Kapitel 2
Vom Problem zum Programm

Kapiteltelegramm

- **Softwareentwicklungsmethoden**
  Praktiken, die es erlauben, aus dem Kundenwunsch ein Programm zu machen. Sie haben das Ziel, den Prozess der Softwareentwicklung reibungslos zu gestalten, einen zufriedenen Kunden und ein gutes Programm zu erhalten.
- **Software Craftsmanship**
  Vielleicht eine Softwareentwicklungsmethode, vielleicht auch nur die Sicht des Programmierers auf sein Werk als Produkt seiner Professionalität.
- **Funktionstests**
  Im Rahmen der Methodik die Sicherstellung, dass das Programm »tut, was es soll«. Es gibt mehrere Arten von Tests, und viele beziehen sich stark auf die Kundensicht.
- **Unittest**
  Anders die Unittests, deren Fokus der Programmierer ist. Er legt sie (meist) fest, er implementiert sie (meist) und führt sie sehr häufig selbst aus. Weil diese Tests die kleinen Einheiten (daher der Name) des Programms prüfen, geben sie dem Entwickler gut lokalisiert Meldungen im Fehlerfall.
- **Refactoring**
  Das Anpassen von Programmteilen, die an anderer Stelle sind als die gerade bearbeitete Änderung.
- **Entwurfsmuster**
  Auch »Patterns« genannt. Mit Entwurfsmustern gibt man wiederkehrenden Aufgaben und Lösungsansätzen einen Namen, um sich effizient über sie austauschen zu können.
- **Algorithmus**
  In der Informatik etwas, das die Lösung einer Aufgabe formal beschreibt.
- **Komplexität**
  Im Sinne der Informatik ein Maß für die Schwierigkeit, eine Aufgabe (mit dem Computer) zu lösen.
- **Ressource**
  An der Menge der verbrauchten Ressourcen kann man den Aufwand eines Algorithmus messen. Typischerweise werden benötigte Zeit und verbrauchter Speicher betrachtet.

Mein erstes Buch, in dem strukturiert darauf eingegangen wurde, was ein Programm eigentlich ist, hieß wie dieses Kapitel. Das, was mir jenes Buch *nicht* beigebracht hat, war, wie ich *programmiere*. In dem Buch ging es darum, wie man ein Programm mathematisch auffasst, beschreibt und analysiert. Es war kein *praktischer* Leitfaden zum Programmieren, oh nein. Dennoch sind die Dinge darin enorm wichtig, denn sie schulen den Blick für das Wesentliche, und aus diesem Blickwinkel lohnt es sich oft, Dinge zu hinterfragen.

Aus dieser Theorie wurde die Praxis des Programmieralltags. Alle Dinge jenes Buches sind immer noch wahr, doch treten sie in den Hintergrund. Die beiden wichtigsten, so habe ich gelernt, sind:

- die Praxis, das Üben: Trauen Sie sich, »hinter die Kulissen« zu blicken, und probieren Sie neue Wege aus.
- die Kommunikation: Reden Sie mit anderen über Ihr Programm, diskutieren Sie Ihre Lösung, dokumentieren Sie, machen Sie Code-Reviews – werden Sie besser durch »Input und Output«.

Beides zusammen führt zum *Verstehen*. Und damit haben Sie alle Werkzeuge in der Hand, um Probleme vom Kopf in den Computer zu übertragen.

2.1 Was ist Programmieren?

In den Vordergrund treten die konkreten Methoden, wie Sie aus einer Aufgabe bzw. einer Fragestellung ein fertiges Programm machen. Für mich ist somit der Alltag des Programmierens ein gänzlich anderer als der, den mir jenes Buch zu vermitteln schien. Die Grundprinzipien sind geblieben, doch die *Wege und Werkzeuge* sind anders.

So steht am Anfang immer ein Wunsch – mal mehr und mal weniger gut beschrieben. Diesen Wunsch muss jemand in eine genaue Vorstellung packen. Das kann derjenige in Textform tun, oder er liefert eine mündliche Beschreibung.

Spätestens jetzt kommt der Programmierer ins Spiel, vielleicht zuvor noch ein Architekt, der die Aufgaben zerteilt und auf einer größeren Landkarte unterbringt. Sie müssen sich dann hinsetzen und die Aufgabe weiter herunterbrechen – immer weiter, immer weiter, bis Sie einzelne »unteilbare« Bausteine haben.

Dann setzen Sie sich hin und schreiben in der Sprache des Computers – beziehungsweise einer Zwischensprache, die ein Compiler oder Interpreter übersetzt – das Programm, das die Aufgabe erfüllt. Ziel erreicht?

2.2 Softwareentwicklungsmethoden

Der Programmierer muss *testen*, ob sein Programm auch wirklich die Aufgabe erfüllt. Der Kunde muss das Gleiche prüfen und muss gleichzeitig sicher sein, dass seine Wünsche durch Abstraktionen und Konkretisierungen auch richtig verstanden worden sind.

Es hat sich herausgestellt, dass man deshalb von Anfang an die *Testbarkeit* und *Überprüfbarkeit* mit einplanen muss – vom Zeitaufwand her, aber auch in dem Prozess selbst. Der Programmierer kann zum Beispiel schon beim Coden darauf achten, dass sein Programm auch gut zu testen ist.

Gleiches gilt für Änderungen, die der Kunde wünscht: In allen Stufen dieses Ablaufs wird man sicherstellen müssen, dass sich geänderte Wünsche, geänderte Anforderungen, geänderte Voraussetzungen auch in dem fertigen Programm noch umsetzen lassen.

Um dies alles sicherzustellen, gibt es viele Ansätze, viele Methoden der *Softwareentwicklung*. Sei es das *Wasserfallmodell*, das *V-Modell*, *Agil*, *Extreme Programming* oder *Scrum* – alle diese Ansätze beschäftigen sich (von unterschiedlichen Standpunkten aus) damit, wie aus Wünschen gute Programme werden. Über die konkreten Methoden lesen Sie bitte die entsprechenden Bücher, Ihnen stehen mehrere Regalmeter zur Verfügung.

Es gibt jedoch Gemeinsamkeiten, die Ihnen aus der Perspektive dieses Buches bei Ihrem »Einstieg in modernes C++« weiterhelfen werden. Mein persönlicher Wunsch beim Programmieren ist – egal nach welcher Methode –, dass ich zufrieden mit meinem Produkt bin. Meiner Meinung nach gibt es einige universelle Anforderungen an ein Programm, ohne deren Erfüllung ein Programm nicht wert ist, so genannt zu werden. Ähnlich wie ein Haus als Produkt nur dann ein Haus ist, wenn es mich bei Regen trocken und im Winter warm hält. Ein Haus ist das Produkt von Fachleuten, eine Teamarbeit, an der unter anderem viele Handwerker beteiligt sind. Und ähnlich wie ein Dachdecker stolz auf ein von ihm gebautes Dach ist, wenn es – laienhaft ausgedrückt – oben dicht und unten warm ist, so muss sich der professionelle Programmierer als Handwerker sehen, der ebenfalls Kriterien an sein Produkt anlegen sollte – Kriterien, ohne die es nicht geht.

Um den Vergleich zum Handwerk zu verdeutlichen, hat sich in letzter Zeit der Begriff der *Software Craftsmanship* (in etwa: *Handwerk der Softwareentwicklung*) entwickelt. Während die einen dies auch nur als eine weitere (programmiererfokussierte) Methode der Softwareentwicklung betrachten, erlaube ich mir dabei vor allem den Wunsch nach *Professionalität* des Softwareentwicklers herauszuheben. Ein Entwickler(-Team), der professionelle Kriterien an sein eigenes Produkt hat, wird sich weniger gegenüber seinen Partnern rechtfertigen müssen. Die professionelle Herangehensweise stellt sicher, dass das Abgelieferte als wertvolles Ganzes gesehen wird, von Kunde und Macher zugleich. Das Programm wird zum Haus.

Ich möchte hier ein paar dieser universellen professionellen Methoden nennen, die Ihnen vor allem im Coding-Alltag beim Einstieg in C++ helfen werden.

- **Testen**
  Testen Sie Ihren Code. Sie werden schnell merken, dass Sie – mit der Absicht im Hinterkopf, Ihren Code testen zu wollen – schon anders programmieren werden. Sie werden den Schnittstellen schaffen wollen, die Ihnen das *automatische Testen* vereinfachen. Bei Tests ist die einfache Ausführbarkeit und Reproduzierbarkeit sehr wichtig.

- **Funktions- und Gesamttest**
  Irgendwann muss Ihr Programm in der Gesamtheit getestet werden. Stellen Sie auch hier schon während des Designs und der Programmierung sicher, dass diese Tests leicht durchzuführen, reproduzierbar und aussagekräftig sind. In der Praxis werden unterschiedliche Geschmacksrichtungen dieser Tests vorkommen. Hier einige Stichworte, die aber bei Weitem nicht alle Testarten abdecken:
  - Ein *Smoke-Test* überprüft »auf die Schnelle« die Mindestfunktionalität. Mit einem *Massentest* testen Sie große Datenmengen, die möglichst alle kritischen Fälle abdecken sollen.
  - Ein *Integrationstest* findet meist auf einer eigenen Umgebung statt, die so nah wie möglich an dem wirklichen Einsatz der Software ist.
  - Bei einem *End-to-End-Test* ist Ihr Programm womöglich nur eine Komponente einer langen Kette, die von vorne bis hinten überprüft wird.
- **Unittest**
  Diese Art zu testen hat sich in vielen Softwareentwicklungsmethoden etabliert. Dabei testet der Entwickler persönlich die kleinsten Funktionseinheiten seines Programms (*Units*), bei C++ üblicherweise einzelne Funktionen und Methoden, weniger das Zusammenspiel der Komponenten (weswegen Unittests gute Funktionstests ergänzen, nicht ersetzen!). Unittests müssen schnell ausgeführt werden können, da der Entwickler sie üblicherweise jeden Tag mehrmals durchführt.
- **Refactoring**
  Wenn man seinen Code so verändert, dass sich seine Funktionalität nicht verändert, ist das Refactoring. Zum Beispiel reorganisiert man ihn, benennt Variablen um oder ergänzt Parameter. Häufig ist dies eine Vorbereitung auf eine notwendige Erweiterung: Eine momentan spezielle Funktion soll mit einem weiteren Parameter allgemeiner werden, oder eine an mehreren Stellen ausprogrammierte Funktionalität soll in eine Funktion ausgelagert werden und von diesen Stellen aufgerufen werden.

  Solche Änderungen sind oft über große Quelltextbereiche nötig, und es ist schwierig abzusehen, welche Konsequenzen eine Änderung hat. Das kann dazu führen, dass man sich nicht traut, seinen Code anzufassen – was wiederum die notwendige Erweiterung erschwert. Daher müssen Sie von Anfang an so entwickeln, dass Refactoring möglich ist, und das erreichen Sie am besten durch testbaren Code – siehe Unit- und Funktionstests. Ohne diese Kombination können Sie auf Dauer die Codequalität nicht hoch halten.

2.3 Entwurfsmuster

Im Vergleich zu den anderen Gewerken ist dasjenige der Softwareentwicklung noch sehr jung. Während man sich beim Hausbau auf »Sparren« und »Giebel« beziehen kann und jedem Fachmann klar ist, was gemeint ist, waren ähnliche Dinge beim Programmieren noch lange undefiniert. De facto haben aber alle Programmierer, die einen austauschbaren Treiber geschrieben haben, ähnliche Vorgehensweisen gewählt. Kommunikation zwischen Programmteilen? Wird häufig ähnlich implementiert. Ein Programm in Komponenten aufteilen? Das hat jeder schon gemacht.

Dies hat die »Gang of Four« um Erich Gamma erkannt und in dem Buch »Design Patterns: Elements of Reusable Object-Oriented Software« 1994 veröffentlicht. Darin wurde vielen Dingen ein Name gegeben, die Programmierer seit jeher sowieso schon gemacht haben. Die Methoden selbst sind (damals wie heute) nicht wirklich weltbewegend und wenig revolutionär. Der Grund, warum man dieses Buch beachten muss, ist, dass den Dingen darin ein *Name* gegeben wurde. Statt »dann habe ich dieses eine Objekt initialisiert, auf das alle zugreifen, und ich musste noch sicherstellen, dass alle es benutzen können, und…« zu erzählen, konnte der Programmierer nun einfach »Singleton« sagen und jeder wusste (in etwa) worum es geht und schon viel über die Rolle und Aufgabe des Konstrukts. Der Name sagt noch nichts über die konkrete Implementierung, aber nun konnte man austauschbare Implementierungen liefern, die zu großen Teilen mit einem Wort beschrieben werden konnten. Verschiedene Fachbereiche können sich mit den Namen von Entwurfsmustern – im Englischen *Design Patterns* genannt – über das Gleiche unterhalten, und auch der Austausch von Erfahrungen zwischen unterschiedlichen Programmiersprachen wurde einfacher.

Während die Entwurfsmuster des ursprünglichen Buchs heute vielfach modernisiert und ergänzt wurden, bleibt als Nutzen der Patterns an sich übrig, dass man ein gemeinsames Vokabular hat, mit dem man sich effizienter untereinander austauschen kann – sei es im Gespräch oder in der Dokumentation oder schon bei der Namensgebung der Klassen, Funktionen und Variablen. Dies ist der Grund, warum nach diesen Entwurfsmustern weitere aus dem Boden sprossen wie Pilze: *Enterprise Integration Patterns* für große Software-Infrastrukturen, *Architekturpatterns*, *Projektplanungspatterns* und so weiter und so fort. Selten stellt ein Entwurfsmuster aus irgendeinem dieser Bereiche etwas Neues dar, die Existenz ist aber wichtig, da es die Kommunikation verbessert und in seinem Bereich einen »Werkzeugkasten« an Methoden definiert, die man im Repertoire haben sollte – ob man sie nun verwendet oder nicht.

Für konkrete Entwurfsmuster möchte ich wieder auf andere Bücher verweisen. Sie sollten aber über ihre Existenz an sich sowohl Bescheid wissen als auch sich im Laufe der Zeit das Vokabular und die dazugehörigen Methoden der Patterns aus Ihrem Bereich aneignen.

2.4 Algorithmen

Von einer ganz anderen Seite als die Entwurfsmuster beleuchtet die Informatik das Programmieren. Sie beschäftigt sich eher mathematisch mit der Softwareentwicklung, befasst sich weniger mit konkretem Code und leistet doch praxisrelevante Arbeit – indem sie zum Beispiel Algorithmen entwickelt.

Was macht einen *Algorithmus* aus? Zunächst einmal muss er eine Lösung für ein gestelltes Problem darstellen. Ein Algorithmus ist noch kein Programm, er *beschreibt* dies eher. Algorithmen müssen eindeutig, verständlich und vollständig sein. Damit das leicht fällt, bietet es sich an, eine formale Sprache zu verwenden (zum Beispiel mathematische Formeln oder eine der Programmierung ähnliche Sprache). Das ist aber nicht unbedingt nötig; Sie können, wenn Sie präzise sind, auch eine natürliche Sprache nehmen.[1]

Zu einem Algorithmus gehören immer die folgenden Punkte:

- **Eingabe**
  Definieren Sie, was genau Ihr Programm verarbeitet.
- **Ausgabe**
  Beschreiben Sie, was Ihr Programm produziert.
- **Ablauf**
  Schreiben Sie auf, was getan werden soll.

Die einzige formale Anforderung an diese Elemente ist, dass ihre Beschreibung *endlich* sein muss. Es mag ein dicker Wälzer sein, eine ganze Enzyklopädie oder auch nur ein einzelner Satz – Sie müssen mit dem Aufschreiben nur jemals fertig werden können. Jeder Algorithmus legt fest, wie seine Eingabe zu seiner Ausgabe gemacht wird.

In der Praxis werden Sie viele Algorithmen sehen, die eine eher kleine Aufgabem beschreiben, und die eigentliche Beschreibung nimmt meist weniger als eine Buchseite ein – mit Erklärungen vielleicht ein Buchkapitel.

Ein Algorithmus sollte natürlich irgendwann mit seiner Berechnung fertig werden, und zwar für jede mögliche Eingabe. Das ist in der Praxis wichtig, damit man garantiert irgendwann sein Ergebnis bekommt.[2]

2.5 Ressourcen

Aus der Beschreibung des Algorithmus folgt, wie *aufwendig* es ist, ihn auszuführen. Dabei ist die *Größe der Eingabe* häufig entscheidend. Wenn Sie einen Satz mit 100 Zeichen in Ihren Algorithmus hineingeben oder ein Buch mit 1000 Seiten, wie viele Ressourcen verbraucht Ihr Algorithmus?

Die wichtigsten Ressourcen sind dabei *Zeit* und *Speicher*. Beides wohlgemerkt nicht unbedingt konkret in Bits, Bytes oder Sekunden und Tagen, sondern in einer abstrakten Einheit im Verhältnis zur Größe der Eingabe n.

Es ist etwa sehr häufig nötig, Elemente zu sortieren. Es gibt viele Algorithmen zum Sortieren, und sie unterscheiden sich unter anderem in ihrem Ressourcenverbrauch. Zwei extreme Beispiele sind *Heapsort* und *Bubblesort*. Sie unterscheiden sich in der Anzahl der Operationen (Vergleiche, Vertauschungen), die sie auf der Eingabe durchführen. Während Heapsort eine Eingabe mit n Elementen in $O(n \log n)$ Schritten sortiert, benötigt Bubblesort dafür $O(n^2)$.

1 *Algorithms*, Sedgewick, Wayne, Addison-Wesley 2001
2 Für theoretische Überlegung kann es in gewissen Bereichen sinnvoll sein, terminierende und nicht-terminierende Algorithmen zu unterscheiden.

Diese *O-Notation* ist eine Angabe dafür, wieviel Ressourcen ein Algorithmus typischerweise verbraucht. Die dort angegebene mathematische Funktion beschreibt eine Kurve – je schneller sie wächst, desto größer der Ressourcenverbrauch bei wachsender Eingabe. Wenn Sie sich unter $n \log n$ und n^2 nichts vorstellen können, sehen Sie in Tabelle 2.1 konkrete Werte.

Eingabegröße	Heapsort Schritte	–	Zeit ca.	Bubblesort Schritte	–	Zeit ca.
10	33	–	0 Sek.	100	–	0 Sek.
100	664	–	0 Sek.	10.000	–	0 Sek.
1000	9965	–	0 Sek.	1.000.000	–	1 Sek.
10.000	132.877	–	0 Sek.	100.000.000	–	100 Sek.
100.000	1.660.964	–	1 Sek.	10.000.000.000	–	3 Std.
1.000.000	19.931.568	–	19 Sek.	1.000.000.000.000	–	12 Tage

Tabelle 2.1 Die Laufzeit von Heapsort und Bubblesort bei unterschiedlichen Eingabegrößen, wenn der Computer etwa 1.000.000 Operationen pro Sekunde schafft

Die Wahl des richtigen Algorithmus ist enorm wichtig. Ich habe das Sortieren als Beispiel gewählt, weil diese Aufgabe so häufig vorkommt, man dabei besonders viel verkehrt machen kann, und weil Sie, als angehender C++-Nutzer, das Problem nahezu als »gelöst« betrachten können. Sie sollten normalerweise nicht versuchen, einen Sortieralgorithmus selbst zu entwickeln. Nehmen Sie das, was die Sprache C++ Ihnen anbietet: die Funktion `std::sort()`. Sie basiert auf Heapsort und ist bis auf ganz wenige Ausnahmen das Mittel der Wahl für Ihre Sortieraufgaben.

Was für das Sortieren gilt, gilt auch für viele andere Algorithmen. Wenn Sie ein Problem lösen wollen, dann schauen Sie zuerst in der Standardbibliothek von C++ nach, bevor Sie es selbst programmieren (außer zu Lernzwecken). C++ hat für viele Algorithmen verlässlich programmierte Implementierungen, die oft sogar theoretisch optimal sind. Und eine der großen Stärken von C++ ist, dass die Algorithmen auf beinahe allen Daten arbeiten können.

Kapitel 3
Programmieren in C++

C++ ist eine Programmiersprache für viele Zwecke. Generell kann man sie für nahezu alles einsetzen. Durch ihren Fokus auf Performance und Interoperabilität findet man sie häufig in der Systemprogrammierung. Betriebssysteme, Treiber und andere maschinennahe Programme sind besonders häufig in C++ geschrieben.

In der Klassifizierung der vielen existierenden Programmiersprachen zeichnet sich C++ durch folgende Unterschiede und Eigenschaften aus:

- **C++ wird in Maschinencode übersetzt.**
  JavaScript wird interpretiert, Java in einen Zwischencode übersetzt, der dann interpretiert wird. C++ wird vom Compiler direkt in die Sprache übersetzt, die die Maschine spricht.[1]
- **C++ ist imperativ.**
  Die meisten Sprachen, von denen Sie gehört haben, fallen in diese Gruppe, denn sie gehen Zeile für Zeile vor. Berühmte nichtimperative Beispiele sind Scala, Haskell und F#. Wenn Sie SQL als Programmiersprache betrachten, dann fällt SQL ebenfalls nicht in diese Kategorie.
- **C++ ist objektorientiert.**
  Sie können Klassen und Vererbung verwenden. C ist in diesem Sinne nicht objektorientiert. JavaScript simuliert Objektorientierung mittels Prototypen.
- **C++ ist typsicher.**
  C ist in diesem Maße nicht typsicher, ebenso wenig wie Python oder JavaScript.
- **C++ ist parallel.**
  Spätestens seit C++11 ist die Sprache ohne Zweifel für das Abarbeiten mehrerer gleichzeitiger Programmpfade ausgelegt. In kaum einer anderen Sprache hat man sich so viele Gedanken über das Zusammenspiel mit den Fähigkeiten moderner Hardware gemacht wie im aktuellen C++. Selbst C99 und C++98 mussten Kompromisse eingehen, wenn es um Parallelität ging.
- **C++ ist generisch.**
  Sie schreiben mit Templates allgemeingültige Vorgehensweisen für mehrere Datentypen. Das ist in C schwerer.
- **C++ erlaubt Metaprogrammierung.**
  Sie können Programme schreiben, die zur Compilezeit ausgeführt werden.

1 Zumindest ist das der übliche Weg. Der Standard schreibt dies nicht vor, und es gibt C++-Varianten, die das anders machen.

- **C++ ist ISO-Standard.**
  Das heißt, ein weltweit internationales Komitee entscheidet über die Sprache. Hinter Java steht hauptsächlich Oracle, hinter Python die »Community«, in der der »wohlwollende Diktator auf Lebenszeit« Guido van Rossum das letzte Wort hat.

All dies hat den Erfinder von C++, Bjarne Stroustrup, zu der Aussage veranlasst, dass C++ eine *Multiparadigmensprache* ist. Er meint damit, dass Sie in C++ viele Möglichkeiten haben, ein Programm zu schreiben, und dass das Paradigma, das Sie einsetzen wollen, frei wählen können. Sie *können* parallel programmieren, müssen es aber nicht. Sie *können* typsicher sein, müssen es aber nicht. Sie *können* generische Datentypen verwenden, werden aber nicht dazu gezwungen.

3.1 Übersetzen

Wenn Sie ein C++-Programm schreiben, dann heißt das, Sie schreiben *Quellcode* als Text, den C++-Werkzeuge in ein ausführbares Programm übersetzen. Wir reden bei diesen Werkzeugen häufig vom *Compiler*, doch in Wirklichkeit sind damit mehrere Tools gemeint. Ich möchte in diesem Abschnitt präzise sein und Ihnen die Aufgaben der unterschiedlichen Werkzeuge nennen. Später werde ich sie wieder unter dem eigentlich nicht ganz richtigen Begriff »Compiler« zusammenfassen.

Diese unklare Benennung liegt unter anderem auch daran, dass heutzutage die Werkzeuge selten noch getrennte Programme sind. Häufig sind es nur noch *Phasen* eines einzigen Programms. Und tatsächlich spiegelt auch Abbildung 3.1 nur einen vereinfachten Ablauf wider. Außen vor habe ich die Optimierungen gelassen sowie die Tatsache, und dass ein Programm, wenn Sie es ausführen, noch zusätzliche Bibliotheken verwendet (dynamische Bibliotheken).

Dieser ganze Prozess wird entweder aus der *integrierten Entwicklungsumgebung* (IDE) angestoßen, oder man führt ihn von Hand auf der Kommandozeile aus. Wobei »von Hand« hier übertrieben ist, denn sehr häufig verwendet man auch hier ein Werkzeug, ein *Buildtool* (in etwa: Bauwerkzeug). Sehr verbreitet sind »Makefiles«. Das sind Textdateien, in denen steht, welche Komponenten zum Programm gehören.

Später im Kapitel werden wir sowohl ein kleines Programm mit einer IDE bauen als auch auf der Kommandozeile ausführen und uns dafür eines Makefiles bedienen.

3.2 Aktuelle Compiler

Kommen wir zurück zu dem, was meistens mit *Compiler* gemeint ist: die gesamte Werkzeugkette vom Präprozessor bis zum Linker. Hinzu kommt, dass die *Standardbibliothek* integraler Bestandteil von C++ ist. Wenn Sie also einen Compiler auf Ihrem System installieren, dann erhalten Sie auch immer eine Standardbibliothek dazu.

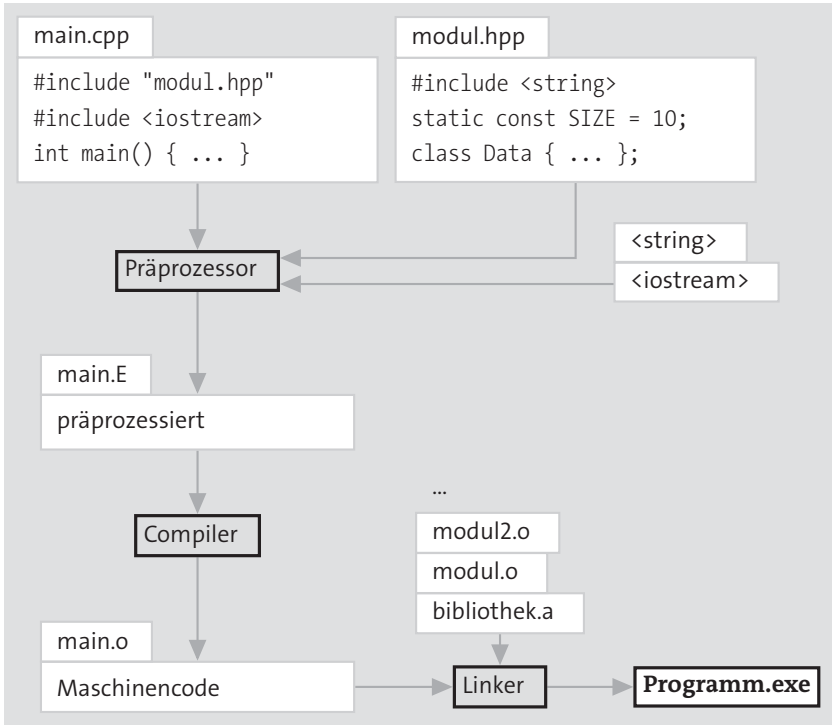

Abbildung 3.1 Die Phasen der Compilierung vom Quellcode zum Programm

Mit wohlgemeinter Absicht bezieht sich dieses Buch zu großen Teilen auf den aktuellen Standard, der unter dem Namen C++11 bekannt ist. Bei dieser Generalüberholung hat die Sprache viel Potenzial bekommen, um eine rundere, sicherere, konsistentere und nicht zuletzt auch einfacher zu erlernende Sprache zu sein. Sie, als Neuling, sind in der glücklichen Lage, dass Sie gleich mit diesen guten Features anfangen können. Wenn Sie die C++11-Möglichkeiten nutzen, werden Sie die Sprache schneller erlernen und besser einsetzen können, als das zum Beispiel noch mit der Vorläuferversion C++98 der Fall gewesen wäre.

C++ macht mit einem aktuellen Compiler auf jeden Fall mehr Spaß. Zum Glück sind die meisten Compiler mehr oder weniger auf dem neusten Stand.

Eine exakte Auflistung der Umsetzung des neuen Standards C++11 und des zur Drucklegung frisch verabschiedeten Standards C++14 in den diversen Produkten der Hersteller wäre müßig, dann fast wöchentlich ergänzen diese Features und beheben Fehler in ihren Compilern. Aber ein paar Anmerkungen zu einigen weit verbreiteten Produkten kann ich hier liefern, damit Sie wissen, was Sie erwartet.

3.2.1 Gnu C++

Der C++-Compiler g++ aus der *Gnu Compiler Collection* (GCC genannt) ist der auf den meisten Plattformen verfügbare Compiler. Er ist gleichzeitig die Experimentierwiese, um neue Dinge auszuprobieren, sodass Sie hier beinahe immer zuerst die neuen Features implementiert finden. Auf Linux ist GCC meist die erste Wahl. GCC ist zwar weit verbreitet, hat aber den Ruf, eine sehr komplexe Codebasis zu haben. Die kompilierten Programme fallen gegenüber kostenpflichtigen Compilern, was die Geschwindigkeit angeht, etwas zurück.

3.2.2 Clang++ der LLVM

Was die Codebasis angeht, hat der LLVM mit seinem C++-Compiler namens Clang++ einen besseren Ruf. Die Umsetzung der C++11- und C++14-Features ist vorbildlich. Clang++ ist der Standardcompiler für die MacOS-Entwicklung. Für Linux steht er kostenlos zur Verfügung, muss jedoch zu einer bestehenden Standardbibliothek hinzuinstalliert werden, sodass Sie den g++ am besten vorher installieren.

3.2.3 Microsoft Visual Studio

Die Compilersuite des Windows-Herstellers ist unter den hier aufgezählten Werkzeugen am wenigsten weit, was die Umsetzung des C++11-Standards angeht. Jedoch sind die meisten und wichtigsten Features enthalten. Sie sollten auch hier keine größeren Schwierigkeiten haben, jedoch die Unterschiede zu diesem Buch im Blick haben. Dafür, so der Dr. Dobbs-Blogger Gaston Hillar, habe »das Visual Studio Team die nützlichsten Features implementiert, und sich somit auch schon an die Umsetzung einiger C++14-Neuigkeiten gemacht«.[2]

Microsoft hatte zu der Zeit, als der C++11-Standard verabschiedet wurde, gerade alle Hände voll zu tun mit der Einführung der Metro-Oberfläche und des damit einhergehenden neuen Konzeptes zur asynchronen Programmierung. Der Fokus lag daher vermutlich eher auf den sogenannten »CLI«-Sprachen des Windows-Biotops und kam mehr .NET und C# zugute.

Es gibt eine kostenlose Express-Version, die Ihnen ausreichende Dienste leisten wird. Die professionellen Werkzeuge kosten Geld – außer wenn Sie Student oder Schüler sind, dann können Sie die Produkte gegen Nachweis Ihres Status kostenlos erhalten.

3.3 Entwicklungsumgebungen

Es gibt für Sie zwei hauptsächliche Möglichkeiten, C++-Programme zu entwickeln:

2 *More New C++ Features in VS2013*, Gaston Hillar, *http://www.drdobbs.com/240166013*, Dr. Dobbs 2014-02-11, [2014-02-11]

- **Kommandozeile**
  Sie arbeiten auf der Kommandozeile und rufen den Compiler und andere Werkzeuge von Hand auf. Später nutzen Sie dann Hilfsmittel wie Makefiles, um diese Aufgaben zu automatisieren. Ich empfehle, den Weg über die Kommandozeile zumindest auszuprobieren. Zum einen lernt man dabei auch andere nützliche Dinge über Programme und das Programmieren. Zum anderen lassen sich Abläufe auf der Kommandozeile besser automatisieren – und um das Automatisieren geht es uns beim Programmieren ja letzten Endes.
- **Integrierte Entwicklungsumgebung**
  Sie verwenden eine sogenannte IDE (*Integrated Development Environment*; dt. *Integrierte Entwicklungsumgebung*). Gerade im späteren Programmieralltag kann eine auf einen persönlich zugeschnittene IDE die Produktivität immens erhöhen. Auf der anderen Seite kann eine IDE einen Anfänger mit ihrer Feature-Flut auch erschlagen. Es gibt Assistenten, die den Einstieg zu beschleunigen versuchen. Ob das klappt, hängt von Ihrer Persönlichkeit ab. Wenn Sie mit der Kommandozeile absolut nicht vertraut sind, können Sie hiermit einen Versuch wagen.

In manchen Fällen gibt die Wahl des Compilers die Wahl der IDE vor. Wenn Sie sich für Microsoft entscheiden, dann geht das mit *Microsoft Visual Studio* einher (professionelle Kaufversion, kostenlos für Schüler und Studenten) oder mit *Microsoft Visual Studio Express* (kostenlose Version). Zur Drucklegung dieses Buches war die Version »2013 mit Update 2« beziehungsweise »2013 für Windows Desktop« aktuell. Die professionelle Version ist für Schüler und Studenten unter Vorlage eines Nachweises kostenlos. Die Express-Variante ist für jeden kostenlos. Auf sie werde ich mich im Verlauf dieses Buches beziehen, sie enthält die Version 12.0 des C++-Compilers. Sehen Sie sich auf der Webseite von Visual Studio[3] die Optionen an.

Auf dem Mac ist das von Apple gelieferte XCode der De-facto-Standard. Damit haben Sie die Wahl zwischen einer exzellenten IDE und einer Sammlung an Werkzeugen für die Kommandozeile. Bei Apple[4] können Sie diese herunterladen. Die aktuelle Version des C++-Compilers ist 5.1, basierend auf der Version 3.4 des LLVM-Backends.[5]

Sowohl unter Unix als auch unter Windows und auf dem Mac steht Ihnen als Alternative auch die *Gnu Compiler Collection* zur Verfügung. Der C++-Compiler heißt *g++*. Sie bedienen ihn in erster Linie von der Kommandozeile aus, er integriert sich aber auch in IDEs wie Eclipse mit CDT, Netbeans, KDevelop, Code::Blocks, dem Qt Creator und anderen. Manche von diesen Tools gibt es sogar für mehrere Plattformen. Wenn Sie mit dem g++ unter Windows entwickeln wollen, dann schauen Sie nach MinGW-Integration und ob Sie sie getrennt herunterladen müssen oder ob sie schon mitgeliefert wird (*Minimal Gnu for Windows*).

3 *http://www.visualstudio.com/downloads*
4 *https://developer.apple.com/xcode*
5 *https://en.wikipedia.org/wiki/Xcode*

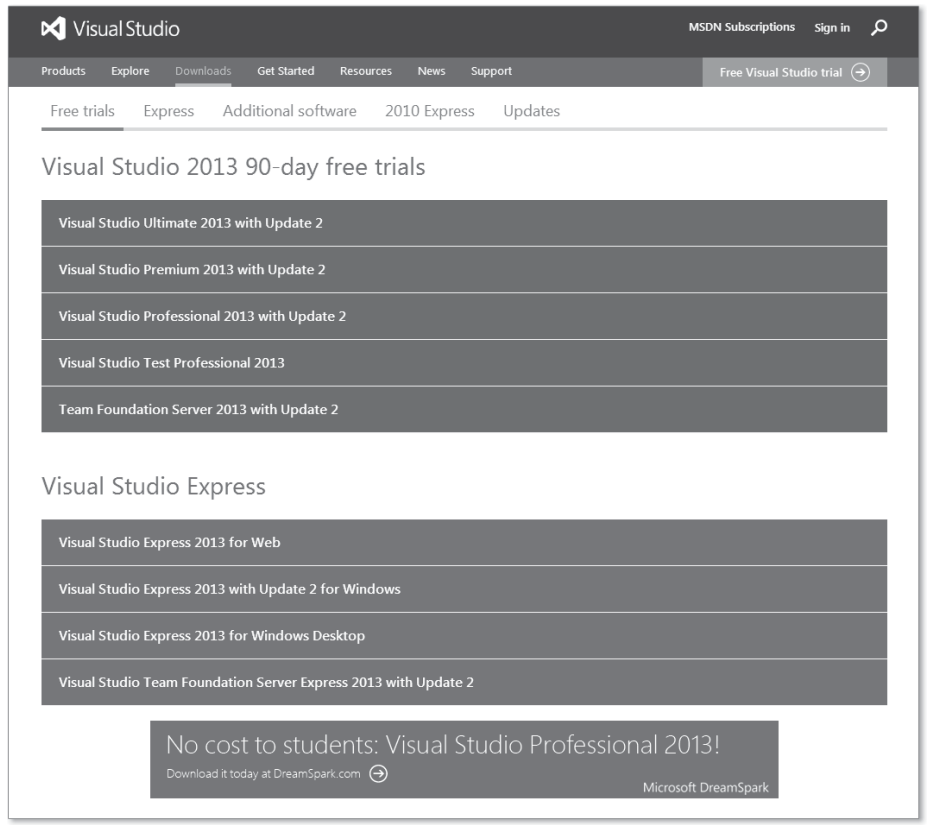

Abbildung 3.2 Aktuelle Microsoft-Produkte für die C++-Entwicklung

3.4 Die Kommandozeile unter Ubuntu

Exemplarisch für die Entwicklung mit der Kommandozeile gebe ich Ihnen eine Kurzanleitung für die aktuelle Langzeitversion von Ubuntu, einer weitverbreiteten Linux-Distribution. Wenn Sie ein anderes Linux verwenden, sind die Kommandos vielleicht unterschiedlich.

Sie installieren den g++ und einige nützliche Werkzeuge so:

```
sudo apt-get install g++ make
```

Den Programmcode geben Sie in einem Editor ein, und da beginnt die wirkliche Qual der Wahl. Wenn Sie eine IDE verwenden, dann ist der Editor mit dabei. Ohne IDE geben Sie Ihr Programm in einem beliebigen allgemeinen Texteditor ein. Texteditoren gibt es wie Sand am Meer.

Ich schlage hier nur drei vor, die unterschiedlichen Anforderungen gerecht werden: *jedit*, *gedit* und *kate*. Weil jedit in Java geschrieben ist, gibt es ihn wiederum auf allen Platt-

formen. Die Wahl zwischen gedit und kate sollten Sie abhängig davon fällen, ob Sie als Desktop Gnome oder KDE einsetzen. Probieren Sie einfach aus, was mit

```
sudo apt-get install gedit
sudo apt-get install kate
```

weniger Pakete automatisch installieren würde, und wählen Sie den Editor.

Wenn Sie in ein Team mit mehreren Entwicklern kommen, erkundigen Sie sich, ob *Emacs* oder *Vim* eingesetzt wird. Dabei handelt es sich um unter Programmierern sehr verbreitete Texteditoren, die aber eine steile Lernkurve haben. Wenn Sie Kollegen haben, die Ihnen beim Einstieg helfen, dann wählen Sie ruhig einen dieser beiden Editoren.

```
sudo apt-get install emacs
sudo apt-get install vim
```

3.4.1 Ein Programm erstellen

Wie gesagt, zur IDE kommen wir gleich, wenn wir exemplarisch das Microsoft Developer Studio unter Windows besprechen. Jetzt gehen Sie den Weg einmal zu Fuß.

Öffnen Sie eine Kommandozeile, manchmal auch Terminal oder Konsole genannt. Dazu finden Sie im Menü sicherlich einen Eintrag. Bei Ubuntu mit Gnome können Sie auch Strg + Alt + T drücken. Es sollte sich ein neues Fenster mit einem blinkenden Cursor öffnen, das eine Kommandozeile ähnlich wie diese zeigt, das sogenannte Prompt:

```
towi@havaloc:~$
```

Im weiteren Verlauf dieses Buches spare ich mir `towi@havaloc:`, das für Benutzer- und Rechnernamen steht, und meist auch Tilde ~ für das aktuelle Abeitsverzeichnis. Mit dem Prompt `$` meine ich, dass Sie dahinter ein Kommando eingeben sollen. Üben Sie einmal ein neues Verzeichnis zu erstellen und dieses zu betreten.

```
~$ mkdir quellcode
~$ cd quellcode
```

Nun sollte Ihr (gesamtes) Prompt das Verzeichnis beinhalten, in das Sie gewechselt sind:

```
~/quellcode$
```

Da es so viele Linux-Geschmacksrichtungen gibt, ist es durchaus möglich, dass Ihre Anzeige anders aussieht, obwohl Sie alles richtig gemacht haben. Sie können mit `pwd` überprüfen, in welchem Verzeichnis Sie gerade stehen.

Öffnen Sie nun den Editor Ihrer Wahl. Sie können das über die Menüs erledigen oder auf der Kommandozeile gleich den Namen der Datei angeben, die Sie bearbeiten wollen. Fügen Sie noch ein Ampersand `&` an, damit Sie trotz geöffnetem Editor weitertippen können (sollten Sie das vergessen, drücken Sie in der Kommandozeile Strg + Z und tippen Sie danach den Befehl `bg`, gefolgt von ↵, ein). Ich selbst bin ein Emacs-Nutzer, Sie setzen hier Ihren Lieblingseditor ein:


```
$ emacs modern101.cpp &
```

Tippen Sie den folgenden Quellcode in das Editorfenster. Sie sollten irgendwo erkennen, dass Sie wirklich `modern101.cpp` bearbeiten.

// modern101.cpp : Fibonacci-Konsole

```
#include <iostream>
#include <map>
int fib(int n) {
    return n<2 ? 1 : fib(n-2) + fib(n-1);
}
int main() {
    std::cout << "Die wievielte Fibonacci-Zahl? ";
    int n = 0;
    std::cin >> n;
    std::cout << "fib(" << n << ")=" << fib(n) << "\n";
}
```

Listing 3.1 Jede Fibonacci-Zahl ist die Summe der beiden Zahlen davor.

Speichern Sie und übersetzen Sie diesen Quellcode in das ausführbare Programm `modern101.x`:

```
$ g++ modern101.cpp -o modern101.x
```

Hier ist `g++` der Compiler. Mit `modern101.cpp` geben Sie die Quelldatei an. Haben Sie mehrere Quelldateien, die Sie zu einem Programm zusammensetzen wollen, geben Sie hier mehrere `*.cpp`-Dateien an. Mit `-o modern101.x` teilen Sie diesem Compiler den gewünschten Ausgabedateinamen mit. Wenn Sie den vergessen, ist das nicht schlimm, dann landet das fertige Programm bei g++ in `a.out`.

Probieren Sie es aus:

```
$ ./modern101.x
Die wievielte Fibonacci-Zahl? 33
fib(33)=5702887
```

Ihr erstes C++-Programm – herzlichen Glückwunsch!

Übrigens: Unter Windows werden Sie ausführbare Programme normalerweise mit der Endung `*.exe` versehen. Unter Linux ist eine Endung für ausführbare C++-Programme eher unüblich. Zur Verdeutlichung erzeuge ich hier Linux-Programme aber mit der Endung `*.x` – eine Praxis, die ich auch in der »wirklichen Welt« zuweilen pflege. Ob Sie mir das nachmachen oder nicht, bleibt Ihnen überlassen.

Wenn Sie interessiert, wie Sie das Programm beschleunigen können, blättern Sie zu Abschnitt 3.6, »Schneller«, vor.

3.4.2 Automatisieren mit Makefile

Da es aber mühselig ist, den Compiler auf diese Art immer wieder aufzurufen, erstellen Sie sich am besten ein Makefile, in dem die nötigen Befehle verzeichnet sind. Starten Sie dazu wieder einen Editor oder wählen Sie Datei • Neu im Menü, und speichern Sie danach die Datei unter `Makefile`:

```
$ emacs Makefile &
```

Der Inhalt der Datei ist dann:

```
# -*- Makefile -*-
all: modern101.x
modern101.x: modern101.cpp
  →  g++ modern101.cpp -o modern101.x
# aufräumen:
clean:
  →  rm -f *.x *.o
```

Die Kommentarzeilen mit `#` sind nicht essenziell. Achten Sie *unbedingt* darauf, dass die eingerückten Zeilen nicht mit Leerzeichen, sondern einem *Tabulator* anfangen, das deute ich hier mit → an. Auf alle Details gehe ich hier nicht ein, aber die beiden Zeilen

```
modern101.x: modern101.cpp
  →  g++ modern101.cpp -o modern101.x
```

sagen `make`: Wenn du `modern101.x` erstellen sollst, dann benötigst du dazu `modern101.cpp`; um es zu erstellen, führe den Befehl `g++ modern101.cpp -o modern101.x` aus.

Wenn Sie nun

```
$ make
```

ausführen, dann wird bei der `all`:-Regel nachgeschaut, was Sie alles gebaut haben wollen. Dort können Sie auch mehrere Programme auflisten, die `make` dann nacheinander erstellt. Nun sollten Sie wieder Ihr Programm gebaut bekommen. Eventuell merkt `make`, dass sich nichts geändert hat, dann hilft ein `make clean` (Aufräumregel ausführen) oder `make -B` (tu so, als hätte sich alles geändert).

Sie können auch mit `make all`, `make modern101.x` oder `make clean` eine der anderen Regeln ausführen lassen. Das ist alles schon sehr praktisch.

3.5 Die IDE »Microsoft Visual Studio Express« unter Windows

Nach dem Download,[6] der Installation und Registrierung von Microsoft Visual Studio Express begrüßt die IDE Sie mit einem Startbildschirm. In IDEs dreht sich meist alles um *Projekte*.

6 *http://www.visualstudio.com/downloads*

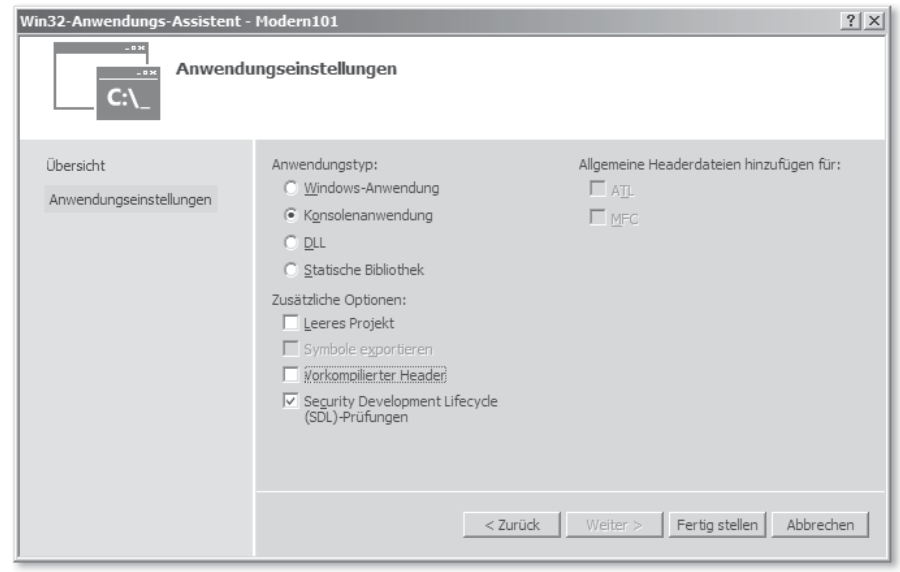

Abbildung 3.3 So erstellen Sie das Grundgerüst einer Konsolenanwendung.

- Beginnen Sie mit Datei • Neues Projekt...
- Wählen Sie in dem Assistenten Installiert, Vorlagen, Visual C++, Win32, Win32-Konsolenanwendung. Beachten Sie, dass Sie später hier auch Allgemein, Makefile-Projekt statt Win32 wählen können.
- Tragen Sie als Name `Modern101` ein.
- Überprüfen Sie, ob Ihnen der Ort zum Speichern gefällt. Der Haken Projektmappenverzeichnis erstellen sollte gesetzt bleiben.
- Klicken Sie aus Ok, um zum eigentlichen Assistenten zu gelangen.
- In der Übersicht des erscheinenden Win32-Anwendungs-Assistenten klicken Sie auf Weiter.
- Sie gelangen zu den Anwendungseinstellungen (siehe Abbildung 3.3).
- Stellen Sie sicher, dass als Anwendungstyp Konsolenanwendung ausgewählt ist.
- In den Zusätzlichen Optionen lassen Sie Leeres Projekt und Vorkompilierte Header frei, SDL-Prüfungen können Sie gesetzt lassen.
- Nach dem Klick auf Fertig stellen erstellt die IDE Ihnen das Grundgerüst des Programms.

Sie erhalten ein Grundgerüst von Projekt-, Quell-, Header- und Dokumentationsdateien.

Sie können zwar mit dem vom Assistenten erstellten Quellcode beginnen, doch enthält der das plattformspezifische `_tmain` statt `main` und die Dateien `stdafx.h` und `stdafx.cpp`. Für dieses Buch müssen wir portabler sein, und so ersetzen Sie im Editor den Inhalt der Datei `Modern101.cpp` durch den Text aus Abbildung 3.4.

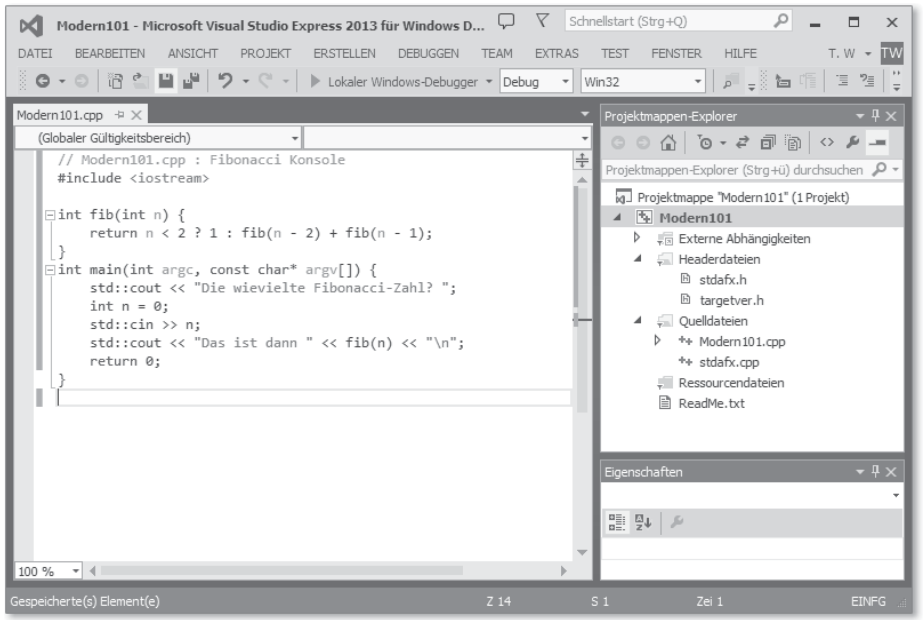

Abbildung 3.4 Das Grundgerüst einer Win32-Konsolenapplikation

Wählen Sie den Menüpunkt DEBUGGEN • STARTEN OHNE DEBUGGING, und bestätigen Sie den Bau des Programms mit JA. Wenn alles geklappt hat (und Sie sich beim Abtippen nicht vertan haben), dann erscheint nach kurzer Übersetzungszeit ein schwarzes Konsolenfenster und erwartet von Ihnen die Eingabe einer Zahl. Herzlichen Glückwunsch, Sie haben Ihr erstes C++-Programm geschrieben. Geben Sie 42 ein, und die Antwort 433494437 sollte nach einiger Zeit auf dem Bildschirm erscheinen (siehe Abbildung 3.5).

Abbildung 3.5 Ein selbst erstelltes Konsolenprogramm unter Windows

Auf meinem Computer dauert die Berechnung etwa eine Minute. Sie können auch mit einer kleineren Zahl wie 20 starten. Wenn Sie eine größere Zahl als 45 wählen, sprengen Sie die Fähigkeiten des Programms und bekommen unsinnige Ausgaben.

Ein Hinweis: Die Zahlenbereiche beziehen sich auf eine Win32-Applikation auf einem 64-Bit-Windows. Auf anderen Plattformen mögen die Grenzen andere sein.

3.6 Schneller

Wenn Ihnen das Programm zu lange läuft, dann *tabulieren* Sie die Zwischenergebnisse. Dann erhalten Sie die Ergebnisse schneller, als Sie gucken können. Erstellen Sie ein neues Projekt, oder modifizieren Sie das Programm:

```
// modern102.cpp : Fibonacci-Konsole
#include <iostream>
#include <map>
int fib(int n) {
    static std::map<int, int> table{};
    table[n] = n<2 ? 1 : table[n-2] + table[n-1];
    return table[n];
}
int main() {
    std::cout << "Wie viele Fibonacci-Zahlen? ";
    int n = 0;
    std::cin >> n;
    for (int i = 0; i <= n; ++i)
        std::cout << "fib(" << i << ")=" << fib(i) << "\n";
}
```

Listing 3.2 Eine zügig erstellte Tabelle von Fibonacci-Zahlen

Wenn Sie hier zum Beispiel 50 eingeben, sehen Sie, dass die Ergebnisse ab 46 auch mal negativ sind – ein Zeichen für einen Überlauf.[7] Das heißt, die Zahlen werden für dieses Programm zu groß und das Programm macht »nicht mehr was es soll.« Einen Überlauf in einem Programm zu haben, ist normalerweise keine gute Idee. Daher werden Sie in Kapitel 8, »Eingebaute Typen«, lernen, worauf Sie achten müssen und wie Sie sie vermeiden.

3.7 Aufgaben

Wiederholungsfrage

Erstellen Sie auf jeden Fall ein in diesem Kapitel beschriebenes Beispielprojekt.

Vertiefungsfrage

Probieren Sie selbst herum.

- Sehen Sie sich viele Beispielprogramme an.
- Googeln Sie, durchstöbern Sie Foren und die Hilfe Ihrer IDE.

7 Wieder: Auf anderen Plattformen als 32-/64-Bit-Windows-7 haben Sie vielleicht andere Grenzen.

- Schauen Sie sich auf *Stackoverflow*[8] um.
- Tauschen Sie mit Freunden, Kollegen und mir Erfahrungen aus.

Erweiterungsfragen

Die Zahlenbereiche beziehen sich wieder auf eine Win32-Applikation eines 64-Bit-Windows. Auf anderen Plattformen mögen die Grenzen andere sein.

1. Listing 3.2 liefert Ihnen nur korrekte Ergebnisse, wenn Sie es tatsächlich in einer Schleife starten lassen, die bei null beginnt. Wenn Sie direkt `fib(20)` anfordern, ohne zuvor `fib(18)` und `fib(19)` berechnet zu haben, stimmt `table` nicht. Können Sie dieses Problem beheben?
2. Verändern Sie Listing 3.2 so, dass es mehr Fibonacci-Zahlen als bis 45 berechnen kann.
   - Nehmen Sie dafür `unsigned long long` anstatt `int`. Was ist das größte korrekte Ergebnis, das Sie erhalten?
   - Was passiert, wenn Sie `double` (oder `long double`) nehmen?

8 *http://stackoverflow.com*

Kapitel 4
Ein ganz schneller Überblick

Kapiteltelegramm

- `main`
  Der Einstiegspunkt in jedes Programm
- `#include`
  Einbinden anderer Programmteile und Bibliotheken
- **Variable**
  Name für einen Speicherbereich, der einen Wert aufnehmen kann
- **Initialisierung**
  Der Wert, den eine Variable bei ihrer Entstehung haben soll
- **Zuweisung**
  Die Veränderung des Inhalts einer Variablen mittels `=` (engl. *Assignment*)
- `return`
  Das Verlassen einer Funktion; in `main` das Ende des Programms
- **Kommentar**
  Anmerkungen des Programmierers, die der Compiler nicht auswertet
- **Anweisung**
  Ein Programm ist die prinzipiell sequenzielle Abarbeitung unterschiedlicher Anweisungen (engl. *Statements*).
- **Ausdruck**
  Eine Folge von Operationen auf Operanden für Zuweisungen oder Ähnliches (engl. *Expression*)
- **Block**
  Eine Gruppe von Anweisungen zwischen geschweiften Klammern
- **Typ**
  Für den Compiler hat jeder Ausdruck einen Typ.

In diesem Kapitel machen wir einen ganz schnellen »Rundflug« über ein einfaches C++-Programm. Dadurch lernen Sie die wichtigsten Elemente kennen und verstehen sie besser, wenn wir im nächsten Kapitel mehr Dinge erklären.

Hier ist also ein einfaches C++-Programm.

```
#include <iostream>                          // Module/Bibliotheken einbinden
int main()                                   // main() ist der Beginn des Programms
{
    int wert = 100;                          // Variable mit Anfangswert
    std::cout << "Teiler von " << wert << " sind:\n";      // Ausgabe von Text
    for(int teiler=1; teiler <= wert; teiler = teiler+1) // Schleife von 1 bis 100
    {                                        // hier beginnt der Wiederholungsteil
        if(wert % teiler == 0)               // Test für eine bedingte Ausführung
            std::cout << teiler << ", ";     // nur bei positivem Test
    }                                        // Ende der Schleife
    std::cout << "\n";                       // einmalige Ausgabe
    return 0;                                // bedeutet in main() Programmende
}                                            // Ende von main()
```

Listing 4.1 Ein sehr einfaches C++-Programm

Wenn Sie dieses Programm übersetzen und laufen lassen, dann erhalten Sie die Ausgabe

```
Teiler von 100 sind:
1, 2, 5, 10, 20, 25, 50, 100,
```

auf dem Bildschirm. An diesem einfachen Programm können Sie schon viele grundlegende und wichtige Dinge von C++ sehen.

4.1 Kommentare

Wie Sie sehen, habe ich hier Programmtext und erklärende Worte gemischt. Die Zeilen beginnen immer mit Programmtext, dann folgt manchmal ein Doppel-Schrägstrich `//`, und dann kommen die erklärenden Worte – der *Kommentar*. In C++ können Sie hinter `//` beliebigen Text schreiben, der Compiler ignoriert diesen (oder, um genau zu sein, interpretiert ihn ähnlich wie ein Leerzeichen) bis auf wenige Ausnahmen. So können Sie anderen Programmierern, sich oder der Nachwelt Ihre Intentionen mitteilen, die zu der aktuellen Programmzeile führten.

4.2 Die »include«-Direktive

Die allererste Zeile des Beispiels lautet:

```
#incude <iostream>
```

Mit `#include` machen Sie dem Compiler bekannt, dass Sie Elemente eines Moduls in dieser Datei verwenden wollen. Der Name zwischen den Klammern ist der Name einer *Headerdatei*, in der sich die Deklarationen jenes Moduls befinden.

4.3 Die Standardbibliothek

Die spezielle Datei `iostream` binde ich ein, weil sich in ihr `std::cout` befindet. Die benötigt das Programm, um die Bildschirmausgaben zu erzeugen. Sie ist Teil der *Standardbibliothek* und wird mit dem Compiler mitgeliefert.

Alle Namen der Standardbibliothek beginnen mit `std`, gefolgt vom *Bereichsauflösungsoperator* `::` (engl. *Scope Resolution Operator*). Das ist ein grässlicher, wenn auch präziser Begriff, den niemand verwendet – es tut auch Doppel-Doppelpunkt (engl. *Double Colon*).

4.4 Die Funktion »main()«

Nun sollte der Blick auf die Zeile fallen, in der `main` steht:

```
int main()
```

Dies definiert eine *Funktion* mit einem besonderen Namen. Die `main`-Funktion ist immer der Einstiegspunkt in ein C++-Programm – es geht nicht ohne, und es kann nie zwei geben. Wenn Ihr System das Programm ausführt, dann wird `main()` aufgerufen werden.

Ansonsten bedeutet eine Funktion in C++, dass Sie an eine andere Stelle des Programms springen und später wieder hierhin zurückkehren. Funktionen können Argumente entgegennehmen – das sind die *Funktionsparameter* – und ein Ergebnis zurückgeben (siehe Kapitel 11, »Funktionen«).

Konkret lesen Sie die Definition dieser `main`-Funktion so:

- `main` soll eine Zahl zurückgeben – einen Wert vom Typ `int`, um genau zu sein.
- Der Name der Funktion ist `main`.
- Das leere runde Klammerpaar `()` bedeutet, dass die Funktion keine Parameter erhält.
- Dann folgt, was die Funktion eigentlich macht. Dieser *Funktionskörper* steht immer zwischen zwei geschweiften Klammern `{...}`.

Je nach Betriebssystem kann der Rückgabewert ausgewertet werden. Soll Ihr Programm Argumente auf der Kommandozeile bekommen können, dann wären die runden Klammern für die Parameter nicht leer. Sie sehen später, was Sie dafür anstellen müssen.

4.5 Typen

In C++ hat fast alles einen Typ, zum Beispiel Variablen und Zwischenergebnisse. Der Typ legt fest, welche Eigenschaften das Konstrukt hat und welche Werte es aufnehmen kann.

In Listing 4.1 wird nur `int` als Typ konkret genannt. Alles andere erschließt sich der Compiler selbst. Im Programm werden mit `int wert` und `int teiler` zwei *Variablen* mit diesem Typ eingeführt. Immer, wenn Sie sich im Verlauf Ihres Programms auf diese Variablen beziehen, müssen Sie deren Typ `int` berücksichtigen. Auf die genauen Eigenschaften von `int` werde ich noch eingehen. Hier sei es ausreichend, dass `int` für eine »Ganzzahl« steht.

4.6 Variablen

Eine *Variable* ist der Name für einen Speicherbereich, der einen Wert aufnehmen kann. Ja, in C++ muss eine Variable immer einen Typ haben. In dem Moment, wenn Sie eine Variable das erste Mal verwenden, müssen Sie dem Compiler ihren Typ mitteilen. Der Typ begleitet die Variable, solange sie lebt, und kann nicht mehr geändert werden.

Im Programm verwende ich zwei *Variablen*. `wert` repräsentiert die Zahl, deren Teiler ich ausgebe, und `teiler` verwende ich, um diese nacheinander alle zu prüfen. Zunächst definiere ich `wert`. Ab dem Zeitpunkt kann ich `wert` verwenden, was auch gleich für die Ausgabe geschieht:

```
int wert = 100;                                  // Variable mit Anfangswert
std::cout << "Teiler von " << wert << " sind:\n"; // Ausgabe von Text
```

Weil `wert` vom Typ `int` ist, können Sie sie nur für Ganzzahlen verwenden – das heißt sie in Berechnungen verwenden oder verändern.

Die andere Variable ist `teiler` in der `for`-Schleife:

```
for(int teiler = 1; teiler <= wert; teiler = teiler+1) // Schleife von 1 bis 100
```

Für sie gilt Ähnliches wie für `wert` – außer dem Namen gibt es zwei Unterschiede:

- `teiler` wird tatsächlich im Programmablauf verändert: Bei `teiler = ...` wird ihr eine neue Zahl zugewiesen.
- Sie ist nur innerhalb von `for` bekannt.

Der *Gültigkeitsbereich* einer Variablen (engl. *Scope*) beschränkt sich auf ihren Block, danach ist sie buchstäblich »weg«. Und zwar so »weg«, dass Sie außerhalb des `for` eine *neue* andere Variable `teiler` definieren könnten. Die kann dann auch einen ganz anderen Typ haben. In Listing 12.4 finden Sie ein Beispiel.

4.7 Initialisierung

Das Gleichheitszeichen `=` erfüllt im Beispielprogramm zwei Zwecke, die oft miteinander vermischt werden. Weil die Unterscheidung aber so wichtig ist, möchte ich Sie schon früh dafür sensibilisieren.

Sie sehen im Beispiel die folgenden Gleichheitszeichen:

```
int wert = 100;
int teiler = 1;
teiler = teiler+1;
```

Die ersten beiden Zeilen sind jeweils die *Initialisierung* einer im gleichen Atemzug definierten Variablen. Dieser Zeitpunkt, zu dem Sie auch ihren Typ festlegen, ist ihre *Deklaration*. Nur bei der Deklaration können Sie etwas initialisieren.

In der letzten Zeile ist die Variable schon deklariert. Somit weisen Sie einer bestehenden Variablen einen *neuen* Wert zu – daher sprechen wir von einer *Zuweisung*. Bei einer Zuweisung können Sie den Typ der Variablen nicht ändern. Sind die Typen unterschiedlich, kann der Compiler in gewissen Grenzen eine Konvertierung vornehmen.

4.8 Ausgabe auf der Konsole

Mit dem `#include <iostream>` haben Sie den Teil der Standardbibliothek importiert, der für die Ein- und Ausgabe zuständig ist. Die Ausgabe auf die Konsole geschieht mittels des Operators `<<`.

```
std::cout << teiler << ", ";
```

Links steht mit `std::cout` die aus `<iostream>` stammende Variable, die für die Ausgabe auf der Konsole steht. Rechts von jedem `<<` stehen die Dinge, die Sie ausgeben wollen. Wie Sie sehen, können Sie `<<` ähnlich verketten wie ein normales Plus `+` und somit mehrere Dinge nacheinander ausgeben.

4.9 Anweisungen

Die geschweiften Klammern `{...}` von `main()` halten eine Gruppe von *Anweisungen* zusammen – sie definieren einen *Anweisungsblock*. Sie bilden die Begrenzung dessen, was für `main()` ausgeführt wird:

```
int main()
{
    ...
}
```

Dazwischen stehen Anweisungen, die nacheinander ausgeführt werden (engl. *Statements*). Anweisungen sind wichtige Grundelemente in C++, und es gibt unterschiedliche Arten davon. Zu erkennen, was eine Anweisung ist und welcher Art sie ist, wird Sie mit C++ schnell vorwärts bringen. In den nächsten Kapiteln werden Sie alle kennenlernen. An dieser Stelle zeige ich Ihnen, was Sie in Listing 4.1 für Anweisungen finden:

- Die *Deklaration* `int wert = 100;` macht die Variable `wert` bekannt und initialisiert sie mit einem Anfangswert – zusammengenommen manchmal *Initialisierungsanweisung* genannt.
- Bei `cout << "Teiler von " << wert << " sind:\n";` handelt es sich um einen *Ausdruck*, der etwas auf der Konsole ausgibt.
- Dann folgt eine *for-Schleife*. Sie wird verwendet, um andere Anweisungen wiederholt auszuführen. Ich gehe später genauer auf die `for`-Anweisung ein, hier achten Sie bitte auf die Besonderheit, dass der Teil, der wiederholt werden soll, wieder in *geschweiften Klammern* `{...}` hinter dem `for` steht.
- Denn die beiden Klammern, die zum `for` gehören, sind mit ihrem Inhalt eine *zusammengesetzte Anweisung* oder auch ein *Anweisungsblock*. Darin sind wieder eine Serie

von Anweisungen enthalten, die von den umschließenden Klammern zusammengehalten werden. Diese Gruppierung von Anweisungen hat in mehrerlei Hinsicht eine besondere Bedeutung. Einerseits können sie so gemeinsam durch die `for`-Schleife wiederholt werden, und andererseits bildet diese Gruppierung einen *Sichtbarkeitsbereich* für darin enthaltene Variablen.

- Bei `if(wert % teiler == 0)...` handelt es sich im eine `if`-Anweisung, eine *Verzweigung*. Es wird eine Bedingung getestet, und die dann folgende *Anweisung* `std::cout << teiler << ", ";` wird nur ausgeführt, wenn diese Bedingung wahr ist. Wie bei der `for`-Schleife des Beispiels hätten wir hier auch einen Anweisungsblock in `{...}` folgen lassen können.[1] Zur Demonstration folgt dem `if` nur eine einzelne Anweisung. So konnte ich mir die umgebenden `{...}` für einen Anweisungsblock sparen.
- Die *Return-Anweisung* `return 0;` schließt diese Aufzählung ab.

4.10 Aufgaben

Wiederholungsfragen

1. Probieren Sie Listing 4.1 mit anderen Zahlen als 100 aus.
2. Was ist in Listing 4.2 falsch? Korrigieren Sie es so, dass es die Ergebnisse der Rechnungen ausgibt, auch wenn Sie die Werte der Variablen verändern würden.

```
#include <iostream>
int main()
    int a = 20;
    int b = 30;
    std::cout << "20+30 ist " << (a+b) << "\n";
    int a = 2;
    int b = 3;
    std::cout << "2*3 ist " << (a*b) << "\n";
```

Listing 4.2 Was ist in diesem Listing falsch?

Vertiefungsfragen

1. Stört Sie das abschließende Komma `,` in der Ausgabe von Listing 4.1 nicht auch? Modifizieren Sie das Programm so, dass Kommas nur zwischen Teilern ausgegeben werden. Und nicht schummeln: Das Programm soll immer noch eine korrekte Ausgabe liefern, auch wenn `wert` auf eine andere Zahl als `100` gesetzt wird. Vielleicht hilft Ihnen der Fakt, dass Sie den letzten Teiler schon im Voraus kennen?
2. Wenn Sie einen Teiler `n` von `wert` gefunden haben, dann kennen Sie auch einen zweiten `m`. Weil `(wert / n)` ohne Rest teilt, kommt `m` heraus, und `(wert / n) == m`. Somit ist `(n * m) == wert` und, wen wundert©s, auch `m` ein Teiler: `(wert / m) == n`. Schreiben

1 Das wäre guter Stil gewesen.

Sie Listing 4.1 so um, dass Sie jedes Mal beide Teiler ausgeben (wenn sie verschieden sind). Beachten Sie, bis wohin die Schleife laufen muss, damit Sie keine doppelten Zahlen ausgeben. Probieren Sie es zum Beispiel mit den folgenden `werten` aus: `100`, `101`, `103`, `96`, `64`, `256`.

Erweiterungsfrage

Verändern Sie das Programm in Listing 4.1 so, dass es nicht nur die Teiler einer einzelnen Zahl ausgibt, sondern schreiben Sie eine weitere Schleife – sodass `wert` von `1` bis `100` zählt und für alle diese Zahlen wie bisher jeweils die Teiler ausgibt.

Kapitel 7
Operatoren

Kapiteltelegramm

- **Operator**
  Meist ein Symbol, das zwischen zwei Operanden steht (oder vor einem); funktioniert wie eine Funktion mit den Operanden als Argumente
- **Operand**
  Argument für einen Operator
- **arithmetischer Operator**
  Dient zum klassischen Rechnen mit `+`, `-`, `*`, `/`, `%` sowie dem bitweisen Rechnen mit `|`, `&`, `~`, `<<` und `>>`
- **relationaler Operator**
  Größer-als, kleiner-als, gleich, Kombinationen davon und ungleich: `>`, `<`, `==`, `<=`, `>=`, `!=`.
- **logischer Operator**
  Verknüpft boolesche Werte: `&&`, `||` und `!`
- **Zuweisungsoperator oder zusammengesetzte Zuweisung**
  `=`, aber auch kombiniert mit einem der arithmetischen Operatoren
- **Binärsystem**
  Das Stellenwert-Zahlensystem des Computers mit Nullen und Einsen

Sehr häufig bestehen Ausdrücke aus ein- und zweistelligen Operatoren, wie zum Beispiel `3+4`, es könnte aber auch ein Ausdruck wie `!isBad && (x >= x0) && (x <= x1)` auftauchen. Mit solchen Aneinanderreihungen von Operatoren und Operanden können Sie in C++ eine Menge bewegen. Da Sie nun über Variablen, Typen und Ausdrücke eine Menge wissen, sollen Sie die möglichen Operatoren kennenlernen.

Exemplarisch erkläre ich Operatoren hauptsächlich anhand der Typen `int` und `bool`, damit Sie das Repertoire erst einmal kennenlernen. Aber viele Operatoren sind auch auf andere Typen anwendbar. Das sind durchaus eingebaute Typen, wie zum Beispiel `float`, aber auch solche der Standardbibliothek, wie `std::string` und `std::stream`.

In C++ können Sie eigene Typen definieren, die Operatoren ebenfalls unterstützen. Dass diese dann auch etwas machen, was für die eingebauten Typen gilt, liegt in Ihrer Hand. Wir erwarten zum Beispiel, dass `+` eine Addition ausführt – wie bei einem `int`. Die Klasse `string` verwendet `+` aber zur Konkatenation, was noch »additionsartig« ist. Aber Sie können, wenn Sie wollen, eine Klasse `Image` schreiben, die mit `+` auf `string` sich in eine Datei

speichert (Bitte tun Sie das nicht!). Das behandeln wir an geeigneter Stelle. Hier werden Sie zunächst erfahren, welche Operatoren es überhaupt gibt.

Operatoren für eingebaute Typen können nicht überschrieben werden

Wenn ich also in diesem Kapitel die Rolle der Operatoren beschreibe, dann meine ich die auf den eingebauten Typen. In C++ können Sie diese nicht verändern. Ein `+` für `int` gibt es schon, und Sie als Programmierer können dessen Bedeutung nicht verändern. An einigen Stellen gehe ich auf die Typen der Standardbibliothek ein, doch für vieles muss die Referenz herhalten.

7.1 Operatoren und Operanden

Ein *Operator* ist etwas, das sich so ähnlich verhält wie eine Funktion, ohne aber eine zu sein. Die meisten Operatoren schreiben Sie mit Symbolen, wie zum Beispiel `+` oder `<<`. Viele haben zwei Argumente und heißen deshalb zweistellig oder binär (engl. *Binary Operators*). Diese Argumente stehen dann als *Operanden* rechts und links vom Operator, wie in `3+4` oder `cout << name`. Es gilt also die Reihenfolge:

- **Operand Operatorsymbol Operand**

Wenn sie einstellig, unär, sind (engl. *Unary Operators*), steht der Operator vor dem Operanden wie in `-4` oder seltener auch danach wie in `idx++`:

- **Operatorsymbol Operand**
- **Operand Operatorsymbol**

Es gibt ein paar Ausnahmen. So ist zum Beispiel `sizeof()` eigentlich auch ein Operator. In diesem Kapitel präsentiere ich Ihnen aber vor allem die klassischen Operatoren.

7.2 Überblick über Operatoren

Man kann die Operatorsymbole in einige Gruppen einteilen:

- **arithmetische Operatoren**
  Dies sind die vier Grundrechenarten `+`, `-`, `*`, `/` sowie `%` für den Divisionsrest (Modulo). Das Vorzeichen können Sie mit den unären Operatoren `+` und `-` beeinflussen.
- **bitweise Arithmetik**
  Zahlen können Sie mit `|`, `&`, `^`, `~`, `<<` und `>>` bitweise miteinander verknüpfen.
- **Zuweisungsoperatoren**
  Neben dem `=` gibt es auch die zusammengesetzten Zuweisungen (engl. *Compound Assignments*) `+=`, `-=`, `*=`, `/=`, `%=`, `>>=`, `<<=`, `&=`, `^=` und `|=`.
- **Inkrement und Dekrement**
  Die beiden einstelligen Operatoren `++` und `--` gibt es jeweils in einer vorangestellten und einer nachgestellten Variante (Präfix und Postfix). Bevorzugen Sie möglichst die Präfixvariante, denn die kommt ohne temporäre Variable aus.
- **relationale Operatoren**
  Relationale Operatoren führen einen Vergleich aus und liefern einen Wahrheitswert `bool` zurück: `==`, `<`, `>`, `<=`, `>=` und `!=`. Wenn Sie mit der Standardbibliothek arbeiten, sind `==` und `<` die wichtigsten, denn viele Algorithmen benutzen nur diese beiden, um nötigenfalls die anderen herzuleiten. Das ist wichtig, wenn Sie eigene Datentypen für die Standardbibliothek fit machen wollen.
- **logische Operatoren**
  `&&`, `||` und `!` verknüpfen Wahrheitswerte zu komplexeren Ausdrücken.
- **Pointeroperator und Dereferenzierungsoperator**
  Mit den unären Operatoren `&`, `*` sowie den binären Operatoren `->` und `.` adressieren und dereferenzieren Sie. Das heißt, Sie holen eine Adresse, machen aus einer Adresse ein Datum oder greifen in eine Struktur hinein. Sie werden später den Einsatz im Detail sehen.
- **besondere Operatoren**
  Es gibt einen einzigen ternären (dreistelligen) Operator `? :`, der eine `if-else`-Abfrage als Ausdruck ermöglicht. Das Komma `,` kann als Sequenzoperator in Ausdrücken verwendet werden.
- **funktionsähnliche Operatoren**
  Streng genommen gehören auch einige Sonderlinge zu den Operatoren, die echte Namen haben und wie Funktionen verwendet werden. Das sind die Typumwandlungen wie `(int)wert` sowie `sizeof()` und einige andere.

7.3 Arithmetische Operatoren

Sie können in C++ ganz normal mit Zahlen rechnen. Neben den Grundrechenarten `+`, `-`, `*` und `/` gibt es noch `%` für den Divisionsrest. Wenn Sie keine Klammern verwenden, gilt Punkt- vor Strichrechnung.

```
#include <iostream>
int main() {
    std::cout << "3+4*5+6=" << 3+4*5+6 << "\n";            // Punkt- vor Strich; = 29
    std::cout << "(3+4)*(5+6)=" << (3+4)*(5+6) << "\n";    // Klammern; = 77
    std::cout << "22/7=" << 22/7 << " Rest " << 22%7 << "\n"; // 22/7 = 3 Rest 1
    for(int n=0; n < 10; ++n) {
        std::cout << -2*n*n + 13*n - 4 << " ";            // mit unärem Minus
    }
    std::cout << "\n";
    // Ausgabe: -4 7 14 17 16 11 2 -11 -28 -49
}
```

Listing 7.1 Arithmetische Operatoren in der Anwendung

Was hier für `int` gezeigt wurde, geht mit allen Ganzzahltypen. Und außer bei `%` geht es auch mit allen Fließkommatypen (`float` etc.). In der Standardbibliothek finden Sie `std::complex<>`, mit dem Sie ebenfalls diese Operatoren anwenden können.

Den Plus-Operator `+` verwenden viele Typen zum Zusammenfügen. Sie können zum Beispiel aus

```
std::string vor="Hans";
std::string nach="Huber";
```

mit `vor+" "+nach` den neuen String `"Hans Huber"` machen.

7.4 Bitweise Arithmetik

Die bitweise Arithmetik sieht wahrscheinlich zu Anfang etwas seltsam aus.

```
int a = 41;      // dezimale 41
int b = a & 15;  // ergibt 9
```

Die Erklärung ist, dass Zahlen im Computer ja als Folge von `0` und `1` dargestellt werden – eben als »Bits«. Die dezimale 41 ist in Bit-Darstellung `101001` – »binär«.

Binärsystem

Weil im Dezimalsystem 10 die Basis ist, schreiben wir »vierhundertzwölf« als 412 als Abkürzung für $4\times10^2+1\times10^1+2\times10^0$ = $_{10}412$. Für den Computer mit der Basis 2 ist das $1\times2^8+1\times2^7+0\times2^6+0\times2^5+1\times2^4+1\times2^3+1\times2^2+0\times2^1+0\times2^0$ oder abgekürzt $_2$1 1001 1100. Die unten stehende 10 beziehungsweise 2 illustriert, in welchem Zahlensystem die Zahl dargestellt ist.

Der Computer rechnet bei einem `int` oft mit 32 Bit. Daher füllt er vorne mit `0` auf, also `0000 0000 0010 1001`.

Operation	Binär	Dezimal
a	1001	9
b	0011	3
Und `a & b`	0001	1
Oder `a | b`	1011	11
Xor `a ^ c`	1010	10
Nicht ~b	1100	-4

Tabelle 7.1 Beispiel für bitweise Arithmetik

Die Arithmetik ist nun die bitweise Kombination der im Zweiersystem geschriebenen Zahlen. Wenn Sie sich die `1` als »wahr« denken und die `0` als »falsch«, dann können Sie bitweise die Operationen für *Und*, *Oder* und *Exklusiv-Oder* (*Xor*) selbst durchführen. Der Einfachheit halber beschränke ich mich für das Beispiel in Tabelle 7.1 auf 4 Bits.

Das bitweise Invertieren mit ~ ist etwas knifflig, denn da werden die aufgefüllten Nullen mit invertiert. Bei 4 Bits wird aus `~0011` dann `1100`, aber bei 32 Bits eben `111...1100`. Und da kommen Sie für `int` beim Computer in den Bereich der negativen Zahlen. Der Computer stellt diese im sogenannten *Zweierkomplement* dar – und daher interpretiert er diese lange Folge von Einsen als dezimal `-4`.

Die Operatoren `<<` und `>>` sollten Sie nun auch nicht mehr erschrecken: Schieben Sie die Bitdarstellung des ersten Operanden einfach um die Anzahl Bits des zweiten Operanden nach links oder rechts. Im Computer ist das identisch damit, als würden Sie ebenso oft mit Zwei multiplizieren oder dividieren:

- `345 << 3` ist wie `345*2*2*2` und ergibt `2760`.
- `345 >> 3` ist wie `345/2/2/2` und ergibt `43`.

Sobald Sie solche Zahlenmagie benötigen, müssen Sie sich mit der internen Zahlendarstellung im Computer ohnehin noch einmal beschäftigen. Bis dahin merken Sie sich, dass C++ diese Operationen hat.

Operatoren für Streams

Die Ein- und Ausgabedatenströme der Standardbibliothek (engl. *Streams*) verwenden die eigentlich für bitweise Arithmetik vorgesehenen Operatoren `<<` und `>>` zum Schreiben und Lesen. Das haben Sie für `std::cout` und `std::cin` schon in vielen Listings gesehen.

In C++ werden `<<` und `>>` heutzutage weit häufiger für Streams verwendet als für echte Bitarithmetik.

7.5 Zuweisungsoperatoren

Wenn Sie einen arithmetischen Operator anwenden, dann entsteht ein neuer Wert. Wenn der Ausdruck komplizierter als nur ein einzelner Operator ist, dann ist dieser nur ganz kurz – temporär – vorhanden. Es werden dann innerhalb eines Ausdrucks ständig neue »Temporarys« erzeugt und wieder verworfen. Um das zu vermeiden, gibt es alle arithmetischen Operatoren in Varianten, die stattdessen eine Variable direkt verändern.

Sie können statt `int a = 3; a = (a * 4 + 7 - 3)/4;` auch Folgendes schreiben:

```
int a = 3;
a *= 4;
a += 7;
a -= 3;
a /= 4;
```

In beiden Fällen enthält a dann den Wert 4. Sie sollten aber lange Rechnungen auf diese Weise vermeiden, denn der Code wird doch schnell unübersichtlich. Die Temporarys einzusparen ist nur in den seltensten Fällen wirklich ein großer Gewinn.

Auch die Operatoren der Bitarithmetik können Sie auf diese Weise anwenden.

```
#include <iostream>
#include <bitset>                       // hilft bei der Ausgabe von Zahlen als Bitfolge
int main() {
    int a = 0;
    for(int idx=0; idx<8; idx++) {
        a <<= 2;                        // um zwei Bits nach links schieben: "...100"
        a |= 1;                         // unterstes Bit setzen: "...1"
    }
    std::cout << std::bitset<16>(a) << "\n"; // 01010101010101
    std::cout << a << "\n";             // 21845
}
```

Somit sind die verfügbaren *zusammengesetzten Zuweisungen* (engl. *Compound Assignments*):

- die standard-arithmetischen +=, -=, *=, /= und %=
- und die binär-arithmetischen |=, &=, ^=, <<= und >>=

7.6 Post- und Präinkrement sowie Post- und Prädekrement

Zu den unären Operatoren ++ und -- habe ich eigentlich das Wichtigste schon gesagt. Ich fasse es noch einmal zusammen:

- Bei ++zahl wird zahl wird zunächst um eins erhöht, bei --zahl um eins erniedrigt. Das Ergebnis dieser Berechnung können Sie im umgebenden Ausdruck weiterverwenden. Weil die Operation hier zuerst ausgeführt wird, ist dies die »Prä«-Variante.
- Wenn Sie den Operator nachstellen, wenden Sie die »Post«-Variante an. Der Wert des Ausdrucks (zum Beispiel zahl++) ist dann der Wert der Variable vor der Veränderung. Sie wird dann erst am Ende der gesamten Anweisung wirklich ausgeführt. Bis dahin muss der Computer sich den neuen Wert irgendwo merken. Das verbraucht möglicherweise Speicher und Zeit. Deswegen ist es generell besser, sich die »Prä«-Varianten anzugewöhnen.
- Noch wichtiger ist, dass Sie niemals zwei dieser Operatoren auf der gleichen Variablen innerhalb einer Anweisung anwenden. Der Compiler lässt das zu, das Ergebnis ist aber nicht definiert.

7.7 Relationale Operatoren

Sie können Werte auch miteinander vergleichen. Sehr häufig benötigen Sie == für Gleichheit und != für deren Gegenteil. Zahlen können Sie natürlich auch auf kleiner und größer mit < und > vergleichen sowie in Kombinationen mit gleich: <= und >=.

Das Ergebnis eines solchen Vergleichs ist ein »wahr« oder »falsch«, also true oder false, und somit vom Typ bool. In Schleifen- und if-Bedingungen verwendet man diese dann besonders gerne. Sie können das Ergebnis aber auch in einer bool-Variablen zwischenspeichern oder aus einer Funktion zurückgeben:

```
// als Bedingungen:
if(x < 10) ...
for(int idx=0; idx < 12; ++idx) ...
while(it != end) ...
// zwischenspeichern:
bool isLarge = value >= 100;
// zurückgeben:
bool isPositive(int a) {
    return a > 0;
}
```

7.8 Logische Operatoren

Wenn Sie wissen wollen, ob x zwischen 100 und 200 liegt, dann müssen Sie die beiden relationalen Ausdrücke x>100 und x<200 beide prüfen, und es müssen beide zutreffen. Das könnten Sie mit zwei aufeinanderfolgenden ifs machen. Doch um Ausdrücke vom Typ bool miteinander zu kombinieren, gibt es die *logischen Operatoren*: Sie haben jeweils zwei Operanden und liefern wieder bool zurück. Wenn die Operanden u und v jeweils bool-Ausdrücke sind, dann ist

- u && w »wahr«, wenn u und v beide true sind,
- u || w »wahr«, wenn u oder v true sind, und
- ! u »wahr«, wenn u false ist.

So können Sie dann den Ausdruck kombinieren:

```
if( x > 100 && x < 200 ) ...
```

7.8.1 Kurzschluss-Auswertung

Die obige if-Anweisung ist in der Tat äquivalent zu den verschachtelten if-Anweisungen:

```
if(x > 100)
    if(x < 200)
        ...
```

Beachten Sie, dass der Vergleich `x < 200` nur dann ausgewertet wird, wenn `x > 100` auch tatsächlich `true` war. Falls `x` zum Beispiel `2` ist, wird `x < 200` nicht erreicht.

Das ist dann wichtig, wenn der zweite Vergleich nur dann Sinn ergibt (oder ausgeführt werden darf), wenn der erste positiv war. Zum Beispiel:

```
if( y != 0 && x/y > 5 ) ...
```

Sie wissen sicherlich, dass man niemals durch Null teilen darf. Wenn also in `x/y` die Variable `y` den Wert `0` hat, dann tun Sie etwas Verbotenes. Wenn Sie dies jedoch vorher prüfen, dann kann nichts mehr schiefgehen. In C++ wird

- in `u && v` der Ausdruck `v` nur dann ausgewertet, wenn `u` »wahr« ist, und
- in `u || v` der Ausdruck `v` nur dann ausgewertet, wenn `u` »falsch« ist.

Die Kurzschluss-Auswertung wird im Englischen *Short-Circuit Evaluation* genannt.

»Und« ist immer »und«, »oder« ist immer »oder«

Wenn Sie später eigene Typen definieren, dann werden Sie lernen, dass Sie auch die Operatoren darauf selbst definieren können. Dazu gehören auch die logischen Operatoren.

Definieren Sie jedoch *niemals* einen logischen Operator so um, dass er etwas anderes macht, als die intuitive Berechnung auszuführen. Denn im Zusammenspiel mit der Kurzschluss-Auswertung wären böse Überraschungen vorprogrammiert: Teile Ihres Ausdrucks werden überraschenderweise ausgeführt.

Wenn Sie für einen eigenen Typ `&&` oder `||` überladen, gibt es dort keine Short-Circuit-Auswertung. Auf anderen Typen als `bool` wird ein Ausdruck immer ganz ausgewertet.

7.8.2 Alternative Token

Noch eine kleine Anmerkung zum `!`-Operator: Ich persönlich finde ihn im Quelltext etwas schwer zu sehen. Trotz seiner Unauffälligkeit kehrt er schließlich die ganze Bedeutung des Ausdrucks komplett um. Es ist Geschmackssache, aber ich greife ab und zu auf einen Syntax-Trick zurück. Sie können das `!` durch das Wort `not` ersetzen. So wird zum Beispiel aus `while(!file.eof())...` in Listing 14.1 `while(not file.eof())...`

Man nennt dies *alternatives Token*, und es gibt ein paar davon – am nützlichsten finde ich das `not`. Ich mag Ihnen gar nicht alle nennen, nicht dass Sie anfangen, sie dann überall einzusetzen. Wenn Sie neugierig sind, schauen Sie in der Sprachreferenz unter »alternative Token« oder den verwandten »Di- und Trigraphen« nach.

Wie gesagt, das ist nicht jedermanns Geschmack, und Sie sollten sich dazu in Ihrem Team absprechen. Vom breiten Einsatz dieser Zeichenkombinationen kann ich eher abraten, weil sie wirklich sehr selten eingesetzt werden. Ihre Leser könnten verwirrt werden.

7.9 Pointer- und Dereferenzierungsoperatoren

Sie haben schon gesehen, dass wir Methoden mit einem Punkt `.` aufrufen, zum Beispiel bei `string`:

```
void checkName(std::string& name) {
    if( name.length() ) ...
}
```

Sie werden später sehen, wenn wir Zeiger und C-Arrays besprechen, dass Sie statt der Referenz `&` auch einen *Zeiger* (engl. *Pointer*) auf diese Variable verwenden können:

```
void checkName(std::string* pname) {  // Pointer auf einen string
    if( (*pname).length() ) ...
}
```

Wenn der Typ Ihrer Variable `string*` und nicht `string&` oder `string` ist, dann ist sie nur »indirekt« mit dem Wert verbunden. Um zum Wert zu kommen, verwenden Sie den einstelligen *-Operator `*pname`. Der Methodenaufruf lautet dann `(*pname).length()`. Da das jedoch etwas umständlich ist, gibt es den zweistelligen `->`-Operator als kürzere Form:

```
void checkName(std::string* pname) {
    if( pname->length() ) ...
}
```

Erwähnenswert ist das alles vor allem, weil beide Dereferenzierungsoperatoren (das unäre * und das binäre `->`) auf eigenen Typen selbst definiert werden können. Sie zu kennen ist also nicht nur für Zeiger und C-Arrays wichtig.

Tatsächlich sind Zeiger nur eine sehr spezielle Form von Indirektion. Die Standardbibliothek ist durchzogen von *Iteratoren* – der Verallgemeinerung des Zeigerkonzepts. Sie werden bei den Containern und Algorithmen auf Iteratoren stoßen und dort * und `->` wie selbstverständlich anwenden (siehe Kapitel 10, »Behälter und Zeiger«, und Kapitel 23, »Zeiger«).

7.10 Besondere Operatoren

Da wir hier Operatoren besprechen, müssen auch zwei Sonderlinge erwähnt werden.

Es gibt einen einzigen ternären (dreistelligen) Operator `? :`, der eine `if-else`-Abfrage als Ausdruck ermöglicht. Er hat die Form

- **Bedingungs-Ausdruck** `?` **Wenn-Ausdruck** `:` **Sonst-Ausdruck**

Je nachdem, ob die Bedingung zu »wahr« oder »falsch« ausgewertet wird, ist entweder der »Wenn-Ausdruck« oder der »Sonst-Ausdruck« das Gesamtergebnis. Beachten Sie, dass deswegen die beiden Teile vom gleichen Typ sein müssen, damit der Compiler den Typ des Gesamtausdrucks festlegen kann.

```
int main() {
    for(int w1 = 1; w1 <= 6; ++w1) { // 1..6
        for(int w2 = 0; w2 < 10; ++w2) { // 1..6
            int max = w1 > w2 ? w1 : w2;   // ternärer Operator
        }
    }
}
```

Hier wird `max` der größere der beiden Werte `w1` und `w2` zugewiesen.

Das Komma kann als *Sequenzoperator* in Ausdrücken verwendet werden. Wenn Sie mehrere Ausdrücke in runde Klammern `(...)` schreiben und mit Kommas trennen, dann wertet der Compiler die Ausdrücke von links nach rechts aus, behält aber nur das Ergebnis des letzten Ausdrucks als Gesamtergebnis.

Für zwei Ausdrücke sieht das also so aus:

- (**Ausdruck-1** , **Ausdruck-2**)

Der Compiler berechnet zunächst »Ausdruck-1« und danach »Ausdruck-2«. Das Ergebnis des ersten Ausdrucks versickert, sodass der Gesamtausdruck den Wert und Typ des zweiten Ausdrucks erhält:

```
int main() {
    int a = 0;
    int b = 0;
    for(int w1 = 1; w1 <= 6; ++w1) { // 1..6
        for(int w2 = 0; w2 < 10; ++w2) { // 1..6
            int max = w1 > w2 ? (a+=b , w1) :( b+=1 , w2); // Sequenzoperator
        }
    }
}
```

Listing 7.2 Mit Kommas in Klammern können Sie mehrere Ausdrücke verketten.

Wenn `w1 > w2` ist, dann wird der Ausdruck `(a+=b, w1)` ausgewertet. Zwar wird `w1` als Ergebnis für `max` zurückgegeben, aber zuvor wird noch `a+=b` ausgeführt. Im Falle von `w <= w2` wird `w2` aus `(b+=1, w2)` der Variablen `max` zugewiesen, nachdem noch `b+=1` ausgewertet wurde.

Ich habe absichtlich ein besonders »skurriles« Beispiel für den Sequenzoperator gewählt, denn erstens fällt es schwer, ein sinnvolles Beispiel zu finden, und zweitens sollten Sie dieses Spezialkonstrukt meiden wie der Teufel das Weihwasser. Es gibt Ausnahmen, in denen der Einsatz sinnvoll ist, nämlich dort, wo nur ein einzelner Ausdruck erlaubt ist und es sonst komplizierter würde. Das kann zum Beispiel in dem Inkrementierungsteil von `for`-Schleifen der Fall sein:


```
#include <iostream>
int main() {
int arr[] = { 8,3,7,3,11,999,5,6,7 };
int len = 9;
    for(int i=0, *p=arr; i<len && *p!=999; ++i, ++p) { // erst ++i, dann ++p
        std::cout << i << ":" << *p << " ";
    }
    std::cout << "\n";
    // Ausgabe: 0:8 1:3 2:7 3:3 4:11
}
```

Listing 7.3 In for-Schleifen kann das Sequenzkomma nützlich sein.

Beachten Sie, dass in einer Deklaration `int a, b, c;` das Komma nicht der Sequenzoperator ist. Das gilt auch für den Deklarationsteil der `for`-Schleife, `int i=0, *p=arr`, bei dem das Komma nur die Deklarationen voneinander trennt.

Auch das Komma, das Funktionsargumente in `func(x, y, z)` oder Listenelemente in `{1,2,3}` trennt, ist nicht der Sequenzoperator.

7.11 Funktionsähnliche Operatoren

Auch wenn sie nicht so aussehen, so gehören auch die folgenden Fälle zu den Operatoren:

- **`(typ)`**`wert`
  In `(int)wert` versucht der Compiler, `wert` in einen `int` umzuwandeln, egal, welchen Typ `wert` hat. Wenn ihm das nicht möglich ist, meldet er einen Fehler. Ansonsten wird `wert` »passend gemacht«, was seine Gefahren hat. So kann zum Beispiel bei der Umwandlung von `long` zu `short` Information verloren gehen, ohne dass Sie es merken. Diese Schreibweise der *Typumwandlung* (engl. *Type Cast*) nennt sich »C-Style-Cast«. Das C++-Pendant `static_cast<int>(wert)` ist meist besser geeignet.
- **`sizeof`**
  Mit **`sizeof`**`(typ)` und **`sizeof`**`(wert)` können Sie die Größe eines Typs oder einer Variablen in `char`-Einheiten herausfinden. Ein `sizeof(char)` liefert immer `1`.
- **`new`** **und** **`delete`**
  `new` `Klasse{}` und **`delete`** `var` verwenden Sie, um dynamischen Speicher zu verwalten, wie Sie in Kapitel 23, »Zeiger«, sehen werden.
- **`throw`**
  Mit **`throw`** `ExceptionClass{};` lösen Sie eine Ausnahme aus, was in Kapitel 14, »Fehlerbehandlung«, erklärt wird.

7.12 Operatorreihenfolge

Wenn Sie in einem Ausdruck mehrere Operatoren verwenden – möglicherweise unterschiedliche –, dann werden diese in einer bestimmten Reihenfolge ausgewertet.

So, wie Sie von einem ordentlichen Taschenrechner verlangen können, dass er bei `3+4*5+6` die Regel *Punkt- vor Strichrechnung* beachtet und das korrekte Ergebnis von 29 produziert, so beherrscht C++ dies auch. Darüber hinaus haben auch alle anderen Operatoren eine *Präzedenz* – je höher ein Operator in dieser Rangfolge ist, desto früher wird er im Vergleich zu anderen Operatoren ausgewertet.

Die ausführliche Liste finden Sie in Anhang B, »Operator-Präzendenzen«, merken Sie sich fürs Erste diese einfache Reihenfolge, die mit der stärksten Bindung beginnt:

- Multiplikative: *, / und %
- Additive: + und -
- Streamoperatoren: << und>>
- Vergleiche <, <=, > und >=
- Gleichheit == und !=
- Logische, in dieser Reihenfolge: &&, ||
- Zuweisungen mit =, aber auch alle zusammengesetzten wie +=.

So können Sie so manchen komplexen Ausdruck schreiben, ohne mit Klammern die Bedeutung korrigieren zu müssen:

```
bool janein = 3*4 > 2*6 && 10/2 < 13%8;
```

spart Ihnen die Klammern:

```
bool janein = (((3*4) > (2*6)) && ((10/2) < (13%8)));
```

Beachten Sie, dass der Stream-Ausgabeoperator << loser bindet als normale Arithmetik mit + und *. Haben Sie aber Vergleiche in der Ausgabe, benötigen Sie Klammern um den Vergleich:

```
std::cout << 2*7 << x+1 << n/3-m << "\n";  // Keine Klammern zwischen << nötig
std::cout << (x0 >= x1) << (a<b || b<c);   // Klammern: << würde enger binden
```

Aber was passiert, wenn mehrere Operatoren der gleichen Präzedenz nebeneinander stehen? In Ausdrücken wie `10-5-2` wird das linke - zuerst ausgewertet, denn - ist *links-assoziativ* – als wäre der Ausdruck `((10-5)-3)` geklammert. Das Ergebnis ist also `2`. Alle zweistelligen arithmetischen, booleschen und vergleichenden Operatoren sind linksassoziativ, sodass Sie intuitiv damit rechnen können. Manchmal wird dies auch *links-nach-rechts-assoziativ* oder *LR-assoziativ* genannt.

Die Gruppe der Zuweisungsoperatoren wiederum ist durchgehend *rechtsassoziativ*, weswegen der Compiler für `x += y += z += 1` erwartungsgemäß `(x += (y += (z += 1)))` ausführt und zuerst `z` um `1` inkrementiert, um sich dann nacheinander mit den anderen Variablen zu beschäftigen. Würde der Ausdruck `(((x += y) += z) += 1)` ausgewertet, dann würde `x` mehrmals einen neuen Wert erhalten, denn `x += y` gibt ja auch `x` zurück, und darauf würde dann `+= z` ausgeführt – `y` würde nicht verändert. Und noch einmal würde `x` zurückgeliefert, und `+= 1` inkrementierte dieses um `1` anstatt `z` zu erhöhen.

Meistens funktioniert die Präzendenz und Assoziativität intuitiv und wie erwartet. Im Zweifelsfall klammern Sie besser, denn spätere Leser stellen sich wahrscheinlich die gleichen Fragen wie Sie.

7.13 Aufgaben

Wiederholungsfragen

1. Klammern Sie `std::cout << x << y << "\n";` so, wie der Compiler den Ausdruck sieht und auswertet. Was ist mit `x += y += z += 1;`, und warum?
2. Berechnen Sie jeweils `a` und `b`:
   1. `int a = 55 & 1; int b = 55 | 1;`
   2. `int a = 55 & 2; int b = 55 | 2;`
   3. `int a = 55 & 4; int b = 55 | 4;`
   4. `int a = 55 & 8; int b = 55 | 8;`
   5. `int a = 55 & 15; int b = 55 | 15;`
   6. `int a = 55 & 31; int b = 55 | 31;`
   7. `int a = 55 & 63; int b = 55 | 63;`
   8. `int a = 55 & 85; int b = 55 | 85;`
   9. `int a = 55 & 42; int b = 55 | 42;`
   10. `int a = 55 & 55 & 42 & 42; int b = 55 | 55 | 42 | 42;`

Vertiefungsfragen

1. Rechnen Sie die folgenden Binärzahlen ins Dezimalsystem um (»0b« ist das Präfix, um anzuzeigen, dass es sich um eine Binärzahl handelt): 0b1111, 0b1001, 0b101010101, 0b11110000
2. Rechnen Sie die folgenden Dezimalzahlen ins Binärsystem um: 31, 42, 1, 2, 73, 256, 92, 1001
3. Sei `x` ein `int` mit irgendeinem Wert. Was ist das Ergebnis von `x ^ 0`, `x ^ x` sowie `x ^ ~0`? Tipp: `~0` enthält nur Einsen.

Erweiterungsfragen

Bei der Erklärung des Operators ~ für die bitweise Invertierung haben wir das *Zweierkomplement* erwähnt. Das gibt es nur für Typen wie `int`, deren Wert negativ sein kann.

1. Schreiben Sie einen Ausdruck, der für `int value` das Gleiche macht wie `~value`, aber stattdessen nur die anderen bitweisen und arithmetischen Operatoren, also `&`, `|`, `~`, `<<`, `>>`,`+` und `-`, benötigt.
2. Schreiben Sie mit `|`, `&` und ~ das Äquivalent zu `a ^ b`. Bei `33 ^ 44` kommt beispielsweise `13` heraus, ebenso soll das bei Ihrer Lösung ohne `^` sein.