Freitag

Samstag

Sonntag

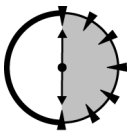
Freitagabend

Teil 1

Was ist Programmierung?

Checkliste

- ☑ Die Prinzipien des Programmierens erlernen
- ☑ Lernen, ein menschlicher Prozessor zu sein
- ☑ Lernen, einen Reifen zu wechseln



30 Min.

Der Webster's New World College Dictionary bietet mehrere Definitionen des Substantivs »Programm« an. Die erste lautet »eine Proklamation, ein Prospekt, oder eine Inhaltsangabe«. Das trifft das, worum es uns geht, nicht so richtig. Erst die sechste Definition passt besser: »eine logische Sequenz codierter Instruktionen, die Operationen beschreiben, die von einem Computer ausgeführt werden sollen, um ein Problem zu lösen oder Daten zu verarbeiten«.

Nach kurzem Nachdenken habe ich festgestellt, dass diese Definition ein wenig restriktiv ist. Zum einen weiß man in der Phrase »eine logische Sequenz codierter Instruktionen ...«, nicht, ob die Instruktionen gekrypted, d.h. kodiert sind oder nicht, und der Begriff »logisch« ist sehr einschränkend. Ich selber habe schon Programme geschrieben, die nicht sehr viel gemacht haben, bevor sie abgestürzt sind. In der Tat stürzen die meisten meiner Programme ab, bevor sie irgend etwas tun. Das scheint nicht sehr logisch zu sein. Zum anderen »... um ein Problem zu lösen oder Daten zu verarbeiten«: Was ist mit dem Steuerungscomputer meiner Klimaanlage im Auto? Er löst kein Problem, das mir bewusst ist. Ich mag die Klimaanlage so wie sie ist – anschalten und ausschalten auf Knopfdruck.

Das größte Problem mit Webster's Definition ist die Phrase »die von einem Computer ausgeführt werden sollen ...«. Ein Programm hat nicht unbedingt etwas mit Computern zu tun. (Es sei denn, Sie zählen das konfuse Etwas zwischen Ihrem Stereokopfhörer dazu. In diesem Fall können Sie behaupten, dass alles, was Sie tun, etwas mit Computern zu tun hat.) Ein Programm kann ein Leitfaden sein, für etwas, das wenigstens ein Körnchen Intelligenz besitzt – sogar für mich. (Vorausgesetzt, ich liege nicht unterhalb dieser Körnchengrenze.) Lassen Sie uns betrachten, wie wir ein Programm schreiben würden, um ein menschliches Verhalten anzuleiten.

1.1 Ein menschliches Programm

Ein Programm für einen Menschen zu schreiben ist viel einfacher, als ein Programm für eine Maschine zu schreiben. Uns sind Menschen vertraut – wir verstehen Menschen gut. Ein besonders wichtiger Aspekt dieser Vertrautheit ist die gemeinsame Sprache. In diesem Abschnitt schreiben wir ein »menschliches Programm« und studieren seine Teile. Lassen Sie uns das Problem des Reifenwechsels betrachten.

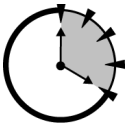
1.1.1 Der Algorithmus

Das Wechseln eines Reifens ist recht einfach. Die Schritte gehen etwa so:

```
1. Heben Sie den Wagen mit dem Wagenheber an.
2. Entfernen Sie die Radmuttern, die den Reifen am Wagen befestigen.
3. Entfernen Sie den platten Reifen.
4. Setzen Sie den Ersatzreifen ein.
5. Schrauben Sie ihn fest.
6. Lassen Sie Ihren Wagen wieder herunter.
```

(Ich weiß, dass Rad und Reifen nicht das Gleiche sind – Sie entfernen nicht den Reifen vom Auto, sondern das Rad. Zwischen diesen beiden Begriffen hin und her zu springen, ist verwirrend. Nehmen Sie daher einfach an, dass das Wort »Reifen« in diesem Beispiel synonym zu »Rad« verwendet wird.)

Das ist das grundlegende Programm. Ich könnte mit diesen Anweisungen alle meine platten Reifen ersetzen, die ich bisher hatte. Um genauer zu sein, ist dies ein *Algorithmus*. Ein Algorithmus ist eine Beschreibung von auszuführenden Schritten, in der Regel auf einem hohen Abstraktionsniveau. Ein Algorithmus ist für ein Programm wie eine Beschreibung der Prinzipien des Fernsehens für die Schaltkreise in einem Fernseher.



20 Min.

1.2 Der Prozessor

Um etwas ans Laufen zu bekommen, muss ein Algorithmus mit etwas kombiniert werden, das ihn ausführt: einem *Prozessor*. Unser Programm zum Wechseln des Reifens setzt z.B. voraus, dass ein Mann (hups, ich meine, eine *Person*) vorhanden ist, um den Wagen anzuheben, die Schrauben zu lösen und den neuen Reifen an seinen Platz zu heben. Die erwähnten Objekte – Auto, Reifen und Schrauben – sind nicht in der Lage, sich alleine zu bewegen.

Lassen Sie uns annehmen, dass unser Prozessor nur einige Wörter versteht, und diese sehr wörtlich. Lassen Sie uns weiter annehmen, dass unser Prozessor die folgenden Substantive versteht, die im Reifenwechselgeschäft geläufig sind:

```
Auto
Reifen
Radmutter
Wagenheber
Schraubenschlüssel
```

(Die beiden letzten Objekte wurden in unserem Algorithmus zum Reifenwechseln nicht erwähnt, aber sie waren in Phrasen wie »ersetzen Sie den Reifen« enthalten. Das ist das Problem mit Algorithmen – so vieles ist nicht explizit ausgesprochen).

Lassen Sie uns weiterhin annehmen, dass unser Prozessor folgende Verben versteht:

```
greifen
bewegen
loslassen
drehen
```

Schließlich muss unsere Prozessorperson zählen und einfache Entscheidungen treffen können.

Das ist alles, was unsere Prozessorperson fürs Reifenwechseln versteht. Alle weiteren Anweisungen rufen bei ihr nur einen ratlosen Blick hervor.

1.3 Das Programm

Mit dem gegebenen Vokabular ist es unserem Prozessor nicht möglich, Anweisungen der Form »entferne die Radmuttern vom Auto« auszuführen. Das Wort »entfernen« kommt nicht im Vokabular des Prozessors vor. Außerdem wird der Schraubenschlüssel, mit dem die Radmuttern gelöst werden müssen, nicht erwähnt. (Das sind die Dinge, die in der normalen Sprache auch unausgesprochen funktionieren.)

Die folgenden Schritte beschreiben den Vorgang »Radmuttern entfernen« unter Verwendung von Begriffen, die der Prozessor versteht:

```
1. Schraubenschlüsseln greifen
2. Schraubenschlüssel auf Radmutter bewegen
3. Schraubenschlüssel fünfmal gegen Uhrzeigersinn drehen
4. Schraubenschlüssel von Radmutter entfernen
5. Schraubenschlüssel loslassen
```

Lassen Sie uns jeden Schritt einzeln ansehen.

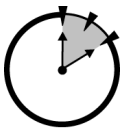
Der Prozessor beginnt mit Schritt 1 und arbeitet jeden Schritt einzeln ab, bis er bei Schritt 5 angekommen ist. In der Ausdrucksweise der Programmierung sagen wir, dass das Programm von Schritt 1 nach Schritt 5 fließt, obwohl das Programm nirgends hingeht – die Prozessorperson tut dies.

In Schritt 1 nimmt die Prozessorperson den Schraubenschlüssel. Es kann sein, dass der Prozessor den Schraubenschlüssel bereits in der Hand hat, aber wir können nicht davon ausgehen. In Schritt 2 wird der Schraubenschlüssel auf der Radmutter platziert. Schritt 3 löst die Radmutter. Die Schritte 4 und 5 schließlich geben den Schraubenschlüssel zurück.

Ein Problem mit diesem Algorithmus ist ganz offensichtlich. Wie können wir wissen, dass fünf Umdrehungen ausreichen, um die Radmutter zu lösen? Wir könnten die Anzahl der Umdrehungen einfach so groß machen, dass sie ausreicht, um jede beliebige Radmutter zu lösen. Diese Lösung wäre nicht nur Verschwendung, sondern kann auch mal nicht funktionieren. Was wird unser Prozessor tun, wenn die Radmutter herunterfällt und der Prozessor die Anweisung erhält, den Schraubenschlüssel erneut zu drehen? Wird das den Prozessor verwirren und zum Anhalten bewegen?

Das folgende Programm nutzt die eingeschränkten Fähigkeiten des Prozessors, einfache Entscheidungen zu treffen, um eine Radmutter korrekt zu entfernen:

```
1. Schraubenschlüssel greifen
2. Schraubenschlüssel auf Radmutter bewegen
3. Solange Radmutter nicht gelöst
4. <
5.   Schraubenschlüssel gegen Uhrzeigersinn drehen
6. >
7. Schraubenschlüssel von Radmutter entfernen
8. Schraubenschlüssel loslassen
```



10 Min.

Das Programm durchläuft die Schritte 1 und 2 wie zuvor. Schritt 3 ist vollständig anders. Der Prozessor wird angewiesen, die Schritte, die in den auf Schritt 3 folgenden Klammern eingeschlossen sind, so lange auszuführen, bis eine bestimmte Bedingung erfüllt ist. In diesem Fall, bis die Radmutter gelöst ist. Sobald die Radmutter gelöst ist, fährt die Prozessorperson bei Schritt 7 fort. Die Schritte 3 bis 6 werden als *Schleife* bezeichnet, da der Prozessor sie wie einen Kreis durchläuft.

Diese Lösung ist viel besser, da sie keine Annahmen in Bezug auf die Anzahl der Umdrehungen macht, die benötigt werden, um eine Radmutter zu lösen. Das Programm ist außerdem nicht verschwenderisch, da keine unnötigen Umdrehungen ausgeführt werden. Und das Programm weist den Prozessor nie an, eine Schraube zu drehen, die nicht mehr da ist.

So schön das Programm ist, hat es doch noch ein Problem: Es entfernt nur eine einzige Radmutter. Die meisten Autos haben fünf Radmuttern pro Reifen. Wir können die Schritte 2 bis 7 fünfmal wiederholen, einmal für jede Radmutter. Immer fünf Radmuttern zu entfernen, funktioniert aber auch nicht. Kleinwagen haben manchmal nur vier Radmuttern, größere Autos und kleine Lastwagen haben meist sechs Radmuttern.

Das folgende Programm erweitert die letzte Lösung auf alle Radmuttern eines Reifens, unabhängig von der Anzahl der Radmuttern.

```
 1. Schraubenschlüssel greifen
 2. Für jede Radmutter
 3. <
 4.   Schraubenschlüssel auf Radmutter bewegen
 5.   Solange Radmutter nicht gelöst
 6.   <
 7.     Schraubenschlüssel gegen Uhrzeigersinn drehen
 8.   >
 9.   Schraubenschlüssel von Radmutter entfernen
10. >
11. Schraubenschlüssel loslassen
```

Das Programm fängt wie immer mit dem Greifen des Schraubenschlüssels an. Danach durchläuft das Programm eine Schleife zwischen den Schritten 2 bis 10 über alle Radmuttern. Schritt 9 entfernt den Schraubenschlüssel von der Radmutter, bevor in Schritt 2 mit der nächsten Radmutter fortgefahren wird.

Beachten Sie, wie die Schritte 5 bis 8 wiederholt werden, bis die Radmutter gelöst ist. Die Schritte 5 bis 8 werden als *innere Schleife* bezeichnet, während die Schritte 2 bis 10 als *äußere Schleife* bezeichnet werden.

Das gesamte Programm besteht aus einer Kombination gleichartiger Lösungen für jeden der sechs Schritte im ursprünglichen Programm.

1.4 Computerprozessoren

Ein Computerprozessor arbeitet sehr ähnlich wie ein menschlicher Prozessor. Ein Computerprozessor folgt wörtlich einer Kette von Kommandos, die mit einem endlichen Vokabular erzeugt wurde.

Einen Reifen von einem Auto zu entfernen, scheint eine einfache Aufgabe zu sein, und unsere Prozessorperson benötigt 11 Anweisungen, um ein einzelnen Reifen zu wechseln. Wie viele Anweisungen werden benötigt, um die vielen tausend Pixel auf dem Bildschirm zu bewegen, wenn der Benutzer die Maus bewegt?

0 Min.

Im Gegensatz zu einem menschlichen Prozessor sind Prozessoren aus Silizium extrem schnell. Ein Pentium III-Prozessor kann einige 100 Millionen Schritte pro Sekunde ausführen. Es bedarf einiger Millionen Anweisungen, um ein Fenster zu bewegen, aber weil der Computerprozessor so schnell ist, bewegt sich das Fenster flüssig über den Bildschirm.

Zusammenfassung

Dieses Kapitel hat grundlegende Prinzipien der Programmierung eingeführt anhand eines Beispielprogramms, um einen sehr dummen, aber außerordentlich folgsamen Mechaniker in die Kunst des Reifenwechselns einzuweisen.

- Computer tun das, was Sie ihnen sagen – nicht weniger, aber natürlich auch nicht mehr.
- Computerprozessoren haben ein kleines, aber wohldefiniertes Vokabular.
- Computerprozessoren sind clever genug, einfache Entscheidungen zu treffen.

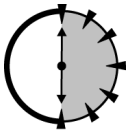
Selbsttest

1. Nennen Sie Substantive, die ein »menschlicher Prozessor« verstehen müsste, um Geschirr abzuwaschen.
2. Nenne Sie einige Verben.
3. Welche Art von Entscheidungen müsste ein Prozessor treffen können?

Ihr erstes Programm in Visual C++

Checkliste

- ☑ Ihr erstes C++-Programm in Visual C++ schreiben
- ☑ Aus Ihrem C++-Code ein ausführbares Programm erzeugen
- ☑ Ihr Programm ausführen
- ☑ Hilfe bei der Programmierung bekommen



30 Min.

Kapitel 1 handelte von Programmen für Menschen. Dieses und das nächste Kapitel beschreiben, wie Sie einen Computer in C++ programmieren. Dieses Kapitel behandelt die Programmierung mit Visual C++, während sich das nächste Kapitel mit dem frei verfügbaren GNU C++ befasst, das auch auf der beiliegenden CD-ROM zu finden ist.

Haben Sie keine Angst vor den Bezeichnungen Visual C++ und GNU C++. Beide Compiler stellen Implementierungen des C++-Standards dar. Jeder der Compiler kann jedes Programm in diesem Buch übersetzen.

Das Programm, das wir schreiben wollen, konvertiert eine Temperatur, die der Benutzer in Grad Celsius eingibt, in Grad Fahrenheit.

2.1 Installation von Visual C++

Sie müssen Visual C++ auf Ihrem Rechner installieren, bevor Sie ein Visual C++-Programm schreiben können. Das Paket Visual C++ wird benutzt, um C++-Programme zu schreiben, und daraus .EXE-Programme zu erzeugen, die der Computer versteht.

Visual C++ ist nicht auf der beiliegenden CD-ROM enthalten. Sie müssen Visual C++ separat erwerben, entweder als Bestandteil von Visual Studio, oder als Einzelprodukt. Den sehr guten GNU C++-Compiler finden Sie auf der CD-ROM.

2.2 Ihr erstes Programm

Ein C++-Programm beginnt sein Leben als Textdatei, die C++-Anweisungen enthält. Ich werde Sie Schritt für Schritt durch das erste Programm führen.

Starten Sie Visual C++. Für Visual Studio 6.0, klicken Sie auf »Start«, gefolgt von den Menüoptionen »Programme« und »Microsoft Visual Studio 6.0«. Von dort wählen Sie »Microsoft Visual C++ 6.0« aus.

Visual C++ sollte zwei leere Fenster zeigen, die mit Ausgabe und Arbeitsbereich bezeichnet sind. Wenn noch andere Fenster gezeigt werden, oder die beiden genannten Fenster nicht leer sind, dann hat jemand bereits Visual C++ auf Ihrem Computer benutzt. Um all dies zu schließen, wählen Sie »Datei« gefolgt von »Arbeitsbereich schließen« aus.

Erzeugen Sie eine leere Textdatei durch Klicken auf das kleine Icon ganz links in der Leiste, wie in Abbildung 2.1 zu sehen ist.

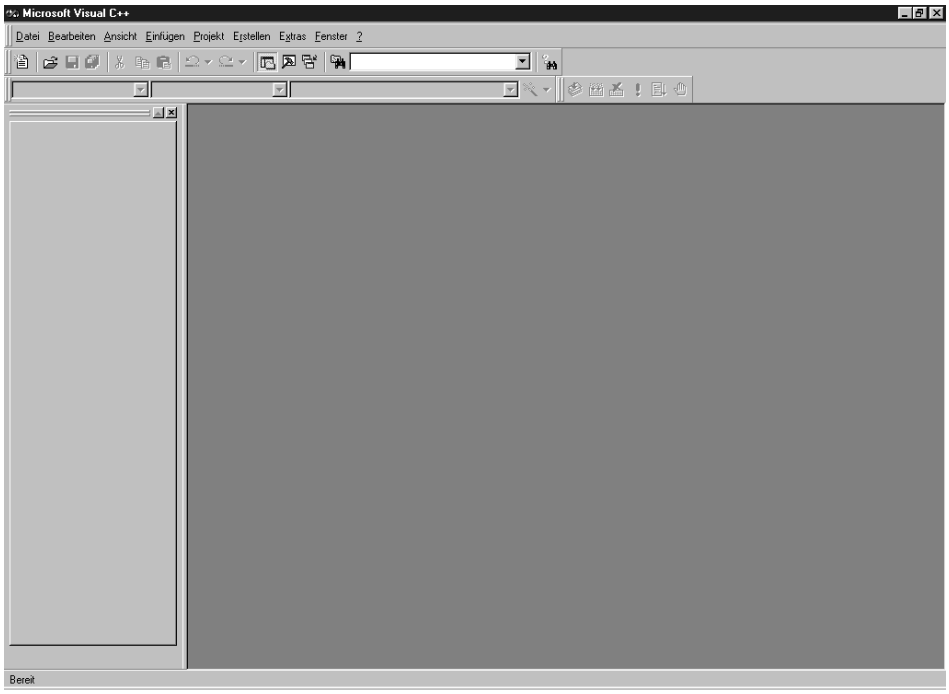


Abbildung 2.1: Sie beginnen damit, ein C++-Programm zu schreiben, indem Sie eine neue Textdatei anlegen.

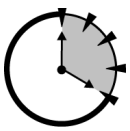
Machen Sie sich keine Gedanken über das Einrücken – es kommt nicht darauf an, ob eine Zeile zwei oder drei Zeichen eingerückt ist. Groß- und Kleinschreibung sind jedoch wichtig. Für C++ sind »Betrügen« und »betrügen« nicht gleich.

Sie können sich das Programm Conversion.cpp von der beiliegenden CD-ROM herunterziehen.

Geben Sie das folgende Programm genau wie hier abgedruckt ein. (Oder kopieren Sie es sich von der CD-ROM.)

```
//
// Programm konvertiert Temperaturen von Grad Celsius
// nach Grad Fahrenheit
// Fahrenheit = Celsius  * (212 - 32)/100 + 32
//
#include <stdio.h>
#include <iostream.h>
int main(int nNumberofArgs, char* pszArgs[])
<
    // Eingabe der Temperatur in Grad Celsius
    int nCelsius;
    cout << »Temperatur in Grad Celsius:«;
    cin >> nCelsius;

    // berechne Umrechnungsfaktor von Celsius
    // nach Fahrenheit
    int nFactor;
    nFactor = 212 - 32;

    // verwende Umrechnungsfaktor zur Konvertierung
    // von Celsius in Fahrenheit
    int nFahrenheit;
    nFahrenheit = nFactor * nCelsius/100 + 32;

    // Ausgabe des Ergebnisses
    cout << »Fahrenheit Wert ist:«;
    cout << nFahrenheit;

    return 0;
>
```

Speichern Sie die Datei unter dem Namen Conversion.cpp. Das Standardverzeichnis ist eines der Verzeichnisse von Visual Studio. Ich bevorzuge es, in ein von mir selbst generiertes Verzeichnis zu wechseln, bevor ich die Datei speichere.



20 Min.

2.3 Erzeugen Ihres Programms

Wir haben in Sitzung 1 eine begrenzte Anzahl von Anweisungen verwendet, um den menschlichen Computer anzuweisen, einen Reifen zu wechseln. Obwohl sehr eingeschränkt, werden Sie vom Durchschnittsmenschen verstanden (zumindest von Deutsch Sprechenden).

Das Programm Conversion.cpp, das Sie gerade eingegeben haben, enthält C++-Anweisungen, eine Sprache, die Sie in keiner Tageszeitung finden werden. So kryptisch und grob diese C++-Anweisungen auch auf Sie wirken, versteht der Computer eine Sprache, die noch viel elementarer ist als C++. Die Sprache, die Ihr Computer versteht, wird als *Maschinensprache* bezeichnet.

Der C++-Compiler übersetzt Ihr C++-Programm in die Maschinensprache Ihrer Mikroprozessor-CPU in Ihrem PC. Programme, die Sie von der Option »Programme« des Menüs »Start« aus aufrufen können, Visual C++ eingeschlossen, sind nichts anderes als Dateien, die Maschinenanweisungen enthalten.

Es ist möglich, Programme direkt in Maschinensprache zu schreiben. Dies ist aber viel schwieriger, als das gleiche Programm in C++ zu schreiben.

Die wichtigste Aufgabe von Visual C++ ist, Ihr C++-Programm in eine ausführbare Datei zu übersetzen. Der Vorgang der Übersetzung in eine ausführbare .EXE-Datei wird als *Erzeugen* bezeichnet. Dieser Prozess wird manchmal auch als *Kompilieren* bezeichnet (es gibt einen Unterschied dieser beiden Begriffe, der aber hier nicht relevant ist). Der Teil des C++-Paketes, der die Übersetzung des Programms ausführt, wird als *Compiler* bezeichnet.

Um Ihr Programm Conversion.cpp zu erzeugen, klicken Sie auf »Erstellen« im Menü »Erstellen«. (Nein, ich habe nicht gestottert.) Visual C++ antwortet darauf mit der Warnung, dass Sie noch keinen Arbeitsbereich angelegt haben, was immer das ist. Dies ist in Abbildung 2.2 zu sehen.

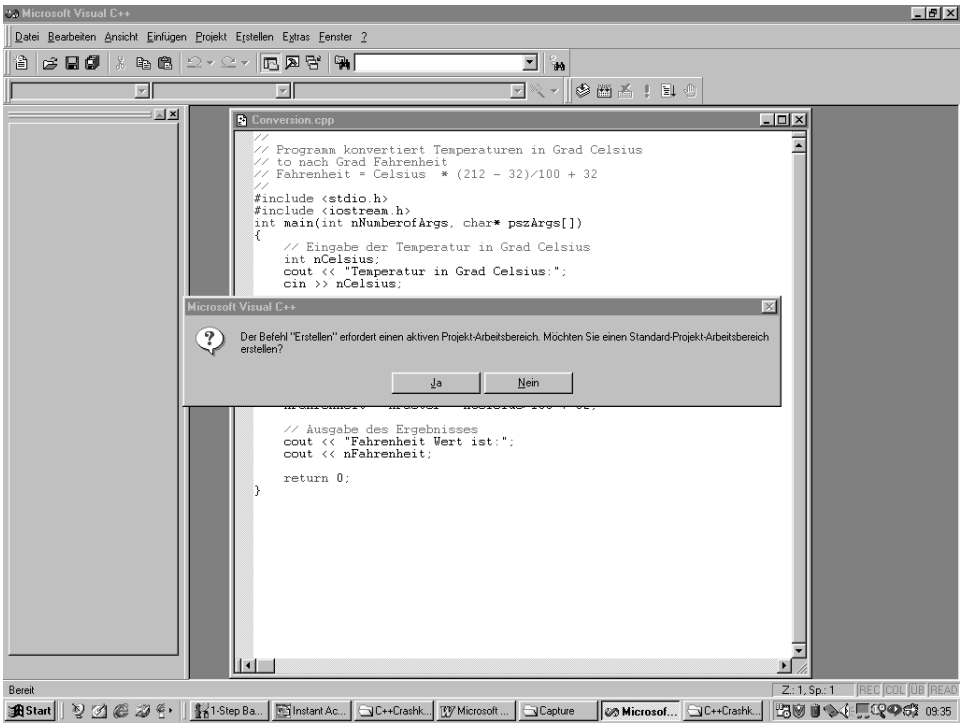


Abbildung 2.2: Ein Arbeitsbereich wird benötigt, bevor Visual C++ Ihr Programm erzeugen kann.

Klicken Sie auf »Ja«, um eine Arbeitsbereichsdatei zu erzeugen und mit dem Erzeugungsprozess fortzufahren.

Die .cpp-Quelldatei ist nichts anderes, als eine Textdatei, ähnlich zu dem, was Sie etwa mit WordPad erzeugen würden. Der Arbeitsbereich Conversion.pwd, der von Visual C++ angelegt wird, ist eine Datei, in der Visual C++ spezielle Informationen über Ihr Programm speichern kann, Informationen, die in die Datei Conversion.cpp nicht hinein gehören.

Nach einigen Minuten Festplattenaktivität antwortet Visual C++ mit einem zufriedenen Klingelzeichen, das anzeigt, dass der Erzeugungsprozess abgeschlossen ist. Das Ausgabefenster sollte eine Meldung ähnlich zu Abbildung 2.3 enthalten, die anzeigt, dass die Datei Conversion.exe ohne Fehler und ohne Warnungen erzeugt wurde.

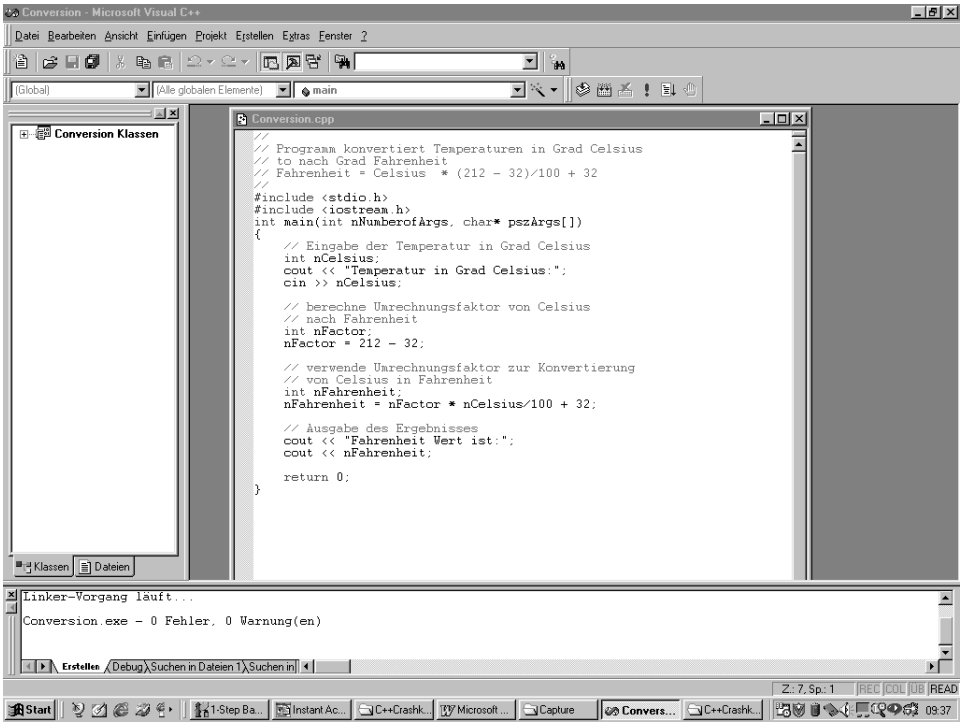


Abbildung 2.3: Keine Fehler und keine Warnungen – das Programm wurde erfolgreich erzeugt.

Visual C++ erzeugt einen unangenehmen Ton, wenn es während des Erzeugungsprozesses auf einen Fehler stößt (wenigstens denkt Microsoft, dass er unangenehm ist – ich habe ihn schon so oft gehört, dass er fast ein Teil von mir geworden ist). Zusätzlich enthält das Ausgabefenster eine Erklärung, welchen Fehler Visual C++ gefunden hat.

Ich habe ein Semikolon am Ende einer Zeile im Programm entfernt und habe das Programm neu kompiliert, nur um die Fehlermeldung zu demonstrieren. Das Ergebnis finden Sie in Abbildung 2.4.

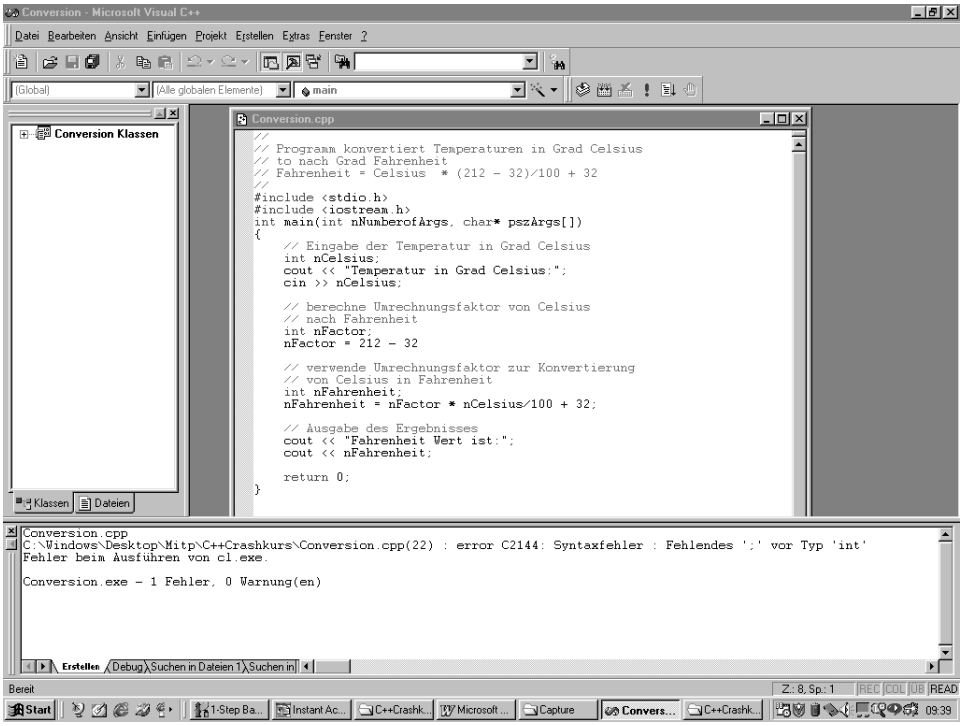


Abbildung 2.4:Visual C++ gibt während des Erzeugungsprozesses eine Fehlermeldung aus.

Die Fehlermeldung in Abbildung 2.4 ist tatsächlich sehr ausführlich. Sie beschreibt das Problem und den Ort des Fehlers (Zeile 22 in der Datei Conversion.cpp). Ich habe das Semikolon wieder eingefügt und das Programm neu kompiliert, um das Problem zu beheben.

Nicht alle Fehlermeldungen sind so klar wie diese. Oft kann ein einziger Fehler mehrere Fehlermeldungen erzeugen. Am Anfang können diese Fehlermeldungen verwirrend sein. Mit der Zeit bekommen Sie ein Gefühl dafür, was Visual C++ während des Erzeugungsprozesses denkt, und was Visual C++ verwirrt haben könnte.

Sie werden ohne Zweifel den unangenehmen Ton eines Fehlers hören, der von Visual C++ entdeckt wurden, bevor Sie das Programm Conversion.cpp fehlerfrei eingegeben haben. Wenn Sie es gar nicht schaffen, den Code so einzugeben, dass Visual C++ damit zufrieden ist, kopieren Sie die Datei Conversion.cpp aus wecc\ Programs\ lesson02\ Conversion.cpp auf der beiliegenden CD-ROM.

C++-Fehlermeldungen

Warum sind alle C++-Pakete, Visual C++ eingeschlossen, so pingelig, wenn es um die Syntax von C++ geht? Wenn Visual C++ erkennt, dass ich ein Semikolon vergessen habe, warum kann es dieses Problem nicht einfach selber lösen und fortfahren?

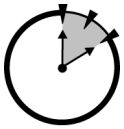
Die Antwort ist einfach aber profund. Visual C++ denkt, dass Sie ein Semikolon vergessen haben. Ich könnte beliebig viele andere Fehler eingebaut haben, die Visual C++ als Fehlen eines Semikolons fehldiagnostiziert haben könnte. Wenn der Compiler einfach das Problem durch Einfügen eines Semikolons behebt, würde Visual C++ möglicherweise dadurch das eigentliche Problem verschleiern.

Wie Sie sehen werden, ist das Auffinden eines Fehlers in einem Programm, das ohne Probleme den Erzeugungsprozess durchläuft, schwierig und zeitaufwendig. Es ist besser, den Compiler Fehler finden zu lassen, wenn möglich.

Diese Lektion war hart zu Beginn. In den frühen Tagen des Computers versuchten Compiler alle möglichen Fehler zu erkennen und selber zu korrigieren. Dies hatte manchmal lächerliche Züge.

Meine Freunde und ich machten uns einen Spaß daraus, einen »freundlichen« Compiler damit zu quälen, indem wir ein Programm eingaben, das nichts als die existenzielle Frage IF enthielt. (Rückschauend waren meine Freunde und ich ein wenig verrückt). Durch eine Reihe schmerzhafter Drehungen hat der besagte Compiler aus diesem einen Wort eine Kommandozeile generiert, die sich ohne Fehler übersetzen ließ. Ich weiß, dass der Compiler meine Absicht mit dem Wort IF missverstanden haben muss, weil ich nichts damit beabsichtigt hatte.

Meine Erfahrung ist, dass jedes Mal, wenn der Compiler versucht hat, ein Problem in einem Programm zu beheben, das Ergebnis falsch war. Trotz Fehlinformation war es keine Schwierigkeit, das Problem zu beheben, wenn der Compiler den Fehler gemeldet hat, bevor er ihn versuchte zu beheben. Compiler, die Fehler behoben haben, ohne entsprechende Fehlermeldungen auszugeben, haben mehr Schaden angerichtet als dass sie geholfen haben.



10 Min.

2.4 Ausführen Ihres Programms

Sie können das erfolgreich erzeugte Programm Conversion.exe durch Klicken auf das Icon »Ausführen« von Conversion.exe unter dem Menü »Erzeugen« ausführen. Alternativ können Sie Ctrl-F5 drücken.

Vermeiden Sie das Ausführen-Menü-Kommando oder die äquivalente F5-Taste fürs Erste.

Visual C++ öffnet ein Programmfenster ähnlich zu dem in Abbildung 2.5, das die Eingabe einer Temperatur in Grad Celsius erwartet.

Geben Sie eine Temperatur ein, z.B. 100 Grad Celsius. Nach Drücken der Entertaste gibt das Programm die äquivalente Temperatur in Grad Fahrenheit aus, wie in Abbildung 2.6 zu sehen ist. Die »Press any key to continue«-Meldung, die vielleicht hinter die Temperaturausgabe gequetscht ist, ist ästhetisch nicht zufriedenstellend, aber die konvertierte Temperatur ist unmissverständlich – wir beheben das in Kapitel 5.

Abbildung 2.5: Das Programm Conversion.exe beginnt mit der Frage nach einer Temperatur.

Abbildung 2.6: Das Programm Conversion.exe wartet auf eine Eingabe, nachdem das Programm beendet ist.

Die »Press any key to continue«-Meldung gibt dem Benutzer Zeit, die Ausgabe des Programms anzusehen, bevor das Fenster nach Beendigung des Programms geschlossen wird. Diese Meldung erscheint nicht, wenn Sie das Go-Kommando oder die F5-Taste verwenden.

Glückwunsch! Sie haben Ihr erstes Programm eingegeben, erzeugt und ausgeführt.

2.5 Abschluss

Es gibt noch zwei Punkte, die erwähnt werden sollten, bevor wir weitergehen. Zum einen könnte Sie die Ausgabe des Programms Conversion.exe überraschen. Zum anderen bietet Visual C++ viel mehr Hilfe an als nur Fehlermeldungen.

2.5.1 Programmausgabe

Windows-Programme haben eine visuell ausgerichtete, Fenster-basierte Ausgabe. Conversion.exe ist ein 32-Bit-Programm, das unter Windows ausgeführt wird, ist aber kein Windows-Programm im visuellen Sinne.

Wenn Sie nicht wissen, was die Phrase »32-Bit-Programm« bedeutet, brauchen Sie sich keine Sorgen zu machen.

Wie ich bereits in der Einleitung erläutert habe, ist dies kein Buch über das Schreiben von Windows-Programmen. Die C++-Programme, die Sie in diesem Buch schreiben, haben ein Kommandozeilen-Interface, das innerhalb einer DOS-Box ausgeführt wird. Angehende Windows-Programmierer sollten nicht verzweifeln – Sie haben Ihr Geld nicht umsonst ausgegeben. Das Erlernen von C++ ist Grundvoraussetzung für das Schreiben von Windows-Programmen mit C++.

0 Min.

2.5.2 Visual C++-Hilfe

Visual C++ bietet ein Hilfesystem an, das den C++-Programmierer signifikant unterstützt. Um zu sehen, wie diese Hilfe funktioniert, führen Sie einen Doppelklick auf dem Wort `#include` aus, bis es vollständig selektiert ist. Jetzt drücken Sie F1.

Visual C++ antwortet darauf mit dem Öffnen der MSDN-Bibliothek und der Anzeige einer ganzen Seite von Informationen über `#include`, wie in Abbildung 1.7 zu sehen ist (Sie verstehen vielleicht nicht alles, was da steht, sie werden es aber später verstehen).

Sie können die gleiche Information über die Auswahl von Index... unter dem Hilfe-Menü finden. Geben Sie `#include` im Indexfenster ein, was die gleichen Informationen liefert.

Wenn Sie als C++-Programmierer erfahrener werden, werden Sie sich mehr und mehr auf das Hilfesystem der MSDN-Bibliothek stützen.

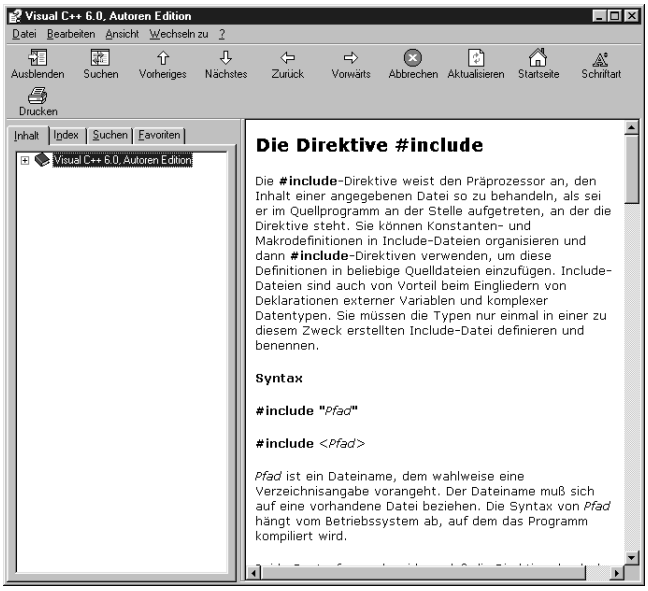


Abbildung 2.7: Die Taste F1 stellt dem C++-Programmierer Hilfe zur Verfügung.

Zusammenfassung

Visual C++ 6.0 hat eine benutzerfreundliche Umgebung, in der Sie Programme erzeugen und testen können. Sie verwenden den Editor von Visual C++, um Ihren Quellcode einzugeben. Einmal eingegeben, werden die C++-Anweisungen durch den Erzeugungsprozess in eine ausführbare Datei überführt. Schließlich können Sie Ihr fertiges Programm von Visual C++ aus ausführen.

Im nächsten Kapitel sehen Sie, wie Sie das gleiche Programm mit dem GNU C++-Compiler erzeugen können, den Sie auf der beiliegenden CD-ROM finden. Wenn Sie absolut überzeugt sind von Visual C++, können Sie zu Sitzung 4 springen, die erklärt, wie das Programm funktioniert, das Sie gerade eingegeben haben.

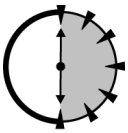
Selbsttest

1. Was für eine Datei ist ein C++-Quellprogramm? (Ist es eine Word-Datei? Eine Excel-Datei? Eine Textdatei?) (Sehen Sie sich den ersten Abschnitt von »Ihr erstes Programm« an.)
2. Beachtet C++ die Einrückung? Achtet es auf Groß- und Kleinschreibung? (Siehe »Ihr erstes Programm«)
3. Was bedeutet »Erzeugen Ihres Programms«? (Siehe »Erzeugen Ihres Programms«)
4. Warum erzeugt C++ Fehlermeldungen? Warum versucht C++ nicht einfach, daraus schlau zu werden, was ich eingebe? (Siehe Kasten »C++-Fehlermeldungen«)

Ihr erstes C++-Programm mit GNU C++

Checkliste

- ☑ GNU C++ von der beiliegenden CD-ROM installieren
- ☑ Ihr erstes C++-Programm in GNU C++ schreiben
- ☑ Aus ihrem C++-Code ein ausführbares Programm erzeugen
- ☑ Ihr Programm ausführen
- ☑ Hilfe bei der Programmierung bekommen



30 Min.

Kapitel 2 behandelte das Schreiben, Erzeugen und Ausführen von C++-Programmen mit Visual C++. Viele Leser des *C++ Wochenend Crashkurses* haben keinen Zugang zu Visual C++. Für diese Leser enthält dieses Buch den frei verfügbaren GNU C++-Compiler, der auf der CD-ROM zu finden ist.

GNU wird »guh-new« gesprochen. GNU steht für die Ringdefinition »GNU is Not Unix«. Dieser Witz geht auf die Anfänge von C++ zurück. Nehmen Sie es einfach wie es ist. GNU ist eine Reihe von Werkzeugen der Free Software Foundation. Diese Werkzeuge stehen der Öffentlichkeit zur Verfügung, mit einigen Nutzungseinschränkungen, aber kostenlos.

Dieses Kapitel zeigt Schritt für Schritt, wie das gleiche Programm Conversion.cpp aus Kapitel 2 mit GNU C++ in ein ausführbares Programm verwandelt werden kann. Das Programm, um das es gehen soll, konvertiert eine Temperatureingabe von Grad Celsius nach Grad Fahrenheit.

3.1 Installation von CNU C++

Die diesem Buch beiliegende CD-ROM enthält die zum Zeitpunkt der Drucklegung neueste Version der GNU C++-Umgebung. Die Installationsanweisungen sind in Anhang C zu finden; diese Sitzung enthält Anweisungen zum Herunterladen und Installieren von GNU C++ vom Web aus.

Die GNU Umgebung wird von einer Reihe freiwilliger Programmierer gepflegt. Wenn Sie dies bevorzugen, können Sie die allerneueste Version von GNU C++ vom Web herunterladen.

Die GNU Entwicklungsumgebung ist ein sehr großes Paket. GNU enthält eine Anzahl von Hilfsprogrammen und anderen Programmiersprachen außer C++. GNU C++ selber unterstützt eine Vielzahl von Computerprozessoren und Betriebssystemen. Glücklicherweise müssen Sie nicht alles von GNU herunterladen, wenn Sie nur C++-Programme entwickeln wollen. Die GNU Entwicklungsumgebung ist auf verschiedene ZIP-Dateien aufgeteilt. Das Hilfsprogramm "ZIP-Picker", das auf der Website von Delorie-Software zu finden ist, teilt Ihnen mit, welche ZIP-Dateien Sie herunterladen müssen, basierend auf Ihren Antworten auf eine Reihe einfacherer Fragen.

So installieren Sie GNU C++ vom Web:

1. Gehen Sie zur Webseite `http://www.elorie.com/djgpp/zip-picker.html`.
2. Die Site zeigt Ihnen die Fragen, die wir im Folgenden wiedergeben. Beantworten Sie die Fragen des Programms wie fett gedruckt, um eine minimale Konfiguration zu erhalten.
   FTP Site
   Select a suitable FTP site: **Pick one for me**
   Basic Functionality
   Pick one of the following: **Build and run programs with DJGPP**
   Which operating system will you be using? <**Ihr Betriebssystem**>
   Do you want to be able to read the on-line documentation? **No**
   Which programming languages will you be using? <**Klicken Sie C++**>
   Integrated Development Environments and Tools
   Which IDE(s) would you ike? <**Klicken Sie auf RHIDE. Lassen Sie die emacs-options unausgewählt**>
   Would you like gdb, the text mode GNU debugger? **No**
   Extra Stuff
   Please Check off each extra thing that you want. **Don't check anything in this list.**
3. Dann klicken Sie auf »Tell me what files I need«. Der ZIP-Picker antwortet mit einigen einfachen Installationsanweisungen und einer Liste von ZIP-Dateien, die Sie brauchen werden. Das Listing unten zeigt die Dateien, die für eine minimale Installation, wie sie hier beschrieben wird, benötigt werden – die Dateinamen, die Sie erhalten, können davon verschieden sein, weil sie eine aktuellere Version anzeigen.

```
Read the file README.1ST before you do anything else with DJGPP! It has
important installation and usage instructions.

v2/djdev202.zip           DJGPP Basic Development Kit        1.4 mb
v2/faq211b.zip            Frequently Asked Questions         551 kb

v2apps/rhide14b.zip       RHIDE                              1.6 mb

v2gnu/bnu281b.zip         Basic assembler, linker            1.8 mb
v2gnu/gcc2952b.zip        Basic GCC compiler                 1.7 mb
v2gnu/gpp2952b.zip        C++ compiler                       1.6 mb
v2gnu/lgpp295b.zip        C++ libraries                      484 kb
v2gnu/mak377b.zip         Make (processes makefiles)         242 kb

Total bytes to download: 9,812,732
```

4. Legen Sie ein Verzeichnis \DJGPP an.
5. Laden Sie in dieses Verzeichnis alle ZIP-Dateien, die Ihnen der ZIP-Picker aufgelistet hat, indem Sie auf den jeweiligen Dateinamen klicken.
6. Entpacken Sie die Dateien im Ordner DJGPP.
7. Fügen Sie die folgenden Kommandos in die Datei AUTOEXEC.BAT ein:

```
set PATH=C:\DJGPP\BIN;%PATH%
set DJGPP=C:\DJGPP\DJGPP.ENV
```

   Beachten Sie, dass die obigen Zeilen davon ausgehen, dass ihr Ordner DJGPP direkt unterhalb von C:\ steht. Wenn Sie Ihren Ordner an einer anderen Stelle platziert haben, ersetzen Sie bitte den Pfad in den obigen Kommandos entsprechend.

8. Booten Sie neu, um die Installation abzuschließen.

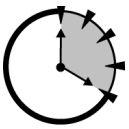
Der Ordner \BIN enthält die ausführbaren Dateien der GNU-Werkzeuge. Die Datei DJGPP.ENV setzt eine Reihe von Optionen, um die GNU C++-»Umgebung« von Windows zu beschreiben.

Bevor Sie GNU C++ benutzen, sehen Sie in der Datei DJGPP.ENV nach, ob der Support langer Dateinamen angeschaltet ist. Diese Option ausgeschaltet zu haben ist der meist begangene Fehler bei der Installation von GNU C++.

Öffnen Sie die Datei DJGPP.ENV mit einem Texteditor, z.B. mit Microsoft WordPad. Erschrecken Sie nicht, wenn Sie nur eine lange Zeichenkette sehen, die von kleinen schwarzen Kästchen unterbrochen ist. Unix verwendet ein anderes Zeichen für Zeilenumbrüche als Windows. Suchen Sie nach der Phrase "LFN=y" oder "LFN=Y" (Groß- oder Kleinschreibung spielen also keine Rolle). Wenn Sie stattdessen "LFN=n" finden (oder "LFN" überhaupt nicht vorkommt), ändern Sie das "n" in "y". Speichern Sie die Datei. (Stellen Sie sicher, dass Sie die Datei als Textdatei speichern und nicht in einem anderen Format, z.B. als Word .DOC-Datei.)

Fügen Sie die folgende Zeile in die Datei DJGPP.ENV im Block [rhide] ein, um rhide in deutscher Sprache zu verwenden:

```
LANGUAGE=DE
```



20 Min.

3.2 Ihr erstes Programm

Das Herz des GNU C++-Paketes ist ein Hilfsprogramm, das als `rhide` bekannt ist. Im Wesentlichen ist `rhide` ein Editor, der die verbleibenden Teile von GNU C++ zusammenbindet zu einer integrierten Umgebung, ähnlich wie Visual C++.

3.2.1 Eingabe des C++-Codes

Öffnen Sie ein MS-DOS-Fenster, indem Sie auf das MS-DOS-Icon unter dem Menü »Programme« klicken. Erzeugen Sie ein Verzeichnis, in dem Sie Ihr Programm erzeugen möchten. Ich habe ein Verzeichnis c:\wecc\programs\lesson03\ erzeugt. In diesem Verzeichnis führen Sie das Kommando `rhide` aus.

Das rhide*-Interface*

Das Interface von rhide ist grundsätzlich anders als das von Windows-Programmen. Windows-Programme »zeichnen« ihre Ausgaben auf den Bildschirm, was ihnen ein feineres Aussehen verschafft.
Im Vergleich dazu arbeitet das Interface von rhide zeichenbasiert. rhide verwendet eine Reihe von Block-Zeichen, die auf dem PC zur Verfügung stehen, um ein Windows-Interface zu simulieren, was rhide weniger elegant aussehen lässt. Z.B. lässt rhide es nicht zu, das Fenster auf eine andere Größe zu bringen als die Standardeinstellung von 80x25 Zeichen, was Standard für MS-DOS-Programme ist.

Für die unter Ihnen, die alt genug sind, um sich daran zu erinnern, sieht das* rhide*-Interface sehr ähnlich aus wie das Interface der Borland-Suite von Programmierwerkzeugen, die es heute nicht mehr gibt.

Wie dem aus sei, das Interface von* rhide *funktioniert und gibt bequemen Zugriff auf die übrigen Werkzeuge von GNU C++.

Erzeugen Sie eine leere Datei, indem Sie »Neu« aus dem Datei-Menü auswählen. Geben Sie das Programm genau so ein, wie Sie es hier vorfinden.

Machen Sie sich keine Gedanken über das Einrücken – es kommt nicht darauf an, ob eine Zeile zwei oder drei Zeichen eingerückt ist oder ob zwei Worte durch ein oder zwei Leerzeichen getrennt sind.

Sie können natürlich mogeln und die Datei Conversion.cpp von der beiliegenden CD-ROM kopieren.

```
//
// Programm konvertiert Temperaturen in Grad Celsius
// nach Grad Fahrenheit
// Fahrenheit = Celsius  * (212 - 32)/100 + 32
//
#include <stdio.h>
#include <iostream.h>
int main(int nNumberofArgs, char* pszArgs[])
{
    // Eingabe der Temperatur in Grad Celsius
    int nCelsius;
    cout << »Temperatur in Grad Celsius:«;
    cin >> nCelsius;

    // berechne Umrechnungsfaktor von Celsius
    // nach Fahrenheit
    int nFactor;
```

```
    nFactor = 212 - 32;

    // verwende Umrechnungsfaktor zur Konvertierung
    // von Celsius in Fahrenheit
    int nFahrenheit;
    nFahrenheit = nFactor * nCelsius/100 + 32;

    // Ausgabe des Ergebnisses
    cout << »Fahrenheit Wert ist:«;
    cout << nFahrenheit;

    return 0;
}
```

Wenn Sie damit fertig sind, sollte Ihr `rhide`-Fenster wie in Abbildung 3.1 aussehen.

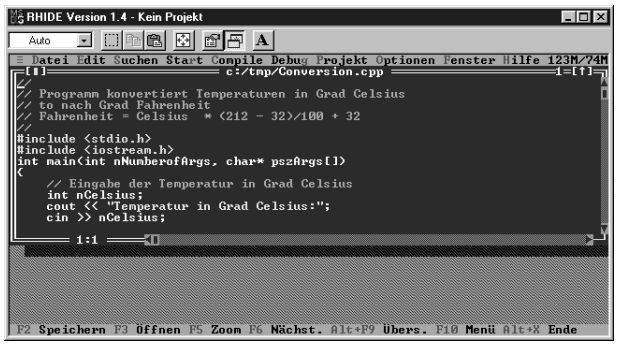


Abbildung 3.1: `rhide` ***stellt ein zeichenbasiertes Interface bereit, um C++-Programme zu erzeugen.***

Wählen Sie »Speichern als...« im Datei-Menü aus, wie in Abbildung 3.2 zu sehen ist, und speichern Sie die Datei unter dem Namen Conversion.cpp.

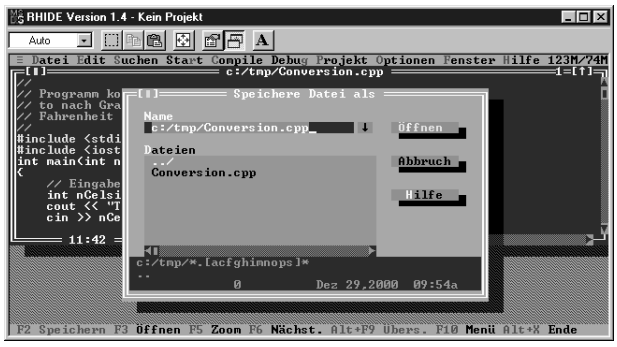


Abbildung 3.2: Mit dem Kommando »Save As...« kann der Benutzer eine C++-Datei erzeugen.

3.3 Erzeugen Ihres Programms

Wir haben in Sitzung 1 eine begrenzte Anzahl von Anweisungen verwendet, um den menschlichen Computer anzuweisen, einen Reifen zu wechseln. Obwohl sehr eingeschränkt, werden Sie vom Durchschnittsmenschen verstanden (zumindest von Deutsch Sprechenden).

Das Programm Conversion.cpp, das Sie gerade eingegeben haben, enthält C++-Anweisungen, eine Sprache, die Sie in keiner Tageszeitung finden werden. So kryptisch und grob diese C++-Anweisungen auch auf Sie wirken, versteht der Computer eine Sprache, die noch viel elementarer ist als C++. Die Sprache, die Ihr Computer versteht, wird als Maschinensprache bezeichnet.

Der C++-Compiler übersetzt Ihr C++-Programm in die Maschinensprache Ihrer Mikroprozessor-CPU in Ihrem PC. Programme, die Sie unter Windows aufrufen können, GNU C++ eingeschlossen, sind nichts anderes als Dateien, die Maschinenanweisungen enthalten.

Es ist möglich, Programme direkt in Maschinensprache zu schreiben. Dies ist aber viel schwieriger, als das gleiche Programm in C++ zu schreiben.

Die wichtigste Aufgabe von GNU C++ ist, Ihr C++-Programm in eine ausführbare Datei zu übersetzen. Der Vorgang der Übersetzung in eine ausführbare .EXE-Datei wird als *Erzeugen* bezeichnet. Dieser Prozess wird manchmal auch als *Kompilieren* bezeichnet (es gibt einen Unterschied zwischen diesen beiden Begriffen, der aber hier nicht relevant ist). Der Teil des C++-Paketes, der die Übersetzung des Programms ausführt, wird als Compiler bezeichnet.

Um Ihr Programm Conversion.cpp zu erzeugen, klicken Sie auf »Compile« und dann auf »Make«, oder drücken Sie F9. `rhide` öffnet ein kleines Fenster am unteren Rand des Fensters, um den Fortschritt des Prozesses anzuzeigen. Wenn alles gut geht, erscheint die Meldung »Creating: Conversion.exe« gefolgt von »no errors« wie in Abbildung 3.3 zu sehen ist.

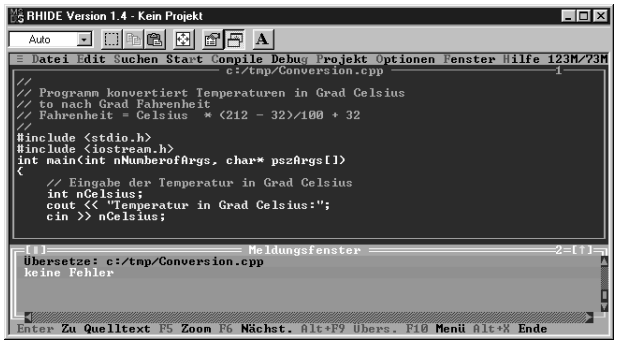


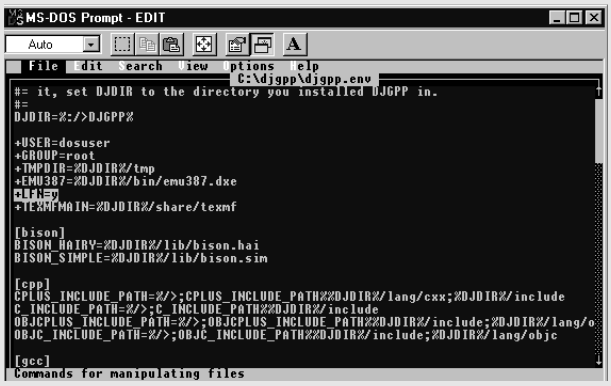
Abbildung 3.3: `rhide` ***zeigt »no errors« an, wenn kein Fehler aufgetreten ist.***

GNU C++-Installationsfehler

Einige häufig gemachte Fehler bei der Installation können Ihnen den Spaß an Ihrer ersten Programmiererfahrung verderben.

Die Meldung »Bad command or file name« bedeutet, dass MS-DOS die Datei gcc.exe nicht finden kann, d.h. den GNU C++-Compiler. Entweder haben Sie GNU C++ nicht richtig installiert oder Ihr Pfad enthält nicht c:\djgpp\bin, wo gcc.exe zu finden ist. Versuchen Sie, GNU C++ erneut zu installieren und stellen Sie sicher, dass das Kommando `SET PATH=c:\djgpp\bin;%PATH%` in Ihrer Datei autoexec.bat vorkommt. Nachdem Sie GNU C++ erneut installiert haben, starten Sie den Rechner neu.

Die Meldung »gcc.exe: Conversion.cpp: No such file or directory (ENOENT)« zeigt an, dass gcc nicht weiß, dass Sie lange Dateinamen verwenden (im Gegensatz zu den alten MS-DOS 8.3 Dateinamen). Um dies zu korrigieren, editieren Sie die Datei c:\djgpp\djgpp.env. Setzen Sie die Eigenschaft LFN auf Y, wie in der Abbildung zu sehen.



GNU C++ erzeugt eine Fehlermeldung, wenn es einen Fehler in Ihrem C++-Programm findet. Um diesen Prozess des Fehlermeldens zu demonstrieren, habe ich ein Semikolon am Ende einer Zeile entfernt und das Programm neu kompiliert. Das Ergebnis finden Sie in Abbildung 3.4.

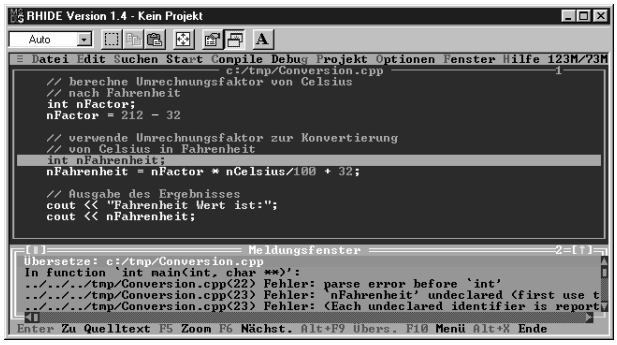


Abbildung 3.4: GNU C++ gibt Fehlermeldungen während des Erzeugungsprozesses aus.

Die Fehlermeldung in Abbildung 3.4 ist ein wenig imposant; sie ist aber einigermaßen ausdrucksvoll, wenn Sie sie Zeile für Zeile betrachten.

Die erste Zeile zeigt an, dass der Fehler entdeckt wurde, während der Code innerhalb von `main( )` analysiert wurde, d.h. der Code zwischen der öffnenden und schließenden Klammer, die auf das Schlüsselwort `main( )` folgen.

Die zweite Zeile zeigt an, dass nicht verstanden wurde, wie `int` in Zeile 22 da passt. Natürlich passt `int` nicht, aber ohne das Semikolon hat GNU C++ gedacht, dass die Zeilen 18 und 22 eine einzige Anweisung sind. Die übrigen Fehler stammen daher, dass Zeile 22 nicht verstanden werden konnte.

Um das Problem zu beheben, habe ich zuerst Zeile 22 analysiert (beachten Sie die Zeile 22:5 unten links im Code-Fenster – der Cursor ist in Spalte 5 von Zeile 22). Da Zeile 22 in Ordnung zu sein scheint, gehe ich zu Zeile 18 zurück und stelle fest, dass ein Semikolon fehlt. Ich füge das Semikolon ein und kompiliere neu. Diesmal kompiliert GNU C++ ohne Schwierigkeiten.

C++ Fehlermeldungen

Warum sind alle C++-Pakete so pingelig, wenn es um die Syntax von C++ geht? GNU C++ war in der Lage, das Fehlen des Semikolons in obigem Beispiel zu erkennen. Wenn ein C++-Compiler erkennt, dass ich ein Semikolon vergessen habe, warum kann es dieses Problem nicht einfach selber lösen und fortfahren?

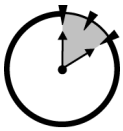
Die Antwort ist einfach aber profund. GNU C++ *denkt*, dass ich ein Semikolon vergessen habe. Ich könnte beliebig viele andere Fehler eingebaut haben, die GNU C++ als Fehlen eines Semikolons fehldiagnostiziert haben könnte. Wenn der Compiler einfach das Problem durch Einfügen eines Semikolons behebt, würde GNU C++ möglicherweise dadurch das eigentliche Problem verschleiern.

Wie Sie sehen werden, ist das Auffinden eines Fehlers in einem Programm, das ohne Probleme den Erzeugungsprozess durchläuft, schwierig und zeitaufwendig. Es ist besser, den Compiler Fehler finden zu lassen, wenn möglich.

Diese Lektion war hart zu Beginn. In den frühen Tagen des Computers versuchten Compiler, alle möglichen Fehler zu erkennen und selber zu korrigieren. Dies hatte manchmal lächerliche Züge.

Meine Freunde und ich machten uns einen Spaß daraus, einen »freundlichen« Compiler damit zu quälen, indem wir ein Programm eingaben, das nichts als die existenzielle Frage IF enthielt. (Rückschauend waren meine Freunde und ich ein wenig verrückt). Durch eine Reihe schmerzhafter Drehungen hat der besagte Compiler aus diesem einen Wort eine Kommandozeile generiert, die sich ohne Fehler übersetzen ließ. Ich weiß, dass der Compiler meine Absicht mit dem Wort IF missverstanden haben muss, weil ich nichts damit beabsichtigt hatte.

Meine Erfahrung ist, dass jedes Mal, wenn der Compiler versucht hat, ein Problem in einem Programm zu beheben, das Ergebnis falsch war. Trotz Fehlinformation war es keine Schwierigkeit, das Problem zu beheben, wenn der Compiler den Fehler gemeldet hat, bevor er ihn versuchte zu beheben. Compiler, die Fehler behoben haben, ohne entsprechende Fehlermeldungen auszugeben, haben mehr Schaden angerichtet als genützt.



10 Min.

3.4 Ausführen Ihres Programms

Um das Programm Conversion auszuführen, klicken Sie auf Start und wieder Start, oder drücken Sie Ctrl+F9 wie in Abbildung 3.5 zu sehen ist.

Abbildung 3.5: `rhide` ***öffnet ein Fenster, in dem das Programm ausgeführt wird.***

Sofort erscheint ein Fenster, in dem das Programm die Eingabe einer Temperatur in Grad Celsius erwartet, wie in Abbildung 3.6.

Abbildung 3.6: Die Temperatur wird in Fahrenheit angezeigt.

Geben Sie eine bekannte Temperatur ein, z.B. 100 Grad. Nach Drücken der Returntaste gibt das Programm die äquivalente Temperatur von 212 Grad Fahrenheit zurück. Weil `rhide` das Fenster des Programms sofort schließt, wenn das Programm beendet ist, haben Sie keine Chance, den Inhalt des Fensters zu lesen, bevor es geschlossen wird. `rhide` öffnet einen Dialog mit der Meldung, dass das Programm mit einem Fehlercode von Null beendet wurde. Abgesehen von der Bezeichnung »Fehlercode« bedeutet Null, dass kein Fehler aufgetreten ist.

Um die Ausgabe des bereits beendeten Programms einzusehen, klicken Sie auf den Menüpunkt »Nutzerbildschirm« des Fenster-Menüs oder drücken Sie Alt+F5. Dieses Fenster zeigt das aktuelle

MS-DOS-Fenster. In diesem Fenster sehen Sie die letzten 25 Zeilen der Ausgabe Ihres Programms, die auch die Ausgabe der berechneten Temperatur in Fahrenheit enthält, wie in Abbildung 3.6 zu sehen ist.

Glückwunsch! Sie haben Ihr erstes Programm mit GNU C++ eingegeben, erzeugt und ausgeführt.

3.5 Abschluss

Es gibt zwei Punkte, auf die hingewiesen werden sollte. Erstens ist GNU C++ nicht dafür gedacht, Windows-Programme zu schreiben. Theoretisch könnten Sie mit GNU C++ ein Windows-Programm schreiben, das wäre aber nicht einfach, ohne die Bibliotheken von Visual C++ zu verwenden. Zweitens bietet GNU C++ eine Art Hilfe an, die sehr nützlich sein kann.

3.5.1 Programmausgabe

Windows-Programme haben eine sehr visuell ausgerichtete, Fenster-basierte Ausgabe. Conversion.exe ist ein 32-Bit-Programm, das unter Windows ausgeführt wird, aber kein Windows-Programm im eigentlichen Sinne ist.

Wenn Sie nicht wissen, was die Phrase »32-Bit-Programm« bedeutet, brauchen Sie sich keine Sorgen zu machen.

Wie ich bereits in der Einleitung erläutert habe, ist dies kein Buch über das Schreiben von Windows-Programmen. Die C++-Programme, die Sie in diesem Buch schreiben, haben ein Kommandozeilen-Interface, das innerhalb einer DOS-Box ausgeführt wird. Angehende Windows-Programmierer sollten nicht verzweifeln – Sie haben Ihr Geld nicht umsonst ausgegeben. Das Erlernen von C++ ist Grundvoraussetzung für das Schreiben von Windows-Programmen in C++.

0 Min.

3.5.2 GNU C++ Hilfe

GNU C++ stellt über das Benutzerinterface von `rhide` ein Hilfesystem bereit. Platzieren Sie den Cursor auf ein Konstrukt, das Sie nicht verstehen, und drücken Sie F1; ein Fenster erscheint wie in Abbildung 3.7. Alternativ können Sie auch »Index« im Hilfe-Menü auswählen, um sich eine Liste von Hilfepunkten anzeigen zu lassen. Klicken Sie auf den Hilfepunkt, den Sie angezeigt haben möchten.

Abbildung 3.7: `rhide` ***stellt Hilfe über F1 und einen Index bereit.***

Die Hilfe von GNU C++ ist nicht so umfangreich, wie die von Visual C++. Wenn Sie z.B. mit dem Cursor auf `int` ***gehen und F1 drücken, erscheint ein Fenster, das den Editor beschreibt. Nicht gerade das, was ich haben wollte. Die Hilfe von GNU C++ konzentriert sich auf Bibliotheksfunktionen und Compiler-optionen. Glücklicherweise ist die Hilfe von GNU C++ in den meisten Fällen ausreichend, wenn Sie erst einmal C++ beherrschen.***

Zusammenfassung

GNU C++ stellt eine benutzerfreundliche Umgebung bereit, in der Sie Programme mit Hilfe des Hilfsprogramms `rhide` erzeugen und testen können. Sie können `rhide` in ähnlicher Weise wie Visual C++ benutzen. Sie können den Editor von `rhide` verwenden, um den Code einzugeben, und den `rhide`-Erzeuger, um den Quelltext in Maschinencode zu überführen. Schließlich ermöglicht es `rhide`, das fertige Programm in derselben Umgebung laufen zu lassen.

Das nächste Kapitel geht Schritt für Schritt durch das C++-Programm.

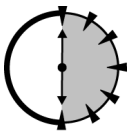
Selbsttest

1. Was für eine Datei ist ein C++-Quellprogramm? (Ist es eine Word-Datei? Eine Excel-Datei? Eine Textdatei?) (Sehen Sie sich den ersten Abschnitt von »Ihr erstes Programm« an)
2. Beachtet C++ das Einrücken? Achtet es auf Groß- und Kleinschreibung? (Siehe »Ihr erstes Programm«)
3. Was bedeutet »Erzeugen Ihres Programms«? (Siehe »Erzeugen Ihres Programms«)
4. Warum erzeugt C++ Fehlermeldungen? Warum versucht C++ nicht einfach, daraus schlau zu werden, was ich eingebe? (Siehe Kasten »C++-Fehlermeldungen«)

C++-Instruktionen

Checkliste

- ☑ Programm »Conversion« aus Sitzungen 2 und 3 erneut betrachten
- ☑ Die Teile eines C++-Programms verstehen
- ☑ Häufig verwendete C++-Kommandos einführen



30 Min.

In den Sitzungen 2 und 3 haben Sie ein C++-Programm eingegeben. Die Idee dahinter war, die C++-Umgebung kennen zu lernen (welche Umgebung auch immer Sie gewählt haben) und weniger, das Programmieren dabei zu erlernen. In dieser Sitzung wird das Programm Conversion.cpp genauer analysiert. Sie werden sehen, was genau jeder Teil des Programms tut, und wie jeder Programmteil seinen Beitrag zur Lösung leistet.

4.1 Das Programm

Listing 4-1 ist (wieder) das Programm Conversion.cpp, außer, dass es nun mit Kommentaren versehen ist, die wir im Rest der Sitzung behandeln wollen.

Es gibt mehrere Aspekte des Programms, die Sie erst einmal glauben müssen. Seien Sie geduldig. Jede Struktur des Programms wird zu ihrer Zeit erklärt werden.

Diese Version hat Extra-Kommentare ...

Listing 4-1

```
//
// Conversion konvertiert Temperaturen
// in Grad Celsius nach Grad Fahrenheit:
// Fahrenheit = Celsius  * (212 - 32)/100 + 32
//
#include <stdio.h>          // Rahmen
#include <iostream.h>
int main(int nNumberofArgs, char* pszArgs[])
{
    // hier ist unsere erste Anweisung -
    // es ist eine Deklaration
    int nCelsius;

    // unsere ersten I/O-Anweisungen
    cout << »Temperatur in Grad Celsius:«;
    cin  >> nCelsius;

    // die Zuweisung enthält eine Berechnung
    int nFactor;
    nFactor = 212 - 32;

    // eine Zuweisung, die einen Ausdruck enthält,
    // in dem eine Variable vorkommt
    int nFahrenheit;
    nFahrenheit = nFactor * nCelsius/100 + 32;

    // Ausgabe des Ergebnisses
    cout << »Fahrenheit Wert ist:«;
    cout << nFahrenheit;

    return 0;
}
```

4.2 Das C++-Programm erklärt

Unser menschliches Programm in Sitzung 1 bestand aus einer Folge von Anweisungen. In gleicher Weise besteht ein C++-Programm aus einer Folge von C++-Anweisungen, die der Computer in dieser Reihenfolge verarbeitet. Diese Anweisungen lassen sich in eine Reihe von Typen einteilen. Jeder Typ wird hier beschrieben.

4.2.1 Der grundlegende Programmaufbau

Jedes Programm, das in C++ geschrieben wird, beginnt mit dem gleichen grundlegenden Aufbau:

```
#include <stdio.h>
#include <iostream.h>
int main(int nNumberofArgs, char* pzArgs[])
{
    ... Ihr Code steht hier ...
    return 0;
}
```

Sie brauchen sich über die Details dieses Aufbaus keine Gedanken zu machen – die Details werden später behandelt – aber Sie sollten so einen ersten Eindruck haben, wie sie aussehen. Die ersten beiden Zeilen werden **Include-Anweisungen** genannt, weil Sie dafür sorgen, dass der Inhalt der bezeichneten Datei an diesem Punkt in das Programm eingefügt wird. Wir verstehen sie zu diesem Zeitpunkt einfach als Magie.

Die nächste Anweisung in dem Programmrahmen ist die Anweisung `int main`(...). Diese wird gefolgt von einer öffnenden und einer schließenden Klammer. Ihr Programm steht zwischen diesen beiden Klammern. Die Ausführung des Programms beginnt nach der öffnenden Klammer und endet bei der `return`-Anweisung, die unmittelbar vor der schließenden Klammer steht.

Unglücklicherweise müssen wir eine detailliertere Beschreibung des Programmrahmens auf spätere Kapitel verschieben. Machen Sie sich keine Sorgen ... wir kommen noch dazu, bevor das Wochenende vorbei ist.

4.2.2 Kommentare

Die ersten Programmzeilen scheinen frei formuliert zu sein. Entweder ist dieser »Code« für das menschliche Auge gedacht, oder der Computer ist doch schlauer, als wir immer gedacht haben. Die ersten sechs Zeilen werden als Kommentare bezeichnet. Ein **Kommentar** ist eine Zeile oder ein Teil einer Zeile, die vom C++-Compiler ignoriert wird. Kommentare ermöglichen es dem Programmierer, zu erklären, was er oder sie beim Schreiben des Codes gedacht hat.

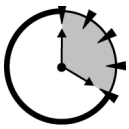
Ein C++-Kommentar beginnt mit einem Doppelslash (»//«) und endet mit einer neuen Zeile. Sie können beliebige Zeichen in Ihren Kommentaren verwenden. Kommentare können so lang sein wie Sie wollen, aber es ist übersichtlicher, wenn sie nicht länger als ca. 80 Zeichen werden; das ist die Länge, die vom Bildschirm dargestellt werden kann.

Ein **Zeilenumbruch** würde in den frühen Tagen der Schreibmaschine als »Carriage Return« bekannt geworden sein, als der Vorgang der Zeicheneingabe in eine Maschine als »Tippen« und nicht als »Keyboarding« bezeichnet wurde. Ein Zeilenumbruch ist das Zeichen, das eine Kommandozeile beendet.

C++ erlaubt eine zweite Art Kommentar, in der alles nach /* und vor dem nächsten */ ignoriert wird; diese Art des Kommentars wird in C++ normalerweise nicht mehr verwendet.

Es kommt Ihnen vielleicht komisch vor, dass es in C++ oder einer anderen Programmiersprache ein Kommando gibt, das vom Computer ignoriert wird. Jede Programmiersprache hat so etwas in irgendeiner Form. Es ist wichtig, dass der Programmierer erklärt, was ihm oder ihr durch den Kopf gegangen ist, als der Code geschrieben wurde. Für jemanden, der das Programm nimmt, um es zu benutzen oder zu modifizieren, muss das sonst nicht offensichtlich sein. In der Tat kann die Idee hinter dem Programm selbst für den Programmierer nach einigen Monaten nicht mehr offensichtlich sein.

Verwenden Sie Kommentare früh und oft.



20 Min.

4.2.3 Noch mal der Rahmen

Die nächsten vier Zeilen beschreiben den Programmrahmen, den ich schon früher einmal erwähnt habe. Erinnern Sie sich daran, dass die Programmausführung bei der ersten Anweisung nach der öffnenden Klammer beginnt.

4.2.4 Anweisungen

Die erste Zeile, die kein Kommentar ist, ist eine C++-Anweisung. Eine **Anweisung** ist eine einzelne Menge von Kommandos. Alle Anweisungen, die keine Kommentare sind, enden mit einem Semikolon (;). (Es gibt einen Grund, weshalb Kommentare dies nicht tun, aber er ist obskur. Meiner Meinung nach sollten auch Kommentare mit einem Semikolon enden und wenn es nur wegen der Konsistenz ist.)

Wenn Sie sich das Programm ansehen, werden Sie bemerken, dass es Leerzeichen, Tabulatorzeichen und Zeilenumbrüche enthält. Und tatsächlich habe ich jeder Anweisung im Programm einen Zeilenumbruch folgen lassen. Diese Zeichen werden unter dem Sammelbegriff Leerraum zusammengefasst, weil Sie keines dieser Zeichen auf dem Bildschirm sehen können. Ein **Leerraum** ist ein Leerzeichen, ein Tabulator, ein vertikaler Tabulator oder ein Zeilenumbruch. C++ ignoriert Leerraum.

Sie können Leerraum an jeder beliebigen Stelle in Ihrem Programmtext einfügen, um die Lesbarkeit zu erhöhen, außer innerhalb von Worten.

Während C++ Leerraum ignoriert, unterscheidet es Groß- und Kleinschreibung. Die Variablen `fullspeed` und `FullSpeed` haben nichts miteinander zu tun. Während das Kommando `int` verstanden wird, hat C++ keine Ahnung, was `INT` bedeuten soll.

4.2.5 Deklarationen

Die Zeile `int nCelsius;` ist eine Deklarationsanweisung. Eine **Deklaration** ist eine Anweisung, die eine Variable definiert. Eine **Variable** ist ein Platzhalter für Werte eines bestimmten Typs. Eine Variable enthält einen Wert, wie eine Zahl oder ein Zeichen.

Der Begriff *Variable* kommt von algebraischen Gleichungen der Form:

```
x = 10
y = 3 * x
```

Im zweiten Ausdruck, ist `y` gleich 3 mal `x`, aber was ist `x`? Die Variable `x` fungiert als Platzhalter für einen Wert. In diesem Fall ist der Wert von `x` gleich 10, aber wir hätten den Wert von `x` ebensogut auf 20, 30 oder -1 setzen können. Die zweite Formel macht immer Sinn, unabhängig vom Wert von `x`.

In der Algebra ist es erlaubt, mit einer Anweisung wie *`x = 10`* zu beginnen. In C++ muss der Programmierer eine Variable erst definieren, bevor er sie benutzen kann.

In C++ hat jede Variable einen Typ und einen Namen. Die Zeile `int nCelsius;` deklariert eine Variable `nCelsius`, die einen Integerwert aufnehmen kann. (Warum haben sie es nicht einfach

integer anstatt int genannt? Ich weiß es nicht. Das ist einfach eines der Dinge, mit denen Sie zu leben lernen müssen.)

Der Name einer Variable hat keine besondere Bedeutung für C++. Eine Variable muss mit einem Buchstaben beginnen ('A' bis 'Z' oder 'a' bis 'z'). Alle weiteren Zeichen müssen entweder ein Buchstabe, eine Ziffer ('0' bis '9') oder ein Unterstrich ('_') sein. Variablennamen können so lang sein, wie es für Sie Sinn macht.

Es gibt natürlich eine Beschränkung, aber die ist viel größer als die Grenze des Lesers. Gehen Sie nicht über eine Länge hinaus, die sich der Leser nicht bequem behalten kann, sagen wir 20 Zeichen.

Nach Konvention beginnen Variablen mit einem kleinen Buchstaben. Jedes neue Wort in einer Variablen beginnt mit einem Großbuchstaben wie in der Variable `myVariable`***. Ich erkläre die Bedeutung des*** `n` ***in*** `nCelsius` ***in Sitzung 5.***

Versuchen Sie, Variablennamen kurz, aber aussagekräftig zu wählen. Vermeiden Sie Namen wie `x`***, weil*** `x` ***keine Bedeutung hat. Eine Variable mit dem Namen*** `lengthOfLineSegment` ***ist viel aussagekräftiger.***

10 Min.

4.2.6 Eingabe/Ausgabe

Die Zeilen, die mit `cin` und `cout` beginnen, werden als Eingabe-/Ausgabe-Anweisungen bezeichnet, oder kurz I/O-Anweisungen (von Input/Output). (Wie alle Ingenieure lieben Programmierer Abkürzungen und Kürzel.)

Die erste I/O-Anweisung gibt die Phrase »Temperatur in Grad Celsius:« auf `cout` (gesprochen »see-out«) aus. `cout` ist der Name des Standard-Ausgabe-Device von C++. In diesem Fall ist der Monitor das Ausgabe-Device.

Die nächste Zeile ist genau das Gegenteil. Die Zeile besagt, dass ein Wert aus dem Eingabe-Device von C++ bezogen, und in der Variable `nCelsius` gespeichert werden soll. Das Standard-Eingabe-Device von C++ ist die Tastatur. Das ist das C++-Analogon zu der algebraischen Formel *x=10*, die oben erwähnt wurde. Im Rest des Programms ist der Wert von `nCelsius` so, wie der Benutzer ihn hier eingegeben hat.

4.2.7 Ausdrücke

In den nächsten beiden Zeilen, die als Berechnungsausdrücke gekennzeichnet sind, deklariert das Programm eine Variable `nFactor`, und weist ihr den Ergebniswert einer Rechnung zu. Die Rechnung berechnet die Differenz von 212 und 32. In C++ wird eine solche Formel als Ausdruck bezeichnet.

Ein **Operator** ist ein Kommando, das einen Wert generiert. Der Operator in dieser Berechnung ist »-«.

Ein **Ausdruck** ist ein Kommando, das einen Wert hat. Der Ausdruck ist hier »212 – 32«.

4.2.8 Zuweisung

Die gesprochene Sprache kann mehrdeutig sein. Der Begriff *gleich* ist eine dieser Mehrdeutigkeiten.

Das Wort *gleich* kann bedeuten, dass zwei Dinge den gleichen Wert haben, wie etwa 100 Cents gleich einem Dollar sind. Gleich kann auch wie in der Mathematik eine Zuweisung bedeuten, wie etwa y gleich 3 mal x.

Um Mehrdeutigkeiten zu vermeiden, rufen C++-Programmierer den **Zuweisungsoperator** = auf. Der Zuweisungsoperator speichert den Wert auf der rechten Seite in der Variablen auf der linken Seite. Programmierer sagen, dass `nFactor` der Wert 212 – 32 zugewiesen wird.

0 Min.

4.2.9 Ausdrücke (Fortsetzung)

Der zweite Ausdruck im Conversion.cpp ist ein wenig komplexer als der erste Ausdruck. Dieser Ausdruck benutzt einige mathematische Symbole: * für die Multiplikation, / für Division und + für Addition. In diesem Fall wird die Berechnung auf Variablen, und nicht auf einfachen Konstanten ausgeführt.

Der Wert in der Variable `nFactor` (der unmittelbar vorher berechnet wurde) wird mit dem Wert in `nCelsius` multipliziert (der gleich der Tastatureingabe ist). Das Ergebnis wird durch 100 geteilt und um 32 erhöht. Das Ergebnis des gesamten Ausdrucks wird der Integervariablen `nFahrenheit` zugewiesen.

Die letzten beiden Kommandos geben den Text »Fahrenheit Wert ist:« auf dem Bildschirm aus, gefolgt vom Wert von `nFahrenheit`.

Zusammenfassung

Sie haben schließlich eine Erklärung des Programms Conversion gesehen, das Sie in den Sitzungen 2 und 3 eingegeben haben. Notwendigerweise waren diese Erklärungen auf einem hohen Abstraktionsniveau. Die Details kommen später.

- Alle Programme beginnen mit demselben Rahmen.
- C++ erlaubt Ihnen, Kommentare einzubinden, die Ihnen und anderen erklären, was die einzelnen Programmteile tun.
- C++-Ausdrücke sehen aus wie algebraische Ausdrücke, mit dem Unterschied, dass C++-Variablen deklariert sein müssen, bevor sie benutzt werden.
- = wird Zuweisung genannt.
- Eingabe- und Ausgabeanweisungen beziehen sich defaultmäßig in C++ auf die Tastatur bzw. den Bildschirm oder das MS-DOS-Fenster.

Selbsttest

1. Was tut die folgende C++-Anweisung (Siehe »Kommentare«)

```
// Ich habe mich verlaufen
```

2. Was tut die folgende C++-Anweisung (Siehe »Deklarationen«)

```
int nQuizYourself; // hilf mir hier raus
```

3. Was tut die folgende C++-Anweisung (Siehe »Eingabe/Ausgabe«)

```
cout << »Hilf mir hier raus«
```

4. Was tut die folgende C++-Anweisung (Siehe »Ausdrücke«)

```
nHelpMeOutHere = 32;
```

Freitagabend – Zusammenfassung

1. *Schreiben Sie ein Programm, das ein Auto herunterlässt, unter Verwendung dieser Objekte:*

```
Auto
Reifen
Radmutter
Wagenheber
Schraubenschlüssel
```

Lassen Sie uns weiterhin annehmen, dass unser Prozessor die folgenden Aktionen versteht:

```
greifen
bewegen, nach oben bewegen, nach unten bewegen
loslassen
drehen
```

Hinweise:

a. Sie müssen annehmen, dass die Prozessorperson hoch und runter versteht und dass ein Wagenheber einen Griff hat.

b. Nicht alle zur Verfügung stehenden Substantive und Verben werden verwendet.

2. *Entfernen Sie das Minuszeichen zwischen 212 und 32 und erzeugen Sie das Programm neu. Zeichnen Sie die Fehlermeldung auf, und erklären Sie sie.*
3. *Beheben Sie das »Problem« so, wie es von Visual C++/GNU C++ vorgeschlagen wird, und erzeugen Sie Ihr Programm neu.*
4. *Führen Sie das Programm aus, und geben Sie einen bekannten Wert ein, wie etwa 100 Grad Celsius. Zeichnen Sie das Ergebnis auf.*
5. *Erklären Sie das resultierende Verhalten.*
6. *Erklären Sie, warum dies eine sehr unglückliche Situation wäre, wenn Visual C++/GNU C++ selbstkorrigierende Compiler wären.*

   Hinweis: Glauben Sie es oder nicht, »32;« ist ein gültiges Kommando.

7. Entfernen Sie die schließenden Hochkommata in Zeile 28 von Conversion.cpp innerhalb von rhide, so dass die Zeile so aussieht:

```
cout << »Fahrenheit Wert ist:;
```

Erzeugen Sie das Programm, und achten Sie auf Fehlermeldungen.

8. Können Sie die Fehlermeldungen erklären, die Sie sehen?

9. Wenn GNU C++ versuchen würde, das Problem automatisch zu beheben, was würde es tun? Probieren Sie diese Lösung aus, und erzeugen Sie das Programm neu. Achten Sie auf das Ergebnis.

10. Führen Sie das »berichtigte« Programm aus. Geben Sie eine Temperatur von 100 Grad Celsius ein, und achten Sie auf das Ergebnis.

11. Denken Sie einen Augenblick darüber nach, was passiert wäre, wenn GNU C++ seine eigene Lösung angewendet hätte. Hilft ein solcher Zugang? Ist er schädlich?

12. Irgendwelche Kommentare?

Hinweise:

a. GNU C++ denkt, dass eine Zeichenkette bei einem Hochkomma beginnt und bei dem nächsten Hochkomma endet.

b. Der rhide-Editor hebt Worte hervor in Abhängigkeit davon, als was er die Worte interpretiert. Zeichenketten erscheinen hellblau.

c. Wenn Sie verwirrt sind, gehen Sie zur Antwort von Frage 9 (siehe Anhang A). Wenn Sie diesen Hinweis verstanden haben, dann gehen sie zurück und beantworten alle diese Fragen.

13. Welche der folgenden Namen sind gültige Variablennamen?

a. twoFeetOfRope

b. 2FeetOfRope

c. two Feet Of Rope

d. engthOf2Ropes

e. &moreRope

14. Schreiben Sie ein Programm, das den Benutzer auffordert, drei Zahlen einzugeben, und dann die Summe dieser Zahlen ausgibt. Zusatz: Geben Sie das Mittel der eingegebenen Zahlen aus anstelle ihrer Summe.

Hinweise:

a. Beginnen Sie mit dem Standardrahmen von C++.

b. Vergessen Sie nicht, Ihre Variablen zu deklarieren.

c. Vergessen Sie nicht, dass Division eine höhere Priorität hat als Addition. Es könnte sein, dass Sie Klammern verwenden müssen, insbesondere im Zusatzprogramm.


Freitag
Samstag
Sonntag

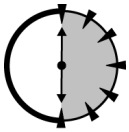
Samstagmorgen

Teil 2

Variablentypen

Checkliste

- ☑ Variablen deklarieren
- ☑ Mit den Grenzen von Integervariablen umgehen
- ☑ float-Variablen verwenden
- ☑ Deklaration und Benutzung anderer Variablentypen



30 Min.

Ein Problem des Programms Conversion in Teil I ist, dass es nur mit Integerwerten arbeitet. Das ist kein Problem im alltäglichen Leben – es ist unwahrscheinlich, dass jemand eine Temperatur wie z.B. 10.5 eingeben wird. Ein schlimmeres Problem sind die Rundungsfehler. Die meisten Integerwerte in Celsius werden auf einen nicht ganzzahligen Wert in Fahrenheit abgebildet. Das Programm Conversion kümmert sich nicht darum. Es schneidet einfach die Nachkommastellen ab, ohne eine Warnung auszugeben.

Es muss nicht offensichtlich sein, aber das Programm wurde sorgfältig geschrieben, um die Auswirkungen des Rundens auf die Ausgabe so gering wie möglich zu halten.

Dieses Kapitel untersucht die Begrenzungen von Integervariablen. Auch andere Variablentypen werden untersucht, die eingeschlossen, die zur Reduktion von Rundungsfehlern eingeführt wurden. Wir werden uns ihre Vor- und Nachteile anschauen.

5.1 Dezimalzahlen

Integerzahlen sind die Zahlen, an die Sie am meisten gewöhnt sind: 1, 2, 3, usw. und die negativen Zahlen -1, -2, -3 usw. Im alltäglichen Leben sind Integerzahlen am nützlichsten; leider können sie keine gebrochenen Zahlen darstellen. Brüche aus Integerzahlen wie z.B. ⅔, 15⁄16 oder 3 11⁄26 sind zu umständlich, um damit zu arbeiten. Wenn zwei Brüche nicht den gleichen Nenner haben, ist es schwer, sie zu vergleichen. Z.B. bei der Arbeit am Auto hat es lange gedauert, bis ich wusste, welche Schraube größer ist – ¾ oder 25⁄32 (es ist die letztere).

Mir wurde gesagt, aber ich kann es nicht beweisen, dass das Problem mit den Integerbrüchen zu der ansonsten unerklärlichen Bedeutung der 12 in unserem Alltag führt. 12 ist die kleinste Integerzahl, die durch 4, 3 und 2 teilbar ist. Ein Viertel von etwas ist in den meisten Fällen genug.

Die Einführung dezimaler Bruchteile hat sich als große Verbesserung herausgestellt. Es ist klar, dass 0.75 kleiner ist als 0.78 (das sind die gleichen Werte wie oben, ausgedrückt als Dezimalzahlen. Außerdem sind mathematische Berechnungen auf Gleitkommazahlen leichter, weil wir keinen kleinsten gemeinsamen Nenner finden oder anderen Unsinn machen müssen.

5.1.1 Begrenzungen von int in C++

Der Variablentyp `int` ist die C++-Version von Integer. Variablen vom Typ `int` leiden unter den gleichen Einschränkungen, unter denen auch ihre Zahläquivalente leiden.

Integer runden

Betrachten Sie das Problem, den Mittelwert von 3 Zahlen zu berechnen. (Das war eine Zusatzaufgabe in Sitzung 4.)

Gegeben seien drei `int`-Variablen `nValue1`, `nValue2` und `nValue3`. Eine Formel für die Berechnung des Mittelwertes ist

```
(nValue1 + nValue2 + nValue3) / 3
```

Diese Gleichung ist richtig und einsichtig. Lassen Sie uns die folgende ebenfalls korrekte und einsichtige Lösung betrachten:

```
nValue1/3 + nValue2/3 + nValue3/3
```

Um zu sehen, welchen Effekt das haben kann, betrachten Sie das folgende einfache Programm, das beide Methoden zur Berechnung des Mittelwertes benutzt:

```
// IntAverage - berechnet Mittelwert von drei
//              Zahlen mit Typ int
#include <stdio.h>
#include <iostream.h>

int main(int nArg, char* pszArgs[])
{
    int nValue1;
    int nValue2;
    int nValue3;

    cout << »Integerversion\n«;
    cout << »Geben Sie drei Zahlen ein.\n«;
    cout << »Drücken Sie Return nach jeder Zahl.\n«;
    cout << »#1:«;
    cin  >> nValue1;

    cout << »#2:«;
    cin  >> nValue2;

    cout << »#3:«;
    cin  >> nValue3;

    // die folgende Lösung hat keine so großen
    // Probleme mit Rundungsfehlern
    cout << »Addieren vor Teilen ergibt Mittelwert:«;
    cout << (nValue1 + nValue2 + nValue3)/3;
    cout << »\n«;

    // diese Version hat große Probleme mit
    // Rundungsfehlern
    cout << »Teilen vor Addieren ergibt Mittelwert:«;
    cout << nValue1/3 + nValue2/3 + nValue3/3;
    cout << »\n«;

    cout << »\n\n«;
    return 0;
 }
```

Dieses Programm bekommt die drei Werte `nValue1`, `nValue2` und `nValue3` von `cin`, d.h. über die Tastatur. Es gibt dann den Mittelwert aus, der mit der Addition-vor-Division-Methode berechnet wurde, gefolgt von dem Mittelwert der mit der Division-vor-Addition-Methode berechnet wurde.

Nachdem ich das Programm erzeugt hatte, habe ich es ausgeführt, und die Werte 1, 2 und 3 eingegeben. Erwartet hatte ich, zweimal das Ergebnis 2 zu bekommen. Stattdessen bekam ich das Ergebnis, das Sie in Abbildung 5.1 sehen.

Abbildung 5.1: Der Division-vor-Addition-Algorithmus hat große Probleme mit Rundungsfehlern.

Um den Grund für dieses merkwürdige Verhalten zu verstehen, lassen Sie uns die Werte 1, 2 und 3 direkt in die Gleichung 2 einfügen.

```
1/3 +> 2/3 + 3/3
```

Da Integerzahlen keine Brüche darstellen können, ist das Ergebnis einer Integeroperation immer der abgerundete Bruch.

Dieses Abrunden von Integerzahlen wird auch als Abschneiden bezeichnet.

Unter Berücksichtigung des Integerabschneidens wird der obige Ausdruck zu

```
0 + 0 + 1
```

oder 1.

Der Addition-vor-Division-Algorithmus verhält sich entscheidend besser:

```
(1 + 2 + 3) / 3
```

ist gleich ⅔ oder 2.

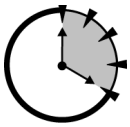
Doch auch der Addition-vor-Division-Algorithmus ist nicht immer korrekt. Geben Sie die Werte 1, 1 und 3 ein. Beide Algorithmen geben 1 anstelle von 1 ⅔ zurück.

Sie müssen sehr vorsichtig sein, wenn Sie Divisionen mit Integerzahlen durchführen, weil Abschneiden schnell passiert ist.

Eingeschränkter Bereich

Ein zweites Problem mit den `int`-Variablen ist der beschränkte Bereich. Eine normale Integervariable kann einen maximalen Wert von 2.147.483.647 und einen minimalen Wert von -2.147.483.648 annehmen, also ungefähr plus/minus zwei Milliarden.

Einige ältere (in der Tat »sehr alte«) Compiler beschränken den Bereich von `int`***-Variablen auf -32.768 bis 32.767.***



20 Min.

5.1.2 Lösen des Abschneideproblems

Glücklicherweise versteht C++ Dezimalzahlen. Dezimalzahlen werden in C++ als *Gleitkommazahlen*, oder kurz *floats*, bezeichnet. Der Begriff Gleitkomma kommt daher, dass der Dezimalpunkt hin- und hergleiten kann, so wie es notwendig ist, um einen bestimmten Wert darzustellen.

Gleitkommavariablen werden in der gleichen Weise deklariert wie `int`-Variablen:

```
float fValue1;
```

Um zu sehen, wie Gleitkommazahlen das Rundungsproblem lösen, das Integerzahlen inhärent ist, habe ich alle int-Variablen im Programm IntAverage durch float-Variablen ersetzt. (Das resultierende Programm finden Sie auf der beiliegenden CD-ROM unter dem Namen FloatAverage.)

FloatAverage konvertiert die Zahlen 1, 2 und 3 in den richtigen Mittelwert 2, für beide Algorithmen, wie in Abbildung 5.2 zu sehen ist.

Wenn Sie die Absicht haben, Berechnungen durchzuführen, halten Sie sich an Gleitkommazahlen.

Abbildung 5.2: Gleitkommavariablen berechnen den Mittelwert von 1, 2 und 3 korrekt.

5.1.3 Grenzen von Gleitkommazahlen

Während Gleitkommazahlen viele Berechnungsprobleme lösen können, wie z.B. Abschneidungen, haben Sie eine Reihe von Einschränkungen.

Zählen

An einer Stelle können Gleitkommavariablen nicht verwendet werden: wenn gezählt wird. Das betrifft auch C++-Konstrukte, bei denen ein Zähler verwendet wird. Das liegt daran, dass C++ nicht sicher sein kann, welche ganze Zahl mit einer Gleitkommazahl gemeint ist. Z.B. ist klar, dass 1.0 gleich 1 ist, aber was ist mit 0.9 und 1.1? Sollen diese auch als 1 angesehen werden?

C++ schließt diese Probleme aus, indem zum Zählen Variablen vom Typ `int` verwendet werden.

Berechnungsgeschwindigkeit

Historisch gesehen kann ein Computerprozessor ganzzahlige Arithmetik schneller ausführen als Arithmetik auf Gleitkommazahlen. Wenn also ein Prozessor in einer gegebenen Zeit 1000 ganze Zahlen addieren kann, so kann es sein, dass dieser Prozessor in der gleichen Zeit nur 200 Gleitkommaberechnungen ausführen kann. Das Problem der Berechnungsgeschwindigkeit wurde immer kleiner durch die Weiterentwicklung der Mikroprozessoren. Die meisten modernen Prozessoren enthalten spezielle Schaltkreise, mit denen Sie Gleitkommaberechnungen fast so schnell wie Ganzzahlenberechnungen ausführen können.

Genauigkeitsverluste

Auch Gleitkommavariablen lösen nicht alle Berechnungsprobleme. Gleitkommavariablen haben eine begrenzte Genauigkeit: ungefähr sechs Stellen.

Um zu sehen, warum dies ein Problem ist, betrachten Sie die Zahl ⅓, die als 0.333 ... mit Fortsetzung dieser Reihe dargestellt wird. Das Prinzip einer endlosen Fortsetzung dieser Reihe macht in der Mathematik Sinn, aber nicht in einem Computer. Der Computer hat eine begrenzte Genauigkeit. So ist eine Gleitkommaversion von ⅓ ungefähr 0.333333. Wenn diese 6 Nachkommastellen wieder mit 3 multipliziert werden, berechnet der Prozessor einen Wert von 0.999999 anstatt des mathematisch erwarteten Wertes 1. Der Genauigkeitsverlust, der auf die Beschränkungen der Gleitkommazahlen zurückzuführen ist, wird als *Rundungsfehler* bezeichnet. C++ kann viele Formen von Rundungsfehlern beheben. Z.B. kann C++ feststellen in der Ausgabe , dass der Benutzer 1 anstelle von 0.999999 gemeint hat. In anderen Fällen jedoch kann sogar C++ die Rundungsfehler nicht beheben.

»Nicht so begrenzter« Bereich

Der Datentyp `float` hat auch einen begrenzten Bereich, obwohl dieser Bereich viel größer ist als vom Datentyp `int`. Der maximale Wert ist ungefähr 10 hoch 38, das ist eine 1 mit 38 Nullen.

Nur die ersten sechs Ziffern haben eine Bedeutung, weil die restlichen 32 Ziffern unter Rundungsfehlern zu leiden haben. Eine Gleitkommavariable kann einen Wert von 123.000.000 ohne Rundungsfehler speichern, nicht aber 123.456.789.

5.2 Andere Variablentypen

C++ stellte andere Variablentypen neben `int` und `float` bereit. Diese sind in Tabelle 5-1 zu sehen. Jeder Typ hat seine Vorteile und seine Grenzen.

Tabelle 5-1: Andere C++-Variablentypen

Name	Beispiel	Sinn
`char`	'c'	Eine einzige `char`-Variable kann ein einzelnes alphabetisches Zeichen oder eine Ziffer speichern. (Die einfachen Hochkommata zeigen an, dass es sich um ein einzelnes Zeichen handelt.)
`string`	"Zeichenkette"	Eine Kette von Zeichen; ein Satz. Wird benutzt, um ganze Phrasen zu speichern. (Die doppelten Hochkommata zeigen an, dass es sich um eine *Zeichenkette* handelt.)
`double`	1.0	Ein größerer Gleitkommatyp, der 15 signifikante Stellen und ein Maximum von 10 hoch 308 hat.
`long`	10L	Ein großer Integertyp mit einem Bereich von -2 Milliarden bis 2 Milliarden.

So deklariert

```
// deklariere eine long-Variable und setze sie auf 1
long lVariable;
lVariable = 1;

// deklariere eine double-Variable und setze
// sie auf 1.0
double dVariable;
dVariable = 1.0;
```

die Variable `lVariable` als Variable vom Typ `long` und setzt ihren Wert auf 1, während `dVariable` vom Typ `double` ist, und auf den Wert 1.0 gesetzt wird.

Es ist möglich, eine Variable in der gleichen Anweisung zu deklarieren und zu initialisieren:

```
int nVariable = 1;  // deklariere eine Variable und
                    // initialisiere sie mit 1
```

Variablennamen haben für den C++-Compiler keine besondere Bedeutung.

Eine Variable von Typ `char` kann ein einzelnes Zeichen speichern, während eine Zeichenkette eine Kette von Zeichen enthält. So ist 'a' das Zeichen *a*, während "a" eine Zeichenkette ist, die nur das Zeichen a enthält. (»Zeichenkette« ist eigentlich kein Variablentyp, aber in den meisten Fällen können Sie sie als solchen behandeln. Sie erfahren mehr Details über Zeichenketten in Sitzung 14).

Das Zeichen 'a' und die Zeichenkette "a" sind nicht das Gleiche. Wenn eine Anwendung eine Zeichenkette benötigt, können Sie nicht ein einzelnes Zeichen zur Verfügung stellen, selbst wenn die erwartete Zeichenkette nur ein einzelnes Zeichen enthält.

`long` und `double` sind erweiterte Formen von `int` und `float` – long steht für langes Integer, und double steht für doppeltes float.

5.2.1 Typen von Konstanten

Beachten Sie, wie jeder Datentyp ausgedrückt wird.

In einem Ausdruck wie `n = 1;` ist die Konstante 1 ein `int`. Wenn Sie 1 als long gemeint haben, müssten Sie `n = 1L;` schreiben. Vergleichen könnte man das mit einer 1, die einem Ball auf der Ladefläche eines Pickups entspricht, während `1L` einem Ball auf einem LKW entspricht. Der Ball ist der gleiche, aber die Kapazität seines Behälters ist viel größer.

In gleicher Weise repräsentiert 1.0 den Wert 1 in einem Gleitkommacontainer. Beachten Sie jedoch, dass Gleitkommakonstanten defaultmäßig als `double` angenommen werden. Somit ist 1.0 eine Zahl vom Typ `double` und nicht vom Typ `float`.

5.2.2 Sonderzeichen

Allgemein können Sie jedes druckbare Zeichen in einer `char`-Variable oder in einer Zeichenkette speichern. Es gibt auch eine Menge von nicht druckbaren Zeichen, die so wichtig sind, dass das Gleiche auch für sie gilt. Tabelle 5-2 listet diese Zeichen auf.

Sie haben bereits das Zeichen für den Zeilenumbruch gesehen. Dieses Zeichen bricht eine Zeichenkette in einzelne Zeilen um.

Tabelle 5-2: Nicht druckbare, aber oft verwendete Zeichen

Zeichenkonstante	Bedeutung
'\n'	neue Zeile
'\t'	Tabulator
'\0'	Null
'\\'	Backslash

Bis jetzt haben wir einen Zeilenumbruch nur ans Ende einer Zeichenkette gestellt. Dieses Zeichen kann aber an jeder beliebigen Stelle in der Zeichenkette vorkommen. Betrachten Sie z.B. die folgende Zeichenkette:

```
cout << »Das ist Zeile 1\n Das ist Zeile 2«;
```

erzeugt die Ausgabe

```
Das ist Zeile 1
Das ist Zeile 2
```

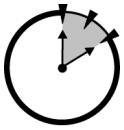
In gleicher Weise bewegt das '\t'-Zeichen die Ausgabe an die nächste Tabulatorposition. Was das genau bedeutet, hängt vom Typ des Computers ab, den Sie benutzen.

Da das Backslash-Zeichen verwendet wird, um spezielle Zeichen darzustellen, muss es ein Zeichenpaar geben, um den Backslash selber darzustellen. Das Zeichen '\\' stellt den Backslash dar.

C++-Konflikte mit MS-DOS-Dateiname

MS-DOS benutzt das Backslash-Zeichen, um Verzeichnisnamen im Pfad zu einer Datei zu trennen. So stellt Root\FolderA\File eine Datei in *FolderA* dar, das ein Unterverzeichnis von *Root* ist.

Leider erzeugt der Gebrauch des Backslash von MS-DOS und C++ einen Konflikt. Der MS-DOS-Pfad Root\FolderA\File wird durch die C++-Zeichenkette "Root\\FolderA\\File" dargestellt. Der doppelte Backslash wird durch den speziellen Gebrauch des Backslash-Zeichens in C++ notwendig.



10 Min.

5.3 Gemischte Ausdrücke

Ich hasse es fast, das hier darzustellen, aber C++ ermöglicht Ihnen, gemischte Variablentypen in einem Ausdruck zu verwenden. Das heißt, Sie können eine `int`-Zahl und eine `double`-Zahl addieren. Der folgende Ausdruck, in dem `nValue1` ein `int` ist, ist erlaubt:

```
// im folgenden Ausdruck wird der Wert von nValue1
// in ein double konvertiert, bevor die Zuweisung
// ausgeführt wird
int nValue = 1;
nValue1 = nValue1 + 1.0;
```

Ein Ausdruck, in dem die beiden Operanden nicht vom gleichen Typ sind, wird *Mischmodusausdruck* genannt. Mischmodusausdrücke erzeugen einen Wert von dem mächtigeren Typ der beiden Operanden. In diesem Fall wird `nValue1` in ein `double` konvertiert, bevor die Berechnung fortgesetzt wird.

In gleicher Weise kann ein Ausdruck eines Typs einer Variablen eines anderen Typs zugewiesen werden wie in der folgenden Anweisung:

```
// in der folgenden Anweisung wird der ganzzahlige
// Anteil von fVariable in nVariable gespeichert
fVariable = 1.0;
int nVariable;
nVariable = fVariable;
```

Genauigkeit oder Teile des Zahlenbereichs können verloren gehen, wenn die Variable auf der linken Seite der Anweisung »kleiner« ist. Im vorangegangenen Beispiel muss der Wert von fVariable abgeschnitten werden, bevor er in nVariable gespeichert werden kann.

0 Min.

Einen »größeren« Wert in eine »kleinere« Variable zu konvertieren, wird »*Promotion*« genannt, während die Konvertierung von Werten in der umgekehrten Richtung als »*Demotion*« bezeichnet wird. Wir sagen, dass der Wert der `int`-Variable `nVariable1` in ein `double` befördert (promoviert) wurde:

```
int nVariable = 1;
double dVariable = nVariable1;
```

Mischmodusausdrücke sind keine besonders gute Idee. Sie sollten Ihre eigenen Entscheidungen treffen, anstatt sie C++ zu überlassen. Es kann sein, dass C++ nicht richtig versteht, was Sie eigentlich wollen.

Namenskonventionen

Sie werden bemerkt haben, dass ich jeden Variablennamen mit einem bestimmten Zeichen beginne, das nichts mit dem Namen zu tun zu haben scheint. Diese speziellen Zeichen sind unten dargestellt. Wenn Sie diese Konvention verwenden, können Sie sofort erkennen, dass die Variable `dVariable` vom Typ double ist.

Zeichen	Typ
n	int
l	long
f	float
d	double
c	char (Zeichen)
sz	Zeichenkette

Diese führenden Zeichen helfen dem Programmierer, den Überblick über die Variablentypen zu behalten. Sie können sofort erkennen, dass es sich bei dem folgenden Ausdruck um einen Mischmodusausdruck handelt, der eine `long`-Variable und eine `int`-Variable enthält.

```
nVariable = lVariable;
```

Bedenken Sie jedoch, dass die Verwendung spezieller Zeichen in Variablennamen für C++ keine besondere Bedeutung hat. Ich hätte genauso `q` zur Identifizierung von `int` verwenden können. Manche Programmierer verwenden überhaupt keine Namenskonvention.

Zusammenfassung

Wie Sie gesehen haben, sind Integervariable effizient in Bezug auf Rechenzeit, und sie sind einfach zu handhaben. Sie haben jedoch Begrenzungen, wenn sie in Berechnungen eingesetzt werden. Gleitkommazahlen sind ideal für die Verwendung in mathematischen Gleichungen, da sie nicht unter signifikanten Rundungsfehlern oder einer signifikanten Beschränkung ihres Bereiches leiden. Auf der anderen Seite sind Gleitkommavariablen schwerer zu handhaben und nicht so universell einsetzbar wie Integervariablen.

- Integervariablen stellen Zähler dar, wie 1, 2, usw.
- Integervariablen haben einen Bereich von -2 Milliarden bis 2 Milliarden.
- Gleitkommavariablen stellen Dezimalbrüche dar.
- Gleitkommavariablen haben praktisch einen unbeschränkten Bereich.
- Der Variablentyp `char` wird zur Darstellung von ANSI-Zeichen verwendet.

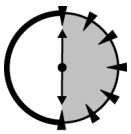
Selbsttest

1. Was ist der Bereich von `int`-Variablen? (Siehe »Eingeschränkter Bereich«)
2. Warum leiden `float`-Variablen nicht unter signifikanten Rundungsfehlern? (Siehe »Lösen des Abschneideproblems«)
3. Was ist der Typ der Konstante 1? Was ist der Typ von 1.0? (Siehe »Typen von Konstanten«)

Mathematische Operationen

Checkliste

- ☑ Mathematische Operatoren von C++ gebrauchen
- ☑ Ausdrücke erkennen
- ☑ Klarheit durch spezielle mathematische Operatoren erhöhen



30 Min.

Die Programme Conversion und Average haben einfache mathematische Operationen wie Addition, Multiplikation und Division verwendet. Ich habe diese Operatoren verwendet, ohne sie vorher zu beschreiben, weil sie intuitiv klar sind. Diese Sitzung beschreibt die Menge der mathematischen Operatoren.

Die mathematischen Operatoren sind in Tabelle 6-1 aufgelistet. Die erste Spalte listet die Operatoren von oben nach unten geordnet, gemäß ihrer Priorität.

Tabelle 6-1: Mathematische Operatoren von C++

Operator	Bedeutung
+ (unär)	tut effektiv nichts
- (unär)	gibt das Negative des Arguments zurück
++ (unär)	Inkrement
— (unär)	Dekrement
*	Multiplikation
/	Division
%	Modulo
+	Addition
-	Subtraktion
=, *=, %=, += -=	Zuweisungstypen

Jeder dieser Operatoren wird in den folgenden Abschnitten behandelt.

6.1 Arithmetische Operatoren

Die Operatoren Multiplikation, Division, Modulo, Addition und Subtraktion sind die Operatoren, die zur Ausführung gewöhnlicher Arithmetik verwendet werden. Jeder dieser Operatoren hat die übliche Bedeutung, die Sie aus der Schule kennen werden, vielleicht mit Ausnahme von Modulo.

Der Modulo-Operator ist dem sehr ähnlich, was mein Lehrer als »Rest nach Division« bezeichnet hat. Z.B. passt 4 drei mal in 15 hinein mit einem Rest von 3. Ausgedrückt in C++-Notation lautet dies »15 modulo 4 ist 3«.

Weil Programmierer immer bemüht sind, Nichtprogrammierer mit den einfachsten Dingen zu beeindrucken, definieren C++-Programmierer modulo wie folgt:

```
IntValue % IntDivisor
```

ist gleich

```
IntVaue - (IntValue / IntDivisor) * IndDivisor
```

Lassen Sie uns das an unserem früheren Beispiel ausprobieren:

```
15 % 4 ist gleich    15 - (15/4) * 4
                     15 - 3 * 4
                     15 - 12
                     3
```

Weil modulo auf Rundungsfehlern basiert, die Integerzahlen inhärent sind, ist modulo für Gleitkommazahlen nicht definiert.

6.2 Ausdrücke

Der häufigste Typ von C++-Anweisungen ist der Ausdruck. Ein *Ausdruck* ist eine Anweisung, die einen Wert hat. Alle Ausdrücke haben einen Wert.

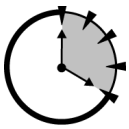
Z.B. ist eine Anweisung, die einen mathematischen Operator enthält, ein Ausdruck, da alle diese Operatoren einen Wert zurückgeben. Ausdrücke können einfach oder auch kompliziert sein. In der Tat ist »1« ein Ausdruck. Es gibt fünf Ausdrücke in der folgenden Anweisung:

```
z = x * y + w;
1. x * y + w
2. x * y
3. x
4. y
5. w
```

Ein ungewöhnlicher Gesichtspunkt in C++ ist, dass ein Ausdruck eine vollständige Anweisung ist. Somit ist das folgende eine gültige C++-Anweisung.

```
1;
```

Alle Ausdrücke haben einen Typ. Wie wir bereits festgestellt haben, ist der Typ des Ausdrucks `1` gleich `int`. In einer Zuweisung ist der Typ des Ausdrucks auf der rechten Seite der Anweisung gleich dem Typ der Variablen auf der linken Seite des Ausdrucks – wenn nicht, führt C++ die notwendigen Konvertierungen durch.



20 Min.

6.3 Vorrang von Operatoren

Jeder der C++-Operatoren hat eine Priorität, mit der er innerhalb von zusammengesetzten Ausdrücken (d.h. Ausdrücken mit mehr als einem Operator) ausgewertet wird. Diese Eigenschaft wird als Vorrang bezeichnet.

Der Ausdruck

```
x/100 + 32
```

teilt *x* durch 100 bevor 32 addiert wird. In einem gegebenen Ausdruck führt C++ Multiplikationen und Divisionen vor Additionen und Subtraktionen aus. Wir sagen, dass Multiplikation und Division *Vorrang* vor Addition und Subtraktion haben.

Was ist, wenn der Programmierer *x* durch 100 plus 32 teilen möchte? Der Programmierer kann Ausdrücke durch Klammern verbinden:

```
x/(100 + 32)
```

Dies hat den gleichen Effekt, wie *x* durch 132 zu teilen. Der Ausdruck innerhalb der Klammern wird zuerst ausgewertet. Dies ermöglicht dem Programmierer die Vorrangsregeln einzelner Operatoren zu überschreiben.

Der ursprüngliche Ausdruck

```
x / 100 + 32
```

ist identisch mit dem Ausdruck

```
(x / 100) + 32
```

Der Vorrang der Operatoren aus Tabelle 6-1 ist in Tabelle 6-2 zu sehen.

Tabelle 6-2: Mathematische Operatoren von C++ und ihre Priorität

Priorität	Operator	Bedeutung
1	+ (unär)	tut effektiv nichts
1	- (unär)	gibt das Negative des Argumentes zurück
2	++ (unär)	Inkrement
2	— (unär)	Dekrement
3	*	Multiplikation
3	/	Division
3	%	Modulo
4	+	Addition
4	-	Subtraktion
5	=, *=, %=, += -=	Zuweisungstypen

Operatoren mit der gleichen Priorität werden von links nach rechts ausgewertet. Somit ist der Ausdruck

```
x / 10 / 2
```

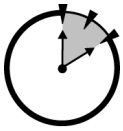
das Gleiche wie

```
(x / 10) / 2
```

Mehrstufige Klammerungen werden von innen nach außen ausgewertet. Im folgenden Ausdruck

```
(y / (2 + 3)) / x
```

wird die Variable y durch 5 geteilt, und das Ergebnis wird durch x geteilt.



10 Min.

6.4 Unäre Operatoren

Unäre Operatoren sind die Operatoren, die nur ein Argument haben.

Ein binärer Operator ist ein Operator, der zwei Argumente hat. Betrachten Sie z.B. a + b. In C++-Sprechweise sind die Argumente des Additionsoperators die Ausdrücke auf der linken und der rechten Seite des Operators.

Die unären Operatoren sind +, -, ++ und —.

Der Minusoperator ändert das Vorzeichen seines Arguments. Positive Zahlen werden negativ und umgekehrt. Der Plusoperator ändert das Vorzeichen seines Arguments nicht. Letztendlich hat der Plusoperator keinen Effekt.

Die Operatoren ++ und — inkrementieren bzw. dekrementieren ihre Argumente um eins.

Wozu einen eigenen Inkrement-Operator?

Die Autoren von C++ haben erkannt, dass Programmierer »1« mehr als jede andere Konstante addieren. Als Bequemlichkeitsfaktor wurde eine spezielle »Addiere 1«-Instruktion in der Sprache eingeführt.

Zusätzlich haben die meisten Computerprozessoren eine Inkrementanweisung, die schneller ist als die Additionsanweisung. Als C++ entstanden ist, war das Einsparen von Anweisungen eine wichtige Sache, wenn man sich den damaligen Entwicklungsstand der Mikroprozessoren bewusst macht.

Unabhängig davon, aus welchem Grund die Inkrement- und Dekrementoperatoren eingeführt wurden, werden Sie in Sitzung 7 sehen, dass sie nützlicher sind, als Sie jetzt vielleicht denken.

Die Inkrement- und Dekrementoperatoren sind auf nicht-Gleitkomma-Variablen beschränkt.

Die Inkrement- und Dekrementoperatoren sind in der Hinsicht besonders, dass Sie in zwei Formen vorkommen: einer Präfix-Version und einer Postfix-Version.

Die Präfix-Version des Inkrements wird als ++x geschrieben, während das Postfix durch x++ ausgedrückt wird.

Betrachten Sie den Inkrement-Operator (der Dekrement-Operator ist genau analog). Nehmen Sie an, dass die Variable n den Wert 5 hat. Beide, n++ und ++n, inkrementieren den Wert von n zu 6. Der Unterschied ist, dass der Wert von ++n in einem Ausdruck gleich 6 ist, während der Wert von n++ in einem Ausdruck gleich 5 ist. Das folgende Beispiel demonstriert das:

```
// deklariere drei int-Variablen
int n1, n2, n3;

// der Wert von n1 und n2 ist gleich 6
n1 = 5;
n2 = ++n1;

// der Wert von n1 ist 6, aber der Wert n3 ist 5
n1 = 5;
n3 = n1++;
```

Somit bekommt `n2` den Wert von `n1`, *nachdem* `n1` über die Präfix-Version inkrementiert wurde, während `n3` den Wert von `n1` erhält, *bevor* `n1` über die Postfix-Version inkrementiert wird.

6.5 Zuweisungsoperatoren

Die Zuweisungsoperatoren sind binäre Operatoren, die das Argument auf ihrer linken Seite verändern.

Der einfache Zuweisungsoperator '=' ist eine absolute Notwendigkeit in jeder Programmiersprache. Der Operator speichert den Wert des Arguments auf der rechten Seite im Argument auf der linken Seite. Die anderen Zuweisungsoperatoren scheinen irgendeiner Laune entsprungen zu sein.

Die Autoren von C++ haben festgestellt, dass Zuweisungen oft die folgende Form haben:

```
variable = variable # constant
```

wobei '#' ein binärer Operator ist. Um also einen Integeroperanden um 2 zu inkrementieren, könnte der Programmierer schreiben:

```
nVariable = nVariable + 2;
```

Dies besagt, dass 2 zum Wert von `nVariable` hinzugefügt und das Ergebnis in `nVariable` gespeichert werden soll.

Es ist üblich, dieselbe Variable auf der linken und auf der rechten Seite der Zuweisung stehen zu haben.

0 Min.

Weil die gleiche Variable auf der linken und der rechten Seite des Gleichheitszeichens steht, haben sie entschieden, dem Zuweisungsoperator einen weiteren Operator hinzuzufügen. Alle binären Operatoren haben eine Zuweisungsversion. Somit kann die obige Anweisung wie folgt geschrieben werden:

```
nVariable += 2;
```

Noch mal, dies besagt, dass 2 zum Wert von `nVariable` hinzugefügt werden soll.

Anders als die Zuweisung selber, werden diese Zuweisungsoperatoren nicht allzu oft benutzt. In bestimmten Fällen kann Ihre Verwendung jedoch das Programm deutlich lesbarer machen.

Zusammenfassung

Die mathematischen Operatoren werden in C++-Programmen öfter verwendet als alle anderen Operatoren. Das ist wenig verwunderlich: C++-Programme konvertieren immer Temperaturen von Grad Celsius in Grad Fahrenheit und zurück und führen unzählige andere Operationen durch, die Addition, Subtraktion und Zählen erforderlich machen.

- Alle Ausrücke haben einen Wert und einen Typ.
- Die Ordnung der Auswertung innerhalb eines Ausdrucks wird normalerweise bestimmt durch den Vorrang der Operatoren. Diese Reihenfolge kann mit Hilfe von Klammern überschrieben werden.
- Für viele der am häufigsten verwendeten Ausdrücke stellt C++ Kurzformen bereit. Der geläufigste ist der `i++`-Operator anstelle von `i = i + 1`.

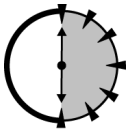
Selbsttest

1. Was ist der Wert von 9 % 4? (Siehe »Arithmetische Operatoren«)
2. Ist 9 % 4 ein Ausdruck? (Siehe »Ausdrücke«)
3. Wenn `n = 4`, was ist der Wert von `n + 10 / 2`? Warum ist er 9 und nicht 7? (Siehe »Vorrang von Operatoren«)
4. Wenn `n = 4`, was ist der Unterschied zwischen `++n` und `n++`? (Siehe »Unäre Operatoren«)
5. Wie würde man `n = n + 2` unter Verwendung des Operators += schreiben? (Siehe »Zuweisungsoperatoren«)

Logische Operationen

Checkliste

- ☑ Einfache logische Operatoren und Variablen einsetzen
- ☑ Mit binären Zahlen arbeiten
- ☑ Bitweise Operationen ausführen
- ☑ Logischer Zuweisungsanweisungen erzeugen



30 Min.

Vielleicht mit Ausnahme der Inkrement- und Dekrementoperatoren sind die mathematischen Operatoren von C++ geläufige Operatoren und kommen im Alltag vor. Im Vergleich dazu sind die logischen Operatoren von C++ unbekannt.

Es ist nicht so, dass sich die Menschen nicht mit logischen Operationen beschäftigen. Wenn Sie sich an den mechanischen Reifenwechsler in Sitzung 1 erinnern, ist die Fähigkeit, logische Berechnungen auszudrücken, wie »wenn der Reifen platt ist UND ich einen Schraubenschlüssel habe...« ein Muss. Die Menschen berechnen ständig AND und OR, sie sind nur nicht daran gewöhnt, das hinzuschreiben.

Logische Operatoren zerfallen in zwei Klassen. Der erste Typ, den ich einfache logische Operatoren nenne, sind Operatoren, die im alltäglichen Leben vorkommen. Der zweite Typ, die bitweisen logischen Operatoren, gibt es nur in der Welt des Computers. Ich werde erst die einfachen Operatoren vorstellen, bevor wir uns den bitweisen Operatoren zuwenden.

7.1 Einfache logische Operatoren

Die einfachen logischen Operatoren repräsentieren dieselbe Art logische Operation, die Sie in Ihrem alltäglichen Leben treffen: wie z.B. ob etwas wahr oder falsch ist oder der Vergleich zweier Dinge. Die einfachen logischen Operatoren sind in Tabelle 7-1 zu sehen.

Tabelle 7-1: Einfache logische Operatoren

Operator	Bedeutung
==	Gleichheit; wahr, wenn das Argument auf der linken Seite den gleichen Wert wie das Argument auf der rechten Seite hat
!=	Ungleichheit; Gegenteil von Gleichheit
>, <	größer als, kleiner als; wahr, wenn das Argument auf der linken Seite größer/kleiner als das Argument auf der rechten Seite ist
>=, < =	größer oder gleich, kleiner oder gleich; wahr, wenn entweder > oder == wahr sind / < oder == wahr sind
&&	AND; wahr, wenn beide Argumente auf der rechten und der linken Seite wahr sind
||	OR; wahr, wenn das linke Argument oder das rechte Argument wahr ist
!	NOT; wahr, wenn das Argument falsch ist

Die ersten sechs Einträge in Tabelle 7-1 sind die Vergleichsoperatoren. Der Gleichheitsoperator wird verwendet, um zwei Zahlen miteinander zu vergleichen. Z.B. ist das Folgende wahr (true), wenn der Wert der Variable `nVariable` gleich 0 ist:

```
nVariable == 0;
```

Verwechseln Sie nicht den Gleichheitsoperator == mit dem Zuweisungsoperator =. Das ist nicht nur ein oft gemachter Fehler, das ist auch ein Fehler, den der C++-Compiler nicht abfangen kann.

```
nVariable = 0; // Programmierer meinte nVariable == 0
```

Die Operatoren Größer-als (>) und Kleiner-als (<) sind ähnlich geläufig im alltäglichen Leben. Der folgende logische Vergleichsausdruck ist wahr:

```
int nVariable1 = 1;
int nVariable2 = 2;
nVariable1 <  nVariable2;
```

Die Operatoren Größer-oder-gleich (>=) und Kleiner-oder-gleich (< =) sind ähnlich, nur dass sie die Gleichheit mit einschließen, was die anderen Operatoren nicht tun.

Die Operatoren && (AND) und || (OR) sind ähnlich geläufig. Diese Operatoren werden typischerweise mit den anderen logischen Operatoren kombiniert:

```
// wahr wenn nV2 größer als nV1, aber kleiner als nV3
(nV1 <  nV2) && (nV2 <  nV3)
```

Seien Sie vorsichtig mit der Verwendung des Vergleichsoperators auf Gleitkommazahlen. Betrachten Sie das folgende Beispiel:

```
float fVariable1 = 10.0;
float fVariable2 = (10 / 3) * 3;
```

fVariable1 == fVariable2; // sind die beiden gleich?
Der Vergleich in obigem Beispiel liefert nicht notwendigerweise wahr (true). 10/3 ist gleich 3.333..., aber C++ kann nicht eine unendliche Anzahl von Nachkommastellen darstellen. Als Gleitkommazahl gespeichert, wird 10/3 etwa zu 3.333333. Wenn Sie diese Zahl mit 3 multiplizieren, bekommen Sie 9.999999 als Ergebnis, was nicht genau gleich 10 ist.
Genauso wie `(10.0 / 3) * 3` ***nicht genau 10.0 ist, kann das Ergebnis einer Gleitkommarechnung ein klein wenig daneben liegen. Solche kleinen Abweichungen werden Sie oder ich kaum beachten, aber der Computer. Gleichheit bedeutete exakte Gleichheit.***
Ein sicherer Vergleich sieht so aus:

```
float fDelta = fVariable1 - fVariable2;
fDelta <  0.01 && fDelta > -0.01;
```

Der Vergleich liefert wahr, wenn die Variablen `fVariable1` ***und*** `fVariable2` ***innerhalb eines Deltawertes nebeneinander liegen. (Der Begriff »Delta« ist mit Absicht vage – er soll einen akzeptablen Fehler darstellen.)***

Der nummerische Prozessor in ihrem PC wird große Anstrengungen unternehmen, um solche Rundungsfehler zu vermeiden – anhängig von der Situation, kann der nummerische Prozessor automatisch ein kleines Delta bereitstellen. Darauf sollten Sie sich aber nicht verlassen.

7.1.1 Kurze Schaltkreise und C++

Die Operatoren && und || führen das aus, was als »*Kurze-Schaltkreis-Evaluierung*« bezeichnet wird. Betrachten Sie das Folgende:

```
condition1 && condition2;
```

Wenn `condition1` nicht wahr ist, dann ist das Ergebnis nicht wahr, unabhängig davon, welchen Wert `condition2` hat (d.h. `condition2` kann wahr oder falsch sein ohne Einfluss auf das Ergebnis). In gleicher Weise ist

```
condition1 || condition2;
```

wahr, wenn `condition1` wahr ist, unabhängig davon, welchen Wert `condition2` hat.

Um Zeit zu sparen, wertet C++ `condition1` zuerst aus. C++ wertet `condition2` nicht aus, wenn `condition1` bereits FALSE ist im Falle von &&, bzw. TRUE im Falle von ||.

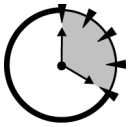
7.1.2 Logische Variablentypen

Wenn > und && Operatoren sind, dann muss ein Vergleich wie a > 10 ein Ausdruck sein. Natürlich muss das Ergebnis eines solchen Ausdrucks entweder TRUE oder FALSE sein.

Sie haben sicher bemerkt, dass in Sitzung 5 kein Boolscher Variablentyp erwähnt wurde. Es gibt also keinen Variablentyp, der nur die Werte TRUE und FALSE und sonst nichts annehmen kann. Was ist denn dann der Typ eines Ausdrucks von der Form a > 10?

C++ verwendet den Typ `int`, um Boolsche Werte zu speichern. Der Wert 0 steht für FALSE. Jeder andere Wert ungleich 0 bedeutet TRUE. Ein Ausdruck wie a > 10 wird ausgewertet zu 0 (FALSE) oder 1 (TRUE).

Microsoft Visual Basic verwendet auch Integerwerte zur Speicherung von TRUE und FALSE, Vergleichsoperatoren geben in Visual Basic jedoch 0 (FALSE) oder -1 (TRUE) zurück.



20 Min.

7.2 Binäre Zahlen

Die so genannten bitweisen logischen Operatoren arbeiten auf ihren Argumenten auf Bitebene. Um zu verstehen, wie sie arbeiten, lassen Sie uns zuerst binäre Zahlen ansehen, d.h. Zahlen, so wie Computer sie darstellen.

Die Zahlen, mit denen wir vertraut sind, werden als Dezimalzahlen bezeichnet, da sie auf der Zahl 10 basieren. Eine Zahl wie 123 bedeutet 1 x 100 plus 2 x 10 plus 3 x 1. Jede der Basiszahlen 100, 10 und 1 ist eine Potenz von 10.

$123_{10} = 1 * 100 + 2 * 10 + 3 * 1$

Die Verwendung von 10 als Basis für unser Zahlensystem kommt sehr wahrscheinlich daher, dass wir 10 Finger haben und diese ursprünglich als Zählhilfsmittel verwendet wurden. Wenn unser Zahlensystem von Hunden erfunden worden wäre, würde es sicher auf der Zahl 8 basieren (ein »Finger« jeder Tatze ist auf ihrer Rückseite und daher nicht sichtbar). Solch ein Oktalsystem würde ebensogut arbeiten:

$123_{10} = 173_8 = 1 + 64_{10} + 7 * 8_{10} + 3$

Die kleinen Ziffern 10 und 8 weisen auf das Zahlensystem hin, 10 für dezimal (Basis 10) und 8 für oktal (Basis 8). Ein Zahlensystem kann jede Basis verwenden, nur nicht 1.

Computer haben im Wesentlichen zwei Finger. Aufgrund Ihres Aufbaus bevorzugen Sie es, mit der Basis 2 zu rechnen. Die Zahl 123_{10} wird ausgedrückt als

$0*128 + 1*64 + 1*32 + 1*16 + 1*8 + 0*4 +1*2 + 1*1$

oder 01111011_2.

Es ist Konvention, binäre Zahlen entweder mit 4, 8, 16, 32 oder 64 Bits darzustellen, selbst wenn die führenden Bits null sind. Das hängt auch wieder mit dem internen Aufbau von Computern zusammen.

Weil der Begriff »Ziffer« sich auf ein Vielfaches von 10 bezieht, wird eine binäre Ziffer als Bit bezeichnet. Bit kommt von binary digit. Acht Bits bilden ein Byte.

Mit einer so kleinen Basis ist es nötig, viele Bits zur Darstellung großer Zahlen zu verwenden. Es ist unbequem, einen Ausdruck wie 01111011_2 für die Darstellung eines einfachen Wertes wie 123_{10} zu verwenden. Programmierer bevorzugen es, Zahlen als Einheiten von jeweils 4 Bits darzustellen.

Eine einzelne 4-Bit-Ziffer ist im Wesentlichen die Basis 16, weil 4 Bits jeden Wert zwischen 0 und 15 darstellen können. Die Basis 16 ist unter dem Namen Hexadezimalsystem bekannt. *Hexadezimal* wird oft mit *hex* abgekürzt.

Hexadezimalzahlen verwenden die gleichen Ziffern für die Zahlen von 0 bis 9. Für die Ziffern zwischen 9 und 16 verwenden die Hexadezimalzahlen die ersten sechs Buchstaben des Alphabets. A für 10, B für 11, usw. Es ist also 123_{10} gleich $7B_{16}$.

```
7 * 16 + B (d.h. 11) * 1 = 123
```

Weil Programmierer es bevorzugen, Zahlen in 4, 8, 32 oder 64 Bits auszudrücken, bevorzugen Sie in ähnlicher Weise eine Darstellung hexadezimaler Zahlen mit 1, 2, 4 oder 8 hexadezimalen Ziffern, selbst wenn die führenden Ziffern 0 sind.

Schließlich ist es unbequem, eine Hexadezimalzahl wie 7B mit Hilfe des Subscripts 16 auszudrücken, weil Terminals das nicht unterstützen. Selbst mit einem Textprogramm, wie ich es gerade benutze, ist es unbequem, den Zeichensatz auf Subscript umzuschalten und wieder zurück, nur um diese beiden Ziffern zu schreiben. Deshalb verwenden Programmierer die Konvention, dass eine Hexadezimalzahl mit "0x" beginnt (der Grund für eine solch merkwürdige Konvention geht auf die frühen Tage von C zurück). Dann wird 7B zu 0x7B. Mit dieser Konvention sind die Zahlen 0x123 und 123 voneinander unterscheidbar. (0x123 entspricht 291 dezimal.)

7.3 Bitweise logische Operationen

Alle Operatoren die bisher definiert wurden, können auf hexadezimale Zahlen genauso angewendet werden, wie sie auf Dezimalzahlen angewendet werden konnten. Der Grund, weshalb wir eine Multiplikation wie 0xC x 0xE nicht im Kopf ausführen können, liegt vielmehr darin, wie wir die Multiplikation in der Schule gelernt haben, als an irgendwelchen Beschränkungen der Operatoren.

Zusätzlich zu den mathematischen Operatoren gibt es eine Menge von Operationen, die auf Einzelbit-Operatoren basieren. Diese Basisoperationen sind nur für 1-Bit-Zahlen definiert.

7.3.1 Die Einzelbit-Operatoren

Die bitweisen Operatoren führen logische Operationen auf einzelnen Bits aus. Wenn Sie 0 als falsch (false) und 1 als wahr (true) annehmen (das muss nicht so sein, aber es ist eine übliche Konvention), dann können Sie für den bitweisen AND-Operator Dinge sagen wie die folgenden:

```
1 (true) AND 1 (true) ist 1 (true)
1 (true) AND 0 (false) ist 0 (false)
```

Und in gleicher Weise für den OR-Operator.

```
1 (true) OR 0 (false) is 1 (true)
0 (false) OR 0 (false) is 0 (false)
```

Geschrieben in Tabellenform sieht das wie in den Tabellen 7-2 und 7-3 aus.

Tabelle 7-2: Wahrheitstafel für den Operator AND

AND	1	0
1	1	0
0	0	0

Tabelle 7-3: Wahrheitstafel für den Operator OR

OR	1	0
1	1	1
0	1	0

In Tabelle 7-2 wird das Ergebnis von 1 AND 0 in Zeile 1 gezeigt (1 in Zeilenkopf) und Spalte 2 (0 im Spaltenkopf).

Ein anderer logischer Operator, der so im alltäglichen Leben nicht vorkommt, ist der Operator »oder sonst«, der üblicherweise mit XOR abgekürzt wird. XOR liefert wahr, wenn eines der Argumente wahr ist, aber nicht beide Argumente gleichzeitig wahr sind. XOR ist in Tabelle 7-4 dargestellt.

Tabelle 7-4: Wahrheitstabelle für den Operator XOR

XOR	1	0
1	0	1
0	1	0

Ausgerüstet mit diesen Einzelbit-Operatoren können wir uns den logischen Operatoren von C++ zuwenden.

7.3.2 Die bitweisen Operatoren

Die bitweisen Operatoren von C++ führen Bitoperationen auf jedem einzelnen Bit ihres Argumentes aus. Die einzelnen Operatoren sind in der Tabelle 7.5 zu sehen.

Tabelle 7-5: Die bitweisen Operatoren von C++

Operator	Funktion
~	NOT: invertiere jedes Bit von 0 nach 1 und von 1 nach 0
&	AND: verknüpfe jedes Bit der linken Seite mit dem entsprechenden Bit auf der rechten Seite durch »und«
|	OR
^	XOR

Der Operator NOT ist am einfachsten zu verstehen. NOT konvertiert 1 in 0 und 0 in 1. (D.h. 0 ist NOT 1 und 1 ist NOT 0.)

```
~0110₂ (0x6)
 1001₂ (0x9)
```

Der Operator NOT ist der einzige unäre bitweise logische Operator. Die folgende Rechnung demonstriert den &-Operator.

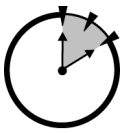
```
0110₂
&
0011₂
0010₂
```

Von links nach rechts: 0 AND 0 ist 0 (erstes Bit), 1 AND 0 ist 0 (zweites Bit), 1 AND 1 ist 1 (drittes Bit) und 0 AND 1 ist 0 (am wenigsten signifikantes Bit).

Die gleichen Berechnungen können auf Zahlen ausgeführt werden, die als Hexadezimalzahl dargestellt sind, indem sie erst in Binärzahlen verwandelt werden und dann das Ergebnis in Hexadezimaldarstellung konvertiert wird.

```
0x6          0110₂
 &    ->      &
0x3          0011₂
             0010₂     ->  0x2
```

Solche Hin- und Herkonvertierungen sind viel einfacher mit Hexadezimalzahlen auszuführen als mit Dezimalzahlen. Glauben Sie es oder nicht, mit ein wenig Erfahrung können Sie bitweise Operationen im Kopf ausführen.



10 Min.

7.3.3 Ein einfacher Test

Wir benötigen ein einfaches Programm, um Ihre Fähigkeiten, bitweise Operationen auszuführen, zu testen; erst auf Papier und dann im Kopf. Listing 7-1 ist ein Programm, das zwei Hexadezimalzahlen von der Tastatur erwartet, und das Ergebnis der Operatoren AND, OR und XOR ausgibt.

Listing 1: 7-1: Test der bitweisen Operatoren

```
// BitTest - Eingabe von zwei Hexadezimalzahlen
//           über die Tastatur und dann Ausgabe
//           der Ergebnisse von Anwendung der
//           Operatoren &, | and ^
#include < stdio.h>
#include < iostream.h>

int main(int nArg, char* nArgs[])
{
    // setze Ausgabeformat auf hexadecimal
    cout.setf(ios::hex);
```

```
    // Eingabe des ersten Argumentes
    int nArg1;
    cout < <  "Eingabe nArg1 hexadezimal (4 Ziffern):";
    cin  >> nArg1;

    int nArg2;
    cout < <  "Eingabe nArg2 hexadezimal (4 Ziffern):";
    cin  >> nArg2;

    cout < <  "nArg1 & nArg2 = 0x"
         < <  (nArg1 & nArg2) < <  "\n";
    cout < <  "nArg1 | nArg2 = 0x"
         < <  (nArg1 | nArg2) < <  "\n";
    cout < <  "nArg1 ^ nArg2 = 0x"
         < <  (nArg1 ^ nArg2) < <  "\n";

    return 0;
}
```

Ausgabe:

```
Eingabe nArg1 hexadezimal (4 Ziffern): 0x1234
Eingabe nArg2 hexadezimal (4 Ziffern): 0x00ff
nArg1 & nArg2 = 0x34
nArg1 | nArg2 = 0x12ff
nArg1 ^ nArg2 = 0x12cb
```

Die erste Anweisung, die `cout.setf(ios::hex);` lautet, setzt das Ausgabeformat von standardmäßig dezimal auf hexadezimal (im Augenblick müssen Sie mir glauben, dass das so funktioniert).

Der Rest des Programms ist einfach. Das Programm liest `nArg1` und `nArg2` von der Tastatur und gibt dann alle Kombinationen von bitweisen Berechnungen aus.

Die Ausgabe des Programms bei Eingabe von 0x1234 und 0x00ff sehen Sie oben am Ende des Listings.

Hexadezimalzahlen werden mit einem führenden 0x geschrieben.

7.3.4 Warum?

Der Sinn der meisten Operatoren ist klar. Niemand würde nach dem Sinn des Plus-Operators oder des Minus-Operators fragen. Der Sinn der Operatoren < und > ist klar. Für den Anfänger muss nicht klar sein, wann und warum man bitweise Operatoren verwendet.

Der Operator AND wird oft verwendet, um Information auszumaskieren. Nehmen Sie z.B. an, dass wir die am wenigsten signifikante Hexadezimalstelle aus einer Zahl mit vier Ziffern extrahieren möchten.

```
0x1234      0001 0010 0011 0100
  &        -> &
0x000F      0000 0000 0000 1111
            0000 0000 0000 0100  ->  0x0004
```

Eine andere Anwendung ist das Setzen und Auslesen einzelner Bits.

Nehmen Sie an, dass wir in einer Datenbank Informationen über Personen in einem einzelnen Byte pro Person speichern. Das signifikanteste Bit könnte z.B. gesetzt werden, wenn die Person männlich ist, das nächste Bit, wenn es ein Programmierer ist, das nächste, wenn die Person attraktiv ist, und das am wenigsten signifikante Bit, wenn die Person einen Hund hat.

Bit	Bedeutung
0	1 -> männlich
1	1 -> Programmierer
2	1 -> attraktiv
3	1 -> hat einen Hund

0 Min.

Dieses Byte würde für jede Person kodiert und zusammen mit dem Namen, Versicherungsnummer und allen weiteren legalen Informationen gespeichert.

Ein hässlicher männlicher Programmierer, der einen Hund besitzt, würde als 1101_2 kodiert. Um alle Einträge in der Datenbank zu testen, um nach nicht attraktiven Programmierern zu suchen, die keinen Hund haben, unabhängig vom Geschlecht, würden wir den folgenden Ausdruck verwenden:

```
value & 0x0110) == 0x0100
         *^^*        ^ -> 0 = nicht attraktiv
                    ^     1 = ist Programmierer
                     * -> nicht von Interesse
                     ^ -> von Interesse
```

In diesem Fall wird der Wert 0110 als Maske bezeichnet, weil er Bits ausmaskiert, die nicht von Interesse sind.

Zusammenfassung

Sie haben die mathematischen Operatoren aus Kapitel 6 bereits in der Schule gelernt. Sie haben dort sicherlich nicht die einfachen logischen Operatoren gelernt, sie kommen aber im alltäglichen Leben vor. Operatoren wie AND und OR sind ohne Erklärung verständlich. Da C++ keinen logischen Variablentyp hat, verwendet es 0 zur Darstellung von FALSE und alles andere zur Darstellung von TRUE.

Im Vergleich dazu sind die binären Operatoren etwas Neues. Diese Operatoren führen die gleichen Operationen AND und OR aus, aber auf jedem Bit separat.

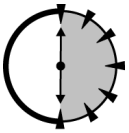
Selbsttest

1. Was ist der Unterschied zwischen den Operatoren && und &? (Siehe »Einfache logische Operatoren«)
2. Was ist der Wert von (1 && 5)? (Siehe »Einfache logische Operatoren«)
3. Was ist der Wert von 1 & 5? (Siehe »Bitweise logische Operatoren«)
4. Drücken Sie 215 als Summe von Zehnerpotenzen aus. (Siehe »Binäre Zahlen«)
5. Drücken Sie 215 als Summe von Zweierpotenzen aus. (Siehe »Binäre Zahlen«)
6. Was ist 215 in Hexadezimaldarstellung? (Siehe »Binäre Zahlen«)

Kommandos zur Flusskontrolle

Checkliste

- ☑ Kontrolle über den Programmfluss
- ☑ Wiederholte Ausführung einer Gruppe
- ☑ Vermeidung von »Endlosschleifen«



30 Min.

Die Programme, die bisher im Buch vorgekommen sind, waren sehr einfach. Jedes Programm hat eine Menge von Eingabewerten entgegengenommen, das Ergebnis ausgegeben und die Ausführung beendet. Das ist ähnlich wie bei unserem computerisierten Mechaniker, wenn wir ihn anweisen, wie eine Schraube zu lösen ist, ohne ihm die Möglichkeit zu geben, zur nächsten Schraube oder zum nächsten Reifen zu gehen.

Was in unseren Programmen fehlt, ist eine Form von Flusskontrolle. Wir haben bisher keine Möglichkeit, kleinere Tests zu machen, und können keine Entscheidungen basierend auf diesen Tests treffen.

Dieses Kapitel widmet sich den verschiedenen C++-Kommandos zur Flusskontrolle.

8.1 Das Verzweigungskommando

Die einfachste Form der Flusskontrolle ist die Verzweigung. Diese Instruktion ermöglicht es dem Computer, zu entscheiden, welcher Pfad durch C++-Instruktionen gewählt werden soll auf der Basis logischer Bedingungen. In C++ wird das Verzweigungskommando durch die `if`-Anweisung implementiert:

```
if (m > n)
{
    // Instruktionen, die ausgeführt werden,
    // wenn m größer als n ist
}
else
{
    // ... Instruktionen, die ausgeführt werden
    //     wenn nicht
}
```

Zuerst wird die Bedingung `m > n` ausgewertet. Wenn das Ergebnis wahr ist, wird die Kontrolle an die Anweisungen übergeben, die der öffnenden Klammer { folgen. Wenn `m` nicht größer ist als `n`, wird die Kontrolle an die Anweisungen übergeben, die der öffnenden Klammer unmittelbar nach dem `else` folgen.

Der `else`-Zweig ist optional. Wenn er nicht da ist, verhält sich C++ so, als wäre er da, aber leer.

Tatsächlich sind die Klammern optional, wenn nur eine Anweisung in einem Zweig ausgeführt werden soll; es ist aber so einfach, Fehler zu machen, die der Compiler nicht abfangen kann, wenn nicht die Klammern als Marker verwendet werden können. Es ist viel sicherer, immer Klammern zu verwenden. Wenn Ihre Freunde Sie verführen wollen, keine Klammern zu verwenden, sagen Sie einfach »NEIN«.

Das folgende Programm demonstriert die if-Anweisung:

```
// BranchDemo - Eingabe von zwei Zahlen. Gehe
//              einen Pfad des Programms entlang,
//              wenn das erste Argument größer ist
//              als das zweite, oder sonst den
//              anderen Pfad
#include <stdio.h>
#include <iostream.h>

int main(int nArg, char* pszArgs[])
{

    // Eingabe des ersten Argumentes ...
    int nArg1;
    cout << »Eingabe nArg1: «;
    cin  >> nArg1;

    // ... und des zweiten
    int nArg2;
    cout << »Eingabe nArg2: »;
    cin  >> nArg2;

    // entscheide nun, was zu tun ist:
    if (nArg1 > nArg2)
    {
        cout << »nArg1 größer als nArg2\n«;
    }
    else
    {
        cout << »nArg1 nicht größer nArg2\n«;
    }

    return 0;
}
```

Hier liest das Programm Integerzahlen von der Tastatur und verzweigt entsprechend. Das Programm erzeugt die folgende typische Ausgabe:

```
Eingabe nArg1: 10
Eingabe nArg1: 8
nArg1 größer als nArg2
```

8.2 Schleifenkommandos

Verzweigungskommandos ermöglichen Ihnen, den Programmfluss den einen oder den anderen Pfad hinunter zu leiten. Das ist das C++-Äquivalent dazu, den computerisierten Mechaniker entscheiden zu lassen, ob er einen Schraubenschlüssel oder einen Schraubendreher verwendet, abhängig von der Problemstellung. Das bringt uns aber noch nicht zu dem Punkt, an dem der Mechaniker den Schraubenschlüssel mehr als einmal drehen kann, mehr als eine Radmutter entfernen kann oder mehr als einen Reifen des Autos bearbeiten kann. Dafür brauchen wir Schleifenanweisungen.

8.2.1 Die while-Schleife

Die einfachste Form der Schleifenanweisung ist eine `while`-Schleife, die wie folgt aussieht:

```
while(condition)
{
    // ... wird wiederholt ausgeführt, solange
    //     condition erfüllt ist
}
```

Die Bedingung condition wird geprüft. Wenn sie wahr ist, dann werden die Anweisungen innerhalb der Klammern ausgeführt. Sobald die schließende Klammer angetroffen wird, wird die Kontrolle an den Anfang zurückgegeben, und der Prozess beginnt von vorne. Der Effekt ist, dass der C++-Code zwischen den Klammern so lange ausgeführt wird, wie die Bedingung wahr ist.

Die Bedingung wird nur am Anfang der Schleife überprüft. Selbst wenn die Bedingung in der Schleife nicht mehr erfüllt ist, wird die Kontrolle die Schleife nicht verlassen, bis sie wieder an den Anfang der Schleife kommt.

Wenn die Bedingung zum ersten Mal wahr ist, was wird sie dann später falsch machen? Betrachen Sie das folgende Programm.

```
// WhileDemo - Eingabe einer Schleifenanzahl.
//             Ausgabe einer Zeichenkette in jedem
//             der nArg Durchläufe.
#include <stdio.h>
#include <iostream.h>

int main(int nArg, char* pszArgs[])
{

    // Eingabe der Schleifenanzahl
    int nLoopCount;
     cout << »Eingabe Schleifenanzahl: «;
```

```
    cin  >> nLoopCount;

    // so viele Schleifen wie eingegeben
    while (nLoopCount > 0)
    {
        nLoopCount = nLoopCount - 1;
        cout << »Nur « << nLoopCount
             << » weitere Schleifendurchläufe\n«;
    }
    return 0;
}
```

WhileDemo beginnt mit der Abfrage einer Schleifenanzahl vom Benutzer, die in der Variablem `nLoopCount` gespeichert wird. Wenn dies getan ist, fährt das Programm mit dem Testen von nLoopCount fort. Wenn `nLoopCount` größer ist als null, dann wird `nLoopCount` um eins erniedrigt, und das Ergebnis wird auf dem Bildschirm ausgegeben. Das Programm geht dann an den Anfang der Schleife zurück, um zu testen, ob `nLoopCount` immer noch positiv ist.

Wenn das Programm WhileDemo ausgeführt wird, liefert es folgende Ausgabe:

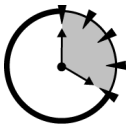
```
Eingabe Schleifenanzahl: 5
Nur 4 weitere Schleifendurchläufe
Nur 3 weitere Schleifendurchläufe
Nur 2 weitere Schleifendurchläufe
Nur 1 weitere Schleifendurchläufe
Nur 0 weitere Schleifendurchläufe
```

Als ich eine Schleifenanzahl von 5 eingegeben habe, hat das Programm die Schleife fünfmal durchlaufen und hat jedes Mal den heruntergezählten Wert ausgegeben.

Wenn der Benutzer eine negative Schleifenanzahl eingibt, überspringt das Programm die Schleife. Weil die Bedingung nie war ist, wird die Schleife nie betreten. Wenn der Benutzer eine sehr große Zahl eingibt, läuft das Programm sehr lange in der Schleife, bis es fertig ist.

Eine andere, selten benutzte Version der while-Schleife, bekannt unter dem Namen do-while, ist identisch mit der while-Schleife, außer dass die Bedingung am Ende der Schleife getestet wird.

```
do
{
    // ... das Innere der Schleife
} while (condition);
```



20 Min.

Verwendung von Autodekrement

Das Programm dekrementiert den Schleifenzähler unter Verwendung von Zuweisungs- und Subtraktionsanweisungen. Eine kompaktere Anweisung würde Autodekrement verwenden.

Die folgende Schleife ist ein wenig einfacher als die obige.

```
while (nLoopCount > 0)
    {
        nLoopCount--;
        cout << »Nur « << nLoopCount
             << » weitere Schleifendurchläufe\n«;
    }
```

Die Logik in dieser Version ist die gleiche wie im Original – der einzige Unterschied ist die Art und Weise, in der `nLoopCount` dekrementiert wird.

Weil Autodekrement sowohl das Argument dekrementiert als auch dessen Wert zurückliefert, kann der Dekrementoperator mit jeder der anderen Anweisungen verknüpft werden. Z.B. ist die folgende Version die bisher kürzeste Schleifenkonstruktion:

```
while (nLoopCount-- > 0)
{
    cout << »Nur « << nLoopCount
         << » weitere Schleifendurchläufe\n«;
}
```

Das ist die Version, die von den meisten C++-Programmierern verwendet wird.

Das ist die Stelle, wo der Unterschied zwischen Prädekrement und Postdekrement auftaucht.

Beide, `nLoopCount--` ***und*** `--nLoopCount` ***derementieren*** `nLoopCount`***; der erste gibt den Wert von*** `nLoopCount` ***vor dem Dekrementieren zurück, der zweite danach.***

Möchten Sie, dass die Schleife ausgeführt wird, wenn der Benutzer als Schleifenanzahl 1 eingibt? Wenn Sie die Prädekrement-Version verwenden, ist der Wert von `--nLoopCount` gleich 0 und der Rumpf der Schleife wird nie betreten. Mit der Postdekrement-Version ist der Wert von `nLoopCount--` gleich 1 und die Kontrolle wird an die Schleife übergeben.

Die gefürchtete Endlosschleife

Nehmen Sie an, dass der Programmierer einen Fehler gemacht hat, und vergessen hat, die Variable `nLoopCount` zu dekrementieren, wie im Beispiel unten zu sehen ist. Das Ergebnis ist ein Schleifenzähler, der seinen Wert nie ändert. Die Bedingung ist entweder immer falsch oder immer wahr.

```
while (nLoopCount > 0)
{
    cout << »Nur « << nLoopCount
         << » weitere Schleifendurchläufe\n«;
}
```

Weil der Wert von `nLoopCount` sich niemals verändert, läuft das Programm in einer endlosen Schleife. Ein Ausführungspfad, der unendlich oft ausgeführt wird, wird als Endlosschleife bezeichnet. Eine Endlosschleife tritt dann auf, wenn die Bedingung, die zum Schleifenabbruch führen sollte, nie erfüllt werden kann – im Allgemeinen durch einen Programmierfehler.

Es gibt viele Wege, eine Endlosschleife zu produzieren, die meisten sind viel komplizierter als das hier dargestellte Beispiel.

8.2.2 Die for-Schleife

Eine zweite Form der Schleife ist die `for`-Schleife, die folgendes Format besitzt:

```
for (initialization; conditional; increment)
{
    // ... Body der Schleife
}
```

Die Ausführung der `for`-Schleife beginnt mit der Initialisierung. Die Initialisierung hat diesen Namen bekommen, weil dort üblicherweise Zählvariablen initialisiert werden. Die Initialisierung wird nur ein einziges Mal ausgeführt, wenn die `for`-Schleife zum ersten Mal durchlaufen wird.

Die Ausführung wird bei der Bedingung fortgesetzt. Ähnlich zu der `while`-Schleife wird die `for`-Schleife so lange ausgeführt, wie die Bedingung wahr ist.

Nachdem die Ausführung des Code im Body der `for`-Schleife abgeschlossen wurde, wird die Kontrolle an die Inkrementanweisung übergeben. Danach wird die Bedingung erneut geprüft und der Prozess wiederholt. Die Inkrementklausel enthält normalerweise eine Autoinkrement- oder Autodekrementanweisung zum Updaten der Zählvariablen.

Alle drei Klauseln sind optional. Wenn die Initialisierung oder die Inkrementklausel fehlen, ignoriert C++ sie. Wenn die Bedingung fehlt, führt C++ die Schleife unendlich oft aus (oder bis sonst etwas den Kontrollfluss abbricht).

Die `for`-Schleife kann am Beispiel besser verstanden werden. Das folgende Programm ForDemo ist nichts anderes als das Programm WhileDemo, nur dass eine `for`-Schleife verwendet wird.

```
// ForDemo - Eingabe einer Schleifenanzahl.
//           Ausgabe einer Zeichenkette in jedem
//           Durchlauf.
#include <stdio.h>
#include <iostream.h>

int main(int nArg, char* pszArgs[])
{
    // Eingabe der Schleifenanzahl
    int nLoopCount;

    cout << »Eingabe Schleifenanzahl: «;
    cin  >> nLoopCount;

    // so viele Schleifen wie eingegeben
    for (int i = nLoopCount; i > 0; i--)
    {
        cout << »Durchlauf für « << i << » \n«;
    }
    return 0;
}
```

Diese Version durchläuft die gleichen Schleifen wie zuvor. Der Unterschied ist jedoch, dass nicht der Wert von `nLoopCount` verändert, sondern eine Zählvariable verwendet wird.

Die Kontrolle beginnt mit der Deklaration einer Variablen `i`, die mit 1 initialisiert wird. Die `for`-Schleife überprüft dann die Variable `i`, um sicherzustellen, dass sie kleiner oder gleich dem Wert von `nLoopCount` ist. Wenn dies der Fall ist, führt das Programm die Ausgabeanweisung aus, inkrementiert `i` und fährt mit der Ausführung der Schleife fort.

Die `for`-Schleife ist auch bequem, wenn Sie von 0 bis zu einer Schleifenanzahl zählen wollen, statt von einer Schleifenanzahl auf 1 herunterzuzählen. Dies wird durch eine kleine Änderung in der Implementierung erreicht:

```
// ForDemo - Eingabe einer Schleifenanzahl.
//           Ausgabe einer Zeichenkette in jedem
//           Durchlauf.
#include <stdio.h>
#include <iostream.h>

int main(int nArg, char* pszArgs[])
{
    // Eingabe der Schleifenanzahl
    int nLoopCount;
    cout << »Eingabe Schleifenanzahl: «;
    cin  >> nLoopCount;

    // so viele Schleifen wie eingegeben
    for (int i = 1; i <= nLoopCount; i++)
    {
        cout << »Durchlauf für « << i << » \n«;
    }
    return 0;
}
```

Anstatt mit der Schleifenanzahl zu beginnen, startet diese Version bei 1 und zählt hoch, bis der vom Benutzer eingegebene Wert erreicht wird.

Die Verwendung der Variable `i` für `for`-Schleifen ist historisch bedingt (aus den frühen Tagen der Programmiersprache FORTRAN). Das ist der Grund, weshalb diese Schleifenvariablen sich nicht an die Standardkonventionen der Namensbildung halten.

Wenn die Indexvariable innerhalb der Initialisierungsklausel deklariert wird, ist sie nur innerhalb der `for`-Schleife bekannt. C++-Programmierer sagen, dass der Gültigkeitsbereich (scope) der Variablen die `for`-Schleife ist. Im obigen Beispiel ist die Variable `i` für die `return`-Anweisung nicht zugreifbar, weil diese Anweisung nicht in der `for`-Schleife steht.

8.2.3 Spezielle Schleifenkontrolle

Es kann vorkommen, dass die Bedingung, die zum Abbruch der Schleife führen soll, nicht am Anfang noch am Ende der Schleife eintritt. Betrachten Sie das folgende Programm, das Zahlen summiert, die der Benutzer eingibt. Die Schleife bricht ab, wenn der Benutzer eine negative Zahl eingibt, d.h. die Schleife muss verlassen werden, bevor der negative Wert zu der Summe hinzugefügt wird.

Die Herausforderung dieses Problems besteht darin, dass das Programm die Schleife erst verlassen kann, wenn der Benutzer einen Wert eingegeben hat. Die Schleife muss aber verlassen werden, bevor der Wert zur Summe addiert wird.

Für diese Fälle definiert C++ das Kommando break. Wenn `break` angetroffen wird, wird die aktuelle Schleife sofort verlassen. D.h. die Kontrolle wird dann an die Anweisung übergeben, die unmittelbar auf die schließende Klammer folgt.

Das Format der `break`-Anweisung ist wie folgt:

```
while(condition) // break geht bei for genauso
{
    if (condition2)
    {
        break;   // Schleife verlassen
    }
}               // Kontrolle kommt hier her, wenn
                // Programm break antrifft
```

Ausgerüstet mit diesem neuen Kommando `break` sieht meine Lösung für das Additionsproblem wie das Programm BreakDemo aus:

```
// BreakDemo - Eingabe einer Reihe von Zahlen.
//             Bilde die Summe dieser Zahlen
//             bis der Benutzer eine negative
//             Zahl eingibt.
#include <stdio.h>
#include <iostream.h>

int main(int nArg, char* pszArgs[])
{

    // initialisiere Summe
    int nAccumulator = 0;
    cout << »Dieses Programms addiert «
         << »die eingegebenen Zahlen.\n«;
    cout << »Beenden Sie die Schleife mit «
         << »einer negativen Zahl.\n«;

    // »Endlosschleife«
    for(;;)
    {
        // hole eine weitere Zahl
        int nValue = 0;
        cout << »Nächste Zahl: «;
        cin  >> nValue;

        // wenn sie negativ ist ...
        if (nValue < 0)
        {
            // ... dann Abbruch
            break;
        }

        // ... sonst addiere sie zur Summe
        nAccumulator = nAccumulator + nValue;
    }
```

```
    // jetzt haben wir die Schleife verlassen
    // Ausgabe der Gesamtsumme
    cout << »\nDie Gesamtsumme ist «
         << nAccumulator
         << »\n«;

    return 0;
}
```

Nachdem dem Benutzer die Spielregeln erklärt wurden (Eingabe einer negativen Zahl zum Abbruch usw.), durchläuft das Programm eine Endlosschleife.

Eine `for`***-Schleife ohne Bedingung ist eine Endlosschleife.***

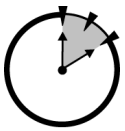
Diese Schleife ist nicht wirklich endlos, weil sie eine `break`***-Anweisung enthält. Trotzdem wird sie als Endlosschleife bezeichnet, da die Abbruchbedingung nicht im Kommando selber enthalten ist.***

Wenn das Programm BreakDemo erst einmal in der Schleife ist, bekommt es eine Zahl über die Tastatur. Erst wenn das Programm eine Zahl gelesen hat, kann es bestimmen, ob diese Zahl zum Abbruch der Schleife führt. Wenn die eingegebene Zahl negativ ist, wird die Kontrolle an `break` übergeben, was zum Abbruch der Schleife führt. Wenn die Zahl nicht negativ ist, wird die `break`-Anweisung übersprungen und die Kontrolle wird an den Ausdruck übergeben, der diese Zahl zur Summe addiert.

Wenn das Programm erst einmal die Schleife verlassen hat, gibt es die Gesamtsumme aus, und beendet die Ausführung. Hier ist die Ausgabe einer Beispielsitzung:

```
Dieses Programms addiert die eingegebenen Zahlen.
Beenden Sie die Schleife mit einer negativen Zahl.
Nächste Zahl: 1
Nächste Zahl: 2
Nächste Zahl: 3
Nächste Zahl: 4
Nächste Zahl: -1

Die Gesamtsumme ist 10
```

Wenn Sie eine Operation wiederholt auf einer Variablen ausführen, stellen Sie sicher, dass die Variable korrekt initialisiert wurde, bevor die Schleife betreten wird. In diesem Fall setzt das Programm `nAccumulator` ***auf Null, bevor die Schleife betreten wird, in der Werte*** `nValue` ***zu nAccumulator addiert werden.***



10 Min.

8.3 Geschachtelte Kontrollkommandos

Die drei bisherigen Programme in diesem Kapitel entsprechen den Anweisungen an den Mechaniker, wie er eine Radmutter lösen soll: den Schraubenschlüssel so lange drehen, bis die Radmutter herunterfällt. Wie funktioniert es, den Mechaniker zu instruieren, so lange Radmuttern zu entfernen, bis der Reifen herunterfällt? Dafür müssen wir geschachtelte Schleifen implementieren.

Eine Schleifenanweisung innerhalb einer anderen Schleifen wird als **geschachtelte Schleife** bezeichnet. Lassen Sie uns exemplarisch das Programm BreakDemo zu einem Programm umbauen, das eine beliebige Anzahl von Folgen summiert. In diesem Programm NestedDemo summiert die innere Schleife Werte auf, die von der Tastatur kommen, bis eine negative Zahl eingegeben wird. Die äußere Schleife läuft so lange, bis die Summe einer Sequenz gleich 0 ist.

```
// NestedDemo - Eingabe einer Reihe von Zahlen.
//              Bilde die Summe dieser Zahlen,
//              bis der Benutzer eine negative
//              Zahl eingibt. Setze den Prozess
//              fort, bis die Summe gleich 0 ist.
#include <stdio.h>
#include <iostream.h>

int main(int nArg, char* pszArgs[])
{
    // die äußere Schleife
    cout << »Dieses Programm summiert mehrere \n«
         << »Reihen von Zahlen. Beenden Sie jede «
         << »Reihe mit einer negativen Zahl.\n«
         << »Beenden Sie die Eingabe der Reihen\n«
         << »so, dass die Summe gleich 0 ist; \n«;
         << »geben Sie z.B. 1, -1 ein. \n«;
    // Summieren von Zahlenreihen
    int nAccumulator;
    do
    {
        // nächste Zahlenreihe
        nAccumulator = 0;
        cout << »\Nächste Zahlenreihe\n«;

        // Endlosschleife
        for(;;)
        {
            // hole nächste Zahl
            int nValue = 0;
            cout << »Nächste Zahl: «;
            cin  >> nValue;

            // wenn sie negativ ist ...
            if (nValue < 0)
            {
                // ... dann Abbruch
                break;
            }
```

```
            // ... sonst addiere sie zur Summe
            nAccumulator = nAccumulator + nValue;
        }

        // Ausgabe der Gesamtsumme
        cout << »\nDie Gesamtsumme ist «
            << nAccumulator
            << »\n«;

        // ...und beginne mit der nächsten Zahlen-
        // folge, wenn die Summe nicht null war.
    } while (nAccumulator != 0);
    cout << »Programm wird beendet.\n«;
    return 0;
}
```

8.4 Können wir switchen?

Eine letzte Kontrollanweisung ist nützlich, wenn wir eine endliche Anzahl von Fällen haben. Die `switch`-Anweisung ist wie eine zusammengesetzte `if`-Anweisung, in dem Sinne, dass sie eine Anzahl verschiedener Möglichkeiten enthält anstatt eines einzelnen Tests:

```
switch(expression)
{
    case c1:
        // gehe hierhin, wenn expression == c1
        break;
    case c2:
        // gehe hierhin, wenn expression == c2
        break;
    default:
        // ansonsten gehe hierhin
}
```

0 Min.

Der Wert von expression muss ein Integer sein (`int`, `long` oder `char`). Die case-Werte `c1`, `c2`, `c3` müssen konstant sein. Wenn die `switch`-Anweisung angetroffen wird, wird der Ausruck ausgewertet und verglichen mit den case-Konstanten. Die Kontrolle wird an die case-Konstante übergeben, die passt. Wenn keine passende case-Konstante gefunden wird, geht die Kontrolle an `default` über.

Die `break`***-Anweisungen sind nötig, um das*** `switch`***-Kommando zu verlassen. Ohne die*** `break`***-Anweisungen würde der Fluss von einem Fall zum nächsten fließen.***

Zusammenfassung

Die einfache `if`-Anweisung ermöglicht es dem Programmierer, den Programmfluss abhängig vom Wert eines Ausdrucks den einen oder den anderen Pfad entlang fließen zu lassen. Die Schleifenanweisungen fügen die Fähigkeit hinzu, Codeblöcke wiederholt auszuführen, so lange bis eine Bedingung falsch wird. Schließlich stellt die `break`-Anweisung einen Extralevel Kontrolle bereit, die einen Schleifenabbruch an jeder beliebigen Stelle gestattet.

- Die `if`-Anweisung wertet einen Ausdruck aus. Wenn der Ausdruck nicht 0 (d.h. er ist `true`) ist, wird die Kontrolle an den Block übergeben, der dem `if` unmittelbar folgt. Wenn nicht, wird die Kontrolle an den `else`-Zweig übergeben. Wenn es keinen `else`-Zweig gibt, geht die Kontrolle auf die Anweisung über, die der `if`-Anweisung folgt.
- Die Schleifenkommandos `while`, `do while` und `for` führen einen Codeblock so lange aus, bis eine Bedingung nicht mehr wahr ist.
- Die `break`-Anweisung ermöglicht den Abbruch von Schleifen an jeder beliebigen Stelle.

In Sitzung 9 werden wir uns mit Wegen beschäftigen, wie C++-Programme durch die Verwendung von Funktionen vereinfacht werden können.

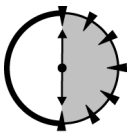
Selbsttest

1. Ist es ein Fehler, den `else`-Zweig eines `if`-Kommandos wegzulassen? Was passiert? (Siehe »Das Verzweigungskommando«)
2. Welches sind die drei Typen von Schleifen-Kommandos? (Siehe »Schleifenkommandos«)
3. Was ist eine Endlosschleife? (Siehe »Die gefürchtete Endlosschleife«)
4. Welches sind die drei »Klauseln«, die eine `for`-Schleife ausmachen? (Siehe »Die for-Schleife«)

Funktionen

Checkliste

- ☑ void-Funktionen schreiben
- ☑ Funktionen mit mehreren Argumenten schreiben
- ☑ Funktionen überladen
- ☑ Funktionstemplates erzeugen
- ☑ Speicherklassen von Variablen bestimmen



30 Min.

Einige der Beispielprogramme in Sitzung 8 sind schon ein wenig fortgeschrittener, aber wir müssen weiter C++ lernen. Programme mit mehreren Schachtelungsebenen können schwer zu überblicken sein. Fügen Sie die vielfältigen und komplizierten Verzweigungen hinzu, wie sie von realen Anwendungen erwartet werden, und die Programme sind überhaupt nicht mehr nachvollziehbar.

Glücklicherweise stellt C++ eine Möglichkeit bereit, selbstständige Blöcke von Code zu separieren, die als Funktionen bezeichnet werden. In dieser Sitzung werden wir sehen, wie C++ Funktionen deklariert, erzeugt und benutzt.

9.1 Code einer Sammelfunktion

Wie so vieles, werden auch Funktionen am besten an einem Beispiel verstanden. Dieser Abschnitt beginnt mit einem Beispielprogramm FunctionDemo, das das Programm NestedDemo aus Sitzung 8 vereinfacht, indem es für einen bestimmten Teil der Logik eine Funktion definiert. Dieser Abschnitt erklärt dann, wie diese Funktion definiert wird, und wie sie aufgerufen wird, indem der Beispielcode als Muster verwendet wird.

9.1.1 Sammelcode

Das Programm NestedDemo in Sitzung 8 enthält eine innere Schleife, die eine Folge von Zahlen summiert, und eine äußere Schleife, die das wiederholt ausführt, bis sich der Benutzer dazu entschließt, das Programm zu beenden. Von der Logik her könnten wir die innere Schleife separieren, d.i. der Teil des Programms, der eine Folge von Zahlen aufsummiert, von der äußeren Schleife, die den Prozess wiederholt.

Der folgende Beispielcode zeigt das vereinfachte Programm NestedDemo, in dem die Funktion `sumSequence( )` eingeführt wurde.

Den Namen von Funktionen folgen normalerweise unmittelbar ein Paar Klammern.

```
// FunctionDemo - demonstriert den Gebrauch von
//                Funktionen, indem die innere
//                Schleife von NestedDemo in eine
//                eigene Funktion gepackt wird

#include <stdio.h>
#include <iostream.h>

// sumSequence - addiere eine Reihe von Zahlen, die
//               der Benutzer über die Tastatur
//               eingibt, bis zur ersten negativen
//               Zahl - gib dann die Summe zurück
int sumSequence(void)
{
    // Endlosschleife
    int nAccumulator = 0;
    for(;;)
       {
         // hole weitere Zahl
          int nValue = 0;
          cout << »Nächste Zahl: »;
          cin  >> nValue;

        // wenn sie negativ ist ...
        if (nValue < 0)
        {
            // ...dann Schleife verlassen
            break;
        }

        // ... ansonsten Zahl zur Summe addieren
        nAccumulator = nAccumulator + nValue;
    }

    // Rückgabe der Summe
    return nAccumulator;
}

int main(int nArg, char* pszArgs[])
{                                   // Anfang Main
```

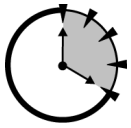
```
    cout << »Dieses Programm summiert mehrere \n«
         << »Reihen von Zahlen. Beenden Sie jede «
         << »Reihe mit einer negativen Zahl.\n«
         << »Beenden Sie die Eingabe der Reihen\n«
         << »so, dass die Summe gleich 0 ist; \n«;
         << »geben Sie z.B. 1, -1 ein. \n«;
    // summiere Folgen von Zahlen ...
    int nAccumulatedValue;
    do
    {
        // summiere eine Zahlenreihe, die über
        // die Tastatur eingegeben wird
        cout << »\Nächste Zahlenreihe\n«;
        nAccumulatedValue = sumSequence();

        // Ausgabe der Gesamtsumme
        cout << »\nDie Gesamtsumme ist »
             << nAccumulatedValue
             << »\n«;

    // ... bis die Summe gleich 0 ist.
    } while (nAccumulatedValue != 0);
    cout << »Programm wird beendet.\n«;
    return 0;
}                                   // Ende Main
```

Aufrufen der Funktion sumSequence( )

Lassen Sie uns erst einmal das Hauptprogramm ansehen, das sich zwischen den beiden Klammern befindet, die mit den beiden Kommentaren `Anfang Main` und `Ende Main` markiert sind. Dieses Codesegment sieht ähnlich aus wie Code, den wir schon einmal geschrieben haben. Die Zeile

```
nAccumulatedValue = sumSequene();
```

ruft die Funktion `sumSequence( )` auf und speichert ihren Rückgabewert in der Variablen `nAccumulatedValue`. Dieser Wert wird in den folgenden drei Programmzeilen nacheinander ausgegeben. Das Programm setzt die Schleife so lange fort, bis die Summe, die von der inneren Funktion zurückgegeben wird, 0 ist, was anzeigt, dass der Benutzer das Programm beenden möchte.



20 Min.

Definition der Funktion sumSequence( )

Der Codeblock, der in Zeile 13 beginnt und bis Zeile 38 geht, bildet die Funktion `sumSequence( )`.

Wenn das Hauptprogramm die Funktion `sumSequence( )` in Zeile 55 aufruft, geht die Kontrolle von diesem Aufruf an den Anfang dieser Funktion in Zeile 14 über. Die Programmausführung wird dort fortgesetzt.

Die Zeilen 16 bis 34 sind identisch mit denen der inneren Schleife im Programm NestedDemo. Nachdem das Programm diese Schleife verlassen hat, geht die Kontrolle auf die return-Anweisung in Zeile 37 über. Diese bewirkt einen Rücksprung zum Aufruf der Funktion in Zeile 55, zusammen mit dem Wert, der in `nAccumulatedValue` gespeichert wird. In Zeile 55 speichert das Programm den zurückgegebenen `int`-Wert in der lokalen Variable `nAccumulatedValue`, und setzt die Ausführung fort.

In diesem Fall ist der Aufruf von `sumSequence( )` ***ein Ausdruck, weil er einen Wert hat. Ein solcher Aufruf kann überall da verwendet werden, wo ein Ausdruck erwartet wird.***

9.2 Funktion

Das Programm FunctionDemo demonstriert die Definition und den Gebrauch einer einfachen Funktion.

Eine **Funktion** ist ein logisch separater C++-Codeblock. Die Funktion hat die folgende allgemeine Form:

```
<return type> name(<arguments to the function>)
{
    // ...
    return <expression>;
}
```

Die Argumente einer Funktion sind Werte, die an die Funktion als Eingaben übergeben werden können. Der Rückgabewert ist ein Wert, der von der Funktion zurückgegeben wird. Z.B. im Aufruf `square(10)` ist 10 das Argument der Funktion `square( )`, der Rückgabewert ist 100.

Sowohl Argumente als auch Rückgabewerte sind optional. Wenn eines von beiden fehlt, wird stattdessen das Schlüsselwort `void` verwendet. D.h., wenn eine Funktion eine `void`-Argumentliste hat, erwartet die Funktion keine Argumente, wenn sie aufgerufen wird. Wenn der Rückgabewert `void` ist, gibt die Funktion keinen Wert an den Aufrufenden zurück.

Im Beispielprogramm FunctionDemo ist der Name der Funktion `sumSequence( )`, der Typ des Rückgabewertes ist `int`, und die Funktion hat keine Argumente.

Der Default-Argumenttyp einer Funktion ist `void`***. Somit kann eine Funktion*** `int fn(void)` ***auch als*** `int fn( )` ***deklariert werden.***

9.2.1 Warum Funkionen?

Der Vorteil von Funktionen vor anderen C++-Kontrollkommandos ist, dass sie einen Teil des Codes für eine bestimmten Zweck vom Rest des Programms separieren. Durch diese Trennung kann sich der Programmierer auf eine Funktion konzentrieren, wenn er den Code schreibt.

Eine gute Funktion kann in einem Satz beschrieben werden, der nur wenige Unds und Oders enthält. Z.B. »die Funktion `sumSequence( )` ***summiert eine Folge von Zahlen, die vom Benutzer eingegeben werden.« Diese Definition ist kurz und klar.***

Die Funktionskonstruktion machte es mir möglich, im Wesentlichen zwei verschiedene Teile des Programms FunctionDemo zu schreiben. Ich habe mich darauf konzentriert, die Summe einer Folge von Zahlen zu erzeugen, als ich die Funktion `sumSequence( )` geschrieben habe. Ich habe mir an dieser Stelle keine Gedanken um irgendeinen anderen Code gemacht, der diese Funktion aufrufen könnte.

Genauso konnte ich mich beim Schreiben der Funktion `main( )` auf die Behandlung der von `sumSequence( )` zurückgegebenen Werte konzentrieren. Dabei musste ich nur wissen, was die Funktion zurückgibt, aber nicht, wie sie intern arbeitet.

9.2.2 Einfache Funktionen

Die einfache Funktion `sumSequence( )` gibt einen Integerwert zurück, den sie berechnet hat. Funktionen können jede Art eines regulären Variablentyps zurückgeben. Z.B. kann eine Funktion `double` oder `char` zurückgeben.

Wenn eine Funktion keinen Wert zurückgibt, dann wird der Rückgabewert dieser Funktion mit `void` angegeben.

Eine Funktion kann durch ihren Rückgabewert bezeichnet werden. Eine Funktion, die ein `int` zurückgibt, wird oft als Integer-Funktion bezeichnet. Eine Funktion, die keinen Wert zurückgibt, wird als void-Funktion bezeichnet.

Z.B. führt die folgende void-Funktion eine Operation durch, gibt aber keinen Wert zurück.

```
void echoSquare()
{
    int nValue;
    cout << »Wert:«;
    cin >> nValue;
    cout << »\nQuadrat: » << nValue * nValue << »\n«;
    return;
}
```

Die Kontrolle beginnt bei der öffnenden Klammer und wird fortgesetzt bis zur `return`-Anweisung. Die `return`-Anweisung in einer void-Funktion darf keinen Rückgabewert enthalten.

Die `return`-Anweisung in einer `void`-Funktion ist optional. Wenn sie nicht da ist, wird die Kontrolle an die aufrufende Funktion zurückgegeben, wenn die schließende Klammer angetroffen wird.

9.2.3 Funktionen mit Argumenten

Einfache Funktionen haben nur einen begrenzten Nutzen, da die Kommunikation dieser Funktionen durch ihren Rückgabewert wie eine Einbahnstraße ist. Kommunikation in beiden Richtungen ist aber besser; diese wird durch Argumente erreicht. Ein *Funktionsargument* ist eine Variable, deren Wert an die Funktion bei ihrem Aufruf übergeben wird.

Beispielfunktion mit Argumenten

Das folgende Beispiel definiert und benutzt eine Funktion `square( )`, die das Quadrat einer double-Gleitkommazahl zurückgibt, die ihr übergeben wurde:

```
// SquareDemo - demonstriert den Gebrauch von
//              Funktionen mit Argumenten

#include <stdio.h>
#include <iostream.h>
// square - gibt das Quadrat ihres Argumentes
//          dVar zurück.
double square(double dVar)
{
    return dVar * dVar;
}

int sumSequence(void)
{
    // Endlosschleife
    int nAccumulator = 0;
    for(;;)
    {
        // hole weitere Zahl
         doube dValue = 0;
         cout << »Nächste Zahl: »;
         cin  >> dValue;

        // wenn sie negativ ist ...
        if (dValue < 0)
        {
            // ... dann Schleife verlassen
            break;
        }

        // ... ansonsten berechne Quadrat
        int  nValue = (int)square(dValue);
        nAccumulator = nAccumulator + nValue;
    }

    // Rückgabe der Summe
    return nAccumulator;
}

int main(int nArg, char* pszArgs[])
{                                    // Anfang Main
    cout << »Dieses Programm summiert die Quadrate\n«
         << »von Zahlenreihen. Beenden Sie jede\n«
         << »Reihe mit einer negativen Zahl.\n«
         << »Beenden Sie die Eingabe der Reihen\n«
         << »so, dass die Summe gleich 0 ist; \n«;

    // Summiere Folgen von Zahlen ...
    int nAccumulatedValue;
    do
    {
```

```
        // Summiere Quadrate einer Zahlenfolge,
        // die über die Tastatur eingegeben wird
        cout << »\Nächste Zahlenreihe\n«;
        nAccumulatedValue = sumSequence();

        // Ausgabe der Gesamtsumme
        cout << »\nDie Gesamtsumme ist «
             << nAccumulatedValue
             << »\n«;

    // ... bis die Summe gleich 0 ist.
    } while (nAccumulatedValue != 0);
    cout << »Programm wird beendet.\n«;
    return 0;
}                                   // Ende Main
```

Das ist das gleiche Programm wie FunctionDemo, außer dass SquareDemo die Quadrate der eingegebenen Werte summiert.

Der Wert von `dValue` wird an die Funktion `square( )` übergeben in der Zeile

```
int nValue = (int)square(dValue);
```

innerhalb der Funktion `sumSequence( )`. Die Funktion `square( )` multipliziert den ihr in Zeile 12 übergebenen Wert mit sich selber und gibt das Ergebnis zurück. Das Ergebnis wird in der Variable `dSquare` gespeichert, die in Zeile 33 zu der Summe addiert wird.

Funktionen mit mehreren Argumenten

Funktionen können mehrere Argumente haben, die durch Kommata getrennt werden. Die folgende Funktion gibt das Produkt ihrer beiden Argumente zurück:

```
int product(int nArg1, int nArg2)
{
    return nArg1 * nArg2;
}
```

Wertkonvertierung (Casten)

Zeile 32/33 des Programms SquareDemo enthält einen Operator, den wir vorher noch nicht gesehen haben.

```
nAccumulator = nAccumulator + (int)dValue;
```

Das `(int)` vor `dValue` zeigt an, dass der Programmierer möchte, dass der Wert von `dValue` von seinem aktuellen Typ, in diesem Fall `double`, nach `int` konvertiert wird.

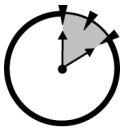
Ein solcher `Cast` ist eine explizite Konvertierung von einem Typ in einen anderen. Jeder nummerische Typ kann in jeden anderen nummerischen Typ konvertiert werden. Ohne diesen Cast hätte C++ diese Typkonvertierung selber vorgenommen, jedoch ohne eine Warnung ausgegeben.

Die Funktion main( )

Es sollte klar sein, dass `main( )` nichts anderes ist als eine Funktion, wenn auch eine Funktion mit merkwürdigen Argumenten.

Wenn ein Programm erzeugt wird, fügt C++ Code hinzu, der ausgeführt wird, bevor Ihr Programm überhaupt startet. Dieser Code initialisiert die Umgebung, in der Ihr Programm operiert. Z.B. öffnet dieser Code die beiden I/O-Objekte `cin` und `cout`.

Wenn die Umgebung erst einmal eingerichtet ist, ruft dieser C++-Code die Funktion `main( )` auf, womit die Ausführung Ihres Codes beginnt. Wenn Ihr Programm beendet ist, wird `main( )` wieder verlassen. Dies ermöglicht dem C++-Code, einige Dinge aufzuräumen, bevor die Kontrolle wieder an das Betriebssystem zurückgegeben wird, das dann das Programm löscht.



10 Min.

9.2.4 Mehrere Funktionen mit gleichem Namen

Zwei Funktionen in einem Programm dürfen nicht den gleichen Namen haben, sonst hat C++ keine Möglichkeit, sie zu unterscheiden, wenn sie aufgerufen werden sollen. In C++ jedoch gehören zum Namen einer Funktion die Anzahl und die Typen der Argumente. Die folgenden Funktionen sind daher voneinander verschieden:

```
void someFunction(void)
{
    // ... führe eine Funktion aus
}
void someFunction(int n)
{
    // ... führe eine andere Funktion aus
}
void someFunction(double d)
{
    // ... führe eine noch andere Funktion aus
}
void someFunction(int n1, int n2)
{
    // ... tue etwas ganz anderes
}
```

`someFunction( )` ist eine Abkürzung für alle vier Funktionen, in gleicher Weise, wie Stephen eine Kurzform meines Namens ist. Ob ich nun die Kurzform zur Beschreibung der Funktion wähle, so kennt C++ doch auch die Funktionen `someFunction(void)`, `someFunction(int)`, `someFunction(double)` und `someFunction(int, int)`.

`void` ***als Argumenttyp ist optional;*** `someFunction(void)` ***und*** `someFunction( )` ***sind die gleiche Funktion.***

Eine typische Anwendung könnte wie folgt aussehen:

```
int nVariable1, nVariable2;    // äquivalent zu
                              // int Variable1;
                              // int Variable2;
double dVariable;

// Funktionen unterscheiden sich durch ihre
// Argumenttypen (aufgerufene Funktion im Kommentar)
someFunction();          // someFunction(void)
someFunction(nVariable1);// someFunction(int)
someFunction(dVariable); // someFunction(double)
someFunction(nVariable1, nVariable2);
                        // someFunction(int, int)

// das funktioniert auch mit Konstanten
someFunction(1);         // someFunction(int)
someFunction(1.0);       // someFunction(double)
someFunction(1, 2);      // someFunction(int, int)
```

In jedem der Fälle stimmt der Typ der Argumente mit dem vollen Namen der drei Funktionen überein.

Der Rückgabetyp ist nicht Teil des Funktionsnamens. D.h., die beiden folgenden Funktionen haben den gleichen Namen und können daher nicht im gleichen Programm vorkommen:

```
int someFunction(int n);    // vollständiger Name der
double someFunction(int n); // beiden Funktionen
                            // ist someFunction(int)
```

9.3 Funktionsprototypen

In den bisherigen Beispielprogrammen wurden die Funktionen `sumSequence( )` und `square( )` beide in Codesegmenten definiert, die vor den Aufrufen der Funktionen standen. Das muss nicht immer so sein: Eine Funktion kann an beliebiger Stelle in einem Modul definiert werden. Ein *Modul* ist ein anderer Name für eine C++-Quelldatei.

Trotzdem muss jemand `main( )` den vollständigen Namen der Funktion mitteilen, bevor sie aufgerufen werden kann. Betrachten Sie den folgenden Codeschnipsel:

```
int main(int argc, char* pArgs[])
{
    someFunc(1, 2);
}
int someFunc(double dArg1, int nArg2)
{
    // ... tue etwas
}
```

Der Aufruf von `someFunc( )` von `main( )` aus kennt nicht den vollen Namen der Funktion. Von den Argumenten her könnte man vermuten, dass der Name `someFunc(int, int)` und der Rückgabetyp `void` ist. Wie sie sehen können, ist das falsch.

Was benötigt wird, ist ein Weg, um `main( )` den vollständigen Namen der Funktion `someFunc( )` mitzuteilen, bevor er benutzt wird. Was gebraucht wird, ist eine Funktionsdeklaration. Eine *Funktionsdeklaration*, oder ein *Prototyp*, sieht so aus wie die Funktion, nur ohne Body. In Gebrauch sieht ein Prototyp wie folgt aus:

```
int someFunc(double, int); // Prototyp
int main(int argc, char* pArgs[])
{
    someFunc(1, 2);
}
int someFunc(double dArg1, int nArg2)
{
    // ... tue etwas
}
```

Der Aufruf in `main( )` weiß nun, dass die 1 erst nach `double` gecastet werden muss, bevor die Funktion aufgerufen wird. Weiterhin weiß `main( )`, dass die Funktion `someFunc( )` ein `int` zurückgibt; dieser Rückgabewert wird hier jedoch ignoriert.

C++ erlaubt es dem Programmierer, Rückgabewerte zu ignorieren.

9.4 Verschiedene Speichertypen

Es gibt drei verschiedene Orte, an denen Funktionsvariablen gespeichert werden können. Variablen, die innerhalb einer Funktion deklariert werden, sind lokal. Im folgenden Beispiel ist die Variable `nLocal` für die Funktion `fn( )` lokal:

```
int nGlobal;
void fn()
{
    int nLocal;
    static int nStatic;
}
```

0 Min.

Die Variable `nLocal` existiert nicht, bis die Funktion `fn( )` aufgerufen wird. Außerdem hat nur die Funktion `fn( )` Zugriff auf `nLocal` – andere Funktionen können nicht in die Funktion »eindringen« und auf die Variable zugreifen.

Im Vergleich dazu existiert die Variable `nGlobal` so lange, wie das Programm läuft. Alle Funktionen haben jederzeit Zugriff auf `nGlobal`.

Die statische Variable `nStatic` ist eine Mischung aus einer lokalen und einer globalen Variable. Die Variable `nStatic` wird erzeugt, wenn die Deklaration erstmalig bei der Ausführung angetroffen wird (ungefähr dann, wenn die Funktion `fn( )` aufgerufen wird). Zusätzlich ist `nStatic` nur innerhalb von `fn( )` zugreifbar. Anders als `nLocal` existiert `nStatic` auch dann noch, wenn das Programm `fn( )` bereits verlassen hat. Wenn `fn( )` der Variablen `nStatic` einen Wert zuweist, bleibt dieser Wert erhalten, d.h. er steht auch im nächsten Aufruf der Funktion wieder zur Verfügung.

Es gibt einen vierten Variablentyp, `auto`***, der aber heute die gleiche Bedeutung hat wie*** `local`***.***

Zusammenfassung

Sie sollten jetzt eine Vorstellung davon haben, wie komplex Programme werden können, und wie kleine Funktionen die Programmlogik vereinfachen können. Wohlgeformte Funktionen sind klein, haben im Idealfall weniger als 50 Zeilen, und haben weniger als 7 `if`- oder Schleifen-Kommandos. Solche Funktionen sind besser zu verstehen, und damit einfacher zu schreiben und zu debuggen. Und ist das nicht das Ziel?

- C++-Funktionen sind das Mittel, um den Code in handliche Teile zu zerlegen.
- Funktionen können beliebig viele Argumente haben, über die Werte beim Aufruf der Funktion übergeben werden.
- Funktionen können einen einzelnen Wert an den Aufrufenden zurückgeben.
- Funktionsnamen können überladen werden, wenn sie durch die Anzahl und die Typen der Argumente unterscheidbar bleiben.

Es ist sehr schön, Ihr Programm mit mehreren Ausdrücken in unterschiedlichen Variablen und auf mehrere Funktionen verteilt auszudrücken, aber das bringt alles nichts, wenn Sie das Programm nicht zum Laufen bekommen. In Sitzung 10 werden Sie die elementaren Techniken kennenlernen, um Fehler in Ihren Programmen zu finden.

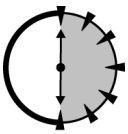
Selbsttest

1. Wie rufen Sie eine Funktion auf? (Siehe »Aufrufen der Funktion `sumSequence( )`«)
2. Was bedeutet der Rückgabewert `void`? (Siehe »Funktion«)
3. Warum sollte man Funktionen schreiben? (Siehe »Warum Funktionen«)
4. Was ist der Unterschied zwischen einer lokalen und einer globalen Variable? (Siehe »Verschiedene Speichertypen«)

Debuggen

Checkliste

- ☑ Fehlertypen unterscheiden
- ☑ Die »Crash-Meldungen« der C++-Umgebung verstehen
- ☑ Die Ausgabetechnik des Debuggens beherrschen
- ☑ Debuggen mit Visual C++ und GNU C++



30 Min.

Sie werden sicher festgestellt haben, dass die Programme, die Sie als Teil der Übungen in früheren Kapiteln geschrieben haben, nicht beim ersten Mal funktioniert haben. In der Tat habe ich selten, wenn überhaupt schon einmal, ein nicht-triviales C++-Programm geschrieben, das nicht irgendeinen Fehler enthielt, als ich es auszuführen versuchte.

Ein Programm, das auf Anhieb funktioniert, wenn Sie es ausprobieren, wird auch Goldstern-Programm *(gold-star program*) genannt.

10.1 Fehlertypen

Es gibt zwei Typen von Fehlern. Die Fehler, die der Compiler abfangen kann, sind sehr einfach zu beheben. Der Compiler wird Sie im Allgemeinen zu dem Fehler hinführen. Manchmal ist vielleicht die Beschreibung des Fehlers nicht korrekt – es ist leicht, einen Compiler zu verwirren – aber wenn Sie Ihr C++-Paket richtig kennen, ist es nicht schwierig, die Fehler zu beheben.

Eine zweite Sorte von Fehlern umfasst die Fehler, die der Compiler nicht finden kann. Diese Fehler werden erst dann sichtbar, wenn Sie das Programm ausführen. Fehler, die erst beim Ausführen des Programms sichtbar werden, werden als *Laufzeitfehler* bezeichnet. Fehler, die der Compiler finden kann, werden als *Compilezeitfehler* bezeichnet.

Laufzeitfehler sind viel schwieriger zu finden, weil Sie keinen Anhaltspunkt dafür haben, was falsch gelaufen ist, außer Fehlerausgaben, die Ihr Programm vielleicht generiert hat.

Es gibt zwei verschiedene Techniken, um Bugs zu finden. Sie können Ausgabeanweisungen an wichtigen Stellen im Programm einbauen und das Programm neu generieren. Sie können eine Vorstellung davon bekommen, was falsch gelaufen ist, wenn diese Ausgabeanweisungen ausgeführt

werden. Ein mächtigerer Zugang ist die Verwendung eines separaten Programms, das als Debugger bezeichnet wird. Ein Debugger gibt Ihnen die Möglichkeit, Ihr Programm bei der Ausführung zu kontrollieren. In dieser Sitzung behandeln wir den Zugang über Ausgabeanweisungen. In Sitzung 16 lernen Sie den Umgang mit den Debuggern von Visual C++ und GNU C++.

10.2 Die Technik der Ausgabeanweisungen

Der Ansatz, zum Debuggen Ausgabeanweisungen in den Code einzufügen, wird als Technik der Ausgabeanweisungen bezeichnet.

Diese Technik wird oft als Schreibanweisungsansatz bezeichnet. Diese Bezeichnung geht auf die frühen Tage der Programme zurück, die in FORTRAN geschrieben wurden. Ausgaben werden in FORTRAN durch `WRITE`***-Anweisungen ausgeführt.***

Um zu sehen, wie das funktionieren könnte, lassen Sie uns den Fehler in folgendem fehlerhaften Programm beheben:

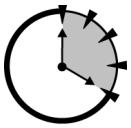
```
// ErrorProgram - dieses Programm berechnet den
//                Mittelwert von Zahlen - es enthält
//                jedoch einen Bug
#include <stdio.h>
#include <iostream.h>

int main(int argc, char* pszArgs[])
{
    cout << »Dieses Programm muss abstürzen!\n«;

    // summiere Zahlen, bis der Benutzer eine
    // negative Zahl eingibt, dann gib den
    // Mittelwert aus
    int nSum;
    for (int nNums = 0; ;)
    {
        // eine weitere Zahl
        int nValue;
        cout << »\nNächste Zahl:«;
        cin  >> nValue;

        // wenn sie negativ ist ...
        if (nValue < 0)
        {
            // ... dann gib Mittelwert aus
            cout << »\nMittelwert ist: »
                 << nSum/nNums
                 << »\n«;
            break;
        }
```

```
        // nicht negativ, addierte zur Summe
        nSum += nValue;
    }
    return 0;
}
```

Nachdem ich das Programm eingegeben habe, erzeuge ich das Programm, um die ausführbare Datei ErrorProgram.exe zu generieren. Ungeduldig lenke ich den Windows-Explorer auf den Ordner, der das Programm enthält, und führe voll Vertrauen einen Doppelklick auf ErrorProgram aus, um das Programm laufen zu lassen. Ich geben die Werte 1, 2 und 3 ein, gefolgt von -1 um die Eingabe abzuschließen. Aber anstatt den erwarteten Wert 2 auszugeben, beendet sich das Programm mit der nicht sehr freundlichen Fehlermeldung in Abbildung 10.1 und ohne jegliche Ausgabe.

Abbildung 10.1: Die erste Version von ErrorProgram bricht plötzlich ab, ohne eine Ausgabe zu erzeugen.

Führen Sie ErrorProgramm noch nicht in der C++-Umgebung aus.



20 Min.

10.3 Abfangen von Bug Nr. 1

Die Fehlermeldung in Abbildung 10.1 zeigt den allgegenwärtigen Fehlertext »Dieses Programm wurde aufgrund eines ungültigen Vorgangs geschlossen ...«. Selbst von den Informationen über die Registerinhalte haben Sie nichts, womit Sie das Debuggen beginnen könnten.

Tatsächlich ist doch ein klein wenig Information enthalten: Oben in der Fehlermeldung steht »ERRORPROGRAM verursachte einen Teilungsfehler«. Das würde normalerweise nicht viel helfen, hier jedoch schon, weil nur eine einzige Division im Programm vorkommt. Aber lassen Sie uns diese Meldung aus diesem Grund ignorieren.

In der Hoffnung, mehr Informationen über ErrorProgramm innerhalb der C++-Umgebung zu bekommen, kehre ich dahin zurück. Was dann passiert, ist bei Visual C++ und GNU C++ etwas verschieden.

10.3.1 Visual C++

Ich gebe dieselben Werte 1, 2, 3 und -1 ein, genauso wie eben, als das Programm einen Crash erlebte.

Eines der ersten Dinge, die Sie tun sollten, wenn Sie ein Problem finden wollen ist, eine Menge von Operationen zu finden, die Ihr Programm fehlerhaft verlaufen lassen. Indem Sie das Problem reproduzierbar machen, können Sie den Fehler nicht nur fürs Debuggen immer wieder neu erzeugen, sondern Sie wissen auch, wann das Problem nicht mehr auftritt.

Visual C++ erzeugt eine Ausgabe wie in Abbildung 10.2 zu sehen ist. Das Fenster zeigt an, dass das Programm fehlerhaft beendet wurde, weil durch null geteilt wurde.

Abbildung 10.2: Die Fehlermeldung von Visual C++ ist nur wenig besser als die Windowsfehlermeldungen.

Diese Fehlermeldung ist nur ein bisschen besser als die Fehlermeldung in Abbildung 10.1. Wenn ich jedoch OK klicke, zeigt Visual C++ das Programm ErrorProgram mit einem gelben Pfeil, der auf die Division zeigt, wie in Abbildung 10.3 zu sehen ist. Das ist die C++-Zeile, in der der Fehler aufgetreten ist.

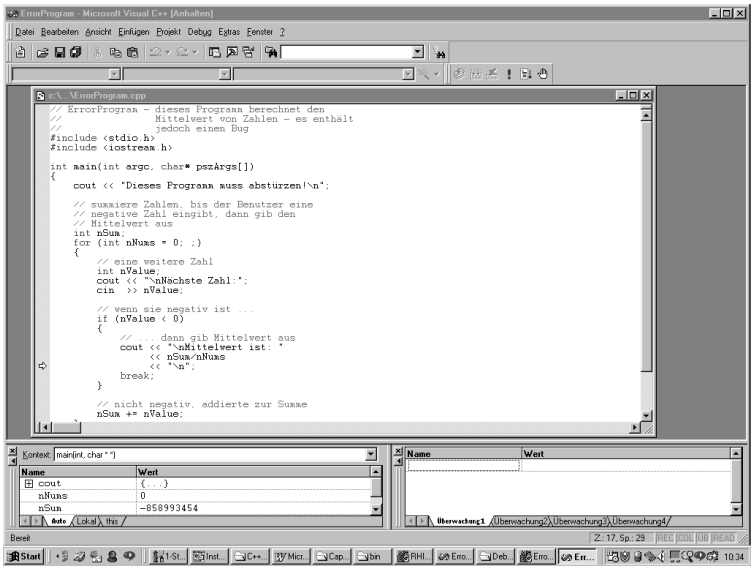


Abbildung 10.3: Visual C++ zeigt klar auf die Division durch nNums hin.

Jetzt weiß ich, dass, als die Division durchgeführt wurde, `nNums` gleich 0 gewesen sein muss. (Ansonsten kann es dort keinen Fehler durch Teilen durch null geben.) Ich kann sehen, wo `nNums` mit 0 initialisiert wird, aber wo wird es inkrementiert? Es wird nicht inkrementiert, und das ist der Fehler. Natürlich soll `nNums` in der Inkrement-Klausel der `for`-Schleife inkrementiert werden.

Um den Fehler zu beheben, ersetze ich die `for`-Schleife wie folgt:

```
for(int nNums = 0;  ; nNums++);
```

10.3.2 GNU C++

Die Fehlersuche verläuft in der GNU C++-Umgebung ähnlich. Ich lasse das Programm innerhalb von `rhide` laufen. Als Antwort auf meine Eingabe 1, 2, 3 und -1, terminiert ErrorProgram mit der Fehlermeldung, die in Abbildung 10.4 zu sehen ist. Anstelle des normalen Exit-Codes von 0x00 (0 als Dezimalzahl) sehe ich den Fehlercode von 0xff (255 dezimal). Das sagt mir nicht mehr, als dass ein Fehler beim Ausführen des Programms aufgetreten ist (so viel wusste ich auch vorher schon).

Abbildung 10.4: Der Fehlercode von ErrorProgram in rhide ist 0xff.

Nach dem Klicken auf OK öffnet `rhide` ein kleines Fenster am unteren Rand des Displays. Dieses Fenster, das in Abbildung 10.5 zu sehen ist, teilt mir mit, dass der Fehler im Programm ErrorProgram.cpp(28) in der Funktion `main( )` aufgetreten ist. Diese etwas kryptische Fehlermeldung zeigt an, dass der Fehler in Zeile 28 von ErrorProgram.cpp aufgetreten ist und dass sich diese Zeile innerhalb der Funktion `main( )` befindet (Letzteres hätte ich auch durch einen Blick auf das Listing herausfinden können, aber es ist trotzdem eine schöne Information).

Von hier aus kann ich das Problem auf die gleiche Art und Weise lösen, wie im Fall von Visual C++.

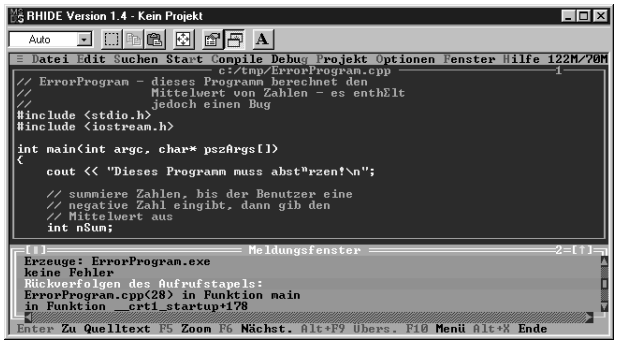


Abbildung 10.5: Die Fehlermeldung von rhide ist genauso informativ wie die von Visual C++, nur noch kryptischer.

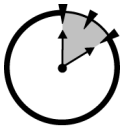
Wie kann C++ eine Fehlermeldung an den Sourcecode binden?

Die Information, die ich erhalten haben, als ich das Programm direkt von Windows bzw. von einem MS-DOS-Fenster aus aufgerufen habe, waren nicht sehr aussagekräftig. Beide, Visual C++ und `rhide`, waren in der Lage, mich an die Stelle im Programm zu führen, in der der Fehler aufgetreten ist. Wie machen sie das?

C++ hat zwei Modi, ein Programm zu erzeugen. Defaultmäßig wird eine C++ im so genannten Debug-Modus erzeugt. Im Debug-Modus fügt C++ Informationen über Zeilennummern hinzu, die Zeilen im C++-Code auf Maschinenanweisungen abbilden. Diese Abbildung kann beispielsweise besagen, dass Zeile 200 des Maschinencodes von der Zeile 16 im C++-Quellcode erzeugt wurde.

Als der Fehler Division-durch-null aufgetreten ist, wusste das Betriebssystem, dass der Fehler bei Offset 0x200 des ausführbaren Maschinencodes aufgetreten ist. C++ kann dies mittels der Debug-Informationen bis zur Zeile 16 des Quellcodes zurückverfolgen.

Wie Sie sich vorstellen können, benötigt die Debug-Information viel Speicher. Das macht die ausführbare Datei größer und langsamer. Zur Lösung dieses Problems stellen beide, GNU C++ und Visual C++, einen Erzeugungsmodus bereit, der keine Debug-Informationen enthält. Dieser Modus wird Release-Modus genannt.



10 Min.

10.4 Abfangen von Bug Nr. 2

Stolz auf meinen Erfolg, führe ich das Programm wieder mit der bekannten Eingabe 1, 2, 3 und -1 aus, die vorher das Programm zum Absturz gebracht hat. Diesmal stürzt das Programm zwar nicht ab, es läuft aber auch nicht. Die Ausgabe sieht merkwürdig aus:

```
Dieses Programm muss abstürzen!
Nächste Zahl: 1
Nächste Zahl: 2
Nächste Zahl: 3
Nächste Zahl: -1
Mittelwert ist: -286331151
Press any key to continue
```

Offensichtlich wurde entweder nSum oder nNums (oder beide) nicht richtig deklariert. Um fortzufahren, benötige ich den Inhalt dieser beiden Variablen. In der Tat würde es helfen, wenn ich auch den Inhalt von nValue kennen würde, weil nValue verwendet wird, um die Summe nSum zu berechnen.

Um die Werte von nSum, nNums und nValue zu erfahren, modifiziere ich die for-Schleife wie folgt:

```
for (int nNums = 0; ;)
{
    int nValue;
    cout << »\nNächste Zahl:«;
    cin  >> nValue;
    if (nValue < 0)
    {
        cout << »\nMittelwert ist: »
             << nSum/nNums << »\n«;
        break;
    }
    // Ausgabe kritischer Informationen
    cout << »nSum = » << nSum   << »\n«;
    cout << »nNums = » << nNums  << »\n«;
    cout << »nValue = »<< nValue << »\n«;
    cout << »\n«;

    nSum += nValue;
}
```

Beachten Sie die hinzugefügten Ausgabeanweisungen. Diese drei Zeilen geben die Werte von nSum, nNums und nValue in jeder Iteration der Schleife aus.

Die Ergebnisse der Programmausführung mit der mittlerweile Standard gewordenen Eingabe 1, 2, 3 und -1 sind unten dargestellt. Schon im ersten Schleifendurchlauf scheint der Wert von nSum nicht korrekt zu sein. In der Tat addiert das Programm bereits beim ersten Schleifendurchlauf einen Wert zu nSum. An diesem Punkt würde ich erwarten, dass der Wert von nSum gleich 0 ist. Das scheint das Problem zu sein.

```
Dieses Programm muss abstürzen!
Nächste Zahl: 1
nSum = -858993460
nNums = 0
nValue = 1

Nächste Zahl: 2
nSum = -858993459
nNums = 1
nValue = 2

Nächste Zahl: 3
nSum = -858993457
nNums = 2
nValue = 3

Nächste Zahl:
```

Eine genaue Untersuchung des Programms zeigt, dass `nSum` deklariert, aber nicht initialisiert wurde. Die Lösung ist, die Deklaration von `nSum` wie folgt zu verändern:

```
int nSum = 0;
```

So lange, bis eine Variable initialisiert wurde, ist der Wert der Variablen unbestimmt.

Sobald ich selbst davon überzeugt bin, dass das Ergebnis korrekt ist, räume ich das Programm wie folgt auf:

```
// ErrorProgram - dieses Programm berechnet den
//              Mittelwert von Zahlen - der Bug
//              wurde entfernt
#include <stdio.h>
#include <iostream.h>

int main(int argc, char* pszArgs[])
{
    cout << »Dieses Programm funktioniert!\n«;

    // summiere Zahlen, bis der Benutzer eine
    // negative Zahl eingibt, dann gib den
    // Mittelwert aus
    int nSum = 0;
    for (int nNums = 0; ;nNums++)
    {
        // eine weitere Zahl
        int nValue;
        cout << »\nNächste Zahl:«;
        cin  >> nValue;

        // wenn sie negativ ist ...
        if (nValue < 0)
        {
            // ... dann gib Mittelwert aus
            cout << »\nMittelwert ist: »
                 << nSum/nNums << »\n«;
            break;
        }

        // nicht negativ, addierte zur Summe
        nSum += nValue;
    }
    return 0;
}
```

Ich erzeuge das Programm neu, und teste die Folge 1, 2, 3 und -1 erneut. Diesmal sehe ich den erwarteten Mittelwert von 2:

```
Dieses Programm funktioniert!
Nächste Zahl: 1
Nächste Zahl: 2
Nächste Zahl: 3
Nächste Zahl: -1
Mittelwert ist: 2
```

0 Min.

Nachdem ich das Programm mit einer Reihe von Eingaben getestet habe, bin ich davon überzeugt, dass das Programm nun richtig ist. Ich entferne die zusätzlichen Ausgabeanweisungen und erzeuge das Programm neu, um das Debuggen des Programms abzuschließen.

Zusammenfassung

Es gibt zwei Arten von Fehlern: Compilezeitfehler, die von C++-Compiler erzeugt werden, wenn er auf eine nicht logische Code-Struktur trifft, und Laufzeitfehler, die erzeugt werden, wenn das Programm eine nicht logische Sequenz legaler Instruktionen ausführt.

Compilezeitfehler sind relativ leicht zu beheben, weil die C++-Compiler Sie direkt an die Fehlerstelle führen können. Die Umgebungen von Visual C++ und GNU C++ versuchen, Sie so gut wie möglich dabei zu unterstützen. In dem hier vorgestellten Beispielprogramm waren sie sehr erfolgreich, genau auf das Problem zu zeigen.

Wenn die Meldungen zu Laufzeitfehlern, die von der C++-Umgebung erzeugt werden, nicht ausreichen, bleibt es dem Programmierer überlassen, den Code zu debuggen. In dieser Sitzung haben wir die so genannte Technik der Ausgabeanweisungen verwendet.

- C++-Compiler sind sehr penibel bei dem, was sie akzeptieren, um Programmierfehler nach Möglichkeit während der Programmerzeugung zu finden, in der Fehler leichter zu beheben sind.
- Das Betriebssystem versucht ebenfalls, Laufzeitfehler abzufangen. Wenn diese Fehler zu einem Programmabsturz führen, gibt das Betriebssystem Fehlerinformationen zurück, die Visual C++ und GNU C++ zu interpretieren versuchen.
- Ausgabeanweisungen, die an kritischen Stellen im Programm eingefügt werden, können den Programmierer zur Quelle des Laufzeitfehlers führen.

Obwohl sehr einfach, ist die Technik mit Ausgabeanweisungen sehr effektiv bei kleinen Programmen. Sitzung 16 zeigt Ihnen noch effektivere Debug-Techniken.

Selbsttest

1. Was sind die beiden grundlegenden Fehlertypen? (Siehe »Fehlertypen«)
2. Wie können Ausgabeanweisungen helfen, Fehler zu finden? (Siehe »Die Technik der Ausgabeanweisungen«)
3. Was ist der Debug-Modus? (Siehe Kasten »Wie kann C++ eine Fehlermeldung an den Sourcecode binden?«)

Samstagmorgen – Zusammenfassung

1. ***Ich führe die folgende Funktion aus, um kleine Tonnen in Kilogramm umzurechnen. Wenn ich den Wert 2 eingebe, erhalte ich ein falsches Ergebnis:***

```
int ton2kg(int nTons)
{
    int nLongTons = nTons / 1.1;
    return 1000 * nLongTons;
}
```

   a. Was ist falsch an dem Programm?

   b. Welches Ergebnis erhalte ich?

   Zusatz: Welche Warnung erhalte ich?

2. ***Stellen Sie sicher, dass Sie die Anwort zu 1 kennen (sie können spicken, wenn das nötig ist). Nun, was kann ich tun, um das Problem in 1 zu beheben?***

3. ***Ich habe den folgenden kleinen Bruder der Funktion*** `ton2kg( )` ***geschrieben:***

```
int ton2g(int nTon)
{
    return nTon * 1000000;
}
```

   Weil ich mit großen Schiffen arbeite, übergebe ich der Funktion einen Wert von 5000 Tonnen. Der Wert, den ich zurückbekomme, ist offensichtlich falsch.

   a. Was ist passiert?

   b. Was kann ich zur Lösung des Problems tun?

4. ***Welche der folgenden Aussagen sind wahr?***

```
int n1 = 1, n2 = -3, n3 = 10, n4 = 4
```

   a. `n1 < n2`

   b. `(n1 + n4) == 5`

   c. `(n3 > n4) && (n4 > n1)`

d. `(n3 / 4.0) == 2.5`

e. `(n1 > n3) && (-n4 > n2)`

Was ist der Wert von `n4` ***nach diesen Berechnungen?***

5. ***Gegeben sei*** `n1 = 0101 1101`$_2$

a. ***Was ist das hexadezimale Äquivalent von*** `n1`***?***

b. ***Was ist der dezimale Wert von*** `n1`***?***

c. Was ist der Wert von `n1 * 2` ***in Binärformat?***

Zusatz: Was ist der Unterschied im Bitmuster von `n1` ***und*** `2 * n1`***?***

d. ***Was ist der Wert von*** `n1 | 2`***?***

e. ***Was ist der Wert von*** `(n1 & 2) == 0`***?***

6. ***(Das ist eine harte Nuss) Was ist der endgültige Wert von*** `n1`***?***

```
int n1 = 10;
if (n1 > 11)
{
    if (n1 > 12)
    {
        n1 = 0;
    }
else
{
n1 = 1;
}
}
```

7. ***Was ist der Unterschied zwischen*** `while( )` ***und*** `do...while( )` ***im folgenden Beispielcode:***

```
int n1 = 10;
while(n1 < 5)
{
    n1++;
}

do
{
    n1++;
} while(n1 < 5);
```

8. ***Schreiben Sie eine Funktion*** `double cube(double d)` ***als Ergänzung der ersten Funktion.***

9. ***Was passiert im folgenden Ausdruck in Anwesenheit beider Funktionen?***

```
int n = cube(3.0);
```

10. ***Fügen Sie schließlich die Funktion*** `double cube(int n)` ***zu den ersten beiden Funktionen hinzu. Was passiert?***

    Hinweis: Schreiben Sie die Prototypdeklarationen der drei Funktionen `cube( )` ***auf.***