neben den *Funktionen* noch eine zweite Sorte von Routinen, meist als *Prozeduren* bezeichnet, die keinen Rückgabewert liefern. In C++ dienen solchen Zwecken einfach Funktionen mit dem leeren Rückgabewert **void**, z.B.

srand

Sets a random starting point.

void srand(unsigned int *seed* **);**

Routine	Required Header	Compatibility
srand	<stdlib.h>	ANSI, Win 95, Win NT

„Wert“-lose Funktionen kann man in Zuweisungen oder sonstigen Ausdrücken nicht gebrauchen und ruft sie statt dessen in einer eigenen Anweisung auf, z.B.:

```
srand(2);
cout << (rand() % 49 + 1) << endl;
```

Das Lotto-Programm ist nun einen erheblichen Schritt voran gekommen und liefert (leider bei jedem Aufruf) die Zahl 46.
Im Rahmen der anschließenden Übungsaufgabe werden Sie eine perfekte Programmversion erstellen, die aus persönlichen Daten eine optimierte Lottozahl ermittelt.

3.4.5 Übungsaufgabe zu Abschnitt 3.4

Erstellen Sie ein Programm zum Berechnen einer persönlich-optimierten Lottozahl, indem Sie den Anfangsbuchstaben des Vornamens sowie den Anfangsbuchstaben des Geburtsmonats erfragen und die Summe aus den ASCII-Code-Ordnungszahlen der beiden Zeichen als Startwert für den Pseudozufallszahlengenerator verwenden.

3.5 Anweisungen

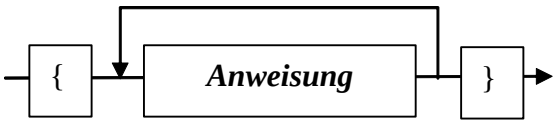
Wir haben uns in diesem Kurs zunächst mit (lokalen) **Variablen** und **Standarddatentypen** vertraut gemacht. Dann haben wir gelernt, aus Variablen, Literalen und Funktionsaufrufen mit Hilfe von **Operatoren** komplexe **Ausdrücken** zu bilden. Diese wurden entweder auf dem Bildschirm ausgegeben oder in Wertzuweisungen verwendet.
Die **main()**-Funktionen unserer Beispielprogramme bestanden aus einer Sequenz von **Anweisungen**, die wir nach bedarfsgerecht eingeführten Regeln gebildet haben. Nun werden wir uns systematisch mit dem allgemeinen Begriff einer Anweisung und vor allem mit wichtigen Spezialfällen befassen.

3.5.1 Überblick

Die Funktionen eines C++ - Programms bestehen aus Anweisungen (engl. statements). Nach der Lektüre von Abschnitt 3.5 werden Sie die folgenden Typen kennen:

- **Variablendeklarationsanweisung**
  Die Variablendeklarationsanweisung wurde schon in Abschnitt 3.2.2 eingeführt.
  Beispiel: `int i = 1, k;`
- **Ausdrucksanweisungen**
  Grundsätzlich kann man aus jedem Ausdruck eine Anweisung erstellen, indem man ein Semikolon dahinter setzt. Sinnvoll (und vom Compiler ohne Warnung akzeptiert) ist dieses Vorgehen jedoch nur bei speziellen Ausdrücken:
  - Wertzuweisungen (vgl. Abschnitt 3.2.3)
    Beispiel: `k = i + j;`
  - Verwendung eines Prä- bzw. Postinkrement- oder dekrementoperators
    Beispiel: `i++;`
    Hier ist nur der „Nebeneffekt“ des Ausdrucks `i++` von Bedeutung. Sein Wert bleibt ungenutzt.
  - Funktionsaufrufe
    Beispiel: `srand(vn + mon);`
- **Leere Anweisung**
  Beispiel: `;`
  Die durch ein einsames Semikolon ausgedrückte leere Anweisung hat keinerlei Effekte und kann dann verwendet werden, wenn die Syntax eine Anweisung verlangt, aber Nichts geschehen soll.
- **Anweisungen zur Ablaufsteuerung**
  Die bisherigen Beispielprogramme (genauer: die **main()**-Funktionen) bestanden meist aus einer *Sequenz* von Anweisungen, die bei jedem Programmlauf alle nacheinander ausgeführt wurden. Oft möchte man jedoch z.B.
  - die Ausführung einer Anweisung oder eines Anweisungsblocks von einer Bedingung abhängig machen
  - oder einen Anweisungsblock wiederholt ausführen lassen

  Für solche Zwecke stellt C++ etliche Auswahl- bzw. Wiederholungsanweisungen zur Verfügung, die bald ausführlich behandelt werden. In syntaktischer Hinsicht handelt es sich dabei jeweils um *eine* zusammengesetzte Anweisung.
- **Verbund- bzw. Blockanweisung**
  Einen Block von Anweisungen, die durch geschweifte Klammern zusammengefasst bzw. abgegrenzt werden, bezeichnet man als Verbund- bzw. Blockanweisung.



Damit fällt ein Anweisungsblock unter den Begriff *Anweisung*, und darf folglich überall dort stehen, wo eine Anweisung erlaubt ist.
Wie gleich näher erläutert wird, fasst man z.B. *dann* mehrere Abweisungen zu einem Block zusammen, wenn diese Anweisungen unter einer (gemeinsamen) Bedingung ausgeführt werden sollen. Es wäre ja sehr unpraktisch, dieselbe Bedingung für jede betroffene Anweisung wiederholen zu müssen.
In Abschnitt 3.2.4 haben wir uns im Zusammenhang mit dem Gültigkeitsbereich für Variablen bereits mit Anweisungsblöcken beschäftigt.

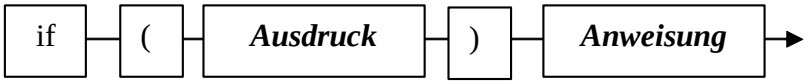
Als Syntaxregel halten wir fest, dass jede Anweisung mit einem Semikolon abgeschlossen werden muss, falls sie nicht mit einer schließenden Block-Klammer endet.

Im weiteren Verlauf von Abschnitt 3.5 beschäftigen wir uns mit Anweisungen zur Ablaufsteuerung.

3.5.2 Auswahlanweisungen

3.5.2.1 if-Anweisung

Mit der folgenden Syntax sorgt man dafür, dass eine Anweisung nur dann ausgeführt wird, wenn ein Ausdruck einen von Null verschiedenen Wert liefert (den logischen Wert **true** annimmt):



Im ersten Beispiel wird in Abhängigkeit von einem logischen Ausdruck (mit potentiellen Werten 0 und 1) eine einzelne Anweisung ausgeführt:

```
if (anz <= 0)
  cout << "Die Anzahl muss > 0 sein!\n";
```

Im zweiten Beispiel entscheidet der Wert einer numerischen Variablen darüber, ob eine Verbundanweisung ausgeführt wird:

```
if (k) {
  i++;
  cout << "k ist ungleich 0\n";}
```

Wie Sie aus Abschnitt 3.1.1 wissen, dient der Zeilenumbruch zwischen dem logischen Ausdruck und dem Anweisung(sblock) nur der Übersichtlichkeit und ist aus der Sicht des Compilers irrelevant.
Im zweiten Beispiel wurden die Blockklammern möglichst Platz sparend gesetzt.

Soll auch Etwas passieren, wenn der Ausdruck den Wert 0 (**false**) liefert, erweitert man die **if**-Anweisung um eine **else**-Klausel.
Zur Beschreibung der **if**-**else** - Anweisung wird an Stelle eines Syntaxdiagramms eine alternative Darstellungsform gewählt, die sich am typischen C++ - Quellcode-Layout orientiert:

if (***Ausdruck***)
  Anweisung 1
else
  Anweisung 2

Wie bei den Syntaxdiagrammen gilt auch für diese Form der Syntaxbeschreibung:

- Für Sprachbestandteile, die exakt in der angegebenen Form in konkreten Quellcode zu übernehmen sind, wird der Standard-Zeichensatz verwendet.
- *Platzhalter* sind durch ***fett-kursive*** Schrift gekennzeichnet.

Während die Syntaxbeschreibung im Quellcode-Layout sehr übersichtlich ist, bietet das Syntaxdiagramm den Vorteil, bei komplizierter, variantenreicher Syntax alle zulässigen Formulierungen kompakt und präzise als Pfade durch das Diagramm zu beschreiben.

Im folgenden **if-else** - Beispiel wird der natürliche Logarithmus zu einer Zahl geliefert, falls diese positiv ist. Anderenfalls wird eine Fehlermeldung mit Alarmton (Escape-Sequenz `\a`) ausgegeben:

Quellcode	Ein- und Ausgabe

`#include <iostream>`
`#include <cmath>`
`using namespace std;`
 	
`void main()`
`{`
`  double arg;`
 	
`  cout << "Argument: ";`
`  cin >> arg;`
 	
`  if (arg > 0)`
`  cout << "Log:  " << log(arg) << endl;`
`  else`
`  cout << "\aDer Log. ist nur f. pos. Arg. def.!\n";`
`}`
	`Argument: 2`
	`Log:  0.693147`

Unter den bedingt auszuführenden Anweisungen dürfen durchaus wiederum **if**- bzw. **if-else**-Anweisungen auftreten, so dass sich komplexere Fallunterscheidungen formulieren lassen:

if (***Ausdruck 1***)
  Anweisung 1
else if (***Ausdruck 2***)
  Anweisung 2
  . . .
  . . .
else if (***Ausdruck k***)
  Anweisung k
else
  Default-Anweisung

Enthält die letzte **if**-Anweisung eine **else**-Klausel, wird die zugehörige Anweisung ausgeführt, wenn alle logischen Ausdrücke den Wert 0 (**false**) haben. Die Wahl der Bezeichnung *Default-Anweisung* in der Syntaxdarstellung orientierte sich an der im Anschluss vorzustellenden **switch**-Anweisung.

Beim Schachteln von bedingten Anweisungen kann es zum genannten **dangling-else** - Problem kommen, wobei ein Missverständnis zwischen Compiler und Programmierer hinsichtlich der Zuordnung einer **else**-Klausel besteht. Im folgenden Codefragment lässt die Einrückung vermuten, dass der Programmierer die **else**-Klausel der *ersten* **if**-Anweisung zuordnen wollte:

```
if (i > 0)
   if (j > i) k = j;
else
   k = i;
```

Der Compiler ordnet die **else**-Klausel jedoch dem in Aufwärtsrichtung nächstgelegenen **if** zu, das noch keine **else**-Klausel besitzt. In unserem Beispiel bezieht er die **else**-Klausel also auf die *zweite* **if**-Anweisung.

Mit Hilfe von Blockklammern (nötigenfalls auch für eine einzelne Anweisung) kann die gewünschte Zuordnung erzwungen werden:

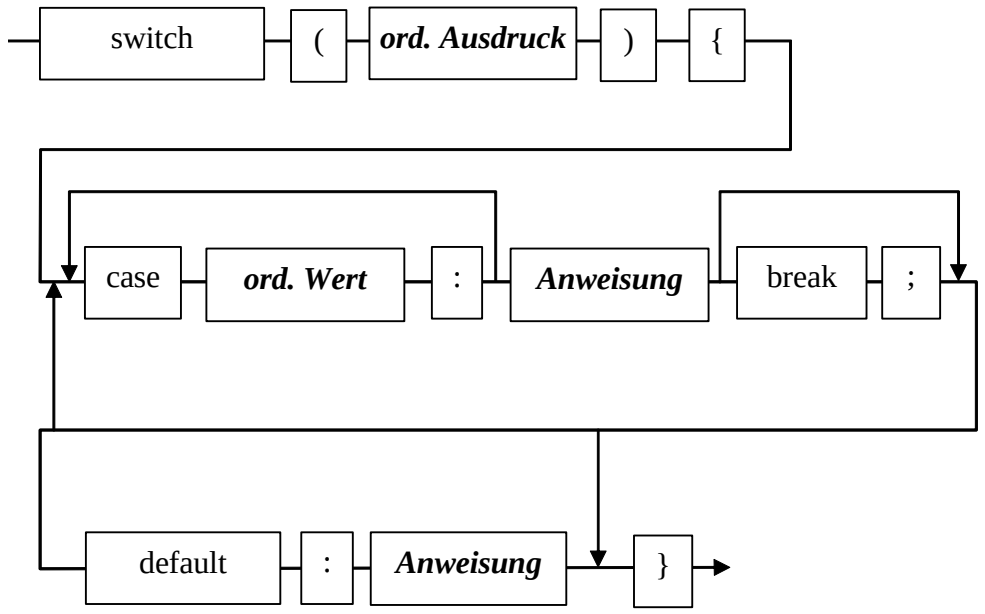
```
if (i > 0)
   {if (j > i) k = j;}
else
   k = i;
```

Alternativ könnte man auch dem zweiten **if** eine **else**-Klausel spendieren und dabei eine leere Anweisung verwenden:

```
if (i > 0)
   if (j > i) k = j;
   else;
else
   k = i;
```

3.5.2.2 switch-Anweisung

Wenn eine Fallunterscheidung mit mehr als zwei Alternativen in Abhängigkeit von einem ordinalen[1] Ausdruck vorgenommen werden soll, dann ist die **switch**-Anweisung weitaus handlicher als eine verschachtelte **if**-**else**-Konstruktion.
Der Genauigkeit halber wird die **switch**-Anweisung mit einem Syntaxdiagramm beschrieben. Wer die Pseudocode-Darstellung bevorzugt, kann ersatzweise einen Blick auf das gleich folgende Beispiel werfen.



Als ***ordinale Werte*** sind Konstanten und Literale von ganzzahligem Datentyp erlaubt.
Stimmt der Wert des ordinalen Ausdrucks mit einem Auswahlwert überein, wird die zugehörige Anweisung ausgeführt, ansonsten (falls vorhanden) die **default**-Anweisung. Nach dem Ausführen einer „angesprungenen“ Anweisung wird die **switch**-Konstruktion jedoch nur dann verlassen, wenn der Fall mit einer **break**-Anweisung abgeschlossen wird. Ansonsten werden auch noch die Anwei-

[1] Dieser oft verwendete Begriff ist in unserem Kontext synonym zu *ganzzahlig*.

sungen der nächsten Fälle (ggf. inkl. **default**) ausgeführt, bis der „Durchfall“ nach unten entweder durch eine **break**-Anweisung gestoppt wird, oder die **switch**-Anweisung endet. Mit dem etwas gewöhnungsbedürftigen **Durchfall**-Prinzip kann man für geeignet angeordnete Fälle sehr elegant kumulative Effekte kodieren, aber auch ärgerliche Programmierfehler durch vergessene **break**-Anweisungen produzieren.
Soll für mehrere Auswahlwerte dieselbe Anweisung ausgeführt werden, setzt man die zugehörigen **case**-Marken unmittelbar hintereinander und lässt die Anweisung auf die letzte Marke folgen. Leider gibt es keine Möglichkeit, eine *Serie* von Fällen durch Angabe der Randwerte festzulegen.
Im folgenden Beispielprogramm wird die Persönlichkeit des Benutzers mit Hilfe seiner Farb- und Zahlpräferenzen analysiert. Während bei einer Vorliebe für Rot oder Schwarz die Diagnose sofort fest steht, wird bei den restlichen Farben auch die Lieblingszahl berücksichtigt:

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

void main()
{
   char farbe;
   int  zahl;

   cout << "Waehlen Sie Ihre Lieblingsfarbe?\n\n"
       << "Tippen Sie ...\n   'r' fuer Rot\n   'g' fuer Gruen\n   'b' fuer Blau\n"
       << "   's' fuer Schwarz\n\nIhre Wahl: ";
   cin >> farbe;

   switch (farbe)
   {
      case 'r':  cout << "\nSie sind ein emotionaler Typ.\n\n"; break;
      case 'g':
      case 'b':  {
                   cout << "\nSie scheinen ein sachlicher Typ zu sein\n"
                      << "Fuer eine genauere Diagnose werden weitere Daten benoetigt.\n\n"
                      << "Waehlen Sie bitte eine Zahl zwischen 0 und 10: ";
                   cin >> zahl;
                   if (zahl%2 == 0)
                      cout << "\nSie haben einen geradlinigen Charakter.\n\n";
                   else
                      cout << "\nSie machen wohl gerne krumme Touren.\n\n";
                 }
                 break;
      case 's':  cout << "\nNehmen Sie nicht Alles so tragisch.\n\n"; break;
      default:   cout << "\nOffenbar mangelt es Ihnen an Disziplin.\n\n";
   }
}
```

3.5.3 Wiederholungsanweisung

Eine Wiederholungsanweisung (oder schlicht: Schleife) kommt zum Einsatz, wenn eine Anweisung *mehrfach* ausgeführt werden soll, wobei sich in der Regel schon der Gedanke daran verbietet, die Anweisung entsprechend oft in den Quelltext zu schreiben.
Bei leicht vereinfachender Betrachtung kann man hinsichtlich der Schleifensteuerung unterscheiden:

- **Zählergesteuerte Schleife** (**for**)
  Die Anzahl der Wiederholungen steht typischerweise schon vor Schleifenbeginn fest. Zur Ablaufsteuerung wird am Anfang eine Zählvariable initialisiert und dann bei jedem Durchlauf aktualisiert. Die zur Schleife gehörige Anweisung wird ausgeführt, solange die Zählvariable eine festgelegte Grenze nicht überschreitet.
- **Bedingungsabhängige Schleife** (**while**, **do**)
  Bei jedem Schleifendurchgang wird eine Bedingung überprüft, und das Ergebnis entscheidet über das weitere Vorgehen:

- **true**: die zur Schleife gehörige Anweisung ein weiteres Mal ausführen
- **false**: Schleife beenden

Bei der *kopfgesteuerten* **while**-Schleife wird die Bedingung *vor Beginn* eines Durchgangs geprüft, bei der *fußgesteuerten* **do**-Schleife hingegen *am Ende*.

Die gesamte Konstruktion aus Schleifensteuerung und (Verbund-)anweisung stellt in syntaktischer Hinsicht *eine* zusammengesetzte Anweisung dar.

3.5.3.1 Zählergesteuerte Schleife

Die Anweisung einer **for**-Schleife wird ausgeführt, solange eine Bedingung erfüllt ist, die normalerweise auf eine ganzzahlige Indexvariable Bezug nimmt. Die syntaktische Struktur der **for**-Schleife ermöglicht es, eine Initialisierung der Indexvariablen, die Spezifikation der Abbruchbedingung und die Inkrementierungsvorschrift an exponierter Stelle im Schleifenkopf unter zu bringen:

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

Zu den drei Bestandteilen des Schleifenkopfes sind einige Erläuterungen erforderlich, wobei hier etliche weniger typische bzw. sinnvolle Möglichkeiten weggelassen werden:

- ***Initialisierung***
  In der Regel wird man sich auf *eine* Indexvariable beschränken und dabei einen ordinalen Typ wählen. Diese Variable kann im Schleifenkopf auf einen Startwert gesetzt und nötigenfalls dabei auch deklariert werden, z.B.:

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

void main()
{
   int n, sq = 0;
   cout << "Summe der Quadrate von i = 1,...,n berechnen\n\n";
   cout << "Obergrenze: ";
   cin >> n;

   for (int i = 1; i <= n; i++) sq += i*i;

   cout << "\n\nSumme = " << sq << "\n\n";
}
```

  Deklarationen und Initialisierungen werden vor dem ersten Durchlauf ausgeführt. Eine im Schleifenkopf deklarierte Variable hat in Visual C++ als Gültigkeitsbereich den Block, in dem sich die **for**-Anweisung befindet. Hier weicht Visual C++ vom ANSI/ISO - C++ - Standard ab, der eine lokale, auf die **for**-Anweisung beschränkte Gültigkeit verlangt (wie in Java).
  Für eine Variablendeklaration im Schleifenkopf spricht die Nähe zum Einsatzort. Falls eine Variable jedoch auch außerhalb der Schleife benutzt wird, sollte sie möglichst an anderer Stelle definiert werden.

- ***Bedingung***
  Üblicherweise wird eine Ober- bzw. Untergrenze für eine zu in- bzw. dekrementierende Indexvariable gesetzt, doch erlaubt C++ beliebige Ausdrücke.
  Die Bedingung wird *zu Beginn* eines Schleifendurchgangs geprüft. Resultiert der Wert **true**, so wird der Anweisungsteil ausgeführt, anderenfalls wird die **for**-Anweisung verlassen. Folglich kann es auch passieren, dass überhaupt kein Durchlauf zustande kommt.

- ***Inkrementierung***
  Hier ist zu beachten, dass der Inkrementierungsausdruck *am Ende* eines Schleifendurchgangs (nach Ausführung der zugehörigen (Verbund-)Anweisung) ausgeführt wird.

Zu den (zumindest stilistisch) bedenklichen Konstruktionen, die der Compiler klaglos umsetzt, gehören **for**-Schleifenköpfe ohne Initialisierung, ohne Bedingung (wirkt wie **true**) oder ohne Inkrementierung, wobei die trennenden Strichpunkte trotzdem zu setzen sind. Es ist ebenfalls erlaubt, aber kaum zu empfehlen, die Indexvariable im Anweisungsteil der Schleife zu verändern.

3.5.3.2 Bedingungsabhängige Schleifen

Wie die Erläuterungen zur **for**-Schleife gezeigt haben, ist die Überschrift dieses Abschnitts nicht sehr trennscharf, weil bei der **for**-Schleife ebenfalls eine beliebige Terminierungsbedingung angegeben werden kann. In vielen Fällen ist es eine Frage des persönlichen Geschmacks, welche C++ - Wiederholungsanweisung man zur Lösung eines konkreten Iterationsproblems benutzt.
Unter der aktuellen Abschnittsüberschrift diskutiert man traditionsgemäß die **while**- und die **do**-Anweisung.

3.5.3.2.1 while-Anweisung

Die **while**-Anweisung kann als vereinfachte **for**-Anweisung beschreiben kann: Wer im Kopf einer **for**-Schleife auf Initialisierung und Inkrementierung verzichten möchte, ersetzt besser das Schlüsselwort **for** durch **while** und erhält dann folgende Syntax:

while (***Bedingung***) ***Anweisung***

Wie bei der **for**-Anweisung wird die Bedingung *zu Beginn* eines Schleifendurchgangs geprüft. Resultiert der Wert **true**, so wird der Anweisungsteil ausgeführt, anderenfalls wird die **while**-Anweisung verlassen.

Im obigen Beispielprogramm kann man nach minimalen Änderungen auch eine **while**-Schleife verwenden:

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

void main()
{
   int n, sq = 0, i = 1;
   cout << "Summe der Quadrate von i = 1,...,n berechnen\n\n";
   cout << "Obergrenze: ";
   cin >> n;

   while (i <= n)
   {
      sq += i*i;
      i++;
   }

   cout << "\n\nSumme = " << sq << "\n\n";
}
```

3.5.3.2.2 do-Anweisung

Bei der **do**-Schleife wird die Fortsetzungsbedingung *am Ende* der Schleifendurchläufe geprüft, so dass wenigstens *ein* Durchlauf stattfindet. Dies kommt auch in der Syntax zum Ausdruck:

do
   Anweisung
while (***Bedingung***);

do-Schleifen werden seltener benötigt als **while**-Schleifen, sind aber z.B. dann von Vorteil, wenn man vom Benutzer eine Eingabe mit bestimmten Eigenschaften einfordern möchte:

```
char antwort;
do
{
   cout << "Soll das Programm beendet werden (j/n)? ";
   cin >> antwort;
} while (antwort != 'j' && antwort != 'n' );
```

Wird ein Anweisungsblock verwendet, sollte die **while**-Klausel in einer Zeile direkt hinter die schließende Blockklammer gesetzt werden, um sie optisch von einer selbständigen **while**-Anweisung abzuheben.

Bei jeder Wiederholungsanweisung (**for**, **while** oder **do**) kann es in Abhängigkeit vom logischen Ausdruck passieren, dass der Anweisungsteil unendlich oft ausgeführt wird. Endlosschleifen sind als gravierende Programmierfehler unbedingt zu vermeiden. Befindet sich ein Programm in diesem Zustand muss es mit Hilfe des Betriebssystems abgebrochen werden, bei unseren Konsolenanwendungen z.B. über die Tastenkombination **<Strg>+<C>**.

3.5.3.3 Schleifen(durchgänge) vorzeitig beenden

Mit der **break**-Anweisung, die uns schon im Zusammenhang mit der **switch**-Anweisung begegnet ist, kann eine **for**-, **while**- oder **do**-Schleife vorzeitig verlassen werden. Mit der **continue**-Anweisung veranlasst man C++, den aktuellen Schleifendurchgang zu beenden und sofort mit dem Nächsten zu beginnen. In der Regel kommen **break** bzw. **continue** im Rahmen einer Auswahlanweisung zum Einsatz, z.B. in folgendem Programm zur (relativ primitiven) Primzahlendiagnose:

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

void main()
{
   bool tg;
   int i, zahl;
   double limit;
   cout << "Primzahlen-Diagnose\n-------------------\n\n";
   while (true)
   {
      cout << "\nZu untersuchende Zahl > 2 oder 0 zum Beenden: ";
      cin >> zahl;
      if (zahl <= 2  &&  zahl != 0)
      {
         cout << "Falsche Eingabe!\n";
         continue;
      }
      if (zahl == 0) break;
      tg = false;
      limit = sqrt(double(zahl));
      for (i = 2; i <= limit; i++)
         if (zahl % i == 0)
         {
            tg = true;
            break;
         }
      if (tg)
         cout << zahl << " ist keine Primzahl (Teiler: " << i << ").\n";
      else
         cout << zahl << " ist eine Primzahl.\n";
   }
   cout << "\n\nVielen Dank fuer den Einsatz der Primzahlen-Diagnose!\n";
}
```

Man hätte die **continue**- bzw. **break**-Anweisungen zwar mit einer geeigneten Auswahlanweisungen vermeiden können, doch werden bei dem vorgeschlagenen Verfahren die lästigen Spezialfälle auf besonders übersichtliche Weise abgehakt, bevor der „Kern-Algorithmus“ startet.

Zum Kern-Algorithmus der obigen Primzahlen-Suche sollte vielleicht noch erläutert werden, warum die Suche bei der Wurzel des Kandidaten beendet werden kann:
Sei *d* ein Teiler der positiven, ganzen Zahl *z*, d.h. es gibt eine Zahl *k* (≥ 2) mit

$$z = k \cdot d$$

Dann ist auch *k* ein Teiler von *z*, und es gilt:

$$d \leq \sqrt{z} \quad \text{oder} \quad k \leq \sqrt{z}$$

Anderenfalls wäre das Produkt $k \cdot d$ ja größer als *z*. Wir haben also folgendes Ergebnis: Wenn eine Zahl *z* keine Primzahl ist, dann hat sie einen Teiler kleiner oder gleich $\sqrt{z}$.

3.5.4 Übungsaufgaben zu Abschnitt 3.5

1) Schreiben Sie ein Programm, das ein Roulette-ähnliches Glücksspiel simuliert. Zwei Mitspieler investieren den selben Einsatz und akzeptieren den folgenden Gewinnplan:

Ergebnis	Wahrscheinlichkeit	Spielausgang
Bank	$\frac{1}{4}$	Beide Spieler verlieren ihren Einsatz an die Bank.
Rot	$\frac{3}{8}$	Spieler A gewinnt das eingesetzte Geld.
Schwarz	$\frac{3}{8}$	Spieler B gewinnt das eingesetzte Geld.

Tipp: Zum Initialisieren des Pseudozufallszahlengenerators eignet sich die von der Funktion **time()** gelieferte Systemzeit, z.B. in folgendem Aufruf:

```
srand(time(NULL));
```

2) Wie oft wird die folgende **while**-Schleife ausgeführt?

```
int i = 0;
while (i < 100);
{
   i++;
   cout << i << endl;
}
```

3) Verbessern Sie das im Übungsaufgaben-Abschnitt 0 in Auftrag gegebene Programm zur Berechnung von Netto-Beträgen so, dass es nicht für jeden Betrag neu gestartet werden muss. Vereinbaren Sie mit dem Benutzer eine geeignete Methode für den Fall, dass er das Programm doch irgendwann einmal beenden möchte.

4) Erstellen Sie eine Variante des Primzahlen-Testprogramms aus Abschnitt 3.5.3.3, die ohne **break**- bzw. **continue**-Anweisung auskommt.

4 Strukturierung und Modularisierung von Programmen

Wir haben uns bereits damit beschäftigt, wie vordefinierte Bibliotheksfunktionen (z.B. aus der C++ - Standardbibliothek) verwendet werden können. Dabei schicken wir einen Auftrag mit den erforderlichen Eingabedaten an eine Funktion (an ein Unterprogramm) und erhalten ein Ergebnis, ohne uns um die Implementierung (im Rahmen einer Objektcode-Bibliothek) kümmern zu müssen.
Durch die Vorarbeit anderer Programmierer stehen fertige Funktionen zur Lösung von Standardaufgaben zur Verfügung, so dass wir uns auf die eigene Problemstellung konzentrieren können. In der Regel ist diese nicht mit einem trivialen „Dreizeiler" zu lösen, sondern erfordert eine sorgfältige Analyse, also (laut Duden) eine „systematische Untersuchung eines Gegenstandes oder Sachverhaltes hinsichtlich aller einzelnen Komponenten oder Faktoren, die ihn bestimmen". Nach dem altbewährten Motto „teile und herrsche" sollte man ein Gesamtproblem möglichst in Teilprobleme zerlegen. Ein sinnvolles C-Programm zu einer ernsthaften Aufgabenstellung besteht aus mehreren, möglichst nicht allzu großen, Funktionen für die bei einer Problemanalyse isolierten Teilaufgaben.
Durch diese Strukturierung ergeben sich übersichtliche und kompakte Programme, die relativ leicht entwickelt, analysiert, getestet, korrigiert und erweitert werden können. Insbesondere können die in Funktionen implementierten Lösungen sehr gut in anderen Programmen/Projekten wieder verwendet und/oder an andere Programmierer weiter gegeben werden. Schließlich lässt sich ein strukturiertes Projekt sehr gut in Teamarbeit bewältigen.

Selbst erstellte Funktionen können innerhalb des Programms genauso aufgerufen werden wie vordefinierte Bibliotheksfunktionen. Das Definieren einer Funktion kann sich durch die bessere Übersichtlichkeit selbst dann lohnen, wenn sie nur ein einziges Mal aufgerufen wird. Richtig effizient wird die Unterprogrammtechnik vor allem dann, wenn eine Funktion wiederholt (z.B. an verschiedenen Stellen eines Programms) zum Einsatz kommt.

Hat man ein Problem in mehrere Funktionen zerlegt, ist es oft sinnvoll, im nächsten Schritt diese Funktionen auf mehrere Quellcodedateien aufzuteilen. Man gewinnt so genannte *Module*, die separat kompiliert werden können. Nach einer Modifikation des Programms müssen nur die veränderten Module neu übersetzt werden.

Für eine Menge von zusammengehörigen Funktionen, die in *mehreren* Programmen genutzt werden sollen, erstellt man am besten eine *Bibliothek*, die nach Fertigstellung analog zur Standardbibliothek verwendet werden kann. Mit diesem Thema werden wir uns in Abschnitt 4.5.3 beschäftigen.

Mit der objektorientierten Programmierung lässt sich das Ziel der weitgehenden Modularisierung eines Softwaresystems noch besser erreichen als mit den Techniken, auf die wir uns vorläufig beschränken. Dadurch werden jedoch die Inhalte des aktuellen Abschnitts nicht obsolet:

- Sie sind großenteils auch für die objektorientierte Programmierung relevant.
- Die kaum unbestreitbaren Vorteile der objektorientierte Programmierung kommen vornehmlich bei großen Projekten zum Tragen, so dass bei kleineren Problemen nach wie vor traditionelle Lösungen in Frage kommen.

4.1 Funktionsdefinition

Im Zusammenhang mit der Verwendung von Bibliotheksfunktionen haben wir uns schon mit Funktionsschnittstellen beschäftigt und z.B. erfahren, dass C++ neben „normalen" Funktionen mit Eingabeparametern und einem Rückgabewert auch zwei Spezialfälle kennt:

- Funktionen ohne Rückgabewert bzw. mit dem Rückgabetyp **void**
  Sie können wie die *Prozeduren* anderer Programmiersprachen eingesetzt werden.
- Funktionen ohne Parameter
  Hier darf die leere Parameterliste nicht weggelassen werden.

4.1.1 Beispiel

Bevor wir die Syntax und Semantik einer Funktionsdefinition im Detail besprechen, soll ein Beispiel vorgestellt werden. In der folgenden Variante des Primzahlendiagnoseprogramms aus dem Abschnitt über Wiederholungsanweisungen wird die Gesamtaufgabe folgendermaßen aufgeteilt:

- Die Funktion `ask()` fragt den Benutzer nach der zu untersuchenden Zahl und meldet diese an den Aufrufer zurück.
- Die Funktion `primd()` nimmt die Primzahlendiagnose vor. Sie liefert für ein ganzzahliges Argument:
  - Als Rückgabewert **true** bzw. **false** je nach Ergebnis des Primzahlentests
  - Über den Referenzparameter (s.u.) `kt` den kleinsten Teiler (größer 2), falls das Argument keine Primzahl ist.
- Die Funktion `main()` kann sich im Wesentlichen darauf beschränken, die anderen Funktionen aufzurufen.

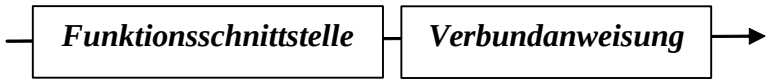
Soll später z.B. der Benutzerdialog verbessert werden, um auf falsche Eingaben sinnvoll reagieren zu können, dann muss nur die Funktion `ask()` geändert werden. Wird in einem anderen Programm eine Primzahlendiagnose benötigt, dann kann die Funktion `primd()` leicht übernommen werden.

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

int ask()
{
   int zahl;
   while (true)
   {
      cout << "\nZu untersuchende Zahl > 2 oder 0 zum Beenden: ";
      cin >> zahl;
      if (zahl <= 2  &&  zahl != 0)
         cout << "Falsche Eingabe!\n";
      else
         return zahl;
   }
}

bool primd(int zahl, int &kt)
{
   bool tg = false;
   int limit = int(sqrt(double(zahl)));
   for (int i = 2; i <= limit; i++)
      if (zahl % i == 0)
      {
         kt = i;
         return false;
      }
   return true;
}

void main()
{
   int arg, t;
   cout << "Primzahlen-Diagnose\n-------------------\n\n";
   while (arg = ask())
      if (primd(arg, t))
           cout << arg << " ist eine Primzahl.\n";
      else
           cout << arg << " ist keine Primzahl (Teiler: " << t << ").\n";
}
```

4.1.2 Funktionsschnittstelle

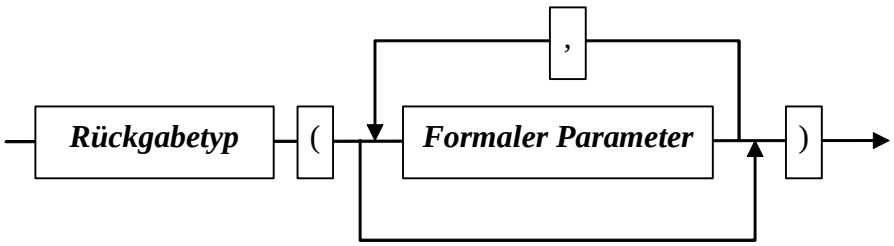
Die Funktionsdefinition besteht aus einer **Funktionsschnittstelle**, wie sie uns bereits bei den vordefinierten Bibliotheksfunktionen begegnet ist, und einer **Verbundanweisung**:

Funktionsdefinition



Weil der Begriff *Funktionsschnittstelle* nun von unserem „passiven“ C++ - Wortschatz in den aktiven übergehen soll, betrachten wir die exakte Definition durch ein Syntaxdiagramm:

Funktionsschnittstelle



Zwei Hinweise zum Rückgabetyp:

- Soll eine Funktion *keinen* Rückgabewert liefern, ordnet man ihr den leeren Typ **void** zu.
- Wenn Sie von der (im Syntaxdiagramm nicht erwähnten) Möglichkeit Gebrauch machen, den Rückgabetyp wegzulassen, wird der Typ **int** verwendet.

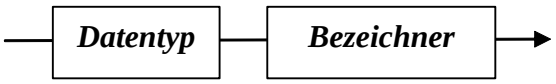
Die formalen Parameter spezifizieren Daten von bestimmtem Typ, die entweder den Ablauf der Funktion steuern, oder durch die Funktion verändert werden sollen. Beim späteren Aufruf der Funktion sind diese durch *Aktualparameter* zu ersetzen, wobei Variablen oder Ausdrücke in Frage kommen.
Bevor wir die Syntaxvorschriften für Formalparameter betrachten, soll noch eine wichtige Einteilung der Parameter nach ihrem Verhalten bei der Wertübergabe behandelt werden:

- **Wertparameter**
  Ein **Wertparameter** verhält sich innerhalb der Funktion wie eine lokale Variable, die durch den im Aufruf übergebenen Wert initialisiert wird. Funktionsinterne Änderungen dieser lokalen Variablen bleiben ohne Effekt auf eine als Aktualparameter fungierende Variable der rufenden Programmeinheit, was als Einschränkung oder Schutz interpretiert werden kann.
  Beim Funktionsaufruf sind als Wertparameter nicht nur Variablen erlaubt, sondern beliebige Ausdrücke mit kompatiblem Typ.
- **Referenzparameter**
  Beim **Referenzparameter** erhält die Funktion *keine* lokale Kopie der betroffenen Variablen, sondern die Speicheradresse des Originals. Alle funktionsintern vorgenommenen Modifikationen wirken sich direkt auf das Original aus. So wird es einer Funktion ermöglicht, mehr als eine Variable in der aufrufenden Programmeinheit zu verändern.
  Außerdem wird bei Variablen mit großem Speicherplatzbedarf (, die wir bisher noch nicht kennen,) die eventuell sehr aufwändige Wertübergabe vermieden.
  Beim Funktionsaufruf sind als Referenzparameter nur Variablen erlaubt.

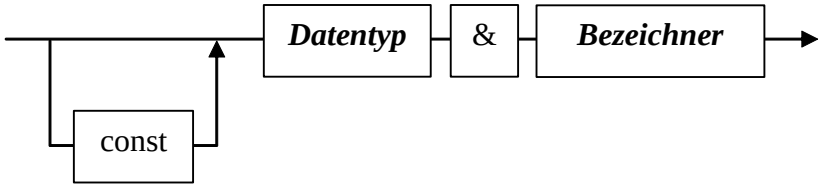
- **Konstante Referenzparameter**
  Bei einem konstanten Referenzparameter erhält die Funktion zwar die Speicheradresse des Originals, darf jedoch den dortigen Zustand nicht ändern[1]. Damit wird die Rechenzeit- und Speicherplatzökonomie der Referenzparameter mit dem Schutzeffekt der Wertparameter kombiniert. Außerdem kann wie bei Wertparametern im Funktionsaufruf ein beliebiger Ausdruck übergeben werden. Damit sollten Wertparameter aus Performanzgründen durch konstante Referenzparameter ersetzt werden, wenn Aktualparameter mit großem Speicherplatzbedarf zu erwarten sind.

Nach dieser wichtigen Einteilung der Parameter nach *semantischen* Gesichtspunkten vervollständigen wir die Syntaxregeln für eine Funktionsdefinition:

Formaler Wertparameter



Formaler Referenzparameter



Bei der Funktionsdefinition wird dem Namen eines formalen Referenzparameters das Zeichen **&** vorangestellt. Ob das Zeichen am Typbezeichner oder am Parameternamen klebt, oder von beiden einen Abstand halten sollte, ist eine Stilfrage, die den Compiler nicht interessiert.

Beim Funktions*aufruf* gibt es keine syntaktischen Unterschiede zwischen Wert- und Referenzparametern.

In obigem Beispiel wird mit der Funktionsschnittstelle

```
bool primd(int zahl, int &kt)
```

vereinbart:

- ein Rückgabewert vom Typ **bool**
- ein Wertparameter `zahl` vom Typ **int**
- ein Referenzparameter `kt` vom Typ **int**

Die Funktion soll mit dem Rückgabewert darüber informieren, ob der Wert von `zahl` eine Primzahl ist. Ist dies der Fall, soll sie zusätzlich den kleinsten Teiler über den Verweisparameter `kt` zurückmelden.

Parameter mit Voreinstellungen

Wird für einen Parameter beim Funktionsaufruf in der Regel ein bestimmter Wert benötigt, dann bietet sich folgendes Verfahren an, um die Benutzung der Funktion zu vereinfachen:

- Man setzt den Parameter in der Funktionsschnittstelle ans Ende der Parameterliste und weist ihm einen **Voreinstellungswert** zu.
- Wird der fragliche Parameter bei einem konkreten Funktionsaufruf weggelassen, dann gilt der Voreinstellungswert.

---

[1] Später wird allerdings eine Möglichkeit vorgeführt, wie der der Schutz in der Funktions-Implementation wieder aufgehoben werden kann.

Im folgendem Beispiel wird eine Wurzelfunktion realisiert, die beim Aufruf mit *einem* Argument dessen Quadratwurzel liefert, optional aber über einen zweiten Parameter dazu veranlasst werden kann, eine Wurzel beliebiger Ordnung zu berechnen:

Quellcode	Ausgabe

`#include <iostream>`
`#include <cmath>`
`using namespace std;`
 	
`double wurzel(double basis, double ordnung = 2)`
`{`
`  return pow(basis, 1.0/ordnung);`
`}`
 	
`void main()`
`{`
`  cout << wurzel(9, 2) << endl;`
`  cout << wurzel(16) << endl;`
`  cout << wurzel(125, 3) << endl;`
`}`
	`3`
	`4`
	`5`

Man kann auch mehrere Voreinstellungsparameter definieren, die diese natürlich am Ende der Parameterliste stehen müssen.

4.1.3 Anweisungsblock

Nachdem wir uns ausführlich mit der Schnittstelle einer Funktion befasst haben, folgen nun noch einige Informationen zu ihrem Anweisungsblock. Dort ist vor allem die **return**-Anweisung von Interesse. Sie beendet die Ausführung einer Funktion und gibt die Kontrolle an die aufrufende Programmeinheit zurück. Dabei wird noch ein Wert abgeliefert, falls die Funktion nicht den Rückgabetyp **void** hat.

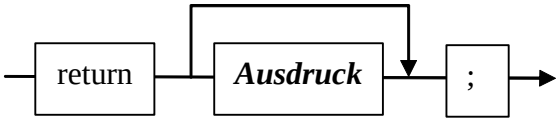
Wie unser Beispiel zeigt, müssen Sie sich nicht unbedingt auf *eine* **return**-Anweisung am Ende der Funktion beschränken, sondern können (in gewisser Analogie zum Verlassen einer Schleife per **break**-Anweisung) die Funktion an verschiedene Stellen mit **return** beenden und dabei einen passenden Wert zurückmelden. Einige Autoren raten jedoch der Übersichtlichkeit halber davon ab, mehrere **return**-Anweisungen zu verwenden (z.B. Lee & Phillips 1997,S. 515).

Bei einer Funktion *mit* Rückgabewert muss mindestens *eine* **return**-Anweisung vorhanden sein, und der Compiler warnt, wenn ein potentiell möglicher Ausführungs-Pfad ohne **return**-Anweisung endet:

```
warning C4715: 'test' : Nicht alle Steuerelementpfade geben einen Wert zurück
```

Diese Warnung sollte sehr ernst genommen werden, weil die aufrufende Programmeinheit eventuell auf einen sinnvollen Rückgabewert angewiesen ist und im fraglichen Fall versucht, mit einem undefinierten (zufälligen) Ergebnis weiter zu arbeiten.
Bei **void**-Funktionen ist keine **return**-Anweisung verlangt, sie kann jedoch (auch mehrfach) verwendet werden, um die Funktion zu verlassen

Das Syntaxdiagramm zur **return**-Anweisung:



Um den Ausdruck mit dem Rückgabewert dürfen optional runde Klammern gesetzt werden.

Zum Einsatz von Variablen in Funktionen wissen wir bereits:

- Die innerhalb des Anweisungsblocks einer Funktion definierten Variablen besitzen lokale Gültigkeit.
- Formalparameter können im Anweisungsblock einer Funktion wie lokale Variablen behandelt werden, wobei sich Wertveränderungen im Fall von Referenzparametern auch außerhalb der Funktion bemerkbar machen.

4.1.4 Gültigkeit

In C/C++ besitzen alle Funktionen **globale Gültigkeit**, sie können also im gesamten Programm von beliebigen Funktion aufgerufen werden.
Es gibt keine lokalen, innerhalb eines Anweisungsblocks definierten, Funktionen. Damit können Funktionen auch *nicht* verschachtelt werden (wie z.B. in Pascal/Delphi).

4.2 Abwicklung eines Funktionsaufrufs

Es lohnt sich, eine elementare Vorstellung davon zu gewinnen, was beim Funktionsaufruf hinter den Kulissen vor sich geht. Dabei spielt ein als *Stack* (dt.: Stapel) bezeichneter Bereich des programmeigenen Speichers eine zentrale Rolle. In einer vereinfachten Darstellung kann man folgende Schritte unterscheiden[1]:

- Aktualparameter in festgelegter Reihenfolge auf dem Stack abgelegen
- Rücksprungadresse auf dem Stack abgelegen
- Sprung zum Maschinencode der Funktion
- Lokale Variablen der Funktion auf dem Stack angelegen (ohne Initialisierung!)
- Anweisungsblock der Funktion ausführen
- Rückgabewert der Funktion in ein Register schreiben
  Dies ist ein Speicherplatz innerhalb des Prozessors und wegen der hohen Zugriffsgeschwindigkeit für die Kommunikation zwischen zwei Funktionen sehr gut geeignet.
- Lokale Variablen, Rücksprungadresse und Parameter vom Stapel entfernen, Rücksprung
  Es hängt teilweise von der von der so genannten Aufrufkonvention ab, wie sich die Aufräumarbeiten auf die aufgerufene Funktion und den Aufrufer verteilen.

Während der Prozessor die Befehle im Anweisungsblock einer Funktion ausführt, liegen also für den Aufruf folgende Informationen auf dem Stack:

[1] Eine weitaus präzisere Beschreibung der Speicherverwaltung und der Abläufe finden Sie z.B. bei Zimmermann (2002).

Wenn im Anweisungsblock weitere Funktionsaufrufe stattfinden, erhöht sich der Stapel entsprechend. Kommt es zum Konflikt mit der maximalen Stack-Größe (*stack overflow*), wird das Programm abgebrochen. Mit diesem Problem ist z.B. bei Funktionen zu rechnen, die sich selbst aufrufen. Vielleicht scheint es Ihnen ratsam, Merkwürdigkeiten wie den Selbstaufruf einer Funktion tunlichst zu vermeiden, um Problemen mit dem Stack-Speicher aus dem Weg zu gehen. In Abschnitt 4.7 wird aber demonstriert, dass über den Selbstaufruf einer Funktion sinnvolle rekursive Algorithmen realisiert werden können.

Es ist nur selten sinnvoll, das in Visual C++ 6.0 voreingestellte Stack-Maximum von 1 MB zu erhöhen. Wer an dieser Schraube doch einmal drehen möchte, findet sie bei den Linker-Einstellungen zum Projekt über:

Projekt > Einstellungen > Linker > Ausgabe

Der nicht ganz unerhebliche Aufwand bei einem Funktionsaufruf ist auch für Optimierungsüberlegungen des Compilers oder Programmierers relevant (vgl. Abschnitt 4.6 über **inline**-Funktionen).

4.3 Exkurs: Der exit code eines Programms

Für die Hauptfunktion **main()** gelten hinsichtlich des Rückgabewertes dieselben Spieleregeln wie für jede andere Funktion, wobei hier das Betriebssystem als Aufrufer und Empfänger des Rückgabewertes fungiert. Wir haben bisher für die **main()**-Funktion der Einfachheit halber stets den Rückgabetyp **void** verwendet, so dass am Ende der Funktion keine **return**-Anweisung erforderlich war. Nach dem Ausführen eines so erstellten Programms steht allerdings kein zuverlässiger **exit code** zur Verfügung.
Wird ein solcher benötigt (z.B. beim Aufruf eines Programms im Rahmen einer Skriptdatei), dann kann man für die **main()**-Funktion den Rückgabetyp **int** festlegen und geeignete **return**-Anweisungen formulieren, die das Programm beenden und dem Betriebssystem einen exit code übergeben, z.B.:

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

int main()
{
  char c;
  cout << "Akzeptieren Sie die Lizenzbedingungen? ";
  cin >> c;
  if (c == 'j')
  {
    cout << "Ihre Seriennummer lautet: 123\n";
    return 0;
  }
  else return 1;
}
```

Das folgende Perlskript startet das unter dem Namen **test.exe** erzeugte Beispielprogramm und überprüft anschließend seinen exit code:

```
$rc = (system "test.exe")/256;
print "\nexit code von test.exe: $rc\n";
```

Bei einer zweimaligen Ausführung des Perlskripts entstand folgender Dialog:

```
U:\Eigene Dateien\VCpp\Test\Debug>perl test.pl
Akzeptieren Sie die Lizenzbedingungen? j
Ihre Seriennummer lautet: 123

exit code von test.exe: 0

U:\Eigene Dateien\VCpp\Test\Debug>perl test.pl
Akzeptieren Sie die Lizenzbedingungen? n

exit code von test.exe: 1
```

An Stelle der **return**-Anweisung kann auch die **exit()**-Funktion verwendet werden:

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

void main()
{
  char c;
  cout << "Akzeptieren Sie die Lizenzbedingungen? ";
  cin >> c;
  if (c == 'j')
  {
    cout << "Ihre Seriennummer lautet: 123\n";
    exit(0);
  }
  else exit(1);
}
```

Während die **return**-Anweisung nur in **main()** zur direkten Beendigung des Programms führt, hat die Funktion **exit()** an beliebiger Stelle diesen Effekt.

4.4 Überladen von Funktionen

Wenn in einem Programm oft das Maximum zweier **int**-Zahlen benötigt wird, kann die in folgendem Quelltext definierte Funktion `max()` von Nutzen sein:

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

int max(int arg1, int arg2)
{
  if (arg2 > arg1)
    return arg2;
  else
    return arg1;
}

double max(double arg1, double arg2)
{
  if (arg2 > arg1)
    return arg2;
  else
    return arg1;
}

void main()
{
  cout << "Maximum von int-Werten:    " << max(12, 3) << endl << endl;
  cout << "Maximum von double-Werten: " << max(1.2, 3.5) << endl << endl;
}
```

Wenn auch das Maximum zweier **double**-Zahlen gefragt ist, muss (wie im Beispiel praktiziert) aufgrund der strengen Typisierung in C++ eine weitere Funktion definiert werden.[1]
Dass für die beiden Funktionen derselbe Name `max()` verwendet wird, stellt aber (zur Verwunderung der C-Programmierer) *keinen* Syntax-Fehler dar, sondern eine in C++ erlaubte **Überladung von Funktionen**.
Unser Beispielprogramm liefert nach dem fehlerfreien Übersetzen und Linken das Ergebnis:

```
Maximum von int-Werten:    12

Maximum von double-Werten: 3.5
```

In C++ dürfen Sie für verschiedene Funktionen den selben Namen verwenden, solange die Funktionssignaturen (also die Parameterlisten) verschieden sind.

Während *wir* uns die Gedächtnisbelastung durch typspezifische Funktionsnamen ersparen, muss der Compiler bei überladenen Funktionen anhand der Aktualparameter die richtige Instanz auswählen. Dies stellt kein Problem dar, solange die Aktualparameter-Typen perfekt zu einem Element aus der Menge der namensgleichen Funktionen passen. Aufgrund verständlicher Zweifel streikt der Compiler jedoch bei einer Erweiterung des obigen Beispielprogramms um die Zeile:

```
  cout << max(1, 3.5) << endl;
```

mit der Fehlermeldung:

```
error C2666: 'max' : 2 Ueberladungen haben aehnliche Konvertierungen
```

Die Suche nach der Funktion mit der „größten Ähnlichkeit“ zur Aktualparameterliste erbrachte kein eindeutiges Ergebnis. Wenn wie in folgendem Beispiel

```
cout << max(3.45, "2.55") << endl;
```

[1] Mit den später vorzustellenden *Templates* kann man die überladenen Funktionen gewissermaßen automatisch erzeugen lassen.

keine Instanz eine hinreichende Nähe zur Aktualparameterliste aufweist, beschwert sich der Compiler ebenfalls:

```
error C2665: 'max' : Durch keine der 2 Ueberladungen kann Parameter 2
                     vom Typ 'char [5]' konvertiert werden
```

Eine wesentliche Rolle bei der Ähnlichkeitsbestimmung spielen die impliziten Typumwandlungen, deren Risiken schon erwähnt wurden. Solange Sie diese vermeiden, werden Sie auch mit überladenen Funktionen keine Probleme haben.

4.5 Funktionen organisieren

4.5.1 Vorgezogene Schnittstellen-Deklaration

Vor dem ersten Aufruf einer Funktion muss der Compiler ihre Schnittstelle kennen. Wir haben diese Regel bisher eingehalten, indem wir alle Funktionen oberhalb der Hauptfunktion vollständig definiert haben (mit Schnittstelle und Ausführungsteil). Wenn auf diese Weise zahlreiche und/oder lange Funktionen vorweg zu definieren sind, kann der Quelltext unübersichtlich wirken. Eine erlaubte (und vom ANSI-Gremium sogar empfohlene) Alternative besteht darin, am Beginn des Quelltextes zu den benötigten Funktionen lediglich die Schnittstellen zu präsentieren und die vollständigen Definitionen zu einem späteren Zeitpunkt (nach der Hauptfunktion) nachzuliefern. Eine ohne Ausführungsteil stehende Funktionsschnittstelle wird auch als **Prototyp** bezeichnet. Wir haben es mit einer reinen *Deklaration* zu tun, die keine *Definition* ist.
In unserem letzten Beispiel mit den überladenen Funktionsnamen dürfen wir folgenden Quellcode verwenden:

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

int max(int arg1, int arg2);
double max(double, double);

void main()
{
  cout << "Maximum von int-Werten:    " << max(12, 3) << endl << endl;
  cout << "Maximum von double-Werten: " << max(1.2, 3.5) << endl << endl;
}

int max(int arg1, int arg2)
{
  if (arg2 > arg1)
    return arg2;
  else
    return arg1;
}

double max(double arg1, double arg2)
{
  if (arg2 > arg1)
    return arg2;
  else
    return arg1;
}
```

Bei der Syntax für einen Funktions-Prototypen ist zu beachten:

- Am Ende der Schnittstellendeklaration steht ein Semikolon.
- Es ist erlaubt, die Parameter*namen* wegzulassen, z.B.:
  `double max(double, double);`

4.5.2 Module

Hat sich in einem Projekt eine Anzahl von Unterprogrammen angesammelt, so sollten Sie diese in eine eigene, separat zu übersetzende, Quellcode-Datei auslagern und die Prototypen über eine Header-Datei für die Hauptfunktion verfügbar machen. Bei größeren Projekten kommen auch mehrere Quellcodedateien in Frage.
Beim Erstellen des Projektes mit  oder **<F7>** produziert bzw. erneuert der Compiler zu jeder geänderten Quellcodedatei die zugehörige Objektcodedatei (Extension: „.obj“) mit den Kompilaten. Anschließend setzt der Linker aus den Objektcodedateien und den benötigten Bibliotheksroutinen das lauffähige Programm zusammen.
Eine selbständig zu kompilierende Quellcodedatei bezeichnet man als *Quellmodul*, die zugehörige Objektcodedatei als *Objektmodul*.
Der Einsatz von Modulen hat u.a. folgende Vorteile:

- Der Quellcode wird übersichtlicher.
- Es wird Compilerzeit gespart, weil beim Erstellen des Projektes nur veränderte Quellmodule neu übersetzt werden.

Wir haben in Abschnitt 4 zwei wichtige Techniken zur rationellen Programmierung kennen gelernt:

- **Strukturierung**
  Ein Programm wird in viele Funktionen zerlegt, die eine bestimmte Teilaufgabe erfüllen
- **Modularisierung**
  Die Funktionen eines Programms werden auf verschiedene Quellcodedateien verteilt, aus denen jeweils ein Objektmodul entsteht.

Wir modularisieren nun das letzte Beispiel und zerlegen die in Abschnitt 4.5 wiedergegebene Quellcode-Datei in drei Teile:

- **Header-Datei**
  Die Prototypen der beiden `max`-Funktionen werden in die neue Header-Datei **max.h** verschoben. Diese Datei wird anschließend per **include**–Anweisung in den Quellcode mit der Hauptfunktion eingeblendet, damit der Compiler die `max`-Funktionen kennt und somit die Hauptfunktion übersetzen kann. Auf eine Header-Datei mit den Prototypen zu verzichten und die Quellcode-Datei mit den Funktions-Definitionen zu inkludieren, ist zwar erlaubt, aber nicht empfehlenswert. In diesem Fall würden mit **main()** grundsätzlich auch die `max`-Funktionen stets neu übersetzt.
  Erstellen Sie diese Datei über

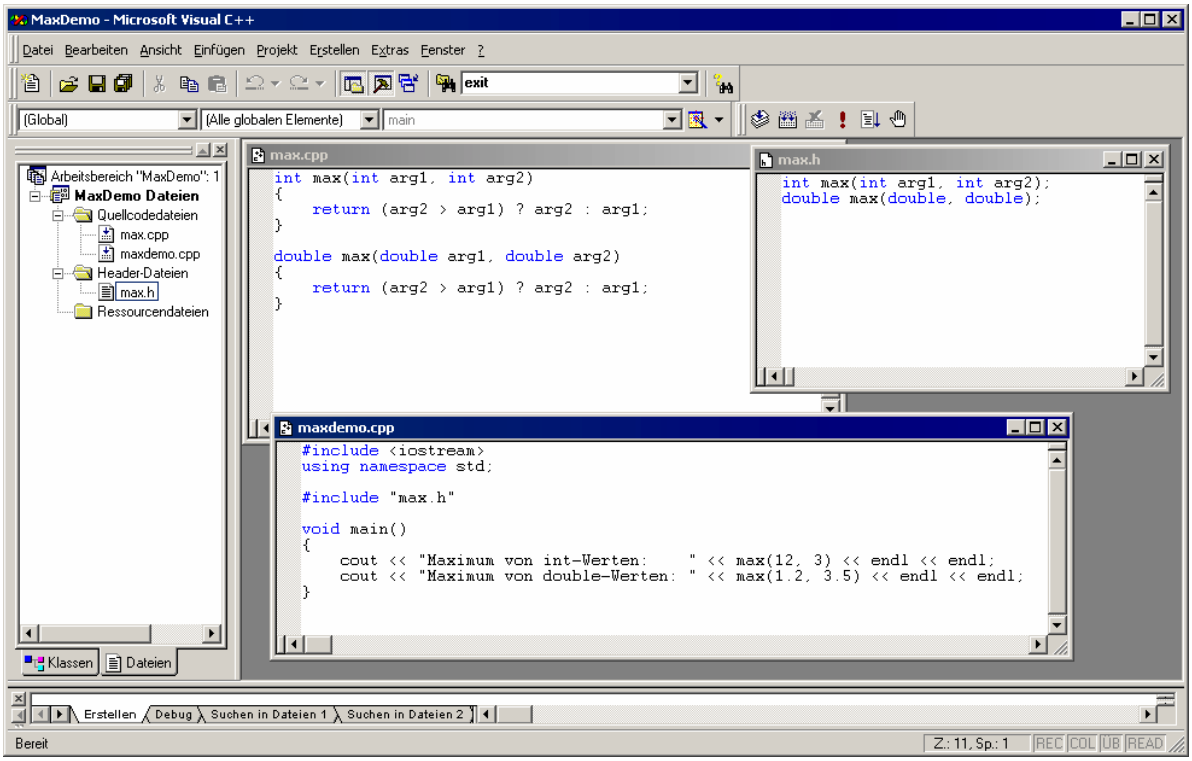
  Datei > Neu > Dateien > C/C++-Header-Datei

  Dabei kommen Sie nicht um die von Visual C++ bevorzugte Extension „.h“ herum, obwohl nach einer ANSI-Empfehlung zumindest in den Header-Dateinamen zu Standardbibliotheken neuerdings auf diese Extension verzichtet werden soll.
- **Quellmodul**
  Die Funktionsdefinitionen werden in die neue Quellcode-Datei **max.cpp** verschoben.
  Erstellen Sie diese Datei über

  Datei > Neu > Dateien > C++-Quellcodedatei

- **Quellcode-Datei mit der Hauptfunktion**
  In der ehemals einzigen Quellcode-Datei verbleiben jetzt nur noch:
  - **include**-Zeilen (und **namescape**-Anweisung)
    Die neue Header-Datei **max.h** wird beim Inkludieren in Hochkommata eingeschlossen, weil sie sich im Projektverzeichnis befindet.
  - Hauptfunktion

In der VC++ - Entwicklungsumgebung sieht das aus zwei Modulen bestehende Projekt nun folgendermaßen aus:



Nach dem nächsten Erstellen des Projektes finden Sie im Debug-Unterverzeichnis zum Projektordner u.a. die Objektdatei **max.obj**.

4.5.3 Eigene Bibliotheken

Wenn Sie Funktionen definieren, die in mehreren Programmen benötigt werden, dann sollten Sie daraus eine eigene Bibliothek erstellen, die analog zu den Standardbibliotheken verwendet werden kann. Bei der dabei entstehenden Datei mit der Namenserweiterung **.lib** handelt es sich im Wesentlichen um ein Objektmodul (Namenserweiterung **.obj**) , doch bietet sie u.a. folgende Vorteile:

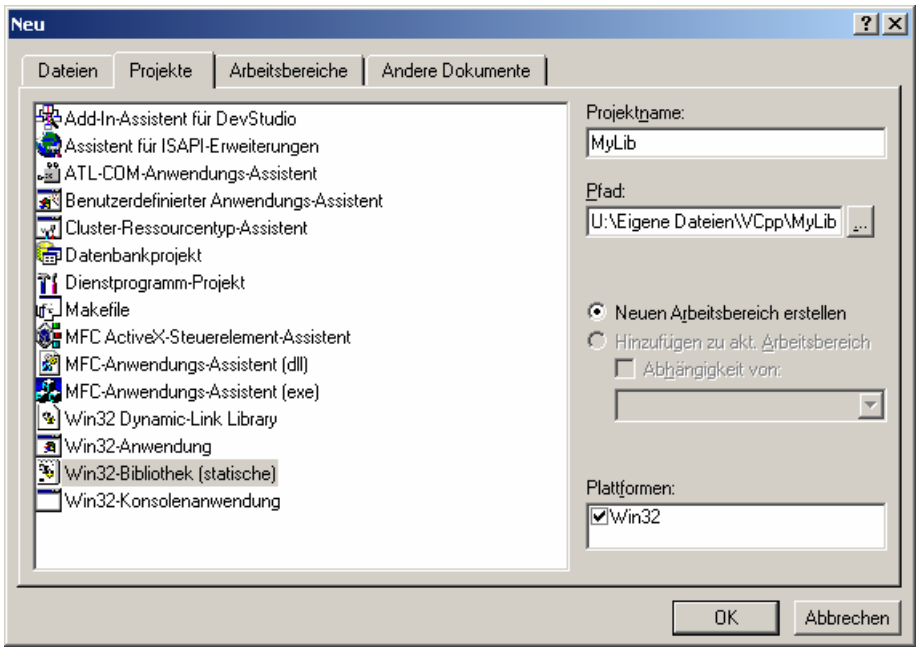
- Eine Bibliothek kann auf *mehreren* Quellcodedateien basieren, während einer **obj**-Datei stets genau *eine* Quellcodedatei zugrunde liegt.
- Visual C++ kennt für Bibliotheken spezielle Projekttypen, wobei der Linker beim Erstellen nicht auf der Existenz einer **main()**-Funktion besteht.

Im Folgenden wird eine mögliche Vorgehensweise am Beispiel der Bibliothek **MyLib.lib** beschrieben:

- Öffnen Sie in der Entwicklungsumgebung ein neues Projekt über

  Datei > Neu > Projekt

- Wählen Sie den Typ **Win32-Bibliothek (statische)** und den Projektnamen **MyLib**:



- Lassen Sie in der Dialogbox zum ersten Assistentenschritt das Projekt ohne Einstellungsänderungen **fertig stellen**, und quittieren Sie die Informationen zum neuen Projekt mit **OK**.
- Über

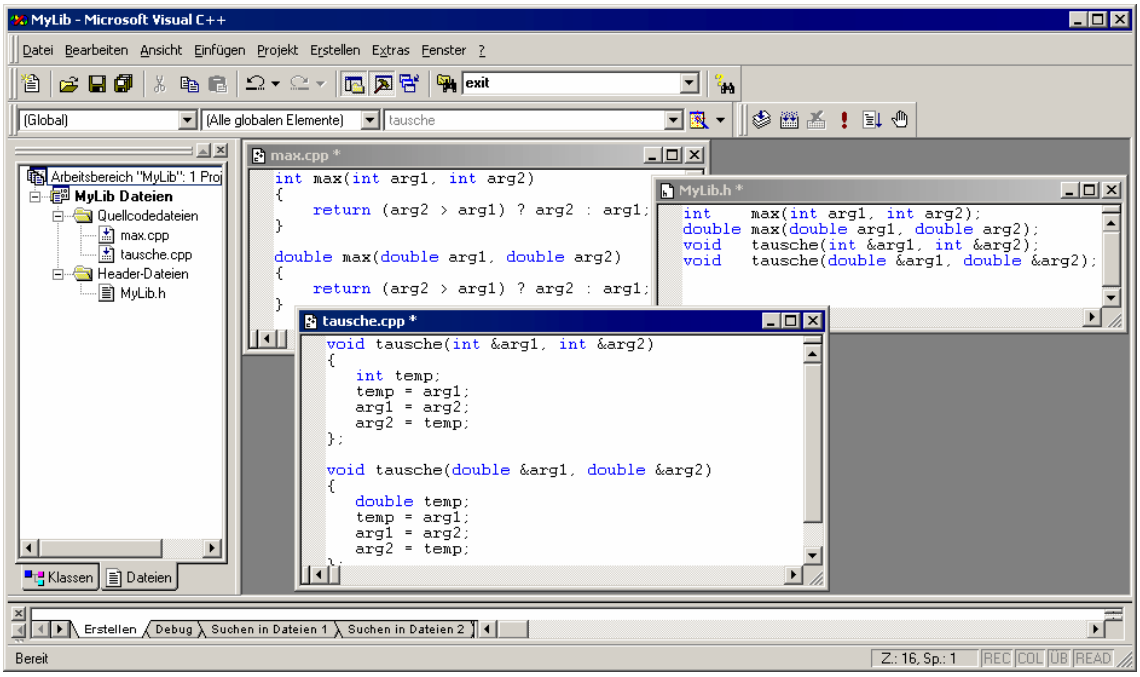
Datei > Neu > Dateien > C++-Quellcodedatei

können Sie eine Quellcodedatei in das Projekt aufnehmen, bei Bedarf auch mehrere. Verfassen Sie hier die Funktionsdefinitionen.

- Erstellen Sie über

Datei > Neu > Dateien > C/C++-Header-Datei

eine Header-Datei mit den Prototypen (Schnittstellen) der Bibliotheksfunktionen.
In folgendem Beispiel enthält die Header-Datei die Prototypen zu Funktionen, die in zwei Implementierungsdateien definiert werden:



- Lassen Sie das Projekt erstellen. Obiges Beispiel liefert im Ausgabefenster folgende Erfolgsmeldung:

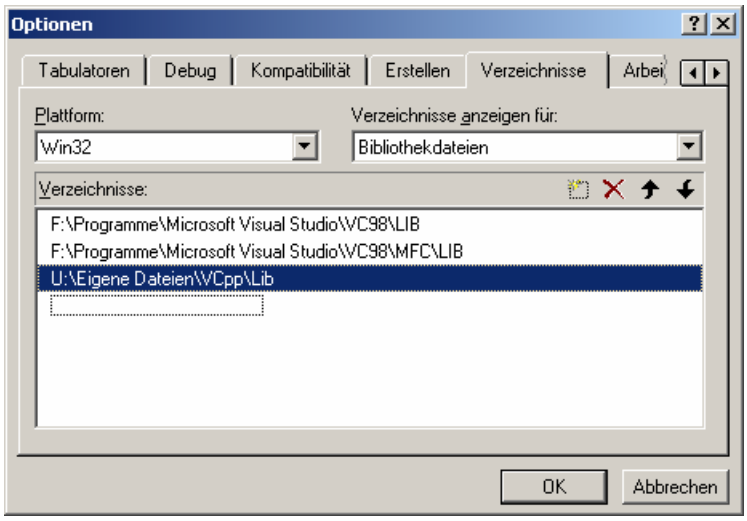
```
------------------Konfiguration: MyLib - Win32 Debug--------------------
Kompilierung läuft...
max.cpp
tausche.cpp
Bibliothek wird erstellt...

MyLib.lib - 0 Fehler, 0 Warnung(en)
```

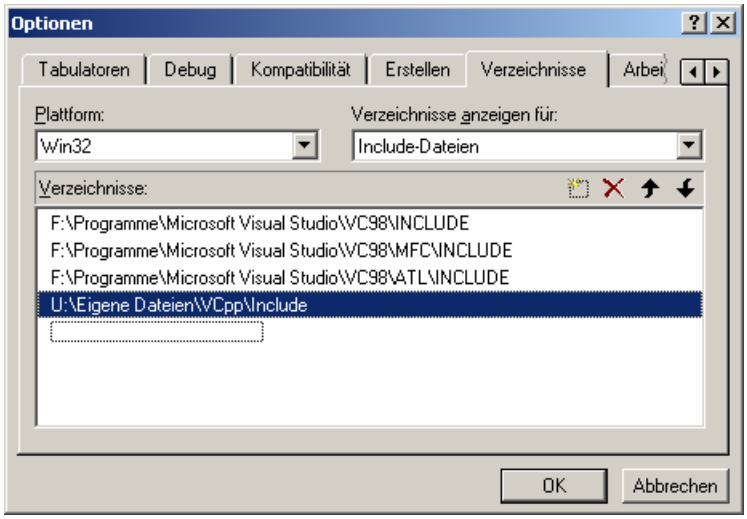
  Nun finden Sie u.a.
  - o im Projektverzeichnis die Headerdatei **MyLib.h**
  - o im Debug-Unterverzeichnis zum Projektordner die Bibliotheksdatei **MyLib.lib** sowie die Objektdateien zu den einzelnen Implementierungsdateien.

Um eigene Bibliotheken bequem verwenden zu können, sollten Sie jeweils ein Verzeichnis für die Lib- und für die Header-Dateien erstellen und der Entwicklungsumgebung über

Extras > Optionen > Verzeichnisse

bekannt geben:





Nach diesen Vorbereitungen, die für alle neuen Bibliotheken wirksam sind, müssen die oben erzeugten Dateien **MyLib.lib** und **Mylib.h** in die passenden Verzeichnisse kopiert werden.

Um die neuen Bibliotheksfunktionen in einem konkreten Projekt verwenden zu können, ist folgendes zu tun:

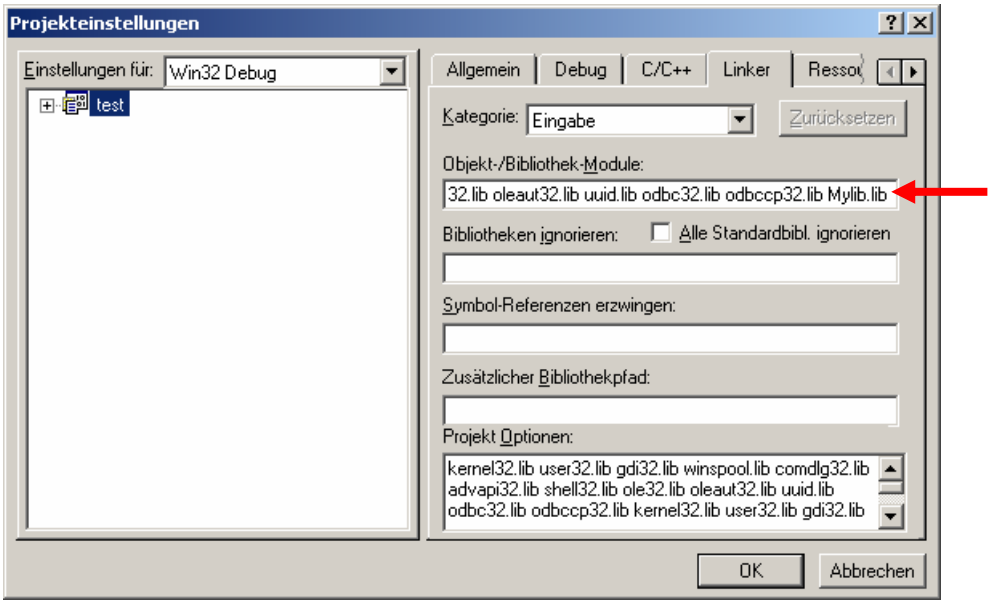
- Die Header-Datei inkludieren, z.B.:

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

void main()
{
cout << "Maximum von int-Werten:    " << max(12, 3) << endl;
}
```

  Weil sich die Datei **MyLib.h** im Suchbereich der Entwicklungsumgebung für Header-Dateien befindet, kann wie bei den Standardbibliotheks-Header-Dateien die „Spitzklammern"-Syntax verwendet werden, die ohne Pfadangabe auskommt.
- Nach dem Menübefehl

  Projekt > Einstellungen > Linker

  muss die Bibliothek in den Linker-Projekteinstellungen angemeldet werden, z.B.:



  Weil sich die Datei **MyLib.lib** im Suchbereich der Entwicklungsumgebung für Bibliotheks-Dateien befindet, ist keine Pfadangabe erforderlich. Im Linker-Aufruf, den die Entwicklungsumgebung für uns übernimmt, müssen alle nach unaufgelösten Referenzen zu durchsuchenden Bibliotheken angegeben werden. Hätte der Linker alle Dateien im Bibliotheks-Suchbereich zu berücksichtigen, würde die Suche unangemessen lange dauern.
  Projektinterne Objektmodule müssen nicht in den Linker-Projekteinstellungen aufgeführt werden (vgl. Abschnitt 4.5.2). Hier hat die DIE den nötigen Überblick und informiert den Linker entsprechend.

Weil wir keine **namespace**-Definition vorgenommen haben, befinden sich die neuen Funktionen im globalen Namensraum. Soll ein spezieller Namensraum vereinbart werden, so muss dies in den betroffenen Implementierungsdateien *und* in der Header-Datei geschehen.
Um für die Funktionen der Bibliothek `MyLib` den Namensraum `MyLib` zu vereinbaren, ändert man die Quellmodule folgendermaßen ab:

max.cpp

```
namespace MyLib
{
   int max(int arg1, int arg2)
   {
      return (arg2 > arg1) ? arg2 : arg1;
   }

   double max(double arg1, double arg2)
   {
      return (arg2 > arg1) ? arg2 : arg1;
   }
}
```

tausche.cpp

```
namespace MyLib
{
   void tausche(int &arg1, int &arg2)
   {
      int temp;
      temp = arg1;
      arg1 = arg2;
      arg2 = temp;
   };

   void tausche(double &arg1, double &arg2)
   {
      double temp;
      temp = arg1;
      arg1 = arg2;
      arg2 = temp;
   };
}
```

Über unser bisheriges Wissen zu Namensräumen hinausgehend wird hier von der Möglichkeit Gebrauch gemacht, Funktionen aus verschiedenen Quellmodulen in *einen* Namensraum einzuordnen.
Als Header-Datei kann nun verwendet werden:

```
namespace MyLib
{
   int    max(int arg1, int arg2);
   double max(double arg1, double arg2);
   void   tausche(int &arg1, int &arg2);
   void   tausche(double &arg1, double &arg2);
}
```

Beim Einsatz der modifizierten Bibliothek muss der spezielle Namensraum angegeben werden (vgl. Abschnitt 3.4.2), z.B.:

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

void main()
{
   cout << "Maximum von int-Werten:    " << MyLib::max(12, 3) << endl;
}
```

4.6 inline-*Funktionen*

Beim Aufruf einer sehr kleinen Funktion besteht der Gesamtaufwand zum großen Teil aus den in Abschnitt 4.2 beschriebenen Übergabeaktivitäten. Wird eine solche Funktion häufig benutzt, kann aus Performanzgründen eine genannte **inline**-Deklaration sinnvoll sein. Dabei wird dem Compiler empfohlen, an jeder Aufrufstelle den kompletten Maschinencode der Funktion in das zu erstellende Programm einzufügen.
Beispiel:

```
inline int max(int arg1, int arg2)
{
   return (arg2 > arg1) ? arg2 : arg1;
}
```

Bei einer längeren Funktion ist die **inline**-Expansion ihres Maschinencodes wenig sinnvoll, denn:

- Hier ist der Geschwindigkeitsvorteil relativ geringfügig.
- Der Größenzuwachs durch den mehrfach eingefügten Code ist hingegen erheblich.

Bei der **inline**-Deklaration handelt es sich nur um eine *Empfehlung* an den Compiler, an die er sich nicht halten muss. Vermutlich wird er bei seiner Entscheidung auch die Optimierungsoptionen berücksichtigen, die im Visual Studio für ein Projekt nach

Projekt > Einstellungen > C/C++

vorgenommen werden können.

In der Programmiersprache C wird oft an der Stelle der dort fehlenden **inline**-Option die Präprozessor-Anweisung **#define** eingesetzt, z.B.:

Quellcode	Ausgabe

`#include <cstdio>`
 	
`#define max(arg1,arg2) (((arg1) > (arg2)) ? (arg1) : (arg2))`
 	
`void main()`
`{`
`  printf("%d\n", max(2,3));`
`  printf("%f\n", max(2.4,3.8));`
`}`
	`3`
	`3.800000`

Wie bei der bereits früher erwähnten Vereinbarung einer Konstanten wird der Präprozessor per **#define** veranlasst, das so genannte *Funktionsmakro* `max(arg1,arg2)` durch den Ausdruck `(((arg1) > (arg2)) ? (arg1) : (arg2))` zu ersetzen. Durch den reichlichen Gebrauch von runden Klammern wird verhindert, dass es bei Makroaufrufen mit Ausdrücken als Argumenten zu unerwünschten Auswertungs-Prioritäten kommt.
Die Beliebtheit der Makros bei C-Programmierern hat neben dem Performanzgewinn noch einen zweiten Grund: Ein Makro verhält sich weitaus typ-generischer als eine Funktion (siehe Beispiel).

Allerdings sind Funktionsmakros sehr „beliebte“ Fehlerquellen, wie das folgende Beispiel demonstriert:

Quellcode	Ausgabe

`#include <cstdio>`
 	
`#define max(arg1,arg2) (((arg1) > (arg2)) ? (arg1) : (arg2))`
 	
`void main()`
`{`
`  int i = 3;`
`  printf("%d\n", max(++i,3));`
`}`
	`5`

Hier bewirkt das Argument `++i` in Abhängigkeit vom bisherigen `i`-Wert eine einfache oder doppelte Inkrementierung!

Wegen der zahlreichen Fehlermöglichkeiten sollte die Präprozessor-Anweisung **#define** in C++ grundsätzlich nicht verwendet werden. Ihre Vorteile sind auch durch sichere Konstruktionen verfügbar. So kann man z.B. generische Funktionen ohne Verlust an Typsicherheit über die so genannten Templates realisieren.

4.7 Rekursive Algorithmen

Es ist erlaubt und in vielen Situationen auch sinnvoll, dass eine Funktion *sich selbst* aufruft. Solche rekursiven Aufrufe stellen bei Problemen mit *iterativer* Struktur oft eine interessante Alternative zur (stets möglichen) Lösung durch einen iterativen Algorithmus (per Wiederholungsanweisung) dar. In folgendem Beispiel wird ein *iterativer* Algorithmus verwendet, um den größten gemeinsamen Teiler (GGT) von zwei ganzen Zahlen zu ermitteln:

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

int ggTi(int a, int b)
{
    do {
        if (a == b)
            return a;
        else
            if (a > b)
              a = a - b;
            else
              b = b - a;
    } while (true);
}

void main()
{
    cout << ggTi(24, 16) << endl;
}
```

Dabei wird ein einfaches Theorem aus der mathematischen Zahlentheorie ausgenutzt:
Für natürliche Zahlen a und b mit $a > b$ gilt:

$$x \text{ ist gemeinsamer Teiler von } a \text{ und } b$$
$$\Leftrightarrow$$
$$x \text{ ist gemeinsamer Teiler von } b \text{ und } a - b$$

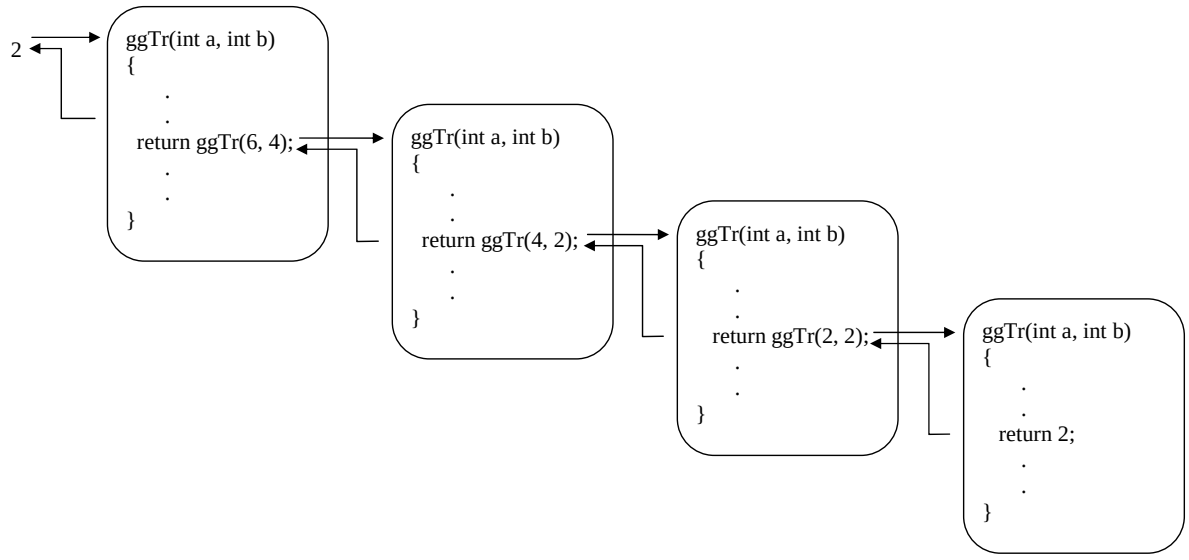
Damit ist der GGT von a und b identisch mit dem GGT von b und $a – b$.
Die iterative Funktion `ggTi()` kann durch folgende rekursive Variante `ggTr()` ersetzt werden:

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

int ggTr(int a, int b)
{
    if (a == b)
        return a;
    else
        if (a > b)
            return ggTr(b, a-b);
        else
            return ggTr(a, b-a);
}

void main()
{
    cout << ggTr(24, 16) << endl;
}
```

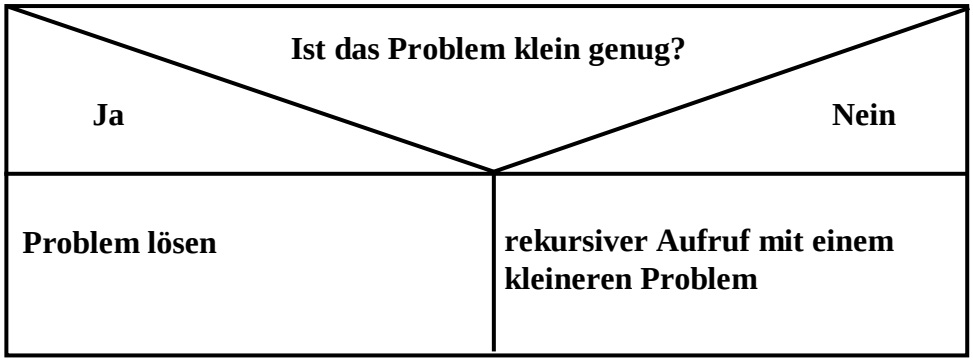
Statt eine Schleife zu benutzen, arbeitet die rekursive Funktion nach folgender Logik:

- Sind die beiden Argumente identisch, dann stimmen sie trivialerweise mit ihrem GGT überein, und der Algorithmus ist beendet.
- Anderenfalls wird das Problem, den GGT von a und b zu finden, auf ein einfacheres Problem zurückgeführt, und die Funktion `ggTr()` ruft sich selbst mit neuen Aktualparametern erneut auf. Bei der Reduktion wird das eben vorgestellte Theorem verwendet.

Wird die Funktion `ggTr()` z.B. mit den Argumenten 10 und 6 aufgerufen, kommt es zu folgender Aufrufverschachtelung:



Generell läuft ein rekursiver Algorithmus nach folgender Logik ab:



Wird in eine fehlerhaften Algorithmus der linke Zweig in diesem Struktogramm nie erreicht, überschreiten die gestapelten Funktionsaufrufe bald die Stack-Kapazität und das Programm endet mit einem Stack-Überlauf.

Rekursive Algorithmen lassen sich zwar oft eleganter formulieren als die iterativen Alternativen, benötigen aber durch die hohe Zahl von Funktionsaufrufen in der Regel mehr Rechenzeit.

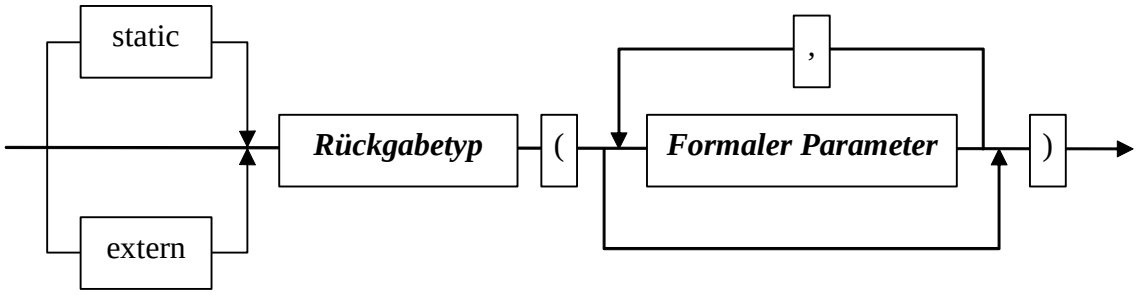
4.8 Gültigkeitsbereiche und Lebensdauer

Nachdem der Modul-Begriff zur Verfügung steht, können die bisherigen Angaben zum Gültigkeitsbereich und zur Lebensdauer von Variablen und Funktionen präzisiert und ergänzt werden.
Bislang haben zwischen lokaler (blockbezogener) und globaler Gültigkeit unterschieden und den Ort der Definition als Einteilungskriterium betrachtet. Außerdem waren die Attribute Gültigkeitsbereich und Lebensdauer gekoppelt, so dass z.B. lokale Variablen bei Blockende mit der Gültigkeit auch ihre Existenz einbüssten. Nun können wir uns ein differenzierteres Bild machen:

- Als mögliche Gültigkeitsbereichen werden *Block*, *Modul* und *Programm* unterschieden.
  Während sich am alten Begriff der *lokalen* Gültigkeit Nichts ändert, unterscheiden wir nun zwischen globaler Gültigkeit in Bezug auf das gesamte Programm oder in Bezug auf die Datei, in der sich eine Definition befindet.
- Gültigkeit und Lebensdauer werden entkoppelt.
  Wir lernen lokale Variablen kennen, die beim Verlassen ihres Blocks erhalten bleiben und bei erneutem Betreten wieder mit dem alten Wert zur Verfügung stehen.

Bei einer Funktionsdeklaration kann über den Modifikator **static** der Gültigkeitsbereich auf das aktuelle Modul eingeschränkt werden:

Funktionsschnittstelle



Mit dem Modifikator **extern** kann optional die (ohnehin voreingestellte) programmweite Gültigkeit deutlich gemacht werden.
Wird in der Header-Datei **max.h** zu dem in Abschnitt 4.5.2 vorgestellten Modul die **int**-Variante der `max()`-Funktion als **static** deklariert

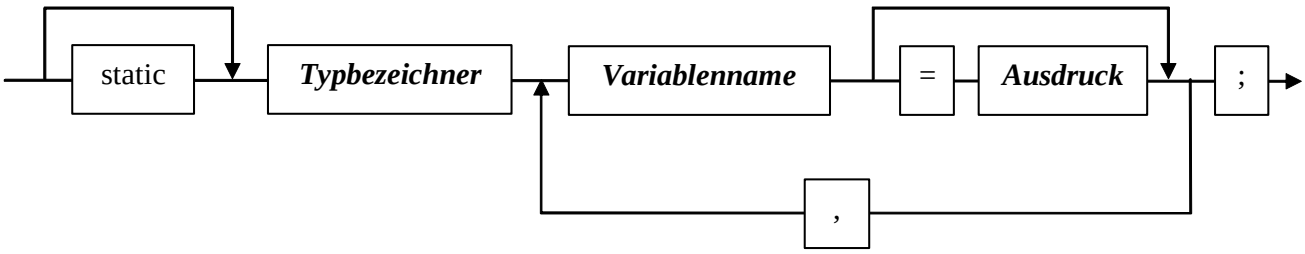
```
static int max(int arg1, int arg2);
extern double max(double, double);
```

dann kann sie in der (modulfremden) Funktion **main()** nicht mehr aufgerufen werden, wie die folgende Compiler-Fehlermeldung zeigt:

```
Statische Funktion "int __cdecl max(int,int)" wurde deklariert, aber nicht definiert
```

Während das Schlüsselwort **static** in einer Funktionsdeklaration den Gültigkeitsbereich reduziert, erweitert es in einer lokalen Variablendeklaration die Lebensdauer von Zeitraum der Blockaktivität auf die gesamte Laufzeit des Programms.

Variablendefinition



Eine betroffene Variable behält ihre lokale Gültigkeit, wird jedoch in die so genannte **statische Speicherklasse** befördert. Neben der höheren Lebenserwartung ist damit auch die Initialisierung bei Programmstart verbunden (auf den Wert 0). In folgendem Beispielprogramm meldet die Funktion `growth()` genau dann den Wert **true** zurück, wenn das aktuelle Argument das beim letzten Aufruf übergebene übertrifft. Für die erforderliche Gedächtnisleistung wird die statische Variable `last` benutzt:

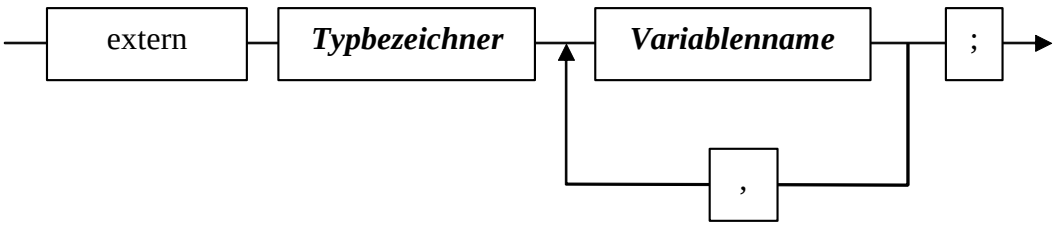
Quellcode	Ausgabe

`#include <iostream>`
`#include <cmath>`
`using namespace std;`
 	
`int growth(int act)`
`{`
`  static int last;`
`  int buffer = last;`
`  last = act;`
`  if (act > buffer)`
`  return true;`
`  else`
`  return false;`
`}`
 	
`void main()`
`{`
`  growth(0);`
`  cout << growth(5) << endl;`
`  cout << growth(4) << endl;`
`  cout << growth(5) << endl;`
`}`
	`1`
	`0`
	`1`

Bei einer Variablendeklaration außerhalb aller Blöcke ist der Modifikator **static** nicht etwa überflüssig und wirkungslos, sondern bewirkt eine Einschränkung auf den Gültigkeitsbereich Datei! Eine im Modul als **static** deklarierte Variable kann mit einer namensgleichen, programmweit gültigen Variablen koexistieren und überdeckt diese.

Mit dem Modifikator **extern** informiert man den Compiler bei einer blockfreien Variablendeklaration darüber, dass eine programmweit gültige Variable verwendet und *keine* neue angelegt werden soll. Weil die Quellmodule separat übersetzt werden, muss eine programmweit gültige Variable „angemeldet“ werden, bevor sie in einem Quellmodul benutzt werden kann.

Deklaration externer Variablen



Hier haben wir es erstmals mit einer Variablen*deklaration* zu tun, die *keine Definition* ist. Weil keine neue Variable ins Spiel kommt, muss ja auch kein Speicherplatz reserviert werden.
In folgendem „Hauptmodul“ wird die Variable `mex` blockfrei definiert, so dass sich ihr Gültigkeitsbereich sich über das gesamte Programm erstreckt:

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

int mex;

void main()
{
  cout << "Maximum von int-Werten: " << max(12, 3) << endl << endl;
  cout << "mex                   : " << mex << endl;
}
```

Damit die Variable `mex` in einem anderen Quellmodul verwendet werden kann, muss sie dort mit dem Modifikator **extern** deklariert werden:

```
extern int mex;

int max(int arg1, int arg2)
{
  mex = arg2;
  return (arg2 > arg1) ? arg2 : arg1;
}
```

Nun kann die Variable `mex` von der modul-fremden Funktion `max()` geändert werden:

```
Maximum von int-Werten: 12

mex                   : 3
```

Der folgenden Tabelle ist zu entnehmen, wie die für Variablen möglichen Gültigkeitsbereich × Lebensdauer – Kombinationen definiert werden müssen:

		Lebensdauer	
		Block	**Programm**
Gültigkeitsbereich	**Block**	blockintern, ohne Modifikator	blockintern, Modifikator **static**
	Modul		blockfrei, Modifikator **static**
	Programm		blockfrei, ohne Modifikator

Anmerkungen:

- Variablen mit Lebensdauer *Programm* werden automatisch mit dem Wert 0 initialisiert.
- Um eine programm-global gültige Variable aus einem fremden Modul verwenden zu können, muss sie im aktuellen Modul mit dem Modifikator **extern** deklariert werden.

4.9 Übungsaufgaben zu Abschnitt 4

1) Ändern Sie bitte das Beispielprogramm in Abschnitt 4.4 so ab, dass Sie im Ausführungsteil der beiden Funktionen jeweils nur eine einzige Anweisung benötigen.

2) Erweitern Sie die in Abschnitt 4.5.2 vorgestellte Bibliothek **MyLib.lib** um Funktionen zur Berechnung des Minimums von zwei **int**- bzw. **double**-Parametern.
Tipp: Um häufiges Umkopieren beim Entwickeln der Bibliothek zu vermeiden, können Sie nach

Projekt > Einstellungen > Bibliothek

den gewünschten Zielort für die **lib**-Datei eintragen (z.B. U:\Eigene Dateien\VCpp\Lib\MyLib.lib). Analog dazu lässt sich die Header-Datei des Bibliotheks-Projektes auch direkt im generell verwendeten Include-Verzeichnis anlegen (z.B. U:\Eigene Dateien\VCpp\Include).

3) Erstellen Sie ein Programm zur Fakultätsberechnung mit einem rekursiven Algorithmus.

5 Abgeleitete Datentypen

Nachdem wir uns bisher auf einfache (skalare) Variablen beschränkt haben, lernen wir nun *zusammengesetzte Datentypen* kennen, zunächst mit homogenen, später auch mit unterschiedlichen Elementen. Dabei beschränken wir uns auf das traditionelle Repertoire an zusammengesetzten Datentypen, die schon in der Programmiersprache C vorhanden sind. Wenngleich später modernere, klassenbasierte Varianten für viele Anwendungsfälle vorgestellt werden, dürfen die Klassiker nicht ignoriert werden. Sie sind in älteren Programmen allgegenwärtig und auch für viele neue Aufgaben durchaus noch angemessen.
Ein wichtiges Thema von Abschnitt 5 bilden auch die so genannten *Zeiger- bzw. Pointervariablen*, deren Inhalte keine Daten im bisherigen Sinne sind, sondern *Speicheradressen* von anderen Variablen.

5.1 Felder (Arrays)

Ein Feld bzw. Array enthält eine feste Anzahl von Elementen des selben Datentyps, die im Speicher hintereinander abgelegt werden. Sie können gemeinsam verarbeitet (z.B. als Aktualparameter), aber auch einzeln über einen Index angesprochen werden.
Hier ist als Beispiel ein eindimensionaler Array namens `uni` mit 5 Elementen vom Typ **int** zu sehen:

uni[0]	uni[1]	uni[2]	uni[3]	uni[4]
2079	1990	1989	1978	1964

Beim Zugriff auf ein einzelnes Element gibt man nach dem Arraynamen den durch eckige Klammern begrenzten Index an, wobei die Nummerierung mit der 0 beginnt. Weil der Index auch über eine *Variable* geliefert werden kann, bietet ein Array z.B. im Zusammenhang mit einer Wiederholungsanweisung erhebliche Vorteile gegenüber einer Anzahl von selbständigen Variablen.
Wir befassen und zunächst mit eindimensionalen Arrays, behandeln später aber auch den mehrdimensionalen Fall.

5.1.1 Beispiel: Beurteilung des Pseudozufallszahlengenerators

Die Elemente des eben wiedergegebenen Arrays `uni` scheinen 5 Jahreszahlen mit einem Ausrutscher in die Zukunft zu enthalten. Tatsächlich handelt sich aber um Ergebnisse des folgenden Programms `unirand` zur Prüfung des Visual C++ - Pseudozufallszahlengenerators **rand()**, den wir schon zur Lottozahlen-Produktion benutzt haben:

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

const int interv = 5;
const int drl = 10000;
const int srs = 9999;

void main()
{
  int i, teiler, uni[interv];

  srand(srs);
  for (i = 0; i < interv; i++)
    uni[i] = 0;

  teiler = 32768/interv;
```

```
  for (i = 0; i <= drl; i++)
    uni[rand()/teiler]++;

  for (i = 0; i < interv; i++)
    cout << double(uni[i])/drl  << " ";
  cout << endl;
}
```

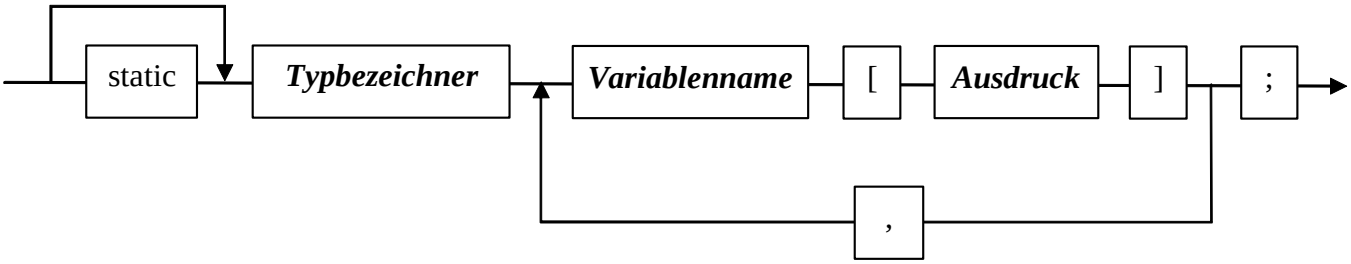
Das Programm prüft die univariate Verteilung der **rand()**-Funktionswerte, indem es den Generator initialisiert, 10000 Pseudozufallswerte ermittelt und in 5 gleich breite Intervalle einordnet. Für die Ausgabe werden dann noch die relativen Häufigkeiten der Intervalle ermittelt. Mit dem Pseudozufalls-Startwert 9999 erhalten wir folgendes Ergebnis, das als zufrieden stellende Approximation der zu erwartenden Gleichverteilung gelten kann:

```
0.2079 0.199 0.1989 0.1978 0.1964
```

5.1.2 Arrays definieren

Die Definition einer Feldvariablen erhält man aus der Definition einer Variablen des Basistyps, indem man hinter den Variablennamen in eckige Klammern eingeschlossen die Anzahl der Elemente setzt:

Arraydefinition



Die Anzahl der Elemente muss über einen **ganzzahligen**, **konstanten** Ausdruck angegeben werden, dessen Wert bereits zur Übersetzungszeit feststeht.

Beispiel:

```
int uni[5];
```

Im Vergleich zu 5 einzelnen **int**-Variablen haben wir im Beispiel einen deutlich verringerten Definitionsaufwand.

Das obige Beispielprogramm ist so entworfen, dass die Anzahl der betrachteten Intervalle (und damit der `uni`-Elemente) leicht geändert werden kann. Es muss jedoch anschließend neu kompiliert werden, unterstützt also z.B. keine dynamische Festlegung der Array-Größe durch den Benutzer.

Bei einer reinen Deklaration muss die Anzahl der Elemente *nicht* angegeben werden, z.B.:

```
extern int mex[];
```

Auch eine Array-Variable kann bei der Definition gleich initialisiert werden, z.B.:

```
int uni[5] = {0,0,0,0,0};
```

5.1.3 Arrays benutzen

In obigem Programm wird der Array `uni` *nicht* im Kontext der Definition initialisiert, sondern in einer speziellen **for**-Schleife, wobei die Feldelemente einzeln über ihren Index angesprochen werden:

```
for (i = 0; i < interv; i++)
  uni[i] = 0;
```

Bei dieser Lösung muss nach einer Änderung der Feldgröße (über die Konstante `interv`) der Initialisierungs-Quellcode *nicht* geändert werden.

Wichtig: Die Feldelemente werden in C/C++ mit **0** beginnend nummeriert. Von *n* Feldelementen hat das Erste also die Nummer 0 und das Letzte die Nummer *n*-1.

Ein per Index angesprochenes Feldelement kann verwendet werden wie eine Variable des Basistyps, z.B.:

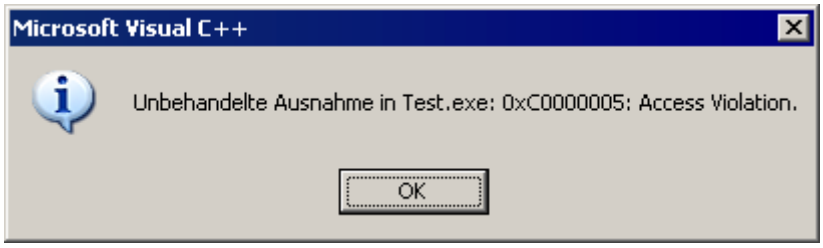
```
cout << double(uni[i])/drl  << " ";
```

Der Indexwert kann dabei durch einen beliebigen ganzzahligen Ausdruck geliefert werden.

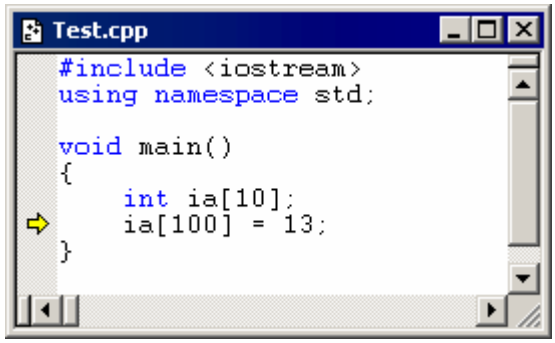
Ein verbreiteter Programmierfehler im Zusammenhang mit Feldern ist der Zugriff auf nicht definierte Elemente, z.B.:

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

void main()
{
   int ia[10];
   ia[10] = 13;
}
```

Um solche Fehler kümmert sich ein C/C++ - Compiler auch dann **nicht**, wenn sie (wie im Beispiel) schon auf Quelltextebene erkennbar sind. Zur Laufzeit tritt eine unbehandelte Ausnahme auf, die zum Abbruch des Programms führt.
Mit derartigen Problemen konfrontiert, sollten Sie ein Programm per Funktionstaste **F5** oder Symbolleistenschalter  im **Debug-Modus** ausführen lassen, mit dem wir uns später noch ausführlich beschäftigen werden. In obigem Beispiel erhält man dann folgende Fehlermeldung:



Außerdem wird die fehlerhafte Anweisung im Quellcode markiert:



5.1.4 Mehrdimensionale Felder

Speziell in der Matrix-Algebra, aber auch auf vielen anderen Anwendungsbereichen, werden mehrdimensionale Felder benötigt. Um den Einsatz eines zweidimensionalen Feldes zu üben, führen wir unsere Analyse des Visual C++ - Pseudozufallszahlengenerators fort und gehen dabei der Beobachtung nach, dass der *erste* **rand()**-Funktionswert eine erstaunliche Abhängigkeit vom Startwert des Generators zeigt.

Srand() - Startwert	Erster rand() - Wert
100	365
200	691
300	1018
1000	3304
2000	6569
3000	9835

Das anschließend vorgestellte Programm `inirand` ermittelt für Generator-Startwerte von 0 bis 32767 jeweils den ersten **rand()**-Funktionswert. Für die Start- und Funktionswerte werden jeweils 5 gleich breite Intervalle gebildet, so dass jedes Paar aus Start- und Funktionswert einer Zelle der folgenden (5×5)-**int**-Matrix **stafir** zugeordnet werden kann:

	0	1	2	3	4
0	`stafir[0][0]`	`stafir[0][1]`	`stafir[0][2]`	`stafir[0][3]`	`stafir[0][4]`
1	`stafir[1][0]`	`stafir[1][1]`	`stafir[1][2]`	`stafir[1][3]`	`stafir[1][4]`
2	`stafir[2][0]`	`stafir[2][1]`	`stafir[2][2]`	`stafir[2][3]`	`stafir[2][4]`
3	`stafir[3][0]`	`stafir[3][1]`	`stafir[3][2]`	`stafir[3][3]`	`stafir[3][4]`
4	`stafir[4][0]`	`stafir[4][1]`	`stafir[4][2]`	`stafir[4][3]`	`stafir[4][4]`

In den Matrix-Elementen wird während des Experimentes die Anzahl der Treffer gezählt. Anschließend gibt `inirand` die beobachteten relativen Häufigkeiten aus, die wir mit der theoretischen Erwartung einer für alle Zellen identischen Proportion von 0,04 (= 0,2 · 0,2) vergleichen können.

Nach der aufwändigen Erläuterung des Anwendungsbeispiels, das diesmal von eigenständigem Interesse ist, sollen syntaktische Aspekte nicht ganz unter gehen: Bei der Definition eines mehrdimensionalen Feldes (und auch beim Zugriff auf Feldelemente) werden die Indizes *einzeln* eingeklammert, z.B.:

Quellcode

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

void main()
{
  int mat[2][2] = {{1,2}, {3,4}};
  for (int i= 0; i <= 1; i++)
  {
    for (int j= 0; j <= 1; j++)
      cout << mat[i][j] << "\t";
    cout << endl;
  }
}
```

Ausgabe

```
1       2
3       4
```

Die Syntax zur Definition und zur Initialisierung lässt erkennen, dass in C/C++ die zweidimensionale Matrix als *Array of Arrays* aufgefasst wird.

Bei einer Definition mit Initialisierung kann die innerste Größenangabe weggelassen werden, z.B.:

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

Dies ist auch bei extern deklarierten Feldern und bei Funktionsparametern erlaubt (siehe unten).

Für Felder höherer Dimensionalität lässt sich das im zweidimensionalen Fall vorgeführte Mehrfachindizierungsschema beliebig verallgemeinern.

Für die Anordnung der Array-Elemente im Speicher gilt das Prinzip: Der letzte Index läuft am schnellsten. Folglich sind z.B. die Elemente einer (2×2) – Matrix `mat` folgendermaßen angeordnet:

mat[0][0]	mat[0][1]	mat[1][0]	mat[1][1]

Nun endlich das Programm `inirand` zur Analyse des Pseudozufallszahlengenerators:

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

const int interv = 5;

void main()
{
  int drl;

  int i, j, teiler, stafir[interv][interv];

  for (i = 0; i < interv; i++)
    for (j = 0; j < interv; j++)
      stafir[i][j] = 0;

  teiler = 32768/interv;
  drl = teiler * interv;
  for (i = 0; i < drl; i++)
  {
    srand(i);
    stafir[i/teiler][rand()/teiler]++;
  }

  cout.setf(ios::fixed);
  cout.precision(2);
  for (i = 0; i < interv; i++)
  {
    for (j = 0; j < interv; j++)
      cout << double(stafir[i][j])/drl << "\t";
    cout << endl;
  }
}
```

Wie befürchtet, erhalten wir für den ersten Wert des Pseudozufallszahlengenerators **rand()** erhebliche Abweichungen von der theoretischen Gleichverteilung:

```
0.06    0.06    0.06    0.02    0.00
0.06    0.03    0.00    0.04    0.06
0.00    0.03    0.06    0.06    0.05
0.06    0.06    0.06    0.00    0.01
0.06    0.02    0.00    0.06    0.06
```

Mit einem Extraaufruf der Funktion **rand()** helfen wir dem Pseudozufall auf die Sprünge:

```
for (i = 0; i < drl; i++)
{
  srand(i); rand()
  stafir[i/teiler][rand()/teiler]++;
}
```

Nun resultiert ein mustergültiges Ergebnis:

```
0.04    0.04    0.04    0.04    0.04
0.04    0.04    0.04    0.04    0.04
0.04    0.04    0.04    0.04    0.04
0.04    0.04    0.04    0.04    0.04
0.04    0.04    0.04    0.04    0.04
```

Im Programm `inirand` werden zwei bislang nicht behandelt Formatierungs-Möglichkeiten benutzt. In der Anweisung

```
cout.setf(ios::fixed);
```

wird mit der Elementfunktion **setf()** zum Objekt **cout** dafür gesorgt, dass **double**-Werte im *Festpunkformat* ausgegeben werden. Von dem kleinen Vorgriff auf die objektorientierte Programmierung sollten Sie sich nicht verwirren lassen. Die Begriffe Elementfunktion und Objekt werden noch ausführlich behandelt.
Mit

```
cout.precision(2);
```

legen wir fest, dass bei Festpunktzahlen zwei Dezimalstellen ausgegeben werden sollen.

5.1.5 Felder als Parameter von Funktionen

Bisher haben Sie Arrays kennen gelernt, deren Größe bereits zur Übersetzungszeit endgültig festzulegen ist. Deren mangelnde Flexibilität stört bei der Entwicklung von Programmen, die eine Aufgabe für Probleme beliebiger Größe erfüllen sollen. Beim Entwerfen von *Funktionen*, die Felder als Parameter übernehmen, ist der Zwang zur Wahl fixierter Feldgrößen besonders hinderlich. Dies bleibt uns zumindest bei *ein*dimensionalen Feldern erspart, wie sich aus folgenden Regeln für Felder als Funktionsparameter ergibt:

- Felder werden in C++ grundsätzlich als **Referenzparameter** übergeben.
  Bei der Deklaration eines Feldparameters in einer Funktionsschnittstelle ist das Adressoperator-Zeichen &, mit dem wir Basistyp-Variablen als Referenzparameter kennzeichnen können, ebenso überflüssig wie verboten.
  Ist in einer Funktion für einen Array-Parameter keine Modifikation vorgesehen, sollte dieser Parameter in der Funktionsschnittstelle als **const** gekennzeichnet werden (vgl. Abschnitte 4.1.2 und 5.2.1.7).
- In der Funktionsdefinition wird *für die erste* Dimension keine Feldgrenze angegeben.
  Bei allen weiteren Dimensionen sind leider Feldgrenzen erforderlich, so dass z.B. für dreidimensionale Felder folgende Syntax für die Deklaration innerhalb einer Funktionsschnittstelle gilt:

  Basistyp-Bezeichner[] [***Größe_Dim_2***] [***Größe_Dim_3***]
- Im Funktions*aufruf* folgen auf den Feldnamen *keine* eckige Klammern.
- Größe der *ersten* Dimension als Parameter übergeben
  Damit sich eine aufgerufene Funktion an die aktuelle Grenze der ersten Dimension halten kann, muss diese als zusätzlicher Parameter übergeben werden. Die restlichen Dimensionsgrößen sind fixiert, und der Compiler lässt nur passende Aktualparameter-Arrays zu.

Im folgenden Beispiel wird eine Funktionen zur Ausgabe von zweidimensionalen **int**-Feldern definiert und verwendet:

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

const int NC = 2;

void prima(int mat[][NC], int nrow);

void main()
{
   int mat[][2] = {{1,2}, {3,4}};
   prima(mat, 2);
}

void prima(int mat[][NC], int nrow)
{
   for (int i = 0; i < nrow; i++)
   {
      for (int j = 0; j < NC; j++)
         cout << mat[i][j] << "\t";
      cout << endl;
   }
}
```

Bei der Variablendefinition in der Hauptfunktion wird ausgenutzt, dass der Compiler die Größe der ersten Dimension aus einer vorhandenen Initialisierung ermitteln kann.

Wenn Ihnen die bei *mehrdimensionalen* Feldern gebotene partielle Feldgrenzen-Flexibilität nicht genügt, können Sie z.B. Folgendes tun:

5.1.6 Mehrdimensionale Felder in eindimensionalen Arrays verwalten

Jede mehrdimensionale Matrix lässt sich auch mit einem eindimensionalen Array verwalten, wobei die Interpretation der Elemente vom Programmierer organisiert werden muss. Mit diesem Array-Konzept kann man bei Funktionen aus der partiellen Feldgrenzen-Flexibilität eine vollständige machen. Die folgende Variante einer Matrix-Ausgabefunktion kommt ohne fixe Feldgrößen aus:

Quellcode

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

void prima(int mat[], int nrow, int ncol);

void main()
{
   int mat[] = {1,2,3,4,5,6,7,8,9};
   prima(mat, 3, 3);
}

void prima(int mat[], int nrow, int ncol)
{
   for (int i = 0; i < nrow; i++)
   {
      for (int j = 0; j < ncol; j++)
         cout << mat[i*ncol + j] << "\t";
   cout << endl;
   }
}
```

Ausgabe

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

Naheliegenderweise wird die Anordnungslogik eines zweidimensionalen C++ - Arrays übernommen.
Bevor Sie eine größere Funktionsbibliothek auf der Basis dieses Vorschlags erstellen, sollten Sie noch abwarten, welche Array-Alternativen in der Standard Template Library (STL, Teil der C++ - Standardbibliothek) und in den Microsoft Foundation Classes (MFC) angeboten werden.

5.1.7 Zeichenketten auf Array-Basis (C-Strings)

In diesem Abschnitt werden elementare Kenntnisse über die traditionelle Zeichenketten-Behandlung in der Programmiersprache C vermittelt. Später lernen wir String-*Objekte* in der C++-Standardbibliothek bzw. in der MFC-Klassenbibliothek (Microsoft Foundation Classes) kennen, die grundsätzlich den traditionellen C-Strings vorzuziehen sind. Man kommt aber in vielen Situationen um den Einsatz von C-Strings nicht herum.

5.1.7.1 Aufbau und Definition von C-Strings

Bei einem C-String handelt es sich um ein eindimensionales Feld mit dem Basistyp **char**, z.B.:

```
char gruss[10] = "Hallo!";
```

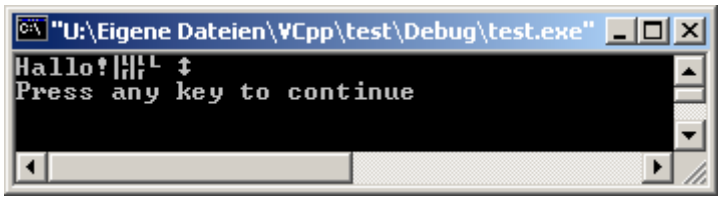
Die maximale Länge eines C-Strings ist dabei nur durch den verfügbaren Speicher begrenzt.
In der Regel werden nicht alle deklarierten C-String-Elemente mit Zeichen belegt, also in definiertem Zustand sein. Um das Ende der definierten Zeichenfolge von den dahinter liegenden Zufallsdaten zu trennen, wird ein Zeichen mit dem hexadezimalen Wert 0x00 verwendet; man spricht daher auch von einem **null-terminierten String**.
Aufgrund der obigen Definition und Initialisierung hat die C-String-Variable `gruss` die folgende Struktur:

gruss[0]	gruss[1]	gruss[2]	gruss[3]	gruss[4]	gruss[5]	gruss[6]	gruss[7]	gruss[8]	gruss[9]
H	a	l	l	o	!	\0			

Als **char**-Array hätte `gruss` aufgrund der Syntax-Regeln für Array-Initialisierungen (siehe Abschnitt 5.1.1) und für **char**-Literale sowie unter Berücksichtigung der Null-Terminierungs-Regel eigentlich recht umständlich initialisiert werden müssen:

```
char gruss[10] = {'H', 'a', 'l', 'l', 'o', '!','\0'};
```

Diese Technik bietet immerhin die Möglichkeit, versuchsweise das terminierende Null-Zeichen wegzulassen, was sich bei einer Ausgabe folgendermaßen auswirkt:



In der Praxis verwendet man zum Initialisieren eines **char**-Arrays meist ein **Zeichenketten-Literal**, wobei der Compiler das terminierende Nullzeichen automatisch anhängt. Man muss das Nullzeichen allerdings in Rechnung stellen, was in folgendem Beispiel versäumt worden ist:

```
char gruss[6] = "Hallo!";
```

Folglich beschwert sich der Compiler:

```
error C2117: 'Hallo!' : Feldgrenze ueberschritten
```

Wie generell bei Feldern erlaubt, kann man es dem Compiler auch bei Strings zumuten, die Anzahl der Elemente aus dem Initialisierungswert zu ermitteln, z.B.:

```
char gruss[] = "Hallo!";
```

Durch diese Deklaration entsteht eine C-String-Variable mit 7 Elementen, wie man mit dem **sizeof**-Operator leicht nachprüfen kann. Die folgende Anweisung:

```
cout << gruss << " " << sizeof(gruss) << endl;
```

produziert die Ausgabe:

```
Hallo! 7
```

Mit dem **sizeof**-Operator, den Sie eben kennen gelernt haben, lässt sich für einen beliebigen Ausdruck und auch für einen beliebigen Datentyp der Speicherbedarf in Bytes feststellen, z.B.:

Quellcode	Ausgabe

`#include <iostream>`
`using namespace std;`
 	
`void main()`
`{`
`  int nVal = 3;`
`  cout << "Breite in Bytes:" << endl;`
`  cout << " int-Variable:  "`
`  << sizeof(int) << endl;`
`  cout << " double-wert. Ausdruck:  "`
`  << sizeof(nVal*4.0) << endl;`
`  int ia[10];`
`  cout << " int-Array mit 10 El.:  "`
`  << sizeof ia << endl;`
`}`
	`Breite in Bytes:`
	`int-Variable:  4`
	`double-wert. Ausdruck:  8`
	`int-Array mit 10 El.:  40`

Das Argument muss nur dann eingeklammert werden, wenn es sich um einen Typbezeichner handelt.

5.1.7.2 Wertzuweisungen bei C-Strings

Leider lässt sich die bequeme Initialisierungs-Syntax nicht auf Wertzuweisungen für C-String-Variablen übertragen. Hier gilt nämlich wie bei allen Feldern, dass nur einzelnen Elementen Werte zugewiesen werden können, z.B.:

```
gruss[0] = 'H';
gruss[1] = 'a';
gruss[2] = 'l';
gruss[3] = 'l';
gruss[4] = 'o';
gruss[5] = '!';
gruss[6] = '\0';
```

In der Standardbibliothek sind einige Funktionen definiert, mit denen sich solche unpraktischen Konstruktionen vermeiden lassen, z.B.:

Quellcode	Ausgabe

`#include <iostream>`
`using namespace std;`
 	
`void main()`
`{`
`  char gruss[20];`
`  strcpy(gruss, "Hallo");`
`  cout << gruss << endl;`
 	
`  strcat(gruss, " Kurt!");`
`  cout << gruss << endl;`
`}`
	`Hallo`
	`Hallo Kurt!`

Mit der Funktion **strcpy()** wird der Stringvariablen `gruss` der Wert `"Hallo"` (samt Terminierungs-Nullzeichen) zugewiesen. Mit der Funktion **strcat()** wird die Stringvariablen `gruss` um die Zeichenfolge `" Kurt!"` verlängert, wobei das terminierende Nullzeichen geeignet verschoben wird.
Achtung: **strcpy()** und **strcat()** kontrollieren **nicht**, ob eine Feldgrenzenüberschreitung auftritt!

Beide Funktionen setzen die Header-Datei **ctring** (**string.h**) voraus. Im Beispielprogramm ist die entsprechende **#include**-Anweisung trotzdem überflüssig, weil **cstring** über die Header-Datei **iostream** (zur *neuen* C++ - Standardbibliothek) indirekt eingeschleust wird.

In der **strcpy()**-Funktionsschnittstellen-Beschreibung aus der Visual C++ - Online-Hilfe (MSDN) begegnen wir erstmals der Syntax für **Zeigertypen**, mit der wir uns im Abschnitt 5.2 ausführlich beschäftigen werden:

```
char *strcpy(char *strDestination, const char *strSource);
```

Vorläufig dürfen Sie sich darauf verlassen, dass *in einer Funktionsschnittstelle* die Zeiger-Schreibweise:

char ***strDestination***

für unsere Zwecke äquivalent ist zu der Array-Schreibweise:

char ***strDestination***[]

Dass in Funktionsprototypen für eindimensionale Felder (mithin auch für C-Strings) keine Feldgröße angegeben werden, ist uns seit Abschnitt 5.1.5 bekannt. Dass Funktionen zur Verarbeitung von C-Strings keine Extraparameter für die Aktualparametergrößen benötigen, um illegale Grenzübertritte zu vermeiden, ist den terminierenden Nullzeichen zu verdanken.

Beim **strcpy()**-Aufruf dürfen als ***strSource*** sowohl String-Variablen als auch String-Literale auftreten. Gemäß obiger Äquivalenz-Zusage handelt es sich bei ***strDestination*** und ***strSource*** um Array-Parameter, also nach einer generellen Regel aus Abschnitt 5.1.5 um Referenzparameter. Durch das Schlüsselwort **const** wird ***strSource*** zum konstanten Referenzparameter (vgl. Abschnitt 4.1.2). Damit verspricht der **strcpy()**-Programmierer, den Parameter nicht zu verändern. Das dürfen wir wohl glauben, wenngleich es ohne weiteres möglich ist, in der Funktions-Implementation diesen Schutz wieder aufzuheben (siehe Abschnitt 5.2.1.7).

Bei der Funktion **strcat** gilt bis auf den Funktionsnamen die selbe Syntax:

```
char *strcat(char *strDestination, const char *strSource);
```

In diesem Zusammenhang sollte vielleicht daran erinnert werden, dass für C-Strings wie für beliebige Felder gilt:

- In Funktionsschnittstellen treten sie stets als *Referenz*parameter auf.
- Bei eindimensionalen Feldern in Funktionsschnittstellen wird die Anzahl der Elemente nicht angegeben.
- Beim Funktionsaufruf wird der Feldname *ohne* eckige Klammern geschrieben.
  In Abschnitt 5.2.1 wird erläutert, dass der Name einer Feldvariablen als Pointer auf das erste Element funktioniert. Damit genügt der Name einer Stringvariablen (also eines **char**-Feldes) den Anforderungen in obigen Funktionsschnittsstellen für **strcpy** bzw. **strcat**: Es ist ein Pointer auf eine **char**-Variable.

5.1.7.3 Konsoleneingabe in Stringvariablen

Um über das Objekt **cin** Konsolen-Eingaben des Benutzers in eine C-String-Variable zu leiten, können wir die **cin**-Methode **getline()** verwenden. Hier bietet sich eine gute Gelegenheit, weitere Erfahrungen beim Umgang mit Objekten zu sammeln.
Mit dem Funktionsaufruf **cin.getline()** schicken wir den Auftrag **getline()** an das Objekt **cin**, wobei die näheren Auftragsparameter in Klammern mitgeliefert werden (wie bei einem „normalen" Funktionsaufruf). Bis auf den vorangestellten Objektnamen unterscheidet sich eine Methodenschnittstelle nicht von einer normalen Funktionsschnittstelle:

cin.getline (char ***strDestination***, int ***nCount***, char ***delimiter*** = '\n');

Die Eingabe soll in der C-String-Variablen ***strDestination*** landen, wobei die Anzahl der zu übertragenden Zeichen durch zwei Bedingungen beschränkt wird:

- Maximal werden ***nCount*** – 1 Zeichen übertragen (Null-Terminierung!).
- Die Übertragung endet vor dem Trennzeichen, dessen Voreinstellung '\n' (Newline) mit dem optionalen Parameter ***delimiter*** geändert werden kann.

Im Beispiel ist der optionale Parameter ***delimiter*** weggelassen, also die Voreinstellung in Kraft:

Quellcode	Ein- und Ausgabe

`#include <iostream>`
`using namespace std;`
 	
`void main()`
`{`
`  char vorname[20];`
`  cout << "Ihr Vorname: ";`
`  **cin.getline(vorname, 20);**`
`  cout << "Hallo, " << vorname << "!" << endl;`
`}`
	`Ihr Vorname: Otto`
	`Hallo, Otto!`

Statt der **getline()**-Methode kann natürlich auch der Extraktionsoperators verwendet werden:

```
cin >> vorname;
```

Dabei hat man jedoch mit zwei Problemen zu kämpfen:

- Das Extrahieren von Zeichen aus der Standardeingabe stoppt mit dem ersten Leerzeichen, was z.B. zu folgendem Ergebnis führt:

  ```
  Ihr Vorname: Otto Ludwig
  Hallo, Otto!
  ```

- Die Anzahl der übergebenen Zeichen kann nicht begrenzt werden, so dass eine überlange Eingabe einen Programmabsturz wegen Feldgrenzenüberschreitung bewirkt.

Aber auch mit **getline()** gibt es ein kleines Problem, wenn die Methode überzählige Eingabezeichen zensieren muss und anschließend erneut aufgerufen wird, z.B.:

Quellcode	Ausgabe

`#include <iostream>`
`using namespace std;`
 	
`void main()`
`{`
`  char vorname[20], name[20];`
`  cout << "Ihr Vorname:  ";`
`  cin.getline (vorname,20);`
`  cout << "Ihr Nachname: ";`
`  cin.getline (name, 20);`
`  cout << "Hallo, " << vorname`
`  << " " << name << "!" << endl;`
`}`
	`Ihr Vorname:  Oskar-Emanuel-Theodor`
	`Ihr Nachname: Hallo, Oskar-Emanuel-Theod !`

Die beim ersten Aufruf im Eingabestrom „stehen gebliebenen“ Zeichen werden beim zweiten Aufruf samt '\n', das als voreingestelltes Trennzeichen wirkt, verarbeitet und rauben dem Benutzer jede Chance, seinen Nachnamen einzutippen.
Dieses Fehlverhalten kann z.B. mit folgender **cin**-Reset-Funktion `recin()` korrigiert werden:

Quellcode	Ausgabe

`#include <iostream>`
`using namespace std;`
 	
`void recin()`
`{`
`  if (cin.rdstate() != 0)`
`  {`
`  cin.clear(0);`
`  cin.ignore(INT_MAX, '\n');`
`  }`
`}`
 	
`void main()`
`{`
`  char vorname[20], name[20];`
`  cout << "Ihr Vorname:  ";`
`  cin.getline (vorname,20); recin();`
`  cout << "Ihr Nachname: ";`
`  cin.getline (name, 20); recin();`
`  cout << "Hallo, " << vorname`
`  << " " << name << "!" << endl;`
`}`
	`Ihr Vorname:  Oskar-Emanuel-Theodor`
	`Ihr Nachname: Schmitz`
	`Hallo, Oskar-Emanuel-Theod Schmitz!`

Falls der Benutzer die maximale Eingabelänge überschritten hat liefert **cin.rdstate()** eine 2, und ein so genanntes **failbit** ist gesetzt. In `recin()` wird mit der Methode **cin.clear()** das failbit zurückgesetzt. Außerdem werden die überzähligen Zeichen bis zum Trennzeichen '\n' (inklusive) mit **cin.ignore()** aus dem Eingabestrom entfernt. Mit der globalen Konstanten INT_MAX (= 2147483647) wird die maximale Zahl zu entsorgender Zeichen dabei so hoch gesetzt, dass in allen realistischen Fällen das Trennzeichen '\n' wirksam wird.
Wer will, kann eine eigene Funktion erstellen (z.B. `mygetline()` genannt), die mit Hilfe von **getline()** einen String von der Konsole entgegen nimmt und dann die vorgeschlagenen Reset-Maßnahmen ausführt.

5.1.7.4 Weitere Funktionen zur Bearbeitung von C-Strings

In diesem Abschnitt werden einige gängige Funktionen zur Bearbeitung von C-Strings (**char**-Arrays) kurz vorgestellt. Sie sind meist von der Header-Datei **cstring** (**string.h**) abhängig, wobei aber z.B. die Header-Datei **iostream** (zur neuen C++ - Standardbibliothek) ein explizites Inkludieren von **cstring** überflüssig macht.

Länge einer Zeichenfolge ermitteln

Mit der Funktion **strlen()** lässt sich die aktuelle Länge eines Strings ermitteln (also die Anzahl der definierten Zeichen), wobei das terminierende Nullzeichen *nicht* mitgezählt wird, z.B.:

Quellcode	Ausgabe

`#include <iostream>`
`using namespace std;`
 	
`void main()`
`{`
`  char szVorname[20] = "Otto";`
`  cout << strlen(szVorname) << endl;`
`  cout << sizeof(szVorname);`
`}`
	`4`
	`20`

Die **strlen()**-Funktionsschnittstelle wird in der VC++ - Online-Hilfe so beschrieben:

```
size_t strlen(const char *string);
```

Der darin verwendete Typbezeichner **size_t** ist nichts anderes als eine von Microsoft definierte Abkürzung für den Standardtyp **unsigned integer**.

Zeichenfolgen vergleichen

Zum Vergleichen zweier Strings eignet sich die Funktion **strcmp()** mit der Schnittstelle:

```
int strcmp(const char *string1, const char *string2);
```

Sie liefert folgende Rückgabewerte:

Wert	**Beziehung zwischen *string1* und *string2* (hinsichtlich der Sortierordnung)**
< 0	***string1*** < ***string2***
0	***string1*** ist identisch mit ***string2***
> 0	***string1*** > ***string2***

Beispiel:

Quellcode	Ausgabe

`#include <iostream>`
`using namespace std;`
 	
`void main()`
`{`
`  char szVorname[20] = "Otto";`
`  cout << strcmp(szVorname, "Otto") << endl;`
`  cout << strcmp(szVorname, "Lotto") << endl;`
`  cout << strcmp("bar", "bank") << endl;`
`  cout << strcmp("alte", "alter");`
`}`
	`0`
	`1`
	`1`
	`-1`

Konvertieren in Groß- bzw. Kleinschreibung

Speziell vor dem Vergleichen zweier Zeichenfolgen ist es oft sinnvoll, alle Buchstaben in Groß- bzw. Kleinschreibung zu übersetzen. Dazu stehen in Microsoft Visual C++ die Funktionen **_strupr()** und **_strlwr()** bereit, die allerdings nicht zum ANSI-Standard gehören:

```
char *_strupr(char *string);
```

```
char *_strlwr(char *string);
```

Beispiel:

Quellcode	Ausgabe

`#include <iostream>`
`using namespace std;`
 	
`void main()`
`{`
`  char strH1[] = "Hase";`
`  char strH2[] = "HasE";`
`  cout << _strupr(strH1) << endl;`
`  cout << _strlwr(strH1) << endl;`
`  cout << strcmp(strH1, strH2) << endl;`
`  cout << strcmp(_strlwr(strH1), _strlwr(strH2));`
`}`
	`HASE`
	`hase`
	`1`
	`0`

Beachten Sie bitte, dass **_strupr()** und **_strlwr()** die als Aktualparameter übergebene Zeichenfolge verändern.
Wer die Einschränkung der Kompatibilität vermeiden will, kann z.B. auf Basis der ANSI-Funktionen **toupper()** und **tolower()**, die jeweils ein einzelnes Zeichen konvertieren, eine eigene Funktion für Strings konstruieren.

Eine Zeichenfolge nach einer anderen Zeichenfolge durchsuchen

Oft ist man daran interessiert, ob eine bestimmte Zeichenfolge in einer anderen enthalten ist, und wo sich ggf. die erste Trefferstelle befindet. Diese Informationen liefert die Funktion **strstr()**:

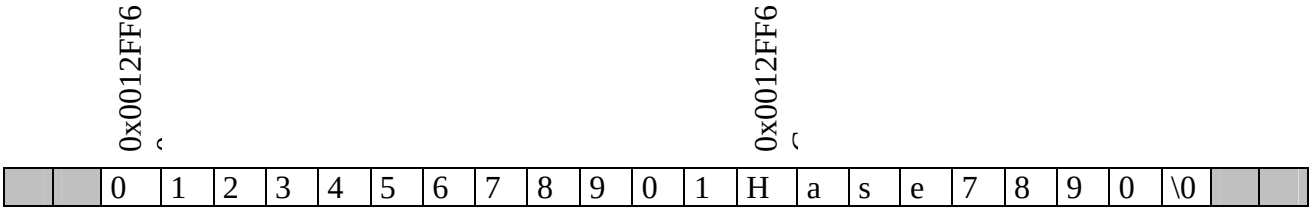
```
char *strstr(const char *string1, const char *string2);
```

In folgendem Beispiel wird Jagd auf einen Hasen gemacht:

Quellcode	Ausgabe

`#include <iostream>`
`using namespace std;`
 	
`void main()`
`{`
`  char strH[] = "Hase";`
`  char strW[] = "012345678901Hase7890";`
`  int position;`
`  position = int(strstr(strW, strH));`
`  if (position)`
`  {`
`  position -= int(strW);`
`  cout << "Hase entdeckt ab strW["`
`  << position << "]\n";`
`  }`
`}`
	`Hase entdeckt ab strW[12]`

Bei der Erläuterung dieses Beispiels macht sich Pointer-Technik breit, die zum Glück in Bälde besprochen wird: Die Funktion **strstr()** meldet im Erfolgsfall die Speicheradresse der ersten Fundstelle, ansonsten einen Zeiger ins Nirwana, der bei einer Wandlung in der Datentyp **int** zur 0 wird.
Wenn von der Adresse einer Fundstelle noch die `strW`-Startadresse abgezogen wird, erhalten wir den Index des `strW`-Zeichens, an dem die Fundstelle beginnt:



Konvertierung von Zeichenketten in Zahlen

Die Funktionen **atoi()** bzw. **atof()** dienen zur Konvertierung einer Zeichenfolge in einen **int**- bzw. **double**-Wert:

```
int atoi(const char *string);
```

```
double atof(const char *string);
```

Beide Funktionen schneiden die zu konvertierende Zeichenfolge beim ersten irregulären Zeichen ab. Ist ein konvertierbarer Anfang vorhanden, wird dessen numerischer Wert zurückgeliert, ansonsten eine 0 (**atoi()**) bzw. 0.0 (**atof()**). Der Umgang mit kritischen Eingabedaten schränkt die Verwendbarkeit der beiden Funktionen stark ein, z.B.:

Quellcode	Ausgabe

`#include <iostream>`
`using namespace std;`
 	
`void main()`
`{`
`  char fint[4] = "3o1";`
`  cout << atoi(fint) << endl;`
 	
`  char fouble[5] = "2ee2";`
`  cout << atof(fouble) << endl;`
 	
`  char nix[4] = "jik";`
`  cout << atoi(nix) << endl;`
`}`
	`3`
	`2`
	`0`

Im ersten Beispiel war wohl eher 301 gemeint, im zweiten Beispiel vielleicht 2e2 (= $2 \cdot 10^2 = 200$). Die Regel, bei vollends unverständlichen Argumenten kommentarlos den Wert 0 zurück zu melden, ist vollends fragwürdig.

Besser arbeiten da schon die Funktionen **strtol()**, die Strings in **int**- bzw. **long**-Werte konvertiert (in VC++ identisch), und **strtod()**, die Strings in **double**-Werte konvertiert.
Beide melden zurück, bei welchem Zeichen die Konvertierung beendet wurde, so dass eine Erfolgskontrolle möglich ist. Dazu verwenden sie einen Parameter vom Typ *Pointer to Pointer to char*, so dass wir eine vollständige Erklärung der beiden Funktionen an dieser Stelle nicht möglich ist. Zum Glück steht die Behandlung der Pointer-Datentypen unmittelbar an, wobei Sie vermutlich aufgrund ihrer Kenntnisse um die Funktionen **strtol()** und **strtod()** eine kleine Extramotivation mitbringen werden.
In folgendem Beispielprogramm ist zu sehen, wie man den Erfolg einer Konvertierung über den Parameter mit dem mysteriösen Typ feststellen kann:

Quellcode	Ausgabe

`#include <iostream>`
`using namespace std;`
 	
`void main()`
`{`
`  char *ep;`
`  char sint[][4] = {"301", "3o1"};`
`  for (int i = 0; i <= 1; i++) {`
`  int ival = strtol(sint[i], &ep, 10);`
`  **if (int(*ep) == 0)**`
`  cout << "Wert  = " << ival << endl;`
`  else`
`  cout << "Fehler: "  << ep << endl;`
`  }`
 	
`  cout << endl << endl;`
`  char sdouble[][5] = {" 2e2", "2ee2"};`
`  for (i = 0; i <= 1; i++) {`
`  double dval = strtod(sdouble[i], &ep);`
`  **if (int(*ep) == 0)**`
`  cout << "Wert  = " << dval << endl;`
`  else`
`  cout << "Fehler: "  << ep << endl;`
`  }`
`}`
	`Wert  = 301`
	`Fehler: o1`
	
	`Wert  = 200`
	`Fehler: ee2`

Eine Konvertierung wird hier als erfolgreich beurteilt, wenn sie erst mit dem terminierenden Nullzeichen beendet wurde. Falls mit Leerzeichen am Ende des zu konvertierenden Strings zu rechnen ist, muss das Erfolgskriterium noch etwas nachgebessert werden.

Teilzeichenfolgen extrahieren

Mit Hilfe der Funktion **strncopy()** kann man aus einer Zeichenkette einen beliebige Teilzeichenfolge extrahieren. In folgendem Programm werden aus dem C-String `puffer` drei Teilzeichenfolgen extrahiert und in andere Stringvariablen übertragen:

`vorname`	erhält die	20	Zeichen von der (relativen) Position	0 bis 19
`name`	erhält die	20	Zeichen von der (relativen) Position	20 bis 39
`tel`	erhält die	11	Zeichen von der (relativen) Position	40 bis 50

Quellcode	Ausgabe

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

void main()
{
  char vorname[21], name[21], tel[11],
    puffer[] = "Konrad              Schmitt             565565    ";
  strncpy(vorname, puffer, 20);
  vorname[20] = '\0';
  strncpy(name, puffer+20, 20);
  name[20] = '\0';
  strncpy(tel, puffer+40, 11);
  cout << vorname << endl << name << endl << tel << endl;
}
```

```
Konrad
Schmitt
565565
```

Während `tel` das Terminierungszeichen von `puffer` übernimmt, muss dieses bei `vorname` und `name` noch nachgetragen werden.
Die Funktionsschnittstelle von **strncpy()** lautet:

char *strncpy(char ***strDest***, const char ****strSource***, size_t ***count***);

Der darin verwendete Typ **size_t** ist nichts anderes als eine von Microsoft definierte Abkürzung für den Standardtyp **unsigned integer**. Über diesen Parameter wir die Anzahl der zu kopierenden Zeichen festgelegt.
Wenn der gewünschte Substring nicht mit dem ersten Zeichen startet, addiert man zum zweiten Aktualparameter die Anzahl der am Anfang des Quellstrings zu ignorierenden Zeichen, z.B. kopiert

```
strncpy(name, puffer+20, 20);
```

den Substring vom 21. bis zum 40. Zeichen.
Warum dieser einfach zu handhabende, aber etwas dubiose Gebrauch des Additionsoperators („C-String + Zahl“) so gut funktioniert, erfahren Sie im Abschnitt 5.2.1.5.

5.1.8 Übungsaufgaben zu Abschnitt 5.1

1) Erstellen Sie bitte ein Programm zur Primzahlensuche mit dem **Sieb des Eratosthenes**. Dieser Algorithmus reduziert sukzessive eine Menge von Primzahl-Kandidaten, die initial alle natürlichen Zahlen bis zu einer Obergrenze enthält. Im ersten Schritt werden alle echten Vielfachen der Basiszahl 2 (also 4, 6, ...) aus der Menge gestrichen. Dann geschieht iterativ folgendes:

> Bestimme in der Restmenge das kleinste, noch nicht untersuchte, Element als neue Basiszahl *b* und streiche seine echten Vielfachen (also 2·*b*, 3·*b*, ...) aus der Menge.

Das Verfahren kann enden, wenn das kleinste, noch nicht untersuchte, Element der Restmenge größer als die halbe Obergrenze ist, weil man bei der laut Algorithmus anstehenden Verdopplung den zu untersuchenden Wertebereich verlassen würde. Anschließend enthält die Kandidatenmenge nur noch Primzahlen.
Zur Veranschaulichung soll der Algorithmus mit der Obergrenze 16 manuell durchgeführt werden:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Nach dem Streichen der echten Vielfachen der 2 verbleiben folgende Zahlen:

1	2	3		5		7		9		11		13		15	

Von der 2 ausgehend wird die nächste verbliebene Zahl bestimmt und als neue Basis festgelegt, im Beispiel also die 3. Nach dem Streichen der echten Vielfachen der 3 verbleiben noch folgende Zahlen:

1	2	3		5		7				11		13			

Von der 3 ausgehend wird die nächste verbliebene Zahl bestimmt und als neue Basis festgelegt, im Beispiel also die 5. Diesmal findet sich kein echtes Vielfaches, so dass sich der Restmenge nicht ändert. Das ist auch nicht verwunderlich, weil sie nur noch Primzahlen enthält:

1	2	3		5		7				11		13			

Von der 5 ausgehend wird die nächste verbliebene Zahl bestimmt und als neue Basis festgelegt, im Beispiel also die 7. Es findet sich erneut kein zu streichendes Vielfaches. Von der 7 ausgehend landen wir schließlich bei 11 und können den langweilig gewordenen Algorithmus endlich verlassen, weil die 11 größer als die halbe Obergrenze ist.
Nun sind Sie sicher davon überzeugt, dass es für eine übrig gebliebene Zahl *z* keinen Teiler (außer *z* und 1) gibt, dass es sich also um eine Primzahl handelt.
Im C++ - Programm zum Eratosthenes-Algorithmus eignet sich wohl ein Array ganz gut zur Aufnahme der Streichkandidaten.

2) Schreiben Sie eine Funktion, die das arithmetische Mittel zu den **double**-Werten in einem eindimensionalen Feld als Funktionswert zurückliefert.

3) Schreiben Sie eine Funktion zum Sortieren eines eindimensionalen **int**-Arrays. Dabei kann z.B. die (sehr einfache) **lineare Suche** eingesetzt werden:

- Für den Ausgangsvektor mit den Elementen 0, .., *k* wird das Minimum gesucht und an den linken Rand befördert. Dann wird der Vektor mit den Elementen 1 bis *k* analog behandelt, bis der gesamte Vektor geordnet ist.
- Bei jeder Teilaufgabe müssen wir das kleinste Element eines Vektors finden, was auf folgende Weise geschehen soll:
  - Wir gehen davon aus, das Element am linken Rand sei das kleinste (genauer: ein Minimum).
  - Es wird sukzessive mit allen anderen Elementen verglichen. Ist das Element an der Position *i* kleiner, so tauscht es mit dem „Linksaußen“ seinen Platz.
  - Nun steht am linken Rand ein Element, das die anderen Elemente mit Positionen kleiner oder gleich *i* nicht übertrifft. Es wird nun sukzessive mit den Elementen an den Positionen ab *i*+1 verglichen.
  - Nachdem auch das Element an der letzten Position mit dem Element am linken Rand verglichen worden ist, steht mit Sicherheit am linken Rand ein Element, zu dem sich kein kleineres findet.

4) In folgendem Programm erhält die Zeichenfolgen-Variable `strH2` eine Kopie des Inhalts von `strH1`. Zu welchem Ergebnis wird der anschließende Vergleich der beiden Variablen kommen?

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

void main()
{
   char strH1[5] = "Hase", strH2[5];
   strcpy(strH2, strH1);
   if (strH1 == strH2)
      cout << "identisch" << endl;
   else
      cout << "verschieden" <<endl;
}
```

Lassen Sie sich beim Ausprobieren des Programms durch das Ergebnis des Vergleichs `strH1 == strH2` nicht beunruhigen. Nach der Lektüre des nächsten Abschnitts werden Sie derlei Beobachtungen souverän erklären können.

5.2 Zeiger (Pointer) und dynamische Variablen

Nicht alle Probleme lassen sich elegant mit einer schon zur Übersetzungszeit genau bekannten Menge von fest definierten Variablen lösen. Betrachten wir als Beispiel eine Funktion, die eine unbekannte Anzahl von Sätzen sortieren soll. Auf die bisher erlernten Techniken angewiesen könnten wir gewisse Obergrenzen festlegen (z.B. maximal 100.000 Sätze zu je 1000 Zeichen) und in der Funktion lokal ein entsprechendes Feld definieren, z.B.:

```
char str[100000][1000];
```

Eine solche Funktion würde bei jedem Aufruf ca. 100 MB Hauptspeicher belegen, obwohl nur sehr selten eine so große Speichermenge erforderlich wäre.[1]
Mit Hilfe **dynamischer Variablen**, die nach Bedarf erzeugt und auch wieder gelöscht werden können, lassen sich Probleme variabler Größe sehr viel besser lösen. Bei ihrer Verwaltung spielen **Zeiger- bzw. Pointervariablen** eine entscheidende Rolle, deren Inhalte keine Daten im bisherigen Sinne sind, sondern Speicheradressen von anderen Variablen. Um von dynamischen Variablen profitieren zu können, müssen wir uns also zunächst mit Zeigervariablen beschäftigen.
Aber auch unabhängig von den außerordentlich nützlichen und wichtigen dynamischen Variablen gibt es für C++ - Programmierer viele Gründe, sich mit der Pointer-Technologie zu beschäftigen, z.B.:

- Viele Eigenschaften von Arrays (und C-Strings) sind mit dem Wissen über Zeigervariablen deutlich besser zu verstehen.
- Gelegentlich müssen wir ganz einfach *deshalb* die Startadresse einer Variablen in Erfahrung bringen, weil eine Funktion eben diese Adresse als Parameter von uns erwartet. C++ ist bekannt (ja berüchtigt) dafür, dass der Programmierer an vielen Stellen explizit mit Pointern jonglieren muss.

5.2.1 Zeigervariablen (Pointer)

Eine Zeigervariable nimmt als Wert die Speicheradresse einer Variablen bestimmten Typs auf. In folgendem Demonstrationsprogramm wird nach der „konventionellen“ **int**-Variablen `ganz` noch die Pointervariable `pganz` deklariert, welche als Wert die Adresse einer **int**-Variablen aufnehmen kann:

Quellcode	Ausgabe

`#include <iostream>`
`using namespace std;`
 	
`void main()`
`{`
`  int ganz = 4711;`
`  int *pganz = &ganz;`
`  cout << pganz << endl;`
`  cout << &pganz << endl;`
`  cout << *pganz << endl;`
`}`
	`0012FF7C`
	`0012FF78`
	`4711`

[1] Um einen Stack-Überlauf beim Funktionsaufruf zu verhindern, müsste die voreingestellten Stack-Größe (bei VC++ 6.0: 1 MB) stark erhöht werden (vgl. Abschnitt 4.2).

5.2.1.1 Deklaration einer Zeigervariablen

Zu jedem Datentyp *T* existiert in C++ der abgeleitete Typ *Zeiger auf T* für Variablen, die als Wert die Speicheradresse einer Variablen vom Typ *T* aufnehmen können. Wegen der auch bei Pointervariablen durchgehaltenen strengen Typisierung kann der Compiler z.B. verhindern, dass einer als **int**-Zeiger definierten Variablen die Speicheradresse eines **double**-Arrays zugewiesen wird.
Bei der Deklaration einer Pointervariablen wird dem Namen ein Stern vorangestellt, ansonsten gibt es keinen syntaktischen Unterschied zur Deklaration einer konventionellen Variablen des zugrunde liegenden Typs, so dass wir uns ein Syntaxdiagramm ersparen können. Beachten Sie jedoch:

- Wenn Sie z.B. mehrere **int**-Pointervariablen deklarieren möchten, müssen Sie den Stern vor *jeden* Variablennamen setzen. Er modifiziert also nicht die Bedeutung des Typ-Bezeichners, sondern die Bedeutung der Variablennamen.
- Der Stern ist *kein* Bestandteil der Variablennamen.

Beispiele:

`int *pganz;`
Hier wird die **int**-Pointervariable `pganz` definiert.

`int *pganz, i;`
Hier wird neben der **int**-Pointervariablen `pganz` die gewöhnliche **int**-Variabe `i` definiert.

`char *str;`
Zwischen einem *Array of char* und einem *Pointer to char* besteht eine enge Verwandtschaft, die bald im Detail erläutert wird.

Oft ist es sinnvoll, Zeigervariablen mit einem „neutralen“ Wert zu initialisieren, um Zugriffe auf unbestimmte Speicherbereiche auf jeden Fall auszuschließen. Für diesen Zweck ist in der Standardbibliotheks-Header-Datei **cstddef** (**stddef.h**) die Konstante **NULL** definiert, die für alle Zeigertypen verwendbar ist und bei den meisten C++ - Compilern die „Speicheradresse“ 0 anspricht, z.B.:

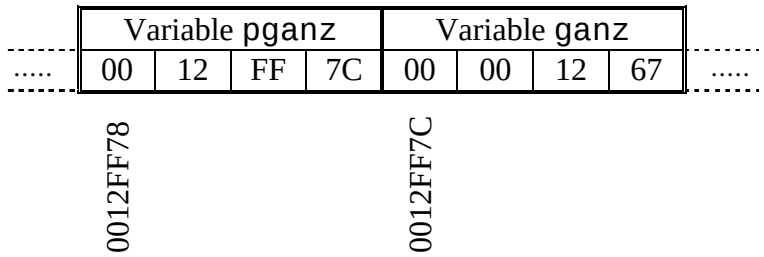
Quellcode	Ausgabe

`#include <iostream>`
`using namespace std;`
 	
`void main()`
`{`
`  int ganz;`
`  int *pganz = NULL;`
`  cout << pganz << endl;`
`  cout <<(int) pganz << endl;`
`}`
	`00000000`
	`0`

5.2.1.2 Adress- bzw. Referenzoperator

Im Beispielprogramm am Anfang von Abschnitt 5.2.1 wird die Pointervariable `pganz` bei der Definition auch gleich initialisiert, indem sie die Startadresse der Variablen `ganz` als Wert erhält, wobei der **Adressoperator &** zum Einsatz kommt, der auch als **Referenzoperator** bezeichnet wird. Das Zeichen **&** ist uns in einer verwandten Funktion schon einmal zur Kennzeichnung von Referenz*parametern* in Funktionsschnittstellen begegnet. (Dummerweise wird das Zeichen **&** aber auch für den bitweisen Und-Operator verwendet.)

Nun wird es Zeit, die Verhältnisse in dem vom Beispielprogramm belegten Speicher zu veranschaulichen. Die Werte in den 8 Bit breiten Speicherzellen und auch die senkrecht notierten Speicheradressen sind als Hexadezimalzahlen zu verstehen:



Dem Compiler hat es gefallen, abweichend von der Deklarationsreihenfolge die Pointervariable `pganz` vor die **int**-Variable `ganz` zu setzen:

- `pganz` beginnt an der Speicherstelle 0x0012FF78 (dezimal: 1245048)
- `ganz` beginnt an der Speicherstelle 0x0012FF7C (dezimal: 1245052)

Der von einer Pointer-Variablen belegte Platz hängt vom Betriebssystem ab und beträgt bei 32-Bit-Systemen gerade 4 Byte. Weil kein Vorzeichen benötigt wird, kann man über eine 32-Bit-Pointervariable 2^{32} = 4294967296 Speicherzellen adressieren (= 4 GB), was in der heutigen Zeit noch für die meisten Anwendungen genügt. Dass die **int**-Variable `ganz` ebenfalls 4 Byte belegt, wissen wir seit der Beschäftigung mit den Standarddatentypen.
Wesentlich an der ganzen Angelegenheit ist die Tatsache, dass der *Inhalt* der Pointervariablen `pganz` gerade die Startadresse der Variablen `ganz` ist, nämlich 0x0012FF7C.
Diesen Inhalt konnten wir im Beispielprogramm auf übliche Weise ausgeben:

```
cout << pganz << endl;
```

Die Startadresse von `pganz` liefert der **&**-Operator, der natürlich auch auf Pointervariablen angewendet werden kann:

```
cout << &pganz << endl;
```

5.2.1.3 Inhalts- bzw. Dereferenzoperator

In der letzten noch zu besprechenden Anweisung aus dem Beispielprogramm am Anfang von Abschnitt 5.2.1 tritt das Gegenstück zum Adressoperator auf: der **Inhaltsoperator**, der auch als **Dereferenzoperator** bezeichnet wird. Er muss sich leider mit der (semantisch deutlich verschiedenen) Pointervariablen-Deklaration (und natürlich auch mit der Multiplikation) das Symbol ***** teilen.

```
cout << *pganz << endl;
```

Durch Anwendung des Dereferenzoperators auf eine Pointervariable erhält man einen L-Wert, der wie ein Variablenbezeichner vollen Zugriff auf die Daten erlaubt, auf die der Pointer zeigt.
In folgender Variante des Beispielprogramms wird demonstriert, dass man auf `*pganz` nicht nur lesend, sondern auch schreibend zugreifen kann:

Quellcode	Ausgabe

`#include <iostream>`
`using namespace std;`
 	
`void main()`
`{`
`  int ganz = 4711;`
`  int *pganz = &ganz;`
`  *pganz = 9999;`
`  cout << ganz << endl;`
`  cout << *&ganz << endl;`
`}`
	`9999`
	`9999`

Weil ein Pointer wie jede andere Variable einen Typ besitzt, weiß der Compiler, wie viel Byte ab Startadresse einbezogen werden müssen, und wie diese zu interpretieren sind.

Der Inhaltsoperator kann als Umkehroperation zum Adressoperator aufgefasst werden. Wendet man beide nacheinander an, ergibt sich die Identität, d.h. für eine **int**-Variable `i` und einen **int***-Pointer `pi`:

- `*&i` ist äquivalent zu `i`
- `&*pi` ist äquivalent zu `pi`

5.2.1.4 Zeiger und Arrays

In C++ ist das Konzept eines Arrays eng verwandt mit dem eines Pointers: Der Name eines Arrays kann als **konstanter Zeiger** auf das erste Feldelement aufgefasst werden. Infolgedessen kann man z.B. einer **int**-Pointervariablen ein **int**-Feld zuweisen:

Quellcode	Ausgabe

`#include <iostream>`
`using namespace std;`
 	
`void main ()`
`{`
`  int ganzvek[3] = {1, 2, 3};`
`  int *ganzpoi, anoi = 7;`
`  ganzpoi = ganzvek;`
`  cout << "ganzvek[0]: " << *ganzpoi << endl;`
`  ganzpoi = &anoi;`
`  cout << "anoi:       " << *ganzpoi << endl;`
`// ganzvek = &anoi;`
`}`
	`ganzvek[0]: 1`
	`anoi:       7`

Während man der **int**-Pointervariablen `ganzpoi` die Adressen beliebiger **int**-Variablen oder -Felder zuweisen kann, ist `ganzvek` als *konstanter* Pointer auf die Speicheradresse eines fest im Speicher „verankerten“ **int[**3**]**-Vektors fixiert. Im Beispiel wird die Zuweisung

```
ganzvek = &anoi;
```

also mit einer Fehlermeldung quittiert:

```
error C2440: '=' : 'int *' kann nicht in 'int [3]' konvertiert werden
```

Nachdem wir tiefere Einblicke in die Natur eines C++ - Arrays gewonnen haben, können wir erklären, warum das für die Ansprache von Feldelementen verwendete Paar eckiger Klammern [] als **Offset-Operator** bezeichnet und in der Operatortabelle (siehe Anhang) aufgelistet wird. Dieser Operator stellt mit seinen Operanden (ein Pointer und eine nichtnegative Ganzzahl) Folgendes an:

- Die im Pointer-Argument gespeicherte Adresse wird um ein entsprechendes Vielfaches der Breite des zugrunde liegenden Datentyps erhöht.
- Die zugehörigen Speicheradressen werden im Sinne des zugrunde liegenden Datentyps interpretiert.

Dies kann man sowohl mit einer normalen Pointer-Variablen als auch mit einem konstanten Pointer (Arraynamen) veranstalten, z.B.:

Quellcode	Ausgabe

`#include <iostream>`
`using namespace std;`
 	
`void main ()`
`{`
`  int ganzvek[3] = {1, 2, 3};`
`  int *ganzpoi;`
`  ganzpoi = ganzvek;`
`  cout << ganzpoi[1] << " " << ganzvek[2] << endl;`
`}`
	`2 3`

Angewandt auf den Pointer `ganzvek`, der auf die Speicheradresse 0012FF74 zeigt, und den Index-Operandenwert 2 liefert der Offset-Operator z.B. das Ergebnis `3`.

5.2.1.5 Pointer-Arithmetik

Das Hantieren mit Speicheradressen wird in C++ durch eine spezielle Pointer-Arithmetik unterstützt: Ein Pointer `ptr` auf einen Datentyp mit ***b*** Byte Speicherbedarf enthalte aktuell die Adresse ***adr***. Eine ganzzahlige Variable `iof` habe den Wert ***i***. Dann liefert der Ausdruck

```
ptr + iof
```

den Wert:

$$adr + i \cdot b$$

Analoges gilt für das Subtrahieren eines ganzzahligen Wertes. Weitere Rechenarten sind aber nicht zugelassen.
In folgendem Programm wird die Pointer-Arithmetik demonstriert:

Quellcode	Ausgabe

`#include <iostream>`
`using namespace std;`
 	
`void main()`
`{`
`  double zahl, *zp = &zahl;`
`  cout << "Breite einer Double-Var.: "`
`  << sizeof(zahl) << endl;`
`  cout << "Wert von zp:  "`
`  << zp << endl;`
`  cout << "Wert von 'zp + 2':  "`
`  << (zp+2) << endl;`
`}`
	`Breite einer Double-Var.: 8`
	`Wert von zp:  0012FF78`
	`Wert von zp + 2:  0012FF88`

Die Differenz der beiden hexadezimalen Speicheradressen ergibt 0x10 = 1 · 16 + 0 = 16 = 2 · 8.
Besonders sinnvoll ist das Programm nicht, weil der Ausdruck `zp+2` auf eine undefinierte Speicherzelle zeigt.

Unter Verwendung der Pointer-Arithmetik kann man folgendermaßen vorgehen, um über eine Zeigervariable in einem eindimensionalen Array das Element mit dem Indexwert *k* anzusprechen:

- Der Zeigervariablen wird die Startadresse des Arrays zugewiesen, z.B.:
  ```
  int ganzvek[3] = {1, 2, 3};
  int *ganzpoi = ganzvek;
  ```
- Die im Pointer enthaltene Adresse wird per Pointer-Arithmetik erhöht.
- Der resultierende Ausdruck wird dereferenziert, z.B.:

```
cout << *(ganzpoi + 2);
```

Das folgende Programm nutzt den Offset-Operator und die Pointer-Arithmetik zum Zugriff auf die Elemente eines Arrays. Beide Techniken arbeiten sowohl mit der Arrayvariablen selbst als auch mit einem Pointer auf das erste Array-Element.

Quellcode	Ausgabe

`#include <iostream>`
`using namespace std;`
 	
`void main ()`
`{`
`  int ganzvek[3] = {1, 2, 3};`
`  int *ganzpoi = ganzvek;`
 	
`  cout << "Inhalte der Elemente:" << endl;`
`  for (int i = 0; i < 3; i++)`
`   cout << ganzvek[i] << " " << ganzpoi[i]`
`        << " " << *(ganzpoi+i) << " "`
`        << *(ganzvek+i)`
`        << endl;`
`  cout << "\nAdressen der Elemente:" << endl;`
`  for (i = 0; i < 3; i++)`
`   cout << &ganzvek[i] << " " << &ganzpoi[i]`
`        << " " << (ganzpoi+i) << " "`
`        << (ganzvek+i) << endl;`
`}`
	`Inhalte der Elemente:`
	`1 1 1 1`
	`2 2 2 2`
	`3 3 3 3`
	
	`Adressen der Elemente:`
	`0012FF74 0012FF74 0012FF74 0012FF74`
	`0012FF78 0012FF78 0012FF78 0012FF78`
	`0012FF7C 0012FF7C 0012FF7C 0012FF7C`

5.2.1.6 Zeiger und C-Strings

Wir wissen inzwischen:

- Ein C-String ist nichts anderes ist als ein **char**-Array.
- Ein Arrayname funktioniert als konstanter Pointer auf das erste Element.

So wie eben mit einer Variablen vom Typ **int*** nach geeigneter Initialisierung ein **int**-Array verwaltet werden konnte, kann man auch mit einer Variablen vom Typ **char*** und einem C-String verfahren, z.B.:

Quellcode	Ausgabe

`#include <iostream>`
`using namespace std;`
 	
`void main ()`
`{`
`  char str[] = "abc";`
`  char *strP = str;`
 	
`  cout << strP`
`     << " " << *strP`
`     << " " << strP[1]`
`     << " " << *(strP+2)<< endl;`
`}`
	`abc a b c`

In Abschnitt 5.1.7.2 haben Sie die Prototypen einiger Funktionen kennen gelernt, die offenbar für Variablen des Typs **char*** konzipiert sind, z.B.:

char *strcpy(char ****strDestination***, const char ****strSource***);

Wir konnten mit unserem damaligen Kenntnisstand die Funktionen nur aufgrund der Zusicherung verwenden, dass in Funktions-Prototypen folgenden Schreibweisen äquivalent seien:

char ***strDestination*** char ***strDestination***[]

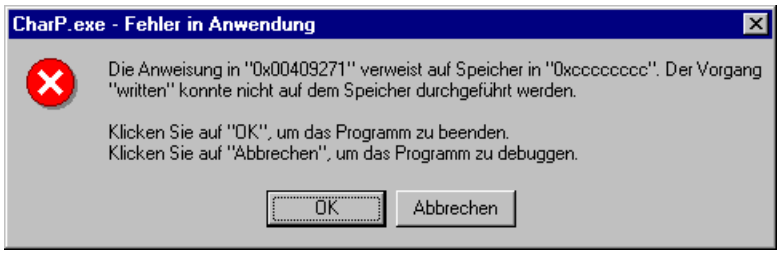
Mittlerweile sollte Ihnen diese Äquivalenzbehauptung einleuchten: In beiden Fällen wird der Funktion ein Zeiger auf einen C-String im Sinne eines Referenzparameters übergeben, wobei die Feldgrenzenüberwachung dem Programmierer obliegt.
Momentan macht der Einsatz von Variablen des Typs **char*** für uns noch keinen rechten Sinn, weil diese nur mit Hilfe von **char**-Arrays so initialisiert werden können, dass sie auf uneingeschränkt (auch schreibend) verwendbare Speicherbereiche zeigen. Unterlässt man eine solche Initialisierung, muss z.B. ein **strcpy()** - Aufruf kläglich scheitern:

```
#include <cstring>

void main ()
{
   char *strP;
   strcpy(strP, "Hallo");
}
```

Der Compiler warnt:

```
Lokale Variable 'strP' wurde ohne Initialisierung verwendet
```

und das Programm scheitert mit einem illegalen Speicherzugriff:



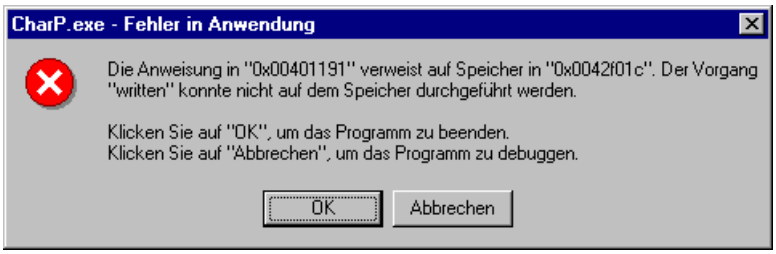
Warum der Datentyp **char*** so nützlich so unerhört verbreitet ist, wird erst nach der Beschäftigung mit dynamischen Variablen klar.

Man kann einer vom Typ **char*** zwar ein String-Literal zuweisen, darf aber anschließend nur lesend auf den Speicherbereich zugreifen, was mit folgendem Beipsielprogramm demonstriert werden soll:

Quellcode	Ausgabe

`#include <iostream>`
`using namespace std;`
`#include <cstring>`
 	
`void main ()`
`{`
`  char *strP = "abc";`
`  cout << strP[0] << endl;`
`  strP[0] = '1';`
`}`
	`a`

Obwohl man annehmen könnte, der Variablen `strP` durch die Initialisierung zu einem sinnvoll verwendbaren Speicherbereich verholfen zu haben, führt ein *Schreibzugriff* zu einem Laufzeitfehler:



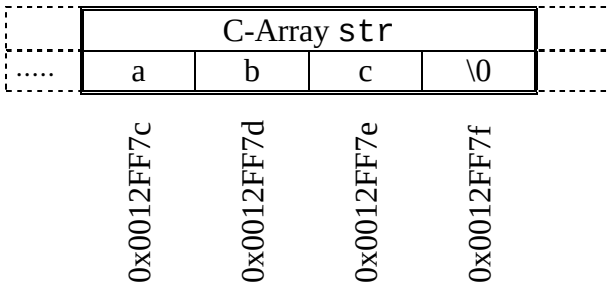
Dasselbe Ergebnis stellt sich auch dann ein, wenn der Variablen `strP` nach ihrer Definition ein String-Literal zugewiesen wird, z.B.:

```
char *strP;
strP = "abc";
```

Mit der Definition

```
char str[]= "abc";
```

bleiben uns derartige Überraschungen selbstverständlich erspart. Dabei reserviert der Compiler nämlich für `str` einen Speicherbereich von 4 Byte (auf dem Stack) und legt dort die Zeichenfolge „abc“ samt dem terminierenden Nullzeichen ab. `str` funktioniert als konstanter Zeiger auf die Speicheradresse mit dem Zeichen „a“.



Die Speicheradressen stammen aus folgendem Programm:

Quellcode	Ausgabe

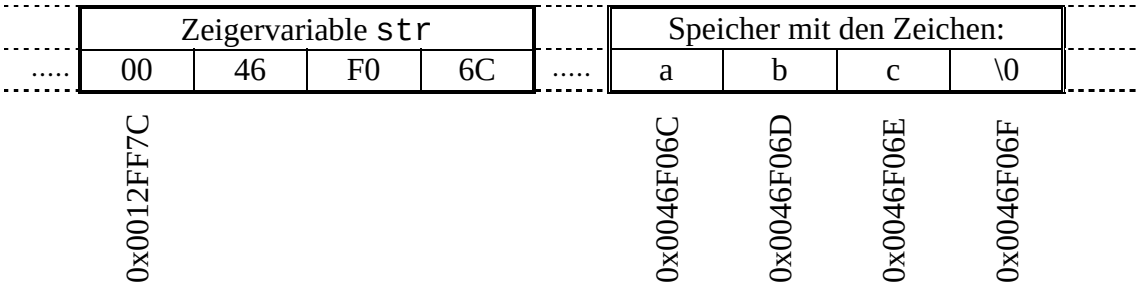
`#include <iostream>`
`using namespace std;`
 	
`void main ()`
`{`
`  char str[] = "abc";`
`  cout << "Adresse von str:  " << &str`
`  << endl << endl;`
 	
`  for (int i = 0; i <= 3; i++) {`
`  cout << "Adr. von Zeichen " << i << " : "`
`  << (int*)(str+i) << endl;`
`  }`
`  cout << endl;`
`  for (i = 0; i <= 3; i++)`
`  cout << "Inh. von Zeichen " << i << " : "`
`  << *(str+i) << endl;`
`}`
	`Adresse von str:  0012FF7C`
	
	`Adr. von Zeichen 0 : 0012FF7C`
	`Adr. von Zeichen 1 : 0012FF7D`
	`Adr. von Zeichen 2 : 0012FF7E`
	`Adr. von Zeichen 3 : 0012FF7F`
	
	`Inh. von Zeichen 0 : a`
	`Inh. von Zeichen 1 : b`
	`Inh. von Zeichen 2 : c`
	`Inh. von Zeichen 3 :`

Arbeitet man im Beispielprogramm mit dem Datentyp **chr***, dann resultieren andere Speicheradressen:

Quellcode	Ausgabe

`#include <iostream>`
`using namespace std;`
 	
`void main ()`
`{`
`  char *strP = "abc";`
`  cout << "Adresse von strP:    " << &strP`
`       << endl << endl;`
 	
`  for (int i = 0; i <= 3; i++) {`
`    cout << "Adr. von Zeichen " << i << " : "`
`         << (int*)(strP+i) << endl;`
`  }`
`  cout << endl;`
`  for (i = 0; i <= 3; i++)`
`    cout << "Inh. von Zeichen " << i << " : "`
`         << *(strP+i) << endl;`
`}`
	`Adresse von strP:    0012FF7C`
	
	`Adr. von Zeichen 0 : 0046F06C`
	`Adr. von Zeichen 1 : 0046F06D`
	`Adr. von Zeichen 2 : 0046F06E`
	`Adr. von Zeichen 3 : 0046F06F`
	
	`Inh. von Zeichen 0 : a`
	`Inh. von Zeichen 1 : b`
	`Inh. von Zeichen 2 : c`
	`Inh. von Zeichen 3 :`

Vor allem liegt der vom Zeiger `strP` angesprochene C-String nun an ganz anderer Stelle:



Der Pointer `strP` zeigt aufgrund der Initialisierung auf einen Array in einem speziellen Speicherbereich, wo der Compiler u.a. String-Literale ablegt. Hier darf das Programm zwar lesen aber aus guten Gründen nicht schreiben. Eine derartige Initialisierung ist nur mit dem Modifikator **const** sinnvoll, z.B.:

```
const char *str = "abc";
```

Nach der Definition sollte einem **char**-Pointer grundsätzlich kein String-Litaral zugewiesen werden.

5.2.1.7 Zeiger und Referenzparameter

Die mit C++ eingeführten Referenzparameter (siehe Abschnitt 4.1.2) sind nicht wirklich neu, sondern basieren auf der Verwendung von Pointern als Funktionsparameter. Sie erleichtern allerdings in vielen Situationen die Schreibarbeit, weil

- beim Funktionsaufruf keine Pointertypen übergeben werden müssen,
- in der Funktionsimplementierung beim Zugriff auf Referenzparameter das Dereferenzieren entfällt.

Das folgende Programm demonstriert die Verwandtschaft von Referenzparametern und Pointer-Parametern sowie die daraus resultierende Möglichkeit, *konstante* Referenzparameter innerhalb einer Funktions-Implementation hinterhältig doch zu verändern, z.B.:

Quellcode	Ausgabe

`#include <iostream>`
`using namespace std;`
 	
`void test(const int & i)`
`{`
`  int *hihi = (int*) &i;`
`  *hihi = 13;`
`}`
 	
`void main()`
`{`
`  int im = 4711;`
`  test(im);`
`  cout << im << endl;`
`}`
	`13`

Aus dem geschützten Referenzparameter `i` wird durch explizite Typumwandlung der freie Zeiger `hihi`.

Das Verfahren arbeitet analog auch bei der mit direkter Pointertechnik realisierten Referenzparameterlösung der Programmiersprache C:

Quellcode	Ausgabe

`#include <iostream>`
`using namespace std;`
 	
`void test(const int * i)`
`{`
`  int * hihi = (int*) i;`
`  *hihi = 13;`
`}`
 	
`void main()`
`{`
`  int im = 4711;`
`  test(&im);`
`  cout << im << endl;`
`}`
	`13`

5.2.2 Dynamische Variablen

Man kann es Ihnen nicht verübeln, wenn Sie in der Pointer-Technik bisher eher eine Denksportaufgabe als ein nützliches Werkzeug zur Lösung von Problemen gesehen haben. In Zusammenarbeit mit dynamischen Variablen machen sich Pointer aber so richtig nützlich.

Hinsichtlich der Lebensdauer haben wir bisher folgende Variablentypen kennen gelernt:

- Lokale Variablen
  Sie existieren im Stack-Speicher, solange der innerste umgebende Block ausgeführt wird.
- Globale und statische Variablen
  Sie leben während der gesamten Ausführungszeit des Programms in dessen Datenbereich.

Für beide Gattungen gilt:

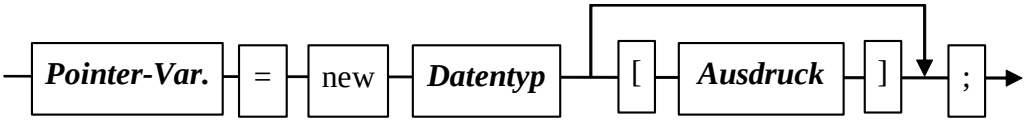
- Die Größe (z.B. eines Feldes) wird bereits zur Übersetzungszeit festgelegt. Wir werden daher im Folgenden gelegentlich zusammenfassend von *fixierten Variablen* sprechen.
- Der Compiler kümmert sich darum, Speicher zu reservieren und wieder frei zu geben.

Viele Probleme lassen sich mit fixierten Variablen schlecht lösen, weil der Speicherbedarf von der konkreten Aufgabe abhängt. Als Beispiel greifen wir die Primzahlen-Übungsaufgabe aus Abschnitt 5.1.8 noch einmal auf, die mit dem Sieb des Eratosthenes gelöst werden sollte. Dieser Algorithmus reduziert sukzessive eine Menge von Primzahl-Kandidaten, die initial alle natürlichen Zahlen bis zu einer Obergrenze *N* enthält. Um den aktuellen Staus der *N* Zahlen (noch Kandidat oder bereits gestrichen) zu speichern, kann man einen Array (z.B. mit Elementen vom Datentyp **bool**) verwenden. Während die benötigte Obergrenze vom Benutzer abhängt, muss man mit dem Kenntnisstand von Abschnitt 5.1.8 eine feste Anzahl von Feldelementen wählen und dabei einen Kompromiss aus Sparsamkeit und Flexibilität suchen.
Nun lernen wir dynamische Variablen kennen, die bedarfsgerecht zur Laufzeit eingesetzt werden können. Sie werden in einem Speicherbereich abgelegt, den man als **Heap** bezeichnet. Wir werden den Heap gleich mit den bereits bekannten Speichersorten vergleichen. Wie so oft im Leben muss auch bei den dynamischen Variablen die größerer Freiheit mit einer höheren Verantwortung erkauft werden. Während bei fixierten Variablen der Speicher vom Compiler reserviert und frei gegeben wird, obliegen diese Arbeiten bei dynamischen Variablen dem Programmierer, wobei potentielle Fehlerquellen im Auge zu behalten sind.

5.2.2.1 new-Operator

Mit dem **new**-Operator reserviert man einen Speicherbereich zur Aufnahme von Daten eines bestimmten Typs *T* und gewinnt einen Zeiger vom Typ *T**auf die Startadresse, der in einer Pointer-Variablen abgelegt wird. Über diesen Pointer und den Inhalts- oder den Offset-Operator lässt sich der dynamisch angelegte Speicherbereich dann wie eine gewöhnliche Variable verwenden.
Das folgende Diagramm beschränkt sich auf die wichtigsten Aspekte der **new**-Syntax:

new-Operator



Im folgenden Beispiel wird über das Syntaxdiagramm hinausgehend hinter dem Datentyp noch ein eingeklammerter Initialisierungswert angegeben:

```
int *ip = new int(4711);
```

Soll ein dynamisches *Feld* angelegt werden, so ist die Anzahl der Elemente über einen **ganzzahligen** Ausdruck anzugeben, dessen Wert aber keinesfalls zur Übersetzungszeit feststehen muss.
Im Eratosthenes-Beispiel kann man auf diese Weise ein Feld anlegen, das genau die benötigte Anzahl von Elementen enthält (nach Benutzerangaben in der Variablen `grenze` abgelegt):

```
bool *prim;
  . . .
prim = new bool [grenze+1];
```

Es ist auch möglich, ein *mehrdimensionales* Feld dynamisch per **new** zu erzeugen, wobei aber alle Dimension mit Ausnahme der ersten eine konstante Größe haben müssen, z.B.:

```
const int ncol = 128;
  . . .
int (*ip)[ncol], nrow;
  . . .
ip = new int[nrow][ncol];
```

Bei der Variablendefinition muss `ip` eingeklammert werden, damit ein Pointer auf ein **int**-Feld mit `ncol` Elementen entsteht und kein `ncol`-elementiges Feld von **int**-Pointern.

Kann beim **new**-Aufruf der gewünschte Speicher *nicht* reserviert werden, dann erhält die Pointer-Variable als Rückgabewert statt der Startadresse des reservierten Speicherblocks den Wert NULL. Anhand des **new**-Rückgabewertes kann also der Erfolg einer Speicheranforderung (alias: Speicherallokation) geprüft werden, was auch unbedingt geschehen sollte, z.B.:

```
if (prim == NULL)
{
  cout << "\aSpeichermangel!" << endl;
  continue;
}
```

5.2.2.2 Zugriff auf dynamische Variablen

Eine *einfache* (skalare) dynamische Variable wird über den zugehörigen Pointer und den Inhaltsoperator angesprochen, wobei kein Unterschied zum Pointerzugriff auf eine fixierte Variable besteht (vgl. Abschnitt 5.2.1.3), z.B.:

Quellcode	Ausgabe

`#include <iostream>`
`using namespace std;`
 	
`void main()`
`{`
`  int iv=4711, *ip = new int;`
`  **\*ip = 4711**;`
`  cout << iv << " " << **\*ip**;`
`}`
	`4711 4711`

Beim Zugriff auf ein dynamisch angelegtes *Feld* verwendet man die Pointervariable mit der Startadresse sowie den Offset-Operator, wobei *kein* Unterschied zum Standardzugriff auf ein fixiertes Feld besteht, z.B.:

Quellcode	Ausgabe

`#include <iostream>`
`using namespace std;`
 	
`void main()`
`{`
`  int fixa[10], *dyna = new int[10];`
`  fixa[5] = **dyna[5] =4711**;`
`  cout << fixa[5] << " " << **dyna[5]**;`
`}`
	`4711 4711`

Dies überrascht nicht, denn:

- Der Name eines normalen Arrays funktioniert wie ein Pointer auf das erste Element.
- Der Offset-Operator liefert den *Inhalt* der Speicherstelle, die er aus seinen Operanden (Pointer auf Startadresse und Offset-Wert) berechnet.

5.2.2.3 delete-Operator

Beim Beenden eines Programms wird der belegte dynamische Speicher automatisch freigegeben. Um bereits *zuvor* nicht mehr benötigten Speicher freizugeben, wendet man den **delete**-Operator auf einen passenden Zeigerwert an, z.B.:

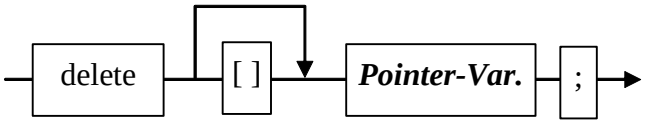
```
int *ip;
ip = new int;
  . . .
delete ip;
```

Handelt es sich um ein dynamisch angelegtes *Feld*, sollte dem Schlüsselwort **delete** ein Paar eckiger Klammern folgen, wenngleich nicht jeder Compiler darauf angewiesen ist, z.B.:

```
int *iap;
iap = new int[256];

delete[] iap;
```

Das Syntaxdiagramm für den **delete**-Operator:

delete-Operator



Nach der Freigabe eines Speicherbereichs sollte dessen Startadresse aus der betroffenen Pointer-Variablen entfernt werden, z.B.

```
delete ip;
ip = NULL;
```

Ansonsten kann ein Zugriff auf den (nunmehr undefinierten) Speicherbereich unangenehme Folgen haben kann. Beim Lesen erhält man zufällige Ergebnisse, beim Schreiben meist einen Absturz des Programms.
Natürlich erlaubt auch ein auf NULL zurück gesetzter Zeiger anschließend keine sinnvollen Speicherzugriffe mehr, doch wird nun auf jeden Fall das unbeabsichtigte und potentiell gefährliche Verarbeiten von Zufallsdaten verhindert.

Der **delete**-Operator setzt voraus, dass sein Argument auf einen per **new** angelegten, noch gültigen Speicherbereich zeigt und verursacht bei Verletzung dieser Voraussetzung meist einen Absturz des Programms.

Werden innerhalb einer Funktion dynamische Variablen angelegt, aber nicht frei gegeben, so überleben diese das Ende der Funktion. Existiert dann kein Pointer mehr, so können die belegten Speicherbereiche erst durch das Beenden des Programms frei gegeben werden. Die so entstehenden Speicherlöcher (memory leaks) sind vor allem solchen Programmen ein großes Problem, die längere Zeit ununterbrochen aktiv bleiben sollen.

Die Operatoren **new** und **delete** wurden mit C++ eingeführt. In den noch zahlreich vorhandenen C-Programmen werden statt dessen folgende Funktionen benutzt (deklariert in **cstdlib** bzw. **stdlib.h**):

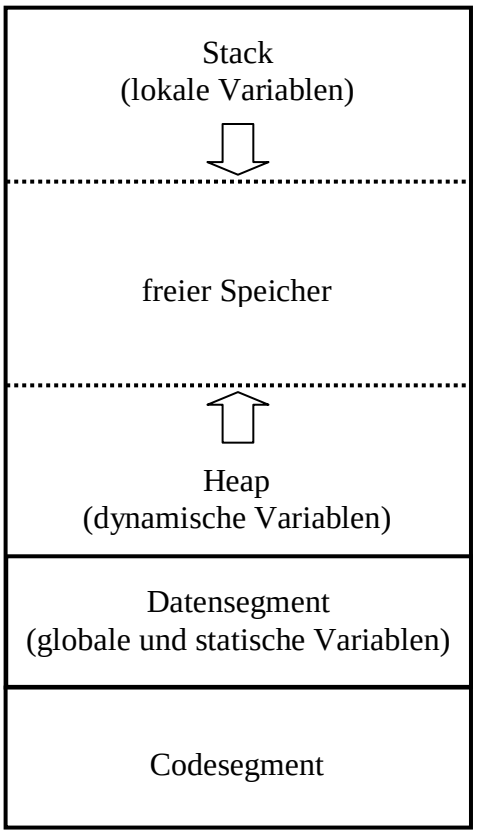
Name	**Funktion**
malloc, calloc	Speicher allokieren
realloc	Allokation ändern
free	Speicher freigeben

5.2.2.4 Speicherklassen

Der im Hauptspeicher von einem C++ - Programm belegte Speicher lässt sich nach Funktion und Dynamik in vier Bereiche einteilen:

- **Stack**
  Hier werden (neben Verwaltungsdaten und Aktualparametern) die lokalen Variablen von Funktionen abgelegt. Nach der Beendigung einer Funktion wird der von ihren lokalen Variablen belegte Speicher automatisch frei gegeben.
- **Heap**
  Hier werden die dynamischen Variablen abgelegt, für deren Verwaltung der Programmierer verantwortlich ist.
- **Datensegment**
  Hier befinden sich die globalen sowie die statischen Variablen (in Funktionen mit Schlüsselwort **static** definiert). Sie werden beim Programmstart angelegt und initialisiert.
- **Codesegment**
  Hier befindet sich der ausführbare Maschinencode.

Abgesehen von Implementationsunterschieden sind die vier Speicherbereiche grundsätzlich folgendermaßen angeordnet:



Während Code- und Datensegment eine feste Größe besitzen, verändern Stack und Heap dynamisch ihren Umfang. Damit sie sich dabei nicht ins Gehege kommen, starten sie beim Laden des Programms an entgegen gesetzten Enden des freien Speichers und wachsen dann ggf. aufeinander zu.

Wir haben bereits die Lebensdauer als Einteilungsgesichtspunkt für Variablen kennen gelernt (siehe Abschnitt 4.8). Bei dieser Dimension gesellt sich durch die Einführung der dynamischen Variablen zu den bisherigen Kategorien *Block* und *Programm* noch eine dritte Alternative: von **new** bis **delete** (bzw. Programmende). Mit unserem erweiterten Kenntnisstand können wir an Stelle der Lebensdauer nun auch den etwas abstrakteren, offiziell zur Einteilung der C++ - Vaiablen verwendeten Begriff der **Speicherklasse** setzen. In der folgenden Tabelle nach Schiedermaier (1997) sind die wichtigsten Attribute der C++ - Speicherklassen beschrieben:

Attribute	Speicherklassen		
	automatisch	**statisch**	**dynamisch**
Lebensdauer	ab Definition bis zum Ende des innersten Blockes	gesamte Programmlaufzeit	ab **new**, bis **delete** oder Programmende
Lebensraum	Stack	Datensegment	Heap
Verwaltung	automatisch	automatisch	Programmierer
Definition	lokal *ohne* Schlüsselwort **static**	global oder lokal *mit* Schlüsselwort **static**	Typvereinbarung per **new**, Zugriff über Zeiger

5.2.3 Übungsaufgaben zu Abschnitt 5.2

1) Verändern Sie Ihr Eratosthenes-Programm so, dass ein dynamisches Feld zum Einsatz kommt.

2) Erstellen Sie ein Programm zur „Produktion von Speicherlöchern“, und beobachten Sie mit Hilfe eines Task-Managers den Effekt auf den Hauptspeicherbedarf. Zumindest unter Windows NT/2000/XP steht ein geeignetes Diagnoseprogramm bereit, das hier mit **Strg + Alt + Entf > Task-Manager** gestartet wird.

5.3 Strukturen und andere benutzerdefinierte Datentypen

5.3.1 Strukturen

Zusammengehörige Daten unter einem Variablennamen ansprechen zu können, vereinfacht das Programmieren erheblich. Bisher haben wir mit den Arrays und C-Strings nur zusammengesetzte Variablentypen mit *homogenen* Elementen kennen gelernt, die in vielen Situationen keine ausreichend flexiblen Modellierungsmöglichkeiten bieten.
Mit den so genannten **Strukturen** bietet C++ die Möglichkeit, eigene Datentypen mit Elementen unterschiedlicher, problemadäquater Bauart zu entwerfen. Eine Struktur fasst also Werte unterschiedlichen Typs zusammen. In der Strukturdeklaration wird der Compiler über die Namen und Datentypen der als **Elementvariablen** bezeichneten Bestandteile informiert.

5.3.1.1 Beispiel

In diesem Abschnitt verwenden wir zur Demonstration erstmals ein etwas aufwändigeres Beispiel, wobei dem Vorteil der größeren Praxisnähe der Nachteil des umfangreicheren Quellcodes gegenüber steht. Das Programm soll Adressen verwalten und verwendet dazu eine Struktur mit drei Elementvariablen:

Elementvariable	Datentyp
vorname	C-String mit 21 Zeichen
name	C-String mit 21 Zeichen
alter	**int**

Das Alter ist kein sonderlich nützlicher Eintrag in einer Adressendatenbank, doch sollte die Beispiel-Struktur aus didaktischen Gründen nach zwei C-Strings auch noch eine numerische Elementvariable enthalten.

Quellcode	Ein- und Ausgabe

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

const int maxadr = 100;

struct Adresse
{
  char vorname[21];
  char name[21];
  int alter;
};

char frageBenutzer();
void zeigeAdressen(Adresse adrdb[], int aktpos);
void neueAdresse(Adresse &neue);

void main()
{
   Adresse adrdb[maxadr];
   int aktpos = -1;
   char antwort;

   while ((antwort = frageBenutzer()) != 'x')
   {
      switch (antwort)
      {
         case 'a': if (aktpos >= 0)
                     zeigeAdressen(adrdb, aktpos);
                   break;
         case 'n': if (aktpos < maxadr-1)
                     neueAdresse(adrdb[++aktpos]);
                   break;
         case 'd': if (aktpos >= 0)
                     aktpos--;
                   break;
         case 'x': break;
      }
   }
}
```

```
Adressenverwaltung:
------------------
Was moechten Sie tun?
  a = Adressen anzeigen
  n = neue Adresse aufnehmen
  d = letzte Adresse loeschen
  x = Programm beenden
Ihre Wahl: a


Anzahl der Adressen: 1

Vorname: Konrad
Name:    Schmitt
Tel.:    56


Adressenverwaltung:
------------------
Was moechten Sie tun?
  a = Adressen anzeigen
  n = neue Adresse aufnehmen
  d = letzte Adresse loeschen
  x = Programm beenden
Ihre Wahl:
```

Den vollständigen Quellcode mit den Funktionsdefinitionen finden Sie in der Datei **AdrVwArray.cpp**, die sich im Verzeichnis **BspUeb** mit Beispielen und Übungsaufgaben befindet (vollständige Ortsangaben: siehe Vorwort). Später werden wir das Beispiel so erweitern, dass es die eingegebenen Adressen beim Beenden abspeichert und beim Starten wieder einliest.

Mit fundierten Kenntnissen über Funktionen und Strukturen werden Sie bald bestens präpariert sein für den Einstieg in die objektorientierte Programmierung.

5.3.1.2 Definition einer Struktur

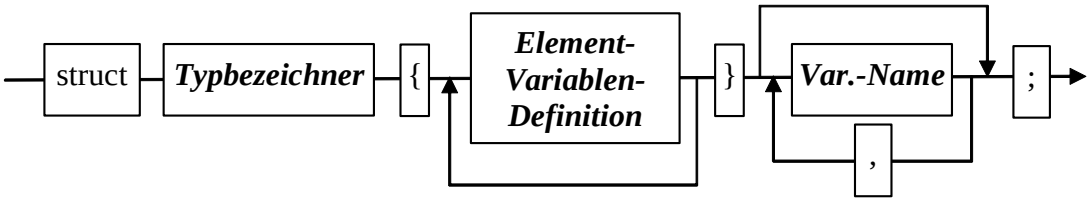
Im Beispielprogramm wird auf globaler Ebene die Struktur `Adresse` definiert, womit wir erstmals einen eigenen Datentyp entwerfen. Einer verbreiteten C++ - Konvention folgend, wollen wir für Typbezeichner folgende Regeln beachten:

- Der erste Buchstabe wird groß geschrieben.
- Der Rest wird in Kleinbuchstaben geschrieben, wobei jedoch Wortbestandteile zur Verbesserung der Lesbarkeit wiederum mit einem Großbuchstaben beginnen sollen.

Für die Elementvariablen sind alle bereits bekannte Datentypen (auch selbst definierte) erlaubt, so dass wir über sehr flexible Möglichkeiten zum Entwurf von problemadäquaten Strukturen verfügen. Im Beispiel wurde der Übersichtlichkeit halber das Konzept *Adresse* nur sehr einfach ausgestaltet.

Das Syntaxdiagramm zur Strukturdeklaration enthält als einzige Neuerung im Vergleich zum obigen Beispiel die Möglichkeit, auch gleich Variablen des neuen Typs zu definieren:

Strukturdefinition



Wir hätten die zentrale Datenbank unseres Beispielprogramms zusammen mit dem zugrunde liegenden Datentyp auch so definieren können:

```
struct Adresse
{
  char vorname[21];
  char name[21];
  int alter;
} adrdb[maxadr];
```

Dies ist allerdings wenig attraktiv: Um den neuen Typ auch in den Unterprogrammen nutzen zu können, muss er global definiert werden. Globale *Variablen* sind allerdings aus Sicherheitsgründen möglichst zu vermeiden.
Offenbar sind die optionalen Variablennamen nach dem Block der Elementvariablen-Definitionen der Grund dafür, dass dem Block auf jeden Fall noch ein Semikolon folgen muss.

Bei der Definition der Elementvariablen ist die Syntax für gewöhnliche Variablendefinitionen anzuwenden.

Werden im Rahmen einer Strukturdefinition Variablen des neuen Typs definiert, dann darf der Typbezeichner fehlen. Allerdings können dann später keine weiteren Variablen dieses Typs mehr definiert werden.

In der Regel wird man Struktur-Datentypen global definieren (am Anfang des Quelltexts oder in einer Header-Datei), jedoch sind auch blockinterne Definitionen mit entsprechend beschränkter Gültigkeit möglich.

5.3.1.3 Zugriff auf Strukturvariablen und ihre Elemente

Aufgrund der globalen Strukturdefinition können in den Funktionen unseres Programms Variablen vom Typ `Adresse` definiert und verwendet werden, wobei es sich im Fall einer Adressenverwaltung anbietet, gleich ein *Feld* mit Elementen des benutzerdefinierten Strukturtyps anzulegen.
Beim Zugriff auf eine Variable oder auf ein Feldelement vom Typ `Adresse` wird stets eine komplette Struktur mit allen Elementvariablen angesprochen, was im Beispiel zu einer einfachen Übergabe einer kompletten Adresse zur Bearbeitung im Unterprogramm `neueAdresse` führt:

```
neueAdresse(adrdb[++aktpos]);
```

Würden wir etwa mehrere Felder benutzen, um die einzelnen Adressbestandteile zu verwalten, wäre eine viel kompliziertere Funktionsschnittstelle erforderlich.

Es ist auch möglich, den Inhalt einer Strukturvariablen per Zuweisungsoperator einer anderen Variblen gleichen Typs zu übergeben, z.B.:

```
Adresse alteAdresse = {"Fritz", "Schulz", 12}, neueAdresse;
neueAdresse = alteAdresse;
```

Wie das Beispiel weiterhin demonstriert, kann eine Strukturvariable im Rahmen der Definition auch gleich initialisiert werden.

Um ein *Element* einer Strukturvariablen anzusprechen, gibt man Variablen- und Elementnamen an und setzt zwischen beide den **Punktoperator**. Abgesehen von dieser plausiblen Bezeichnungsweise können die Elementvariablen wie gewohnt verwendet werden, z.B. in unserer Funktion `zeigeAdresse`, welche die in der Adressdatenbank vorhandenen Einträge nacheinander ausgibt:

```
void zeigeAdressen(Adresse adrdb[], int aktpos)
{
  cout << "Anzahl der Adressen: " << aktpos+1 << endl << endl;
  for (int i=0; i<=aktpos; i++)
  {
    cout << "Vorname: " << adrdb[i].vorname << endl;
    cout << "Name:    " << adrdb[i].name << endl;
    cout << "Tel.:    " << adrdb[i].alter << endl << endl;
  }
}
```

5.3.1.4 Pointer auf Struktur-Variablen

Eigentlich ist bei einer Adressdatenbank die Anzahl der Einträge ja zur Übersetzungszeit, so dass dynamische Datenstrukturen angesagt sind, die wiederum ohne Pointer-Variablen in C++ nicht zu haben sind. Im Beispielprogramm sollte also das lokale `Adresse`-Feld durch eine dynamische Variante ersetzt werden:

```
Adresse *adrdb = new Adresse[maxadr];
```

Nun kann die gewünschte Datenbankgröße `maxadr` zur Laufzeit festgelegt werden. Ansonsten sind im Quellcode keine Änderungen erforderlich, weil der Offset-Operator mit einer Pointervariablen ebenso arbeitet wie mit einem Arraynamen (= Pointer auf das erste Array-Element, vgl. Abschnitt 5.2.1.4). Wegen der im Offset-Operator „eingebauten“ Dereferenzierung ist kein Inhaltsoperator erforderlich.

Der Pointerzugriff auf eine *einfache* Strukturvariable ist etwas aufwändiger, vor allem bei der Ansprache von Elementen per Punktoperator, z.B.:

```
Adresse *adrptr = new Adresse;
strcpy((*adrptr).vorname, "Ludwig");
```

Die runden Klammern um `*adrptr` sind erforderlich, weil nicht `adrptr.vorname` sondern `adrptr` dereferenziert werden soll, und der Punktoperator die höhere Priorität besitzt (siehe Operatorentabelle im Anhang).
Weil derartige Pointerzugriffe bei Strukturen und vor allem bei den eng verwandten Objekten in C++ sehr oft benötigt werden, hat man zur Vermeidung des beschrieben Umstands den **Zeigeroperator** „->“ eingeführt, im Beispiel:

```
strcpy(adrptr->vorname, "Ludwig");
```

5.3.2 Aufzählungstypen

Angenommen, Sie wollten in Ihrer Adressdatenbank auch den *Charakter* der erfassten Personen notieren und sich dabei an den vier Temperament-Typen des griechischen Philosophen Hyppokrates orientieren: cholerisch, melancholisch, sanguin, phlegmatisch. Um das Merkmal mit genau vier Ausprägungen zu speichern, haben Sie verschiedene Möglichkeiten, z.B.

- Eine **char**-Variable mit der Kodierungsvorschrift ‚c' = cholerisch, ‚m' = melancholisch etc.
  Es wird wenig Speicher benötigt. Durch die Verwendung der Abkürzung ist der Quellcode etwas schwer zu lesen, z.B.:
  ```
  if (adr.temp == 'm') ...
  ```
  Die Zuweisung irregulärer Werte (z.B. ‚o') wird nicht automatisch verhindert.
- Eine Zeichkettenvariable zu Aufnahme der vollständigen Bezeichnung
  Hier wird relativ viel Speicherplatz benötigt, und Abfragen sind umständlich, z.B.:
  ```
  if (strcmp(adr.temp, "melancholisch") == 0) ...
  ```
- Eine **int**-Variable mit der Kodierungsvorschrift 0 = cholerisch, 1 = melancholisch etc.
  Diese Lösung hat ähnliche Vor- und Nachteile wie die **char**-Variante z.B.:
  ```
  if (adr.temp == 1) ...
  ```

Eine optimale Lösung sollte u.a. folgende Eigenschaften haben:

- Gut lesbarer, einfach aufgebauter Quellcode
- Geringer Speicherbedarf
- Das Zuweisen irregulärer Werte wird automatisch verhindert.

C++ bietet mit den **Aufzählungstypen** eine Lösung, welche die drei Kriterien gut erfüllt, z.B.:

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

enum Temperament {cholerisch, melancholisch, sanguin, phlegmatisch};

struct Adresse
{
  char  vorname[21];
  char  name[21];
  int  alter;
  Temperament temp;
};

void main()
{
   Adresse adr;

   adr.temp = melancholisch;
   if (adr.temp == melancholisch)
     cout << "Ein Gemuetsmensch!" << endl;
}
```

Nachdem der Aufzählungstyp in einer **enum**-Anweisung definiert worden ist, sind Wertzuweisungen und Abfragen bequem zu formulieren, und es resultiert ein sehr gut lesbarer Code. Weil der Compiler nur die definierten Werte akzeptiert, werden viele Fehler schnell aufgedeckt.
Schließlich wird der Aufzählungstyp intern wie ein vorzeichenfreier Ganzzahl-Typ mit 4 Byte Speicherplatzbedarf behandelt, d.h. letztlich sind die definierten Werte nur Etiketten für die Zahlen 0, 1, 2, ...

Die Syntax für die Definition eines Aufzählungstyps:

enum ***Typ-Bezeichner*** {***Wert1***, ***Wert2***, ***Wert3***, ...};

5.3.3 Datentypen benennen

Bei häufig benötigten, komplizierten Typbezeichnungen empfiehlt es sich, per **typedef**-Deklaration einen bequemen Ersatznamen zu vereinbaren.
Im folgenden Beispiel wird für den Datentyp **unsigned int** die neue Bezeichnung UINT vereinbart:

```
typedef unsigned int UInt;
UInt ui = 4711;
```

Setzt man in der **typedef**-Deklaration vor den neuen Typbezeichner einen Stern, steht er für einen Pointer auf den bereits bekannten Datenytp, z.B.

```
typedef unsigned int *UIntPtr;
```

Um einen Aliasnamen für ein Feld aus Elementen eines bekannten Datentyps zu gewinnen, setzt man hinter neuen Namen noch zwischen eckigen Klammern die Feldgröße, z.B.:

```
typedef char Satz[128];
```

5.3.4 Übungsaufgaben zu Abschnitt 5.3

1) Entwerfen Sie eine Struktur zur Aufnahme von Datumsangaben (Tag, Monat, Jahr). Für die Monatsangabe sollte dabei ein Aufzählungstyp verwendet werden.

2) Entfernen Sie bitte die Fehler aus folgendem Programm:

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

typedef unsigned short Alter;

struct Auto {
   char prod[20];char typ[20];Alter alter;
}

void main()
{
   Auto auf[3] = {{"Ford", "Granada", 20}, {"Opel", "Rekord", 24}};
   auf[2] = auf[1];
   cout << auf[2];
}
```

6 Objektorientiertes Programmieren

Das wesentliche Motiv bei der Entwicklung der objektorientierten Programmierung (OOP) war die Profitgier der Softwarebranche, die mit allen Mitteln versuchte, rationellere Programmierverfahren zu entwickeln und dabei ihr Heil vor allem in einer Weiterentwicklung der traditionellen Strukturierung bzw. Modularisierung sah. Speziell große Projekte profitieren von einer systematischen Zerlegung eines Programms in möglichst selbständige und abgekapselte Einheiten (Klassen), deren Verbesserungen und Weiterentwicklungen keine Anpassungen bei anderen Programmteilen erzwingen. Beim Entwickeln und Testen kann man sich auf einzelne Klassen konzentrieren, was wiederum die reibungsarme Teamarbeit mehrerer Programmierer erleichtert.

6.1 Einige Kernideen

6.1.1 Aufhebung der Trennung von Daten und Operationen

In der OOP wird die traditionelle Trennung von Daten und Funktionen aufgegeben. Die Strukturen der Programmiersprache C durch *Klassen* ersetzt, wobei diese Datentypen nicht nur durch eine Anzahl von *Eigenschaften* (Elementvariablen beliebigen Typs) charakterisiert sind, sondern auch *Handlungskompetenzen* (Methoden) besitzen, welche die Aufgaben der Funktionen übernehmen.
Methoden existieren also nicht *neben* den Datentypen, sondern als *Bestandteile* eines bestimmten Datentyps (einer Klasse).
In erster Näherung könnte sich ein alt gedienter C-Programmierer unter einer Klasse also die Kombination aus einer Struktur und einer Anzahl von Funktionen vorstellen.
Bei einer Klasse handelt es sich (wie bei einer Struktur) um einen Datentyp, wobei die mit diesem Typ erzeugten Variablen aber als *Objekte* bezeichnet werden. Die Begriffe *Klasse* und *Objekt* stehen also in derselben Relation zueinander wie die Begriffe *Datentyp* und *Variable*.

6.1.2 Datenkapselung

Eine Klasse sollte die Daten (Elementvariablen) ihrer Objekte gegenüber anderen Klassen so weit wie möglich verbergen und damit vor ungeschickten Zugriffen schützen. Im Idealfall kooperieren in einem Programm Objekte durch wechselseitigen Aufruf von Methoden miteinander, ohne den inneren Aufbau der jeweils anderen Objekte zu kennen (Datenkapselung, information hiding). Damit darf eine Klasse beliebig umgestaltet werden, solange ihre Methoden unverändert bleiben.
Der eben erwähnte C-Programmierer sollte sich also unter einer Klasse eine Struktur vorstellen, deren Innenleben nur noch dem Urheber bekannt ist. Andere Programmierer, die Objekte der Klasse in eigenen Anwendungen benutzen, können im Idealfall nur die vom Designer für die Öffentlichkeit frei gegebenen Methoden benutzen.

6.1.3 Vererbung

Bei der Deklaration einer neuen Klasse können alle Eigenschaften (Elementvariablen) und Handlungskompetenzen (Methoden) einer Basisklasse übernommen werden. Es ist also leicht, ein Softwaresystem um neue Klassen mit speziellen Leistungen zu erweitern, wobei alle alten Bestandteile erhalten bleiben. Mit diesem Vererbungsprinzip schafft die OOP glänzende Voraussetzungen für die rationellen Entwicklung und Wiederverwendung von Softwarebausteinen. Durch systematische Anwendung des Vererbungsprinzips entstehen mächtige Klassenhierarchien, die in beliebigen Projekten einsetzbar sind. Eine Klasse kann direkt genutzt werden (durch Erzeugen von Objekten) oder als Basisklasse dienen.

6.1.4 Modellierung der Problemstellung

Der objektorientierten Programmierung geht eine objektorientierte *Analyse* voraus, die alle bei einer Problemstellung involvierten Objekt und deren Beziehungen identifizieren möchte. Auf dem Weg der Abstraktion fasst man identische oder zumindest sehr ähnliche Objekte zusammen und definiert jeweils eine Klasse, die durch Eigenschaften (Elementvariablen) und Handlungskompetenzen (Methoden) gekennzeichnet ist. Es kommen Klassen für konkrete Elementen (z.B. Immobilien), aber auch für abstrakte Elementen (z.B. Notarverträge) in Frage.

6.1.5 Vergleich mit der strukturierten Programmierung

Wie Sie aus dem bisherigen Kursverlauf wissen, ist auch schon die strukturierte Programmierung bestrebt, ein Gesamtproblem in unabhängige Teilprobleme aufzuteilen. Auch über die Nutzung und Entwicklung von Funktionsbibliotheken wird die Wiederverwendung von Software ermöglicht. Beim Einsatz einer Funktions-Bibliothek traditioneller Art muss der Programmierer jedoch passende Daten bereitstellen, auf die dann vorgefertigte Funktionen losgelassen werden. Der Programmierer hat also seine Datensammlung *und* die Kollektion der verfügbaren Funktionen zu verwalten und zu koordinieren. Beim Einsatz einer Klassenbibliothek wird die Trennung von Daten und Operationen überwunden. Man erzeugt Objekte (Variablen vom Typ einer Klasse) und benutzt deren Methoden. Die Objekte beinhalten die zur Lösung ihrer Aufgabe erforderlichen Daten, wobei diese dem Anwender einer Klasse aber verborgen bleiben sollten.

6.2 Beispielprogramm zum Bruchrechnungstraining

Bevor die theoretischen Ausführungen allzu langatmig und nebulös werden, betrachten wir ein Beispiel. Wir widmen uns der Aufgabe, eine Software für den Mathematikunterricht zu entwickeln, die das Erlernen der Bruchrechnung unterstützt.
Von den einleitend genannten Kernideen der OOP wird im Beispiel nur die Datenkapselung realisiert. Vererbung und Modellierung gewinnen bei größeren Beispielen rasch an Bedeutung.

6.2.1 Traditionelle Lösung

Vor der ersten objektorientierten Programmvariante betrachten wir eine traditionelle Lösung, um den Unterschied zwischen beiden Ansätzen sowie die Risiken der traditionellen Technik veranschaulichen zu können.
Um Zähler und Nenner eines Bruchs bequem verwalten zu können, definieren wir eine Struktur mit entsprechenden Elementvariablen. Zur Bearbeitung von Variablen vom definierten Strukturtyp definieren wird eine Ausgabe-Funktion sowie eine Funktion zum Erweitern von Brüchen.

Quellcode	Ausgabe

`#include <iostream>`
`using namespace std;`
 	
`struct Bruch`
`{`
`  int zaehler;`
`  int nenner;`
`};`
 	
`void zeige(const Bruch &x);`
`void erweitere(Bruch &x, int faktor);`
	`3`
	`-----`
	`6`
	
	`5`
	`-----`
	`7`
	
	`Gemeinsamer Nenner: 7`

```
void main()
{
  Bruch b1, b2;

  b1.zaehler = 1;  b1.nenner  = 2;
  b2.zaehler = 5;  b2.nenner  = 7;
  erweitere(b1, 3);
  zeige(b1);
  zeige(b2);

// Versehentliche Veränderung von b1.nenner
// durch falschen Vergleichsoperator:
  if (b1.nenner = b2.nenner)
    cout << "\nGemeinsamer Nenner: "
        << b1.nenner << endl;
}

void zeige(const Bruch &x)
{
  cout << endl << x.zaehler << "\n-----\n"
       << x.nenner << endl;
}

void erweitere(Bruch &x, int faktor)
{
  x.zaehler *= faktor;
  x.nenner  *= faktor;
}
```

In der folgenden Anweisung bewirkt ein vergessenes Gleichheitszeichen, dass `b1.nenner` versehentlich auf den Wert von `b2.nenner` gesetzt wird:

```
if (b1.nenner = b2.nenner)
cout << "\nGemeinsamer Nenner: " << b1.nenner << endl;
```

Was hier betont werden soll, ist die „Schutzlosigkeit“ der Elementvariablen `b1.nenner` gegenüber den allgegenwärtigen Fehlern der Programmierer bzw. gegen Übergriffe durch andere Programmbestandteile.
Bei unseren bisherigen Programmen gehörten die Daten zur Funktion **main()** und waren für alle von **main()** aufgerufenen Funktionen frei verfügbar. In einer solchen Situation kann ein Programmierer mit seinen Variablen sehr Vieles anstellen, Sinnvolles, aber auch Gefährliches.
Zur Vermeidung solcher Risiken empfiehlt es sich, Strukturen durch **Klassen** zu ersetzen, deren private Daten von anderen Programmbestandteilen nur über bestimmte Methoden beeinflusst werden können, wobei sich Missgriffe leichter verhindern lassen.

6.2.2 Objektorientierte Lösung

In der ersten objektorientierten Variante unseres Programms zur Didaktik der Bruchrechnung wird eine Klasse namens `Bruch` definiert. Sie verfügt wie die gleichnamige Struktur der traditionellen Programmvariante über die **int**-Elementvariablen `zaehler` und `nenner`. Doch sind diese nun als **private** definiert, so dass andere Programmbestandteile *nicht* direkt darauf zugreifen können. Zwar ist diese Schutzstufe bei Klassen voreingestellt, sollte aber der Klarheit halber (wie im Beispiel) trotzdem explizit angegeben werden.
Das Fehlen *direkter* Zugriffsmöglichkeiten darf natürlich nicht dazu führen, dass die Elementvariablen einer Klasse isoliert und damit nutzlos bleiben. Mit Hilfe klasseneigener Methoden werden sinnvolle Kommunikationsmöglichkeiten geschaffen. Man definiert sie als **public**, wenn anderen Programmbestandteilen die Verwendung erlaubt werden soll.

Im Beispiel werden für den Zugriff auf Zähler bzw. Nenner eines Bruchs die Methoden `sz`, `sn`, `hz` und `hn` definiert. Man kann trotzdem von einem *information hiding* sprechen, weil dem Anwender der genannten Methoden die klasseninterne Speicherorganisation *nicht* bekannt ist. Untypisch an unserem Beispiel ist, dass für *jede* Elementvariable eine Lese- und eine Schreibmethode vorliegt. Dies ist im allgemeinen weder erforderlich noch sinnvoll.

Quellcode	Ausgabe

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

class Bruch
{
 public:
  void sz(int z) {zaehler = z;}
  void sn(int n) {if(n != 0) nenner = n;}
  int hz() {return zaehler;}
  int hn() {return nenner;}
  void zeige();
  void erweitere(int faktor);
 private:
  int zaehler;
  int nenner;
};

void Bruch::zeige()
{
  cout << "\n  " << zaehler << "\n-----\n"
       << "  " << nenner << endl;
}

void Bruch::erweitere(int faktor)
{
  zaehler *= faktor;
  nenner  *= faktor;
}

void main()
{
  Bruch b1, b2;

  b1.sz(1); b1.sn(2);
  b2.sz(5); b2.sn(6);

  b1.zeige();
  b1.erweitere(3);
  b1.zeige();

  if (b1.hn() == b2.hn())
     cout << "\nGemeinsamer Nenner: " << b1.hn() << endl;
}
```

```
  1
-----
  2

  3
-----
  6

Gemeinsamer Nenner: 6
```

Vor der näheren Beschreibung syntaktischer Aspekte bei der Definition von Klassen soll belegt werden, dass die objektorientierte Reformulierung des Beispielprogramms tatsächlich dazu beiträgt, Fehler zu vermeiden. In der **struct**-Variante des Programms war ein Direktzugriff auf Elementvariablen mit falscher Verwendung des Operators „=“ und entsprechenden Folgen eingebaut:

```
if (b1.nenner = b2.nenner)
cout << "\nGemeinsamer Nenner: " << b1.nenner << endl;
```

In der **class**-Variante des Programms würde der Compiler einen solchen Zugriff verhindern:

```
U:\Eigene Dateien\VCpp\test\test.cpp(52) : error C2248: "nenner" : Kein Zugriff
auf private Element, dessen Deklaration in der Klasse "Bruch" erfolgte
       U:\Eigene Dateien\VCpp\test\test.cpp(15) : Siehe Deklaration von 'nenner'
```

Als wesentliche Vorteile der Datenkapselung, d.h. der Beschränkung auf „methodische“ Manipulationen der objektinternen Daten sollen noch einmal herausgestellt werden:

- Wenn jeder Zugriff auf eine Elementvariable über eine bestimmte Methode läuft, lassen sich unsinnige Manipulationen besser verhindern bzw. aufklären.
- Wenn die interne Datenorganisation der Klassenobjekte eine reine **private**-Angelegenheit ist, kann sie jederzeit geändert werden, solange die Schnittstellen der Methoden erhalten bleiben.

Der Vollständigkeit halber soll noch erwähnt werden, dass wir das Schlüsselwort **struct** hätten beibehalten können. In C++ unterscheiden sich Klassen nur darin von Strukturen, dass ihre Elementvariablen und Methoden per Voreinstellung **privat** sind, während bei Strukturen die Voreinstellung **public** gilt. Es hat sich allerdings eingebürgert, Klassendefinitionen mit dem Schlüsselwort **class** einzuleiten.

6.3 Klassendefinition

6.3.1 Genereller Aufbau

Bei der Klassendefinition[1] können Sie sich an folgender Pseudocode-Darstellung orientieren:

```
class Klassenbezeichner
{
public:
   öffentliche Elemente
private:
   private Elemente
};
```

Es hat sich in C++ eingebürgert, Typbezeichner (und damit auch Klassennamen) mit einem Großbuchstaben beginnen zu lassen.
Beachten Sie bitte das Semikolon hinter der schließenden Blockklammer, über deren Hintergrund wir uns schon im Zusammenhang mit der **struct**-Definition Gedanken gemacht haben.
Die Elemente (Variablen und Methoden) werden gelegentlich auch als *Member* bezeichnet, so dass von *Member-Variablen* bzw. *Member-Funktionen* die Rede ist.

Nach den einleitenden Bemerkungen zur objektorientierten Programmierung ist es sehr empfehlenswert, Elementvariablen als **private** und Methoden als **public** zu definieren, doch ist dies nicht vorgeschrieben. In vielen Situationen wird man auch private Methoden verwenden, die nur in anderen Methoden der eigenen Klasse verfügbar sind, und auch für öffentliche Variablen kann es sinnvolle Einsatzfälle geben.

[1] Viele Autoren sprechen von einer *Klassendeklaration*. Für die m.E. richtige Bezeichnung *Klassendefinition* spricht u.a. eine in Abschnitt 6.7.1 wieder gegebene Compiler-Beschwerde gegen das mehrfache Auftreten einer *Klassendefinition*.

6.3.2 Methoden deklarieren und definieren

Die Methoden, oft auch als *Elementfunktionen* bezeichnet, wird man in der Regel innerhalb der Klassendefinition *deklarieren* und an anderer Stelle *definieren* (implementieren), wie dies im Beispiel mit den Methoden `zeige()` und `erweitere()` geschehen ist. Dann ist es sinnvoll, die Klassendefinition in einer Header-Datei und die Implementationen in einer Quellcodedatei unterzubringen.
Bei einer „ausgelagerten“ Methodenimplementation muss dem Compiler allerdings unter Verwendung des Scope-Operatorsdie zugehörige Klasse mitgeteilt werden, weil jede Klasse ihren eigenen Namensraum besitzt, z.B.:

```
void Bruch::sn(int n)
{
  if (n != 0)
    nenner = n;
}
```

Die Funktion `sn` setzt den Nenner eines Bruches nicht blind auf den übergebenen Wert, sondern nimmt eine Plausibilitätsprüfung vor. Auch hier wird deutlich, wie das objektorientierte Programmieren für verbesserte Software-Qualität sorgen kann.

In den Methoden-Implementationen ist beim Zugriff auf Elemente der eigenen Klasse keine Namensbereichsangabe erforderlich.

In diesem Zusammenhang ist auf das potentielle Performanzproblem der objektorientierten Programmierung durch die zahlreichen Funktionsaufrufe hinzuweisen, die aus der Datenkapselung resultieren. Zum Ausgleich kann es bei sehr kleinen Methoden sinnvoll sein, eine **inline**-Definition vorzunehmen, was auf zweierlei Weise geschehen kann:

- Wird eine Methode innerhalb der Klassendefinition implementiert, wird sie vom Compiler auf jeden Fall als **inline**-Funktion behandelt.
- Bei einer „ausgelagerten“ Methodendefinition muss das für allgemeine Funktionen übliche Schlüsselwort **inline** vorangestellt werden, z.B.:

```
inline void Bruch::sz(int z) {zaehler = z;}
```

6.3.3 Klassenansicht im VC++ 6.0 - Arbeitsbereichsfenster

Nachdem wir eine eigene Klasse definiert haben, lohnt sich ein Blick auf das Arbeitsbereichsfenster:

Wie in VC++ 6.0 öffentliche bzw. private Elementfunktionen bzw. -variablen dargestellt werden, ergibt sich sofort aus dem Beispiel. Die Überblicksdarstellung der Klassenelemente lässt sich u.a.

dazu verwenden, um per Doppelklick schnell den Quellcode zu einem Element in den Vordergrund zu holen.

6.4 Objekte verwenden

Beim Definieren einer Objekt-Variablen spricht man zwar vom *Erzeugen einer Instanz*, kann aber genauso vorgehen wie bei beliebigen anderen Variablen. In der Hauptfunktion **main()** der objektorientierten Variante des Beispielprogramms werden durch die im Vergleich zur traditionellen Lösung unveränderte Anweisung

```
Bruch b1, b2;
```

zwei Objekte bzw. Instanzen der Klasse `Bruch` definiert.
Die Begriffe *Klasse* und *Objekt* stehen also in derselben Relation zueinander wie die Begriffe *Datentyp* und *Variable*.

Selbstverständlich lassen sich zu einer Klasse auch Zeigervariablen und dynamische Objekte generieren, z.B.:

```
Bruch *b3;
b3 = new Bruch;
   .   .   .
   .   .   .
delete b3;
```

Im Zusammenhang mit Konstruktoren (siehe unten) werden gleich noch wichtige Möglichkeiten zum *Initialisieren* von Objekten angesprochen.

Beim Aufruf einer Objektmethode und beim (eigentlich unerwünschten) Zugriff auf öffentliche Elementvariablen verwendet man analog zum Zugriff auf Strukturelemente den Punktoperator (bei lokalen oder statischen Objekten) bzw. den Zeigeroperator (bei dynamischen Objekten), z.B.:

```
b1.sz(1);
b3->sz(1);
```

Inspiriert durch die kompromisslos objektorientierten Programmiersprache Smalltalk wird in der OOP-Literatur gelegentlich der Aufruf einer Elementfunktion als **Botschaft** an das betroffene Objekt interpretiert. Im Beispiel erhält das Objekt `b1` (aus der Klasse `Bruch`) mit dem Methodenaufruf `b1.sz(1)` die Nachricht: „Setze Deinen Zähler auf den Wert 1“. Wer sich daran gewöhnt hat, in einem Objekt einen selbständigen Dienstleister zu sehen, wird die Nachrichten-Metapher recht sinnvoll finden.

6.5 Konstruktor und Destruktor

Beim Erzeugen eines Objektes müssen in der Regel seine diversen Elementvariablen initialisiert werden. Sofern ein Objekt dynamische Variablen anlegt, muss bei seinem Ableben der belegte Speicher freigegeben werden. Das Ableben eines Objektes kann wie bei jeder anderen Variablen folgende Ursachen haben:

- Ein lokal definiertes Objekt endet mit dem Block, in dem es erzeugt wurde.
- Ein dynamisch definiertes Objekt muss mit **delete** entsorgt werden.

C++ unterstützt die beim Objekteinsatz erforderlichen Vor- und Nacharbeiten durch spezielle Methoden, die beim Erzeugen bzw. Beenden eines Objektes automatisch ausgeführt werden.

6.5.1 Konstruktor

Beim Erzeugen eines Objektes wird automatisch die so genannte Konstruktor-Methode ausgeführt, die ohne Eigenleistung des Programmierers nur gewisse Basisroutinen enthält und im Quelltext nicht auftaucht. In aller Regel ist es sinnvoll, explizit einen Konstruktor zu definieren, wobei folgende Regeln zu beachten sind:

- Der Konstruktor trägt denselben Namen wie die Klasse.
- Er kann Argumente besitzen, was zum Zweck der Initialisierung ja auch unumgänglich ist. Wie bei einer gewöhnlichen Funktion oder Methode sind auch Parameter mit Voreinstellungen erlaubt.
- Er liefert keinen Rückgabewert, hat daher keinen Typ.
- Er kann für ein vorhandenes Objekt nicht explizit aufgerufen werden (z.B. mit der Syntax *objekt.methode*).
- Er muss als **public** deklariert werden, weil fremde Programmbestandteile diese Methode ausführen sollen.

Der Konstruktor wird aufgerufen, sobald der Speicher des neu erzeugten Objektes bereit gestellt ist:

- bei lokalen Objekten: immer, wenn der zugehörige Block betreten wird
- bei statischen lokalen Objekten: wenn der zugehörige Block zum ersten Mal betreten wird
- bei globalen Objekten: beim Programmstart, noch vor Ausführung der Funktion **main()**
- bei dynamischen Objekten: bei Ausführung des **new**-Operators

Für unser Beispielprogramm eignet sich z.B. der folgenden Konstruktor, der bei neuen `Bruch`-Objekten mit Hilfe seiner beiden Parameter den Zähler und den Nenner initialisiert:

```
Bruch::Bruch(int z, int n)
{
  cout << "Konstruktor Bruch startet\n";
  zaehler = z;
  if (n)
  {
    nenner = n;
    def = 1;
  }
  else
  {
    def = 0;
  }
}
```

Beim Erzeugen neuer `Bruch`-Objekte mit diesem Konstruktor sind nach dem Variablennamen zwischen runden Klammern (wie bei einem Methodenaufruf) die Aktualparameter anzugeben, z.B.:

```
Bruch b1(1, 2);
```

Die folgende Variante unseres Beispielprogramms enthält neben dem expliziten Konstruktor einige weitere Veränderungen:

- Um den fehlgeschlagenen Versuch, eine Nenner-Elementvariable mit dem Wert 0 zu initialisieren, kennzeichnen zu können, wird eine weitere Elementvariable namens `def` eingeführt, die bei sinnvollen Brüchen den Wert 1 zeigen soll, anderenfalls die 0.
- Es wird auch gleich ein Destruktor definiert, der aber (über die Standardroutinen hinaus) lediglich seinen Startzeitpunkt verrät.
- Beim Erzeugen eines Objektes müssen jetzt die Parameter des Konstruktors mit Werten versorgt werden. Die in der Vorgängerversion zum Initialisieren benutzten Methodenaufrufe sind damit überflüssig.
- Mit `addiere()` und `multipliziere()` wurden noch zwei `Bruch`-Methoden ergänzt.

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

class Bruch
{
 public:
   Bruch(int z, int n);
   ~Bruch();
   void sz(int z) {zaehler = z;}
   void sn(int n) {if(n != 0) nenner = n;}
   int hz() {return zaehler;}
   int hn() {return nenner;}
   void zeige();
   void erweitere(int faktor);
   void addiere(Bruch &b1, Bruch &b2);
   void multipliziere(Bruch &b1, Bruch &b2);
 private:
   int zaehler;
   int nenner;
   bool def;
};

Bruch::Bruch(int z, int n)
{
   . . .
   . . .
}

Bruch::~Bruch()
{
   cout << "\nDestruktor Bruch startet";
}

void Bruch::zeige()
{
   cout << "\n  " << zaehler << "\n-----\n"
        << "  " << nenner << endl;
}

void Bruch::erweitere(int faktor)
{
   zaehler *= faktor;
   nenner  *= faktor;
}

void Bruch::addiere(Bruch &b1, Bruch &b2)
{
   zaehler = b1.hz() * b2.hn() + b1.hn() * b2.hz();
   nenner  = b1.hn() * b2.hn();
}

void Bruch::multipliziere(Bruch &b1, Bruch &b2)
{
   zaehler = b1.hz() * b2.hz();
   nenner  = b1.hn() * b2.hn();
}

void main()
{
   Bruch b1(1, 2), b2(5, 6);
   b1.zeige();
   b1.erweitere(3);
   b1.zeige();
}
```

Konstruktoren und andere Methoden dürfen wie C++ - Funktionen *überladen* werden, so dass z.B. unterschiedliche Initialisierungs-Umfänge möglich sind. Wenn es etwa weiterhin möglich sein soll,

`Bruch`-Objekte ohne Initialisierung zu erzeugen, dann muss ein zusätzlicher Konstruktor definiert werden, z.B.:

```
Bruch::Bruch()
{
   cout << "Standard-Konstruktor Bruch startet\n";
}
```

Besitzt die Klasse `Bruch` beide Konstruktoren, dann können `Bruch`-Objekte mit und ohne Initialisierung erzeugt werden, z.B.:

```
Bruch b1(1, 2), b2;
```

Dabei ist zu beachten, dass dem Namen des uninitialisiert erzeugten Objektes *keine* leere Parameterliste folgend darf.

Aufgrund der Ausgabeanweisungen im Konstruktor und im Destruktor ist im Programmablauf zu erkennen, wann die beiden Objekte erzeugt und zerstört werden:

```
Konstruktor Bruch startet
Konstruktor Bruch startet

  1
-----
  2

  3
-----
  6

Destruktor Bruch startet
Destruktor Bruch startet
```

Ein Konstruktor kann beliebig komplexe Aktionen ausführen, was z.B. bei der MS-Windows-Programmierung mit MFC zutrifft, mit der wir uns im weiteren Verlauf des Kurses noch beschäftigen werden. Während bei der Definition einer traditionellen Variablen lediglich Speicher reserviert wird, kann ein frisch erzeugtes Objekt aufgrund seines Konstruktors bereits ein quicklebendiges Verhalten an den Tag legen, bevor andere Methoden aufgerufen werden.

Ein Konstruktor darf auch aufgerufen werden, um ein **temporäres Objekt** zu erzeugen, z.B.:

```
return Bruch(1, 2);
Bruch(1, 2).zeige();
```

Temporäre Objekte entstehen bei der Ausführung einer Anweisung und verschwinden anschließend wieder (unter der Verantwortung des Compilers).

6.5.2 Destruktor

Beim Ableben eines Objektes wird automatisch die so genannte Destruktor-Methode ausgeführt, die ohne Eigenleistung des Programmierers nur gewisse Basisroutinen enthält und im Quelltext nicht auftaucht. Ein expliziter Destruktor ist z.B. dann erforderlich, wenn Objekte mit dynamischen Variablen arbeiten. Anderenfalls verschwinden beim Zerstören des Objektes zwar die Zeiger, nicht aber die Speicherbelegungen, was zu den unbeliebten *Speicherlöchern* (engl.: *memory leaks*) führt.
Für den Destruktor gelten folgende Regeln:

- Sein Name entsteht aus dem Klassennamen durch Voranstellen einer Tilde (siehe obiges Beispiel).
- Es gibt pro Klasse genau *einen* Destruktor, der keine Argumente akzeptiert.
- Der Destruktor liefert keinen Rückgabewert, hat daher keinen Typ.

- Er kann nicht explizit aufgerufen werden.
- Er muss als **public** deklariert werden, weil fremde Programmbestandteile diese Methode ausführen sollen.

6.6 Zusammengesetzte Objekte (Aggregation)

So wie die Elementvariablen einer Struktur wiederum vom Strukturtyp sein dürfen, kann man bei der Klassendefinition auch Elementvariablen von einem Klassentyp verwenden. Damit ist es z.B. möglich, einfache Klassen als Bausteine für komplexere Klassen zu verwenden. Als Beispiel soll für das entstehende Bruchrechnungs-Trainingsprogramm eine Klasse namens `Aufgabe` unter Verwendung der Klasse `Bruch` entworfen werden.
Zunächst wird der Aufzählungstyp `Operation` definiert, dessen Ausprägungen die mit einer Aufgabe zu trainierende Operation speichern sollen.

```
enum Operation
{
   add,
   sub,
   mult
};
```

In der `Aufgabe`-Klassendefinition tauchen mehrere Elementvariablen vom Typ `Bruch` auf:

```
class Aufgabe
{
public:
   Aufgabe(operation typ);
   Aufgabe(operation typ, int b1Z, int b1N, int b2Z, int b2N);
   ~Aufgabe();
   void      anzeigen();
   int       fragen();
private:
   Bruch     b1, b2, lsg, antwort;
   Operation op;
   char      opchar;
   void      init();
};
```

In einer `Aufgabe` dienen die `Bruch`-Objekte folgenden Zwecken:

- `b1` und `b2` werden dem Anwender (in der Methode `fragen()`) im Rahmen eines Arbeitsauftrags vorgelegt, z.B. zum Addieren.
- In `antwort` landet der Lösungsversuch des Prüflings.
- In `lsg` steht das korrekte Ergebnis

Neben den Brüchen enthält die Klasse `Aufgabe` noch Elementvariablen für die vom Prüfling verlangte Operation. Statt eine Nummer und ein bildschirmtaugliches Zeichen zu verwenden, könnte man auch mit der **char**-Variablen auskommen. Weil die Elementvariablen **private** sind, kann diese Änderung später vorgenommen werden, ohne Programm anpassen zu müssen, die `Aufgabe`-Objekte verwenden.
Die Klasse `Aufgabe` besitzt zwei Konstruktoren, die Initialisierungsaufgaben in unterschiedlichem Umfang übernehmen:

```
Aufgabe::Aufgabe(operation typ)
{
   cout << "Standardkonstruktor Aufgabe startet\n";
   op = typ;
   init();
}
```

```
Aufgabe::Aufgabe(operation typ, int b1Z, int b1N, int b2Z, int b2N):
                 b1(b1Z, b1N), b2(b2Z, b2N)
{
   cout << "Konstruktor Aufgabe startet\n";
   op = typ;
   init();
}
```

Im zweiter Konstruktor muss ein Problem gelöst werden, dass sich aus der Aggregation von Klassen ergibt: Seine Parameter sind offenbar teilweise zur Initialisierung der `Bruch`-Elemente gedacht und müssen an einen geeigneten `Bruch`-Konstruktor durchgereicht werden. Zur Lösung des Problems bietet C++ so genannte **Initialisierungslisten** an, deren relativ simple Syntax sich aus unserem Beispiel gut ableiten lässt.
Für die Member-Objekte `b1` und `b2` werden Initialisierungswerte vom `Aufgabe`-Konstruktor an den initialisierenden Konstruktor der Klasse `Bruch` weiter gegeben. Bei `lsg` und `antwort` wird auf eine Initialisierung verzichtet, so dass hier der Einfach-Konstruktor der Klasse `Bruch` zum Einsatz kommt.
An dieser Stelle wird unbedingt ein parameterfreien `Bruch`-Konstruktor benötigt. Sobald zu einer Klasse ein Konstruktor explizit definiert wird, steht der Default-Konstruktor *nicht* mehr zur Verfügung. Nötigenfalls muss also zusätzlich ein parameterfreier Konstruktor definiert werden.

Die Konstruktoren der `Bruch`-Objekte werden vor dem Rumpf des `Aufgabe`-Konstruktors ausgeführt (siehe unten).
Die Implementierungsliste wird nur bei der Konstruktor-*Definition* angegeben, nicht bei seiner Deklaration im Rahmen der Klassendefinition.

Der `Aufgabe`-Destruktor wird explizit definiert, beschränkt sich aber auf eine Ausgabe zum Nachweis seiner Tätigkeit:

```
Aufgabe::~Aufgabe()
{
   cout << "\nDestruktor Aufgabe startet\n";
}
```

Beide Konstruktoren verwenden die Methode `init()`, die von der Öffentlichkeit nicht benötigt und daher als **private** definiert wird.

```
void Aufgabe::init()
{
  switch (op)
  {
     case add: opchar = '+';
            lsg.addiere(b1, b2);
            break;
     case mult:  opchar = '*';
            lsg.multipliziere(b1, b2);
            break;
  }
}
```

Für die Öffentlichkeit gedacht sind hingegen die Methoden `anzeigen()` und `fragen()`, die eine Aufgabe auf den Bildschirm bringen bzw. nach der Lösung fragen:

```
void Aufgabe::anzeigen()
{
   cout << "  " << b1.hz() << "                " << b2.hz() << endl;
   cout << "------     " << opchar << "     -----\n";
   cout << "  " << b1.hn() << "                " << b2.hn() << endl;
}
```

```
int  Aufgabe::fragen()
{
   int z, n;

   cout << "\nBerechne bitte:\n\n";
   anzeigen();
   cout << "\nWelchen Zaehler hat dein Ergebnis:     ";
   cin >> z;
   cout << "                                        ------\n";
   cout << "Welchen Nenner hat dein Ergebnis:      ";
   cin >> n;
   antwort.sz(z);
   antwort.sn(n);
   return 0;
}
```

Mit Hilfe der Hauptfunktion

```
void main()
{
   Aufgabe a1(add, 1, 2, 2, 5);
   a1.fragen();
}
```

komplettieren wir die beiden Klassendefinitionen zu einem Programm. Es ist zwar noch nicht praxistauglich, kann aber immerhin zeigen, wann die verschiedenen Konstruktoren und Destruktoren der geschachtelten Klassen ausgeführt werden:

```
Konstruktor Bruch startet
Konstruktor Bruch startet
Standard-Konstruktor Bruch startet
Standard-Konstruktor Bruch startet
Konstruktor Aufgabe startet

Berechne bitte:

  1               2
------     +     -----
  2               5

Welchen Zaehler hat dein Ergebnis:     9
                                      ------
Welchen Nenner hat dein Ergebnis:      10

Destruktor Aufgabe startet
Destruktor Bruch startet
Destruktor Bruch startet
Destruktor Bruch startet
Destruktor Bruch startet
```

Bei der Destruktion kehrt der Compiler im Vergleich zur Konstruktion sinnvollerweise die Reihenfolge um: Erst wird das (abhängige) komplexe Objekt beseitigt, dann die zugrunde liegenden Objekte.

6.7 Dateiorganisation

Um den wachsenden Quellcode übersichtlich zu halten und unsere Klassen bequem in anderen Anwendungen nutzen zu können, sollten wir für jede Klasse jeweils eine Header- und eine Implementierungsdatei verwenden. Dabei ist zu klären, welche Header-Dateien in den verschiedenen Dateien inkludiert werden müssen. Vorläufig wird folgende Systematik vorgeschlagen:

- Inkludieren Sie in jeder Klassen-Implementierungsdatei ausschließlich die zugehörige Klassen-Header-Datei.
- Inkludieren Sie in der Klassen-Header-Datei alle Standard- oder Projekt-Header-Dateien, die von irgendeinem Klassen-Element benötigt werden.
- Die Quellcodedatei mit der Hauptfunktion **main()** sowie jedes weitere Quellmodul benötigt Header-Dateien zu den jeweils benutzten Klassen.

Bei dieser Gelegenheit sollten wir unser KnowHow für den Umgang mit Header-Dateien noch etwas verfeinern:

6.7.1 Doppeldefinitionen verhindern

In der folgenden Header-Datei **Aufgabe.h** zur Klasse `Aufgabe` wird demonstriert, wie mit den Preprozessor-Anweisungen **`#ifndef`**, **`#define`** und **`#endif`** verhindert wird, dass der Compiler auf doppelte Definitionen durch geschachtelte **#include**-Anweisungen trifft:

```
#ifndef __Aufgabe_h__
  #define __Aufgabe_h__

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

  enum Operation
  {
    add,
    sub,
    mult
  };

  class Aufgabe
  {
  public:
    Aufgabe(Operation typ);
    Aufgabe(Operation typ, int b1Z, int b1N, int b2Z, int b2N);
    ~Aufgabe();
    void   anzeigen();
    int    fragen();

  private:
    Bruch    b1, b2, lsg, antwort;
    Operation op;
    char   opchar;
    double   dlsg;
    void   init();
  };

#endif
```

Wenn die Header-Datei **Aufgabe.h** z.B. in zwei „höheren“ Header-Dateien **UebAufgabe.h** und **TestAufgabe.h** benötigt wird, die ihrerseits von einer anderen Programmeinheit **br.cpp** *beide* inkludiert werden, dann sieht der Compiler z.B. die Definition der Klasse `Aufgabe` doppelt. Mit

dem vorgeschlagenen Verfahren wird dafür gesorgt, dass nur das erste Auftreten berücksichtigt wird, weil sonst der Compiler protestiert, z.B.:

```
u:\eigene dateien\vcpp\include\Aufgabe.h(16) : error C2011:
  'Aufgabe' : 'class'-Typ-Neudefinition
```

Der vorgeschlagene Schutzmechanismus funktioniert folgendermaßen:

- Der Compiler sieht die Anweisungen zwischen **`#ifndef`** und **`#endif`** nur dann, wenn das Symbol `__Aufgabe_h__` im aktuellen Compiler-Lauf noch nicht definiert ist.
- Ist die Sequenz auf diese Art frei gegeben, wird in ihrer ersten Zeile schleunigst die fragliche Definition vorgenommen, damit keine weitere Freigabe erfolgen kann.
- Das Blockade-Symbol leitet man üblicherweise aus dem Namen der Header-Datei ab.

6.7.2 Generelle Verzeichnisse für Header-Dateien

Sobald eine Header-Datei potentiell in mehreren Projekten benötigt wird, sollte sie in einem generell vom Compiler berücksichtigten Include-Verzeichnis abgelegt werden. Dann muss die Datei nur einmal vorhanden sein (Wartungsaufwand!) und kann in den einzelnen Projekten bequem per Spitzklammern-Syntax (ohne Pfadangabe) inkludiert werden.
Ein generell zu berücksichtigendes Include-Verzeichnis muss der Entwicklungsumgebung über

Extras > Optionen > Verzeichnisse

bekannt gemacht werden.
Im Bruchrechnungs-Projekt wird diese Option allerdings nicht genutzt.

6.7.3 Zerlegung des Bruchrechnungs-Quellcodes

Gehen Sie nun folgendermaßen vor, um das Bruchrechnungs-Programm in handliche Teile zu zerlegen:

- Erstellen Sie die beiden Header-Dateien **Bruch.h** sowie **Aufgabe.h** im Projektordner über

  Datei > Neu > Dateien > C/C++-Header-Datei

  Inkludieren Sie in jeder Klassen-Header-Datei alle Standard- oder Projekt-Header-Dateien, die von irgendeinem Klassen-Element benötigt werden.
  Bauen Sie in jeder Header-Datei die oben beschriebene Sicherung gegen Doppel-Definitionen ein.
- Erstellen Sie die Implementationsdateien **Bruch.cpp**, **Aufgabe.cpp** sowie **br.cpp** (mit der Hauptfunktion) über

  Datei > Neu > Dateien > C++-Quellcodedatei

  im Projektordner. Inkludieren Sie in den Klassen-Implementierungsdateien jeweils die Klassen-Header-Datei. In der Datei **br.cpp** mit der Hauptfunktion muss die Header-Datei **Aufgabe.h** inkludiert werden.

Mit Hilfe des Arbeitsbereichsfensters lassen sich die zahlreichen Dateien dennoch bequem verwalten. So holt z.B. ein Doppelklick auf eine Klasse oder auf eine Elementvariable die zugehörige Header-Datei nach vorne, und per Doppelklick auf eine Elementfunktion kann man sofort die zugehörige Quelltextpassage erreichen.

Den aktuellen Zustand des Beispiel-Projektes finden Sie im Unterordner **Bruchrechnen\br1** zum Ordner **BspUeb\Konsolen-Anwendungen** mit den Beispielen und Übungsaufgaben zum Kurs (vollständige Ortsangaben: siehe Vorwort).

6.8 Klassenvariablen

Aus aktuellem Anlass drängt sich ein Thema in den Vordergrund, das eigentlich erst später behandelt worden wäre. Die Klasse `Bruch` soll so verbessert werden, dass der parameterfreie Konstruktor den Zähler und den Nenner mit Hilfe der Pseudozufallsfunktion **rand()** mit Werten versorgt, statt beide undefiniert zu lassen. Zur Initialisierung des Pseudozufallszahlengenerators muss einmalig die Funktion **srand()** ausgeführt werden, z.B.:

```
Bruch::Bruch()
{
   cout << "Standard-Konstruktor Bruch startet\n";
   if (!zginit)
   {
      srand(unsigned(time(NULL)));
      nenner = rand();
      zginit = true;
   }
   nenner = rand() % 9 + 1;
   zaehler = rand() % nenner + 1;
}
```

Die Initialisierung sollte von der `Bruch`-Klasse selbständig ausgeführt werden, damit sie nicht vergessen werden kann. Die in obigem Vorschlag verwendete **time()**-Funktion liefert leider nur nur ganze Sekunden, was zum Problem wird, wenn innerhalb einer Sekunde mehrere `Bruch`-Objekte erzeugt werden: In diesem Fall entstehen identische Brüche. Um dies zu verhindern, sollte nur das zuerst erzeugte `Bruch`-Objekt die Initialisierung ausführen. Dazu muss irgendwo gespeichert werden, ob bereits ein `Bruch`-Objekt existiert. In einem anderen Programm mag es sinnvoll sein, die Anzahl der bereits erzeugten Objekte einer Klasse verfügbar zu haben.
Es handelt sich jeweils um Informationen über eine Klasse, die für alle Objekte dieser Klasse verfügbar sein sollten. In C++ - Klassen können für diesen Zweck mit dem Schlüsselwort **static** so genannte **Klassenvariablen** deklariert werden.
Eine normale Elementvariable gehört zu einer bestimmten Instanz und ist (falls privat) nur für deren Methoden erreichbar. Ihre Existenz endet mit dem Ableben des Objektes. Eine Klassenvariable beginnt ihre *permanente* Existenz vor dem ersten Objekt einer Klasse und kann von allen Objekten der Klasse manipuliert werden. Sie hat also eine statische Lebensdauer und die zugehörige Klasse als Gültigkeitsbereich.

Im Beispiel wird in der boolschen Klassenvariablen `zginit` festgehalten, ob der Pseudozufallszahlengenerator bereits von einem `Bruch`-Objekt initialisiert worden ist. Die *Deklaration* erfolgt im Rahmen der Klassendefinition:

```
class Bruch
{
public:
   Bruch();
   Bruch(int z, int n);
   ~Bruch();
   void zeige();
   void sz(int z);
   void sn(int n);
   int  hz() {return zaehler;};
   int  hn() {return nenner;};
   void erweitere(int faktor);
   void addiere(Bruch &b1, Bruch &b2);
   void multipliziere(Bruch &b1, Bruch &b2);
private:
   int  zaehler;
   int  nenner;
   bool def;
   static bool zginit;
   static int nbrobj;
};
```

Hierbei handelt es sich tatsächlich nur um eine *Deklaration*, die durch eine *Definition* ergänzt werden muss, die außerhalb der Klassendefinition unterzubringen ist, z.B. am Anfang der Implementierungsdatei zur Klasse:

```
// Implementation der Methoden von Bruch

#include <Bruch.h>

//Klassenvariablen definieren und initialisieren:
int Bruch::nbrobj = 0;
bool Bruch::zginit = false;
 .  .  .
.  .  .
```

Im Kontext der Definition kann auch Initialisierung von Klassenvariablen vorgenommen werden. Diese wird vor dem Erzeugen des ersten Objektes ausgeführt.

Im Beispiel wird zu Demonstrationszwecken auch eine Klassenvariable `nbrobj` definiert und deklariert, welche über die augenblickliche Anzahl der `Bruch`-Objekte informieren soll. Setzt man in die Konstruktoren und Destruktoren der Klasse entsprechende Aktualisierungen und Meldungen ein, wird die aktuelle „Klassenstärke“ im Programmablauf sichtbar, z.B.:

```
Konstruktor Bruch startet: Wir sind 1
Konstruktor Bruch startet: Wir sind 2
Standard-Konstruktor Bruch startet: Wir sind 3
Standard-Konstruktor Bruch startet: Wir sind 4
Konstruktor Aufgabe startet

Berechne bitte:

  1               2
------     +     -----
  2               5

Welchen Zaehler hat dein Ergebnis:     9
                                     ------
Welchen Nenner hat dein Ergebnis:      10

Destruktor Aufgabe startet
Destruktor Bruch startet: Wir sind noch 3
Destruktor Bruch startet: Wir sind noch 2
Destruktor Bruch startet: Wir sind noch 1
Destruktor Bruch startet: Wir sind noch 0
```

Wenn Sie in der Klassenansicht des Arbeitsbereichsfensters das Kontextmenü zu einer Klassenvariablen (z.B. `zginit`) öffnen, finden Sie dort die beiden Optionen:

- **Gehe zu Definition**
- **Gehe zu Deklaration**

Die beiden Optionen führen zur jeweils zugehörigen Stelle im Quellcode. Wie man sieht, ist die in diesem Manuskript gelegentlich betonte Unterscheidung zwischen *Definition* und *Deklaration* doch nicht rein akademisch.

6.9 Vererbung

Das zu entwickelnde Bruchrechnungs-Trainingsprogramm soll sowohl einen Übungs- als auch einen Prüfungsteil besitzen, die jeweils Aufgaben vorlegen, auf die Antworten aber unterschiedlich reagieren. Unsere Klasse `Aufgabe` verfügt nicht über die gewünschten Dialogqualitäten, kann aber Bruchrechnungsaufgaben vorlegen (Methode `anzeigen()`) und nach der Lösung fragen (Methode `fragen()`). Genau in einer solchen Situation kommen die versprochenen OOP-Vorteile *Erweiterbarkeit* bzw. *Wiederverwendbarkeit* durch die Vererbungstechnologie zum Tragen. Wir stellen uns zunächst die Aufgabe, eine Klasse `UebAufgabe` zu entwerfen, die Bruchrechnungsaufgaben vorlegen und bei Fehlern didaktische Hilfen geben kann. Offenbar stellt die vorhandene Klasse `Aufgabe` eine gute Ausgangsbasis dar, doch soll ihr Code nicht per *Cut & Paste* genutzt werden, sondern auf rationelle und flexible Weise. Wir definieren die neue Klasse als Nachkomme der vorhandenen Klasse im Sinne einer Spezialisierung bzw. Erweiterung, so dass Objekte der Klasse `UebAufgabe` alle Daten und Methoden der Klasse `Aufgabe` besitzen, darüber hinaus aber für ihren speziellen Einsatzweck ausgerüstet werden. Generell kann eine abgeleitete Klasse ...

- zusätzliche Daten (Elementvariablen) erhalten
- zusätzliche Methoden (Elementfunktionen) erhalten
- geerbte Methoden überschreiben, d.h. unter Beibehaltung des Namens umgestalten

Weil die neue Klasse wesentliche Elemente von der Klasse `Aufgabe` übernimmt, fällt die Definition (hier im Kontext der kompletten Header-Datei **UebAufgabe.h**) relativ kurz aus:

```
#ifndef __UebAufgabe_h__
  #define __UebAufgabe_h__

  #include <iostream>
  using namespace std;
  #include <Aufgabe.h>

  class UebAufgabe : public Aufgabe
  {
  public:
    UebAufgabe(Operation typ);
    UebAufgabe(Operation typ, int b1Z, int b1N, int b2Z, int b2N);
    void dialog(int mxwdhlng = 0);
    void fragen();
  private:
    int nwdhlng;
  };
#endif
```

Das Erbschaftsverhältnis kommt in der **class**-Zeile zum Ausdruck, wobei später erklärt werden soll, welche Bedeutung das Schlüsselwort **public** an dieser Stelle hat.
Für die didaktische Handlungskompetenz der Klasse `UebAufgabe` soll im Wesentlichen die neue Methode `dialog()` sorgen:

```
void UebAufgabe::dialog(int mxwdhlng)
{
  int i = 0;

  nwdhlng = 0;
  Aufgabe::fragen();
  if (double(antwort.hz())/antwort.hn() == hdlsg())
  {
    cout << "\nBravo!\n\n";
    serg(1);
  }
```

```
   else
   {
     nwdhlng++;
     cout << "\nFehler!\n";
     switch (op)
     {
       case add: cout << " …
                 break;
       case mult:cout << " …
                 break;
     }
     while (i++ < mxwdhlng && herg() == 0)
     {
       cout << "Versuche es noch mal!\n";
       fragen();
       if (double(antwort.hz())/antwort.hn() == hdlsg())
       {
         cout << "\nGut!\n\n";
         serg(1);
       }
       else
         cout << "\nImmer noch falsch! ";
     }
     if (!herg())
     {
       cout << "Das korrekte Ergebnis ist:\n";
       lsg.zeige();
     }
   }
}
```

In dieser Methode wird von einigen Erweiterungen der Klasse `Aufgabe` Gebrauch gemacht, von denen Altbenutzer der Klasse nicht negativ tangiert werden. Hinzugekommen sind:

- Eine private Elementvariable zur Aufnahme der numerischen Lösung:
  ```
  double dlsg;
  ```
  Beim Erzeugen eines `Aufgabe`-Objektes wird `dlsg` auf den korrekten Wert gesetzt.
- Eine Methode `hdlsg()`, die den aktuellen `dlsg`-Wert liefert:
  ```
  double hdlsg() {return dlsg;}
  ```
- Eine private Elementvariable, um den Status eines Lösungsversuchs zu speichern:
  ```
  bool erg;
  ```
  Beim Erzeugen eines `Aufgabe`-Objektes wird `erg` auf den Wert **false** gesetzt.
- Eine Methode `serg()`, die den `erg`-Wert setzt:
  ```
  void serg(bool perg) {perg?erg=1:erg=0;}
  ```
- Eine Methode `herg()`, die den aktuellen `erg`-Wert liefert:
  ```
  bool herg() {return erg;}
  ```

Die Elementvariable `dlsg` wird zur Prüfung der vom Schüler vorgeschlagenen Lösung benötigt. Weil diese Prüfung auf dem numerischen Wert eines Bruches basiert, muss die Antwort nicht perfekt gekürzt sein muss, um als korrekt bewertet zu werden.

6.9.1 Überschreiben von Methoden

Unterstützt wird die Methode `dialog()` durch eine mit pädagogischen Ehrgeiz überarbeitete Variante der `Aufgabe`-Methode `fragen()`:

```
void UebAufgabe::fragen()
{
  if (nwdhlng == 0) cout << "\nEs folgt eine '" << opchar << "'-Aufgabe.\n";
  Aufgabe::fragen();
}
```

In `UebAufgabe`-Objekten wird die Methode `Aufgabe::fragen()` **überschrieben** durch `UebAufgabe::fragen()`.
Speziell in der überschreibenden Methode der abgeleiteten Klasse wird es oft von Nutzen sein, auf die überschriebene Methode der Basisklasse zuzugreifen, was durch Voranstellen des Klassenbezeichners samt Namensraumauflösungsoperator ohne weiteres möglich ist (siehe Beispiel).

Die Tatsache, dass `Aufgabe`-Objekte auf die Nachricht `fragen()` anders reagieren als `UebAufgabe`-Objekte, kann man als *Polymorphie* bezeichnen, wenngleich viele Autoren diesen Begriff in einem engeren Sinn verwenden (siehe unten).

6.9.2 Schutzstufe *protected*

In der Methode `fragen()` der abgeleiteten Klasse `UebAufgabe` wird auf die Variable `opchar` der Basisklasse `Aufgabe` zugegriffen, was bei einer privaten `Aufgabe`-Elementvariablen *nicht* möglich wäre. Hier hilft die bisher noch nicht erwähnte Schutzstufe **protected** weiter:

- Objekte abgeleiteter Klassen dürfen ebenso zugreifen wie die Objekte der Klasse selbst.
- Bei fremden Programmbestandteilen ändert sich nichts gegenüber der Schutzstufe **private**.

Aus analogen Gründen müssen in der `Aufgabe`-Klassendefinition noch weitere Variablen als **protected** definiert werden:

```
class Aufgabe
{
public:
  Aufgabe(Operation typ);
  Aufgabe(Operation typ, int b1Z, int b1N, int b2Z, int b2N);
  ~Aufgabe();
  void      anzeigen();
  int       fragen();
  double    hdlsg() {return dlsg;}
  void      serg(bool perg) {perg?erg=1:erg=0;}
  bool      herg() {return erg;}
protected:
  Bruch     lsg, antwort;
  Operation op;
  char      opchar;
private:
  Bruch     b1, b2;
  void      init();
  bool      erg;
  double    dlsg;
};
```

6.9.3 Vererbungsmodi

In der Definition einer abgeleiteten Klasse folgt auf ihren Namen nach einem Doppelpunkt der *Vererbungsmodus*, welcher einige Kontrolle über die Rechte der geerbten Elemente in der neuen Klasse erlaubt. Meist verwendet man (wie in unserem Beispiel) den Modus **public**:

```
class UebAufgabe : public Aufgabe
```

Auch die anderen Vererbungsmodi (**protected**, **private**) tragen Namen, die auch zur Kennzeichnung von Schutzstufen dienen. Über die Wirkung der Vererbungsmodi informiert die folgende Tabelle (nach Hyman & Arnson 1999, S. 307):

Vererbungsmodus	Element ist in der Basisklasse	Element ist in der abgeleiteten Klasse
public	public	public
	protected	protected
	private	unzugänglich
protected	public	protected
	protected	protected
	private	unzugänglich
private	public	private
	protected	private
	private	unzugänglich

Ist ein Element in der Basisklasse private, dann können weder die Methoden der abgeleiteten Klasse noch Methoden sonstiger Klassen darauf zugreifen.

6.9.4 Initialisierungslisten

Sowohl zusammengesetzte Klassen (z.B. `Aufgabe`) wie auch abgeleitete Klassen (z.B.: `UebAufgabe`) haben das Problem, in ihrem Konstruktor eventuell Initialisierungswerte an Konstruktoren von Member-Objekten oder von Basisklassen durchreichen zu müssen. In beiden Fällen wird das Problem über Initialisierungslisten gelöst. Im folgenden `UebAufgabe`-Konstruktor werden Zähler und Nenner für die beiden Argumente einer Übungsaufgabe zunächst an den Konstruktor der Basisklasse `Aufgabe` weitereicht, der sie seinerseits an die Konstruktoren der Member-Objekte aus der `bruch`-Klasse übergibt (siehe oben):

```
UebAufgabe::UebAufgabe(Operation typ, int b1Z, int b1N, int b2Z, int b2N) :
                          Aufgabe(typ, b1Z, b1N, b2Z, b2N)
{
}
```

6.9.5 Objekte als Feldelemente

Wir erzeugen in der Hauptfunktion **main()** einige Objekte aus der Klasse `UebAufgabe` und erteilen ihnen über die Methode `dialog()` einen Lehrauftrag:

Quellcode	Ausgabe

`#include "UebAufgabe.h"`
 	
`void main()`
`{`
`  UebAufgabe ua[3] =`
`  {UebAufgabe(add, 1, 2, 3, 4),`
`  UebAufgabe(add, 2, 3, 3, 5),`
`  UebAufgabe(mult, 1, 4, 3, 7)};`
`  for (int i = 0; i <= 2; i++)`
`  ua[i].dialog(1);`
`}`
	`Es folgt eine '+'-Aufgabe.`
	`Berechne bitte:`
	`  1  3`
	`------  +  -----`
	`  2  4`
	`Welchen Zaehler hat dein Ergebnis:  5`
	`  ------`
	`Welchen Nenner hat dein Ergebnis:  4`
	`Bravo!`
	
	`Es folgt eine '+'-Aufgabe.`
	`Berechne bitte:`
	`  2  3`
	`------  +  -----`
	`  3  5`
	`Welchen Zaehler hat dein Ergebnis:  19`
	`  ------`
	`Welchen Nenner hat dein Ergebnis:  15`
	`Bravo!`
	
	`Es folgt eine '*'-Aufgabe.`
	`Berechne bitte:`
	`  1  3`
	`------  *  -----`
	`  4  7`
	`Welchen Zaehler hat dein Ergebnis:  3`
	`  ------`
	`Welchen Nenner hat dein Ergebnis:  28`
	`Bravo!`

Im Beispiel ist vor allem das eindimensionale Feld `ua` mit 3 Elementen vom Typ `UebAufgabe` von Interesse. Da es sich bei jedem Feldelement um ein Objekt handelt, muss ein Konstruktor ausgeführt werden. Weil `UebAufgabe` keinen parameterfreien Standardkonstruktor besitzt, muss ein verfügbarer Konstruktor explizit angegeben und mit den erforderlichen Aktualparametern versorgt werden.
Die Klasse `UebAufgabe` bietet zwei Konstruktoren, wobei wir uns auf die Wahl einer Operation beschränken oder eine Übungsaufgabe komplett festlegen können. Im Beispiel wird mit der zweiten Konstruktorv-Variante eine Übungssequenz genau festgelegt, anstatt den Zufall mitwirken zu lassen.

Die Konstruktorenaufrufe werden im Rahmen der `ua`-Definition in einer geschweift eingeklammerte Liste untergebracht, wie Sie es analog auch für Felder mit Standardelementen kennen gelernt haben.

Bei Wahl des einfacheren `UebAufgabe`-Konstruktors würde die `ua`-Definition folgendermaßen aussehen:

```
UebAufgabe ua[3] = {add, add, mult};
```

Sie folgt demselben Prinzip, macht aber von einer Abkürzungsmöglichkeit Gebrauch: Bei einparametrigen Konstruktoren darf an Stelle des vollständigen Aufrufs auch der eine Aktualparameter stehen.

Die Zahl der angebotenen Konstruktorenaufrufe darf kleiner sein als die Zahl der zu initialisierenden Feldelemente, wenn ein Standardkonstruktor (ohne Parameter) zur Verfügung steht, der sich um die restlichen Feldelemente kümmern kann.

6.9.6 Stand des Bruchrechnungs-Projektes und Entwicklungsmöglichkeiten

Auf dem noch langen Weg zu einem ausgereiften, objektorientierten Bruchrechnungs-Trainingsprogramm sollen noch einige wenige Schritte zurückgelegt werden. Im zu modellierenden Anwendungsbereich sind nicht nur Übungsaufgaben, sondern auch Testaufgaben gefragt. Daher definieren wir eine Klasse `TestAufgabe`, die wie die Klasse `UebAufgabe` aus der Klasse `Aufgabe` abgeleitet. Auch für Testaufgaben wird eine Methode `dialog()` entworfen, die sich aber weniger durch didaktische Einfühlung als durch diagnostische Präzision auszeichnet. Für eine vorgegebene Bruchrechnungsaufgabe wird festgehalten, ob sie richtig gelöst werden konnte, und welche Zeit damit verbracht wurde. In der Klassendefinition findet sich daher eine (private) Variable `zeit` vom Typ **unsigned long int** und eine zugehörige Zugriffsmethode `hzeit()`:

```
class TestAufgabe : public Aufgabe
{
public:
   TestAufgabe(Operation typ);
   void dialog();
   int hzeit() {return zeit;};
private:
   unsigned long int zeit;
};
```

Als Datentyp für die Zeitvariable ist **unsigned long int** angegeben, obwohl in VC++ 6.0 **unsigned int** und **unsigned long int** die selbe Größe (4 Bytes) und den selben Wertebereich besitzen (Maximalwert: 4294967295 = 0xffffffff). Manche Compiler reservieren aber für **unsigned int**-Variablen nur 2 Bytes, woraus sich ein zu kleiner Wertebereich ergibt. Im Sinne möglichst hoher Portabilität schreiben wir also besser **unsigned long int**.
In der Methode `dialog()` wird zur Zeitmessung dieselbe Funktion **time()** verwendet, mit der wir schon im `bruch`-Standard-Konstruktor den Pseudozufallszahlengenerator initialisiert haben. Die Ergebnisprüfung basiert wie in `UebAufgabe::dialog()` auf dem numerischen Wert des Bruches:

```
void TestAufgabe::dialog()
{
   zeit = time(NULL);
   fragen();
   zeit = time(NULL) - zeit;
   if (double(antwort.hz())/antwort.hn() == hdlsg())
   {
      cout << "\nBravo!\n\n";
      serg(1);
   }
   else
      cout << "\nLeider falsch!\n";
}
```

In der Hauptfunktion des Bruchrechnungs-Trainingsprogramms kann nun mit den beiden Klassen `UebAufgabe` und `TestAufgabe` ein halbwegs praktikables Verhalten realisiert werden:

```
#include "UebAufgabe.h"
#include "TestAufgabe.h"

void ueben();
void testen();
void uebericht(UebAufgabe ua[]);
void tbericht(TestAufgabe ua[]);
```

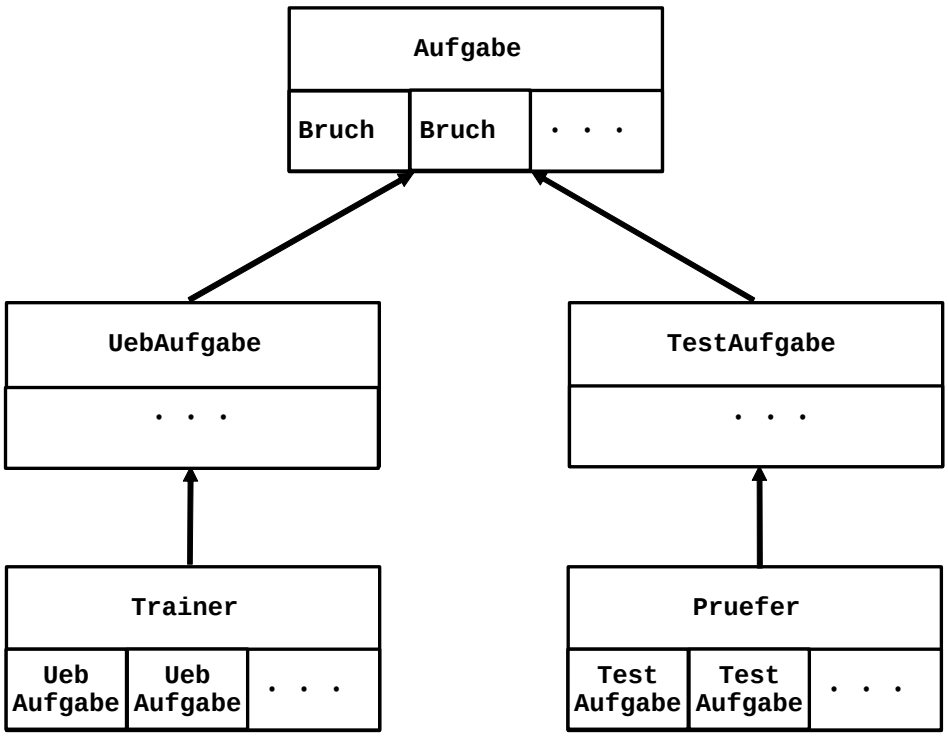
```
void main()
{
  char antwort;

  cout << "Willst du ueben (u) oder testen (t)? ";
  cin >> antwort;
  if (antwort == 'u')
    ueben();
  else
    testen();

}
 .  .  .
 .  .  .
```

Den aktuellen Zustand des Beispiel-Projektes finden Sie im Unterordner **Bruchrechnen\br4** zum Ordner **BspUeb\Konsolen-Anwendungen** mit den Beispielen und Übungsaufgaben zum Kurs (vollständige Ortsangaben: siehe Vorwort).

Bis zu einem „marktfähigen“ Zustand des Programms ist noch einige Arbeit zu leisten. Die vorhandenen Klassen benötigen noch einige Erweiterungen und Verbesserungen. Außerdem erscheint es sinnvoll, die zahlreichen Funktionen, die in der Hauptfunktion benötigt werden, durch neue Klassen zu ersetzen, um die Wiederverwendbarkeit des Codes zu verbessern:

- Eine Klasse `Trainer` könnte unter Verwendung mehrerer Objekte der Klasse `UebAufgabe` selbständige Übungssequenzen durchführen.
- Analog könnte eine Klasse `Pruefer` entworfen werden.
- Denkbar ist auch eine Klasse `Schueler`.

Unter Berücksichtigung der letzten Vorschläge resultiert für das Bruchrechnungsprojekt die folgende Klassenarchitektur:



In dieser informellen Darstellung bedeuten die Pfeile entweder „ist Nachkomme von“ oder „verwendet als Elemente“.

Aufgrund des objektorientierten Programmierstils können bei der Weiterentwicklung unserer Anwendung verschiedene Entwickler einbezogen werden, die sich z.B. jeweils um eine Klasse kümmern.

Am Ende der Ausführungen zur Vererbung soll noch auf eine Besonderheit der C++ - Klassenarchitektur hingewiesen werden. In vielen OOP-Sprachen (z.B. Smalltalk, Java) stammen alle Klassen von einer Urahn-Klasse (oft als **Object** bezeichnet) ab, so dass eine „single-tree"-Hierarchie (Eckel 2000, Band 1, S. 728) vorliegt. In C++ existiert keine Urahn-Klasse, so dass mehrere, unverbundene Stammbäume entstehen können.

6.10 Klassen-Bibliotheken erstellen

Wenn Sie Klassen definieren, die in mehreren Programmen benutzt werden sollen, dann empfiehlt sich die Erstellung einer Bibliothek. Im Abschnitt über Strukturierung und Modularisierung wurde das Erstellen einer *Funktions*-Bibliothek beschrieben. Nun folgt die objektorientierte Variante. Als Beispiel erstellen wir die Bibliothek mit unseren Klassen zur Bruchrechnungs-Didaktik:

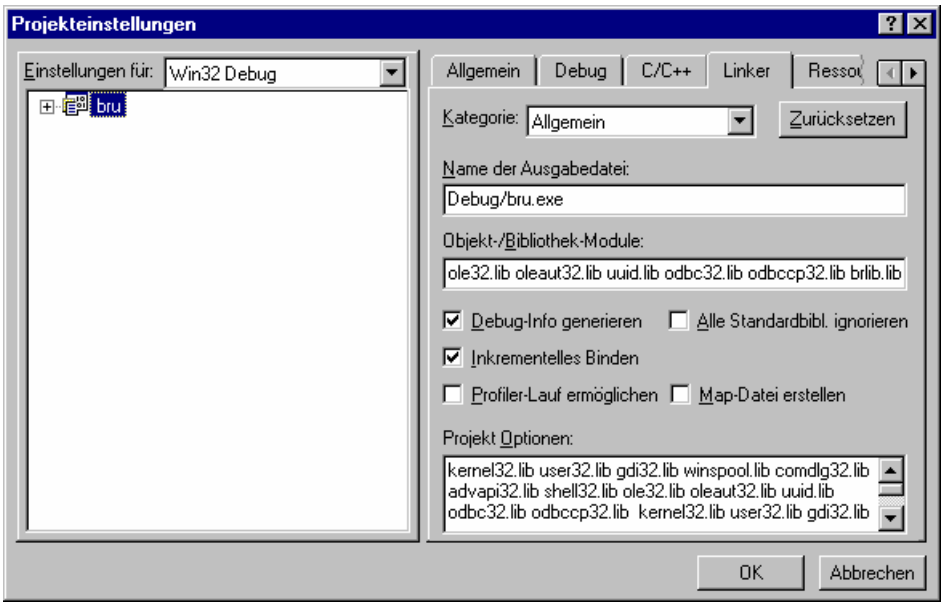
- Öffnen Sie in der Entwicklungsumgebung ein neues Projekt über **Datei > Neu > Projekt**.
- Wählen Sie den Typ **Win32-Bibliothek (statische)** und den gewünschten **Projektnamen**.
- Lassen Sie in der Dialogbox zum ersten Assistentenschritt das Projekt **fertigstellen**, und quittieren Sie die Informationen zum neuen Projekt.
- Über **Projekt > Dem Projekt hinzufügen > Dateien** werden unsere bereits vorhandenen Quellcodedateien unverändert einbezogen. Wenn sich die Header-Dateien in einem generell berücksichtigten Include-Verzeichnis befinden, brauchen die **#include**-Zeilen in den Quellcodedateien nicht geändert zu werden. Anderenfalls müssen Sie entweder die **#include**-Zeilen ändern, oder die Header-Dateien in den Ordner des Bibliotheks-Projekts kopieren.
- Lassen Sie die Bibliothek mit **<F7>** erstellen.
- Kopieren Sie die **lib**-Datei, die per Voreinstellung im Debug-Unterverzeichnis des Projektordners entsteht, in ein generell berücksichtigtes Bibliotheks-Verzeichnis.

Um zu demonstrieren, wie einfach Bruchrechnungs-Klassen in neue Anwendungen einzubinden sind, erstellen wir eine neue Win32-Konsolenanwendung, fügen eine Quelldatei ein und schreiben dort z.B. das folgende Programm, das vier Objekte aus der Klasse `UebAufgabe` für eine Trainingssitzung engagiert:

```
#include "UebAufgabe.h"

void main()
{
  UebAufgabe ua[4] = {add, mult, add, mult};

  for (int i = 0; i <= 3; i++)
  {
    ua[i].dialog(1);
  }
}
```

Zu den Bruchrechnungs-Header-Dateien gelten für das aktuelle Projekt dieselben Hinweise, die eben für das Bibliotheksprojekt formuliert wurden. Vermutlich werden sie sich bald dafür entscheiden, die Header-Dateien doch in einem allgemein berücksichtigten Verzeichnis zu stationieren.

Damit das Programm gebunden werden kann, muss die neue Bruchrechnungs-Bibliotheksdatei nach **Projekt > Einstellungen > Linker** noch in der Liste der **Objekt-/Bibliothek-Module** ergänzt werden, z.B.:



6.11 Virtuelle Funktionen und Polymorphie

Wenn aus einer Basisklasse (z.B. `Flaeche`) mehrere spezielle Klassen abgeleitet wurden (z.B. `Rechteck`, `Kreis`), ist es oft von Vorteil, eine Serie von Objekten, die aus einer beliebigen Spezialklasse stammen dürfen, gemeinsam zu verwalten, z.B. in einem Feld mit Pointern auf Basisklassenobjekte:

- Man legt ein Feld mit Pointern auf *Basisklassenobjekte* an.
- Je nach Aufgabenstellung werden Objekte aus verschiedenen Spezialklassen dynamisch erzeugt.
- Deren Adressen dürfen in Basisklassen-Pointervariablen abgelegt werden.

6.11.1 Frühe Bindung

In diesem Abschnitt verwenden wir zur Demonstration ein übersichtliches Beispiel ohne Anspruch auf Praxistauglichkeit. Aus der Basisklasse `Flaeche` werden die beiden Spezialklassen `Rechteck` und `Kreis` abgeleitet:

```
class Flaeche
{
public:
   double ma() {return a;};
   void geta() {a = 0.0;}
protected:
   double a;
};

class Rechteck : public Flaeche
{
public:
   void geta();
private:
   double x, y;
};

class Kreis: public Flaeche
{
public:
   void geta();
private:
   double r;
};
```

Die Methode **geta()** soll den Benutzer nach passenden Flächenparametern fragen (Seitenlängen beim Rechteck, Radius beim Kreis) und dann den Flächeninhalt berechnen[1]:

```
void Rechteck::geta()
{
  cout << "Erste Seitenlaenge: ";  cin >> x;
  cout << "Zweite Seitenlaenge: "; cin >> y;
  a = x*y;
}

void Kreis::geta()
{
  cout << "Radius: ";  cin >> r;
  a = acos(-1)*r*r;
}
```

Bei einer Fläche unbekannter Form ist dies nicht möglich, so dass die Methode in der Basisklasse provisorisch definiert wird.
In **main()** wird ein Feld `fla` mit 2 Pointern auf Objekte der Basisklasse `Flaeche` definiert. Für jedes `fla`-Element wird der Benutzer zunächst nach einer Flächenform gefragt. Dann wird mit **new** ein Objekt der passenden Spezialklasse erzeugt und seine Adresse dem aktuellen `fla`–Element zugewiesen:

```
void main()
{
  const int anz = 2;
  Flaeche *fla[anz];
  double gesa = 0.0;
  char antwort;

  for (int i=0; i<anz; i++)
  {
    cout << "\nRechteck (r) oder Kreis (k)?";
    cin >> antwort;
    switch (antwort)
    {
      case 'r': fla[i] = new Rechteck; break;
      case 'k': fla[i] = new Kreis; break;
    }
    fla[i]->geta();
    gesa += fla[i]->ma();
  }
  cout << "\nDie GesamtFlaeche betraegt: " << gesa << endl;
}
```

Anschließend wird für jedes Feldelement die Methode `geta()` ausgeführt, wobei aber bedauerlicherweise nicht die Methode der Spezialklasse, sondern diejenige der Basisklasse zum Einsatz kommt:

```
Rechteck (r) oder Kreis (k)?r

Rechteck (r) oder Kreis (k)?k

Die GesamtFlaeche betraegt: 0
```

Offenbar legt die Typangabe `Flaeche` in der `fla`-Variablendefinition die auszuführende Methode unabhängig vom tatsächlichen Typ des dynamisch erzeugten Objektes fest. Weil die auszuführende Methode bereits zur Übersetzungszeit fixiert wird, spricht man hier von einer **frühen Bindung**.

---

[1] In `Kreis::geta()` wird die Kreiszahl π mit Hilfe der Kosinus-Umkehrfunktion Arcus Kosinus ermittelt, die in der C++ - Standardbibliothek durch die Funktion **acos()** berechnet wird. Es gilt:

$$\cos(\pi) = -1, \text{ also } \cos^{-1}(-1) = \pi.$$

6.11.2 Späte Bindung

Die Möglichkeit, Objekte aus verschiedenen Spezialklassen über ein Feld mit Basisklassen-Objekten gemeinsam ansprechen zu können, ist allerdings nur dann so richtig nützlich, wenn die Spezialobjekte ihre Erweiterungen bzw. speziellen Verhaltensmerkmale *behalten*, wenn sie einem Basisklassen-Objekt zugewiesen werden.
Um das Problem zu lösen, muss in obigem Quellcode nur ein einziges Wort ergänzt werden. In C++ lässt sich für eine Methode die Priorität von Spezialklassen-Varianten festlegen, indem sie in der Basisklasse als **virtuell** deklariert wird:

```
class Flaeche
{
public:
   double ma() {return a;};
   virtual void geta() {a = 0.0;}
protected:
   double a;
};
```

Mit `fla[i]->geta()` wird nun für jedes Feldelement die Methode der dynamisch festgelegten Spezialklasse ausgeführt, z.B.:

```
Rechteck (r) oder Kreis (k)?r
Erste Seitenlaenge: 2
Zweite Seitenlaenge: 5

Rechteck (r) oder Kreis (k)?k
Radius: 1

Die Gesamtflaeche betraegt: 13.1416
```

Weil erst zur Laufzeit entschieden wird, welche Methode mit `fla[i]->geta()` tatsächlich aufgerufen wird, spricht man hier von einer **späten Bindung**.
Man könnte auch von einer **Polymorphie zur Laufzeit** sprechen. Viele OOP-Autoren verwenden den Begriff **Polymorphie** generell nur für Methodenaufrufe mit später Bindung.
Nach der Auffassung des C++ - Designers Bjarne Stroustrup (1992) kann man von einer Programmiersprache nur dann behaupten, sie unterstütze das objektorientierte Programmieren, wenn folgende Bedingungen erfüllt sind:

- Abstraktion durch Klassen und Objekte
- Vererbung
- Polymorphie zur Laufzeit

Leider bedingt die Pylomorphie einen beträchtlichen Zeit- und Platzaufwand im Zusammenhang mit den so genannten **vtables**, so dass für zeitkritische Bereiche verschiedene Ersatz-Techniken entwickelt wurden. Wir lernen wichtige Beispiele kennen im Zusammenhang mit der Standard Template Library (STL), die einen wesentlichen Teil der C++ - Standardbibliothek bildet, und bei der Behandlung der Microsoft Foundation Classes (MFC), die eine objektorientierte Windows-Programmierung erlauben.

6.11.3 Abstrakte Klassen

In der Basisklasse `Flaeche` kann die Methode **geta()**, die den Benutzer nach passenden Flächenparametern fragen (Seitenlängen beim Rechteck, Radius beim Kreis) und dann den Flächeninhalt berechnen soll, eigentlich nicht sinnvoll definiert werden, weil die Form der Fläche nicht bekannt ist. In einer solchen Situation kann man in der Basisklasse auf eine Implementation verzichten und die betroffene Methode durch den Zusatz „=0“ als **rein virtuell** deklarieren, z.B.:

```
class Flaeche
{
public:
   double ma() {return a;};
   virtual void geta() = 0;
protected:
   double a;
};
```

Dann liegt eine so genannte **abstrakte Klasse** vor, zu der keine Instanzen erzeugt werden kann. Das Ableiten von Spezialklassen ist aber möglich, wobei dort die virtuellen Methoden implementiert werden müssen, wenn nicht erneut eine abstrakte Klasse entstehen soll.

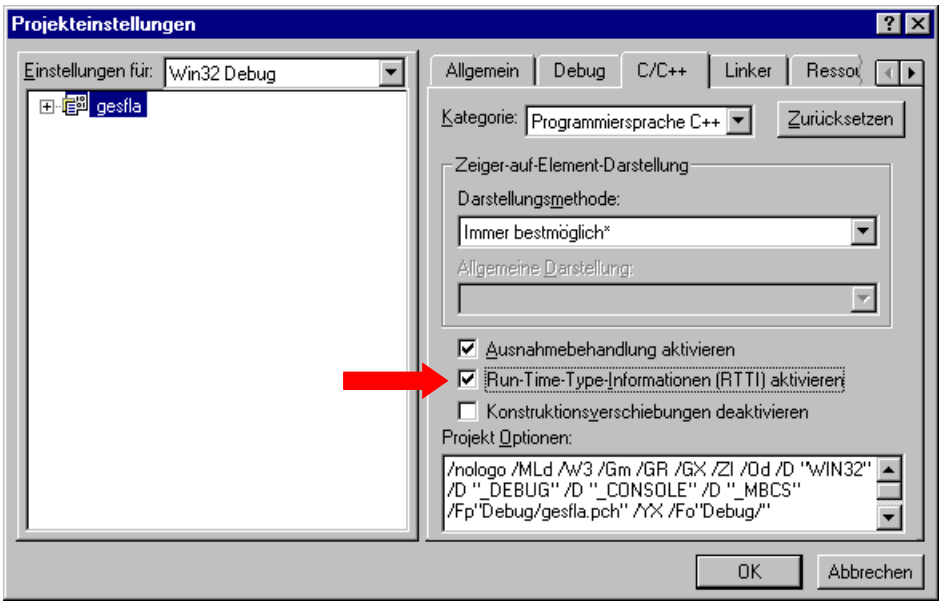
Für den in diesem Abschnitt beschriebenen Nutzen virtueller Funktionen spielt es *keine* Rolle, ob die Basisklasse abstrakt ist oder nicht.

6.11.4 Typ-Identifikation zur Laufzeit (RTTI)

Beim Einsatz *polymorpher Klassen* (mit virtuellen Methoden) ist das Wissen um die exakte Klassenzugehörigkeit von Objekten in der Regel irrelevant, weil diese dank Polymorphie ein „artgerechtes“ Verhalten zeigen. Doch gelegentlich ist es hilfreich, die exakte Klassenzugehörigkeit eines Objektes in Erfahrung zu bringen, zu dem man lediglich eine Basisklassen-Referenz besitzt. Für solche Zwecke wurde in C++ die **Run-Time Type Information (RTTI)** aufgenommen, nachdem mehrere Hersteller von Klassenbibliotheken proprietäre Lösungen entwickelt hatten. Der Mechanismus arbeitet allerdings nur für polymorphe Klassen, weil er die hier vorhandenen Tabellen der virtuellen Methoden benötigt (*vtabs*) benötigt.
Man übergibt dem Operator **typeid()** als Argument einen Pointer auf das zu klassifizierende Objekt und erhält eine Referenz auf ein globales Objekt vom Typ **typeinfo**. Dieses liefert mit seiner Methode **name()** der gewünschte Information. Außerdem können verschiedene **typeinfo**-Objekte per Identitätsoperator verglichen werden.
Um mit dem Operator **typeid()** Erfahrungen zu sammeln, ersetzen wir im Flächen-Beispielprogramm die Hauptfunktion **main()** durch folgende Variante:

```
void main()
{
   Flaeche* fla;
   char antwort;

   while (1)
   {
      cout << "\nRechteck (r) oder Kreis (k) oder Stop (s)?";
      cin >> antwort;
      switch (antwort)
      {
         case 'r': fla = new Rechteck; break;
         case 'k': fla = new Kreis; break;
         case 's': exit(0);
      }
      cout << typeid(*fla).name() << endl;
   }
}
```

Im Visual Studio 6.0 muss allerdings (nach **Projekt > Einstellungen > Kategorie = Programmiersprache C++**) noch eine Projektoption eingestellt werden, damit der Compiler den für RTTI erforderlichen Zusatzcode erzeugt:



Nun liefert die **typeinfo**-Methode **name()** die gewünschten Auskünfte:

```
Rechteck (r) oder Kreis (k) oder Stop (s)?r
class Rechteck

Rechteck (r) oder Kreis (k) oder Stop (s)?k
class Kreis

Rechteck (r) oder Kreis (k) oder Stop (s)?s
```

Ersetzt man in der in der Klassendefinition von `Flaeche` die Methodendekalaration

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

durch

```
void geta() {a = 0.0;}
```

dann geht die Polymorphie verloren. Weil keine vtabs mit Laufzeit-Typinformationen mehr zur Verfügung, liefert **name()** unbrauchbare Informationen (die Basisklasse):

```
Rechteck (r) oder Kreis (k) oder Stop (s)?r
class Flaeche

Rechteck (r) oder Kreis (k) oder Stop (s)?k
class Flaeche

Rechteck (r) oder Kreis (k) oder Stop (s)?s
```

6.12 Sonstige OOP-Themen

6.12.1 Konstante Methoden

Wird eine Methode als **const** definiert, so kann sie keine Elementvariablen des angesprochenen Objekts ändern. Im Bruchrechnungsbeispiel (siehe Abschnitt 6.2.2) sind die Methoden zum Auslesen von Elementvariablen geeignete Kandidaten für diese Vorsichtsmaßnahme, z.B.:

```
int  hz() const {return zaehler;};
```

Das Schlüsselwort **const** wird in der Deklaration sowie in der Definition (Implementation) hinter die Parameterliste gesetzt.

Durch diesen Zusatz ändert der in jeder Methode verfügbare **this**-Zeiger auf das aktuelle Objekt (siehe Abschnitt 6.12.4) seinen Typ von:

konstanter Zeiger auf ein Objekt der Klasse

auf:

konstanter Zeiger auf ein *konstantes* Objekt der Klasse

Wenn in der `Bruch`-Klasse die Methoden `hz()` und `hn()` zusichern, alle Elemente unangetastet zu lassen, kann z.B. in der Definition der Methode `multipliziere()` den Referenzparametern Zugriffsschutz zugesichert werden, ohne Proteste des Compilers auszulösen:

```
void Bruch::multipliziere(const Bruch& b1, const Bruch& b2)
{
  zaehler = b1.hz() * b2.hz();
  nenner  = b1.hn() * b2.hn();
}
```

Konstante Referenzparameter akzeptiert der Compiler nur dann, wenn lediglich konstante Methoden darauf angewendet werden.

6.12.2 Operatoren überladen

Nicht nur Funktionen und Methoden können in C++ überladen werden, sondern auch die *Operatoren* (+ -* / << etc.), was übrigens in der Klasse **ostream_withassign** mit dem Links-Shift-Operator << geschehen ist, so dass wir mit dem **ostream_withassign**-Objekt **cout** recht bequem arbeiten können. Generell geht es beim Überladen von Operatoren um bequemere Lösungen für Aufgaben, die ansonsten einen Funktions- bzw. Methodenaufruf erfordern würden. Statt die Argumente in einer Aktualparameterliste anzugeben, können sie bei reduziertem Syntaxaufwand um ein Operatorzeichen gruppiert werden, das bisher nur für andere Aufgaben zuständig war.
Eine Operator-Neudefinition ist jedoch nur für Einsatzfälle erlaubt, bei denen mindestens *ein* benutzerdefinierter Datentyp beteiligt ist. Das Verhalten der Operatoren bei vordefinierten Datentypen kann also (zum Glück) nicht geändert werden.

Im Bruchrechnungs-Projekt besitzt die Klasse `Bruch` zum Addieren zweier Brüche bereits eine Methode, deren Verhalten noch verbessert werden kann mit Hilfe der Methode `kuerze()`, die Sie als Übungsaufgabe erstellen sollen (siehe Abschnitt 6.13):

```
void Bruch::addiere(Bruch& b1, Bruch& b2)
{
  zaehler = b1.hz() * b2.hn() + b1.hn() * b2.hz();
  nenner  = b1.hn() * b2.hn();
  kuerze();
}
```

In der Anweisung

```
lsg.addiere(b1, b2);
```

wird dem `Bruch`-Objekt `lsg` die folgende Nachricht zugeschickt: „Nimm die Summe der beiden Argumente `b1` und `b2` als neuen Wert an.“
Es wäre etwas anschaulicher, den selben Zweck mit folgender Formulierung zu erreichen:

```
lsg = b1 + b2;
```

Um dies zu ermöglichen, definieren wir eine neue `Bruch`-Methode mit dem merkwürdigen Namen **operator+**, die vom Compiler ausgeführt werden soll, wenn das Pluszeichen zwischen zwei `Bruch`-Objekten auftaucht:

```
Bruch Bruch::operator+(const Bruch& re) const
{
   Bruch temp(zaehler * re.hn() + nenner * re.hz(), nenner * re.hn());
   temp.kuerze();
   return temp;
}
```

Man verfährt wie bei einer gewöhnlichen Methodendefinition, verwendet aber als Namen das Schlüsselwort **operator** mit dem jeweiligen Operationszeichen als Suffix.
Von den beiden `Bruch`-Argumenten, welche die binäre Additionsoperation benötigt, wird das erste von dem Objekt repräsentiert, das die Nachricht erhält. Das zweite Argument (rechts vom Operatorzeichen anzuliefern) taucht in der Implementation als Parameter auf. Die Elementfunktion **operator+** wird also für das links vom Operatorzeichen stehende `Bruch`-Objekt aufgerufen.
Weil im Beispiel beide Argumente unangetastet bleiben, wird die Methode als **const** definiert (vgl. Abschnitt 6.12.1).
Wäre die neue Methode nicht mit dem Anspruch belastet, einen perfekt gekürzten Ergebnisbruch zu liefern, hätte sie noch knapper formuliert werden können, z.B.:

```
Bruch Bruch::operator+(Bruch& b2)
{
   return Bruch(zaehler * b2.hn() + nenner * b2.hz(), nenner * b2.hn());
}
```

In der ersten Lösung wird ein lokales `Bruch`-Objekt verwendet, um die vorhandene `Bruch`-Methode `kuerze()` benutzen zu können. Sein Inhalt nach dem Kürzen in der **return**-Anweisung als Funktionswert übergeben.
Auch im zweiten Lösungsvorschlag wird im Rahmen der **return**-Anweisung ein lokales `Bruch`-Objekt erzeugt.

Mit dem überladenen Operator lassen sich `Bruch`-Additionen nun besonders übersichtlich formulieren:

Quellcode	Ausgabe

`#include <Bruch.h>`
 	
`void main()`
`{`
`  Bruch b1(1,2), b2(3, 4), lsg;`
 	
`  b1.zeige(); cout << endl;`
`  b2.zeige(); cout << endl;`
 	
`  lsg = b1 + b2;`
 	
`  lsg.zeige(); cout << endl;`
`}`
	`  1`
	`-----`
	`  2`
	
	`  3`
	`-----`
	`  4`
	
	`  5`
	`-----`
	`  4`

Überladene Operatoren behalten in komplexen Ausdrücken ihre ursprüngliche Priorität

Um Missverständnissen vorzubeugen soll noch erwähnt werden, dass Operatoren nicht nur durch Methoden, sondern auch durch Funktionen überschrieben werden können.

6.12.3 friend-Funktionen und friend-Klassen

In der Definition einer Klasse `K1` kann man fremde Funktionen bzw. Methoden oder auch komplette Klassen (mit sämtlichen Methoden) zu Freunden erklären, womit diese Zugriff auf die privaten `K1`-Elementvariablen erhalten. Diese Aufweichung der Datenkapselung ist akzeptabel, wenn ein

benötigter Datenzugriff anderenfalls nur sehr aufwändig realisierbar wäre. Durch eine **friend**-Deklaration kann die Datensicherheit sogar erhöht werden, wenn ein Zugriff nur für ausgewählte Freunde erlaubt und für alle anderen Programmbestandteile (durch Verzicht auf eine öffentliche Zugriffsmethode) verboten wird.
Im folgenden Beispielprogramm erklärt die Klasse `Rechteck` ihre Freundschaft gegenüber der Klasse `Kreis`. Dies ermöglicht der `Kreis`-Methode `rinre(Rechteck& reck)` auf die privaten `Rechteck`-Variablen `x` und `y` zuzugreifen, um passend zu einem konkreten `Rechteck`-Objekt den Radius des größten Kreises zu bestimmen, der in das Rechteck hinein passt.

Quellcode	Ausgabe

`#include <iostream>`
`#include <cmath>`
`using namespace std;`
 	
`class Rechteck;`
 	
`class Kreis`
`{`
`public:`
`  Kreis(){};`
`  Kreis(double rad) {r = rad;};`
`  double ma() {return acos(-1)*r*r;};`
`  double mr() {return r;};`
`  void rinre(const Rechteck &reck);`
`private:`
`  double r;`
`};`
 	
`class Rechteck`
`{`
`public:`
`  Rechteck(double a, double b) {x = a; y = b;};`
`  double ma() {return x*y;};`
`  **friend class Kreis;**`
`private:`
`  double x, y;`
`};`
 	
`void Kreis::rinre(const Rechteck &reck)`
`{`
`  if (reck.x <= reck.y)`
`    r = reck.x/2;`
`  else`
`    r = reck.y/2;`
`}`
 	
`void main()`
`{`
`  Rechteck reck(4, 3);`
`  Kreis kr;`
 	
`  kr.rinre(reck);`
`  cout  << "\nKreis-Radius = " << kr.mr() << endl;`
`}`
	`Kreis-Radius = 1.5`

Weil sich die beiden Klassendefinitionen gegenseitig voraussetzen, erhält der Compiler vor der `Kreis`-Definition die Vorabinformation

```
class Rechteck;
```

über die zu erwartende `Rechteck`-Definition. Außerdem darf die Methodendefinition

```
void Kreis::rinre(Rechteck &reck)
{
...
}
```

erst *nach* der `Rechteck`-Klassendefinition vorgenommen werden, weil sie einen Parameter von diesem Typ verwendet.

Abschließend soll noch demonstriert werden, wie eine Klasse ihre Freundschaft gegenüber einer einzelnen Funktion bekunden kann. Weil dabei durchaus auch eine Elementfunktion einer anderen Klasse in Frage kommt, kann das obige Beispiel in leicht abgewandelter Form erneut verwendet werden:

```
class Rechteck
{
public:
   Rechteck(double a, double b) {x = a; y = b;};
   double ma() {return x*y;};
   friend void Kreis::rinre(const Rechteck &reck);
private:
   double x, y;
};
```

6.12.4 Der Pointer this

Innerhalb einer Elementfunktion einer Klasse kann man selbstverständlich auf jede Elementvariable des aktuellen Objekts direkt zugreifen. Gelegentlich ist es aber von Nutzen, auf das *komplette* Objekt (mit all seinen Elementen) zugreifen zu können, um etwa eine Kopie davon anzulegen. Wir könnten z.B. unsere `Bruch`-Klasse (siehe oben) um die folgende Methode `kf()` erweitern, die als Funktionswert eine gekürzte Kopie eines vorhandenen Bruchs liefert:

```
Bruch Bruch::kf()
{
   Bruch temp = *this;
   temp.kuerze();
   return temp;
}
```

Hier wird die `Bruch`-Methode `kuerze()` verwendet, die Sie als Übungsaufgabe erstellen sollen (siehe unten). Weil die Methode `kuerze()` die Daten ihres Objektes verändert, erzeugt `kf()` zunächst eine Kopie des aktuellen Objektes, die dann gekürzt und schließlich als Funktionswert übergeben wird.
Zum Erstellen der Kopie wird ein neues `Bruch`-Objekt erzeugt und gleich durch Zuweisung des aktuellen Objektes initialisiert. Bei der Zuweisung wird das Schlüsselwort **this** verwendet, das einen Pointer auf das aktuelle Objekt repräsentiert und deshalb im Beispiel dereferenziert werden muss.
In folgendem Programm wird die neue `Bruch`-Methode verwendet:

Quellcode	Ausgabe

`#include "Bruch.h"`
 	
`void main()`
`{`
`  Bruch b(4,8), kb;`
`  kb = b.kf();`
`  kb.zeige();`
`}`
	`  1`
	`-----`
	`  2`

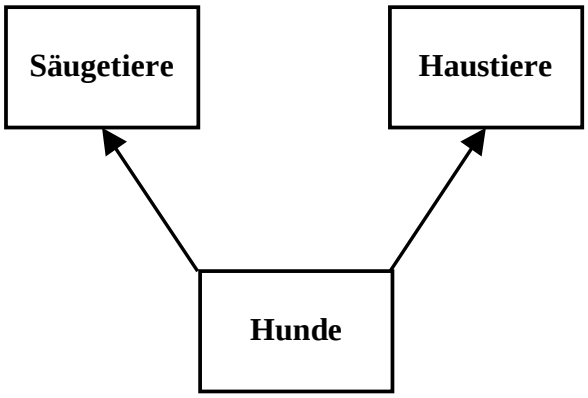
Zugegebenermaßen bringt die neue Methode `kf()` beim Erzeugen einer gekürzten Kopie zu einem vorhandenen Bruch nur wenige Vorteile gegenüber folgendem Vorgehen:

```
void main()
{
  Bruch b(4,8), kb = b;
  kb.kuerze();
  kb.zeige();
}
```

Aber in einigen Situationen ist es für eine Methode wirklich nützlich, mit Hilfe des **this**-Pointers ermitteln zu können, für welches Objekt sie aufgerufen wurde, wessen Daten sie also gerade bearbeitet.

6.12.5 Mehrfache Vererbung

Für Klassen bzw. Objekte der realen Welt ist es typisch, zu *mehreren* übergeordneten Klassen zu gehören. Z.B. ist ein Hund sowohl ein Säugetier, als auch ein Haustier:



Um die reale Welt möglichst gut modellieren zu können, erlaubt C++ (im Gegensatz zu anderen Programmiersprachen wie Java und Delphi) die Definition von Klassen, die mehrere Basisklassen beerben, d.h. die Elementvariablen und Methoden mehrerer Basisklassen übernehmen.
Dazu gibt man in der Definition der abgeleiteten Klasse mehrere Ahnen mit jeweiligem Vererbungsmodus an, z.B.:

Quellcode	Ausgabe

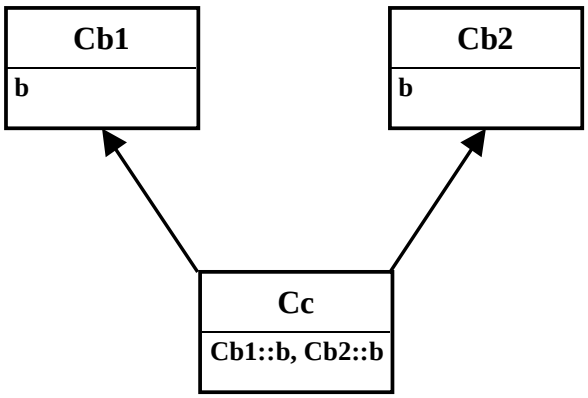
`#include <iostream>`
`using namespace std;`
 	
`class Ca`
`{`
`public:`
`  Ca() {a = 1;};`
`protected:`
`  int a;`
`};`
 	
`class Cb1 : public Ca`
`{`
`public:`
`  Cb1() {b = 21;};`
`protected:`
`  int b;`
`};`
	`a: 1`
	`b: 22`

```
class Cb2 : public Ca
{
public:
  Cb2() {b = 22;};
protected:
  int b;
};

class Cc : public Cb1, public Cb2
{
public:
  Cc() {c = 3;};
  int za() {return Cb1::a;};
  int zb() {return Cb2::b;};
private:
  int c;
};

void main()
{
  Cc vc;

  cout << "a: " << vc.za() << endl;
  cout << "b: " << vc.zb() << endl;
}
```

Bei der Mehrfachvererbung kann es zu Namenskonflikten kommen:

a) In mehreren Basisklassen werden Objekte mit dem selben Namen deklariert

In obigem Beispiel erbt die Klasse `Cc` von den beiden Basisklassen `Cb1` und `Cb2` eine Elementvariable namens `b`:



Um die Mehrdeutigkeit zu beseitigen, werden Klassenname und Scope-Operator vorangestellt, z.B.:

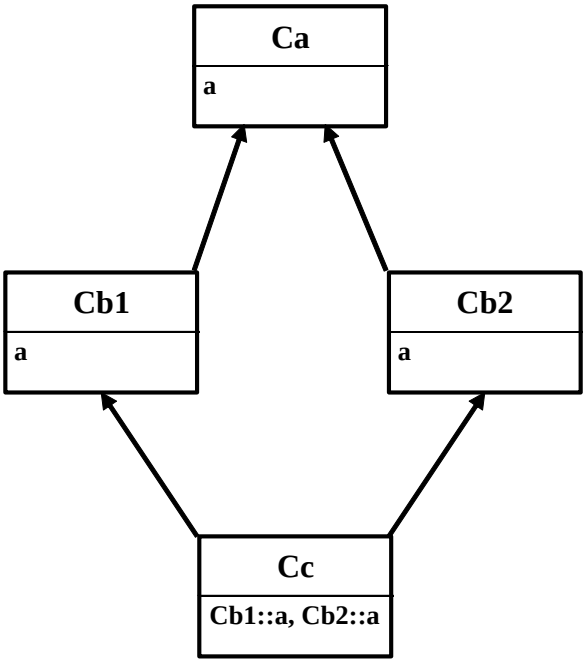
```
  int zb() {return Cb2::b;};
```

Anderenfalls gibt es eine recht eindeutige Fehlermeldung des Compilers:

```
error C2385: ‚Cc::b' ist mehrdeutig
warning C4385: ‘kann sein ‘b’ in Basisklasse ‘Cb1’ of class ‘Cc’
warning C4385: ‘oder ‘b’ in Basisklasse ‘Cb2’ of class ‘Cc’
```

b) Mehrere Basisklassen stammen von einem gemeinsamen Urahn ab

In diesem Fall gelangen Elemente des Urahns über alle betroffenen Basisklassen in die abgeleitete Klasse, wo sie dann entsprechend mehrfach existieren. Auch diese Konstellation findet sich in obigem Beispiel:



Auch hier lässt sich die Mehrdeutigkeit durch Voranstellen von Basisklassenname und Scope-Operator beseitigen:

```
int za() {return Cb1::a;};
```

Nötigenfalls erläutert der Compiler den Konflikt:

```
error C2385: 'Cc::a' ist mehrdeutig
warning C4385: 'kann sein 'a' in Basisklasse 'Ca' of base 'Cb1' of class 'Cc'
warning C4385: 'oder 'a' in Basisklasse 'Ca' of base 'Cb2' of class 'Cc'
```

Um zu verhindern, dass Urahn-Elemente mehrfach geerbt werden, kann man mit **virtuellen Basisklassen** arbeiten, z.B.:

```
class Cb1 : virtual public Ca
{
public:
   Cb1() {b = 21;};
protected:
   int b;
};

class Cb2 : virtual public Ca
{
public:
   Cb2() {b = 22;};
protected:
   int b;
};

class Cc : public Cb1, public Cb2
{
public:
   Cc() {c = 3;};
   int za() {return a;};
   int zb() {return Cb2::b;};
protected:
   int c;
};
```

Nähere Hinweise zur Verwendung von virtuellen Basisklassen finden sich z.B. RRZN (1998, S. 72ff).

Insgesamt ist die Verwendung der Mehrfachvererbung als Fehlerquelle nicht zu unterschätzen, weshalb die Designer einiger OOP-Programmiersprachen (z.B. Java, C#) bewusst darauf verzichtet haben. Natürlich ist die Verfügbarkeit der Mehrfachvererbung nicht als Nachteil von C++ zu werten, weil ja niemand zur (fehlerhaften) Verwendung dieser Technik verpflichtet ist.

6.13 Übungsaufgaben zu Abschnitt 6

1) Erweitern Sie die Klasse `Bruch` um eine Elementfunktion zum vollständigen Kürzen von Brüchen.

2) Testen Sie, in welcher Reihenfolge die Konstruktoren bzw. Destruktoren der Basisklasse bzw. der abgeleiteten Klasse aufgerufen werden, wenn ein Objekt einer abgeleiteten Klasse erzeugt bzw. zerstört wird.

3) Überladen Sie den Operator ***** mit der für `Bruch`-Objekte definierten Multiplikation, und testen Sie den Operator in folgender Hauptfunktion:

Quellcode	Ausgabe

`#include "Bruch.h"`
 	
`void main()`
`{`
`  Bruch b1(1,2), b2(3, 4), lsg;`
 	
`  b1.zeige(); cout << endl;`
`  b2.zeige(); cout << endl;`
 	
`  lsg = b1 * b2;`
 	
`  lsg.zeige(); cout << endl;`
`}`
	`  1`
	`-----`
	`  2`
	
	`  3`
	`-----`
	`  4`
	
	`  3`
	`-----`
	`  8`

4) Erstellen Sie eine Klasse für zweidimensionale Vektoren, die mindestens über Elementfunktionen mit folgenden Leistungen erfügt:

- Erste/zweite Koordinate lesen/setzen
- Länge (Betrag) ermitteln

  Der Betrag eines Vektors $x = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$ ist definiert durch:

  $$|x| := \sqrt{x_1^2 + x_2^2}$$

  Verwenden Sie die Funktion **sqrt()**, um die Quadratwurzel aus einer **double**-Zahl zu berechnen. Dazu muss die Header-Datei **cmath** inkludiert werden.

- Vektor auf Länge 1 normieren
  Dazu dividiert man beide Komponenten durch die Länge des Vektors, denn mit

  $$\tilde{x} := (\tilde{x}_1, \tilde{x}_2) \text{ sowie } \tilde{x}_1 := \frac{x_1}{\sqrt{x_1^2 + x_2^2}} \text{ und } \tilde{x}_2 := \frac{x_2}{\sqrt{x_1^2 + x_2^2}} \text{ gilt:}$$

  $$|\tilde{x}| := \sqrt{\tilde{x}_1^2 + \tilde{x}_2^2} = \sqrt{\left(\frac{x_1}{\sqrt{x_1^2 + x_2^2}}\right)^2 + \left(\frac{x_2}{\sqrt{x_1^2 + x_2^2}}\right)^2} = \sqrt{\frac{x_1^2}{x_1^2 + x_2^2} + \frac{x_2^2}{x_1^2 + x_2^2}} = 1$$

- Vektoren (komponentenweise) addieren und subtrahieren

  Die Summe der Vektoren $x = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$ und $y = \begin{pmatrix} y_1 \\ y_2 \end{pmatrix}$ ist definiert durch:

  $$\begin{pmatrix} x_1 + y_1 \\ x_2 + y_2 \end{pmatrix}$$

- Skalarprodukt zweier Vektoren ermitteln

  Das Skalaprodukt der Vektoren $x = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$ und $y = \begin{pmatrix} y_1 \\ y_2 \end{pmatrix}$ ist definiert durch:

  $$x \cdot y := x_1 y_1 + x_2 y_2$$

- Winkel zwischen zwei Vektoren ermitteln
  Für den Kosinus des Winkels, den zwei Vektoren *x* und *y* im mathematischen Sinn (links herum) einschließen, gilt:

  $$\cos(x, y) = \frac{x \cdot y}{|x||y|}$$

  Um daraus den Winkel (im Bogenmaß), können Sie die **cmath**-Funktion **acos()** verwenden.

- Rotation eines Vektors um einen bestimmten Winkelgrad
  Mit Hilfe der Rotationsmatrix

  $$D := \begin{pmatrix} \cos(\alpha) & -\sin(\alpha) \\ \sin(\alpha) & \cos(\alpha) \end{pmatrix}$$

  kann der Vektor *x* um den Winkel α (im Bogenmaß) gedreht werden:

  $$x' = D\,x = \begin{pmatrix} \cos(\alpha) & -\sin(\alpha) \\ \sin(\alpha) & \cos(\alpha) \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} \cos(\alpha)\,x_1 - \sin(\alpha)\,x_2 \\ \sin(\alpha)\,x_1 + \cos(\alpha)\,x_2 \end{pmatrix}$$

  Die trigonometrischen Funktionen können mit den **cmath**-Funktionen **cos()** und **sin()** berechnet werden.

Erstellen Sie ein Demonstrationsprogramm, das Ihre Vektoren-Klasse verwendet und näherungsweise die folgende Ausgabe erstellt:

```
Vektor v1:              (1.00,  0.00)
Vektor v2:              (1.00,  1.00)

Laenge von Vektor v1:   1.00
Laenge von Vektor v2:   1.41

Winkel zw. v1 und v2:   0.79

Wie soll Vektor 2 gedreht werden (Bogenmass): 0.785

Vektor v2:              (0.00,  1.41)

Vektor v2 normiert:     (0.00,  1.00)

v1 + v2:                (1.00,  1.00)
```

Beim Lösungsvorschlag wurde mit

```
cout.setf(ios::fixed);
cout.precision(2);
```

für eine Festkommaschreibweise mit 2 Dezimalstellen gesorgt. Außerdem wurden horizontale Tabulatoren (\t) zur Gestaltung der Ausgabe verwendet.

7 Templates

Die strenge Typisierung in C++ sorgt einerseits für stabile Programme, droht jedoch andererseits den Programmieraufwand zu steigern, weil in Funktions- und Klassendefinitionen die Datentypen von Argumenten bzw. Elementvariablen genau festgelegt werden müssen. Auf unserem derzeitigen C++ - Entwicklungsstand muss z.B. eine Funktion zum Vertauschen von zwei Variablen für jeden zu unterstützenden Datentyp separat erstellt werden. Da in C++ die Wiederverwendbarkeit von Software eine sehr hohe Priorität besitzt, hat man mit den Templates (Vorlagen, Schablonen) eine Lösung entwickelt, die unproduktive und fehleranfällige Mehrfacharbeit verhindert, ohne die Typkontrolle des Compilers zu reduzieren.
In manchen moderneren Programmiersprachen, die durch Reduktion des C++ - Sprachumfangs das Programmieren einfacher und sicherer machen wollen (z.B. Java), rüstet man die Templates gerade nach.

7.1 Funktions-Templates

Die Definition einer Funktions-Vorlage wird mit dem Schlüsselwort **template** eingeleitet, dem eine spitz eingeklammerte Liste der Template-Parameter folgt. Bei einem Template-Parameter handelt es sich um einen Datentyp-Platzhalter, dem das Schlüsselwort **class** vorangestellt wird, obwohl als Aktualparameter keinesfalls nur Klassen erlaubt sind. Mehrere Parameter werden durch Komma getrennt. Anschließend folgt eine normale Funktionsdeklaration bzw. -definition, in dem die Template-Parameter wie gewöhnliche Typbezeichner verwendet werden.
Im wohl beliebtesten Template-Demonstrationsbeispiel werden die Inhalte zweier Variablen ausgetauscht:

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

template <class T> void tausche(T& a1, T& a2)
{
   T tmp = a1;
   a1 = a2;
   a2 = tmp;
}

void main() {
   int eins = 1, zwei = 2;
   tausche(eins, zwei);
   cout << eins << zwei << endl;

   char a = 'a', b = 'b';
   tausche(a, b);
   cout << a << b << endl;
}
```

Das Schlüsselwort **template** informiert den Compiler darüber, dass in der anschließenden Funktionsdefinition (oder –deklaration) unbestimmte Typen auftreten. Beim ersten Aufruf von `tausche()` mit einem bestimmten Template-Aktualparameter erzeugt der Compiler den Objektcode zu einer konkrete Funktionsdefinition, z.B. für:

```
void tausche(int& a1, int& a2)
{
   int tmp = a1;
   a1 = a2;
   a2 = tmp;
}
```

An Stelle dieser impliziten Instantiierung einer Funktionsvorlage kann man durch Deklaration eines passenden Prototypen auch eine explizite Instantiierung erzwingen, z.B.:

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

template <class T> void tausche(T& a1, T& a2)
{
   T tmp = a1;
   a1 = a2;
   a2 = tmp;
}

template void tausche(double&, double&);

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

Dies ist in sinnvoll in Programmen mit mehreren Modulen, denn für das Erzeugen oder Verwenden von Template-Funktionen gelten folgenden Vorschläge bzw. Regeln (nach RRZN, 1998b, S. 106):

- Der Compiler erzeugt nur die im Quellmodul mit der Template-Definition implizit (durch Funktionsaufruf) oder explizit (durch eine Prototyp-Deklaration) instantiierten Funktionen.
- Aufrufe in einem anderen Quellmodul führen *nicht* zur Definition neuer Überladungen, sondern zu Externreferenzen und entsprechenden Fehlermeldungen des Linkers, falls zu der benötigten Überladung kein Objektcode existiert.
- Steckt man die komplette Template-Definition in eine Header-Datei, können in jedem Modul, das die Header-Datei einbindet, Funktionsdefinitionen generiert werden. Dabei besteht jedoch die Gefahr, dass es zu verbotenen Mehrfachdefinitionen kommt. Daher sollte man nur die Template-Deklaration in einer Header-Datei unterbringen, z.B.:

  ```
  template <class T> void tausche(T& a1, T& a2);
  ```

  Die Template-Definition bringt man mit geeigneten impliziten oder expliziten Funktions-Instantiierungen in einem Quellmodul unter.

Ohne implizite oder explizite Instantiierung bleibt eine Funktions-Schablone ohne jeden Effekt auf das entstehende Programm. Die konkreten Funktionsdefinitionen werden aus der Vorlage mechanisch generiert, wobei eine Serie überladener Funktionen entsteht. Dies erledigt der Compiler zur Übersetzungszeit, so dass zu Laufzeit keinerlei Zeitaufwand entsteht.

Durch Anwendung der Vorlage auf ungeeignete Aktual-Typen können durchaus auch unsinnige Ergebnisse resultieren können. In folgendem Fall passiert ein harmloser Typanpassungsfehler:

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

template <class T> void test(T p)
{
   int i = p;
   cout << i;
}

void main() {
   test("test");
}
```

Der Compiler erkennt den Fehler und meldet:

```
Kompilierung läuft...
Test.cpp
U:\Eigene Dateien\VCpp\Test\Test.cpp(7) : error C2440: 'initializing' : 'char *'
kann nicht in 'int' konvertiert werden
```

Es sind durchaus diffizilere Fehler denkbar, die nicht vom Compiler abgefangen werden. Leider ist es dem Template-Designer nicht möglich, die für einen Template-Parameter zulässigen Aktual-Typen einzuschränken.

7.2 Klassen-Templates

Wenngleich es den Begriff der „überladenen Klassen“ nicht gibt, werden doch oft Serien von Klassen benötigt, die sich bei identischen Handlungskompetenzen nur durch den Datentyp gewisser Elemente unterscheiden. In OOP-Sprachen mit einer Urahn-Klasse, also einer Single-Tree - Klassenhierarchie (z.B. Java, Smalltalk), wird man in einer solchen Situation mit der Vererbungstechnik arbeiten. In C++ ist auch mit unverbundenen Bäumen zu rechnen, so dass hier der Entwurf einer Klassen-Schablone zu bevorzugen ist (Eckel 2000, Band I, S. 727ff). Für die Bezeichnung *Klassen-Template* sind folgende Synonyme im Umlauf, die jeweils spezielle Aspekte des Begriffs hervorheben: parametrisierter Datentyp, generische Klasse, Klassengenerator.
Die Definition einer Klassen-Vorlage wird mit dem Schlüsselwort **template** eingeleitet, dem zwischen spitzen Klammern eine Parameterliste folgt. Dabei sind erlaubt:

- **Datentyp-Parameter**
  Hier wird dem Bezeichner das Schlüsselwort **class** vorangestellt. Ersatzweise ist auch das Schlüsselwort **typename** erlaubt.
- **Variablen-Parameter**
  Anders als bei Funktions-Templates sind auch Variablen-Parameter erlaubt, wobei dem Bezeichner der jeweilige Datentyp vorangestellt wird. So kann man z.B. eine Klassen-Vorlage mit einem Array-Element geeignet parametrisieren.

Nach der Parameterspezifikation folgt eine gewöhnliche Klassendefinition, wobei die vereinbarten Platzhalter zum Einsatz kommen.
Im folgenden Beispiel wird eine Vorlage für fest-dimensionierte Array-Klassen mit Feldgrenzenüberwachung definiert:

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

template <class ElementType, int maxSize> class Charray
{
public:
  ElementType& operator[](int index)
  {
    assert(index >= 0 && index < maxSize);
    return daten[index];
  }
private:
  ElementType daten[maxSize];
};

void main()
{
  Charray<int, 10> chest;
  for (int i=0; i<10;i++)
    chest[i] = i;
  cout << chest[3] << endl;
}
```

Der Compiler erzeugt eine konkrete Klasse, sobald diese benutzt wird. Zu diesem Zweck bildet man aus dem Vorlagennamen durch Anhängen von spitz eingeklammerten Aktualparametern eine Klassenbezeichnung, z.B.:

```
Charray<int, 10> chest;
```

Hier entsteht eine Klasse zur Verwaltung eines 10-elementigen **int**-Arrays. Wird eine umständliche Bezeichnung für eine Template-Klasse oft benötigt, kann man eine Abkürzung vereinbaren, z.B.:

```
typedef Charray<int, 10> CIA10; //checked int array
```

Man kann für die Template-Parameter auch Standardwerte festlegen, z.B.:

```
template <class ElementType=int, int maxSize=10> class Charray
{ . . . };
```

Nach dieser Template-Definition ist z.B. folgende Instantiierung erlaubt:

```
Charray<> chest;
```

Für das Instantiieren einer Template-Klasse in einem Programm mit mehreren Modulen gelten die selben Regeln wie bei Funktions-Templates (siehe Abschnitt 7.1), so dass auch die Option der *explizite* Instantiierung einer Klasse von Interesse ist. Leider ist dazu ist die komplette Klassendefinition erforderlich, z.B.:

```
class Charray<double, 20>
{
public:
   ElementType& operator[](int index)
   {
      assert(index >= 0 && index < maxSize);
      return daten[index];
   }
private:
   ElementType data[maxSize];
};
```

Weil im Beispiel alle Methoden **inline** definiert sind, ist so überhaupt keine Schreibersparnis mehr zu erzielen. Normalerweise erspart man sich immerhin das Implementieren der nicht **inline** definierten Methoden. Vermutlich ist oft das Erzeugen eines Dummy-Objektes die sinnvollste Vorgehensweise.

Es sind noch einige Details zu der obigen Template-Definition zu erklären. Zentrales Element ist eine Überladung des Offset-Operators „**[]**“. Als Rückgabetyp der Elementfunktion `operator[]` wird eine *Referenz* angegeben, die Ihnen bisher nur im Kontext von Referenz*parametern* vorgestellt wurde. Im Falle der konkreten Klasse `Charray<int, 10>` liefert `operator[]` einen Rückgabewert vom Typ **int&**.
Generell handelt es sich bei Referenzen um sehr handlichen Zeiger, weil sie vom Compiler automatisch dereferenziert werden, was im Vergleich zu Pointern zu einer eleganteren Syntax führt.
Ist `chest` ein Objekt aus der Klasse `Charray<int, 10>`, dann liefert z.B. der Ausdruck `chest[2]` eine Referenz auf das dritte Element des internen (privaten) **int**-Arrays des Objektes. Dieses Element kann über die Referenz ohne die Umständlichkeit des Inhaltsoperators angesprochen werden, z.B. in:

```
chest[i] = i;
```

Damit dürfte klar sein, dass Referenzen gerade beim Überladen von Operatoren unverzichtbar sind. Zwar sind Referenztypen auch in normalen Variablen-Definitionen (mit obligatorischer Initialisierung!) erlaubt, doch werden sie in der Regel nur für Parameter und Ruckgabewerte von Funktionen bzw. Methoden eingesetzt.

Es erscheint mir im Übrigen recht bemerkenswert, dass die vom überladenen Offset-Operator gelieferte Referenz einen problemlosen Zugriff auf die private Elementvariable `daten` erlaubt. Im vorliegenden Beispiel ist dies natürlich kein Nachteil, sondern eine grundlegende Voraussetzung für die sinnvolle Verwendung der Klassen-Vorlage.

Motiv für das Design der Klassen-Vorlage war das Ziel, für Arrays (mit beliebigem Elementendatentyp) eine Feldgrenzenüberwachung zu realisieren. Dabei spielt die Funktion **assert()** eine wesentliche Rolle. Hat der als Aktualparameter übergebene Ausdruck den Wert **false**, dann gibt sie eine Meldung aus (via **stderr** und per Windows-Dialogbox) und beendet das Programm. Zwar kann man über **exceptions** noch weit flexibler auf Feldgrenzenüberschreitungen reagieren, doch ist ein Beenden des Programms in der Regel sinnvoller, als das Weiterarbeiten mit falschen Daten.
Ohne den **assert()**-Aufruf bleibt in obigem `chest`-Beispiel der Zugriff auf das Element 13 unentdeckt:

Quellcode	Ausgabe

`void main()`
`{`
`  Charray<int, 10> chest;`
`  for (int i=0; i<10;i++)`
`  chest[i] = i;`
`  cout << chest[13] << endl;`
`}`
	`3280480`

Mit dem **assert()**-Aufruf endet das Programm hingegen mit einer ausführlichen Fehlerbeschreibung:

```
Assertion failed: index >= 0 && index < maxSize, file u:\eigene dateien\vcpp\cla
sstemp\classtemp.cpp, line 10
```

Mit den Klassen-Templates gibt es in C++ neben der Vererbung eine zweite Option zum rationellen Generieren von Klassen und es gilt, beide Techniken geschickt zu kombinieren. Es ist z.B. zu beachten, dass jede aus einer Vorlage entstehende Klasse einen vollständigen Satz aller Methoden besitzt, was den Speicherbedarf des Programms belasten kann. Folglich empfiehlt es sich, die in allen Template-Klassen identischen Methoden in einer Basislasse zu implementieren, die in der Klassen-Vorlage beerbt wird.

7.3 Übungsaufgaben zu Abschnitt 7

1) Erstellen Sie ein Funktions-Template, aus dem sich Funktionen zur Bestimmung des Maximums zweier Argumente gewinnen lassen, sofern der konkret angebotene Datentyp den Operator „>“ unterstützt. Es soll z.B. die folgende Hauptfunktion übersetzt werden können:

```
void main()
{
  double x = 17.45, y = 23.77;
  string s1 = "Meyer", s2 = "Mayer";
  cout << "max(x, y):\t" << max(x,y) << endl;
  cout << "max(s1, s2):\t" << max(s1,s2) << endl;
};
```

8 Textverarbeitungsklassen

Die Ausführungen über die Architektur von C-Strings (Stichworte: **char**-Arrays, Pointer) und über Funktionen zu ihrer Manipulation gehören vermutlich nicht zu den angenehmsten Erinnerungen aus ihrer bisherigen C++ - Lerngeschichte. Nachdem Sie die erforderlichen Kenntnisse im Umgang mit dem immer noch wichtigen Datentyp **char*** erworben haben, dürfen Sie endlich den Komfort der Standardklasse **string** genießen.
Wer sie nutzen will, muss die Headerdatei **string** inkludieren und sollte der Einfachheit halber (wie bei **iostream**) den Namensraum **std** importieren, zu dem in ANSI/ISO - C++ alle Klassen und Funktionen der C++ - Standardbibliothek gehören. Weil die **string**-Designer von den Möglichkeiten der Operator-Überladung reichlich Gebrauch gemacht haben, kann beim Umgang mit **string**-Objekten eine intuitive Syntax benutzt werden.

8.1 Operationen mit string-Objekten

Wir beschränken uns auf kurze Hinweise zu elementaren Operationen mit **string**-Objekten. Eine ausführliche Darstellung findet sich z.B. bei Eckel (2000, Band 2, S. 27ff).

Wertzuweisung

Über den Zuweisungsoperator „=“ kann einem **string**-Objekt ein Zeichenketten-Literal, ein C-String oder ein anderes **string**-Objekt zugewiesen werden, wobei der alte Inhalt ersetzt wird, z.B.:

```
string s1 = "Ich bin s1";
s2 = s1;
```

Bei der Initialisierung kann auch die Konstruktor-Syntax verwendet werden, z.B.:

```
string s2("Ich bin s2");
```

Verkettung

Dazu kann der überladene Operator „+“ benutzt werden, z.B.:

```
cout << s1 + s2 << endl;
```

Dabei werden auch Zeichenketten-Literale und C-Strings unterstützt, z.B.:

```
cout << s1 + "!" << endl;
```

Wie die letzten Beispiel zeigen, wissen die **iostream**-Klassen, mit **string**-Objekten umzugehen.
Selbstverständlich können auch die Aktualisierungs-Operatoren benutzt werden, z.B.:

```
string s = "Hello";
s += " World!";
```

Vergleiche

Für einfache Aufgaben können die überladenen Operatoren (z.B. „==“, „!=“, „<“ etc.) verwendet werden, z.B.:

Quellcode	Ausgabe

`#include <iostream>`
`#include <string>`
`using namespace std;`
 	
`void main() {`
`  string s1 = "Hallo";`
`  string s2 = "Hallo";`
`  string s3 = "HalliHallo";`
`  cout << (s1 == s2) << endl`
`  << (s1 == s3) << endl;`
`}`
	`1`

Die Methode **compare()** erlaubt präzisere Vergleiche.

Durchsuchen

Es gibt eine ganze Sammlung von Methoden, einen **string** zu durchsuchen, z.B. **find()**, **find_first_of()**, **find_first_not_of()**.

Einfügen

Mit den diversen Überladungen der Methode **insert()** lassen sich Zeichenfolgen in einen **string** einfügen, z.B.:

Quellcode	Ausgabe

`#include <iostream>`
`#include <string>`
`using namespace std;`
 	
`void main() {`
`  string s = "Ho";`
`  s.insert(1, "all");`
`  cout << s << endl;`
`}`
	`Hallo`

Ersetzen

Mit den diversen Überladungen der Methode **replace()** lassen sich Teilzeichenfolgen ersetzen, z.B.:

Quellcode	Ausgabe

`#include <iostream>`
`#include <string>`
`using namespace std;`
 	
`void main() {`
`  string s = "Hallo World!";`
`  s.replace(6, 5, "Welt");`
`  cout << s << endl;`
`}`
	`Hallo Welt!`

Löschen

Mit den diversen Überladungen der Methode **erase()** lassen sich Teilzeichenfolgen löschen, z.B.:

Quellcode	Ausgabe

`#include <iostream>`
`#include <string>`
`using namespace std;`
 	
`void main() {`
`  string s = "Hallo World!";`
`  s.erase(5, 6);`
`  cout << s << endl;`
`}`
	`Hallo!`

Teilzeichenfolge extrahieren

Mit der Methode **substr()** lässt sich eine Teilzeichenfolge extrahieren, z.B.:

Quellcode	Ausgabe

`#include <iostream>`
`#include <string>`
`using namespace std;`
 	
`void main() {`
`  string s = "Hallo World!";`
`  cout << s.substr(6,5) << endl;`
`}`
	`World`

Informieren

Über die Methode **length()** erhält man die aktuelle Länge der Zeichenfolge.

Zeilenorientierte Eingabe von der Konsole

Bei Verwendung der **getline()**-Funktion muss nach meinen Erfahrungen zum Quittieren der Eingabe die **Enter**-Taste doppelt gedrückt werden:

```
getline(cin, s2);
```

Diese Unbequemlichkeit kann dem Benutzer durch folgenden Umweg über einen C-String und Verwendung der **istream**-Methode **getline()** erspart werden:

```
char s[80];
cin.getline(s, 80);
s2 = s;
```

8.2 Hinweise zum Klassen-Template basic_string

Auf der Suche nach Informationen zur Klasse **string** sollten Sie beachten, dass „string“ ein Aliasname ist:

```
typedef basic_string<char> string;
```

In der Online-Hilfe zu VC++ 6.0 (MSDN Library) finden Sie ausführliche Informationen zu den oben genannten Methoden im Zusammenhang mit dem Klassen-Template **basic_string**. Die Definition dieses Templates hier einzufügen, verbietet sich aus Platzgründen. Bei **basic_string<char>** alias **string** handelt es sich um eine konkrete Klasse zur Verwaltung von **char**-Elementen. In der C++ - Standardbibliothek findet sich auch die auf dem Klassen-Template **basic_string** basierende Klasse **wstring** zur Verwaltung von 16-Bit – Zeichen.

9 Standard Template Library (STL)

In vielen Computerprogrammen treten Gruppen von Objekten aus derselben (Basis-)Klasse in einer zum Zeitpunkt der Übersetzung unbekannten Gruppenstärke auf (z.B. Zeichnungselemente in einem Grafikprogramm, Zeichenfolgen in einer Textverarbeitung). Bisher haben wir solche Gruppen, deren Mitglieder durchaus auch von einem Standardtyp sein können, in traditionellen Arrays gesammelt. Jedoch fehlt einem Array in vielen Situationen die erforderliche Flexibilität (z.B. bei einer Steigerung der Kapazität oder beim Einfügen von Elementen), und vor allem fehlt ihm die Handlungskompetenz eines Objektes, Routineaufgaben (z.B. Anpassung der Kapazität) selbständig zu erledigen.
Auch in traditionellen Programmiersprachen finden sich schon Datenstrukturen, die dem Array in vielen Situationen überlegen sind (z.B. verkettete Listen, Schlüssel-Wert-Tabellen). Angereichert durch geeignete Handlungskompetenzen sind in C++ daraus die so genannten **Container-Klassen** entstanden, mit denen sich anspruchsvollere Projekt derzeit wohl am besten realisieren lassen. C++ enthält unter dem Namen **Standard Template Library** (**STL**)[1] in seiner Standardbibliothek eine umfangreiche Sammlung von Container-Klassen, die für viele Aufgaben geeignet sind und im Vergleich zu herstellerabhängigen Lösungen den Vorteil einer sehr guten Portabilität bieten.
Auch die Microsoft Foundation Classes (MFC), eine von Microsoft mit seinem Visual Studio ausgelieferte Klassenbibliothek, die uns als Basis für die objektorientierte Windows-Programmierung dienen wird, enthält leistungsfähige Container-Klassen. In diesem Kurs wird jedoch die STL-Lösung der Portabilität wegen bevorzugt. Dabei ist nicht nur die Portabilität auf eine andere Betriebssystem-Plattform gemeint, sondern auch die Unabhängigkeit vom Hersteller des Compilers.

Bei den STL-Containern kann man sequentielle und assoziative Varianten unterscheiden, wobei dank Template-Technik für die Elemente jeweils beliebige Datentypen erlaubt sind.

- Bei einem **sequentiellen Container** existiert eine wohldefinierte Anordnung aller Elemente. Wir werden die folgenden sequentiellen Klassen-Templates kennen lernen:
  - **vector** Eindimensionales, dynamisch wachsendes Feld
  - **list** Doppelt verkettete Liste

  Außerdem kennt die STL-Implementation der C++-Standardbibliothek noch:
  - **stack** LIFO-Stapel. Einfügen und Entnehmen von Elementen sind nur am oberen Ende möglich.
  - **deque** Bidirektionale Warteschlange. Einfügen und Entnehmen von Elementen sind an beiden Enden möglich.
- Die Elemente eines **assoziativen Containers** bestehen aus (Schlüssel - Wert) – Paaren, wobei man über den Schlüssel auf den Wert zugreifen kann.
  Wir werden die folgenden assoziativen Klassen-Templates kennen lernen:
  - **map** Sortierte Menge von Schlüsseln, die jeweils mit einem Wert assoziiert sind
  - **set** Sortierte Menge von Schlüsseln. Im Vergleich zur **map** fehlen also die Werte

  Außerdem kennt die STL-Implementation der C++-Standardbibliothek noch:
  - **multimap** Im Unterschied zur **map** kann ein Schlüssel mit mehreren Werten verknüpft sein.
  - **multiset** Im Unterschied zum **set** darf ein Element mehrfach auftreten.

[1] Diese allgemein übliche Darstellung ist nicht ganz korrekt, weil die STL nur teilweise in die C++ - Standardbibliothek integriert wurde und weiterhin als eigenständige, von der Firma SGI gepflegte, C++ - Bibliothek besteht. (vgl. Eckel, 2000, Band I, S. 113).

Neben den Containern beinhaltet die STL auch die so genannten **Algorithmen**, die über den Containern operieren. Dabei handelt es sich um Funktions-Templates, die durch einige Tricks so generisch (datentypunabhängig) formuliert wurden, dass sie auf möglichst viele Container angewendet werden können, z.B.: **find()**, **sort()**, **replace()**, **copy()**, **merge()**.
Die strikte Trennung von Containern zur Datenverwaltung einerseits und (möglichst typfrei programmierten) Algorithmen andererseits ist eine Kernidee der STL. Wenn mich nicht alles täuscht, wird hier ein wichtiges OOP-Prinzip „gelockert". Offenbar war das Ziel, unterschiedliche Container möglichst rationell mit Handlungskompetenzen für Routineaufgaben auszustatten, so besser zu realisieren. Es wird sich zeigen, dass die Container-Klassen einen entscheidenden Beitrag zur Typfreiheit der Algorithmen leisten, indem sie so genannte **Iteratoren** zur Verfügung stellen. Diese Objekte zeigen wie Pointer auf Container-Elemente und leisten eine Anordnung der Elemente, die sich per Pointer-Arithmetik (z.B. durch Inkrementieren) unabhängig von der internen Struktur durchqueren lässt.
Weil die Algorithmen über Iteratoren auf die Container-Elemente zugreifen, ergibt sich für die einzelnen Container durchaus ein teilweise individuelles Verhalten, wobei die von virtuellen Methoden bekannten Performanz-Probleme aber vermieden werden.
Die STL stellt keinesfalls eine Abkehr von der OOP dar; schließlich besteht sie wesentlich aus Klassen-Schablonen. Jedoch spielt der Vererbungsmechanismus nur eine untergeordnete Rolle und wird als Klassengenerator von der Template-Technologie ersetzt. Insbesondere wird die elegante, aber zeitaufwendige, Polymorphie durch eine performantere Alternative ersetzt, was bei zeitkritischen Operationen wie Sortieren oder Suchen durchaus sinnvoll ist. Wir werden später im Zusammenhang mit den Microsoft Foundation Classes (MFC) noch einen anderen Polymorphie-Ersatz (namens Message Mapping) kennen lernen.

Nach diesem etwas zu lang und zu abstrakt geratenen Überblick, wenden wir uns den unerhört praktischen **vector** - Klassen zu, die den klassischen Arrays vorzuziehen sind, wenn die Anzahl der zu verarbeitenden Elemente zur Laufzeit wachsen kann.

9.1 Vektoren

Um auf Basis der Klassen-Schablone **vector** eine Klasse mit passendem Elementtyp zu instantiieren und ein zugehöriges Objekt erzeugen zu können, muss die Header-Datei **vector** inkludiert werden. In folgendem Beispiel werden **string**-Elemente mit einem **vector**-Objekt verwaltet.
Wir verzichten vorläufig auf STL-Algorithmen und -Iteratoren, benutzen aber mit **ifstream** (Input File Stream) erstmals eine **IOStream**-Klasse aus der C++ - Standardbibliothek, weshalb auch die Header-Datei **ifstream** inkludiert werden muss. Durch die Anweisung

```
ifstream in("test.txt");
```

wird versucht, die Datei **test.txt** zum Lesen zu öffnen. Zur Erfolgskontrolle prüft man mit Hilfe einer Überladung des „**!**"-Operators, ob ein Fehlerbit gesetzt ist, z.B.:

```
if (!in) { … }
```

War das Öffnen erfolgreich, kann ein **ifstream**-Objekt im Wesentlichen genau so zum Lesen aus einer Datei verwendet werden, wie wir es mit dem Standardeingabeobjekt **cin** zum Lesen von der Konsole gewohnt sind.

```
#include <string>
#include <iostream>
#include <fstream>
#include <vector>
using namespace std;
```

```
void main()
{
  vector<string> sv;
  string line;
  ifstream in("test.txt");
  if (!in)
  {
    cout << "Eingabedatei nicht gefunden!";
    exit(1);
  }
  while(getline(in, line))
    sv.push_back(line);
  for(int i = 0; i < sv.size(); i++)
    cout << "Zeile " << i+1 << ": "
         << sv[i] << endl;
  sv.clear();
  cout << sv.size() << endl;
}
```

Nach dem IOStream-Einschub nun wieder zur **vector**-Klasse:

- Im Beispiel werden mit der Methode **push_back()** neue Elemente am Ende des **vector**-Objektes eingefügt:

  ```
  sv.push_back(line);
  ```

  Das Objekt legt seine Elemente auf dem Heap ab und erledigt die Speicherverwaltung selbständig.
- Mit der für alle STL-Container verfügbaren Methode **size()** stellt man fest, wie viele Elemente aktuell vorhanden sind:

  ```
  sv.size();
  ```
- Beim Zugriff auf **vector**-Elemente kann dank einer geeigneten Überladung der Offset-Operator benutzt werden:

  ```
  sv[i]
  ```

  Während das Einfügen neuer Elemente per **push-back()** narrensicher ist, kann man mit dem Offset-Operator wie beim C-Array durchaus Zugriffsverletzungen begehen.
- Mit der für alle STL-Container verfügbaren Methode **clear()** kann man alle Elemente löschen:

  ```
  sv.clear();
  ```

Mit einer Beispiel-Textdatei passenden Namens im aktuellen Verzeichnis liefert obiges Programm:

```
Zeile 1: Mit einer vector-Klasse
Zeile 2: arbeitet man viel bequemer
Zeile 3: und sicherer als mit einem
Zeile 4: klassischen C-Array.
0
```

Man kann ein **vector**-Objekt per Konstruktor-Parameter auch mit vorgegebener Größe erzeugen, um die Anzahl der zeitaufwändigen Speicheranforderungen zu reduzieren. In folgendem Beispiel ist außerdem zu sehen, dass Elemente mit einem Standardtyp automatisch initialisiert werden:

Quellcode-Segment	Ausgabe
`vector<int> iv(3);` `for (int i=0; i < iv.size(); i++) cout << iv[i] << " ";`	`0 0 0`

Zur Eignung der **vector**-Klassen halten wir fest:

- Der wahlfreie Zugriff ist ungefähr genauso schnell wie bei einem C-Array.
- Das Anhängen neuer Element ist jederzeit möglich.
- Das Einfügen neuer Elemente ist zwar möglich, aber sehr zeitaufwändig. Wenn diese Operation oft benötigt wird, sind die **list**-Klassen überlegen (siehe Abschnitt 9.5).

9.2 Iteratoren

Die STL-Algorithmen beschränken sich auf folgende Anforderungen an eine zu verarbeitenden Container-Klasse:

- Über den Elementen existiert eine **Anordnung**, die durch eine **Iterator-Klasse** hergestellt wird.
  In der Regel besitzen die STL-Container eine eingeschachtelte Iterator-Klasse (innerhalb der Container-Klasse definiert) mit dem (klein geschriebenen!) Namen **iterator**, woraus sich z.B. beim Erzeugen eines Iterator-Objektes für einen **vector<int>** - Container folgende Syntax ergibt:
  ```
  vector<int>::iterator pos;
  ```
  Vor der Scope-Operator steht der Name der „umhüllenden“ Container-Klasse, danach der Klassenname **iterator**; beide zusammen stehen für einen Konstruktor der Iterator-Klasse.
- Ein Objekt der Iterator-Klasse kann wie ein Pointer **dereferenziert** werden und zeigt dann auf ein Element der Container-Klasse, z.B.:
  ```
  cout << "in iv gefunden: " << *pos << endl;
  ```
  Die Container-Klasse muss in der Lage sein, Iteratoren zu liefern, die auf spezielle Positionen ihrer Elementenreihe zeigen:
  - einen Iterator, der auf das erste Element zeigt (Start-Iterator)
  - einen Iterator, der auf die Speicherposition hinter dem letzten Element zeigt (Terminierungs-Iterator)
- Ein Iterator-Objekt kann **inkrementiert** werden und zeigt danach auf das nächste Container-Element, z.B.:
  ```
  for(;pos != ende && *pos != gesucht; pos++);
  ```
  In dieser **for**-Schleife wird auf eine Initialisierungs-Vorschrift verzichtet. Die Inkrementierungsvorschrift wirkt auf ein Iterator-Objekt, wie gleich anhand des vollständigen Beispiels zu sehen ist.
  Zu der oben geforderten Ordnung über den Container-Elementen leistet also die Inkrementierungslogik der Iteratoren den entscheidenden Beitrag.
- Zwei Iterator-Objekte können über die Operatoren == und **!= verglichen** werden.

Es ist erstaunlich, wie viele Algorithmen mit dem skizzierten Minimalwissen über die zu verarbeitenden Container formuliert werden können. Meist sind die STL-Algorithmen nach dem selben Muster entworfen wie das folgende Template für Funktionen, die einen Container mit Elementen beliebigen Typs nach einem bestimmten Wert durchsuchen:

```
template<class IteratorTyp, class ElememtTyp>
IteratorTyp suche(IteratorTyp pos, IteratorTyp ende, const ElememtTyp& gesucht)
{
  for(;pos != ende && *pos != gesucht; ++pos);
  return pos;
}
```

In der **for**-Schleife werden die Container-Elemente mit Hilfe des Iterators von Anfang bis Ende durchlaufen. In der Terminierungs-Vorschrift finden zwei Vergleiche statt:

- `pos != ende`
  Hier werden zwei Iteratoren vergleichen, was nach obigen Voraussetzungen möglich ist. Die Suche soll enden, sobald der in jedem Schleifendurchgang inkrementierte „Lauf-Iterator“ den Wert des Terminierungs-Iterators erreicht hat, wenn also das Ende der Elementenreihe erreicht ist.
- `*pos != gesucht`
  Hier werden zwei Werte des Container-Elemententyps verglichen, in der folgenden Anwendung der Funktions-Schablone z.B. zwei **int**-Werte.

Am Ende der **for**-Schleife zeigt der Iterator `pos` entweder auf die Fundstelle oder auf die Position hinter dem letzten Element. Der Aufrufer muss also den als Funktionswert erhaltenen Iterator mit dem Terminierungs-Iterator vergleichen, um das Ergebnis richtig zu verstehen:

```
void main()
{
  const int nele = 5;
  vector<int> iv;
  for (int i = 1; i <= nele; i++)
    iv.push_back(i);
  vector<int>::iterator it = suche(iv.begin(), iv.end(), 2);
  if (it != iv.end())
    cout << "in iv gefunden: " << *it << endl;
}
```

Wie das Beispiel weiter zeigt, werden die von STL-Algorithmen benötigten Start- und Terminierungs-Iteratoren von den Methoden **begin()** und **end()** geliefert, die in jeder STL-Container-Klasse vorhanden sind.

Die STL-Algorithmen können also nur deshalb mit allen Containern arbeiten, weil diese über ihre Iteratoren eine einheitliche Schnittstelle zur Verfügung stellen. Wichtige Unterschiede in der Container-Architektur (z.B. zwischen einem dynamischen Array und einer verketteten Liste) werden durch die Abstraktionsleistung der Iterator-Klassen neutralisiert.
Damit wird es auch möglich, in einer Anwendung den Container-Typ ohne großen Aufwand zu wechseln.
Zudem ist der gesamte Iteratoren-Ansatz kompatibel mit der klassischen Array- und Pointer-Technik, so dass viele STL-Algorithmen auch auf Arrays angewendet werden können. Bekanntlich funktioniert in C ein Arrayname als Pointer auf das erste Element, womit er als Start-Iterator taugt. Durch Pointer-Arithmetik gewinnt man leicht einen Ersatz für den Terminierungs-Iterator. Das Inkrementieren und Derefererenzieren klappt mit den klassischen Pointern ohnehin, so dass sich aus obiger Funktions-Schablone sofort `suche()`-Funktionen für C-Arrays (mit beliebigem Elementtyp) ergeben, z.B.:

```
void main()
{
  const int nele = 5;
  int ia[nele] = {1,2,3,4,5};
  int* ip = suche(ia, ia+nele, 4);
  if (ip != ia+nele)
    cout << "in ia gefunden: " << *ip << endl;
}
```

Auch das Klassen-Template **basic_string** erfüllt die Anforderungen an einen STL-Container, wobei die Iteratoren über die STL-konformen Methoden **begin()** und **end()** geliefert werden, z.B.:

```
void main()
{
  string str = "12345";
  char* c = suche(str.begin(), str.end(), '2');
  if (c != str.end())
    cout << "in str gefunden: " << *c << endl;
}
```

Jeder Iterator kann über den Operator „++“ inkrementiert, über die Operatoren „==“ bzw. „!=“ verglichen sowie dereferenziert werden. Es gibt in der STL etliche Iterator-Sorten mit erweiterten Fähigkeiten. Die so genannten **reversiblen Container** bieten z.B. auch einen **reverse_iterator**, mit dem man den Container rückwärts durchqueren kann (vom letzten Element bis zum ersten). Reversibel sind z.B. die sequentiellen Container **vector**, **list** und **deque**.
Weitere Iterator-Details erfahren Sie z.B. bei Eckel (2000, Band II, S. 159ff).

9.3 Algorithmen

Wer ohne Wissen über Templates und Iteratoren einen Algorithmus für *c* verschiedene Container-Architekturen (z.B. dynamisches Feld, verkettete Liste, Schlüssel-Wert-Tabelle) und *t* verschiedene Element-Datentypen realisieren möchte, hat immerhin $c \cdot t$ Methoden-Überladungen zu implementieren. *Mit* Templates und Iteratoren kommt man hingegen mit einer einzigen Implementierung aus.

Nachdem bisher mit der selbst gestrickten Funktions-Schablone `suche()` die Logik der STL-Algorithmen demonstriert wurde, soll nun endlich der echte STL-Algorithmus **find()** vorgestellt werden:

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

void main()
{
  const int nele = 5;
  int ia[nele] = {1,2,3,4,5};
  vector<int> iv(ia, ia+nele);

  int* pos = find(iv.begin(), iv.end(), 2);
  if (pos != iv.end())
    cout << "in iv gefunden: " << *pos << endl;
}
```

Zu dem Programm ist anzumerken:

- Um die STL-Algorithmen verwenden zu können, muss man die Header-Datei **algorithm** inkludieren.
- Der **vector<int>**-Container wird mit Hilfe eines **int**-Arrays initialisiert, wobei im Konstruktor des Containers wiederum Iteratoren-Technik zum Einsatz kommt.
- An Stelle des beschwerlich zu definierenden **vector<int>**-Iterator-Objektes kann auch ein **int**-Pointer die von **find()** gelieferte Trefferadresse aufnehmen.

Bei Eckel (2000, Band I, S. 285ff) findet sich ein umfangreicher Katalog der STL-Algorithmen (und -Methoden), wobei u.a. folgende Kategorien unterschieden werden:

- **Füllen und Generieren**
  Die hier eingeordneten Algorithmen erzeugen Werte und/oder füllen einen Bereich eines Containers mit Werten. Im folgenden Beispiel wird ein **vector<int>**-Objekt über den Algorithmus **generate_n()** mit Fibonacci-Zahlen[1] gefüllt:

[1] Was Fibonacci-Zahlen sind, ergibt sich aus der Implementation der Funktion `fibo()`.

Quellcode	Ausgabe

`#include <iostream>`
`#include <vector>`
`#include <algorithm>`
`using namespace std;`
 	
`int fibo()`
`{`
`  static int a = 0, b = 1, c;`
`  c = a + b;`
`  a = b;`
`  b = c;`
`  return a;`
`}`
 	
`void main()`
`{`
`  const int nele = 10;`
`  vector<int> iv;`
`  generate_n(back_inserter(iv), nele, fibo);`
`  for(int i= 0; i < nele; i++)`
`  cout << iv[i] << " ";`
`  cout << endl;`
`}`
	`1 1 2 3 5 8 13 21 34 55`

Zu diesem Programm ist anzumerken:

- **generate_n()** hat eine andere Signatur als die bisher betrachteten Algorithmen. Als erster Parameter wird ein **back_insert_iterator** übergeben, den die Methode **back_inserter()** erzeugt. Über den zweiten Parameter erfährt **generate_n()**, wie viele Elemente eingefügt werden sollen.
- Als dritter Aktualparameter wird ein **Zeiger auf eine Funktion** übergeben. Diese wird von **generate_n()** bei jeder Iteration aufgerufen, um ein neues Container-Element zu liefern. Bald werden wir mit der gleichen Aufgabe so genannte Funktionsobjekte befassen, die ein höheres Arbeitstempo vorlegen.

- **Sequenzen manipulieren**
  Bereiche von Container-Elementen werden kopiert, rotiert, invers oder zufällig angeordnet usw.
- **Suchen und Ersetzen**
- **Bereiche vergleichen**
- **Elemente entfernen**
- **Sortieren**
- **Alle Elemente eines Bereichs einer Operation unterwerfen**
- **Numerische Algorithmen**

Hat man für eine Aufgabe sowohl eine Container-Methode als auch einen generischen Algorithmus zur Verfügung, dann ist von der Methode in der Regel eine bessere Performanz zu erwarten, weil sie besser an Spezifika des Containers angepasst ist.

Bei über 60 generischen Algorithmen und zahlreichen generischen Container-Klassen wird man wohl für sehr viele Anwendungen in der STL günstige Entwicklungsbedingungen finden, ohne sich auf proprietäre Lösungen stützen zu müssen.

9.4 Funktionsobjekte

Im eben vorgestellten **find()**-Algorithmus werden Treffer über den Identitätsoperator festgestellt. Um auch andere Kriterien realisieren zu können (z.B. Suche nach der ersten ungeraden Zahl), muss ein Algorithmus noch allgemeiner entworfen werden. Eine in der STL bevorzugte Strategie besteht darin, den fixierten Identitätsoperator durch so genanntes **Funktionsobjekt** zu ersetzen, das an Stelle des Vergleichswertes übergeben wird. Hierbei handelt es sich um ein Objekt aus einer Klasse, die den Funktionsaufruf-Operator überladen hat, so dass ein Funktionsobjekt syntaktisch genauso benutzt werden kann wie eine Funktion (vgl. Lippman 2000, S. 85f; Eckel 2000, Band 2, S. 263ff). Um dieses nicht ganz triviale Zusammenspiel zu veranschaulichen, definieren wir eine eigene Funktionsklasse und setzen sie in einem eigenen Algorithmus ein.

```
class Odd
{
  public:
    bool operator() (int x) const
    { return ((x % 2) != 0); }
};
```

Wendet man auf ein Objekt der Klasse `Odd` den Funktionsaufruf-Operator „**()**“ an, so kommt dies einem Methodenaufruf gleich, wobei ein **int**-Parameter zu übergeben ist. Die Methode liefert einen boolschen Funktionswert, wobei (im Unterschied zu früher vorgestellten Operator-Überladungen) keine Elementvariablen beteiligt sind. In folgendem Beispiel wird ein `Odd`-Objekt erzeugt und dann durch den überladenen „**()**“-Operator beauftragt, für die Zahl 3 zu überprüfen, ob sie ungerade ist:

Quellcode-Fragment	Ausgabe
`Odd od;` `cout << od(3) << endl;`	`1`

In einer modifizierten Version der Funktions-Schablone `suche()` wird als dritter Parameter (nach den beiden Iteratoren) ein Funktionsobjekt erwartet:

```
template<class IteratorTyp, class FunctionClass>
IteratorTyp suche(IteratorTyp pos, IteratorTyp ende, FunctionClass pred)
{
  for(;pos != ende && !pred(*pos); ++pos);
  return pos;
}
```

Seine als Operator-Überladung realisierte Methode wird auf das vom ersten Iterator referenzierte Container-Element angewendet.
Beim Aufruf einer konkreten Instanz der Funktions-Schablone `suche()` wird per Konstruktor ein unbenanntes `Odd`-Objekt übergeben. In der folgenden Hauptfunktion wird außerdem ein `Odd`-Funktionsobjekt zusammen mit dem STL-Algorithmus **find_if()** eingesetzt:

Quellcode-Fragment	Ausgabe

`void main()`
`{`
`  const int nele = 5;`
`  int ia[nele] = {2,4,8,4,5};`
`  vector<int> iv(ia, ia+nele);`
 	
`// Funktionsobjekt als Parameter für den`
`// eigenen suche()-Algorithmus`
`  int *pos = suche(iv.begin(), iv.end(), Odd());`
`  if (pos != iv.end())`
`    cout << "in iv gefunden: " << *pos << endl;`
 	
`// Funktionsobjekt als Parameter für den`
`// STL-Algorithmus find_if()`
`  pos = find_if(iv.begin(), iv.end(), Odd());`
`  if (pos != iv.end())`
`    cout << "in iv gefunden: " << *pos << endl;`
`}`
	`in iv gefunden: 5`
	`in iv gefunden: 5`

Funktionsobjekte mit der von `Odd` demonstrierten Bauart (ein Argument, boolscher Funktionswert) bezeichnet man als *Prädikate*. Weitere Funktionsobjekt-Typen beschreibt z.B. Eckel (2000, Band II, S. 264f). Wie man sich leicht vorstellen kann benötigt z.B. der STL-Algorithmus **sort()** Funktionsobjekte mit zwei Argumenten. Im folgenden Programm wird zum *absteigenden* Sortieren der Elemente eines **vector<int>**-Objektes ein Funktionsobjekt aus der STL-Klasse **greater<int>** verwendet:

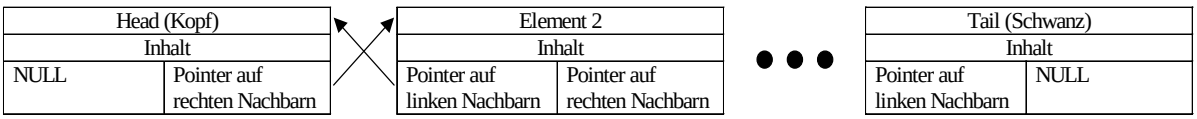
Quellcode	Ausgabe

`#include <algorithm>`
`#include <functional>`
`#include <vector>`
`#include <iostream>`
`using namespace std;`
 	
`void main()`
`{`
`  const int nele = 5;`
`  int ia[nele] = {2,4,8,3,5};`
`  vector<int> iv(ia, ia+nele);`
`//aufsteigendes Sortieren`
`  sort(iv.begin(), iv.end());`
`  cout << "aufsteigend: ";`
`  for(int i = 0; i < nele; i++)`
`    cout << iv[i];`
`//absteigendes Sortieren`
`  sort(iv.begin(), iv.end(), greater<int>());`
`  cout << "\nabsteigend:  ";`
`  for(i = 0; i < nele; i++)`
`    cout << iv[i];`
`}`
	`aufsteigend: 23458`
	`absteigend:  85432`

Im zweiten **sort()**-Einsatz wird über den Konstruktor-Aufruf **greater<int>** ein unbenanntes Funktionsobjekt erzeugt, das innerhalb von **sort()** zum Vergleichen der Elemente herangezogen wird.
Um die eingebauten STL-Funktionsklassen verwenden zu können, muss die Header-Datei **functional** inkludiert werden.

9.5 Listen

Zwar sind **list** und **vector** sequentielle STL-Container mit sehr ähnlichen Schnittstellen, doch gibt es markante Unterschiede bzgl. der geeigneten Einsatzbereiche. Bei einer verketteten Liste sind wahlfreie Zugriffe sehr aufwändig, und auf Offset-Operator hat man wohl aus diesem Grund verzichtet. Andererseits ist das Einfügen oder Entfernen von Elementen auch innerhalb einer Liste kein Problem, während bei einem **vector** in dieser Situation der Speicher mühsam neu organisiert werden muss.
Während **vector**-Elemente im Speicher hintereinander liegen, so dass beim wahlfreien Zugriff nur einfache Pointer-Arithmetik erforderlich ist, nehmen die **list**-Elemente im Speicher individuelle Plätze ein. Jedes Element besitzt einen Vorwärts- und einen Rückwärts-Zeiger, wodurch die sequentielle Struktur hergestellt wird.



Beim Einfügen eines neuen Elementes muss kein altes Element seinen Platz wechseln; es sind lediglich einige Pointer neu zu setzen.

Im folgenden Beispielprogramm wird in einem **list<string>**-Objekt sukzessive ein Satz aufgebaut:

- Die Methode **push_back()** zum Einfügen neuer Elemente am Ende des Containers kennen wir schon von **vector**.
- Die bei Vektoren nicht vorhandene Methode **push_front()** dient dazu, am *Anfang* einer Liste ein neues Element einzufügen.
- Bevor man mit der **insert()**-Methode innerhalb einer Liste ein neues Element ergänzt, muss man einen Iterator geeignet inkrementieren.

Quellcode	Ausgabe

`#include <algorithm>`
`#include <list>`
`#include <string>`
`#include <iostream>`
`#include <iterator>`
`using namespace std;`
 	
`void main()`
`{`
`  list<string> lis;`
`  lis.push_back("jagt");`
`  lis.push_back("im");`
`  lis.push_back("durch");`
`  lis.push_back("Bayern.");`
`  ostream_iterator<string> ostr (cout, " ");`
`  copy(lis.begin(), lis.end(), ostr);`
`  cout << endl;`
`  lis.push_front("Franz");`
`  copy(lis.begin(), lis.end(), ostr);`
`  cout << endl;`
`  list<string>::iterator iter = lis.begin();`
`  for (int i=0; i<3; i++) ++iter;`
`  lis.insert(iter, "Taxi");`
`  copy(lis.begin(), lis.end(), ostr);`
`  cout << endl;`
`}`
	`jagt im durch Bayern.`
	`Franz jagt im durch Bayern.`
	`Franz jagt im Taxi durch Bayern.`

Es bleibt noch zu erklären, wie mit dem **copy()**-Algorithmus die Elemente eines Containers komfortabel zum Standardausgabeobjekt **cout** befördert werden. Er benötigt dazu einen **ostream-iterator**, der an **cout** gebunden ist. Im Beispiel wird der Konstruktor gebeten, nach jedem ausgegebenen Element noch ein Leerzeichen einzufügen.

9.6 Schlüssel-Wert-Tabellen

Unter einem **map**-Container kann man sich eine zweispaltige Tabelle mit Schlüssel-Wert-Paaren vorstellen, wobei gilt:

- Die Schlüsselwerte kommen *nicht* mehrfach vor.
- Die Tabelle wird automatisch nach den Schlüsselwerten aufsteigend sortiert.

In folgendem Beispielprogramm werden aus einer Textdatei nacheinander alle Wörter (durch Leerzeichen getrennt) in ein pufferndes **string**-Objekt gelesen und dann in einen **map<string, int>**-Container eingefügt. Dies geschieht mit dem Offset-Operator, wobei der der gepufferte **string** als Index dient. Der folgende Ausdruck

```
wc[buffer]++
```

fügt nötigenfalls ein neues Schlüssel-Wert-Paar ein (Achtung beim Lesen!) und erhöht den Wert bei einem bereits vorhandenen Paar. Nach dem Lesen der Datei enthält die Tabelle alle aufgetretenen Wörter mit ihrer Häufigkeit.

Quellcode	Ausgabe

```
#pragma warning(disable : 4786)
#include <map>
#include <string>
#include <fstream>
#include <iostream>
using namespace std;

void main()
{
  map<string,int> wc;
  ifstream source("text.txt");
  if (!in)
  {
    cout << "Eingabedatei nicht gefunden!";
    exit(1);
  }
  string buffer;
  while (source >> buffer)
    wc[buffer]++;

  for(map<string,int>::iterator it=wc.begin(); it != wc.end(); ++it)
    cout << it->first  << "\t" << it->second << endl;
}
```

```
Baum    1
Huhn    1
Hund    3
Katze   4
Maus    5
```

Die **pragma**-Direktive am Anfang fordert den Compiler auf, die Warnung mit der Nummer 4786 zu unterdrücken. Diese informiert darüber, dass ein Bezeichner in den Debug-Informationen auf 255 Zeichen reduziert worden sei. Weil STL-Bezeichner (z.B. aufgrund der Template-Technik) schnell auf stattliche Länge anschwellen, hat obiges Programm immerhin 77 Exemplare der Meldung 4786 hervorgebracht, zwischen denen Fehler und andere Warnungen kaum noch auszumachen waren.
Um **map**-Container verwenden zu können, muss man die Header-Datei **map** inkludieren.

Die folgenden Tabelle enthält für **vector**-, **list**- und **map**-Container wichtige Leistungsmerkmale:

Container	*Index*	*Wahlfreier Zugriff*	*Element einfügen oder entfernen*
vector	Ganzzahlen	sehr schnell	langsam
list	nein	langsam	schnell
map	beliebiger Typ	langsam	schnell

9.7 Mengen

Ein **set**-Container enthält eine Menge unterscheidbarer und automatisch sortierter Elemente. Interessante Operationen sind das Einfügen und Entfernen von Elementen sowie die Existenzprüfung für ein fragliches Element.
Wir erweitern das Wörter zählende **map**-Beispielprogramm (siehe Abschnit 9.6) um eine Negativliste zu ignorierender Wörter, realisiert über einen **set<string>**-Container. In diesen Container werden die Wörter „Hund“, „Katze“ und „Maus“ per **insert()**-Merthode eingefügt. Für die Kandidaten-Wörter wird per **count()**-Methode festgestellt, ob sie in der kritischen Menge enthalten sind.

Quellcode	Ausgabe

```
#pragma warning(disable : 4786)
#include <set>
#include <map>
#include <string>
#include <fstream>
#include <iostream>
using namespace std;

void main()
{
  map<string,int> wc;
  set<string> hkm;
  hkm.insert("Hund");
  hkm.insert("Katze");
  hkm.insert("Maus");
  ifstream source("text.txt");

  string buffer;
  while (source >> buffer)
    if (!hkm.count(buffer)) wc[buffer]++;

  for(map<string,int>::iterator it=wc.begin(); it != wc.end(); ++it)
    cout << it->first  << "\t" << it->second << endl;
}
```

```
Baum    1
Huhn    1
```

10 Windows-Programme mit den Microsoft Foundation Classes

Wie Sie vermutlich aus jahrelanger Anwendungspraxis wissen, hat Windows (wie auch die meisten anderen aktuellen Betriebssysteme) eine grafik- und fensterorientierte Benutzeroberfläche. Nun müssen Sie noch lernen, mit VC++ die vielen Fenster zum Vorteil der Anwender einzusetzen. Wie Sie bald feststellen werden, spielen dabei *Nachrichten* des Betriebssystems an seine Fenster eine entscheidende Rolle. Aus der Sicht des Anwenders besteht das Betriebsystem hauptsächlich aus Fenstern; aus der Sicht des Programmierers sind Nachrichten mindestens ebenso bedeutsam.
Ihre Aussichten, Windows beherrschen zu lernen, sind glänzend:

- Windows ist in C/C++ geschrieben, so dass diese Programmiersprache über optimale Schnittstellen zum Betriebssystem verfügt.
- Die Windows-Welt der Fenster und sonstigen Objekte schreit geradezu nach einer OOP-Sprache wie C++.
- Sie haben im bisherigen Kursverlauf gute C++ - Kenntnisse erworben.

Trotz dieser günstigen Voraussetzungen verlangt der Umstieg von den bisherigen Konsolenanwendungen auf die mehrfach komplizierteren Windows-Anwendungen ein hohes Maß an Lernbereitschaft und Einsatz.

10.1 Die Windows-Programmierschnittstelle

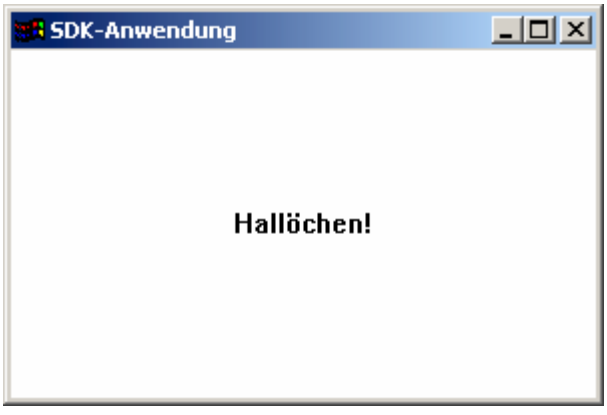
Es erscheint mir sinnvoll, die elementare Architektur und Funktionsweise eines Windows-Programms zu behandeln und anhand eines einfachen Beispiels zu illustrieren, das bewusst auf die Annehmlichkeiten der MFC-Bibliothek verzichtet. Zwar werden bei der Softwareentwicklung mit VC++ und MFC einige Details der Windows-Programmstruktur in MFC-Klassen gekapselt, doch lässt sich nach meinen Erfahrungen der von Anwendungs- und Klassen-Assistenten erzeugte Code für ein MFC-Programm besser verstehen, wenn man die versteckten Details ansatzweise kennt.
Wo uns beim Ausflug in die Welt der „direkten“ Windows-Programmierung Details von dauerhaftem Interesse (auch für die MFC-Programmierung) begegnen, werden wir etwas genauer hinsehen.

10.1.1 Beispielprogramm

Als Beispielprogramm greifen wir den in einer sehr frühen Übungsaufgabe vorgestellten C-Klassiker von Kernighan & Ritchie (1988) wieder auf:

```
#include <stdio.h>

void main()
{
   printf("Hallo, world!\n");
}
```

Nun ist aber die Konsolen-Textausgabe durch einen zeitgemäßen Auftritt im GUI (Graphical User Interface) von Windows zu ersetzen:



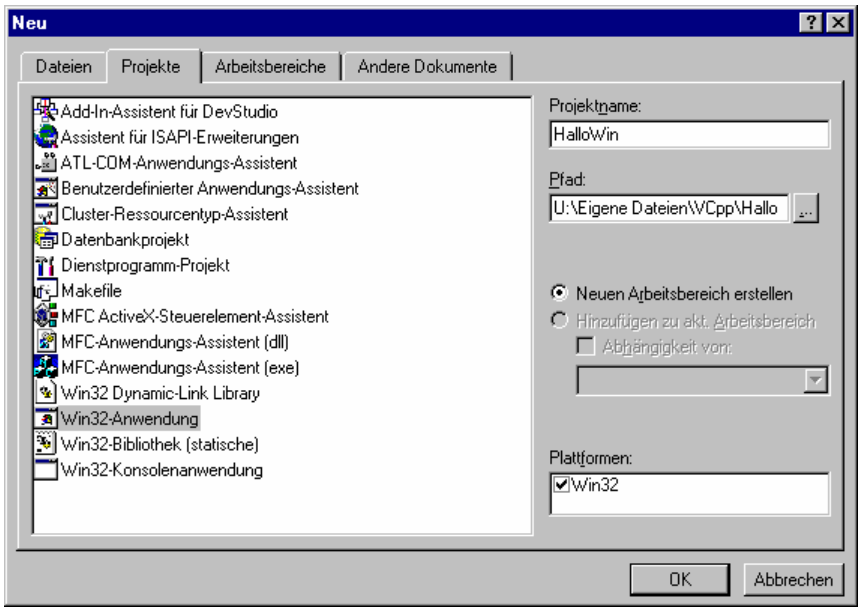
Die Abkürzung *SDK* im Fenstertitel steht übrigens für *Software Developer Kit* und soll den Verzicht auf eine Unterprogramm- bzw. Klassenbibliothek oberhalb des Windows-APIs (*Application Program Interface*) ausdrücken.
Welcher Aufwand bereits hinter einem derart simplen Windows-„Programm“ steckt, zeigt der Quellcode (teilweise übernommen von Petzold, 1996, S. 20ff), mit dem Sie im Verlauf von Abschnitt 10.1 vertraut werden sollen:

```
#include <windows.h>
#include <mmsystem.h> //wird nur für PlaySound() benötigt.

LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
             LPSTR lpszCmdLine, int nCmdShow)
{
  WNDCLASS wc;
  HWND hwnd;
  MSG msg;

  wc.style = CS_HREDRAW | CS_VREDRAW;
  wc.lpfnWndProc = WndProc;
  wc.cbClsExtra = 0;
  wc.cbWndExtra = 0;
  wc.hInstance = hInstance;
  wc.hIcon = LoadIcon(NULL, IDI_WINLOGO);
  wc.hCursor = LoadCursor(NULL, IDC_ARROW);
  wc.hbrBackground = (HBRUSH) (COLOR_WINDOW + 1);
  wc.lpszMenuName = NULL;
  wc.lpszClassName = "MeineFensterSorte";

  RegisterClass(&wc);

  hwnd = CreateWindow(
        "MeineFensterSorte",      //Fenster-Sorte
        "SDK-Anwendung",          //Fenster-Titel
        WS_OVERLAPPEDWINDOW,      //Fenster-Stil
        300,                      //Horizontale Fenster-Startposition
        200,                      //Vertikale Fenster-Startposition
        300,                      //Breite beim Start
        200,                      //Höhe beim Start
        NULL,                     //Handle des Eltern-Fensters
        NULL,
        hInstance,
        NULL);

  ShowWindow(hwnd, nCmdShow);
  UpdateWindow(hwnd);

  while (GetMessage(&msg, NULL, 0, 0))
  {
    TranslateMessage(&msg);
    DispatchMessage(&msg);
  }

  return msg.wParam;
}

LRESULT CALLBACK WndProc(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
  PAINTSTRUCT ps;
  HDC hdc;
  RECT rect;
```

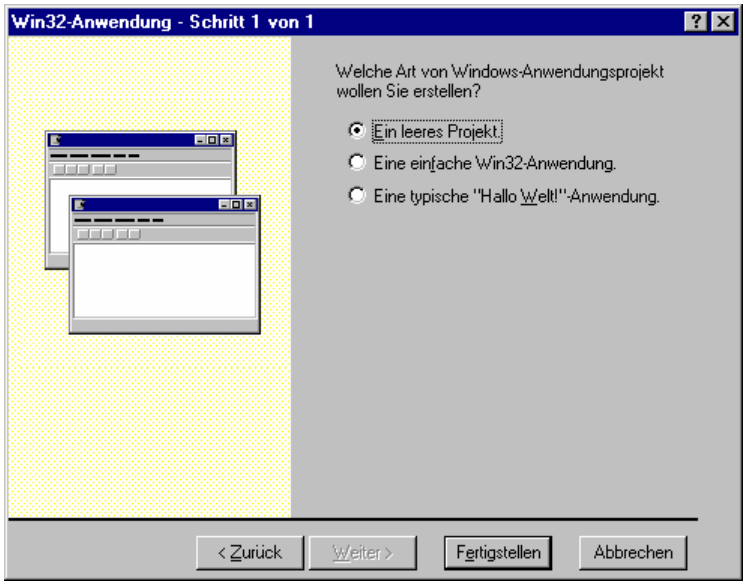
```
   switch (message)
   {
      case WM_CREATE:  PlaySound("ringin.wav", NULL, SND_FILENAME | SND_ASYNC);
                       return 0;
      case WM_PAINT:   PlaySound("ding.wav", NULL, SND_FILENAME | SND_ASYNC);
                       hdc = BeginPaint(hwnd, &ps);
                       GetClientRect(hwnd, &rect);
                       DrawText(hdc, "Hallöchen!", -1, &rect,
                                DT_SINGLELINE|DT_CENTER|DT_VCENTER);
                       EndPaint(hwnd, &ps);
                       return 0;
      case WM_DESTROY: PostQuitMessage(0);
                       return 0;
   }

   return DefWindowProc(hwnd, message, wParam, lParam);
}
```

Wer das Beispielprogramm mit dem Visual Studio erstellen möchte, kann folgendermaßen vorgehen:

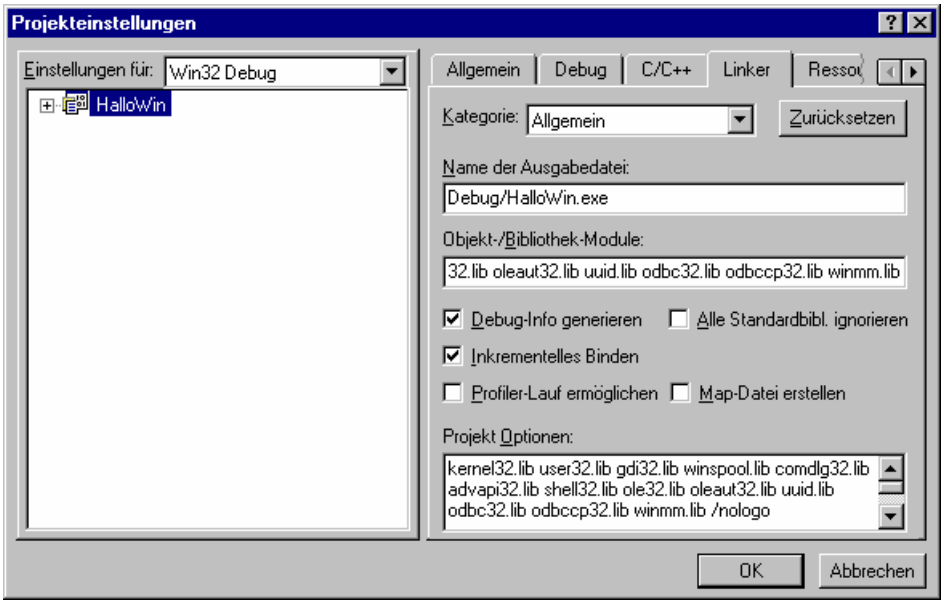
- Über **Datei > Neu** ein neues Projekt vom Typ **Win32-Anwendung** beginnen:



- Im Anwendungsassistenten ein **leeres Projekt** wählen:



- Über **Datei > Neu** eine C++-Quellcodedatei anlegen und den oben diskutierten Quellcode eintragen
- Wegen des im Beispiel eingebauten API-Aufrufs **PlaySound()** müssen Sie nach dem Menübefehl **Projekt > Einstellungen** auf der Registerkarte **Linker** der Dialogbox **Projekteinstellungen** in dem mit **Object-/Bibliothek-Module** überschriebenen Textfeld noch die Bibliotheksdatei **winmm.lib** eintragen:



- Nun sollte sich das Programm erstellen und ausführen lassen.
  Es ist zwar nicht sehr nützlich, sorgt aber z.B. recht geschickt dafür, dass der Grußtext nach einer beliebigen Veränderung der Fenstergröße stets zentriert angezeigt wird.
- Sollten die im Quelltext genannten Sounddateien auf Ihrem System nicht verfügbar sein, können sie durch beliebige andere (am besten möglichst kurze) wav-Dateien ersetzt werden. Auch ohne Sound erfüllt das Beispiel im Wesentlichen seinen Zweck.

10.1.2 Ereignisse und Nachrichten

Ein Konsolenprogramm entscheidet selbst darüber, welche Funktion als Nächste aufgerufen wird, wobei das Programm durchaus flexibel auf Benutzerwünsche, Umgebungsbedingungen (z.B. Speichermangel, Sensorwerte) oder eingelesene Daten reagieren kann. Alle ablaufrelevanten Informationen werden vom Programm aktiv ermittelt, wobei gelegentlich ein zyklisches Abfragen (Polling) zum Einsatz kommt, das bei einer großen Zahl zu überwachender Informationsquellen aufwändig (und unproduktiv) sein kann.
Um seine Aufgaben zu erledigen, ruft ein Konsolenprogramm vielfach Funktionen des Betriebssystems auf, z.B. zum Öffnen einer Datei.
Zwar stellt auch das Windows-API einem Anwendungsprogramm zahlreiche Funktionen zur Verfügung (ca. 1000), die in der Header-Datei **WINDOWS.H** (oder nachgeordneten Dateien) definiert und in diversen Bibliotheksdateien implementiert werden. Jedoch ist für den Ablauf eines Windows-Programms ein **ereignisorientiertes Paradigma** wesentlich, wobei das Betriebssystem als Quelle oder Vermittler von Nachrichten (über Ereignisse) in erheblichem Maße den Ablauf mitbestimmt, indem es **Funktionen des Anwendungsprogramms aufruft**. Wird z.B. ein Fenster vom Benutzer in die Taskleiste geschickt und wieder zurück geholt, dann fordert das Betriebssystem die Anwendung mit der **WM_PAINT**-Nachricht auf, den Fensterinhalt neu zu zeichnen. Man kann ein Windows-Programm als eine Sammlung von Nachrichten-Behandlungsroutinen auffassen. In welcher Reihenfolge diese Routinen ausgeführt werden, hängt wesentlich von den eingehenden Nachrichten ab, also z.B. von den Absichten des Benutzers.

Leicht vereinfachend bzw. idealisierend kann man gegenüber stellen:

Interne Ablaufkontrolle beim Konsolenprogramm	**Externe Ablaufkontrolle** beim Windows-Programm
Ein Konsolenprogramm kümmert sich selbst um die ablaufrelevanten Informationen und entscheidet dann z.B. darüber, welche Funktion als Nächste aufgerufen wird. Zur Informationsbeschaffung sind oft Polling-Aktivitäten erforderlich, die speziell bei simultaner Aktivität mehrer Programme unter einem Multitasking-Betriebssystem eine inakzeptable Vergeudung von Rechenzeit darstellen.	Ein Windows-Programm kann als Sammlung von Ereignisbehandlungsroutinen beschrieben werden. Das Betriebssystem stellt Ereignisse fest (z.B. aufgrund von Benutzeraktivitäten) und ruft passende Behandlungsroutinen auf (sofern vorhanden). Wie gleich zu sehen sein wird, ist das tatsächliche Zusammenspiel zwischen Betriebsystem und Anwendungsprogrammen unter Windows etwas komplizierter.

10.1.3 Fensterprozeduren

Das „Senden einer Nachricht an ein Programm“ hat nichts mit E-Mail etc. zu tun, sondern meint den Aufruf einer Funktion dieses Programms, die per Parameter über den Inhalt der Botschaft informiert wird. Alle Nachrichten werden durch Ganzzahl-Konstanten identifiziert, die in der Header-Datei **windows.h** (oder nachgeordneten Dateien) einen für Programmierer handlichen Namen erhalten, z.B.:

```
#define WM_PAINT                    0x000F
```

Jedes Fenster einer Windows-Anwendung besitzt eine so genannte **Fensterprozedur**[1] (engl.: **window procedure**), um Nachrichten entgegen zu nehmen und darauf zu reagieren.
In unserem Beispiel ist die Fensterprozedur `WndProc()` mit folgendem Prototyp im Einsatz:

```
LRESULT CALLBACK WndProc(HWND hwnd,
                         UINT message,
                         WPARAM wParam,
                         LPARAM lParam);
```

Wir wollen die Bestandteile des `WndProc()`-Prototyps näher betrachten, weil uns hier wichtige Details der Windows-Programmierung erstmals begegnen:

- **LRESULT**
  Ein für Windows-Programme gebräuchlicher, in **WTYPES.H** definierter, Typbezeichner für Ganzzahlen mit 4 Bytes, der synonym zu **long** sowie (unter VC++ 6.0) zu **int** ist:

  ```
  typedef long LONG;
  typedef LONG LRESULT;
  ```

  Der `WndProc()`-Rückgabewert informiert über das Ergebnis der Nachrichtenverarbeitung, hängt also von der konkreten Nachricht ab.

- **CALLBACK**
  Unter Windows sind unterschiedliche Konventionen für das Aufrufen einer Funktion möglich. Damit wird z.B. festgelegt, in welcher Ordnung die Parameter auf dem Stack abgelegt werden und wer den Stack nach Beendigung der Funktion aufräumt (die Funktion selbst oder der Aufrufer). Um vom Projektstandard (einzustellen über **Projekt > Einstellungen > C/C++ > Kategorie = Code Generation > Aufruf-Konvention**) abzuweichen, gibt man in der Funktionsdefinition oder im Prototypen eine alternative Aufrufkonvention an.

[1] Ein Hinweis für Leser mit Erfahrungen in Pascal/Delphi: Die Bezeichnung *Prozedur* hat historische Gründe und impliziert keinesfalls die Abwesenheit eines Rückgabewertes.

Im Beispielprogramm begegnen uns die Aufrufkonventionen **CALLBACK** und **WINAPI**, die beide äquivalent sind zum Standard **__stdcall**, der im Windows-API benutzt wird:

```
#define CALLBACK __stdcall
#define WINAPI __stdcall
```

Weil **__stdcall** vom C-Standard **__cdecl** abweicht, der vom Visual Studio 6.0 bei neuen Projekten als Voreinstellung verwendet wird, tragen wir bei Funktionen, die vom API aufgerufen werden sollen, entweder **__stdcall** oder eine synonyme Bezeichnung ein.

Für das oben herausgestellte Grundprinzip, dass eine Windows-Anwendung Funktionen bereit stellt, die vom Betriebssystem ereignisbezogen aufgerufen werden, sind die Fensterprozeduren typische Beispiele. Die Funktion `WndProc()` wird in unserem Programm definiert, hier aber nirgends aufgerufen. Windows selbst macht jedoch reichlich von dieser Funktion Gebrauch, um Nachrichten an das Fenster zu schicken. Wir haben es also mit einem typischen Beispiel für eine Rückruffunktion (engl.: call back) zu tun. Ihre Rolle wird durch den Aliasnamen CALLBACK für die Aufrufkonvention **__stdcall** im Quellcode deutlich gemacht.

- **HWND, Fenster-Handle**
  Dieser Typ steht für ein **Fenster-Handle**, womit wir auf den nächsten wichtigen Begriff der Windows-Betriebsystemtechnik gestoßen sind: Bei einem **Handle** handelt es sich um einen Zeiger auf ein von Windows verwaltetes Objekt (z.B. auf eine Anwendungsinstanz, ein Fenster, einen Gerätekontext, ein Icon oder einen Cursor). Wenn ein Programm mit dem API-Aufruf **CreateWindow()** (s.u.) ein neues Fenster erzeugt, erhält es als Rückgabewert das zugehörige Fenster-Handle und kann sich später darauf beziehen.
  Man könnte *handle* mit *Zugriffsnummer* oder *Bezug* übersetzen, doch wird in diesem Manuskript die stärker verbreitete englische Bezeichnung bevorzugt.
  Im ersten `WndProc()`-Parameter übergibt der Aufrufer (Windows selbst) das Handle des Fensters, an das die Nachricht adressiert ist. Weil aus einer Fensterklasse (s.u.) mehrere Fenster erzeugt werden können, die eine gemeinsame Fensterprozedur zur Nachrichtenverarbeitung benutzen, ist es durchaus sinnvoll, dem Programmierer über den HWND-Parameter die Chance zu einer differenzierten Nachrichten-Verarbeitung zu geben.
- **UINT, Nachrichten-Nummer**
  Dieser Typbezeichner ist als Abkürzung für **unsigned int** definiert. Über den Parameter **message** erhält die Fensterprozedur eine Nummer, welche die Nachricht eindeutig identifiziert.
- **WPARAM, LPARAM, Nachrichten-Parameter**
  Diese Typbezeichner sind synonym zu **UINT** bzw. **long**:

  ```
  typedef UINT WPARAM;
  typedef long LONG;
  typedef LONG LPARAM;
  ```

  Mit den zugehörigen Parametern **wParam** und **lParam** wird die Nachricht näher spezifiziert, z.B. durch Angabe der Position, an der ein linker Mausklick stattgefunden hat.

Eine Fensterprozedur kann z.B. im Rahmen einer **switch**-Anweisung auf die eingehenden Nachrichten reagieren. In unserem Beispiel werden auf diese Weise drei Nachrichten bearbeitet:

- **WM_CREATE**
  Die Fensterprozedur wird informiert, dass es Zeit ist für eventuelle Initialisierungsarbeiten.
- **WM_PAINT**
  Die Fensterprozedur wird informiert, dass ein Teil ihres Klientenbereichs invalidiert wurde, also neu zu zeichnen ist (siehe Abschnitt 10.1.5).
- **WM_DESTROY**
  Das Fenster erfährt, dass es auf Anordnung des Benutzers geschlossen wird.

Windows enthält gleich **zwei** Mechanismen, um Nachrichten an ein Fenster zu übermitteln:

- **Direktzustellung (sending)**
  Dabei wird die Fensterprozedur direkt aufgerufen.
- **Nachrichten-Warteschlange der Anwendung (posting)**
  Windows erzeugt für jedes Programm eine **Nachrichten-Warteschlange** (engl.: **message queue**), in die Nachrichten an eines der Fenster dieser Anwendung eingereiht werden können. Die **WinMain()**-Funktion der Anwendung holt in ihrer **Nachrichtenschleife** (engl.: **message loop**) die Nachrichten mit der Funktion **GetMessage()** aus der Warteschlange und verschickt sie mit der Funktion **DispatchMessage()** an die jeweils zuständige Fensterprozedur. Hier wird also der Anwendung ein Mitspracherecht darüber eingeräumt, wann gewisse Nachrichten an ihre Fenster ausgeliefert werden sollen.

Wie die beiden Nachrichtenwege genutzt werden, ist letztlich die Sache des Betriebssystems. Grob kann man sich merken, dass Nachrichten aufgrund von Aktionen des Benutzers (z.B. **WM_KEYDOWN**, **WM_LBUTTONDOWN**) über die Nachrichtenschleife laufen, alle anderen direkt versandt werden. Auf die eine oder andere Weise führen Ereignisse, die das Fenster betreffen, zum Aufruf ihrer Fensterprozedur mit entsprechenden Nachricht-Parametern.

Eine Fensterprozedur kann ihrerseits Nachrichten in die Warteschlange der Anwendung setzen. Dies ist sinnvoll bei Neuigkeiten, die ein Fenster auf direktem Weg von Betriebssystem erfahren hat und die auch für die Anwendung relevant sind. In unserem Beispielprogramm wird die Anwendung durch die Fensterprozedur per **PostQuitMessage()** darüber informiert, dass ihr einziges Fenster zerstört wurde (die Nachricht **WM_DESTROY** erhalten hat), so dass es nun Zeit zum Quittieren ist:

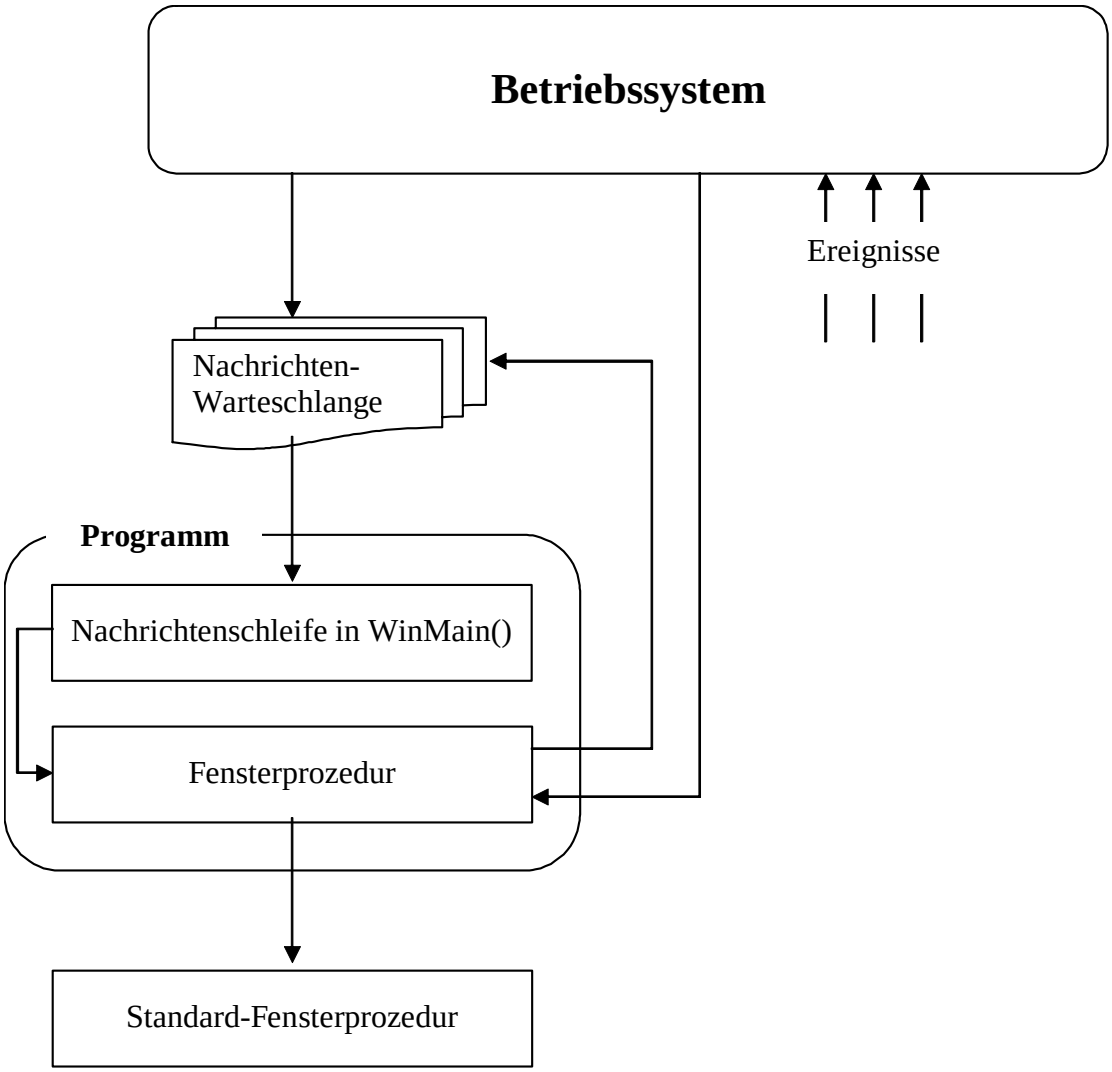
```
case WM_DESTROY:  PostQuitMessage(0);
                  return 0;
```

Eine bereits laufende Nachrichtenbearbeitung wird durch eine neue Nachricht in der Regel *nicht* unterbrochen, es sei denn, die neue Nachricht wurde beim Bearbeiten der Alten erzeugt (z.B. via API-Aufruf). Um für diesen Ausnahmefall gerüstet zu sein, muss die Fensterprozedur **reentrant** sein. Es darf also kein Problem auftreten (z.B. durch überschriebene Variablenwerte), wenn die Fensterprozedur erneut gestartet wird, während ein vorheriger Aufruf noch nicht abgearbeitet ist.

Man kann kaum von jeder Fensterprozedur erwarten, *alle* potentiellen Nachrichten (ca. 200) zu berücksichtigen. Nachrichten, die sie nicht selbst bearbeiten möchte, darf eine Fensterprozedur mit der API-Funktion **DefWindowProc()** an eine Standard-Fensterprozedur (des Betriebssystems) weiterleiten, was unsere Beispielprozedur im Rahmen ihrer **return**-Anweisung tut:

```
return DefWindowProc(hwnd, message, wParam, lParam);
```

In der folgenden Abbildung wird die Nachrichtenverarbeitung bei *einer* Anwendung mit *einem* Fenster (folglich einer Fensterprozedur) grob skizziert:



10.1.4 Aufgaben der Funktion *WinMain()*

Die Beschreibung der Windows-Programmierung konzentrierte sich bisher auf Ereignisse bzw. Nachrichten und deren Behandlung in Fensterprozeduren. Nun findet sich im Quelltext unseres Beispielprogramms sehr wohl mit der Funktion **WinMain()** eine Entsprechung für die Hauptfunktion **main()** der Konsolenprogramme. **WinMain()** beschränkt sich zwar auf „administrative Aufgaben“ und überlässt den Fensterprozeduren die eigentliche Programmlogik, ist aber trotzdem als Einsprungpunkt einer Windows-Anwendung unverzichtbar.

10.1.4.1 Startparameter einer Windows-Anwendung

In der Parameterliste von **WinMain()** zeigt sich, welche Informationen eine Windows-Anwendung beim Start vom Betriebssystem erhält:

```
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                   LPSTR lpszCmdLine, int nCmdShow)
```

Die Parameter haben folgende Bedeutung:

- **hInstance**
  Unter Windows kann ein Programm durchaus mehrfach gestartet werden, wobei jede einzelne Instanz beim Start über den **hInstance**-Aktualparameter ihrer **WinMain()**-Funktion ein eindeutiges Handle erhält, das eine ähnliche Rolle spielt wie eine Prozessnummer. **HINSTANCE** steht wie alle Handle-Datentypen für einen Pointer (nähere Informationen im Anhang).
- **hPrevInstance**
  Dieser Parameter ist in Win32-Betriebssystemen bedeutungslos und systematisch gleich 0.
- **lpszCmdLine**
  Dieser Parameter enthält ggf. einen Zeiger auf einen C-String mit Kommandozeilenparametern. Z.B. sorgt das folgende Kommando am Anfang einer **WinMain()**-Funktion dafür, dass der Sound in **ding.wav** genau dann erklingt, wenn das Programm mit dem Kommandozeilen-String „x“ gestartet wird:

  ```
  if (strcmp(lpszCmdLine, "x") == 0)
   PlaySound("ding.wav", NULL, SND_FILENAME | SND_ASYNC);
  ```

  Der Präfix *lpsz* im Parameternamen steht für *long pointer to zero-terminated string*. **LPSTR** ist ein in Windows üblicher Aliasname für den altbekannten Datentyp **char***.
- **nCmdShow**
  Über diesen Parameter wird ein Programm informiert, wie es initial erscheinen soll (z.B. minimiert). Die möglichen Werte sind in der Datei WINUSER.H definiert, z.B.:

  ```
  #define SW_SHOWMINNOACTIVE  7
  ```

Wie bei einer Fensterprozedur, so wird auch bei **WinMain()** eine spezielle Aufrufkonvention angegeben, wobei **WINAPI** derzeit ein Aliasname für **__stdcall** ist (siehe oben).

10.1.4.2 Fensterklasse definieren

Windows-Anwendungen besitzen in der Regel mindestens ein Fenster, und zu jedem Fenster benötigt Windows eine Beschreibung, die zahlreiche optische Attribute (z.B. Hintergrundfarbe, Ikon) und eine Adresse der Fensterprozedur für die Nachrichtenverarbeitung enthält. Daher macht sich die Funktion **WinMain()** zunächst daran, für das Fenster der Anwendung eine Beschreibung in einer Strukturvariablen vom Typ **WNDCLASS** zu hinterlegen.
In der folgenden Tabelle sind die wichtigsten Elementvariablen der **WNDCLASS**-Struktur (definiert in **WINUSER.H**) erläutert:

<table>
<tr><th colspan="2">Variablen-</th><th rowspan="2">Erläuterung</th></tr>
<tr><th>typ</th><th>name</th></tr>
<tr><td>UINT</td><td>style</td><td>Inhalt: Legt Design- und Verhaltensmerkmale des Fensters fest, wobei man den Wert über binär-Oder-verknüpfte Konstanten (definiert in WINUSER.H) festlegt, z.B.:
<table>
<tr><th>Name</th><th>Wert</th><th>Bedeutung</th></tr>
<tr><td>CS_VREDRAW</td><td>0x0001</td><td>Gesamten Klientenbereich (s.u.) neu zeichnen nach vertikaler Größenänderung.</td></tr>
<tr><td>CS_HREDRAW</td><td>0x0002</td><td>Gesamten Klientenbereich (s.u.) neu zeichnen nach horizontaler Größenänderung.</td></tr>
</table>
Nähere Erläuterungen zu den CS-Konstanten, die auch in MFC-Programmen analog verwendet werden, finden sich in Prosise (1999, S. 120).
Bsp.: <code>wc.style = CS_HREDRAW | CS_VREDRAW;</code></td></tr>
<tr><td>WNDPROC</td><td>lpfnWndProc</td><td>Inhalt: Legt die Fensterprozedur für die aus der Fensterklasse erzeugten Fenster fest.
Der Variablennamen-Präfix steht für long pointer to function.
Bsp.: <code>wc.lpfnWndProc = WndProc;</code></td></tr>
<tr><td>int</td><td>cbClsExtra</td><td>Inhalt: In dem von Windows für die Fensterklasse verwalteten Datenbereich können Extra-Bytes festgelegt werden, die einem Programm für eigene Zwecke zur Verfügung stehen.
Der Variablennamen-Präfix steht für count of bytes.
Bsp.: <code>wc.cbClsExtra = 0;</code></td></tr>
<tr><td>int</td><td>cbWndExtra</td><td>Inhalt: In dem von Windows für Fenster der definierten Klasse verwalteten Datenbereich können Extra-Bytes festgelegt werden, die einem Programm für eigene Zwecke zur Verfügung stehen.
Der Variablennamen-Präfix steht für count of bytes.
Bsp.: <code>wc.cbWndExtra = 0;</code></td></tr>
<tr><td>HINSTANCE</td><td>hInstance</td><td>Inhalt: Hier wird ein Handle auf die Anwendungsinstanz eingetragen, welche die Fensterklasse definiert. Man übernimmt dazu den ersten WinMain()-Parameter. Damit kann Windows die von einem Fenster (z.B. per PostQuitMessage()) abgeschickten Nachrichten in die Warteschlange der zugehörigen Anwendung einreihen.
Bsp.: <code>wc.hInstance = hInstance;</code></td></tr>
<tr><td>HICON</td><td>hIcon</td><td>Inhalt: Man muss festlegen, welches Icon in der Systemmenü-Ecke sowie in der Taskleiste erscheinen soll. Das dazu nötige Icon-Handle besorgt man sich z.B. mit der API-Funktion LoadIcon(). Über ihren zweiten Parameter kann man den Namen eines (in WINUSER.H) vordefinierten Icons angeben.
Bsp.: <code>wc.hIcon = LoadIcon(NULL, IDI_WINLOGO);</code></td></tr>
<tr><td>HCURSOR</td><td>hCursor</td><td>Inhalt: Man muss festlegen, welche Gestalt der Mauszeiger über den Fenstern annehmen soll. Das dazu nötige Cursor-Handle besorgt man sich z.B. mit der API-Funktion Lo-</td></tr>
</table>

Variablen-		Erläuterung
typ	name	
		adCursor(). Über ihren zweiten Parameter kann man den Namen eines (in **WINUSER.H**) vordefinierten Cursors angeben. Bsp.: `wc.hCursor = LoadCursor(NULL, IDC_ARROW);`
HBRUSH	hbrBackground	Inhalt: Mit dieser Elementvariable kann die Hintergrundfarbe festgelegt werden. Im Zusammenhang mit dem GDI werden weitere Bürsten-Optionen behandelt. Mit der API-Funktion **GetStockObject()** kann man auf vorgefertigte Bürsten zugreifen. Der Variablennamen-Präfix steht für *handle to a brush.* Bsp.: `wc.hbrBackground=(HBRUSH) (COLOR_WINDOW + 1);`
LPCSTR	lpszMenuName	Inhalt: Mit dieser Elementvariable kann das Menü zu einer Fensterklasse festgelegt werden. Bsp.: `wc.lpszMenuName = NULL;`
LPCSTR	lpszClassName	Inhalt: Mit dieser Elementvariablen erhält die Fensterklasse einen Namen, der später beim Erzeugen von Fenstern zu verwenden ist. Bsp.: `wc.lpszClassName = "Meine Fenster-Sorte";`

Die neue Fenstersorte wird dem Betriebssystem durch Aufruf der Funktion **RegisterClass()** bekannt gemacht und kann dann verwendet werden.

Obwohl diese vom Windows-API diktierte Vorgehensweise für die Programmiersprache C konzipiert wurde, weist sie eine gewisse Ähnlichkeit mit einer Klassendefinition im Sinne der OOP auf. Allerdings fehlen einer Windows-Fensterklasse essentielle Merkmale einer OOP-Klasse, z.B. die Methoden sowie die Möglichkeit zum (ver)erben. Es ist also streng zwischen Windows-Klassen und C++ - Klassen zu unterscheiden.

Aktuelle Windows-Versionen unterstützen neben **WNDCLASS** und **RegisterClass()** noch die leicht erweiterten Varianten **WNDCLASSEX** und **RegisterClassEx()** (siehe z.B. Petzold 1996, S. 32ff).

10.1.4.3 Fenster erzeugen und anzeigen

Bevor endlich das erste Fenster einer Anwendung auf dem Bildschirm erscheint, sind noch folgende „Verwaltungsakte“ seitens der Funktion **WinMain()** erforderlich:

a) CreateWindow()

WinMain() erzeugt ein Fenster aus der gewünschten Klasse, wobei über **CreateWindow()**-Parameter auch etliche Attribute für jedes Fenster individuell festgelegt werden können:

<table>
<tr><th colspan="2">Parameter-</th><th rowspan="2">Erläuterung</th></tr>
<tr><th>typ</th><th>name</th></tr>
<tr><td>LPCTSTR</td><td>lpClassName</td><td>Inhalt: String mit dem Namen einer registrierten Fensterklasse. LPCTSTR ist ein Pointer auf einen konstanten TCHAR-String. TCHAR steht für …
den Typ wchar_t (ein 16 Bit - Unicode-Zeichen), wenn das Symbol _UNICODE definiert ist,
den Typ char, wenn _UNICODE nicht definiert ist.
Bsp.: <code>"Meine Fenster-Sorte"</code></td></tr>
<tr><td>LPCTSTR</td><td>lpWindowName</td><td>Inhalt: String mit der Fenster-Beschriftung (Titelzeile)
Bsp.: <code>"SDK-Anwendung"</code></td></tr>
<tr><td>DWORD</td><td>dwStyle</td><td>Inhalt: Analog zu den Fensterklassenstilen sind zahlreiche Design- und Verhaltensmerkmale für einzelne Fenster definiert (in WINUSER.H), z.B.:
<table>
<tr><th>Name</th><th>Bedeutung</th></tr>
<tr><td>WS_OVERLAPPED</td><td>Überlappend</td></tr>
<tr><td>WS_CAPTION</td><td>Fenster hat Titelzeile und Rahmen</td></tr>
<tr><td>WS_MINIMIZEBOX</td><td>Symbol zum Minimieren vorhanden</td></tr>
<tr><td>WS_MAXIMIZEBOX</td><td>Symbol zum Maximieren vorhanden</td></tr>
<tr><td>WS_SYSMENU</td><td>Systemmenü vorhanden</td></tr>
<tr><td>WS_OVERLAPPEDWINDOW</td><td>Kombination von WS_OVERLAPPED, WS_CAPTION, WS_SYSMENU, WS_THICKFRAME, WS_MINIMIZEBOX und WS_MAXIMIZEBOX</td></tr>
</table>
Der Variablennamen-Präfix dw steht für DWORD, und dies ist eine Abkürzung für unsigned long. (32-Bit-Ganzzahl ohne Vorzeichen)
Bsp.: <code>WS_OVERLAPPEDWINDOW</code></td></tr>
<tr><td>int</td><td>x</td><td>Inhalt: Initiale horizontale Position des Fensters
Bsp.: <code>300</code></td></tr>
<tr><td>int</td><td>y</td><td>Inhalt: Initiale vertikale Position des Fensters
Bsp.: <code>200</code></td></tr>
<tr><td>int</td><td>nWidth</td><td>Inhalt: Initiale Breite des Fensters
Bsp.: <code>300</code></td></tr>
<tr><td>int</td><td>nHeight</td><td>Inhalt: Initiale Höhe des Fensters
Bsp.: <code>200</code></td></tr>
<tr><td>HWND</td><td>hWndParent</td><td>Inhalt: Handle zum Elternfenster
Bsp.: <code>NULL</code></td></tr>
<tr><td>HMENU</td><td>hMenu</td><td>Inhalt: Handle zum Menü
Bsp.: <code>NULL</code></td></tr>
<tr><td>HINSTANCE</td><td>hInstance</td><td>Inhalt: Handle auf die Instanz der Anwendung
Bsp.: <code>hInstance</code></td></tr>
</table>

Parameter-		Erläuterung
typ	name	
LPVOID	lpParam	Inhalt: Pointer auf Kreationsparameter Der Pointer könnte auf Daten zeigen, die später angesprochen werden sollen. Bsp.: `NULL`

Bei Erfolg meldet **CreateWindow()** ein Fenster-Handle zurück für weitere Aktionen mit dem Fenster, das bisher nur eine vom Betriebssystem verwaltete Datensammlung ist.

b) ShowWindow() und UpdateWindow()

ShowWindow() sorgt für die Anzeige des Fensters, dessen Handle als erster Parameter übergeben wird, wobei der im zweiten Parameter übergebene Anzeigemodus (z.B. minimiert, normal) zum Einsatz kommt. Meist gibt man den vom Betriebssystem beim Aufruf der Funktion **WinMain()** im Parameter **nCmdShow** übermittelten Anzeigemodus weiter.

Der exakte Ablauf gestaltet sich so:

- Beim Aufruf von **ShowWindow()** schickt Windows (auf direktem Weg) die Nachricht **WM_CREATE** an die Fensterprozedur. Unser Beispiel reagiert auf diese Nachricht durch Abspielen einer WAV-Datei.
- Nachdem die Fensterprozedur per **return** die Kontrolle zurück gegeben hat, wird das Fenster angezeigt.
- Nun endet auch der Aufruf von **ShowWindow(),** und die Funktion **WinMain()** setzt ihre Initialisierungsarbeiten fort.

Nach dem **ShowWindow()**-Aufruf ist das neue Fenster zwar auf dem Bildschirm zu bewundern, jedoch ist sein Klientenbereich (siehe unten) noch leer. Im Beispielprogramm wird Windows per API-Funktion **UpdateWindow()** veranlasst, der Fensterprozedur die Nachricht **WM_PAINT** (wieder auf direktem Wege) zu senden. Unser Beispiel spielt daraufhin eine WAV-Datei ab und malt anschließend den Klientenbereich des Fensters. Wie sie das tut, wird nach einigen Bemerkungen zur Nachrichtenschleife in Abschnitt 10.1.5 behandelt.
Wenn Sie im Beispielprogramm den API-Aufruf **UpdateWindow()** entfernen, werden Sie keine sichtbare Änderung im Programmablauf feststellen, weil unmittelbar anschließend die Nachrichtenschleife einsetzt und dabei auch eine **WM_PAINT**–Nachricht zur Fensterprozedur gelangt. Es kann trotzdem sinnvoll sein, den Klientenbereich des Fensters per **UpdateWindow()**–Aufruf zu füllen, weil eventuell vor Inbetriebnahme der Nachrichtenschleife noch andere Arbeiten anstehen.

10.1.4.4 Nachrichtenschleife betreiben

Nachdem die Funktion **WinMain()** das Anwendungsfenster erzeugt und angezeigt hat, führt sie bis zum Ende des Programms die Nachrichtenschleife aus:

```
while (GetMessage(&msg, NULL, 0, 0))
{
   TranslateMessage(&msg);
   DispatchMessage(&msg);
}

return msg.wParam;
```

GetMessage() holt Nachrichten aus der anwendungsspezifischen Warteschlange, **TranslateMessage()** übersetzt Tastatur-Nachrichten in Zeichen-Nachrichten (z.B. bei Tastenkombinationen für Menübefehle), und **DispatchMessage()** leitet Nachrichten an das betroffene Fenster weiter.
Die Nachrichten werden mit Hilfe einer Variablen vom Strukturtyp **MSG** ausgetauscht, deren Elemente uns größtenteils schon als Parameter der Fensterprozedur begegnet sind (siehe oben):

```
struct MSG
{
    HWND   hwnd;
    UINT   message;
    WPARAM wParam;
    LPARAM lParam;
    DWORD  time;
    POINT  pt;
};
```

Zu welchem Fenster die Nachricht gelangen soll, geht aus dem MSG-Elementvariablen **hwnd** hervor.
Neu im Vergleich zur Parameterliste der Fensterprozedur sind:

Typ	Name	Erläuterung
DWORD	time	Zeit, als die Nachricht in die Warteschlange gelangte
POINT	pt	Mauskoordinaten zum Zeitpunkt, als die Nachricht in die Warteschlange gelangte

Während die Fensterprozedur eine per **DispatchMessage()** übergebene Nachricht bearbeitet, wartet **MinMain()**, so dass also keine weiteren Nachrichten aus der Warteschlange abgerufen werden.

Wenn **GetMessage()** die Nachricht **WM_QUIT** aus der Warteschlange fischt, liefert die Funktion eine 0 zurück und beendet damit die **while**-Schleife. Danach beendet sich auch die Hauptfunktion und meldet den letzten **wParam**-Wert als Returncode an das Betriebsystem.

10.1.5 GDI, Gerätekontexte und WM_PAINT

Für die Ausgabe von Text- und Grafikinformationen bietet Windows dem Programmierer sehr leistungsfähige Routinen, die zusammenfassend als **GDI** (**Graphic Device Interface**) bezeichnet werden. Ein herausragendes Windows-Designmerkmal ist die **Geräteunabhängigkeit** seiner Grafikschnittstelle: Beim Erstellen einer Ausgabe ist es unerheblich, ob diese in einem Bildschirmfenster und/oder auf einem Drucker landen soll. Windows sorgt zusammen mit den Gerätetreibern für die Umsetzung. Da Windows auf einem GUI (Graphical User Interface) basiert, werden Text- und Grafikausgaben einheitlich behandelt.
Eine konkrete GDI-Ausgabefunktion benötigt zahlreiche Informationen (Ausgabegerät, Ausgabeursprung, Anzeigefläche, Hintergrundfarbe, Schriftart, Textfarbe, ...), so dass eine entsprechende Parameterliste eine ebenso stattliche wie unpraktische Länge erreichen würde. Statt dessen stehen in Windows vorbereitete Konfigurationspakete bereit, die als **Gerätekontext** (engl.: device context) bezeichnet werden. Bei jeder GDI-Ausgabeanweisung muss ein Gerätekontext angegeben werden, was windows-typisch über ein Handle geschieht, das zuvor beim Betriebssystem zu beantragen ist. Diese Reglementierung von Schreibrechten ist unumgänglich, weil sich mehrere Programme einen Bildschirm teilen.

Nun haben wir über den Gerätekontext gelernt:

- Er hält dem Programmierer lästige Konfigurationsarbeiten vom Hals.
- Über ein Handle für einen Gerätekontext erwirbt ein Anwendungsprogramm das Recht, in einen genau festgelegten Bildschirmbereich zu schreiben. Schreibversuche außerhalb der zulässigen Region werden abgeblockt.
  Nach getaner Arbeit sollte das Handle unbedingt wieder zurück gegeben werden, weil sonst Speicher im Windows-Systemcache vergeudet wird.

Zur Illustration wird aus der Fensterprozedur des SDK-Beipspielprogramms die Passage zur Behandlung der Nachricht **WM_PAINT** wiedergegeben. Der Gerätekontext-Handle `hdc` wird mit **BeginPaint()** erworben, in **DrawText()** benutzt und mit **EndPaint()** wieder zurück gegeben:

```
   . . .
PAINTSTRUCT ps;
HDC hdc;
RECT rect;

case WM_PAINT:   hdc = BeginPaint(hwnd, &ps);
                 GetClientRect(hwnd, &rect);
                 DrawText(hdc, "Hallöchen!", -1, &rect,
                          DT_SINGLELINE|DT_CENTER|DT_VCENTER);
                 EndPaint(hwnd, &ps);
                 return(0);
   . . .
```

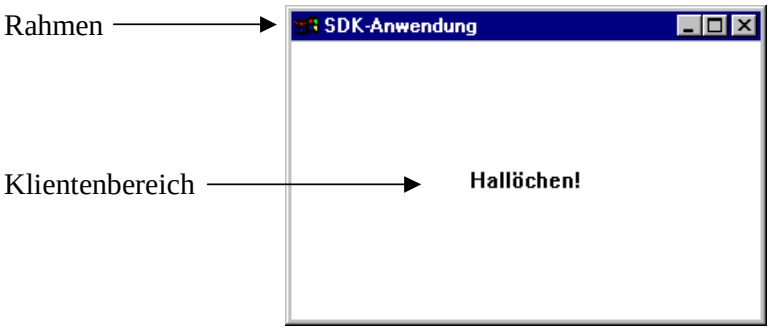
Der **BeginPaint()**–Aufruf hat noch folgende Effekte:

- Der Fensterhintergrund wird unter Verwendung des aktuellen Fensterklassen-Pinsels zurück gesetzt (gelöscht).
- Der Klientenbereich des Fensters wird wieder validiert.
  Ohne diese Wirkung der **BeginPaint()**-Funktion würde Windows fortfahren, dem Fenster **WM_PAINT**-Nachrichten zu schicken.
- In der **PAINTSTRUCT**-Struktur werden Informationen bereit gestellt, die beim Zeichnen des Klientenbereichs von Nutzen sein können. Z.B. ist in der Elementvariablen **rcPaint** vom Typ **RECT** festgehalten, welcher Bereich des Fensters tatsächlich invalidiert worden war. Eventuell kann sich also die Fensterprozedur darauf beschränken, *einen Teil* des Fensters neu zu zeichnen, um Zeit zu sparen.
  Im Beispielprogramm werden die **PAINTSTRUCT**-Informationen nicht genutzt.

Merken brauchen Sie sich die zuletzt beschriebenen API-Funktionen nicht, weil im objektorientierten MFC-Kontext (siehe unten), dem wir letztlich den Vorzug geben, eine leicht modifizierte Vorgehensweise erforderlich ist.

Von großem (und dauerhaftem) Interesse im Zusammenhang mit der Bildschirmausgabe ist die Aufgabenverteilung bei der Verwaltung der Fensterinhalte:

- Jedes Fenster ist für seinen *Klientenberich* selbst verantwortlich und muss ihn bei Bedarf neu zeichnen. Über die Notwendigkeit, den Klientenbereich (partiell) neu zu zeichnen, wird die Fensterprozedur per **WM_PAINT** informiert.
- Windows kümmert sich um den *Fensterrahmen* und dortige Bedienelemente (z.B. Systemmenü).



Wird z.B. ein Fenster überdeckt, speichert Windows keinesfalls den verlorenen Inhalt, sondern fordert das wieder an die Oberfläche gelangte Fenster mit der Nachricht **WM_PAINT** auf, seinen Klientenbereich neu zu zeichnen. Sie können sich vorstellen, dass dies nicht gerade selten passiert.

10.1.6 Notationskonventionen in Windows-Programmen

Für Windows-Programme in C(++) haben sich einige Notationsgewohnheiten eingebürgert, deren Kenntnis das Lesen und die Weitergabe von Quellcode erleichtert:

- **Funktionsnamen**
  Funktionsnamen beginnen mit einem Großbuchstaben. Bestehen sie aus mehreren Wörtern, verwendet man zur optischen Gliederung *keine* Unterstriche, sondern Großbuchstaben an den Wortanfängen.
  Beispiele: **CreateWindow()**, **PostQuitMessage()**
- **Konstanten**
  Die in Header-Dateien für Windows zahlreich definierten Konstanten werden in Übereinstimmung mit der allgemeinen C(++)-Praxis komplett *groß* geschrieben, um Verwechslungen mit Variablennamen zu vermeiden. Häufig beginnen die Namen mit einer durch Unterstrich abgetrennten Kennzeichnung der Familienzugehörigkeit, wobei z.B. *WM* für *Window Message* und *DT* für *Draw Text* stehen.
  Beispiele: **WM_PAINT**, **DT_VCENTER**
- **Typbezeichner**
  Oben sind schon etliche windows-spezifische Typbezeichner erläutert worden. Hinter manchen Exemplaren stecken wesentliche Bausteine für Windows-Programme (z.B. **MSG**), andere stellen nur Synonyme für C/C++ - Bezeichner dar (z.B. **UINT** statt **unsigned integer**). Im Anhang finden Sie eine Liste mit oft benötigten Windows-Typbezeichnungen.
- **Variablennamen**
  Die auf den legendären Microsoft-Programmierer Charles Simonyi zurück gehende **Ungarische Notation** schlägt vor, Variablennamen mit einem klein geschriebenen Kürzel für den Datentyp beginnen zu lassen, damit diese Information bei der Quellcode-Lektüre leicht verfügbar ist. Recht verbreitet sind folgende Kürzel (nach Petzold 1996, S,. 30):

Präfix	**Datentyp**
c	char
by	BYTE (unsigned char)
n	short
i	int
x, y	int, als x- bzw. y-Koordinate
b, f	BOOL (int); *f* steht für *flag*
w	WORD (unsigned short)
l	LONG (long)
dw	DWORD (unsigned long)
fn	Funktion

Präfix	Datentyp
s	string
sz	nullterminierter String (C-String)
h	Handle
p	Pointer

Bei Bedarf können auch mehrere Präfixe hintereinander gesetzt werden.
Beispiele: `hInstance`, `lpszCmdLine`

10.1.7 Zusammenfassung

Es soll kurz zusammengefasst werden, welche Grobstruktur ein traditionelles Windows Programm besitzt:

- **Funktion WinMain()**
  Diese Funktion wird beim Programmstart vom Betriebssystem aufgerufen. Sie definiert Fensterklassen, erzeugt Fenster und betreibt die Nachrichtenschleife.
- **Fensterprozeduren**
  Jede Fensterklasse besitzt eine CALLBACK-Funktion, die vom Betriebssystem (direkt oder über die Nachrichtenwarteschlange der Anwendung) aufgerufen wird, um Nachrichten an Fenster dieser Klasse abzusetzen. In den Fensterprozeduren steckt die eigentliche Anwendungslogik von Windows-Programmen.

Umfangreiche Informationen über das Win32-API finden Sie im Hilfesystem zum Visual Studio 6.0 (MSDN-Library) über

Inhalt > Platform-SDK

10.1.8 Übungsaufgaben zu Abschnitt 10.1

1) Verändern Sie unser Beispielprogramm so, dass es die Sounddatei **ding.wav** abspielt, wenn der Benutzer einen *minimierten* Start veranlasst.
Tipp: - In diesem Fall hat der **WinMain()**-Parameter **nCmdShow** den Wert **SW_SHOWMINNOACTIVE**.

2) Verändern Sie unser Beispielprogramm so, dass über seinem Fenster ein Uhrglas-Cursor angezeigt wird.
Tipp: - Der Uhrglas-Cursor kann über die Konstante **IDC_WAIT** angesprochen werden.

3) Machen Sie die Tätigkeit der Nachrichtenschleife akustisch wahrnehmbar.
Tipps: - Bauen Sie einen geeigneten **PlaySound()**-Aufruf in die Schleife ein.
- Verzögern Sie die Schleife, z.B. mit Hilfe der folgenden `Sleep()`-Funktion:

```
void sleep( clock_t wait )
{
   clock_t goal;
   goal = wait + clock();
   while( goal > clock() );
}
```

Wegen der hier verwendeten **clock()**-Funktion muss die Header-Datei **time.h** inkludiert werden.

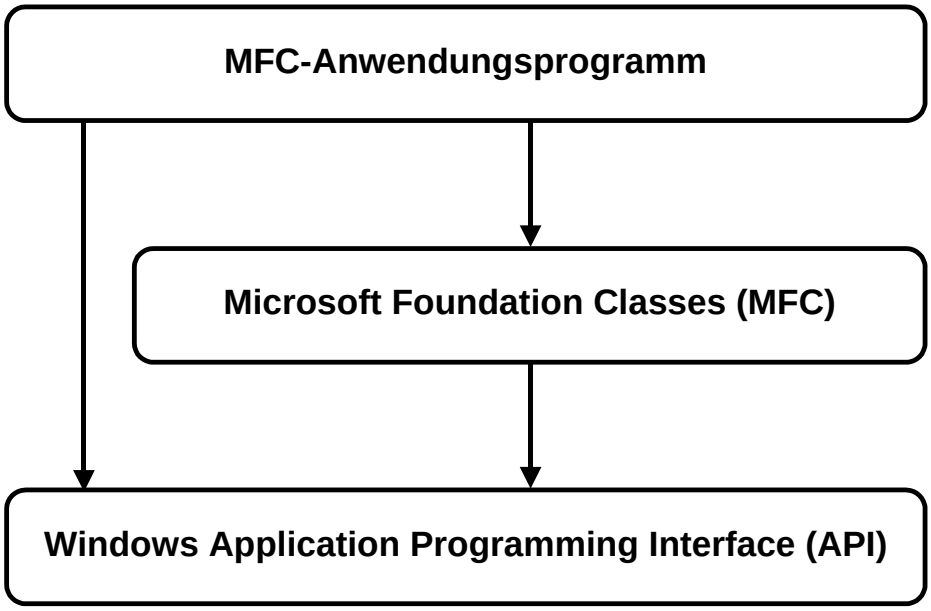
4) Kommentieren Sie in der Fensterprozedur des Beispielprogramms die Funktion **QuitPostMessage()** aus, über die das Fenster nach Eintreffen der Nachricht **WM_DESTROY** die Anwendung zum

Beenden aufordert. Starten Sie das Programm, und schließen Sie (z.B. per **<Alt><F4>**) das Anwendungsfenster. Nun können Sie z.B. mit dem Taskmanager von Windows NT/2000/XP feststellen, dass der Prozess weiterhin aktiv ist.

10.2 Die MFC als Klassenbibliothek und Anwendungsrahmen

Wenngleich das Windows-API (in der Programmiersprache C) prozedural realisiert ist, treffen wir doch vielfach auf eine objekt-*basierte* Arbeitsweise. So können Anwendungsprogramme über ein Handle und definierte Schnittstellen (z.B. Parameterlisten) auf Bildschirmfenster zugreifen und wissen dabei nichts von den Daten, die das Betriebssystem zu einem Fenster verwaltet, und von der Realisation der vielfältigen Fenster-Verhaltensweisen. Damit ist zwar für bestimmte, vorgegebene „Klassen" die Datenkapselung im Sinne der objektorientierten Programmierung weitgehend realisiert, doch ist es z.B. nicht möglich, eine eigene Klasse als Spezialisierung eines Windows-Steuerelementes über die OOP-Vererbung zu gewinnen. Microsoft hat folgerichtigerweise das prozedurale Windows-API, dessen Gründzüge aus dem Jahr 1987 stammen, ab 1992 in eine objektorientierte C++ - Klassenbibliothek „verpackt", damit Windows-Programmierer moderne Prinzipien der Software-Entwicklung nutzen können.

Weil die MFC-Klassen das Windows-API einkapseln, gewinnt man außerdem eine verbesserte Portabilität zwischen verschiedenen Windows-Varianten, die von der MFC unterstützt werden. Eine MFC-Anwendung darf aber auch direkt auf das Windows-API zugreifen, wie gleich in einem Beispiel zu sehen sein wird. Die Beziehungen zwischen dem Windows-API, der MFC-Klassenbibliothek und einer konkreten MFC-Anwendung sind in folgender Abbildung dargestellt:



Die ca. 200 MFC-Klassen stellen mit ihren Elementfunktionen alle wesentlichen Teile des Windows-APIs zur Verfügung, oft mit ähnlichen Bezeichnungen, so dass Umsteiger sich schnell zurecht finden. Mit dem folgenden API-Aufruf hat unser SDK-Beispielprogramm das per Handle ansprechbare Fenster zur Anzeige gebracht:

```
ShowWindow(hwnd, nCmdShow);
```

In der gleich vorzustellenden MFC-Variante des Programms ist das Fenster als Objekt aus der Klasse **CFrameWnd** vertreten, deren Methode **ShowWindow()** die Rolle der gleichnamigen API-Funktion übernimmt:

```
m_pMainWnd->ShowWindow(m_nCmdShow);
```

Das Fensterobjekt der MFC-Anwendung wird von ihrem „Hauptobjekt" (aus der Klasse **CWinApp**) über die Pointervariable **m_pMainWnd** angesprochen.

Hier zeigt sich eine typische Übersetzungsstrategie:

- Ausgangspunkt ist das von Windows zur Verfügung gestellte Fenster, das in der (prozeduralen) Programmiersprache C in API-Funktionsaufrufen über sein Handle angesprochen wird. Um das Fenster auch in objektorientierten C++ - Programmen verwenden zu können, wurde es in die MFC-Klasse **CFrameWnd** „verpackt“.
- Um ein **CFrameWnd**-Objet in eine *andere* MFC-Klasse (z.B. **CWinApp**) einzubauen, haben die MFC-Designer dort eine Zeiger-Elementvariable integriert, die funktional analog zum Fenster-Handle einsetzbar ist.
  Aus dem API-Fenster-Handle wird eine Zeiger-Elementvariable auf ein MFC-Objekt.
- Die Rolle der API-Funktion wird von einer Methode des **CFrameWnd**-Objektes übernommen, die als Bestandteil des Objektes in ihrer Parameterliste natürlich keinen Verweis auf das zu bearbeitende Objekt benötigt, so dass sich die Parameterliste im Vergleich zur API-Funktion entsprechend verkürzt.

Wie in den letzten Ausführungen schon angeklungen ist und bald noch deutlicher zu sehen sein wird, gibt es einige strukturelle Unterschiede zwischen API- und MFC-Programmen, die im wesentlichen aus dem Bemühen der MFC resultieren, Routineaufgaben automatisch im Verborgenen auszuführen (z.B. die Nachrichtenschleife). Damit stellt die MFC nicht nur einige nützliche Klassen bereit, sondern diktiert auch ein Vergleich zum API-Original verändertes **Anwendungsgerüst** (engl.: **application framework**). Die erste objektorientierte Einhüllung des Windows-APIs nannte sich sogar *Application Frameworks*, und im MFC-Quellcode entdecken Sie deren Abkürzung *AFX* noch recht oft als Bestandteil von diversen Bezeichnungen.

Nun wird es Zeit, den Quellcode der MFC-Variante unseres Beispielprogramms zu präsentieren. Wenn Sie selbst damit experimentieren wollen, was sehr zu empfehlen ist, können Sie ein dazu geeignetes VC++ - Projekt folgendermaßen anlegen:

- Legen Sie ein neues Projekt vom Typ **Win32-Anwendung** an.
- Wählen Sie im ersten Schritt des Anwendungsassistenten ein **leeres Projekt**, und lassen Sie es **fertigstellen**.
- Fügen Sie eine Header- und eine Quelltextdatei mit dem unten wiedergegebenen Inhalt ein.
- Aktivieren Sie die MFC-Verwendung für das Projekt, indem Sie nach dem Menübefehl **Projekt > Einstellungen** auf der Registerkarte **Allgemein** der Dialogbox **Projekteinstellungen** in der mit **Microsoft Foundation Classes** überschriebenen Liste den folgenden Eintrag wählen: **MFC in einer gemeinsam genutzten DLL verwenden**.
- Wegen des im Beispiel eingebauten API-Aufrufs **PlaySound()** müssen Sie nach dem Menübefehl **Projekt > Einstellungen** auf der Registerkarte **Linker** der Dialogbox **Projekteinstellungen** in dem mit **Object-/Bibliothek-Module** überschriebenen Textfeld noch die Bibliotheksdatei **winmm.lib** eintragen.

In der Header-Datei werden zwei Klassen definiert:

```
#include <afxwin.h>
#include <mmsystem.h>

class CHalloMFCApp : public CWinApp
{
public:
   BOOL InitInstance();
};
```

```
class CHalloMFCWnd : public CFrameWnd
{
public:
   CHalloMFCWnd();
protected:
   afx_msg int OnCreate(LPCREATESTRUCT lpcs);
   afx_msg void OnPaint();
   DECLARE_MESSAGE_MAP()
};
```

Die Implementierungsdatei ist übersichtlicher (zumindest kürzer) als die SDK-Variante:

```
#include "HalloMFC.h"

CHalloMFCApp halloMFCApp;

BOOL CHalloMFCApp::InitInstance()
{
   m_pMainWnd = new CHalloMFCWnd;
   m_pMainWnd->ShowWindow(m_nCmdShow);
   m_pMainWnd->UpdateWindow();

   return TRUE;
}

BEGIN_MESSAGE_MAP (CHalloMFCWnd, CFrameWnd)
   ON_WM_CREATE()
   ON_WM_PAINT()
END_MESSAGE_MAP()

CHalloMFCWnd::CHalloMFCWnd()
{
   Create(NULL, "Hallo MFC", WS_OVERLAPPEDWINDOW, CRect(300, 200, 600, 400));
}

int CHalloMFCWnd::OnCreate(LPCREATESTRUCT lpcs)
{
   if (CFrameWnd::OnCreate(lpcs) == -1)
      return -1;
   //Hier wird eine API-Funktion aufgerufen
   PlaySound("chord.wav", NULL, SND_FILENAME | SND_ASYNC);
   return 0;
}

void CHalloMFCWnd::OnPaint()
{
   CPaintDC dc(this);
   CRect rect;

   PlaySound("ding.wav", NULL, SND_FILENAME | SND_ASYNC);
   GetClientRect(&rect);
   dc.DrawText("Hallöchen!", -1, &rect, DT_SINGLELINE|DT_CENTER|DT_VCENTER);
}
```

Anhand dieses Beispiels werden Sie nun u.a. kennen lernen:

- mit **CWinApp** und **CFrameWnd** zwei wichtige MFC-Basisklassen für die Entwicklung eigener Anwendungen, aus denen wir das Anwendungsobjekt und ein Fensterobjekt ableiten,
- die MFC-Nachrichten-Tabellen zum Verknüpfen der Windows-Nachrichten mit Methoden der Fensterobjekte,
- den auf der Klasse **CPaintDC** basierenden MFC-Mechanismus zum Beantworten der Windows-Nachricht **WM_PAINT** zum (Wieder)Herstellen des Fensterinhalts.

Zunächst stellen wir fest, dass MFC-Programme die Headerdatei **afxwin.h** inkludieren müssen. Sie enthält mehrere tausend Zeilen und inkludiert ihrerseits weitere Headerdateien, z.B. auch die aus dem SDK-Programm in Abschnitt **Fehler! Verweisquelle konnte nicht gefunden werden.** bekannte Datei **windows.h**.

10.2.1 Anwendungsobjekt und Fensterobjekt

In der sehr übersichtlichen Headerdatei wird für unsere Anwendung die Klasse `CHalloMFCApp` aus der MFC-Basisklasse **CWinApp** abgeleitet. Als einzige Änderung gegenüber der Basisklasse wird Methode **InitInstance()** überschrieben.
In der Implementierungsdatei wird mit:

```
CHalloMFCApp halloMFCApp;
```

ein globales Objekt aus unserer Anwendungsklasse definiert, das wir im Folgenden als **Anwendungsobjekt** bezeichnen wollen. Es hat u.a. folgende Aufgaben:

- Es betreibt (für uns verborgen) in seiner Methode **Run()** die Nachrichtenschleife der Anwendung.
- Durch Überschreiben von **CWinApp**-Methoden können wir unsere Anwendung konfigurieren, z.B. mit **InitInstance()** das Start- und mit **ExitInstance()** das Terminierungsverhalten.

In gewisser Weise entspricht das Anwendungsobjekt der Funktion **WinMain()**, die im Windows-API den Einsprungpunkt für den (vom Programmieren erstellten) Code einer Anwendung bildet (siehe oben), und vom Application Framework der MFC versteckt wird.

Zur Verwendung für das Fenster unserer Anwendung wird in der Header-Datei aus der MFC-Basisklasse **CFrameWnd** die Klasse `CHalloMFCWnd` abgeleitet, die im Folgenden als *MFC-Fensterklasse* unserer Anwendung bezeichnet werden soll. Wo einfach von einer *Fensterklasse* gesprochen wird, geht aus dem Kontext hervor, ob eine *Windows-Fensterklasse* im Sinne von Abschnitt 10.1.4.2 oder eine *MFC-Fensterklasse* gemeint ist.
In der `CHalloMFCWnd`-Definition tauchen neben dem Konstruktor auf:

- Die Methoden **OnCreate()** und **OnPaint()** sind zur Behandlung der Windows-Nachrichten **WM_CREATE** und **WM_PAINT** gedacht, was aus der Namensgebung ersichtlich wird. Wie man leicht ausprobieren kann, erfüllt z.B. die Methode **OnPaint()** ihren Zweck nicht mehr, wenn man das „t“ streicht (in der Deklaration und in der Implementation): Es erscheint zwar keine Fehlermeldung, aber auch kein Fensterinhalt.
  Während es auf die Benennung der Nachrichten-Behandlungsmethoden genau ankommt, dient der Vorsatz *afx_msg* lediglich dazu, den Programmierer an die besondere Rolle der betroffenen Methode zu erinnern. Aufgrund der Definition:

  ```
  #define afx_msg
  ```

  (in der Headerdatei **afxwin.h**) lässt der Precompiler den Präfix rückstandsfrei verschwinden, so dass man ihn auch gleich weglassen könnte.
  Nähere Erläuterungen zu **OnCreate()** und **OnPaint()** folgen in Abschnitt 10.2.3.
- Das Makro **DECLARE_MESSAGE_MAP()** ist an der Verknüpfung von Windows-Nachrichten und Methoden der Fensterklasse beteiligt (siehe Abschnitt **Fehler! Verweisquelle konnte nicht gefunden werden.**).
  Weil es die Klassendefinition um zusätzliche Variablen- und Methodendeklarationen mit zugehörigen Schutzstufen erweitert, sollte es möglichst *am Ende* der Definition stehen, um die unbeabsichtigte Übernahme einer unbekannten Schutzstufe für eigene Variablen/Methoden zu vermeiden.

Selbstverständlich ändert auch die MFC nichts an der für Windows-Programme fundamentalen Ablaufsteuerung durch Ereignisse bzw. Nachrichten. Das entscheidende Leben spielt sich in der Nachrichtenbehandlungsmethoden der Fensterklasse ab.

Als Basis für die Ableitung der Fensterklasse einer Anwendung kommen neben **CFrameWnd** auch noch andere MFC-Klassen in Frage (siehe unten).

Während in traditionellen Windows-Programmen die **WinMain()**-Funktion dafür zuständig ist, ein Fenster zu erzeugen und sichtbar zu machen, wird diese Aufgabe in MFC-Anwendungen folgendermaßen erledigt:

- Das Anwendungsobjekt erzeugt in seiner Methode **InitInstance()**, die dazu vom Application Framework aufgerufen wird, mit dem Operator **new** ein MFC-Objekt aus der anwendungseigenen Fensterklasse. Für das damit entstandene **Fensterobjekt** erhält das Anwendungsobjekt einen Pointer in seiner Elementvariablen **m_pMainWnd** (geerbt von **CWinApp**)
  InitInstance() muss unbedingt in jeder Anwendung überschrieben werden, weil die Methode in der Basisklasse lediglich die Anweisung `return TRUE` enthält.
- Beim Erzeugen des Fensterobjekts wird der zugehörige Konstruktor ausgeführt. Darin wird über die Methode **Create()** der Basisklasse **CFrameWnd** zum MFC-Fensterobjekt ein korrespondierendes Windows-Rahmenfenster erzeugt. Die Methode **Create()** erfüllt zusammen mit dem Operator **new** den selben Zweck wie die API-Funktion **CreateWindow()** in der SDK-Variante des Programms. Im ersten **Create()**-Argument wird **NULL** statt der erwarteten Windows-Fensterklasse angegeben, woraufhin automatisch eine passende Fensterklasse gewählt wird.
  Wer eine eigene Windows-Fensterklasse benötigt, kann diese mit der globalen MFC-Funktion **AfxRegisterWndClass()** definieren und registrieren, um anschließend den Rückgabewert der Funktion (ein Pointer auf einen null-terminierten String mit dem WNDCLASS-Namen) als ersten **Create()**-Parameter zu verwenden.
  Das in der Methode **CHalloMFCApp::InitInstance()** per **new**-Operator auf dem Heap erzeugt Objekt lebt beim Beenden der Methode weiter, was hier auch sehr erwünscht ist.
- Anschließend ruft das **InitInstance()**–Methode des Anwendungsobjektes die Methode **ShowWindow()** des neuen Fensterobjektes auf, um das Fenster sichtbar zu machen.
  Über die Member-Variable **m_nCmdShow** wird dabei der **int**-Parameter **nCmdShow** zum Fenster-Eröffnungsmodus (normal, minmiert etc.) durchgereicht, den die (verborgene) Funktion **MinMain()** beim Start erhalten hat.
  Die Methode **ShowWindow()** entspricht der gleichnamigen API-Funktion, die in der SDK-Variante des Programms an analoger Stelle zum Einsatz kam.
  Schließlich erfüllt die Methode **UpdateWindow()** den selben Zweck wie ihre API-Entsprechung (vgl. Abschnitt 10.1.4.3).

Man kann sich das Geschehen beim Start einer MFC-Anwendung ungefähr so vorstellen:

- Das (globale!) Anwendungsobjekt wird erzeugt.
- Über die (vom Application Framework versteckte) Funktion **WinMain()** kommt deren MFC-Entsprechung **AfxWinMain()** ins Spiel. Diese kopiert wichtige Parameter wie **hInstance** und **nCmdShow** (siehe Abschnitt 10.1.4.1) in Elementvariablen des Anwendungsobjektes und ruft dessen **InitInstance()**-Methode auf.
- **InitInstance()** erzeugt das Fensterobjekt, und dessen Konstruktor erstellt das zugehörige Windows-Rahmenfenster. Anschließend veranlasst **InitInstance()** die Anzeige des Windows-Fensters über Methoden des MFC-Fensterobjekts.
  Die Funktion **InitInstance()** eignet sich übrigens auch sehr gut dazu, anwendungsspezifische Variablen zu initialisieren, was im aktuellen Beispiel allerdings nicht erforderlich ist.

- **AfxWinMain()** ruft die **Run()**-Methode des Anwendungsobjektes auf, welche die Nachrichtenschleife ausführt. Damit gelangen auch die Warteschlangen-Nachrichten (siehe Abschnitt 10.1.2) zum Windows-Anwendungsfenster. Mit Hilfe von MFC-„Hintergrundaktivitäten“ werden die Nachrichten an die zuständigen Behandlungsmethoden des Fensterobjekts übergeben, die dann z.B. für die Ausgabe im Klientenbereich des Anwendungsfensters sorgen.

Zur Beendigung einer Anwendung kommt es, wenn die Methode **Run()** beim Bearbeiten der Nachrichtenschleife die Botschaft **WM_QUIT** findet. Abschließend startet **Run()** dann noch die Methode **ExitInstance()** des Anwendungsobjektes, die bei Bedarf geeignet überschrieben werden muss.

Vermutlich fragen Sie sich, wer eigentlich das in **InitInstance()** per **new**–Operator auf dem Heap erzeugte Fensterobjekt wieder entfernt. Die Antwort ist einerseits beruhigend, zeigt aber andererseits, dass wir neben C++ - und Windows-Kenntnissen noch einiger MFC-Spezialwissen benötigen, um erfolgreich mit diesem Applikations-Gerüst zu arbeiten: Das in **InitInstance()** erzeugte Fensterobjekt löscht sich selbst, sofern seine Klasse von **CFrameWnd** abstammt:

- Die letzte Windows-Nachricht an ein Fenster vor seinem regulären Ende lautet **WM_NCDESTROY**.
- Sie wird in den aus **CWnd** abgeleiteten Klassen von der Methode **OnNcDestroy()** behandelt, die wiederum eine virtuelle Methode namens **PostNcDestroy()** aufruft.
- Die aus **CWnd** abgeleitete Klasse **CFrameWnd** ruft in ihrer **PostNcDestroy()**-Implementation den (zum **new** in **InitInstance()** „antagonistischen“) **delete**-Operator auf:

```
delete this;
```

Weil die MFC-Funktion **AfxWinMain()** eine zentrale Rolle spielt und in wesentlichen Zügen durchaus gut zu verstehen ist, wird sie im folgenden wiedergegeben, u.a. auch wegen der vielen **goto**-Anweisungen, die Microsofts Programmierer an so zentraler Stelle (durchaus zweckmäßigerweise!) hinterlassen haben:

```
int AFXAPI AfxWinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
   LPTSTR lpCmdLine, int nCmdShow)
{
   ASSERT(hPrevInstance == NULL);

   int nReturnCode = -1;
   CWinThread* pThread = AfxGetThread();
   CWinApp* pApp = AfxGetApp();

   // AFX internal initialization
   if (!AfxWinInit(hInstance, hPrevInstance, lpCmdLine, nCmdShow))
      goto InitFailure;

   // App global initializations (rare)
   if (pApp != NULL && !pApp->InitApplication())
      goto InitFailure;

   // Perform specific initializations
   if (!pThread->InitInstance())
   {
      if (pThread->m_pMainWnd != NULL)
      {
         TRACE0("Warning: Destroying non-NULL m_pMainWnd\n");
         pThread->m_pMainWnd->DestroyWindow();
      }
      nReturnCode = pThread->ExitInstance();
      goto InitFailure;
   }
   nReturnCode = pThread->Run();
```

```
InitFailure:
#ifdef _DEBUG
   // Check for missing AfxLockTempMap calls
   if (AfxGetModuleThreadState()->m_nTempMapLock != 0)
   {
      TRACE1("Warning: Temp map lock count non-zero (%ld).\n",
         AfxGetModuleThreadState()->m_nTempMapLock);
   }
   AfxLockTempMaps();
   AfxUnlockTempMaps(-1);
#endif

   AfxWinTerm();
   return nReturnCode;
}
```

Sie finden diese Funktion in der folgenden Datei:

...\Microsoft Visual Studio\VC98\MFC\SRC\WINMAIN.CPP

Damit Sie durch diese **goto**-Besichtigungstour keinen Rückfall in überwundene Programmiertechniken erleiden, betrachten wir auf dem **Klassen**-Registerblatt des Arbeitsbereichsfensters die Struktur unserer objektorientierten MFC-Anwendung:

Sie sehen die beiden Klassen mit ihren Methoden sowie das globale Anwendungsobjekt (Instanz der Anwendungsklasse). Das in `CHalloMFCApp::InitInstance()` per **new**-Operator auf dem Heap angelegte Fensterobjekt wird (als dynamische Variable) hier natürlich nicht angezeigt.

10.2.2 Nachrichtenzuordnungstabelle

Eigentlich wäre es nahe liegend, in MFC-Basisklassen für Objekte, die Nachrichten empfangen können, eine virtuelle Methode für jede potentielle Nachricht zu definieren, die dann in abgeleiteten Klassen entsprechend überschrieben werden könnten. Allerdings hätte diese OOP-Standardtechnik zur Realisation von Polymorphie aufgrund der hohen Anzahl von Nachrichten einen extremen Speicherbedarf zur Folge (wegen sogenannter *vtables*). In der MFC wird das Problem statt dessen über einige komplexe Makros gelöst, z.B.:

```
BEGIN_MESSAGE_MAP (CHalloMFCWnd, CFrameWnd)
   ON_WM_CREATE()
   ON_WM_PAINT()
END_MESSAGE_MAP()
```

In diesem Paket wird vereinbart, dass für Fensterobjekte aus der Klasse `CHalloMFCWnd` (mit Basisklasse **CFrameWnD**) die Windows-Nachrichten **MW_CREATE** und **WM_PAINT** durch die Member-Funktionen **OnCreate()** bzw. **OnPaint()** behandelt werden, wobei die Abbildung der Na-

men festen Regeln folgt(, zu denen es jedoch auch Ausnahmen gibt). Die Makros erzeugen Code für die Elementvariablen und Methoden, um deren Deklarationen das bereits angesprochene Makro **DECLARE_MESSAGE_MAP()** die Definition der anwendungsspezifischen Fensterklasse erweitert.
Sie sollten zwar grundsätzlich wissen, was hinter den Makroaufrufen im Quelltext steckt, müssen sich aber nicht weiter um den Abbildungsmechanismus kümmern. In der Alltagsarbeit mit dem Visual Studio erstellt man MFC-Programme über den Anwendungsassistenten und kann mit Hilfe des Klassenassistenten die Nachrichtenzuordnungstabelle einer Klasse bequem verwalten. Man erweitert sie z.B., indem man die zu behandelnde Windows-Nachricht markiert und auf den Schalter **Funktion hinzufügen** klickt. Daraufhin erledigt der Klassenassistent lästige Routinearbeiten:

- Behandlungs-Methode mit dem korrekten Namen deklarieren (festgelegt durch die Windows-Nachricht)
- Methodenrumpf angelegen
- MFC-Makroeinträge erstellem

10.2.3 Nachrichtenbehandlung

Auch für eine MFC-Anwendung gilt, dass ihre wesentliche Funktionalität in den Nachrichtenbehandlungsroutinen steckt. Ein MFC-Programm benötigt also für jede Fensterklasse geeignete Methoden und eine Nachrichtenzuordnungstabelle. Anschließend werden die elementaren Behandlungsroutinen **OnCreate()** und **OnPaint()** beschrieben.

10.2.3.1 OnCreate()

WM_CREATE ist die erste Nachricht, die einem Fenster (letztlich seiner Fensterprozedur) zugestellt wird. Im MFC-Framework ist sie mit der Methode **CWnd::OnCreate()** zu behandeln. Hier sollten bei ernsthaften Programmen die Membervariablen der Fensterklasse initialisiert werden, wozu in unserem Beispiel kein Anlass besteht. Statt dessen wird die Methode **OnCreate()** durch Abspielen eines Tons wahrnehmbar gemacht, wobei der Aufruf einer API-Funktion (**PlaySound()**) im MFC-Kontext vorgeführt wird.
Wenn Sie für eine eigene Fensterklasse eine **OnCreate()**-Methode definieren und damit die entsprechende Methode der Basisklasse überschreiben, dann sollten Sie (vor allem bei Document-View-Anwendungen, siehe unten) am Anfang den **OnCreate()**-Handler der Basisklasse aufrufen, damit er MFC-seitig erforderliche Initialisierungen vornehmen und deren Erfolg bzw. Misserfolg per Rückgabewert artikulieren kann. Bei einem **OnCreate()**-Rückgabewert von –1 wird die Erstellung des Fensters von Windows abgebrochen.

10.2.3.2 OnPaint()

In der **OnPaint()**-Methode wird zur Bearbeitung der in Windows außerordentlich häufigen **WM_PAINT**-Nachricht ein Objekt der Klasse **CPaintCD** erzeugt. Es stellt einen Gerätekontext (vgl. Abschnitt 10.1.5) zur Verfügung, der über den **CPaintCD**-Konstruktor bzw. -Destruktor automatisch beim Betriebssystem beantragt (per API-Funktion **BeginPaint()**) bzw. wieder freigegeben wird (per API-Funktion **EndPaint()**).
Zu diesem Zweck benötigt der **CPaintDC**-Konstruktor als Parameter einen Pointer auf das betroffenen Fenster-Objekt. Weil in unserem Fall der Konstruktor von einer Methode dieses Fensters aufgerufen wird, kann das Schlüsselwort **this** als Aktualparameter verwendet werden.
Die zur Textausgabe verwendete **CPaintDC**-Methode **DrawText()** entspricht wiederum weitgehend der gleichnamigen API-Funktion, die wir in der SDK-Variante des Programms verwendet haben. Den von **DrawText()** benötigten rechteckigen Ausgabebereich liefert ein **CRect**-Objekt, dessen Koordinaten von der **CWnd**-Methode **GetClientRect()** ermittelt werden.

10.2.4 Notationskonventionen in MFC-Programmen

Uns sind bisher folgende Notations-Gepflogenheiten der MFC-Programmierer begegnet:

- Klassenbezeichnungen beginnen mit einem großen „C“.
- Die Namen von Member-Variablen (also Elementvariablen von Objekten) beginnen mit „m_“.
- Manche MFC-Programmierer kennzeichnen die (grundsätzlich im globalen Gültigkeitsbereich definierten) API-Aufrufe durch einen vorangestellten Scope-Operator (::), um jede Verwechslung mit eventuell vorhandenen Methoden gleichen Namens zu vermeiden, z.B.:

```
::PlaySound("chord.wav", NULL, SND_FILENAME | SND_ASYNC);
```

10.2.5 Übungsaufgaben zu Abschnitt 10.2

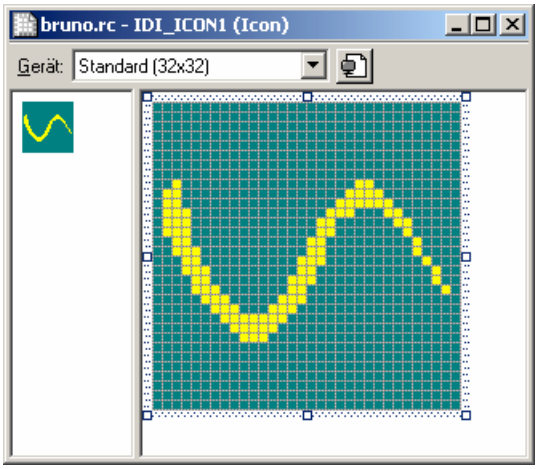
1) Verändern Sie unser MFC-Beispielprogramm aus Abschnitt 10.2 so, dass über dem Fenster ein Uhrglas-Cursor angezeigt wird.
Hinweis: Verwenden Sie die globale Funktion **RegisterWndClass()** im Konstruktor der MFC-Fensterklasse dazu, eine geeignete Windows-Fensterklasse zu definieren und gleichzeitig zu registrieren. Die Parameter von **RegisterWndClass()** entsprechen annähernd den Elementvariablen der API-Struktur **WNDCLASS**, die in Abschnitt 10.1.4.2 vorgestellt wurde. Genauere Angaben finden Sie in der MSDN-Bibliothek (Online-Hilfe).

10.3 Ressourcen

Neben der Ereignisorientierung und der geräteunabhängigen Grafikausgabe ist als weiteres Spezifikum der Windows-Programmierung die Verwendung so genannter **Ressourcen** zu nennen. Darunter versteht man Programmbestandteile, die getrennt vom Quellcode verwaltet und daher prinzipiell von mehreren Anwendungen genutzt werden können. Zu den Ressourcen gehören:

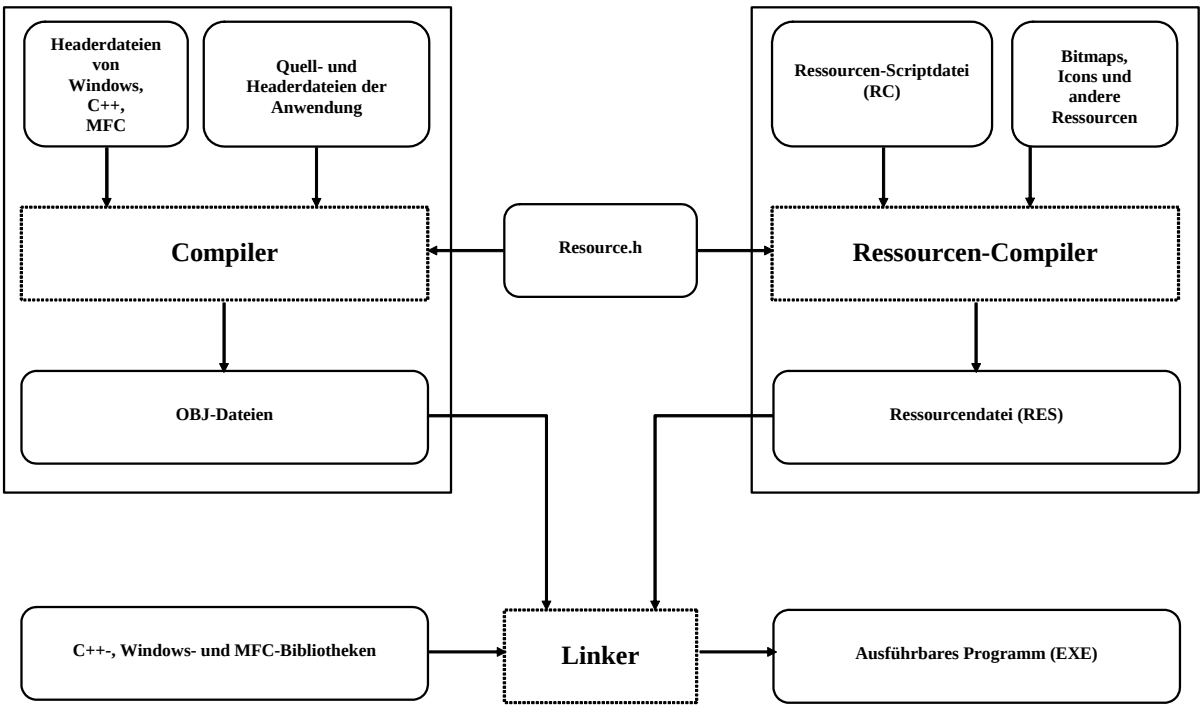
- Dialogfeld-Vorlagen
  Dialogfelder treten in praktisch jeder Windows-Anwendung auf, meist als temporäre Fenster, die z.B. über einen Menübefehl geöffnet werden. Im Dialogfeld kann der Benutzer über standardisierte Steuerelemente (Befehlsschalter, Eingabefelder etc.) entweder seine Wünsche äußern oder Informationen erhalten. Nähere Erläuterungen zur Zweck und zur Verwendung von Dialogfeldern in Windows-Anwendungen sind sicher überflüssig.
  In diesem Abschnitt geht es u.a. darum, wie sich Dialogfelder von anderen Windows-Fenstern unterscheiden, und wie sie mit Hilfe von speziellen Ressourcen, den Dialogfeld-Vorlagen, erstellt werden.
- Menü-Vorlagen
  Analog zu den Dialogbox-Vorlagen erstellt man auch bei Menüs zunächst eine Vorlage, die dann im Quelltext einem Fenster zugewiesen werden kann (z.B. in der **Create()**-Methode).
- Stringtabellen
  Wenn die vom Programm benötigten Zeichenketten (z.B. Texte in Dialogboxen) in einer separaten Datei gespeichert werden, muss beim Erstellen verschiedener Sprachvarianten der Quelltext nicht geändert werden.
- Tastaturbefehle (Acceleratoren)
- Werkzeugpaletten (Toolbars)
- Versionsinformationen
- Bitmaps
- Icons
- Mauszeiger

Offenbar begegnen uns hier unter dem Titel *Ressourcen* zahlreiche Anwendungselemente, die unseren Elementar-Programmen `HalloWin` und `HalloMFC` aus den Abschnitten 10.1 und 10.2 im Vergleich zu „richtigen“ Windows-Programmen noch fehlen.
Während zum Verfassen des Quellcodes zu einem Windows-Programm einfache Textfenster genügen, bietet das Visual Studio zum Erstellen bzw. Bearbeiten von Ressourcen jeweils spezialisierte Editoren, die in der Regel nach dem WYSIWYG-Prinzip arbeiten. Wie Sie bereits aus dem *Einstieg in Visual C++ 6* wissen, arbeitet man z.B. beim Design einer Dialogbox-Vorlage ähnlich wie in einem 2D-Grafikprogramm, indem man Steuerelemente (z.B. Befehlsschalter) per Maus positioniert und in der Größe verändert. Zum Erstellen von Icons oder Bitmaps bietet das Visual Studio einen Pixeleditor, z.B.:



Die Ressourcen eines Programms landen in einer so genannten **Skriptdatei** (Extension „**.rc**“), im Falle binärer Daten (z.B. Icons) aber nur als Verweis auf eine Zusatzdatei mit dem eigentlichen Inhalt.
Alle Ressourcen erhalten eine eindeutige Kennung (ID), über die sie im Quellcode angesprochen werden können. Es handelt sich um Ganzzahlen, die in **#define** - Precompilerdirektiven einen handlichen Namen erhalten. Solche Definitionen werden in der Headerdatei **Resource.h** gesammelt. Sowohl in Skriptdatei wie auch im Programmcode werden in der Regel die symbolischen Ressourcen-IDs verwendet.

Aus der Skriptdatei und den zugehörigen Inhaltsdateien sowie der Header-Datei **Resource.h** erstellt der **Ressourcen-Compiler** eine binäre **Ressourcendatei** (Extension „.res“), die per Voreinstellung im Debug- oder im Release-Unterverzeichnis des Projektordners (je nach aktiver Projektkonfiguration) neben den Objektdateien des Quellcode-Compilers landet. Der Linker kombiniert schließlich die binäre Ressourcendatei zusammen mit den Objektdateien des Compilers und diversen Bibliotheksbeiträgen zum lauffähigen Programm.
Die folgende Abbildung nach Kruglinkski (1997, S. 7) zeigt, wie Quellcode-Compiler, Ressourcen-Compiler und Linker aus diversen Bestandteilen eine fertige Anwendung erzeugen:

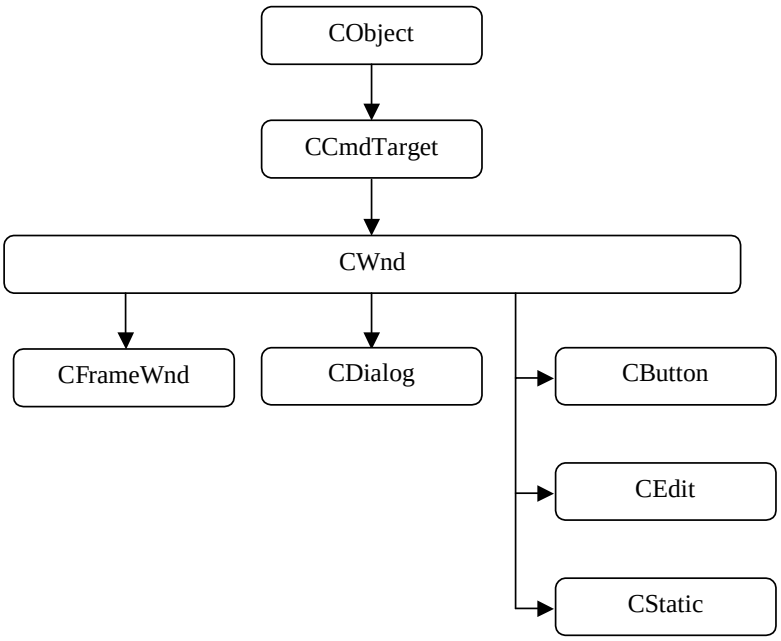


10.3.1 Dialogfelder und Steuerelemente als spezielle Fenster

Dialogfelder treten in Windows-Anwendungen meist als *temporäre* Fenster auf, z.B. angefordert über einen Menü- oder Tastaturbefehl. Jedoch kann ein Dialogfeld auch als *Hauptfenster* einer (meist kleineren) Windows-Anwendung fungieren, was zu Beginn des Kurses im `WinFak`-Beispielprogramm zu sehen war.
Aus der Sicht von Windows handelt es sich bei Dialogfeldern um Fenster aus einer speziellen, vordefinierten, Fensterklasse, die also auch eine bestimmte, von Windows bereitgestellte, Fensterprozedur und damit bestimmte Verhaltensmerkmale gemeinsam haben, von denen hier nur einige genannt werden sollen:

- Dialogfelder werden meist **modal** erzeugt, so dass sie nach dem Öffnen die restliche Anwendung blockieren, bis sie wieder geschlossen werden. So verhalten sich z.B. die **Drucken**-Dialogboxen der meisten Editoren. Es sind aber auch **nonmodale** Dialogboxen möglich, die es dem Benutzer im geöffneten Zustand erlauben, mit anderen Fenstern der Anwendung weiter zu arbeiten. So verhalten sich z.B. die **Suchen**-Dialogboxen vieler Editoren.
- Die Ausdehnung einer Dialogbox kann in der Regel vom Benutzer nicht modifiziert werden.

In der *MFC* wird die Windows-Dialogbox-Fensterklasse durch die Klasse **CDialog** vertreten, eine „Schwester“ der in Abschnitt 10.2 verwendeten Klasse **CFrameWnd**. Wir wollen einen winzigen Ausschnitt aus der MFC-Klassenhierarchie betrachten, der (u.a.) diese Verwandtschaftsbeziehung zeigt:



In der Abbildung sind noch drei weitere „Töchter“ der Klasse **CWnd** zu sehen: In die MFC-Klassen **CButton**, **CEdit** und **CStatic** sind drei elementare Windows-Steuerelemente (engl. controls) verpackt: Befehlsschalter, Eingabefelder und Label. Bei den Steuerelementen handelt es sich aus Windows-Sicht wiederum um Fenster aus speziellen, vordefinierten Klassen (z.B. **BUTTON**, **EDIT**, **STATIC**). Weil damit auch die zugehörigen Fensterprozeduren der Steuerelemente zum Verantwortungsbereich des Betriebsystems gehören, haben Anwender und Programmierer Anlass zur Freude:

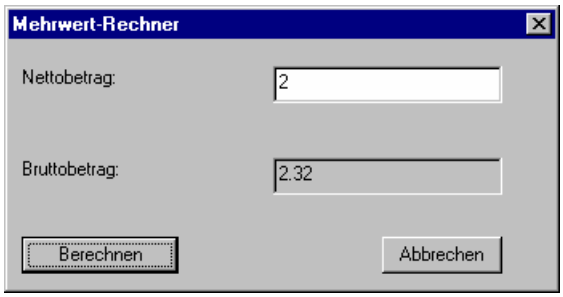
- Die Anwender profitieren von der einheitlichen Erscheinungs- und Verhaltensweise der Steuerelemente.
- Die Programmierer können mit geringem Aufwand eine Benutzerschnittstelle erstellen, weil die Steuerelemente über ihre Fensterprozeduren wichtige Aufgaben selbständig erledigen, sich z.B. beim Eintreffen einer **WM_PAINT**-Nachricht neu zeichnen.

Im Unterschied zum Hauptfenster einer Anwendung besitzen Steuerelemente grundsätzlich ein **Elternfenster**, mit dem Sie folgendermaßen kooperieren:

- Ein Kindfenster ist fest im Elternfenster verankert und macht deshalb alle Bewegungen mit.
- Wird das Elternfenster geschlossen, verschwinden auch die Kindfenster.
- Besonders wichtig: Bei relevanten Ereignisse (vor allem Eingaben) schickt ein Steuerelement eine **WM_COMMAND** - Nachricht an sein Elternfenster, das dann geeignet reagieren kann.

Meist treten Steuerelemente in Dialogboxen auf, doch sind sie auch in anderen Fensterklassen realisierbar. Im letztgenannten Fall erzeugt man die Steuerelemente einzeln als Instanzen der jeweiligen MFC-Klasse und lässt sie dann über ihre **Create()**-Methode auftreten (siehe z.B. Prosise 1999, S. 316). Beim Einsatz von Steuerelementen innerhalb von Dialogboxen, der in diesem Abschnitt ausführlich behandelt wird, entwirft man in der Regel mit dem Dialogbox-Designer eine Vorlage, die später einem **CDialog()**-Konstruktor übergeben wird. Wenn das **CDialog**-Objekt nur auf die per Steuerelement erfassten oder angezeigten Werte, nicht jedoch auf Eigenschaften der Steuerelemente (z.B. Schriftart) zugreift, treten gar keine MFC-Klassen zu einzelnen Steuerelementen in Erscheinung.

Nach so viel Theorie wollen wir endlich wieder ein Programm erstellen, diesmal zur Berechnung der Mehrwertsteuer. Wie das `WinFak`-Beispiel verwendet das neue Programm ein Dialogfeld als Hauptfenster:



Allerdings verzichten wir diesmal weitgehend auf die Hilfe der Assistenten, um Schritt für Schritt einen verständlichen Quellcode aufzubauen, statt durch eine Fülle unbekannter Sprachelemente erschlagen zu werden.
Legen Sie ein neues Projekt vom Typ **Win32-Anwendung** an, wählen Sie im ersten Schritt des Anwendungsassistenten ein **leeres Projekt**, und lassen Sie es **fertigstellen**. Aktivieren Sie die MFC-Verwendung für das Projekt, indem Sie nach dem Menübefehl **Projekt > Einstellungen** auf der Registerkarte **Allgemein** der Dialogbox **Projekteinstellungen** in der mit **Microsoft Foundation Classes** überschriebenen Liste den folgenden Eintrag wählen: **MFC in einer gemeinsam genutzten DLL verwenden**.

Fügen Sie eine Header-Datei namens **Bruno.h** mit folgendem Quelltext ein, der später noch zu erweitern ist:

```
#include <afxwin.h>
#include "resource.h"

class CBrunoApp : public CWinApp
{
public:
   BOOL InitInstance();
};

class CBrunoDlg : public CDialog
{
public:
   CBrunoDlg(CWnd* pParent = NULL);
};
```

Hier wird neben der MFC-Standard-Headerdatei **afxwin.h** auch die (projektspezifische) Header-Datei **resource.h** mit Ressourcen-Identifikationen inkludiert, die gleich entstehen wird.

Fügen Sie eine Implementierungsdatei ein mit dem folgenden, vorläufigen Inhalt, der später noch um wesentliche Bestandteile (z.B. Nachrichten-Handler) erweitert werden muss:

```
#include "Bruno.h"

CBrunoApp anApp;

BOOL CBrunoApp::InitInstance()
{
   CBrunoDlg dlg;
   dlg.DoModal();
   return FALSE;
}

CBrunoDlg::CBrunoDlg(CWnd* pParent) : CDialog(IDD_DIALOG, pParent)
{
}
```

Weil die Fensterklasse nicht aus **CFrameWnd** sondern aus **CDialog** abstammt, wird in der **InitInstance()**–Methode der Anwendungsklasse im Vergleich zu Abschnitt 10.2.1 eine andere Strategie zum Erzeugen des Fensterobjektes gewählt:

- Es wird ein *lokales* `CBrunoDlg`-Objekt erzeugt, das folglich beim Beenden der Methode verschwindet.
- Im `CBrunoDlg`-Konstruktor wird per Initialisierungsliste der **CDialog**-Konstruktor aufgerufen, welcher eine Dialog-Vorlagen-ID als ersten und einen **CWnd**-Pointer als zweiten Parameter benötigt. In unserem Fall hat der zweite Parameter den Wert NULL, weil das Anwendungs kein Elternfenster besitzt. Die Dialogbox-Ressource zu `IDD_DIALOG` müssen wir noch erstellen.
- Aufgrund des Rückgabewertes **FALSE** endet mit **InitInstance()** auch die gesamte Anwendung, was im Verhalten der Funktion **AfxWinMain()** begründet liegt (siehe Abschnitt 10.2.1). Insbesondere wird die **Run()**-Methode und damit die Nachrichtenschleife des Anwendungsobjektes nie ausgeführt. Das Anwendungsobjekt hat also diesmal nur die Aufgabe, eine Dialogbox zu erzeugen und zu starten.
- Die Dialogbox wird **modal** gestartet, d.h. der Methodenaufruf `dlg.DoModal()` endet erst dann, wenn die Dialogbox wieder geschlossen wird.
  Dies ist in unserem Fall auch sinnvoll, weil nach dem Aufruf sowohl **InitInstance()** als auch die Anwendung selbst beendet werden. Oft werden aber nichtmodale Dialogbox benötigt, welche noch zu ihren Lebzeiten die Kontrolle abgeben.

Bis jetzt zeigt das Arbeitsbereichsfenster das gewohnte, ressourcenfreie Bild:

10.3.1.1 Dialogfeldvorlagen

Vor dem Erstellen einer Windows-Dialogbox ist *keine* neue Fensterklasse zu definieren und zu registrieren (die gibt es ja schon), sondern eine **Vorlage** (engl.: *dialog box template*) zu definieren, die dem neuen Fenster beim Erstellen zugewiesen wird (über eine eindeutige Ressourcen-Kennung). Eine Dialogbox-Vorlage wird in Textform in der Ressourcen-Skriptdatei des Projektes abgelegt und enthält u.a. folgende Angaben:

- Bezeichner für die Ressourcen-ID der Dialogbox
- Typ der Ressource
- Position und Maße des Fensters
- Fensterstile
- Beschriftung der Titelzeile
- Schriftart (auch für die Steuerelemente der Dialogbox gültig)

- Eingerahmt von **BEGIN** und **END** für jedes Steuerelement der Dialogbox (teilweise in Anhängigkeit vom Typ):
  - Typ
  - Beschriftung
  - Bezeichner für die Ressourcen-ID des Steuerelementes
  - Position und Maße
  - Fensterstile

Hartgesottene Windows-Entwickler erstellen die Dialogbox-Vorlagen manuell nach dem Motto: „Wir *benutzen* keine Dialogboxen, sondern *erstellen* welche“. Für die oben schon gezeigte Dialogbox zum aktuellen Beispielprogramm sieht die Vorlage folgendermaßen aus:

```
IDD_DIALOG DIALOG DISCARDABLE  0, 0, 227, 95
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "Mehrwert-Rechner"
FONT 8, "MS Sans Serif"
BEGIN
    EDITTEXT        IDC_NETTO,110,10,95,14,ES_AUTOHSCROLL
    DEFPUSHBUTTON   "Berechnen",IDOK,5,75,65,14
    PUSHBUTTON      "Abbrechen",IDCANCEL,155,75,50,14,NOT WS_TABSTOP
    EDITTEXT        IDC_BRUTTO,110,45,95,14,ES_AUTOHSCROLL | ES_READONLY |
                    NOT WS_TABSTOP
    LTEXT           "Nettobetrag:",IDC_STATIC,5,10,90,15
    LTEXT           "Bruttobetrag:",IDC_STATIC,5,45,90,15
END
```

Rationelle VC++ - Entwickler erstellen solche Vorlagen mit dem Dialogbox-Editor:

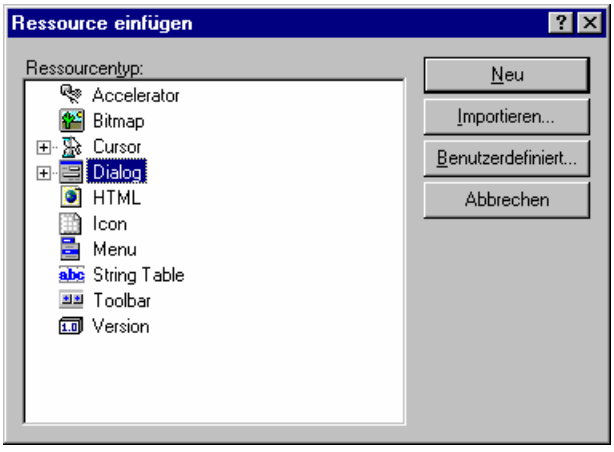
- Ohne Nachschlagen von Bezeichnern, Syntaxregeln etc. übernimmt man die Steuerelemente aus folgender Palette auf ein Dialogfeld:

- Ihr Erscheinungsbild wird mit Hilfe eines speziellen WYSYWIG-Editors gestaltet.
- Die Eigenschaften der Dialogbox und der Steuerelemente (z.B. Ressourcen-IDs, Fensterstile) können über entsprechende Dialoge festgelegt werden.

Starten Sie nun mit

Einfügen > Ressource > Dialog > Neu

den Dialogbox-Editor.



Um unsere Tätigkeit als Dialogbox-Designer zu erleichtern, schalten wir zunächst die (magnetischen) Rasterlinien ein mit **Layout > Einstellungen für Führungslinien** oder 

Die im Rahmen **Rasterweite** angegebene Maßeinheit DLU basiert auf der Schriftart des Dialogfelds (Voreinstellung: 8 Punkt MS Sans Serif). Eine horizontale DLU entspricht einem Viertel der durchschnittlichen Zeichenbreite. Eine vertikale DLU entspricht einem Achtel der durchschnittlichen Zeichenhöhe.

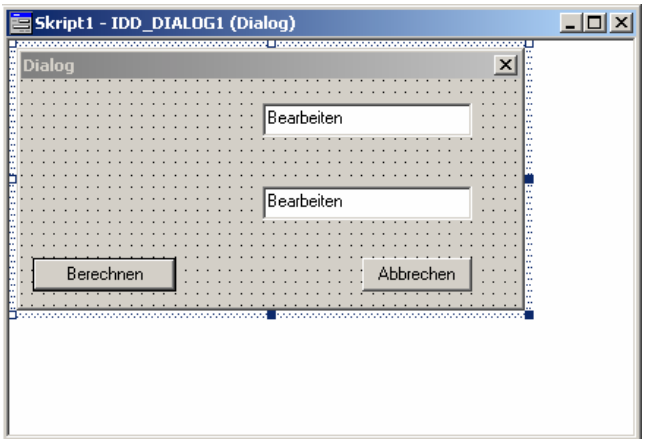
Beim Dialogbox-Design können Sie weitgehend genau so vorgehen wie in typischen Windows-Grafikprogrammen, so dass im folgenden die handwerklichen Aspekte nur knapp erläutert werden.

Der als Starthilfe vorgegebene Befehlsschalter **OK** wird verschoben und erhält in seinem Eigenschaftsdialog (z.B. via Kontextmenü erreichbar) den Titel **Berechnen**:

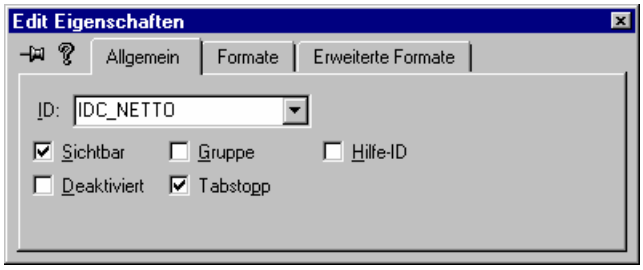
Seine symbolische Ressourcen-ID **IDOK** sollte beibehalten werden, weil das MFC-Framework in einigen Situationen den Befehlsschalter mit dieser ID speziell unterstützt.
Eigenschaftsdialogboxen zu Steuerelementen können übrigens bei markiertem Steuerelement besonders einfach per **Enter**-Taste geöffnet und auch wieder geschlossen werden.

Verschieben Sie den ebenfalls vorgegebenen Schalter **Abbrechen** an eine passende Stelle.

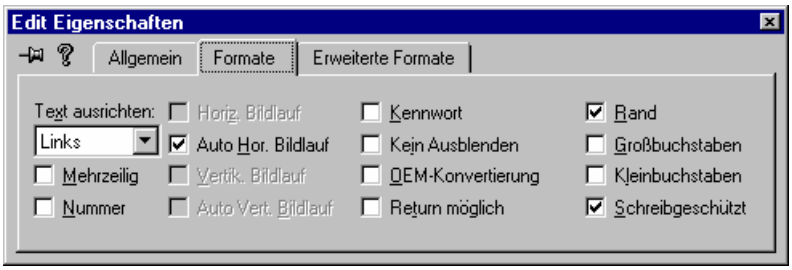
Platzieren Sie zwei **Eingabefelder** für den Netto- und den Bruttobetrag auf Ihrem Formular, indem Sie das zugehörige Symbol  auf der Steuerelemente-Palette markieren und dann mit der Maus im Klientenbereich der Dialogbox ein passendes Rechteck aufziehen. Mittlerweile könnte die entstehende Dialogbox ungefähr so aussehen:



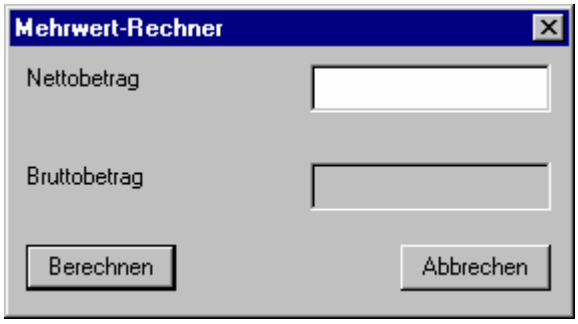
Das erste Eingabefeld erhält per Voreinstellung die symbolische Ressourcen-Identifikation **IDC_EDIT1**, die wir auf **IDC_NETTO** abändern wollen:



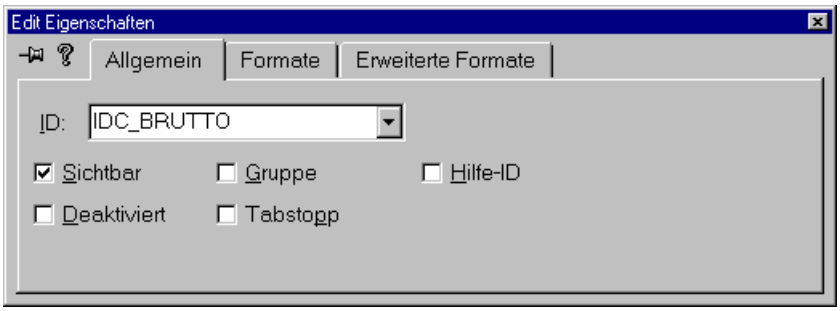
Beim zweiten Eingabefeld vereinbaren wir die Identifikation **IDC_BRUTTO** und auf dem Registerblatt **Formate** auch noch das Attribut **Schreibgeschützt**:



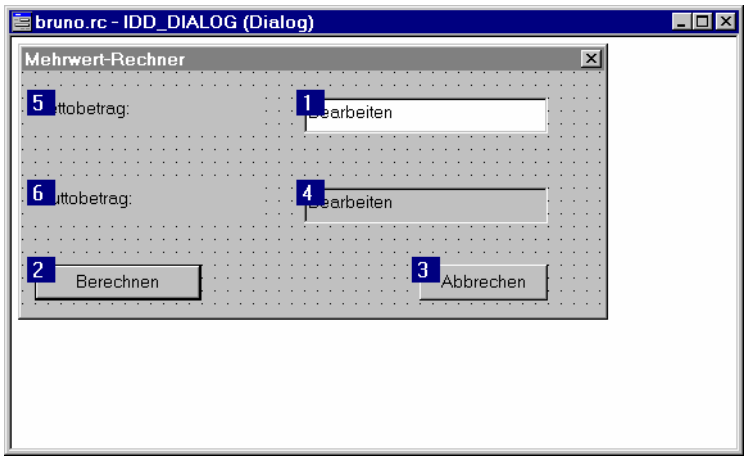
Mit Hilfe von **Text**-Steuerelementen (Aa) werden die Eingabefelder beschriftet.
Über den Schalter können Sie die Dialogbox jetzt schon besichtigen:



Wir sorgen noch für eine günstige **Tabulator-Reihenfolge**, damit sich die Dialogbox möglichst bequem bedienen lässt. Zunächst wird bei allen Steuerelementen mit Ausnahme von IDC_NETTO und IDOK nötigenfalls der Tabstopp abgeschaltet, z.B. bei IDC_BRUTTO:



Nach **Layout > Tabulator-Reihenfolge** kann man die Tabulator-Positionen mit einer Folge von Mausklicks festlegen. In unserem Fall ist nur zu beachten, dass IDC_NETTO in der Sequenz vor IDOK liegt:



Last not least werden nun noch Attribute der Dialogbox *selbst* angepasst, indem diese per Mausklick auf eine freie Stelle markiert und dann ihr Eigenschaftsdialog geöffnet wird. Hier vereinbaren wir die oben schon im Quelltext vorweg genommene Ressourcen-Identifikation **IDD_DIALOG** und wählen noch einen hübschen Text für die Titelzeile:

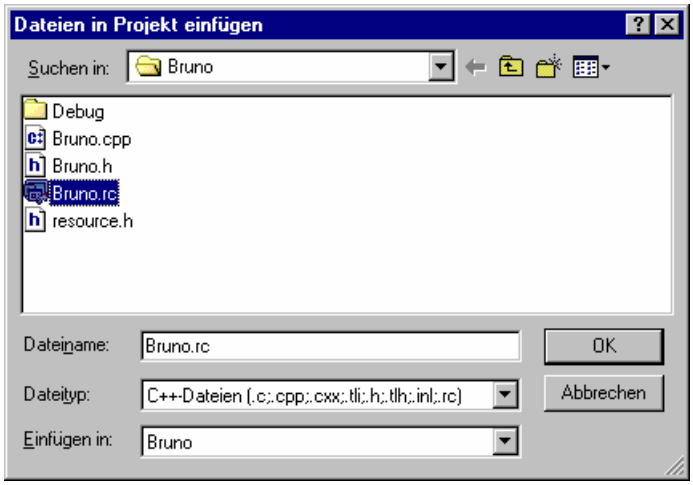
Nun wird es Zeit, unsere Dialogbox-Vorlage, die noch kein Bestandteil irgend einer Anwendung ist, zu sichern, z.B. mit **Datei > Speichern unter** bzw. **<Strg><S>**, wobei sich VC++ nach einem Namen für das Ressourcenskript erkundigt:

Die neue Ressourcen-Skriptdatei ist zwar momentan in der Entwicklungsumgebung geöffnet und in einem Fenster ansprechbar, gehört aber noch nicht zum Projekt.

Diesen Zustand korrigieren wir nach dem Menübefehl:

Projekt > Dem Projekt hinzufügen > Dateien

in folgender Dialogbox:



Schließen Sie nun nötigenfalls das separate Fenster zum Ressourcenskript **Bruno.rc**, damit das Arbeitsbereichsfenster mit dem neuen **Ressourcen**-Registerblatt voll funktionstüchtig wird:

Mittlerweile hat VC++ auch die Header-Datei **resource.h** mit folgendem Inhalt erstellt:

```
//{{NO_DEPENDENCIES}}
// Microsoft Developer Studio generated include file.
// Used by bruno.rc
//
#define IDD_DIALOG                      101
#define IDC_NETTO                       1000
#define IDC_BRUTTO                      1001
#define IDC_STATIC                      -1

// Next default values for new objects
//
#ifdef APSTUDIO_INVOKED
#ifndef APSTUDIO_READONLY_SYMBOLS
#define _APS_NEXT_RESOURCE_VALUE        102
#define _APS_NEXT_COMMAND_VALUE         40001
#define _APS_NEXT_CONTROL_VALUE         1002
#define _APS_NEXT_SYMED_VALUE           101
#endif
#endif
```

Hier werden die symbolischen Ressourcen-IDs mit Ganzzahl-Konstanten assoziiert. Bei **IDC_STATIC** wird durch eine -1 dokumentiert, dass keine eindeutige Zuordnung existiert. Diese wird auch nicht benötigt, weil die beiden **Text**-Steuerelemente im Programm nicht angesprochen werden sollen.

10.3.1.2 Dialogbox-Steuerelemente in eine MFC-Anwendung integrieren

Im Konstruktor unserer Fensterklasse `CBrunoDlg` wird durch Angabe der Ressourcen-Identifikation **IDD_DIALOG** dafür gesorgt, dass ein Windows-Dialogfenster mit dem von uns festgelegten Layout erzeugt wird:

```
CBrunoDlg::CBrunoDlg(CWnd* pParent) : CDialog(IDD_DIALOG, pParent) {}
```

10.3.1.2.1 Daten austauschen

Die Steuerelemente können zwar (dank Windows) mit dem Benutzer interagieren, allerdings gibt es keine nennenswerte Interaktion mit unserer Anwendung. Um diesen Mangel zu beheben, wird zunächst die Definition unserer Fensterklasse `CBrunoDlg` folgendermaßen erweitert:

- `CBrunoDlg` erhält Membervariablen `m_dNetto` und `m_dBrutto` vom Typ **double** zur Verwaltung von Netto- und Bruttobetrag. Wie oben bereits erläutert, wird mit dem Präfix „`m_`“ nach einer unter MFC-Programmierern verbreiteten Konvention zum Ausdruck gebracht, dass es sich um Element- (alias Member-)variablen einer Klasse handelt. Diese können damit spontan von lokalen oder globalen Variablen unterschieden werden. Nach dem Member-Präfix folgt üblicherweise eine Typspezifikation nach ungarischer Tradition.
- Zwecks Datenaustausch zwischen unserem MFC-Fensterobjekt und der Windows-Dialogbox mit den Steuerelementen wird für unsere MFC-Fensterklasse die (von **CDialog** geerbte) Funktion **DoDataExchange()** überschrieben:

  ```
  void DoDataExchange(CDataExchange* pDX);
  ```

- Wir deklarieren die Methode `OnBerechnen()`, die nach einem Mausklick auf den Schalter **Berechnen** ausgeführt werden soll.
- Am Ende der Klassendefinition wird das zur Vorbereitung der Nachrichtenbehandlung erforderliche Makro **DECLARE_MESSAGE_MAP()** eingefügt.

Insgesamt ergibt sich für die Header-Datei **Bruno.h** folgender Stand:

```
#include <afxwin.h>
#include "Resource.h"

class CBrunoApp : public CWinApp
{
public:
   BOOL InitInstance();
};

class CBrunoDlg : public CDialog
{
public:
   CBrunoDlg(CWnd* pParent = NULL);
   void DoDataExchange(CDataExchange* pDX);
   void OnBerechnen();
private:
   double m_dNetto;
   double m_dBrutto;
DECLARE_MESSAGE_MAP()
};
```

In der Implementierungsdatei **Bruno.cpp** werden korrespondierend folgende Erweiterungen vorgenommen:

- Im `CBrunoDlg`-Konstruktor werden die neuen Membervariablen initialisiert.
- In der Methode **DoDataExchange**() wird über die Funktion **DDX_Text()** jeweils eine Elementvariable mit einem Steuerelement verknüpft, wobei letzteres über seine Ressourcen-ID angesprochen wird:

```
void CBrunoDlg::DoDataExchange(CDataExchange* pDX)
{
 CDialog::DoDataExchange(pDX);
 DDX_Text(pDX, IDC_NETTO, m_dNetto);
 DDX_Text(pDX, IDC_BRUTTO, m_dBrutto);
}
```

Einleitend wird die Basisklassenmethode **DoDataExchange**() aufgerufen.

- Makros für die Nachrichtenzuordnungstabelle einer Fensterklasse einzufügen, gehört inzwischen schon zu unseren Routinearbeiten:

```
BEGIN_MESSAGE_MAP(CBrunoDlg, CDialog)
 ON_BN_CLICKED(IDOK, OnBerechnen)
END_MESSAGE_MAP()
```

Hier wird festgelegt, dass die vom Steuerelement **IDOK** an unser Fensterobjekt gerichtete Nachricht **WM_BN_CLICKED** durch die Methode `OnBerechnen()` des Fensterobjektes beantwortet werden soll.

- Schließlich muss die Ereignis-Routine für Mausklicks auf den **Berechnen**-Schalter noch implementiert werden:

```
void CBrunoDlg::OnBerechnen()
{
 UpdateData(TRUE);
 m_dBrutto = 1.16 * m_dNetto;
 UpdateData(FALSE);
}
```

Hier leistet die (von **CDialog** geerbte) Methode **UpdateData()** wertvolle Arbeit, wozu sie die Funktion **DoDataExchange()** aufruft:

  - Ist der Aktualparameter TRUE, dann werden die Daten der Steuerelemente an die Membervariablen des Fensterobjektes übertragen.
  - Ist der Aktualparameter FALSE, dann verläuft der Transfer in die umgekehrte Richtung.

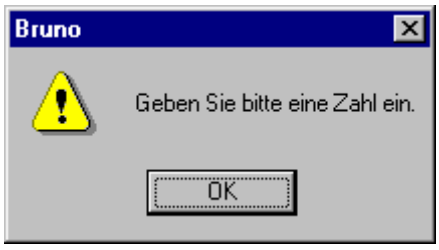
Damit sieht die Implementierungsdatei **Bruno.cpp** folgendermaßen aus:

```
#include "Bruno.h"

CBrunoApp anApp;

BOOL CBrunoApp::InitInstance()
{
   CBrunoDlg dlg;
   dlg.DoModal();
   return FALSE;
}

CBrunoDlg::CBrunoDlg(CWnd* pParent) : CDialog(IDD_DIALOG, pParent)
{
   m_dNetto = 0.0;
   m_dBrutto = 0.0;
}

void CBrunoDlg::DoDataExchange(CDataExchange* pDX)
{
   CDialog::DoDataExchange(pDX);
   DDX_Text(pDX, IDC_NETTO, m_dNetto);
   DDX_Text(pDX, IDC_BRUTTO, m_dBrutto);
}
```

```
BEGIN_MESSAGE_MAP(CBrunoDlg, CDialog)
   ON_BN_CLICKED(IDOK, OnBerechnen)
END_MESSAGE_MAP()

void CBrunoDlg::OnBerechnen()
{
   UpdateData(TRUE);
   m_dBrutto = 1.16 * m_dNetto;
   UpdateData(FALSE);
}
```

10.3.1.2.2 Daten validieren

Beim Datenaustausch über **UpdateData()** werden dankenswerterweise ohne unser Zutun elementare Eingabe- bzw. Konvertierungsprobleme behandelt. Wenn ein Benutzer z.B. statt der beabsichtigten Eingabe „2e2“ (gemeint ist: $2 \cdot 10^2 = 200$) versehentlich „2ee2“ abschickt, dann erhält er folgende Fehlermeldung:



Die C-Funktionen **atof()** liefert übrigens in einer analogen Situation ohne jede Warnung den Wert 2.0 zurück.

Eventuell wollen Sie die erfassten Daten einer strengeren Plausibilitätskontrolle unterwerfen, z.B. den erlaubten Wertebereich durch einen minimalen und/oder maximalen Wert begrenzen. Auch hier bietet die MFC mit ihren **Dialogdaten-Validierungsroutinen** (Abkürzung: DDV) einige Unterstützung an. Wir setzen der Übung halber in der **CBrunoDlg**-Methode **DoDataExchange** hinter den **DDX_Text()**-Aufruf für den Nettobetrag einen **DDV_MinMaxDouble()**-Aufruf mit Grenzen für den minimalen und maximalen Betrag, obwohl dies den Nutzen des Programms nicht unbedingt steigert:

```
void CBrunoDlg::DoDataExchange(CDataExchange* pDX)
{
   CDialog::DoDataExchange(pDX);
   DDX_Text(pDX, IDC_NETTO, m_dNetto);
   DDV_MinMaxDouble(pDX, m_dNetto, 0, 99);
   DDX_Text(pDX, IDC_BRUTTO, m_dBrutto);
}
```

Jedenfalls wird der Benutzer nun bei Grenzüberschreitungen gewarnt:

Ohne weitere Maßnahmen setzt die Ereignisbehandlungsroutine **OnBerechnen()** allerdings nach dem Quittieren der Warnungs-Dialogbox ihre Arbeit unbeeindruckt fort. Um unbrauchbare Eingaben abzuweisen, müssen Sie den BOOLschen Rückgabewert von **UpdateData()** auswerten, z.B.:

```
   if (!UpdateData(TRUE)) return;
```

Neben **DDVMinMaxDouble()** gibt es noch analoge Funktionen für andere Datentypen. Mit **DDV_MaxChars()** lässt sich die Länge eines Eingabetextes begrenzen. Noch mehr Flexibilität gewinnt man durch eigene DDX- und DDV-Routinen.

10.3.1.2.3 Eigenschaften von Steuerelementen verändern

Bislang haben wir die Interaktion des MFC-Fensterobjektes mit den Dialogbox-Steuerelementen auf den Datenaustausch beschränkt. Oft ist es aber auch sinnvoll, Eigenschaften von Steuerelementen zu verändern. In der `Bruno`-Dialogbox wäre es z.B. angebracht, den Brutto-Betrag aus der letzten Berechnung grau („überholt“) darzustellen, sobald der korrespondierende Betrag im Netto-Feld verändert worden ist.
Dazu muss für das zu manipulierende Eingabefeld-Steuerelement (mit der Identifikation IDC_BRUTTO) in die **CBrunoDlg**-Klassendefinition ein MFC-Objekt aus der Klasse **CEdit** aufgenommen werden:

```
class CBrunoDlg : public CDialog
{
public:
   CBrunoDlg(CWnd* pParent = NULL);
   void DoDataExchange(CDataExchange* pDX);
   void OnBerechnen();
   void OnChangeNetto();
private:
   double m_dNetto;
   double m_dBrutto;
   CEdit m_ctrlBrutto;
DECLARE_MESSAGE_MAP()
};
```

Damit das **CEdit**-Objekt über den DDX-Echanismus auf die Eigenschaften des Steuerelementes mit der Kennung `IDC_BRUTTO` zugreifen kann, wird die `CBrunoDlg`-Methode `DoDataExchange()` folgendermaßen erweitert:

```
void CBrunoDlg::DoDataExchange(CDataExchange* pDX)
{
   CDialog::DoDataExchange(pDX);
   DDX_Text(pDX, IDC_NETTO, m_dNetto);
   DDX_Text(pDX, IDC_BRUTTO, m_dBrutto);
   DDX_Control(pDX, IDC_BRUTTO, m_ctrlBrutto);
}
```

Weil das Brutto-Steuerelement verändert werden soll, sobald das Netto-Steuerelement einen neuen Wert annimmt, erhält **CBrunoDlg** außerdem die Nachrichtenbehandlungsmethode `OnChangeNetto()`, die per MESSAGE MAP - Makro mit der vom Steuerelement IDC_NETTO initiierten Benachrichtigung WM_EN_CHANGE verbunden wird:

```
BEGIN_MESSAGE_MAP(CBrunoDlg, CDialog)
   ON_BN_CLICKED(IDOK, OnBerechnen)
   ON_EN_CHANGE(IDC_NETTO, OnChangeNetto)
END_MESSAGE_MAP()
```

Schließlich muss `OnChangeNetto()` noch implementiert werden:

```
void CBrunoDlg::OnChangeNetto()
{
   m_ctrlBrutto.EnableWindow(FALSE);
}
```

Mit der Methode **EnableWindow(FALSE)** wird der IDC_BRUTTO-Inhalt grau eingefärbt. Den ursprünglichen Zustand stellt `OnBerechnen()` mit `m_ctrlBrutto.EnableWindow(TRUE)` wieder her.

10.3.2 Menüvorlagen

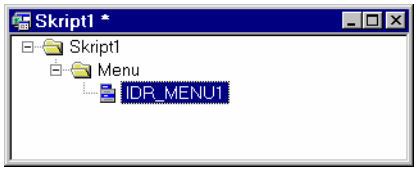
Die Dialogboxvorlagen-Ressource haben wir im Rahmen der Minianwendung **Bruno** kennen gelernt, die eine Dialogbox als einziges Fenster benutzt. Einige andere Ressourcen (z.B. die Symbolleiste) lassen sich nur umständlich in eine dialogboxbasierte Windows-Anwendung integrieren. Andererseits macht es kaum Mühe, **CFrameWnd**-basierte Fenster mit Ressourcen aus den noch zu behandelnden Kategorien auszustatten. Daher kehren wir zum Beispiel **HalloMFC** zurück und erweitern es um:

- Menüzeile
- Icon
- Zeichenfolgentabelle (engl.: string table)
- Beschleunigungstasten (engl.: accelerators)
- Info-Dialogbox
- Symbol- und Statuszeile

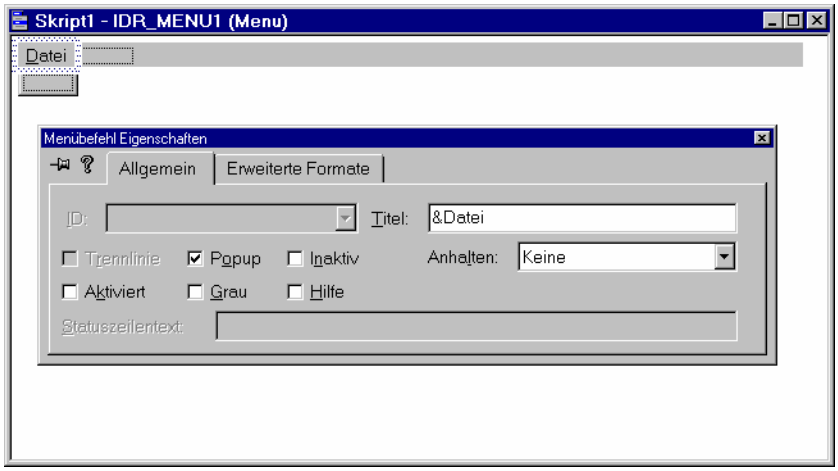
Öffnen Sie das Projekt und starten Sie den Menüeditor mit:

Einfügen > Ressourcen > Menu > Neu

Neben dem Menüeditor erscheint auch ein Fenster für die nun entstehende Ressourcen-Skriptdatei, die wir später in das (noch ressourcenfreie) Projekt aufnehmen werden:

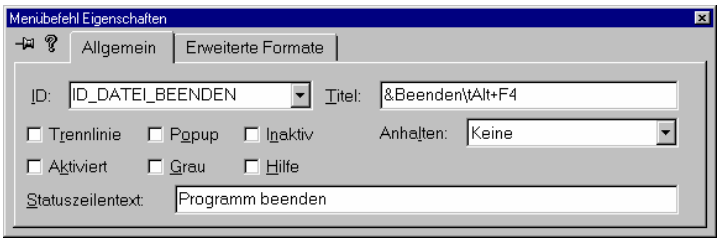


Im Menüeditor ist der Platzhalter für das erste Hauptmenü-Item markiert, so dass wir seine Bezeichnung **Datei** gleich eintippen können. Daraufhin erscheint der folgende **Eigenschaften**-Dialog:



Vor den ersten Buchstaben setzen wir ein **&**, damit er zur Kurzwahl dienen kann.
Weil das Visual Studio damit rechnet, dass von den Hauptmenü-Items ein Aufklappmenü geöffnet werden soll, ist das zuständige Kontrollkästchen **Popup** schon markiert, und es ist keine ID vorhanden.

Wir markieren im Menüeditor den Platzhalter für das erste Item im **Datei**-Aufklappmenü und tragen den Titel ein, woraufhin der folgende **Eigenschaften**-Dialog erscheint:

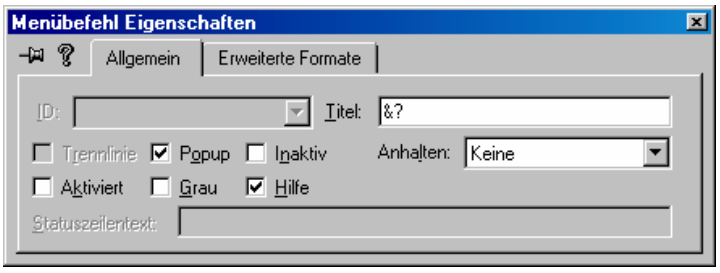


Diesmal rechnet das VS damit, dass mit dem bearbeiteten Menü-Item kein weiteres Aufklappmenü geöffnet, sondern eine WM_COMMAND-Nachricht ausgelöst werden soll:

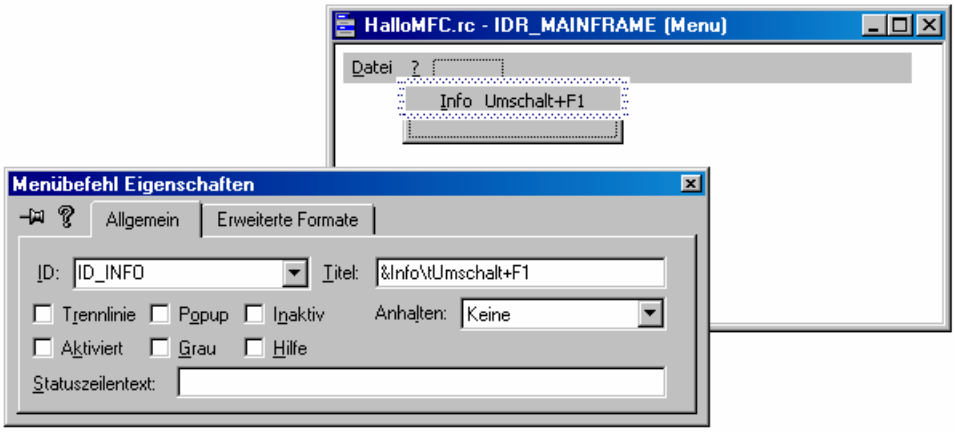
- Das Kontrollkästchen **Popup** ist *nicht* markiert.
- Das Item erhält eine ID

Am Ende des Titels notieren wir noch, durch eine Tabulator-Escapesequenz getrennt, über welche Tastenkombination dieselbe WM_COMMAND-Nachricht alternativ ausgelöst werden kann. Bevor eine entsprechende Accelerator-Ressource angelegt ist (siehe unten), handelt es sich nur um eine spezielle Etikettierung des Menü-Items, wobei die Tabulator-Escapesequenz dafür sorgt, dass die Tastenbeschreibung am rechten Rand des verfügbaren Platzes erscheint.
Der Vollständigkeit halber liefern wir auch noch einen Text für die später anzulegende Statuszeile.

Markieren Sie nun den Platzhalter für das zweite Hauptmenü-Item und benennen Sie es mit einem Fragezeichen. Wird in der **Eigenschaften**-Dialogbox das Kontrollkästchen **Hilfe** markiert, dann erscheint das **?** – Hauptmenü-Item am rechten Rand des Fensters:

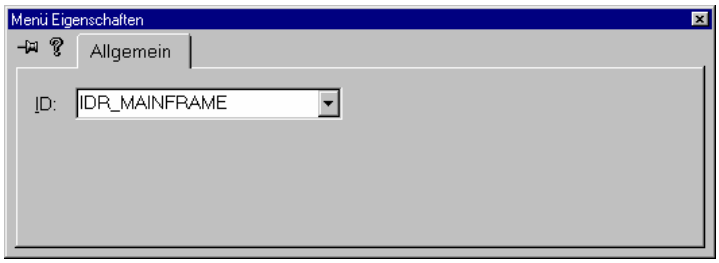


Ergänzen Sie im zugehörigen Aufklappmenü ein **Info**-Item:



Die hier angegebene Tastenkombination **Umschalt+F1** ist bisher nur ein leeres Versprechen, das später über eine Accelerator-Ressource eingelöst wird.

Bevor wir die Arbeit an der Menüvorlage vorläufig beenden, ändern wir ihre Identifikation von IDR_MENU1 ab in IDR_MAINFRAME. Die dazu erforderliche Eigenschaftsdialogbox ist z.B. über das Kontextmenü zu einer freien Stelle der in Bearbeitung befindlichen Menüleiste erreichbar:



Wir werden von der Möglichkeit Gebrauch machen, für zusammengehörige Ressourcen aus den Kategorien

- Menüzeile
- Icon
- Fenstertitel
- Accelerator-Tabelle

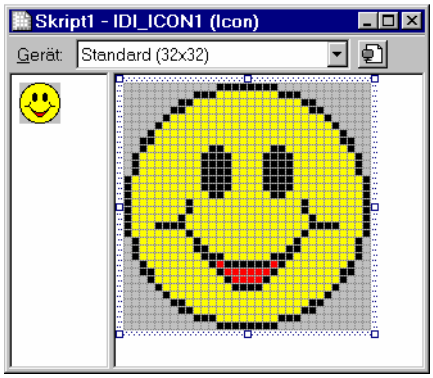
die gemeinsame Bezeichnung IDR_MAINFRAME zu verwenden.

10.3.3 Anwendungs-Icon erstellen

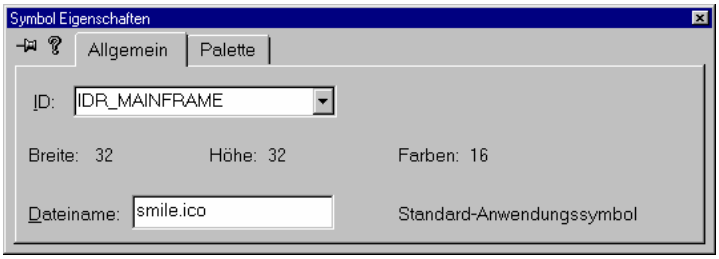
Erstellen Sie nach

Einfügen > Ressourcen > Icon > Neu

mit dem Icon-Editor eine 32×32 - Bitmap als Programmsymbol (für das Systemmenü und die Taskleiste), z.B.:

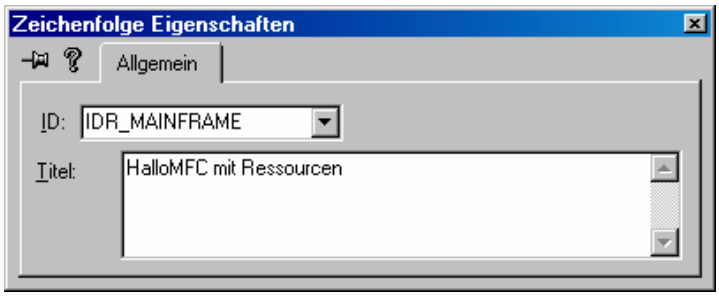


Vergeben Sie in der Eigenschaftsdialogbox (Öffnen per **<Enter>**) wiederum die Identifikation IDR_MAINFRAME:

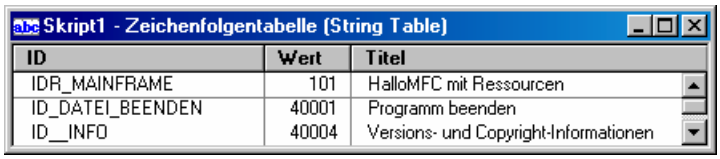


10.3.4 Zeichenfolgentabelle

Weil wir Statuszeilentexte vereinbart haben, existiert schon eine Zeichenfolgentabelle, die wir über das Ressourcen-Skript-Fenster per Doppelklick öffnen und dann per **Einfügen > Neue Zeichenfolge** um einen Eintrag IDR_MAINFRAME für den Fenstertitel ergänzen:



Die Nummern für die Zeichenfolgen werden von der Entwicklungsumgebung festgelegt:



10.3.5 Accelerator-Tabelle

Nun muss noch die Tastenkombination **<Umschalt><F1>** mit dem zugehörigen Menübefehl verknüpft werden. Dazu legen wir mit

Einfügen > Ressourcen > Accelerator > Neu

eine Accelerator-Tabelle an und ordnen über **Einfügen > Neue Zugriffstaste** die Kombination aus der **Taste VK_F1** sowie der **Zusatztaste Umschalt** dem Befehl **ID_INFO** zu:

Die gesamte Tabelle muss nun ebenfalls die Gruppen-Ressourcen-Identifikation IDR_MAINFRAME erhalten. Den Eigenschaftsdialog zum Umbenennen:

erreichen Sie am besten über das Kontextmenü zum Accelerator-Eintrag im Ressourcen-Skript-Fenster.

10.3.6 Ressourcen in ein Projekt einfügen

Nun soll das neue Ressourcen-Skript gespeichert und in unser Projekt aufgenommen werden:

- Öffnen Sie das Ressourcen-Skript-Fenster und speichern Sie den Inhalt (z.B. über **Datei > Speichern unter**) in die Datei **HalloMFC.rc** im Projektverzeichnis.
- Nehmen Sie die neue Datei in das Projekt auf:
  Projekt > Dem Projekt hinzufügen > Dateien

- Nun kann das separate Ressourcen-Skript-Fenster geschlossen werden, weil sein Inhalt über die Ressourcen-Registerkarte des Arbeitsbereichsfensters verfügbar ist.
- Weil neben dem Ressourcen- auch der Quellcode-Compiler die Identifikationsnummern der Ressourcen benötigt, muss die Datei **resource.h** in der Projekt-Headerdatei inkludiert werden:

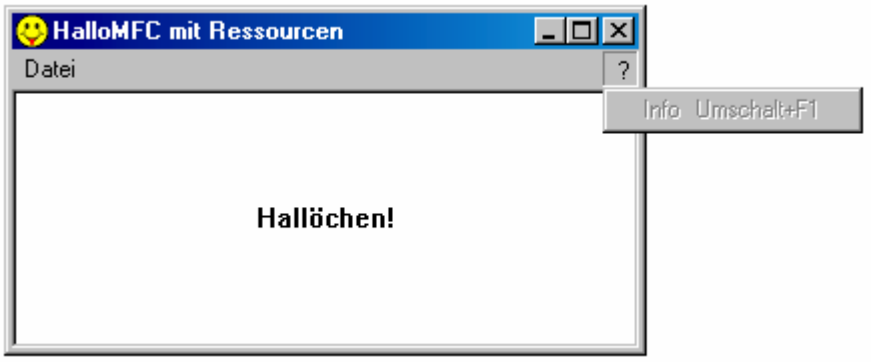
  ```
  #include "resource.h"
  ```

  Hier werden die von uns vergebenen Namen mit Ganzzahl-Werten assoziiert.

Um die Ressourcen mit der gemeinsamen Identifikation IDR_MAINFRAME (ein Menü, ein Icon, eine Tastenkombination und eine Titelzeile) dem Fenster unserer Anwendung zur Verfügung zu stellen, müssen wir lediglich im Konstruktor der zugehörigen Klasse den **Create()**-Aufruf durch einen Aufruf der Methode **LoadFrame()** ersetzen:

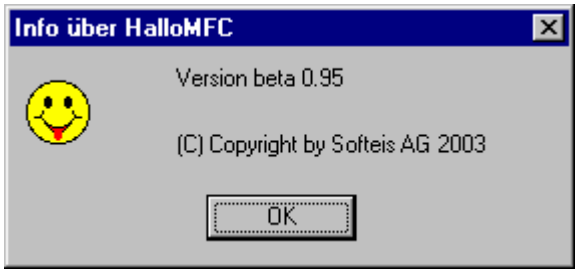
```
CmMFCWnd::CmMFCWnd()
{
   LoadFrame(IDR_MAINFRAME);
}
```

Bevor die Menübefehle klappen, sind noch kleinere Arbeiten erforderlich:



10.3.7 Eine Dialogbox auftreten lassen

Die folgende Info-Dialogbox



zu erstellen, sollte Ihnen keine großen Schwierigkeiten mehr bereiten. Am Anfang steht die Gestaltungsarbeit mit dem Dialog-Editor:

- **Einfügen > Ressourcen > Dialog > Neu**
- Der vorgegebene Schaltfläche **Abbrechen** kann entfernt werden.
- Mit dem Werkzeug ein Steuerelement vom Typ **Bild** einfügen
- Im Eigenschaftsdialog des Bilder-Steuerelementes (zu öffnen per **Enter**-Taste) ...
  - als **Typ** einstellen: **Symbol**
  - als **Abbild** wählen IDR_MAINFRAME
- Mit dem Werkzeug Steuerelemente vom Typ **Text** zur Beschriftung einfügen

- Im Eigenschaftsdialog zur gesamten Dialogbox eine ID und einen Fenstertitel festlegen, z.B.:

Nun benötigt unser Projekt für die neue Dialogbox eine Klassendefinition und eine Implementation der erforderlichen Methoden, die wir der Übersichtlichkeit halber in eigenen Dateien vornehmen. Erweitern Sie also das Projekt um eine Header-Datei namens **Info.h** mit folgendem Inhalt:

```
#include <afxwin.h>
#include "resource.h"
class CInfo : public CDialog
{
public:
   CInfo(CWnd* pParent = NULL);
};
```

Vergessen Sie bitte das abschließende Semikolon nicht, weil der Compiler anderenfalls in Abhängigkeit von den anschließend zu verdauenden Zeilen ziemlich irreführende Fehlermeldungen produzieren kann.
Erweitern Sie dann Ihr Projekt um eine Implementierungsdatei namens **Info.cpp** mit folgendem Inhalt:

```
#include "Info.h"

CInfo::CInfo(CWnd* pParent) : CDialog(IDD_INFO, pParent)
{
}
```

Ihren Auftritt soll die neue Dialogbox im Rahmen einer Nachrichtenbehandlungsroutine haben, die startet, sobald der Benutzer den Befehl ID_INFO auslöst (per Menü oder **Umschalt+F1**).
Die Nachrichtenroutine muss in der Fensterklasse **CHalloMFCWnd** definiert werden:

```
class CHalloMFCWnd : public CFrameWnd
{
public:
   CHalloMFCWnd();
protected:
   afx_msg int OnCreate(LPCREATESTRUCT lpcs);
   afx_msg void OnPaint();
   afx_msg void Info();
   DECLARE_MESSAGE_MAP()
};
```

Die Nachrichtentabelle dieser Klasse wird entsprechend erweitert:

```
BEGIN_MESSAGE_MAP (CHalloMFCWnd, CFrameWnd)
   ON_WM_CREATE()
   ON_WM_PAINT()
   ON_COMMAND(ID_INFO, Info)
END_MESSAGE_MAP()
```

Nun fehlt noch die Implementation:

```
void CmMFCWnd::Info()
{
   CInfo InfoFenster(this);
   InfoFenster.DoModal();
}
```

Schließlich muss die Header-Datei **Info.h** noch in der Projekt-Headerdatei **HalloMFC.h** inkludiert werden.
Nun sollte in Ihrem Programm die Info-Dialogbox per Menübefehl und über die Tastekombination **Umschalt+F1** zur Verfügung stehen.
Wir haben in der Info-Dialogbox die Angabe zur Programmversion schlicht in einer Label-Komponente untergebracht. Wenn die Versionsangabe auch an anderer Stelle und eventuell sogar in verschiedenen Sprachen benötigt wird, sollte man mit einer Ressource vom Typ Version arbeiten.

Um den Menübefehl zum Beenden des Programms zu aktivieren, müssen Sie lediglich die Nachrichtenbehandlungstabelle unserer Fensterklasse **CHalloMFCWnd** erweitern:

```
BEGIN_MESSAGE_MAP (CHalloMFCWnd, CFrameWnd)
   ON_WM_CREATE()
   ON_WM_PAINT()
   ON_COMMAND(ID_INFO, Info)
   ON_COMMAND(ID_DATEI_BEENDEN, OnClose)
END_MESSAGE_MAP()
```

Die hier eingesetzte Funktion **OnClose()** erbt **CHalloMFCWnd** von der Basisklasse.

10.3.8 Menübefehle situationsgerecht deaktivieren

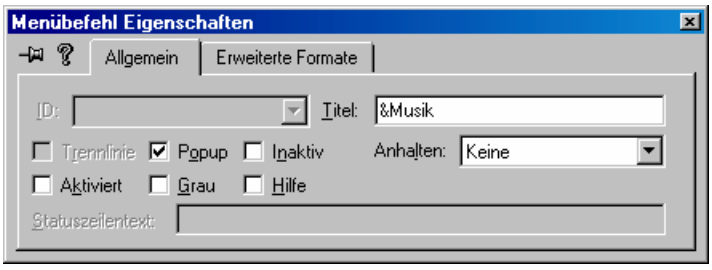
Oft sind Menübefehle unter bestimmten Umständen nicht sinnvoll und müssen daher vorübergehend deaktiviert werden. In Anlehnung an ein Beispiel bei Erlenkötter & Reher (1997) wollen wir unser **HalloMFC**-Programm um ein Menü **Musik** mit folgenden Befehlen erweitern:

- **Musik an/aus**
  Soll die Musikuntermalung ein- oder ausgeschaltet werden?
- **Wahl**
  Bei eingeschalteter Musikuntermalung kann der Benutzer ein Musikstück wählen.

Da wir aus der Methode **CHalloMFCWnd::OnCreate()** wissen, wie man WAV-Dateien per API-Aufruf abspielen lässt, können wir die Musikuntermalung tatsächlich mit relativ geringem Aufwand realisieren, sofern geeignete WAV-Dateien vorhanden sind.

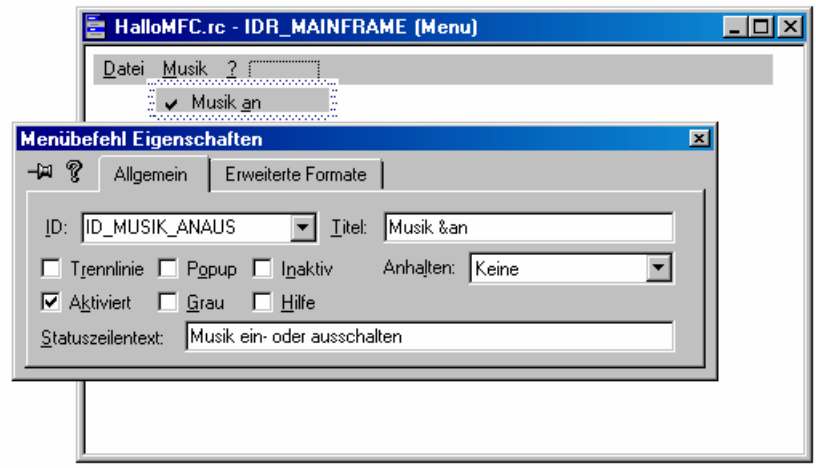
Wir erweitern die Menüvorlage mit der Identifikation IDR_MAINFRAME:

- Das zunächst am rechten Rand erstellte Hauptmenü-Item **Musik**



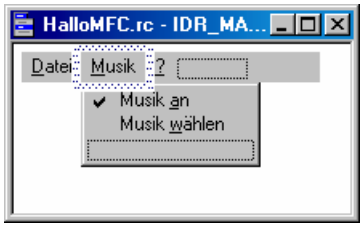
  wird per Maus an eine passende Stelle verschoben.

- Das erste Item im zugehörigen Popup-Menü erhält folgende Eigenschaften:



  Der Menübefehl soll als Umschalter arbeiten und erhält das Initialmerkmal **Aktiviert**, was dem Benutzer durch ein Häkchen in der Menüanzeige signalisiert wird.
- Das zweite Item im Popup-Menü erhält folgende Eigenschaften:

So sieht das erweiterte Menü im Menü-Editor aus:



Um die neuen Menübefehle mit Nachrichtenbehandlungsmethoden zu verknüpfen, wird die Definition unserer Fensterklasse **CHalloMFCWnd** erweitert:

```
class CHalloMFCWnd : public CFrameWnd
{
public:
   CHalloMFCWnd();
protected:
   afx_msg int OnCreate(LPCREATESTRUCT lpcs);
   afx_msg void OnPaint();
   afx_msg void Info();
   afx_msg void MusikAnAus();
   afx_msg void MusikWahl();
   DECLARE_MESSAGE_MAP()
};
```

Routinemäßig erweitern wir die Nachrichtenbehandlungstabelle der Fensterklasse um die Einträge:

```
   ON_COMMAND(ID_MUSIK_ANAUS, MusikAnAus)
   ON_COMMAND(ID_MUSIK_WAHL, MusikWahl)
```

In der boolschen Membervariablen `m_bMusikAnAus` merken wir uns, ob die Musikuntermalung ein oder ausgeschaltet ist. Sie wird im **CHalloMFCWnd**-Konstruktor auf den Wert TRUE initialisiert.

Die Implementationen der beiden neuen Nachrichtenbehandlungsmethoden:

```
void CHalloMFCWnd::MusikAnAus()
{
  if (m_bMusikAnAus)
    m_bMusikAnAus = FALSE;
  else
    m_bMusikAnAus = TRUE;
}

void CHalloMFCWnd::MusikWahl()
{
  MessageBox("Hier könnte eine Auswahl angeboten werden",
    "Wähl was", MB_ICONEXCLAMATION|MB_OK);
}
```

Bei `CHalloMFCWnd::MusikWahl()` beschränken wir uns der Bequemlichkeit halber auf eine Dialogbox, die zudem sehr wenig Arbeit macht, weil sie über die Methode **MessageBox()** der Fensterklasse realisiert wird. Die MFC-Bibliothek enthält übrigens auch eine global verfügbare *Funktion* **AfxMessageBox().**

Aktueller Zwischenstand: Die neuen Menübefehle sind vorhanden, doch von situationsgerechter Verfügbarkeit ist noch nichts zu bemerken. Dazu müssen wir noch Ereignisroutinen erstellen, die von Windows *beim Öffnen* des Menüs ausgeführt werden und dabei Gelegenheit haben, Eigenschaften der Menübefehle anzupassen.
Die Prototypen für die `CHalloMFCWnd`–Definition:

```
afx_msg void UpdateMusikAnAus(CCmdUI* pCmdUI);
afx_msg void UpdateMusikWahl(CCmdUI* pCmdUI);
```

Die Einträge für die `CHalloMFCWnd`–Nachrichtentabelle:

```
ON_UPDATE_COMMAND_UI(ID_MUSIK_ANAUS, UpdateMusikAnAus)
ON_UPDATE_COMMAND_UI(ID_MUSIK_WAHL, UpdateMusikWahl)
```

Die Implementationen:

```
void CHalloMFCWnd::UpdateMusikAnAus(CCmdUI* pCmdUI)
{
  if (m_bMusikAnAus)
  {
    pCmdUI->SetCheck(TRUE);
  }
  else
  {
    pCmdUI->SetCheck(FALSE);
  }
}

void CHalloMFCWnd::UpdateMusikWahl(CCmdUI* pCmdUI)
{
  if (m_bMusikAnAus)
    pCmdUI->Enable(TRUE);
  else
    pCmdUI->Enable(FALSE);
}
```

Auf welches MFC-Objekt wir über den beim Aufruf einer Update-Funktion als Parameter gelieferten Pointer zugreifen, ist von untergeordnetem Interesse. Über die Methoden des Objektes erreichen wir jedenfalls folgende Effekte:

- **SetCheck()**
  Das Häkchen beim Menübefehl **Musik an** wird gesetzt oder beseitigt.
- **Enable()**
  Der Menübefehl wird (de)aktiviert.

Statt die wechselnde Bedeutung des Befehls ID_MUSIK_ANAUS über ein Häkchen zu signalisieren, könnten wir auch die Beschriftung ändern, z.B.:

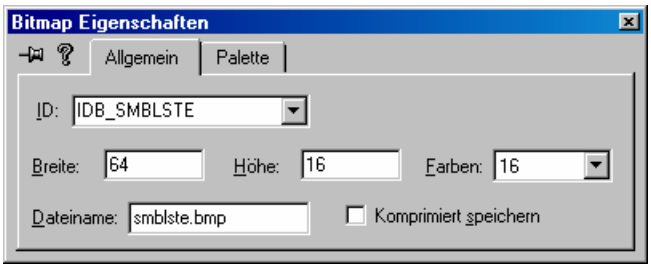
```
pCmdUI->SetText("Musik aus");
```

10.3.9 Symbolleiste

Da unsere Anwendung allmählich Profilook gewinnt, darf auch eine Symbolleiste nicht fehlen. Wir erstellen über

Einfügen > Ressourcen > Bitmap > Neu

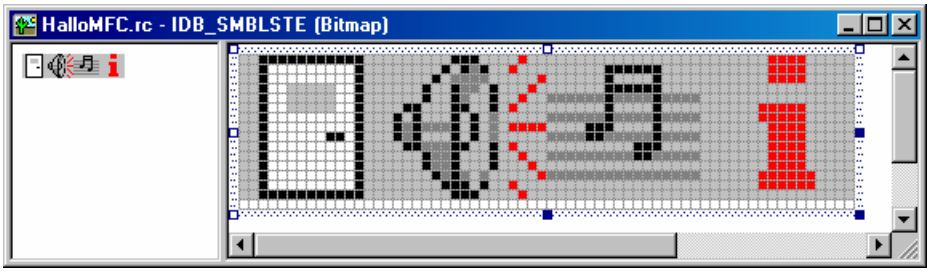
eine Bitmap-Ressouce und legen folgende Eigenschaften fest:



Auf das Rechteck mit insgesamt 64 · 16 Pixeln malen wir sorgfältig vier Bilder mit jeweils 16 · 16 Pixeln für die folgenden vier Menübefehle:

- ID_DATEI_BEENDEN
- ID_MUSIK_ANAUS
- ID_MUSIK_WAHL
- ID_INFO

Jetzt dürfen Sie sich künstlerisch betätigen:



Um die Symbolleiste in unsere Anwendung zu integrieren, muss die Header-Datei **HalloMFC.h** folgendermaßen erweitet werden:

- **afxext.h** inkludieren
- In der Fensterklassen-Definition ein Objekt aus der Klasse **CtoolBar** ergänzen

Die gesamte Header-Datei **HalloMFC.h** sollte wieder mal im Überblick gezeigt werden:

```
#include <afxwin.h>
#include <afxext.h>
#include <mmsystem.h>
#include "resource.h"
#include "Info.h"

class CHalloMFCApp : public CWinApp
{
public:
   BOOL InitInstance();
};

class CHalloMFCWnd : public CFrameWnd
{
public:
   CHalloMFCWnd();
protected:
   BOOL m_bMusikAnAus;
   CToolBar m_ctrlSmblste;
   afx_msg int OnCreate(LPCREATESTRUCT lpcs);
   afx_msg void OnPaint();
   afx_msg void Info();
   afx_msg void MusikAnAus();
   afx_msg void MusikWahl();
   afx_msg void UpdateMusikAnAus(CCmdUI* pCmdUI);
   afx_msg void UpdateMusikWahl(CCmdUI* pCmdUI);
   DECLARE_MESSAGE_MAP()
};
```

Im **CHalloMFCWnd**-Konstruktor wird das Symbolleisten-Tochterfenster `m_ctrlSmblste` folgendermaßen erzeugt und mit Hilfe unserer Bitmap-Ressource IDB_SMBLSTE sowie der Array-Variablen `symbole` vorbereitet:

```
CHalloMFCWnd::CHalloMFCWnd()
{
   m_bMusikAnAus = true;
   LoadFrame(IDR_MAINFRAME);
   UINT symbole[] = {ID_DATEI_BEENDEN,
                     ID_SEPARATOR,
                     ID_MUSIK_ANAUS,
                     ID_MUSIK_WAHL,
                     ID_SEPARATOR,
                     ID_INFO};
   m_ctrlSmblste.Create(this);
   m_ctrlSmblste.LoadBitmap(IDB_SMBLSTE);
   m_ctrlSmblste.SetButtons(symbole, sizeof(symbole)/sizeof(UINT));
}
```

Die UINT-Arrayvariable `symbole` dient zur Beschreibung der Symbolleiste. Als Feldelemente treten die Identifikationen der Menübefehle auf (definiert in **resource.h**) sowie die Konstante ID_SEPARATOR, die für zusätzlichen Abstand an sinnvollen Stellen der Symbolleiste sorgen wird. Dieser Abstand darf in der Bitmap-Ressource *nicht* enthalten sein.

Nun freut sich Smily zu Recht über die nette Anwendung, die einiges zu bieten hat:

Die einzelnen Symbole lösen nicht nur die richtigen Befehle aus, sondern werden sogar von den Menübefehl-Update-Funktionen erfasst.

Über folgende Zusätze für die Symbolleisteninitialisierung im `CHalloMFCWnd`-Konstruktor erhalten wir sogar eine **verschiebbare Symbolleiste**:

```
m_ctrlSmblste.EnableDocking(CBRS_ALIGN_ANY);
EnableDocking(CBRS_ALIGN_LEFT|CBRS_ALIGN_RIGHT|CBRS_ALIGN_TOP);
DockControlBar(&m_ctrlSmblste,AFX_IDW_DOCKBAR_TOP);
```

Im ersten Methodenaufruf wird der Symbolleiste `m_ctrlSmblste` erlaubt, an beliebiger Stelle anzudocken. Der nächste Methodenaufruf richtet sich an unser Fensterobjekt und weist es an, andockende Tochterfenster nur links, oben und rechts zu akzeptieren. Im letzten Methodenaufruf wird das Fensterobjekt angewiesen, mit oben angedockter Symbolleiste `m_ctrlSmblste` zu starten.

Unsere Anwendung mit links angedockter Symbolleiste:

10.3.10 Statuszeile

Auch eine Statuszeile wird als Tochterfenster der Fensterklasse, muss also in deren Definition aufgeführt werden:

```
CStatusBar m_ctrlStatuslste;
```

Inhalt und räumlicher Anordnung der Statuszeile werden (wie bei der Symbolleiste) über eine UINT-Arrayvariable definiert:

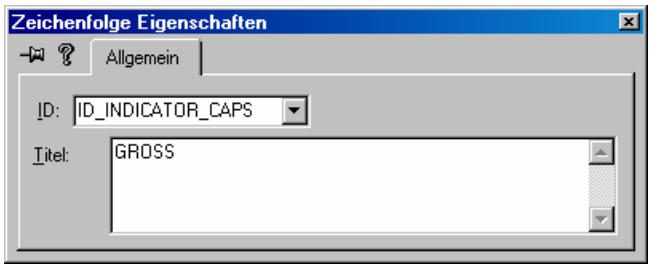
```
UINT status[] =  {ID_SEPARATOR,
                  ID_INDICATOR_NUM,
                  ID_INDICATOR_CAPS,
                  ID_INDICATOR_SCRL};
```

Mit dem führenden Separator werden die ständigen Insassen der Statuszeile (Indikatoren für den Numerik-, Umschalt- und Rollmodus der Tastatur) an den rechten Rand verbannt.

Neben der `status`-Definition sind im Konstruktor der Fensterklasse noch folgende Methodenaufrufe erforderlich:

```
m_ctrlStatuslste.Create(this);
m_ctrlStatuslste.SetIndicators(status, sizeof(status)/sizeof(UINT));
```

Wenn Ihnen die vorgegebenen Beschriftungen für die Indikatoren in der Statusanzeige nicht gefallen, müssen Sie die Zeichenfolgentabelle öffnen und nach **Einfügen > Neue Zeichenfolge** für die betroffenen IDs eine Neudefinition vornehmen, z.B.:

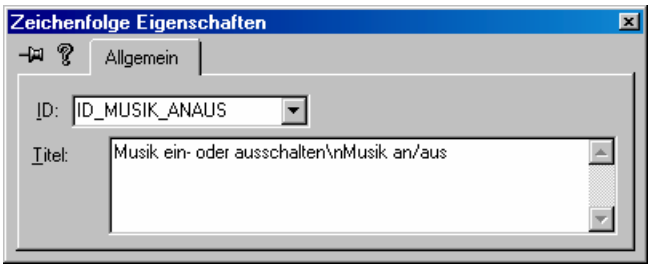


Schließlich können wir noch den **Creator()**-Aufruf für die *Symbolleiste* verbessern, damit die Statuszeilentexte zu den Menübefehlen nicht nur beim Öffnen der Menüs erscheinen, sondern auch beim kurzen Verharren des Mauszeigers auf den zugehörigen Symbolen:

```
m_ctrlSmblste.Create(this, WS_CHILD|WS_VISIBLE|CBRS_TOP|CBRS_FLYBY);
```

10.3.11 Tooltips anzeigen

Damit beim kurzen Verharren des Mauszeigers auf einem Symbol ein Tooltip auf gelbem Hintergrund erscheint, müssen Sie in der Zeichenfolgentabelle die Texte zu den Befehlen um einen mit „\n“ abgetrennten Zusatz erweitern (möglichst kurz), z.B.:



Außerdem muss der **Creator()**-Aufruf für die Symbolleiste nochmals erweitert werden:

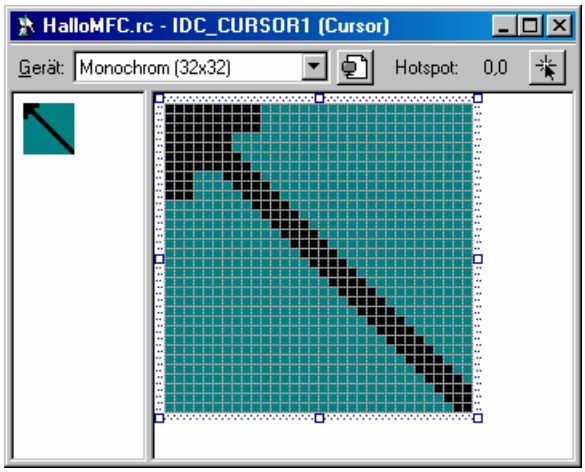
```
m_ctrlSmblste.Create(this,
  WS_CHILD|WS_VISIBLE|CBRS_TOP|CBRS_FLYBY|CBRS_TOOLTIPS);
```

10.3.12 Ein eigener Mauszeiger (Cursor)

Soll der Mauszeiger eine spezielle Form annehmen, sobald er sich über unserem Anwendungsfenster befindet, dann muss eine Cursor-Ressource erstellt und dem Fenster zugeordnet werden.
Nach

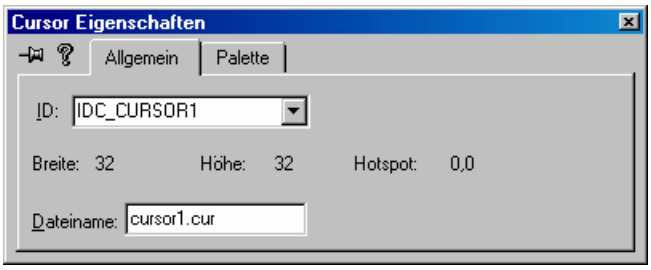
Einfügen > Ressource > Cursor > Neu

kann die 32×32-Schwarz-Weiß-Bitmap eines neuen Mauszeigers gestaltet werden, z.B.:



Die vorgegebene Hintergrundfarbe wird zur Laufzeit transparent dargestellt und sollte daher beibehalten werden. In der Regel wird man auch die Position (0,0) (= links oben) für den **Hotspot**, der die genaue Trefferstelle eines Mausklicks definiert, nicht ändern. Ist dies doch erwünscht, dann steht folgendes Werkzeug zur Verfügung: 

In der Eigenschaftsdialogbox zur Cursor-Ressource kann man die Identifikation und den Dateinamen festlegen, z.B.:



Leider kann man in der **CFrameWnd**-Methode **LoadFrame()** einem neuen Fenster via Mehrfachressource zwar eines Titel, ein Icon, ein Menü und eine Tabelle mit Tastenbefehlen verpassen, aber keinen eigenen Mauszeiger. Daher müssen wir zur alternativen, etwas aufwändigeren Methode **Create()** greifen, die Zugriff auf *alle* Fenster-Kreationsparameter bietet.
Zur Erinnerung: Mit **LoadFrame()** bzw. **Create()** wird das Windows-Fenster zu einem MFC-Fensterobjekt erzeugt.
Die **Create()**-Methode greift noch direkt auf das Konzept der **WNDCLASS** zurück, wobei es sich um eine Struktur zur Definition einer Fenstersorte handelt, die uns bei der Beschreibung von Windows-Programmen auf SDK-Basis begegnet ist (siehe Seite 195). Dort war auch zu erfahren, dass eine **WNDCLASS** vor Verwendung zu registrieren ist, wozu bei MFC-Programmen die globale Funktion **AfxRegisterWndClass()** dient.
Nun zum **Create()**-Aufruf, der in unserem **CmMFCWnd**-Konstruktor den **LoadFrame()**-Aufruf ersetzt und dabei auch die Cursor-Ressource zum Einsatz bringt:

```
CString Titel;
Titel.LoadString(IDR_MAINFRAME);
Create(AfxRegisterWndClass(CS_DBLCLKS | CS_HREDRAW | CS_VREDRAW,
     AfxGetApp()->LoadCursor(IDC_CURSOR1),
     (HBRUSH) (COLOR_WINDOW+1),
     AfxGetApp()->LoadIcon(IDR_MAINFRAME)),
     Titel,
     WS_OVERLAPPEDWINDOW,
     rectDefault,
     NULL,
     MAKEINTRESOURCE(IDR_MAINFRAME));
LoadAccelTable(MAKEINTRESOURCE(IDR_MAINFRAME));
```

Die von **Create()** benötigte WNDCLASS-Struktur wird von **AfxRegisterWndClass()** erstellt, wobei wir die Parameter dieser globalen Funktion folgendermaßen versorgen:

- **UINT** *nClassStyle*
  Für die zu registrierende WNDCLASS werden (verknüpft mit dem bitweisen-Oder-Operator) folgende Klassenstile vereinbart:
  - CS_DBLCLKS
    Die Fenster sollen Doppelklicks (z.B. auf die Titelzeile) erkennen.
  - CS_HREDRAW, CS_VREDRAW
    Bei horizontalen bzw. vertikalen Änderungen der Größe wird der Fensterinhalt neu gezeichnet.
- **HCURSOR** *hCursor* **= 0**
  Den erforderlichen Cursor-Handle liefert in unserem Beispiel die **CWinApp**-Methode **LoadCursor()** aufgrund der von uns vereinbarten Ressourcen-Identifikation. Um unsere Anwendung ansprechen zu können, verwenden wir die globale MFC-Funktion **AfxGetApp()**.
- **HBRUSH** *hbrBackground* **= 0**
  Den Handle auf einen Fensterhintergrund erhalten wir durch Anwendung der expliziten Typumwandlung (**HBRUSH**) auf eine Farbnummer.
- **HICON** *hIcon* **= 0**
  Den Icon-Handle verschaffen wir uns analog zum Cursor-Handle.

Die weiteren **Create()**-Parameter sind uns teilweise schon in früheren Beispielen begegnet:

- **LPCTSTR** *lpszWindowName*
  Statt den Fenstertitel (wie in Abschnitt 10.2) als Zeichenfolgen-Literal einzutragen, übernehmen wir ihn aus dem **CString**-Objekt `Titel`, das sich vor dem **Create()**-Aufruf mit seiner Methode **LoadString()** aus der Mehrfachressource IDR_MAINFRAME bedient.
- **DWORD** *dwStyle* **= WS_OVERLAPPEDWINDOW**
  Neben den *Klassen*stilen gibt es für das nun zu erzeugende konkrete Fenster auch noch *Fenster*stile. Mit **WS_OVERLAPPEDWINDOW** wird ein Fenster gewählt, das über Rahmen, Beschriftung, Systemmenü, Verkleinerungs-, Vergrößerungs- und Beendigungsschalter verfügt.
- **const RECT&** *rect* **= rectDefault**
  Mit diesem Parameter werden Größe und Position des neuen Fensters festgelegt. Während wir bei Verwendung von **LoadFrame()** die Standardgröße hinnehmen müssen, können wir hier endlich wieder einen für unsere Anwendung besser geeigneten Wert einstellen.
  Mit der in Abschnitt 10.2 verwendeten Größe (`CRect(300, 200, 600, 400)`) gibt es leichten Kummer, weil die Tastatur-Indikatoren nicht mehr ganz zu sehen sind. Ursache ist die Breite der Statusleisten-Hauptzone, die aber mit **CStatusBar::SetPaneInfo()** angepasst werden kann, z.B.:

  ```
  m_ctrlStatuslste.SetPaneInfo(0,0,SBPS_STRETCH,0);
  ```

  Mit dem Standardparameterwert **rectDefault** (eine statische **CFrameWnd**-Elementvariable aus der Klasse **CRect**) überlässt man Windows die Entscheidung über die initiale Größe und Position des Fensters.
- **CWnd*** *pParentWnd* **= NULL**
  Ein Elternfenster gibt es in unserem Beispiel nicht.
- **LPCTSTR** *lpszMenuName* **= NULL**
  Das Makro **MAKEINTRESOURCE()** übersetzt die Ressourcen-ID der Menüvorlage in die von **Create()** benötigte Textform.

Die Accelerator-Tabelle wird nicht im **Create()**-Aufruf, sondern über die **CFrameWnd**-Methode **LoadAccelTable()** geladen.

Nach diesen Mühen verfügt unser Programm über einen eigenen Mauszeiger:

10.3.13 Übungsaufgaben zu Abschnitt 10.3

1) Sorgen Sie dafür, dass sich unser HalloMFC-Programm beim Eintreffen einer WM_CLOSE-Nachricht beim Benutzer erkundigt, ob er wirklich schon aufhören will, z.B. mit folgender Dialogbox:

Es bietet sich an, das Problem mit der Nachrichtenbehandlungsmethode **CFrameWnd::OnClose()** zu lösen. Diese ist bereits in der Nachrichtentabelle unserer Fensterklasse eingetragen und muss nun geeignet überschrieben werden. Obige Dialogbox lässt sich z.B. mit der **CWnd**-Methode **MessageBox()** realisieren. In der Online-Hilfe (MSDN-Library) erfahren Sie,

- welcher **MessageBox**-Stil zu wählen ist, damit in der MessageBox die Schalter **Ja** und **Nein** angeboten werden,
- welchen Rückgabewert die Methode **MessageBox()** liefert, wenn der Benutzer auf **Ja** klickt.

1) Auch Dialogboxen können mit einem Menü ausgestattet werden, wobei die entsprechende Ressourcen-ID in der Eigenschaftsdialogbox festgelegt wird, z.B.:

Erweitern Sie unser dialogfeldbasiertes Programm **Bruno** um einige sinnvolle Menüs.

10.4 Document/View-Anwendungen

10.4.1 Überblick

Bei der ab MFC-Version 2.0 eingeführten Document/View-Anwendungsarchitektur werden die Daten einer Anwendung in einem **Document**-Objekt gespeichert, während für die Anzeige der Daten ein **View**-Objekt (dt.: Ansicht-Objekt) zuständig ist.
Mit Hilfe der Document/View-Architektur wird eine verbesserte Arbeitsteilung der Softwarekomponenten und ein höherer Grad an Modularität erreicht. Dies erlaubt eine weitere Rationalisierung bei der Softwareerstellung. So können z.B. große Teile der Benutzerinteraktion beim Öffnen, Sichern oder Drucken von Dokumenten über Standarddialoge abgewickelt werden. Wenn Sie sich gleich um neue Begriffe bemühen müssen, dürfen Sie sich daher mit dem folgenden Prosise-Zitat trösten (1999, S. 539):

Doc + View = Less Work for You

MFC unterstützt zwei Typen von Document/View-Anwendungen:

1. **Einzelnes Dokument (SDI)**
   Bei SDI-Anwendungen (Single Document Interface) kann zu einem Zeitpunkt nur ein Dokument geöffnet sein, zu dem auch nur eine Ansicht existiert. Beispiel für eine SDI-Anwendung: Der Windows-Editor Notepad
2. **Mehrere Dokumente (MDI)**
   Mit MDI-Anwendungen (Multiple Document Interface) können mehrere Dokumente simultan geöffnet werden, wobei auch noch jeweils verschiedene Views eines Dokumentes möglich sind. Beispiel für eine MDI-Anwendung: Microsoft Excel.

SDI-Anwendungen können leicht zu MDI-Anwendungen ausgebaut werden. Dies ist aber nicht unbedingt erforderlich, weil das SDI-Prinzip kaum Einschränkungen mit sich bringt: Wenn der Anwender eines SDI-Programms mehrere Dokumente simultan verarbeiten möchte, hat er meist die Möglichkeit, mehrere Instanzen des Programms zu starten. Es zeichnet sich sogar ein Trend zur SDI-Anwendung ab. So haben sich z.B. die Entwickler von Microsoft Word seit Office 2000 vom lange gepflegten MDI-Prinzip abgewandt und präsentieren nun jeden Text im eigenen Fenster. Wir werden uns daher in der begrenzten Zeit dieses Kurses ausschließlich mit SDI-Anwendungen beschäftigen.

Bei einer SDI-Anwendung sind die folgenden vier Objekte bzw. Klassen beteiligt:

1. Das Anwendungsobjekt aus der Klasse **CWinApp**
   Es erzeugt die übrigen drei Objekte, betreibt die Nachrichtenschleife und beteiligt sich auch an der Nachrichtenverarbeitung.
2. Das Dokumentenobjekt aus der Klasse **CDocument**
   Das Dokumentenobjekt beinhaltet in seinen Membervariablen während der Laufzeit die Daten der Anwendung, speichert sie in eine externe Datei und kann sie auch von dort öffnen. Bei konsequent objektorientiertem Programmierstil sind die Daten gekapselt (Schutzstufe **privat** oder **protected**) und für das auf Benutzeranforderungen reagierende Ansichtsobjekt nur über **public**-Methoden zugänglich.
3. Das Ansichtsobjekt (View-Objekt) aus der Klasse **CView (**oder aus einer abgeleiteten Klasse**)**
   Das Ansichtsobjekt ist ein Tochterfenster des Hauptfensters und belegt genau dessen Klientenbereich (abzüglich Platz für Symbol- und Statusleisten). Es ist für die Anzeige des Dokumentes und für die Benutzerinteraktion zuständig. Tastatur- und Maus-Nachrichten gelangen zum Ansichtsobjekt und werden von diesem bei Bedarf in Befehle an das Dokumentenobjekt umgesetzt.

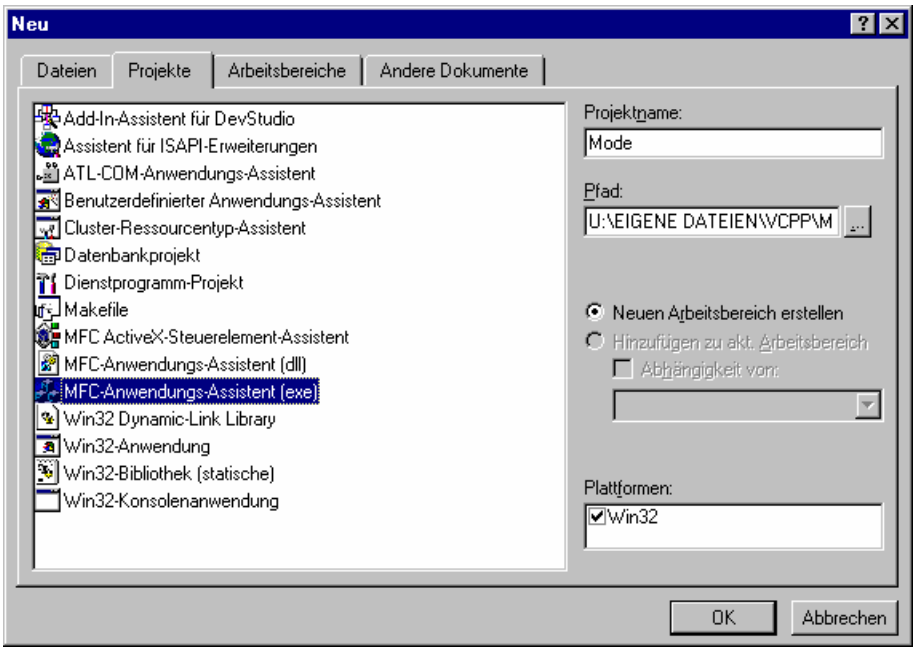
4. Das Hauptfenster aus der Klasse **CFrameWnd**
   Es hat normalerweise den Fensterstil WS_OVERLAPPEDWINDOW (größenveränderlicher Rahmen, Titelzeile, Systemmenü, Schalter zum Minimieren, Maximieren und Schließen).
   Das Hauptfenster bildet den Rahmen für das Ansichtsobjekt, die Symbolleisten und die Statusleiste. Es koordiniert diese Tochterfenster (z.B. bei Veränderung der Fenstergröße) und besitzt auch einige wichtige Nachrichten- Behandlungsroutinen.

10.4.2 Einstieg mit dem Anwendungsassistenten

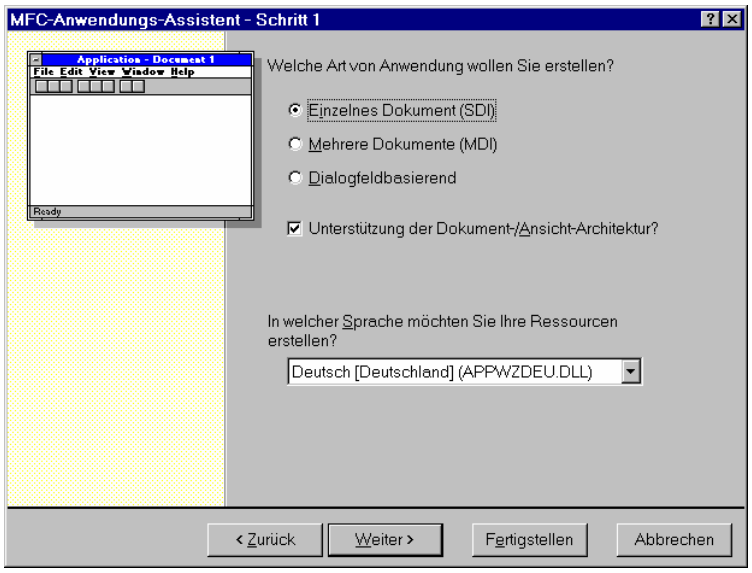
Als einfaches Beispiel für eine SDI-Anwendung wollen wir einen *Mosaik-Designer* erstellen, mit dem sich 16×16 - Mosaiken gestalten, speichern und auch wieder öffnen lassen.
Wir verlangen nach

Datei > Neu > Projekte

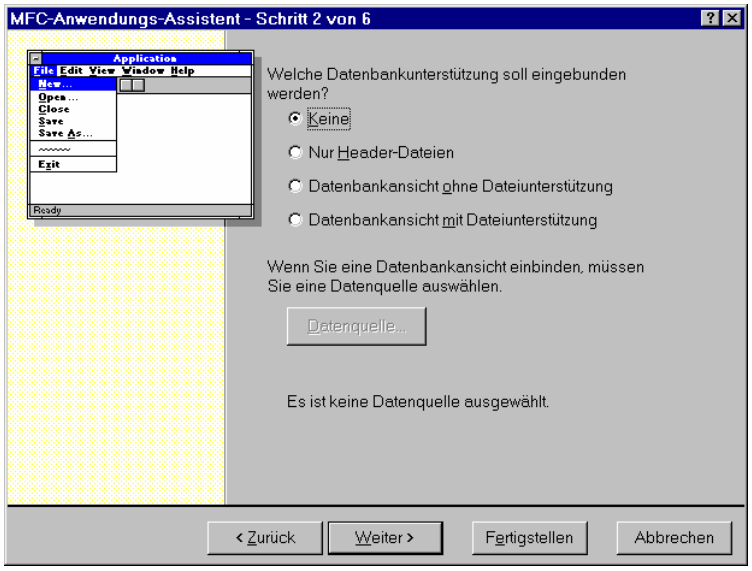
den **MFC-Anwendungsassistenten** und wählen einen **Projektnamen**:



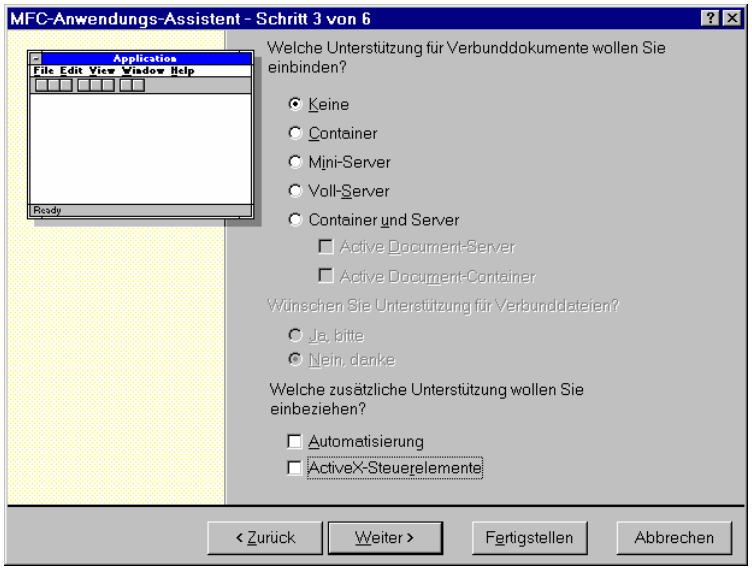
Im ersten Assistentenschritt entscheiden wir uns für eine SDI-Anwendung und behalten Sie die **Unterstützung der Dokument-/Ansicht–Architektur** bei:



Die Dialogbox von Schritt 2 übernehmen wir ohne Änderungen:

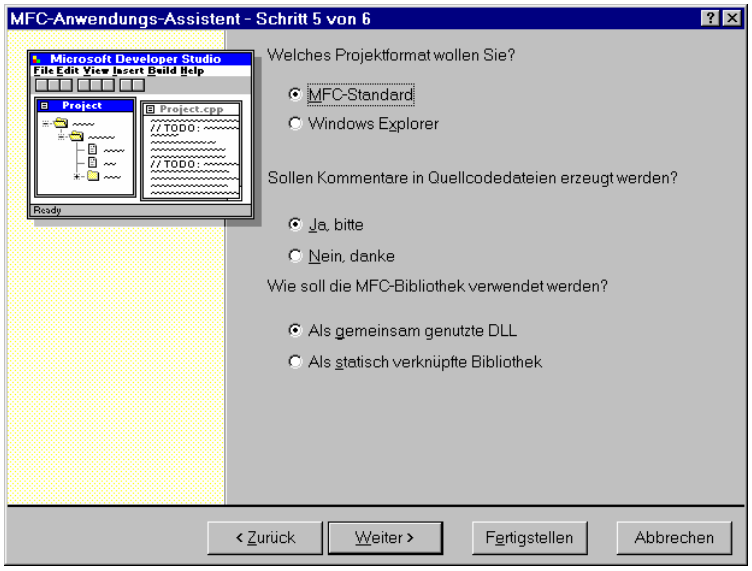


Im dritten Schritt deaktivieren wir die Unterstützung für **ActiveX-Steuerelemente**:

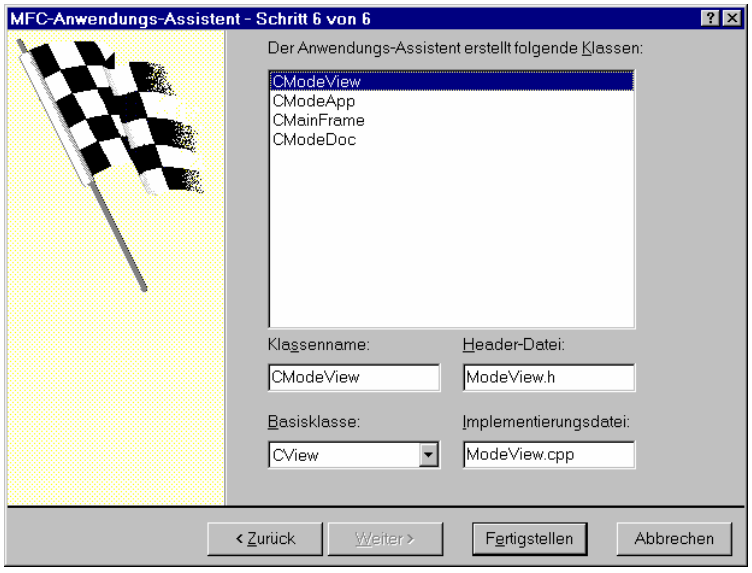


Wenngleich wir die Voreinstellungen von Schritt 4 nur teilweise benötigen, lassen wir diese Dialogbox unverändert:

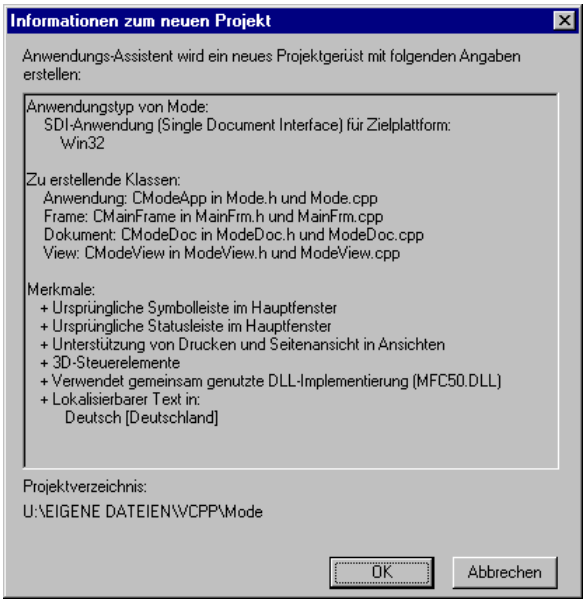
Ebenso unverändert lassen wir die nächste Dialogbox:



Im letzten Schritt akzeptieren wir die vorgeschlagenen Klassenbezeichnungen und die Basisklasse **CView** für unsere Ansichtsklasse:



Nach dem **Fertigstellen** in Schritt 6 und einem **OK** zur folgenden Zusammenfassung wird das Projekt erstellt.



Der Anwendungs-„Rohling“ zeigt folgendes Erscheinungsbild:

In den folgenden Abschnitten beschäftigen wir uns mit dem Aufbau und der Funktionsweise einer Document-/View – Anwendung. Dabei wird der vom Anwendungsassistenten erzeugte Quellcode analysiert und erweitert.

10.4.3 Die Rolle des Anwendungsobjekts

Mit Hilfe des vom Anwendungsassistenten erzeugten Quellcodes in **Mode.cpp** kann die Rolle des Anwendungsobjekts (von der Klasse **CWinApp** abstammend) recht gut beschrieben werden. Es übernimmt wie bei den früheren MFC-Programmen wesentliche Aufgaben in der **Initialisierungsphase** und bei der **Nachrichtenverarbeitung**.
Mit der Initialisierungsphase werden wir uns in diesem Abschnitt noch ausführlich beschäftigen. Wesentliche Aspekte der SDI-Nachrichtenverarbeitung werden später behandelt, doch soll schon jetzt darauf hingewiesen werden, dass sich das Anwendungsobjekt anders als in früheren Programmen nicht mehr auf das Betreiben der Nachrichtenschleife beschränkt, sondern auch an der *Verarbeitung* von Nachrichten beteiligt ist, was die Anwesenheit einer **MESSAGE_MAP** in der Klassenimplementation belegt:

```
BEGIN_MESSAGE_MAP(CModeApp, CWinApp)
   ON_COMMAND(ID_APP_ABOUT, OnAppAbout)
   ON_COMMAND(ID_FILE_NEW, CWinApp::OnFileNew)
   ON_COMMAND(ID_FILE_OPEN, CWinApp::OnFileOpen)
   ON_COMMAND(ID_FILE_PRINT_SETUP, CWinApp::OnFilePrintSetup)
END_MESSAGE_MAP()
```

Der Übersichtlichkeit halber wurden die Kommentare aus diesem **Mode.cpp** - Ausschnitt entfernt, auch die zum Schutz der Klassenassistent-Aktivitäten gedachten Kommentare.

Seine Initialisierungsaufgaben erledigt das Anwendungsobjekt vor allem in der Methode **InitInstance()**, während der Konstruktor in der Regel keine großen Aktivitäten entfaltet. In unserem Beispiel hat der Anwendungsassistent im Konstruktor nur Kommentare hinterlassen:

```
CModeApp::CModeApp()
{
   // ZU ERLEDIGEN: Hier Code zur Konstruktion einfügen
   // Alle wichtigen Initialisierungen in InitInstance platzieren
}
```

Konnte sich in unseren früheren MFC-Programmen die Methode **IniInstance()** darauf beschränken, das Anwendungsfenster (aus **CFrameWnd** oder **CDialog** abgeleitet) zu erzeugen und zu aktivieren (mit **ShowWindow()** bzw. **DoModal()**), gibt es bei SDI-Anwendungen etwas mehr zu tun.
Grob skizziert geht das Anwendungsobjekt einer SDI-Anwendung in der Methode **InitInstance()** folgendermaßen vor:

Es analysiert die beim Programmstart übergebene Kommandozeile. Ist dort ein zu öffnendes Dokument angegeben, führt es die Methode **CWinApp::OpenDocumentFile()** aus, ansonsten kommt die Methode **CWinApp::OnFileNew()** zum Einsatz.

Im weiteren Ablauf, der am Beispiel von **OnFileNew()** näher betrachtet werden soll, spielt die Klasse **CSingleDocTemplate** eine entscheidende Rolle. Ein Objekt dieser Klasse enthält alle in einer SDI-Anwendung vom MFC-Framework benötigten Informationen über die unterstützte Dokumentenart. Um welche Informationen es sich dabei handelt, zeigt der folgende Ausschnitt aus der vom Anwendungsassistenten erstellten **InitInstance()**-Methode:

```
CSingleDocTemplate* pDocTemplate;
pDocTemplate = new CSingleDocTemplate(
   IDR_MAINFRAME,
   RUNTIME_CLASS(CModeDoc),
   RUNTIME_CLASS(CMainFrame),
   RUNTIME_CLASS(CModeView));
AddDocTemplate(pDocTemplate);
```

Dem Konstruktor für die neue Dokumentenvorlage werden übergeben:

- Eine Gruppen-Ressourcen-ID
  Sie kennzeichnet u.a. ein Menü und eine Accelerator-Tabelle und kommt beim Erstellen eines Windows-Rahmenfensters via **LoadFrame()** zum Einsatz.
- Ein Pointer auf die **CRuntimeClass** – Struktur für die Dokumentenklasse der Anwendung (bei uns: `CModeDoc`), den das Makro **RUNTIME_CLASS()** liefert
  In dieser Struktur sind u.a. folgende Informationen über die Klasse enthalten:
  - Name
  - Größe eines Objektes
  - Versionsnummer für die Serialisierung (Speicherung) von Dokumentenobjekten
  - Funktionspointer auf den voreingestellten Konstruktor
- Ein Pointer auf die **CRuntimeClass** – Struktur für die Fensterklasse der Anwendung (bei uns: `CMainFrame`)
- Ein Pointer auf die **CRuntimeClass** – Struktur für die Ansichtsklasse der Anwendung (bei uns: `CModeView`)

Mit Hilfe der per **AddDocTemplate()** registrierten Dokumentenvorlage kann das Anwendungsobjekt in **CWinApp::OnFileNew()** das Dokumentenobjekt, das Fensterobjekt und das Ansichtsobjekt erzeugen (in dieser Reihenfolge).
Der Aufruf von **OnFileNew()** findet in der Methode **ProcessShellCommand()** statt, die das von **ParseCommandLine()** initialisierte **CCommandLineInfo**-Objekt auswertet:

```
CCommandLineInfo cmdInfo;
ParseCommandLine(cmdInfo);
if (!ProcessShellCommand(cmdInfo))
   return FALSE;
```

Wenn der Prozess erfolgreich verlaufen ist, wird in **InitInstance()** das Anwendungsfenster angezeigt und aktualisiert (samt der Ansichts-Tochter):

```
m_pMainWnd->ShowWindow(SW_SHOW);
m_pMainWnd->UpdateWindow();

   return TRUE;
```

Nachdem sich **InitInstance()** mit **return** verabschiedet hat, ruft **AfxWinMain()** ruft die **Run()**-Methode des Anwendungsobjektes auf, welche die Nachrichtenschleife ausführt (vgl. Abschnitt 10.2.1).

Mittlerweile haben wir die wesentlichen Bestandteile der von Anwendungsassistenten produzierten Datei **Mode.cpp** diskutiert. In den restlichen Zeilen wird die Klasse **CAboutDlg()** implementiert, die in **CModeApp::OnAppAbout()** für die Info-Anzeige verwendet wird:

```
void CModeApp::OnAppAbout()
{
   CAboutDlg aboutDlg;
   aboutDlg.DoModal();
}
```

Mit dem Wissen aus Abschnitt 10.3.1 dürften Ihnen die Zeilen zur Definition und zur Verwendung der Dialogbox keine Rätsel aufgeben.

10.4.4 Komplettierung der Dokumentenklasse

Während wir die Anwendungsklasse `CModeApp` sowie die Hauptfensterklasse `CMainFrame` unverändert lassen können, wird über die Dokumentenklasse `CModeDoc` sowie die Ansichtsklasse `CModeView` die individuelle Funktionalität unserer Anwendung realisiert.

10.4.4.1 Membervariablen für die Anwendungsdaten

Für die Verarbeitung der Daten des Programms (z.B. Lesen, Verändern, Sichern) ist ausschließlich das Dokumentenobjekt zuständig. Es kapselt die Daten und bietet dem Ansichtsobjekt über **public**-Methoden die erforderlichen Schnittstellen an.
Wir ergänzen in der `CModeDoc`-Klassendefinition eine Array-Elementvariable vom Typ COLORREF für die Daten eines Dokumentenobjektes:

```
protected:
  COLORREF m_clrMosaik[16][16];
```

Diese Matrix mit 32-Bit-RGB-Farbdefinitionen als Einträgen stellt den Inhalt eines Mode-Dokumentes dar. Wir werden bequeme Methoden kennen lernen, diese Matrixwerte in eine Datei zu sichern bzw. daraus zu laden.

10.4.4.2 Methoden für die Dokument-Initialisierung und -Serialisierung

In der Basisklasse **CDocument** unserer Dokumentenklasse sind einige virtuelle Methoden definiert, die vom MFC-Anwendungsgerüst automatisch aufgerufen werden und in der Regel in jeder Anwendung überschrieben werden müssen, damit ein sinnvolles Verhalten zustande kommt. Dazu zählen die Methoden:

- **OnNewDocument()**
  Sie wird beim Erstellen eines neuen Dokumentes ausgeführt, also z.B. beim Start des Programms oder nach dem Menübefehl **Datei > Neu**.
- **Serialize()**
  Diese Methode ist für das Sichern und Laden eines Dokumentes zuständig.

Ihre Prototypen im Rahmen der `CModeDoc`-Definition sind automatisch eingefügt und über spezielle Kommentare gekennzeichnet worden:

```
// Überladungen
   // Vom Klassenassistenten generierte Überladungen virtueller Funktionen
   //{{AFX_VIRTUAL(CModeDoc)
   public:
   virtual BOOL OnNewDocument();
   virtual void Serialize(CArchive& ar);
   //}}AFX_VIRTUAL
```

Während wir diese Passage der Headerdatei **ModeDoc.h** in der Regel nicht manuell bearbeiten müssen, ist bei den Implementierungen der beiden Methoden in der Quellcodedatei **ModeDoc.cpp** Handarbeit unvermeidlich. Ein besonders schneller Weg an die richtige Stelle Quellcodedatei führt über das Klassen-Registerblatt des Arbeitsbereichsfensters:

- Öffnen Sie nötigenfalls die Liste mit den Elementen der Klasse `CModeDoc`.
- Setzen Sie einen Doppelklick auf den Eintrag der gewünschten Methode.

In der **OnNewDocument()**-Methode initialisiert man die Membervariablen der Dokumenten-Klasse. Im `Mode`-Beispiel sollen alle Mosaikteile schwarz eingefärbt werden. An welcher Stelle unser Beitrag erwünscht ist, wird durch einen Kommentar deutlich gemacht:

```
BOOL CModeDoc::OnNewDocument()
{
  if (!CDocument::OnNewDocument())
    return FALSE;
  // ZU ERLEDIGEN: Hier Code zur Reinitialisierung einfügen
  // (SDI-Dokumente verwenden dieses Dokument)
  for (int i = 0; i < 16; i++)
    for (int j = 0; j < 16; j++)
      m_clrMosaik[i][j] = RGB(0,0,0);

  return TRUE;
}
```

Am Anfang muss unbedingt der Basisklasse **CDocument** Gelegenheit gegeben werden, die ihrerseits erforderlichen Initialisierungen auszuführen.

In der **Serialize()**-Methode der Dokumenten-Klasse erfolgt das Speichern in eine Datei bzw. das Laden aus einer Datei mit den (überladenen) Operatoren << bzw. >> ebenso einfach wie in unseren früheren Konsolen-Programmen die Aus- bzw. Eingabe mit den **iostream**-Objekten **cout** und **cin**:

```
void CModeDoc::Serialize(CArchive& ar)
{
  if (ar.IsStoring())
  {
    // ZU ERLEDIGEN: Hier Code zum Speichern einfügen
    for (int i = 0; i < 16; i++)
      for (int j = 0; j < 16; j++)
        ar << m_clrMosaik[i][j];
  }
  else
  {
    // ZU ERLEDIGEN: Hier Code zum Laden einfügen
    for (int i = 0; i < 16; i++)
      for (int j = 0; j < 16; j++)
        ar >> m_clrMosaik[i][j];
  }
}
```

Indem uns das Anwendungsgerüst zum Speichern und Laden ein bequemes **Archiv**-Objekt aus der MFC-Klasse **CArchive** zur Verfügung stellt, bleiben uns Routinearbeiten wie das Öffnen und Schließen von Dateien erspart.

In der **Serialize()**-Methode der Dokumenten-Klasse haben wir deswegen leichtes Spiel, weil Variablen vom Typ **COLORREF** (alias **DWORD**, vorzeichenfreie Ganzzahl mit 32 Bit) bequem mit den (überladenen) Operatoren << bzw. >> (de)serialisiert werden können.
Wenn Instanzen selbst erstellter Klassen mit der selben Leichtigkeit zwischen Dateien und Hauptspeicher bewegt werden sollen, dann müssen diese Klassen **serialisiert** werden (siehe z.B. Wigard 1999, S. 404ff).

10.4.4.3 Methoden für die Interaktion mit dem Ansichtsobjekt

Um dem Ansichtsobjekt lesenden und schreibenden Zugriff auf die Daten im Dokumentenobjekt zu ermöglichen, definieren wir in der Dokumentenklasse die folgenden Methoden (mit Schutzstufe **public**!):

```
  public:
    COLORREF GetCellColor(int i, int j);
    void SetCellColor(int i, int j, COLORREF color);
```

Ihre Implementierungen setzen wir ans Ende der Datei **ModeDoc.cpp**:

```
COLORREF CModeDoc::GetCellColor(int i, int j)
{
  return m_clrMosaik[i][j];
}

void CModeDoc::SetCellColor(int i, int j, COLORREF color)
{
  m_clrMosaik[i][j] = color;
  SetModifiedFlag(TRUE);
  UpdateAllViews(NULL);
}
```

Nachdem das Dokumentenobjekt (meist im Auftrag des Ansichtsobjektes) seine Daten verändert hat, informiert es mit **SetModifiedFlag()** das Framework darüber, damit der Benutzer bei Bedarf zum Speichern aufgefordert werden kann (z.B. beim Beenden des Programms oder vor dem Öffnen eines anderen Dokumentes). Auch hier zeigt sich, dass MFC-Programmierer mit minimalem Aufwand benutzerfreundliche und professionell auftretende Anwendungen erstellen können.
Außerdem teilt das Dokumentenobjekt mit **UpdateAllViews()** dem Ansichtsobjekt mit, dass die Änderungen der Daten eine Aktualisierung der Anzeige erfordern. Obwohl bei einer SDI-Anwendung nur eine einzige Ansicht vorhanden ist, wird die selbe Methode verwendet wie bei einer MDI-Anwendung, die ja tatsächlich mehrere Ansichten zu einem Dokumentenobjekt unterstützt.

10.4.4.4 Dynamische Objekt-Erstellung

Nachdem wir uns bisher mit den Membervariablen der Document-Klasse zur Aufnahme der Anwendungsdaten sowie mit wichtigen Methoden zur Interaktion mit dem Framework und mit der Ansichtsklasse beschäftigt haben, sind noch einige Erläuterungen zur dynamischen Erstellung des Dokumentenobjektes durch das Framework angebracht.
Oben wurde berichtet, dass sich das Anwendungsobjekt in der **InitInstance()**-Methode eine Dokumentenvorlage erstellt, die u.a. einen Pointer auf die **CRuntimeClass** – Struktur der Dokumentenklasse enthält. Damit eine Klasse eine solche Membervariable vom Strukturtyp **CRuntimeClass** überhaupt besitzt, sind folgende Voraussetzungen erforderlich:

- Die Klasse muss von **CObjekt** abstammen (direkt oder indirekt).
- In der Klassendeklaration muss das Makro **DECLARE_DYNCREATE()** aufgerufen werden, z.B. bei `CModeDoc`:

  ```
  DECLARE_DYNCREATE(CModeDoc)
  ```

  Einziger Parameter für dieses Makro ist der Klassenname.
- In der Klassenimplementation muss das Makro **IMPLEMENT_DYNCREATE()** aufgerufen werden, z.B. bei `CModeDoc`:

  ```
  IMPLEMENT_DYNCREATE(CModeDoc, CDocument)
  ```

  Als Parameter sind der Klassenname und der Basisklassenname anzugeben.

10.4.5 Komplettierung der Ansichtsklasse

10.4.5.1 Membervariablen und Initialisierung

In der `CModeView`-Klassendefinition wird eine Elementvariable vom Typ COLORREF für die aktuelle Zeichenfarbe eingefügt:

```
protected:
  COLORREF m_clrCurrent;
```

Diese Variable wird im `CModeView`-Konstruktor initialisiert:

```
CModeView::CModeView()
{
  // ZU ERLEDIGEN: Hier Code zur Konstruktion einfügen,
  m_clrCurrent = RGB(255, 0, 0);
}
```

Mit Hilfe des MFC-Makros **RGB()** wird Rot als Startwert eingestellt.

10.4.5.2 Die Methode OnDraw()

In der Basisklasse **CView** unserer Ansichtsklasse sind einige virtuelle Methoden definiert, die vom MFC-Anwendungsgerüst automatisch aufgerufen werden und in der Regel in jeder Anwendung überschrieben werden müssen, damit ein sinnvolles Verhalten zustande kommt. Dazu zählt die Methode **OnDraw()** zum Zeichnen das Fensterinhalts (z.B. nach einer Größenänderung), die auch beim Drucken benutzt wird:

```
/////////////////////////////////////////////////////////////////////////////
// CModeView Zeichnen

void CModeView::OnDraw(CDC* pDC)
{

  CModeDoc* pDoc = GetDocument();
  ASSERT_VALID(pDoc);
  // ZU ERLEDIGEN: Hier Code zum Zeichnen der ursprünglichen Daten hinzufügen

  pDC->SetMapMode(MM_LOMETRIC);

  COLORREF color;
  int x1, x2, y1, y2;
  CRect rect;
  for (int i = 0; i < 16; i++)
    for (int j = 0; j < 16; j++)
    {
      color = pDoc->GetCellColor(i, j);
      CBrush brush(color);
      x1 = (j * 100) + 100;
      y1 = (i * -100) - 100;
      x2 = x1 + 100;
      y2 = y1 - 100;

      rect.SetRect(x1, y1, x2, y2);
      pDC->FillRect(rect, &brush);

    }
}
```

Ein geeigneter Gerätekontext wird von Anwendungsgerüst als Parameter gleich mitgeliefert. Über dessen Methode **SetMapMode()** ersetzen wir die bildschirm-orientierten Koordinaten (eine Einheit

= ein Pixel) durch *metrisch* orientierte Koordinaten in niedriger Auflösung (eine Einheit = 0,1 mm), was z.B. die Druckausgabe erleichtert:

```
pDC->SetMapMode(MM_LOMETRIC);
```

Mit dem **Abbildungsmodus** MM_LOMETRIC ist eine *aufwärts* orientierte Y-Achse verbunden. Weil der Ursprung (0, 0) *oben* links angesiedelt ist, erhalten daher die tieferen Punkte eine negative Y-Koordinate.

10.4.5.3 Nachrichtenbehandlungsmethoden zur Benutzerinteraktion

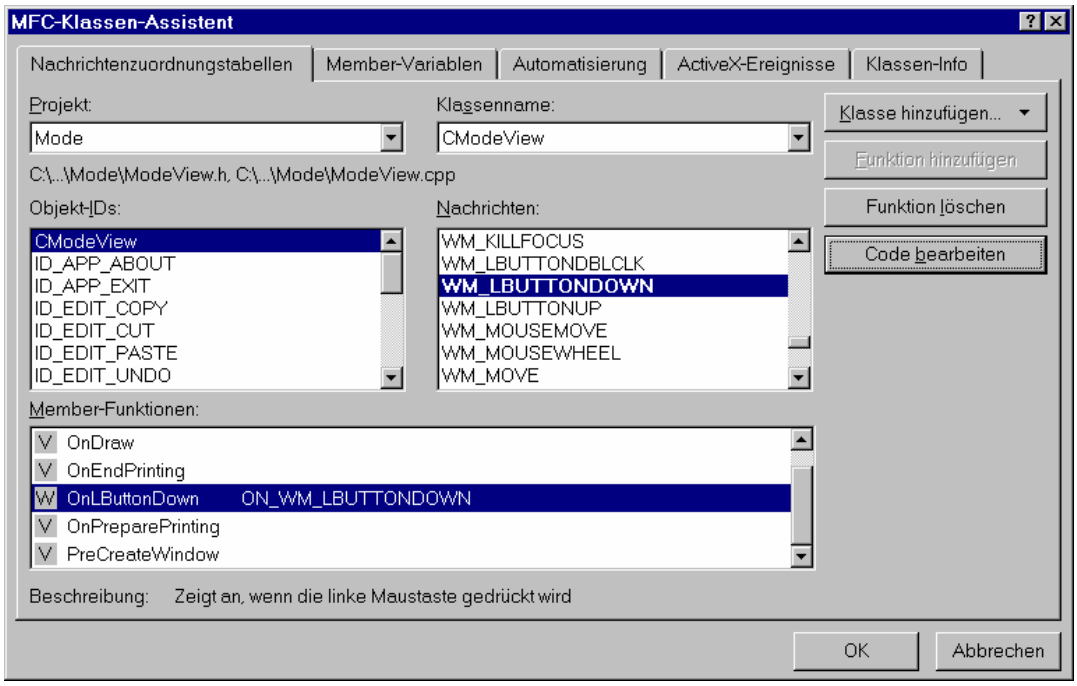
Der `Mode`-Anwender soll über linke Mausklicks auf die vorgegebenen Mosaikfläche die Farbe der einzelnen Zellen bestimmen können. Dazu müssen wir das Ereignis **WM_LBUTTONDOWN** in einer geeigneten Routine behandeln, was sinnvollerweise in der Ansichtsklasse geschehen sollte.
Wir lassen uns vom Klassenassistenten beim Erstellen der Nachrichtenbehandlungsroutine unterstützen. Er wird gestartet mit

Ansicht > Klassenassistent

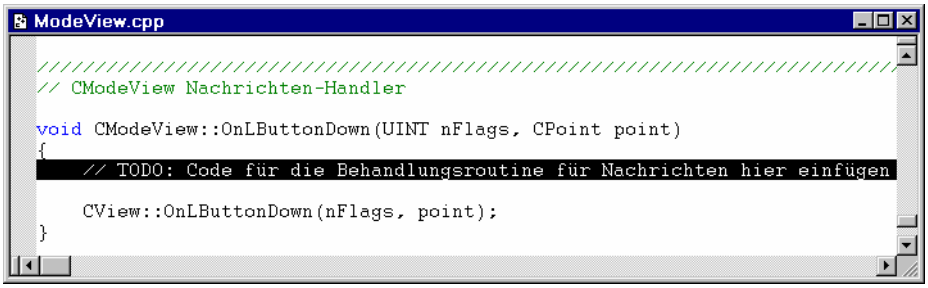
Gehen Sie folgendermaßen vor, um in der Ansichtsklasse einen Handler für die Nachricht **WM_LBUTTONDOWN** (linke Maustaste gedrückt) zu vereinbaren:

- Bleiben Sie bei der Registerkarte **Nachrichtenzuordnungstabellen**.
- Wählen Sie den **Klassennamen** des Ansichtsobjektes.
- Markieren Sie die Ansichtsklasse nötigenfalls auch in der Liste mit den **Objekt-IDs**.
- Markieren Sie die Nachricht **WM_LBUTTONDOWN**.
- Klicken Sie auf den Schalter **Funktion hinzufügen**.

Die neue Nachrichtenbehandlungsroutine erscheint nun in der Liste mit den **Member-Funktionen**:



Nach einem Mausklick auf den Schalter **Code bearbeiten** wechselt das Visual Studio zum Quellcode der Ansichtsklasse und springt zum Rumpf der neuen Funktion:



Die von Windows im Parameter **point** übergebene Klick-Position wird mit der **CClientDC**-Methode **DPtoLP()** in das Koordinatensystem **MM_LOMETRIC** umgewandelt und dann einer Zelle unserer Datenmatrix zugeordnet:

```
//////////////////////////////////////////////////////////////////////////
// CModeView Nachrichten-Handler

void CModeView::OnLButtonDown(UINT nFlags, CPoint point)
{
  // TODO: Code für die Behandlungsroutine für Nachrichten
  // hier einfügen und/oder Standard aufrufen
  CView::OnLButtonDown(nFlags, point);

  int i, j;
  CModeDoc * pDoc = GetDocument();

  CClientDC dc(this);
  dc.SetMapMode(MM_LOMETRIC);
  CPoint lpos = point;
  dc.DPtoLP(&lpos);

  if(lpos.x >= 100  &&  lpos.x <= 1700  &&
     lpos.y <= -100  &&  lpos.y >= -1700)
  {
    i = (-lpos.y - 100) / 100;
    j = (lpos.x - 100) / 100;
    pDoc->SetCellColor(i, j, m_clrCurrent);
  }
}
```

10.4.6 Ressourcen ergänzen

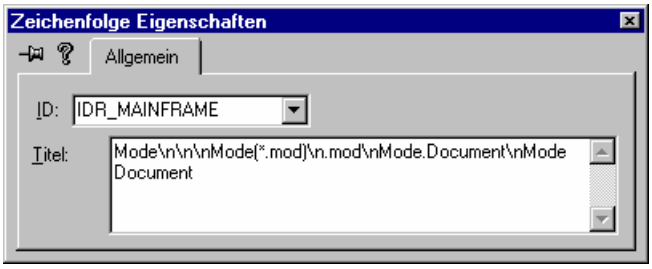
Die Dokumenten-Zeichenfolge

Die String-Ressource **IDR_MAINFRAME** kann bei SDI-Anwendungen bis zu sieben, durch die Escape-Sequenz „\n „getrennte, Teilzeichenfolgen besitzen, die folgende Bedeutungen haben:

1. Beschriftung der Titelzeile im Hauptfenster
2. Bezeichnung für neue Dokumente
   Bei einer *leeren* Teilzeichenfolge (zwei aufeinander folgende NewLine-Zeichen), heißen die neuen Dokumente *Unbenannt*.
3. Bei SDI-Anwendungen irrelevant
4. Bezeichnung des Dokumententyps für die Dialoge zum Öffnen und Speichern
5. Voreingestellte Dateinamenserweiterung
6. Dokumententyp-Bezeichnung für die Registrierungsdatenbank
   Hier sind Leerzeichen verboten.
7. Dokumententyp-Beschreibung für die Registrierungsdatenbank
   Hier sind Leerzeichen erlaubt.

Die beiden letztgenannten Teilzeichenfolgen werden nur relevant, wenn eine Anwendung über **CWinApp::RegisterShellFileTypes()** ihren Dokumententyp in die Registrierungsdatenbank einträgt.

Für unser Beispiel kann folgende Dokumenten-Zeichenfolge verwendet werden:



Wer sich früh festlegen will und kann, hat schon im vierten Schritt des Anwendungsassistenten (siehe oben) Gelegenheit, über **Weitere Optionen** die Bestandteile der Dokumenten-Zeichenfolge anzugeben.

Icon

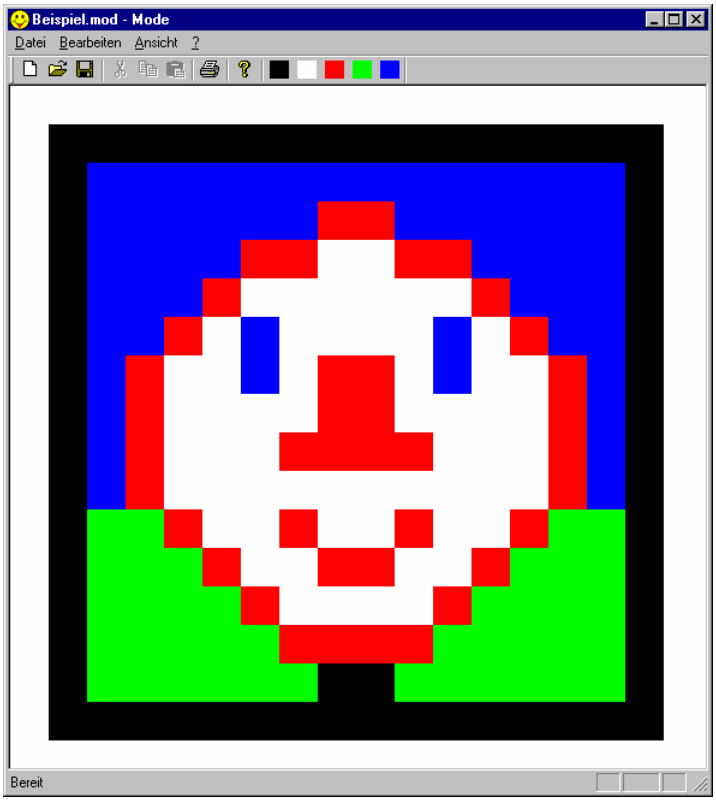
Wir ersetzen das MFC-Standard-Icon durch das schon früher verwendete Smiley-Icon:

- Löschen Sie die vorhandene Icon-Ressource mit dem Namen IDR_MAINFRAME.
- Erstellen Sie eine neue Icon-Ressource über:

  Einfügen > Ressource > Icon > Importieren

  und vergeben Sie den Namen IDR_MAINFRAME.

Nachdem Sie in der Übungsaufgabe zu Abschnitt 10.4 noch einige Symbole zum bequemen Ändern der Zeichenfarbe ergänzt haben, kann man unseren Mosaik-Designer schon zu Kreativitätsübungen einsetzen:



Das Programm kann u.a. ...

- Dokumente speichern und laden,
- Dokumente drucken, wobei auch eine Druckbildvorschau nicht fehlt,
- eine Liste der zuletzt bearbeiteten Dokumente verwalten (siehe **Datei**-Menü)

Zweifellos gibt es noch einiges zu verbessern. So zeigt unser Mosaik-Designer bei jedem Mausklick zum Einfärben einer Matrixzelle ein unansehnliches Flickern, weil die Ansicht komplett neu gezeichnet wird (von **OnDraw()**). Um diesen Effekt zu verhindern, sollte nur die tatsächlich betroffenen Zelle neu eingefärbt werden.

Zur Windows-Programmierung konnten in diesem Kurs nur Grundlagen vermittelt werden. Sie besitzen aber nun günstige Voraussetzungen, sich zügig zu einem erfolgreichen MFC-Programmierer zu entwickeln, wobei z.B. das Buch von Prosise (1999) eine wertvolle Hilfe sein kann.

10.4.7 Aufgaben zum Abschnitt 10.4

1) Erweitern Sie die vorhandene Symbolleiste der `Mode`-Anwendung um fünf Schalter, die ein Ändern der Zeichenfarbe auf Rot, Grün, Blau, Schwarz oder Weiß ermöglichen.
Verwenden Sie dann den Klassenassistenten, um die im Ressourceneditor festgelegten Befehls-Identifikationen mit (neu zu erstellenden) Nachrichtenbehandlungsroutinen des Ansichtsobjekts zu verknüpfen.
Hinweise:

- Die zu bearbeitende Nachricht wird im Klassenassistenten mit **COMMAND** gekennzeichnet.
- Die aktuelle Zeichenfarbe ist in der Membervariable `m_clrCurrent` unseres Ansichtsobjektes gespeichert.

11 Literatur

Eckel, B. (2000). *Thinking in C++*. (2nd ed.). Online-Dokument: http://www.bruceeckel.com/ThinkingInCPP2e.html

Erlenkötter, H. & Reher, V. (1997). *C++ für Windows 95/NT*. Reinbek: rororo.

Hyman, M. & Arnson, B. (1999). *Visual C++6 für Dummies* (2. Aufl.). Bonn: MITP-Verlag.

Kernighan, B. W. & Ritchie, D. M. (1988). *The C Programming Language* (2nd ed.). New Jersey: Prentice Hall Inc.

Kruglinski, D. (1997). *Inside Visual C++*. Redmond, Washington: Microsoft Press.

Lee, P. & Phillips, C. (1997). *Der C++-Kurs*. Bonn: ITP-Verlag.

Lippman, S. B. (2000). *Essential C++*. Reading, Massachusetts: Addison Wesley.

Petzold, C. (1996). *Programming Windows 95*. Redmond, Washington: Microsoft Press.

Prosise, J. (1999). *Programming Windows with MFC* (2nd ed.). Redmond, Washington: Microsoft Press.

RRZN (1998a). *Die Programmiersprache C* (10. Aufl). Hannover

RRZN (1998b). *C++ für C-Programmierer* (10. Aufl). Hannover

RRZN (1999). *Grundlagen der Programmierung mit Beispielen in C++ und Java*. Hannover.

Schiedermeier, R. (1996). *Vorlesung Programmieren I*. Online-Dokument: http://www.informatik.fh-muenchen.de/~schieder/programmieren-1-ws96-97/

Stroustrup, B. (1992). *Die C++ Programmiersprache* (2. Aufl.). Bonn: Addison-Wesley.

Wigard, S. (1999). *Visual C++ 6*. Kaarst: bhv.

Zimmermann, W. (2002). Rechnertechnik 2. Online-Dokument: http://www.it.fht-esslingen.de/~zimmerma/vorlesungen/re2/

12 Anhang

12.1 Operatorentabelle

In der folgenden Tabelle sind alle im Kurs behandelten Operatoren in absteigender Priorität (von oben nach unten) mit Auswertungsrichtung aufgelistet. Gruppen von Operatoren mit gleicher Priorität sind durch fette horizontale Linien begrenzt. Sie verfügen dann stets auch über dieselbe Auswertungsrichtung.

	Operator	**Bedeutung**	**Richtung**
↓ absteigende Priorität ↓	::	Scope-Operator (Namensraumauflösungsoper.)	L ⟶ R
	. -> [] () typeid()	Member-Selektion per Objekt (Punktoperator) Member-Selektion per Zeiger (Zeigeroperator) Array-Zugriff (Offset-Operator) Funktions- bzw. Methodenaufruf Typinformation	L ⟶ R
	! ~ ++, -- * & new delete - (*type*) sizeof	Negation Bitweise Negation Prä- oder Postinkrement bzw. -dekrement Inhaltsoperator (Dereferenzierung) Adressoperator Speicher reservieren (dynam. Variable anlegen) Speicher freigeben Vorzeichenumkehr Typumwandlung Größe eines Datentyps oder einer Variablen	L ⟵ R
	*, / %	Punktrechnung Modulo	L ⟶ R
	+, -	Strichrechnung	L ⟶ R
	<<, >>	Bitweiser Links- bzw. Rechts-Shift	L ⟶ R
	>, <, >=, <=	Vergleichsoperatoren	L ⟶ R
	==, !=	Gleichheit, Ungleichheit	L ⟶ R
	&	Bitweises UND	L ⟶ R
	^	Exklusives bitweises ODER	L ⟶ R

	Operator	Bedeutung	Richtung
	|	Bitweises ODER	L → R
	&&	Logisches UND	L → R
	||	Logisches ODER	L → R
	? :	Konditionaloperator	L ← R
	=, +=, -=, *=, /=, %=	Wertzuweisung, Aktualisierung	L ← R
	,	Kommaoperator	L → R

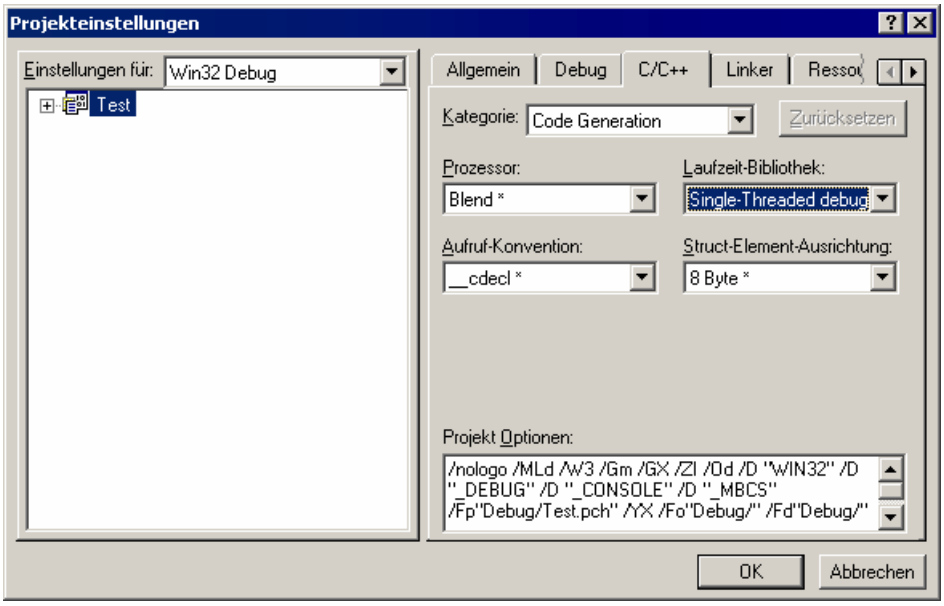
12.2 Standardbibliotheks-Dateien in Visual C++ 6.0

Bibl.typ und zugeh. Compiler-Schalter	C - Standardbibliothek	C++ - Erweiterungen	Alte Iostream - Bibliothek
Single Threaded (ML)	LIBC.LIB	LIBCP.LIB	LIBCI.LIB
Multithreaded (MT)	LIBCMT.LIB	LIBCPMT.LIB	LIBCIMT.LIB
Multithreaded DLL -Version (MD)	MSVCRT.LIB (Import-Bibl. für MSVCRT.DLL)	MSVCPRT.LIB (nutzt MSVCRT.DLL und MSVCP50.DLL)	MSVCIRT.LIB (Import-Bibl. für MSVCIRT.DLL)
Debug Single Threaded (MLd)	LIBCD.LIB	LIBCPD.LIB	LIBCID.LIB
Debug Multi-threaded (MTd)	LIBCMTD.LIB	LIBCPMTD.LIB	LIBCIMTD.LIB
Debug Multi-threaded DLL (MDd)	MSVCRTD.LIB (Import-Bibl. für MSVCRTD.DLL)	MSVCPRTD.LIB (nutzt MSVCRTD.DLL, MSVCP50D.DLL)	MSVCIRTD.LIB (Import-Bibl. für MSVCIRTD.DLL)

Die in der ersten Spalte angegebenen Compiler-Schalter können für ein Projekt nach

Projekt > Einstellungen > C/C++

in der folgenden Dialogbox gesetzt werden:



Nachdem in der versteckten **Kategorien**-Liste die Option **Code Generation** eingestellt ist, kann in der versteckten Liste mit den **Laufzeit-Bibliotheken** die gewünschte Einstellung vorgenommen werden.

12.3 Windows-Namen für Typen und Werte

In der folgenden Tabelle finden Sie eine Auswahl der in Windows-Programmen üblichen Namen für Datentypen und Werte samt C++ - Entsprechung (falls vorhanden):

Windows-Name	C++ - Name	Beschreibung
BYTE	unsigned char	8bit
BOOL	int	
DWORD	unsigned long	32bit
FALSE	false	
HANDLE	void *	Handle für ein Windows-Element
HINSTANCE	bei STRICT-Typüberprüfung: HINSTANCE__ * Sonst: void * (alias HANDLE)	Handle für eine Anwendungsinstanz HINSTANCE__ ist definiert durch: struct HINSTANCE__ {int unused;};
HMENU	bei STRICT-Typüberprüfung: HMENU__ * Sonst: void * (alias HANDLE)	Handle für ein Menü HMENU__ ist definiert durch: struct HMENU __ {int unused;};
HWND	bei STRICT-Typüberprüfung: HWND__ * Sonst: void * (alias HANDLE)	Handle für ein Fenster HWND__ ist definiert durch: struct HWND __ {int unused;};
LONG	long	32bit
LPARAM	long	32bit
LPCSTR	const char *	Konstanter C-String
LPCTSTR		Pointer auf einen konstanten TCHAR-String
LPSTR	char *	C-String
LPTSTR		Pointer auf TCHAR-String

Windows-Name	C++ - Name	Beschreibung
LPVOID	void *	generischer Zeigertyp
LRESULT	long	32bit
NULL	0	
TCHAR		Generischer Datentyp für Zeichen Symbol _UNICODE definiert: wchar_t Sonst: char
TRUE	true	
UINT	unsigned int	32bit
VOID	void	
WINAPI, CALLBACK	__stdcall	Aufrufkonvention im Windows-API
WORD	unsigned short	16bit
WPARAM	unsigned int	32bit

13 Stichwortregister

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Z