# Was ist ein Programm?

1

### In diesem Kapitel

- Programme verstehen
- Ihr erstes »Programm« schreiben
- Computersprachen genauer betrachten

In diesem Kapitel erfahren Sie, was ein Programm ist, und was es bedeutet, ein Programm zu schreiben. Sie lernen einen menschlichen Computer kennen und werden einige Programmausschnitte sehen, die für einen realen Computer geschrieben sind. Und schließlich sehen Sie Ihren ersten Codeausschnitt, geschrieben in C++.

Bisher wurden alle Programme auf Ihrem Computer von jemandem anderen geschrieben. Sehr bald schon wird das nicht mehr der Fall sein. Sie werden sich stolz in die Gemeinschaft der Programmierer einreihen.



## Worin unterscheidet sich mein Sohn von einem Computer?

Ein Computer ist eine bemerkenswert schnelle, aber unglaublich dumme Maschine. Ein Computer kann alles, was Sie ihm befehlen (im zulässigen Bereich, versteht sich), aber er macht *genau* das, was Sie ihm befehlen – nicht mehr und nicht weniger.

In dieser Hinsicht ist ein Computer fast genau das Gegenteil eines Menschen: Menschen reagieren intuitiv. Als ich eine zweite Fremdsprache lernte, stellte ich fest, dass es nicht ausreichend ist, zu verstehen, was gesagt wird – es ist genauso wichtig und deutlich schwieriger, zu verstehen, was ungesagt bleibt. Dabei handelt es sich um Informationen, die der Sprecher durch gemeinsame Erfahrungen oder Ausbildung mit dem Zuhörer gemein hat – Dinge, die nicht gesagt werden müssen.

Beispielsweise sage ich zu meinem Sohn Dinge wie »Wasch das Geschirr ab« (weil es eben gemacht werden muss). Diese Anweisungen scheinen ausreichend klar zu sein, aber ein Großteil der Informationen in diesem Satz sind implizit und unausgesprochen.

Angenommen, mein Sohn weiß, was Geschirr ist, und dass sich schmutziges Geschirr normalerweise im Spülbecken befindet. Aber was ist mit Messern und Gabeln? Schließlich habe ich nur vom Geschirr gesprochen, nicht vom Essbesteck, geschweige denn von Gläsern. Und bedeutet waschen, dass er das von Hand machen muss, oder kann er das Ganze auch in die Spülmaschine packen, wo es automatisch gewaschen, gespült und getrocknet wird?

Tatsächlich ist »Wasch das Geschirr ab« für meinen Sohn eine ausreichende Anweisung. Er kann diesen Satz zerlegen und ihn mit Informationen kombinieren, die wir beide kennen,

unter anderen umfangreiches Arbeitswissen über schmutziges Geschirr, um daraus auf sinnvolle Weise abzuleiten, was ich von ihm will – ob er es dann macht, steht wieder auf einem anderen Blatt. Ich nehme an, er kann genauso schnell begreifen, worum es geht, wie ich es gesagt habe – innerhalb von etwa 1 bis 2 Sekunden.

Ein Computer kann eine so unpräzise Aussage wie »Wasch das Geschirr ab« nicht erweitern oder kürzen. Sie müssen dem Computer genau sagen, was er mit jedem einzelnen Geschirrteil machen soll, Sie müssen ihm mitteilen, dass auch Besteck dazugehört, und ihm erklären, wie eine Gabel, ein Löffel oder eine Tasse abzuwaschen sind. Wann hört das Programm auf, ein Geschirrteil zu waschen (d. h. wie erkennt es, dass es sauber ist)? Wann hört es auf, abzuwaschen (d. h. woher weiß es, dass seine Aufgabe abgeschlossen ist)?

Mein Sohn hat ein enormes Gedächtnis – man weiß nicht genau, wie viel Merkkapazität ein durchschnittlicher Mensch besitzt, aber es ist in jedem Fall sehr viel. Leider ist das Gedächtnis des Menschen nicht immer geordnet. Die Zeugen von Verbrechen erinnern sich meistens an keinerlei Details, selbst nicht unmittelbar nach dem Vorfall. Und zwei Zeugen desselben Vorfalls sind sich meistens höchst uneinig darüber, was eigentlich passiert ist.

Auch Computer haben ein enormes Gedächtnis, und das ist extrem gut. Nachdem eine Tatsache gespeichert wurde, kann ein Computer sie beliebig oft abrufen, ohne dass sich irgendetwas daran ändert. Anfang der 1980er-Jahre war Speicher noch teuer, deshalb hatte der erste IBM-PC nur 16 KB (das sind 16.000 Bytes). Er war auf überwältigende 64 KB aufrüstbar. Vergleichen Sie dies mit dem Arbeitsspeicher von 2 GB bis 6 GB, wie er uns in den meisten modernen Computern zur Verfügung steht (ein 1 GB, ein Gigabyte, entspricht *einer Million Bytes*)!

In den frühen Tagen der PCs war Speicher teuer. Deshalb enthielt der IBM-PC zusätzliche Speicherchips und Decodier-Hardware, die feststellen konnte, wenn ein Speicherfehler auftrat. Fiel ein Speicherchip aus, konnte diese Schaltung das Problem erkennen und melden, bevor das Programm abstürzte. Dieser sogenannte *Paritätsspeicher* wurde schon nach ein paar Jahren verworfen. Soweit ich weiß, gibt es ihn heute nicht mehr, außer in ganz speziellen Anwendungen, die extrem zuverlässig sein müssen – die Speicherkarten von heute fallen so gut wie nicht mehr aus.

Andererseits beherrschen Menschen verschiedene Verarbeitungsformen, mit denen sich Computer extrem schwertun. Beispielsweise können Menschen sehr gut die Bedeutung eines Satzes erkennen, der durch sehr viel Hintergrundlärm gestört wird. Digitale Mobiltelefone dagegen (die mindestens so sehr Computer wie Telefon sind) haben die unangenehme Eigenschaft, dass sie stumm werden, wenn der Lärmpegel über eine eingebaute Schwelle steigt.

Das restliche Kapitel beschäftigt sich mit Anweisungen, die dem Computer schon sehr viel besser mitteilen, wie er »das Geschirr abwaschen« (oder eine vergleichbar unübersichtliche Aufgabe erledigen) soll.

# Einen »menschlichen Computer« programmieren

Bevor ich anfange, Ihnen beizubringen, wie Sie Programme für Computer schreiben, zeige ich Ihnen ein Programm, mit dem Sie das menschliche Verhalten lenken. Dann werden Sie besser verstehen, worum es geht. Die Entwicklung eines Programms für die Lenkung eines Menschen ist sehr viel einfacher als die Entwicklung von Programmen für Computer-Hardware. Das liegt daran, dass wir Menschen sehr geübt im Umgang miteinander sind – woraus sich eine Vertrautheit damit ergibt, wie Menschen ticken und wie sie arbeiten (und auch ein gewisses Verständnis dafür). Außerdem haben wir eine gemeinsame Sprache, wir müssen nicht alles in Einsen und Nullen übersetzen. Aber wir wollen davon ausgehen, dass der menschliche Computer in diesem Gedankenexperiment jede Anweisung sehr wörtlich nimmt – das Programm muss also äußerst genau sein.

Die Aufgabenstellung, die ich für dieses Experiment gewählt habe, ist es, unseren menschlichen Computer anzuweisen, einen platten Reifen zu wechseln.

## Den Algorithmus erstellen

Die Anweisungen für das Wechseln eines platten Reifens sind ganz einfach und könnten etwa wie folgt aussehen:

1. Hebe das Fahrzeug an.
2. Entferne die Radmuttern, mit denen der defekte Reifen am Auto befestigt ist.
3. Entferne den Reifen.
4. Montiere den neuen Reifen.
5. Bringe die Radmuttern an.
6. Senke das Fahrzeug ab.

Selbst diese alltäglichen Begriffe können unübersichtlich sein. Technisch gesehen, halten die Radmuttern das Rad, nicht den Reifen am Auto. Der Einfachheit halber wollen wir annehmen, dass die Begriffe »Rad« und »Reifen« synonym verwendet werden können und dass der Computer sie als dasselbe Konzept versteht.

Zunächst scheinen diese Anweisungen ganz detailliert zu sein, aber sie stellen noch lange kein Programm dar. Eine solche Anweisungsmenge wird als *Algorithmus* bezeichnet – eine Beschreibung der auszuführenden Schritte, die normalerweise sehr abstrakt ist. Ein Algorithmus ist detailliert, aber allgemein. Ich könnte diesen Algorithmus verwenden, um jeden platten Reifen zu reparieren, den ich je hatte, und den ich je haben werde. Ein Algorithmus enthält jedoch nicht ausreichend viele Details, um selbst unserem bewusst einfach gehaltenen menschlichen Computer zu gestatten, die Aufgabe auszuführen.

## Die Entwicklung der Reifenwechsel-Sprache

Bevor wir ein Programm schreiben können, brauchen wir eine Sprache, auf die wir uns alle einigen können. Im restlichen Buch wird das die Sprache C++ sein, aber für dieses Beispiel verwende ich eine imaginäre Sprache: RWS (die Reifenwechsel-Sprache). Ich habe RWS speziell auf die Aufgabe zugeschnitten, Reifen zu wechseln.

RWS enthält ein paar Substantive, die in der Welt des Reifenwechsels üblicherweise vorkommen:

- ✔ `Auto`
- ✔ `Reifen`
- ✔ `Mutter`
- ✔ `Wagenheber`
- ✔ `Werkzeugkasten`
- ✔ `Ersatzreifen`
- ✔ `Schraubenschlüssel`

Außerdem enthält RWS die folgenden Verben:

- ✔ `nehmen`
- ✔ `bewegen`
- ✔ `lösen`
- ✔ `drehen`

Außerdem muss der Prozessor für die Ausführung der RWS in der Lage sein, zu zählen und einfache Entscheidungen zu treffen. Und schließlich kennt der RWS-Prozessor Richtungen wie oben und unten, links und rechts sowie im Uhrzeigersinn und gegen den Uhrzeigersinn.

Diese Wörter in RWS sind alles, was der Reifenwechsel-Roboter (der imaginäre menschliche Computer) versteht. Jeder andere Befehl, der nicht Teil der Reifenwechsel-Sprache ist, bewirkt einen verständnislosen Blick des menschlichen Reifenwechsel-Prozessors.

## Das Programm erstellen

Jetzt wollen wir den Algorithmus, der in natürlicher Sprache geschrieben war, in ein Programm in RWS umwandeln. Das ist nicht ganz so einfach, wie es scheinen mag. Betrachten Sie beispielsweise den Satz »Entferne die Radmuttern.« In diesem Satz fehlen einige Aussagen. Das Wort *entferne* ist nicht im Vokabular des Prozessors enthalten. Außerdem wurde in diesem Satz das Wort *Schraubenschlüssel* nicht erwähnt (das der Computer kennt), obwohl wir alle wissen, dass ein Schraubenschlüssel beteiligt sein muss. Wir können nicht davon ausgehen, dass der Computer weiß, was wir wissen.

(Falls Sie noch keinen platten Reifen gewechselt haben und nicht wissen, dass für das Entfernen einer Radmutter ein Schraubenschlüssel benötigt wird – oder was eine Radmutter überhaupt ist –, dann machen wir einfach weiter. Sie werden es herausfinden.)

Die folgenden Schritte *implementieren* den Satz »Entferne eine Radmutter«, wobei nur die Verben und Substantive der RWS verwendet werden:

```
1. Nimm den Schraubenschlüssel.
2. Bewege den Schraubenschlüssel zur Radmutter.
3. Drehe den Schraubenschlüssel fünfmal gegen den Uhrzeigersinn.
4. Bewege den Schraubenschlüssel zum Werkzeugkasten.
5. Lasse den Schraubenschlüssel los.
```

Jetzt betrachten wir diese Aspekte der *Syntax* (der erforderlichen Anordnung der Wörter), die in diesem Beispiel der RWS vorgegeben ist:

- ✔ Jeder Befehl beginnt mit einem Verb.
- ✔ Für das Verb `nehmen` ist ein einzelnes Substantiv als Objekt erforderlich.
- ✔ Das Verb `drehen` benötigt ein Substantiv, eine Richtung und eine Anzahl, wie viele Umdrehungen gemacht werden sollen.

Der Programmausschnitt sollte jedoch ganz einfach zu lesen sein (schließlich ist dies kein Buch über RWS).

Sie könnten diesen kleinen Exkurs in die Reifenwechsel-Sprache überspringen, aber Sie müssen die *Grammatik* jedes C++-Befehls lernen. Sonst funktioniert es nicht.

Das Programm beginnt mit Schritt 1 und durchläuft die einzelnen Schritte bis hin zu Schritt 5. In Programmier-Terminologie sagen wir, das Programm *läuft* von Schritt 1 bis Schritt 5. Natürlich geht das Programm nirgendwo hin – der Prozessor erledigt die gesamte Arbeit – aber der *Programmablauf* ist ein üblicher Begriff für diese reibungslose Ausführung der einzelnen Schritte.

Selbst bei einer beiläufigen Betrachtung dieses Programms ist ein Problem offensichtlich: Was passiert, wenn es keine Radmutter gibt? Ich nehme an, es ist harmlos, den Schraubenschlüssel an einem Bolzen ohne Mutter zu drehen, aber man vergeudet Zeit und unter einer guten Lösung stelle ich mir etwas Besseres vor. Die Reifenwechsel-Sprache braucht eine Verzweigungsmöglichkeit, die gestattet, dass das Programm abhängig von externen Bedingungen den einen oder den anderen Pfad einschlägt. Wir brauchen eine `if`-Anweisung, etwa wie folgt:

```
1. Nimm den Schraubenschlüssel.
2. Falls eine Radmutter vorhanden ist
3. {
4.       Bewege den Schraubenschlüssel zur Radmutter.
5.       Drehe den Schraubenschlüssel fünfmal gegen den Uhrzeigersinn.
6. }
7. Bewege den Schraubenschlüssel zum Werkzeugkasten.
8. Lasse den Schraubenschlüssel los.
```

Das Programm beginnt mit Schritt 1, wie bereits zuvor, und nimmt einen Schraubenschlüssel. Bevor es im zweiten Schritt den Schraubenschlüssel sinnlos um eine leere Schraube dreht, prüft es, ob eine Radmutter vorhanden ist. Ist dies der Fall, wird der Ablauf wie zuvor mit den Schritten 3, 4 und 5 fortgesetzt. Ist es nicht der Fall, überspringt der Programmablauf diese unnötigen Schritte und fährt direkt mit Schritt 7 fort, um den Schraubenschlüssel wieder in den Werkzeugkasten zu legen.

In Computer-Sprache sagen Sie, dass das Programm den logischen Ausdruck »Ist eine Radmutter vorhanden?« ausführt. Dieser Ausdruck gibt entweder `true` zurück (ja, richtig, die Radmutter ist vorhanden), oder `false` (nein, falsch, es gibt keine Radmutter).

Was ich hier als Schritt bezeichne, würde in einer Programmiersprache normalerweise als *Anweisung* bezeichnet. Ein *Ausdruck* ist eine Art Aussage, die einen Wert zurückgibt. Beispielsweise ist 1 + 2 ein Ausdruck. Ein logischer Ausdruck ist ein Ausdruck, der den Wert `true` (`richtig`) oder `false` (`falsch`) zurückgibt. Beispielsweise ist der Wert von »Ist der Autor dieses Buches ein schöner Mann?« gleich `true`.

Die geschweiften Klammern { } in der Reifenwechsel-Sprache sind erforderlich, um dem Programm mitzuteilen, welche Schritte übersprungen werden sollen, wenn die Bedingung nicht `true` ist. Die Schritte 4 und 5 werden nur ausgeführt, wenn die Bedingung `true` ist.

Ich weiß, dass es nicht notwendig ist, einen Schraubenschlüssel zur Hand zu nehmen, wenn keine Schraubenmutter zu entfernen ist, aber das wollen wir hier einfach ignorieren.

Dieses verbesserte Programm weist immer noch ein Problem auf: Woher erkennen Sie, dass fünf Umdrehungen des Schraubenschlüssels ausreichend sind, um die Radmutter zu entfernen? Für die meisten Reifen, die ich kenne, genügt das. Sie könnten die Anzahl der Umdrehungen erhöhen, um wirklich auf der sicheren Seite zu sein, beispielsweise auf 25. Wenn sich die Radmutter beispielsweise nach der 20. Umdrehung löst, dann dreht sich der Schraubenschlüssel eben fünfmal umsonst. Das ist eine harmlose Lösung, aber vergeudete Arbeitskraft.

Ein besserer Ansatz ist eine Art Anweisung, die in einer Schleife abläuft und eine Prüfung durchführt – in unserer Reifenwechsel-Sprache:

```
1. Nimm den Schraubenschlüssel.
2. Falls eine Radmutter vorhanden ist
3. {
4.       Bewege den Schraubenschlüssel zur Radmutter.
5.       Solange (Radmutter noch am Fahrzeug sitzt)
6.       {
7.             Drehe den Schraubenschlüssel einmal gegen den Uhrzeigersinn.
8.       }
9. }
```

```
10. Bewege den Schraubenschlüssel zum Werkzeugkasten.
11. Lasse den Schraubenschlüssel los.
```

Hier verläuft das Programm von Schritt 1 bis Schritt 4 wie zuvor. In Schritt 5 muss der Prozessor jedoch eine Entscheidung treffen: Sitzt die Radmutter noch am Fahrzeug? Bei dem ersten Durchlauf nehmen wir dies an, sodass der Prozessor Schritt 7 ausführt und den Schraubenschlüssel einmal gegen den Uhrzeigersinn dreht. An dieser Stelle kehrt das Programm zu Schritt 5 zurück und wiederholt die Überprüfung. Sitzt die Radmutter immer noch am Fahrzeug, wiederholt der Prozessor Schritt 7 und kehrt wieder zu Schritt 5 zurück. Irgendwann ist die Radmutter gelöst und die Bedingung in Schritt 5 ergibt den Wert `false`. An dieser Stelle geht die Steuerung im Programm zu Schritt 9 weiter und das Programm wird fortgesetzt wie zuvor.

Diese Lösung ist der vorhergehenden Lösung überlegen: Sie trifft keine Annahmen im Hinblick darauf, wie viele Umdrehungen erforderlich sind, um eine Radmutter zu lösen. Es wird also keine überflüssige Arbeit ausgeführt, bei der der Prozessor eine Radmutter dreht, die schon gar nicht mehr vorhanden ist, und es passiert auch nicht, dass eine Radmutter zurückbleibt, die noch nicht ganz gelöst wurde.

So elegant diese Lösung ist, gibt es immer noch ein Problem: Sie montiert nur eine einzige Radmutter vom Fahrzeug. Die meisten mittelgroßen Fahrzeuge haben fünf Radmuttern an jedem Rad. Wir könnten die Schritte 2 bis 9 fünfmal wiederholen, einmal für jede Radmutter. Aber auch das funktioniert nicht optimal. Die meisten Kleinwagen haben nur vier Radmuttern, große Pickups haben bis zu acht.

Das folgende Programm erweitert unsere Grammatik um die Möglichkeit, die Radmuttern zu durchlaufen. Dieses Programm funktioniert unabhängig von der Anzahl der Radmuttern am Rad:

```
1.  Nimm den Schraubenschlüssel.
2.  Für jede Radmutter am Rad
3.  {
4.        Falls eine Radmutter vorhanden ist
5.        {
6.
7.              Bewege den Schraubenschlüssel zur Radmutter.
8.              Solange (Radmutter noch am Fahrzeug sitzt)
9.              {
10.                Drehe den Schraubenschlüssel einmal gegen den Uhrzeigersinn.
11.             }
12.       }
13. }
14. Bewege den Schraubenschlüssel zum Werkzeugkasten.
15. Lasse den Schraubenschlüssel los.
```

Dieses Programm beginnt wie zuvor damit, einen Schraubenschlüssel aus dem Werkzeugkasten zu nehmen. Ab Schritt 2 durchläuft es jedoch für jede Radmutter am Rad die Schritte bis 12.

Beachten Sie, dass die Schritte 7 bis 10 für jede Schraube wiederholt werden. Man spricht auch von einer *verschachtelten* Schleife. Die Schritte 7 bis 10 werden als die *innere Schleife* bezeichnet. Die Schritte 2 bis 12 bilden die *äußere Schleife*.

Das vollständige Programm entsteht durch die Ergänzung ähnlicher Implementierungen in jedem Schritt des Algorithmus.

### Computerprozessoren

Es scheint eine ganz einfache Aufgabe zu sein, ein Rad von einem Auto zu montieren, und dennoch braucht man 11 Anweisungen in einer Sprache, die speziell für das Wechseln von Reifen entwickelt wurde, um die Radmuttern zu entfernen. Das fertige Programm enthält mehr als 60 oder 70 Schritte mit zahlreichen Schleifen. Und wenn man eine Logik für die Prüfung auf Fehlerbedingungen einbaut, wie beispielsweise festsitzende oder fehlende Radmuttern, braucht man noch mehr Schritte.

Stellen Sie sich jetzt vor, wie viele Anweisungen ausgeführt werden müssten, um etwas Einfaches zu bewerkstelligen, wie beispielsweise ein Fenster über den Anzeigebildschirm zu bewegen (denken Sie daran, dass ein typischer Bildschirm $1280 \times 1024$ Pixel anzeigt, das ist etwas mehr als eine Million). Glücklicherweise ist ein Computerprozessor zwar dumm, aber extrem schnell. Der Prozessor in Ihrem PC beispielsweise kann mehrere Milliarden Anweisungen pro Sekunde ausführen. Die Anweisungen in Ihrem generischen Prozessor leisten nicht besonders viel – man braucht mehrere Anweisungen, nur um ein Pixel zu verschieben –, aber wenn Sie gleichzeitig eine Milliarde bewegen können, ist die Manipulation einer lächerlichen Million Pixel ein Kinderspiel.

Der Computer macht nichts, wofür er nicht programmiert wurde. Die Bereitstellung der Reifenwechsel-Sprache war nicht ausreichend, um meinen platten Reifen zu wechseln – jemand musste die Programmanweisungen schreiben, um Schritt für Schritt festzulegen, was der Computer tun sollte. Und die Entwicklung eines realen Programms, das alle etwa auftretenden speziellen Bedingungen berücksichtigt, ist keine einfache Aufgabe. Ein Programm zu schreiben, das industriell einsetzbar ist, kann eine echte Herausforderung sein.

Die Frage lautet also: »Warum machen wir uns die Mühe überhaupt?« Weil der Computer, nachdem er einmal programmiert ist, die gewünschte Funktion immer wieder, unermüdlich und in der Regel sehr viel schneller als jeder Mensch ausführen kann.

## Computersprachen

Die Reifenwechsel-Sprache ist natürlich keine reale Computersprache. Für reale Computer gibt es keine Maschinenanweisungen wie `nimm` oder `drehe`. Und noch schlimmer ist: Computer »denken« in einer Abfolge aus Einsen und Nullen. Jeder interne Befehl ist nichts anderes als eine Reihe von Binärzahlen. Reale Computer befolgen Anweisungen wie 01011101, womit beispielsweise einer Zahl in einem speziellen Register 1 hinzugefügt wird. So schwierig die Programmierung in der Reifenwechsel-Sprache sein mag, die Programmierung mit langen Zahlenketten ist noch schwieriger.

Die Muttersprache der Computer wird als *Maschinensprache* bezeichnet und wird in der Regel durch eine Zahlenfolge in Binärform (Basis 2) oder in Hexadezimalform (Basis 16) dargestellt. Nachfolgend sehen Sie die ersten 64 Byte aus dem Programm `Conversion` in Kapitel 3.

```
<main+0>: 01010101 10001001 11100101 10000011 11100100
          11110000 10000011 11101100
<main+8>: 00100000 11101000 00011010 01000000 00000000
          00000000 11000111 01000100
<main+16>:00100100 00000100 00100100 01110000 01000111
          00000000 11000111 00000100
<main+24>:00100100 10000000 01011111 01000111 00000000
          11101000 10100110 10001100
<main+32>:00000110 00000000 10001101 01000100 00100100
          00010100 10001001 01000100
```

Glücklicherweise schreibt niemand mehr Programme in Maschinensprache. Schon sehr früh hat jemand festgestellt, dass es für einen Menschen sehr viel einfacher ist, eine Anweisung wie `ADD 1 , REG 1` (»Addiere 1 zu dem Wert in Register 1«) zu verstehen statt `01011101`. In der Ära unmittelbar nach der Maschinensprache schrieben die Programmierer ihre Programme in der sogenannten *Assembler*-Sprache und übergaben sie dann einem Programm, das als *Assembler* bezeichnet wird und das alle diese Anweisungen in äquivalente Befehle in Maschinensprache übersetzte.

Das von Menschen geschriebene Programm wird als *Quellcode* bezeichnet, weil es die Quelle allen Ungemachs ist. Nur ein Scherz – natürlich heißt er so, weil er die Quelle für das Programm darstellt. Die vom Computer ausgeführten Einsen und Nullen werden hingegen als *Objektcode* bezeichnet.

Nachfolgend sehen Sie die ersten Assembler-Anweisungen des Programms `Conversion`, kompiliert für die Ausführung auf einem Intel-Prozessor unter dem Betriebssystem Windows. Dies ist dieselbe Information, die oben in Binärform gezeigt wurde.

```
<main>:       push   ebp
<main+1>:     mov    ebp,esp
<main+3>:     and    esp, 0xfffffff0
<main+6>:     sub    esp, 0x20
<main+9>:     call   0x40530c <_main>
<main+14>:    movl   [esp+0x04],0x477024
<main+22>:    movl   [esp],0x475f80
<main+29>:    call   0x469fac <operator<<>
<main+34>:    lea    eax,[esp+0x14]
<main+38>:    mov    [esp+0x04],eax
```

Das ist immer noch nicht gut verständlich, aber schon sehr viel besser als nur eine Folge aus Nullen und Einsen. Machen Sie sich aber keine Gedanken – in diesem Buch werden wir keine Programme in Assembler-Sprache schreiben.

Der Computer führt auch nie Anweisungen in Assembler-Sprache aus. Er führt die Maschinenbefehle aus, die aus der Umwandlung der Assembler-Anweisungen entstehen.

## Höhere Sprachen

Die Assembler-Sprache merkt man sich vielleicht leichter als die Maschinensprache, aber es besteht immer noch ein riesiger Unterschied zwischen einem Algorithmus wie dem Reifenwechsel-Algorithmus und einer Abfolge von `MOVE`- und `ADD`-Anweisungen. In den 1950er-Jahren begann man, immer aussagekräftigere Sprachen zu entwickeln, die von einem sogenannten *Compiler* automatisch in Maschinensprache umgewandelt werden konnten. Man sprach auch von *höheren Programmiersprachen*, weil sie mit einem höheren Abstraktionsgrad geschrieben wurden als die Assembler-Sprache.

Eine der ersten dieser Sprachen war COBOL (Common Business-Oriented Language). Die Idee hinter COBOL war es, dem Programmierer zu ermöglichen, Befehle zu schreiben, die englischen Sätzen so ähnlich wie möglich waren. Plötzlich konnten die Programmierer Sätze wie die folgenden schreiben, um Temperatur von Celsius in Fahrenheit umzuwandeln (ob Sie es glauben oder nicht, das ist genau das, was die oben gezeigten Ausschnitte aus den Programmen in Maschinen- und Assemblersprache machen):

```
INPUT CELSIUS_TEMP
SET FAHRENHEIT_TEMP TO CELSIUS_TEMP * 9/5 + 32
WRITE FAHRENHEIT_TEMP
```

Die erste Zeile dieses Programms liest eine Zahl von der Tastatur oder aus einer Datei ein und speichert sie in der Variablen `CELSIUS_TEMP`. Die nächste Zeile multipliziert diese Zahl mit 9/5 und addiert 32 zum Ergebnis, um die äquivalente Temperatur in Grad Fahrenheit zu berechnen. Dieses Ergebnis speichert das Programm in einer Variablen namens `FAHRENHEIT_TEMP`. Die letzte Programmzeile gibt diesen umgewandelten Wert auf dem Display aus.

Man fuhr damit fort, weitere Programmiersprachen zu entwickeln, die alle ihre ganz eigenen Stärken und Schwächen hatten. Einige Sprachen, wie beispielsweise COBOL, waren sehr wortreich, aber einfach zu lesen. Andere Sprachen, wie Datenbanksprachen oder die Sprachen für die Erstellung interaktiver Webseiten, wurden für sehr spezifische Aufgaben entworfen. Diese Sprachen verwenden leistungsstarke Konstrukte, um spezifische Problembereiche zu behandeln.

## Die Sprache C++

C++ (ausgesprochen übrigens als »C plus plus«) ist eine symbolisch ausgelegte höhere Sprache. C++ entstand zunächst in den 1970er-Jahren als einfaches C in den Bell Labs. Ein paar Leute arbeiteten an einem neuen Konzept für ein Betriebssystem, das als Unix bezeichnet wurde (der Vorgänger von Linux und Mac OS und in der Industrie und der akademischen Welt heute noch verwendet). Die Originalsprache C, wie sie in den 1970er-Jahren in den Bell Labs entwickelt wurde, wurde 1989 leicht verändert und als weltweiter ISO-Standard eingeführt.

C++ wurde als Erweiterung der grundlegenden Sprache C entwickelt, hauptsächlich, indem die Funktionen eingefügt wurden, die in den Teilen V und VI dieses Buchs beschrieben werden.

Wenn ich sage, C++ ist *symbolisch*, bedeutet das, dass es nicht sehr wortreich ist. Es verwendet Symbole statt der langen Wörter in Sprachen wie COBOL. C++ ist jedoch einfach zu lesen, wenn man sich erst einmal an die Bedeutung der Symbole gewöhnt hat. Der bereits in COBOL gezeigte Code für die Umwandlung von Celsius in Fahrenheit sieht in C++ wie folgt aus:

```
cin >> celsiusTemp;
fahrenheitTemp = celsiusTemp * 9 / 5 + 32;
cout << fahrenheitTemp;
```

Die erste Zeile liest einen Wert in die Variable `celsiusTemp` ein. Die nachfolgende Berechnung wandelt diese Celsius-Temperatur in Fahrenheit um, wie zuvor: Die dritte Zeile gibt das Ergebnis aus.

C++ hat noch mehrere weitere Vorteile im Vergleich zu anderen höheren Sprachen. Erstens ist C++ universell. Es gibt für fast jeden Computer einen C++-Compiler.

Darüber hinaus ist C++ effizient. Je mehr Aufgaben eine höhere Sprache automatisch erledigen will (um Ihnen die Programmierung zu erleichtern), desto weniger effizient ist der daraus erstellte Maschinencode in der Regel. Bei kleinen Programmen, wie sie in diesem Buch gezeigt werden, spielt das keine Rolle. Sehr relevant wird es jedoch, wenn Sie wirklich große Datenmengen manipulieren, wie beispielsweise die Bewegung von Pixeln auf dem Bildschirm, oder wenn Sie hervorragende Echtzeitleistung benötigen. Es ist kein Zufall, dass Unix und Windows in C++ geschrieben sind und das Macintosh-Betriebssystem in einer Sprache, die C++ sehr ähnlich ist.

Das Ziel der verbleibenden Kapitel in diesem Buch ist es, Ihnen die Programmierung in C++ näherzubringen. Sie brauchen sich nicht jedes Detail über C++ zu merken. Aber zum Schluss werden Sie es so gut kennen, dass Sie schon richtig gute Programme damit schreiben können.