2 Grundelemente eines C++-Programms

In diesem Kapitel legen wir den Grundstein zur Programmierung in C++. Wir schauen uns an, welche Elemente immer in einem C++-Programm vorkommen und wie Texte auf dem Bildschirm ausgegeben werden können.

2.1 Das erste Programm

Schreiben wir nun unser erstes C++-Programm:

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

Um das Programm übersetzen und starten zu können, sollten Sie in der Entwicklungsumgebung Ihrer Wahl ein neues Projekt anlegen, dort eine C++-Datei hinzufügen (an der Endung .cpp zu erkennen) und dort das obere Programm einfügen.

Das Programm sollte sich fehlerfrei kompilieren und starten lassen, allerdings wird noch nichts passieren.

Wir haben bisher lediglich das Kernstück eines jeden C++-Programms programmiert, die `main`-Funktion. Jedes C++-Programm muss genau eine `main`-Funktion besitzen, sie ist der Einstiegspunkt in unser Programm. Jeder Start eines C++-Programms beginnt mit der `main`-Funktion.

Dem Funktionsnamen folgt ein Paar runder Klammern. Diese Klammern dienen später dazu, Informationen an die Funktion zu übergeben, bleiben aber fürs Erste leer.

Hinter dem Funktionskopf stehen geschweifte Klammern, mit denen in C++ eine zusammengesetzte Anweisung (compound statement) gebildet wird. Alle Anweisungen innerhalb der geschweiften Klammern werden beim Aufruf der Funktion ausgeführt. Da die Klammern bisher leer sind, passiert beim Start auch noch nichts.

2.1.1 Implizites return

Vor `main` steht immer `int`, das fordert der ISO-Standard. Ohne an dieser Stelle bereits genauer auf die Datentypen von C++ einzugehen, bedeutet dieses `int`, dass `main` immer einen ganzzahligen Wert zurückgibt. Über diesen Wert teilt das Programm der aufrufenden Umgebung mit, ob es fehlerfrei beendet wurde oder nicht. In einigen Fällen wird auch ein Fehlercode zurückgegeben, der den aufgetretenen Fehler genauer spezifiziert.

Der Compiler übersetzt die `main`-Funktion immer so, dass der zurückgegebene Wert die Bedeutung »alles in Ordnung« hat, und das ist üblicherweise der Wert 0. Das vom Compiler erzeugte Programm sieht damit so aus:

```
int main()
{
  return 0;
}
```

Diese automatische Ergänzung mit einer `return`-Anweisung, falls der Programmierer kein eigenes `return` programmiert hat, nimmt der Compiler nur bei der `main`-Funktion vor. In allen anderen Fällen ist der Programmierer dafür verantwortlich, eine geeignete `return`-Anweisung zu programmieren.

2.2 Die Ausgabe

Um unser erstes C++-Programm aus dem Stadium der Sinnlosigkeit herauszuheben, werden wir nun einen der wichtigsten Aspekte besprechen: die Ausgabe. Schauen wir uns dazu zunächst das erweiterte Programm an:

```
#include<iostream>

int main() {
  std::cout << "Hello World";
}
```

Auf dem Bildschirm sollte der Text »Hello World« erscheinen, je nach Entwicklungsumgebung noch direkt gefolgt von der Aufforderung, das Programm mit einem Tastendruck zu beenden.

Dieses kleine Beispiel bietet uns bereits die Möglichkeit, einige grundlegende Dinge von C++ zu besprechen.

Einfache Anweisungen werden in C++ mit einem Semikolon abgeschlossen.

Und noch eine Regel ist wichtig:

Konstante Zeichenfolgen stehen in C++ in Anführungszeichen.

Darüber hinaus muss in C++ penibel auf Groß- und Kleinschreibung geachtet werden. Der Name »Andre« und der Name »andre« sind zwei unterschiedliche Bezeichner.

2.2.1 cout

Der Befehl zur Ausgabe auf die Konsole heißt in C++ `cout`. Warum davor noch ein `std::` steht, besprechen wir gleich.

`cout` ist der sogenannte Standardausgabe-Stream beziehungsweise das Objekt, das der Abstraktion wegen für den Standardausgabestrom steht. Dadurch wird der Programmierer nicht mehr mit den plattformspezifischen Eigenarten der Ausgabe belastet. Er gibt die auszugebenden Daten einfach an das `cout`-Objekt und dieses sorgt dann für eine ordnungsgemäße Ausgabe. Als Standardausgabe wird im Allgemeinen der Bildschirm verwendet.

Der <<-Operator schiebt bildlich gesprochen die Daten in den Ausgabestrom. In diesem Fall handelt es sich bei den auszugebenden Daten um eine Stringkonstante.

Stringkonstante

Stringkonstante ist eine in doppelten Anführungszeichen stehende Folge von Zeichen, die implizit mit dem Wert 0 (`'\0'`) abgeschlossen wird.

Gehen wir den Ablauf des Programms einmal schrittweise durch.

Wenn Sie das Programm kompilieren und starten, wird zuerst die Funktion `main` aufgerufen. Die erste Anweisung ist die Anweisung, die die auszugebenden Daten an das `cout`-Objekt schickt, also in den Standardausgabe-Stream schiebt. `cout` gehört zur Standardbibliothek von C++.

Nachdem die auszugebenden Daten zu `cout` geschickt wurden, fährt das Programm hinter dem Semikolon der Anweisung fort. Dort ist aber nur das Ende der Funktion `main`, was dem Ende des gesamten Programms gleichkommt.

Sie werden sich vielleicht gewundert haben, dass im Programmtext die `cout`-Anweisung (und vorher auch schon die `return`-Anweisung) nach rechts eingerückt ist. Dies ist nicht notwendig, dient aber der Übersichtlichkeit. Wenn Sie die gleiche Einrückung wie hier im Buch verwenden möchten, dann sollten Sie die Abstände auf zwei Zeichen einstellen.

Zeilenumbruch

Um bei der Ausgabe eine neue Zeile zu beginnen, reicht es nicht aus, eine zweite Ausgabe zu tätigen:

```
#include<iostream>

int main() {
  std::cout << "Hello World!";
  std::cout << "Jetzt komme ich!";
}
```

Stattdessen muss an der Stelle, an der die neue Zeile beginnen soll, ein Zeilenumbruch in die Zeichenfolge eingefügt werden. Dies geschieht in Form einer Escape-Sequenz. Tabelle 2–1 listet alle in C++ verfügbaren Escape-Sequenzen auf. An dieser Stelle wollen wir die Escape-Sequenz für Newline einsetzen, sie lautet `\n`.

Das Programm mit Zeilenumbrüchen sieht damit so aus:

```
#include<iostream>

int main() {
  std::cout << "Hello World!\n";
  std::cout << "Jetzt komme ich!\n";
}
```

endl

Eine weitere Möglichkeit, einen Zeilenumbruch zu erhalten, ist der Manipulator `endl`, der über den Ausgabestrom ausgegeben wird:

```
#include<iostream>

int main() {
  std::cout << "Hello World!";
  std::cout << std::endl;
  std::cout << "Jetzt komme ich!";
  std::cout << std::endl;
}
```

Manipulator

Als Manipulator bezeichnet man ein Objekt, das über den Ausgabestrom ausgegeben wird und das Verhalten des Stroms manipuliert.

Das `endl` macht aber noch mehr, als einen Zeilenumbruch zu erzeugen. Dazu müssen wir uns anschauen, wie die Ausgabe funktioniert.

Statt direkt auf dem Bildschirm ausgegeben zu werden, landen die Ausgaben zunächst einmal in einem internen Speicherbereich, dem Ausgabepuffer. Erst wenn dieser Puffer voll ist, wird er auf dem Bildschirm ausgegeben. Unter

Umständen werden Ausgaben deshalb nicht sofort angezeigt, weil der Puffer einfach noch nicht voll ist.

Dieses Problem vermeidet `endl`, denn mit der Ausgabe von `endl` wird zusätzlich ein `flush` ausgeführt. Dieser Flush (aus dem Englischen »to flush«, was unter anderem die Bedienung der Toilettenspülung bedeutet) sorgt dafür, dass der Inhalt des Ausgabepuffers auf den Bildschirm »gespült« wird, auch wenn er noch nicht komplett gefüllt war.

Das `endl` ist aber nicht immer notwendig. Vor einer Eingabe oder am Programmende wird der Ausgabepuffer immer geleert, unabhängig von dessen Füllstand.

2.3 include

Mit der Ausgabe hat noch ein weiterer Befehl Einzug in unser Programm gehalten: die `include`-Direktive des Präprozessors:

```
#include<iostream>
```

Der Präprozessor – wie die Silbe »Prä« erahnen last – durchläuft die Datei vor dem eigentlichen Prozess des Kompilierens. Aber was genau bedeutet »kompilieren«? Abbildung 2–1 zeigt den Vorgang.

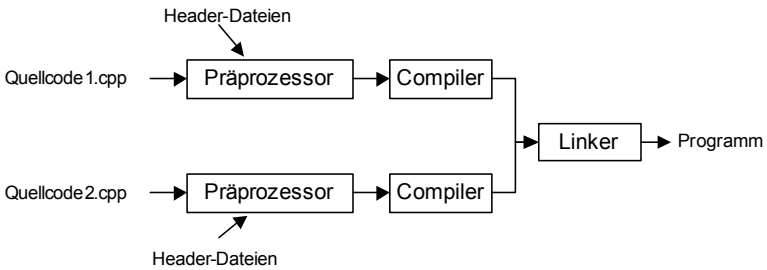

Abb. 2–1 *Der Vorgang des Kompilierens*

C++ ist eine Hochsprache, die vom Prozessor des Computers nicht verstanden wird, denn dieser kennt nur seine Maschinensprache. Maschinensprache ist eine sehr einfache, aus wenigen Befehlen bestehende Sprache. Entsprechend viele dieser Befehle sind notwendig, um selbst einfachste Dinge zu programmieren. Eine typische Szene könnte sein:

Lade Wert an Adresse $92E2 in Register1. Addiere Wert an Adresse $92E6 auf Register1. Speichere Inhalt von Register1 an Adresse $92EA.

Dasselbe in C++ wäre etwa `x=a+b` – viel kürzer und vor allem für einen Menschen viel verständlicher.

Damit also ein in C++ geschriebenes Programm auf einem Rechner laufen kann, muss es in die Maschinensprache des Prozessors übersetzt werden. Und diesen Vorgang nennt man »kompilieren«.

Kompilieren

Das Übersetzen eines Hochsprachenprogramms in die Maschinensprache des Zielprozessors bezeichnet man als Kompilieren. Der Übersetzer wird Compiler genannt.

In C++ besteht dieser Übersetzungsprozess aus mehreren Schritten. Bevor eine cpp-Datei dem Compiler übergeben wird, durchläuft sie der Präprozessor, der nach an ihn gerichteten Befehlen sucht. Der Präprozessor selbst versteht C++ nicht, er arbeitet rein auf Textebene.

Präprozessordirektive

Diese an den Präprozessor gerichteten Befehle beginnen mit einem # in der ersten Spalte. Präprozessordirektiven dürfen nicht nach rechts eingerückt werden.

Befehle an den Präprozessor, sogenannte Präprozessordirektiven, beginnen immer mit einem »#«. Der wohl häufigste Befehl ist #include, was übersetzt so viel wie »Einbinden« bedeutet. Mithilfe dieses Befehls kann eine andere Datei in die Quellcodedatei eingebunden werden. In unserem Fall binden wir die Datei »iostream« ein, in der alle für die Textein- und -ausgabe notwendigen Elemente der C++-Standardbibliothek enthalten sind, unter anderem das in unserem Programm verwendete `cout` und `endl`. Würden wir die Include-Direktive aus dem Programm entfernen, käme bei der Kompilation die Fehlermeldung des Compilers, er würde `cout` und `endl` nicht kennen.

Abbildung 2–1 zeigt noch eine weitere Besonderheit von C++: Jede Quellcodedatei des Programms wird isoliert von den anderen kompiliert, der Compiler hat auch keinerlei Erinnerungen an sein Tun.

Für das Beispiel in der Abbildung heißt das: Während er die Datei »Quellcode1.cpp« kompiliert, weiß er nicht, dass er noch die Datei »Quellcode2.cpp« kompilieren wird. Und während er die Datei »Quellcode2.cpp« kompiliert, weiß er nicht, dass er die Datei »Quellcode1.cpp« bereits kompiliert hat.

Die einzeln kompilierten Dateien werden im letzten Schritt vom Linker (auf Deutsch so viel wie »Binder«) zu einer einzigen Datei zusammengebunden, die dem lauffähigen Programm entspricht.

2.4 Namensbereiche

Bleibt noch zu klären, warum vor `cout` und `endl` dieses `std::` steht.

Bildlich betrachtet ist `std` vergleichbar mit einer Vorwahl. Stellen Sie sich vor, es gäbe keine Vorwahlen. Dann müsste die Vergabe von Telefonnummern global geregelt werden, schließlich dürfen Teilnehmer in Köln und Timbuktu nicht dieselbe Telefonnummer bekommen. Ländervorwahlen lösen das Problem, denn jedes Land kann hinter seiner Vorwahl die Telefonnummern nach eigenen Regeln

vergeben. Jetzt dürfen auch Teilnehmer in München und Rom dieselbe Nummer besitzen, denn sie unterscheiden sich in der Vorwahl.

Dieses Prinzip nennt sich in C++ Namensbereich. Ein Namensbereich ist nichts anderes als eine programmiertechnische Vorwahl, hinter der Namen beliebig vergeben werden können. Der Namensbereich der C++-Standardbibliothek lautet `std` als Abkürzung von »Standard«.

In C++ kann eine Gruppe von Namen (das können Namen von Konstanten, Funktionen, Klassen etc. sein) zu einem Namensbereich zusammengefasst werden. Ganz konkret gehört die Definition von `cout` zum Namensbereich `std`.

Die Namensbereiche wurden eingeführt, um die Möglichkeit einer Doppelbenennung verhindern zu können. Man ist dadurch in der Lage, sein eigenes `cout` zu definieren, wenn man es einem anderen Namensbereich zuordnet.

using namespace

Bei den Telefonnummern gibt es eine Besonderheit: Möchte ich einen Teilnehmer mit derselben Vorwahl wie meine eigene Telefonnummer anrufen, dann muss ich die Vorwahl nicht mit wählen.

Etwas Ähnliches existiert auch in C++: Wir können dem Compiler mitteilen, dass er in bestimmten Namensbereichen automatisch suchen soll. Auf diese Weise können wir uns die explizite Angabe von `std` sparen, wenn wir ein Element der Standardbibliothek ansprechen möchten.

Der Befehl dazu lautet `using namespace`:

```
#include<iostream>

using namespace std;

int main() {
  cout << "Hello World!";
  cout << endl;
  cout << "Jetzt komme ich!";
  cout << endl;
}
```

Man spricht auch davon, einen Namensbereich global verfügbar zu machen. Die Elemente des Namensbereichs lassen sich dann ansprechen, als ständen sie überhaupt nicht in einem Namensbereich.

In unserem Beispiel brauchen wir dann bei Namen, die im Namensbereich `std` definiert sind, nicht mehr explizit angeben, dass wir die Definition aus `std` verwenden wollen. Elemente aus anderen Namensbereichen müssen weiterhin explizit mit ihrem Namensbereich angegeben werden. Es können aber mehrere `using namespace`-Anweisungen verwendet werden, falls weitere Namensbereiche global verfügbar gemacht werden sollen.

Der Einsatz von `using namespace` spart eine Menge Tipparbeit und gestaltet das Programm übersichtlicher. Und wo wir gerade bei dem Thema »Tipparbeit sparen« sind: Bei der Ausgabe lässt sich der Operator << auch verketten:

```
#include<iostream>

using namespace std;

int main() {
  cout << "Hello World!" << endl;
  cout << "Jetzt komme ich!" << endl;
}
```

Das erste `endl` ist eigentlich unnötig, weil es an dieser Stelle nur um einen Zeilenumbruch geht und nicht um das Leeren des Ausgabepuffers. Wir könnten es daher auch mit der Escape-Sequenz `\n` ersetzen:

```
#include<iostream>

using namespace std;

int main() {
  cout << "Hello World!\nJetzt komme ich!" << endl;
}
```

Wie eigene Namensbereiche erstellt werden können, besprechen wir in Kapitel 13.

2.5 Kommentare

Nachdem uns nun das erste Programm komplett verständlich ist, wollen wir in diesem Abschnitt die Möglichkeit besprechen, Kommentare in das eigentliche Programm einfügen zu können.

Kommentare dienen dazu, Informationen im Programm unterzubringen, die nicht Bestandteil des tatsächlichen Programms sind und auch vom Compiler nicht verstanden werden könnten.

So können Sie zum Beispiel hinter bestimmten Programmzeilen Bemerkungen schreiben, damit Sie auch später noch wissen, welche Funktion sie haben. Oder Sie können vor jedem größeren Abschnitt ein paar Zeilen über seine Funktion schreiben. Obwohl immer wieder manche Programmierer behaupten, ein Programmtext sei selbsterklärend und nur unfähige Leute bräuchten Kommentare, sollten Sie sich diesbezüglich nicht einschüchtern lassen und nie mit Kommentaren geizen. Sie tun sich und anderen, die Ihr Programm einmal verstehen müssen, einen großen Gefallen.

Mehrzeilige Kommentare

Mehrzeilige Kommentare beginnen mit den Zeichen /* und werden mit */ beendet. Alles, was zwischen diesen Zeichen steht, wird vom Compiler ignoriert.

Achtung

Mehrzeilige Kommentare dürfen nicht verschachtelt werden!

Deshalb ist das folgende Konstrukt ungültig:

```
/* Innerhalb dieses Kommentares ist /* ein Kommentar */
eingebettet */
```

Einzeilige Kommentare

Benötigen Sie für Ihren Kommentar nur maximal eine Zeile (z.B. um eine kurze Bemerkung hinter eine Anweisung zu schreiben), dann verwenden Sie //.

Wenn Sie die beiden Formen der Kommentare mischen, dann sind auch Verschachtelungen möglich:

```
//  /* Kommentar */
/* // Kommentar */
```

Schauen wir uns als Beispiel unser Programm mit Kommentaren versehen an:

```
/*
** Programm: Einführungsbeispiel zu C++
** Autor: André Willms
** Letzte Änderung: 03.01.2015
*/

#include<iostream>  /* Einbinden von iostream */

using namespace std; // std global verfügbar machen

// Hauptfunktion

int main() {

// Textausgabe
  cout << "Hello World!\nJetzt komme ich!" << endl;
}
```

Wenn Sie sich die Kommentare einmal genau ansehen, müssten Sie sich fragen, warum hinter der Präprozessordirektive die Syntax der mehrzeiligen Kommentare verwendet wurde, obwohl der Kommentar nur eine Zeile lang ist.

Tipp

Benutzen Sie hinter Präprozessordirektiven nur `/* ... */`-Kommentare.

Dafür gibt es eine einfache Erklärung. Die /* ... */-Kommentare existierten auch schon in C, wohingegen die //-Kommentare eine Neuerung von C++ sind. Auch den Präprozessor gab es bereits in der Programmiersprache C, weswegen er bei vielen C++-Compilern nicht neu programmiert, sondern einfach von einem C-

Compiler übernommen wurde. Dadurch kann es passieren, dass ein Präprozessor den neuen C++-Kommentar nicht versteht und einen Fehler meldet.

2.6 Escape-Sequenzen

Der Vollständigkeit halber möchte ich Ihnen am Ende dieses Kapitels noch die restlichen in C++ verfügbaren Escape-Sequenzen vorstellen.

Escape-Sequenz	Zeichen
\'	Das Zeichen '
\"	Das Zeichen "
\?	Das Zeichen ?
\\	Das Zeichen \
\a	BEL (bell), akustisches Warnsignal. Ob es hörbar ist, hängt vom Rechner ab.
\b	BS (backspace), Cursorposition ein Zeichen nach links (Zeichen wird gelöscht)
\f	FF (formfeed), Seitenvorschub
\n	NL (new line), Cursorposition wird auf Anfang der nächsten Zeile gesetzt.
\r	CR (carriage return), Cursorposition wird auf Anfang der aktuellen Zeile gesetzt.
\t	HT (horizontal tab), nächste horizontale Tabulatorposition
\v	VT (vertical tab), nächste vertikale Tabulatorposition
\aaa	Zeichencode `aaa` in oktaler Schreibweise
\xaa	Zeichencode `aa` in hexadezimaler Schreibweise

Tab. 2–1 *Die Escape-Sequenzen von C++*

2.7 Zusammenfassung

Was wir in diesem Kapitel gelernt haben:

- Die Hauptfunktion eines C++-Programms heißt `main` (Das erste Programm).
- Mit `cout` werden Ausgaben auf dem Bildschirm getätigt (Die Ausgabe).
- Über den Befehl `#include` lassen sich andere Dateien einbinden (include).
- Die Elemente der C++-Standardbibliothek stehen im Namensbereich `std` (Namensbereiche).
- Es gibt einzeilige (`//`) und mehrzeilige (`/* … */`) Kommentare (Kommentare).

2.8 Spielprojekt

Mit dem Wissen dieses Kapitels können wir unser Spiel noch nicht sehr weit voranbringen. Aber die Startmeldung können wir bereits programmieren:

```
#include <iostream>

using namespace std;

int main() {

  cout << "\nTextadventure-Engine V1, "
       << "programmiert in C++ von Andre Willms\n"
       << endl;
}
```