2

Einfache Ein- und Ausgabe

Dieses Kapitel behandelt die folgenden Themen:

- Abfrage der Tastatureingabe
- Daten aus Dateien lesen
- Daten in Dateien schreiben

Ein- und Ausgabe werden soweit beschrieben, dass Abfragen der Tastatur und Darstellungen auf dem Bildschirm ebenso wie das Lesen und Schreiben von Dateien auf einfache Weise möglich sind. Speziellere Fragen und Einzelheiten werden zunächst zurückgestellt.

Mehr dazu lesen Sie in Kapitel 10

2.1 Standardein- und -ausgabe

Ein Programm empfängt einen Strom von Eingabedaten, verarbeitet diese Daten und gibt einen Strom von Ausgabedaten aus. Unter »Strom« (englisch *stream*) wird eine Folge

von Bytes verstanden, die nacheinander vom Programm interpretiert beziehungsweise erzeugt werden. In C++ sind einige Ein- und Ausgabekanäle vordefiniert. Sie werden zuerst beschrieben.

`cin`	Standardeingabe (Tastatur)
`cout`	Standardausgabe (Bildschirm)
`cerr`	Standardfehlerausgabe (Bildschirm)

Eingabe

Der Operator >>, der uns in anderem Zusammenhang schon als Bit-Verschiebeoperator begegnet ist, sorgt bei der Eingabe dafür, dass automatisch die nötigen Umformatierungen vorgenommen werden. `int zahl; cin >> zahl;` bewirkt, dass eine Folge von Ziffernzeichen bis zu einem Nicht-Ziffernzeichen eingelesen und in die interne Darstellung einer `int`-Zahl umgewandelt wird. Die Auswertung durch den >>-Operator hat bestimmte Eigenschaften:

- Führende Zwischenraumzeichen (englisch *whitespace*) werden ignoriert. Zwischenraumzeichen sind Leerzeichen, Tabulatorzeichen '‿', Zeilenrücklauf '\r', Zeilensprung '\v', Seitenvorschub '\f' und die Zeilenendekennung '\n'. In der ASCII-Tabelle auf Seite 887 sind es die Zeichen 0x20 und 0x09 bis 0x0d in Hexadezimalschreibweise.
- Zwischenraumzeichen werden als Endekennung genutzt.
- Andere Zeichen werden entsprechend dem verlangten Datentyp interpretiert.

Sollen Zwischenraumzeichen nicht ignoriert werden, ist die Funktion `get()` zu verwenden, die zum Einlesen einzelner Zeichen, also nicht von Zahlen, verwendet werden kann:

```
// einzelnes Zeichen einlesen
char c;
cin.get(c);
```

Der nach außen hin nicht sichtbare Ablauf einer Tastaturabfrage mit »`cin >>`« besteht aus mehreren Schritten, wobei angenommen wird, dass noch nichts auf der Tastatur eingegeben worden ist:

1. Aufforderung an das Betriebssystem zur Zeichenübergabe.
2. Eingabe der Zeichen auf der Tastatur (mit Korrekturmöglichkeit durch die Backspace-Taste). Die Zeichen werden vom Betriebssystem der Reihe nach in einem besonderen Speicherbereich abgelegt, dem Tastaturpuffer.
3. Abschluss der Eingabe mit der ENTER-Taste. Damit wird das '\n'-Zeichen als Zeilenendekennung im Tastaturpuffer abgelegt, und der Puffer wird durch das Betriebssystem an C++ übergeben.
4. Auswertung des Tastaturpufferinhalts durch den Operator >> je nach Datentyp der gefragten Variable.
5. Daten, die nach der Auswertung übrigbleiben, weil sie nicht zu dem Datentyp passen, verbleiben im Tastaturpuffer und werden mit dem nächsten `cin >>` gelesen.

Das Beispielprogramm verlangt der Reihe nach eine `int`- und eine `double`-Zahl. Falls das Format mit dem aktuell erwarteten Datentyp nicht übereinstimmt, wird die Schleife abgebrochen, beispielsweise bei Eingabe eines Buchstabens[1].

[1] Zur Auswertung der `while`-Bedingung siehe Abschnitt 10.5.

Listing 2.1: Zahl einlesen

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

int main( ) {
    int i;
    double d;
    while(cin >> i >> d) {
       cout << i << endl << d << endl;
    }
}
```

Wenn 100␣12.4 ENTER eingegeben wird, wobei ␣ für irgendein Whitespace-Zeichen steht, ist das Ergebnis wie erwartet i == 100 und d == 12.4. Falls die nächste Eingabe 2.7 lautet, erhalten wir 2 und 0.7 als Ausgabe. Warum? An dieser Stelle wird eine `int`-Zahl erwartet. Es wird also die 2 als `int`-Zahl gelesen, der Dezimalpunkt gehört *nicht* mehr dazu. Die folgenden Zeichen .7 werden als `double` interpretiert und als 0.7 ausgegeben.

Eingabe von Strings

Die Eingabe von Strings unterscheidet sich nicht von der Eingabe von Zahlen, wie oben beschrieben. Häufig möchte man jedoch nicht nur durch Zwischenraumzeichen getrennte Zeichenfolgen einlesen, sondern ganze Zeilen. Beispielsweise sollen Vor- und Nachname eingegeben werden. Das Programm dazu sei:

Listing 2.2: Einfache String-Eingabe

```
// cppbuch/k2/eingabe1.cpp
#include<iostream>
#include<string>
using namespace std;

int main() {
   cout << "Bitte Vor– und Nachnamen eingeben:";
   string derName;
   cin >> derName;
   cout << derName;
}
```

Sie tippen ein:

Donald␣Duck ENTER

und sind über die knappe Ausgabe *Donald* nicht erbaut. Das war es nicht, was Sie wollten! Der >>-Operator versteht die hier mit ␣ gekennzeichneten Leerzeichen als Endekennung, weswegen der Rest des Namens im Tastaturpuffer hängen bleibt. Er könnte durch weitere `cin >> ...` Anweisungen ausgelesen werden. Besser geht es mit der Funktion `getline()`, die eine ganze Zeile einliest. Dazu wird `main()` etwas modifiziert:

```
int main() {
   cout << "Bitte Vor– und Nachnamen eingeben:";
   string derName;
   getline(cin, derName); // mehr auf Seite 382
```

```
   cout << derName; // Donald Duck
}
```

Ausgabe

Der Operator << formt automatisch aus der internen Darstellung in eine Textdarstellung um. Es wird stets die mindestens notwendige Weite genommen: `cout << 7 << 11;` erscheint als »711«. Formatierungen sind möglich, zum Beispiel:

```
cout << 7;
cout.width(6);
cout << 11;
```

Jetzt wird »7 11« angezeigt, weil vor der Zahl 11 vier Leerzeichen eingefügt werden, um die Gesamtweite von sechs Zeichen zu erreichen. Weitere Formatierungen sind im Abschnitt 10.1.1 beschrieben. Ein Beispiel zur formatierten tabellarischen Ausgabe von Zahlen mit einer festgelegten Zahl von Nachkommastellen finden Sie dort auf Seite 380.

`cin` und `cout` können auf Betriebssystemebene mit < beziehungsweise > umgeleitet werden. Wenn ein Programm namens *prog1* mit Bildschirmausgabe und Tastatureingabe korrekt compiliert und gelinkt worden ist, können mit den Umleitungszeichen < und > die zugehörigen Eingabedaten aus einer Datei *eingabe* anstelle der Tastatur geholt werden. Anstelle der Bildschirmausgabe kann in eine Datei *ausgabe* geschrieben werden, wobei die Dateinamen natürlich wählbar sind: `prog1 < eingabe > ausgabe`.

2.2 Ein- und Ausgabe mit Dateien

Die Ein- und Ausgabeoperatoren >> und << sind bei Dateien ebenso wie bei der Standardein- und -ausgabe verwendbar. Für die Funktion `get()` zur Eingabe eines Zeichens gibt es das Gegenstück `put()` zur Ausgabe eines Zeichens. Der Header `<fstream>` enthält die vom Compiler verlangten Beschreibungen für Dateiobjekte. Die benutzten Funktionen `get()`, `put()`, `open()` und `close()` und weitere nutzen die Dateifunktionen des zugrunde liegenden Betriebssystems. Ein Programm, das auf Dateien zugreift, enthält folgende wesentliche Elemente:

- Es wird ein Dateiobjekt mit einem beliebigen Namen definiert, das von nun an im Programm verwendet wird. Der Datentyp des Dateiobjekts ist `ifstream` für Ein- und `ofstream` für Ausgabedateien (Abkürzungen für input file stream beziehungsweise output file stream).
- Die Verbindung des Dateiobjekts zu einer existierenden oder anzulegenden Datei auf Betriebssystemebene wird mit der Funktion `open()` hergestellt, d.h. die Datei wird »geöffnet«. Der externe Dateiname wird der Funktion `open` als Zeichenkette übergeben. Dadurch kann ein und dasselbe Programm auf beliebige Dateien zugreifen. Eine zu beschreibende Datei wird normalerweise *exklusiv* reserviert, kann also nicht gleichzeitig von anderen Programmen beschrieben oder gelesen werden.

- Die Verbindung wird mit der Funktion `close()` wieder gelöst; man sagt auch, dass die Datei geschlossen wird. Ab diesem Zeitpunkt kann die Datei wieder von anderen Benutzern oder Programmen benutzt werden.

Das Schreiben geschieht im Allgemeinen gepuffert, indem in einen dafür reservierten Speicherbereich (Puffer) geschrieben wird, der erst bei Überlauf auf die Festplatte transferiert wird. Für das Lesen gilt Entsprechendes. `close()` sorgt dafür, dass der im Puffer befindliche Rest geschrieben und das Inhaltsverzeichnis (englisch *directory*) aktualisiert wird.

Bei Programmende wird automatisch ein `close()` durchgeführt. Es empfiehlt sich jedoch, eine Datei zu schließen, sobald nichts mehr geschrieben werden soll und noch weitere Programmteile folgen. Die Begründung: Ein irregulärer Programmabbruch, aus welchen Gründen auch immer, verhindert das Schreiben des Pufferinhalts und die Aktualisierung des Inhaltsverzeichnisses auf der Festplatte für eine zum Zeitpunkt des Abbruchs noch geöffnete Datei. Sie ist danach, zum Beispiel für ein weiteres Programm, nur teilweise oder gar nicht mehr lesbar.

Beispiel: Kopieren von Dateien

Das Beispielprogramm zum Kopieren beliebiger Dateien teilt sich in die Abschnitte

- Definieren und Öffnen der Eingabedatei mit Programmabbruch, falls die Datei nicht existiert
- Definieren und Öffnen der Ausgabedatei mit Programmabbruch, falls die Datei existiert, aber nicht beschrieben werden darf (*read-only*-Datei)
- Kopiervorgang – die Schleife wird bei Fehlern oder am Dateiende abgebrochen
- (automatisches) Schließen beider Dateien

Der verwendete Aufruf `exit(int)` beendet das Programm im Fehlerfall, wobei der übergebene Parameter an das Betriebssystem gemeldet wird.

Listing 2.3: Datei kopieren

```
// cppbuch/k2/datcopy.cpp kopiert eine Datei
#include<cstdlib> // für exit( )
#include<fstream>
#include<iostream>
#include<string>
using namespace std;

int main( ) {
    // Definieren der Eingangsdatei
    ifstream quelle;          // Datentyp für Eingabestrom
    string quelldateiname;
    cout << "Quelldatei?";
    cin >> quelldateiname;  // Darf wegen der Eigenschaften von cin
                            // keine Leerzeichen enthalten!
    // Datei öffnen. Zu ios::binary siehe Text.
    quelle.open(quelldateiname.c_str(), ios::binary|ios::in);
    if(!quelle) { // Fehlerabfrage
        cerr << quelldateiname
             << " kann nicht geöffnet werden!\n";
```

```
        exit(-1);
    }
    string zieldateiname;
    cout << "Zieldatei? ";
    cin >> zieldateiname;  // Darf wegen der Eigenschaften von cin
                           // keine Leerzeichen enthalten!
    // Definieren und Öffnen der Ausgabedatei hier in einem Schritt.
    // ios::binary muss bei ofstreams mit ios::out verknüpft werden.
    ofstream ziel(zieldateiname.c_str(), ios::binary|ios::out);
    if(!ziel) {
        cerr << zieldateiname
             << " kann nicht geöffnet werden!\n";
        exit(-1);
    }
    char ch;
    while(quelle.get(ch))
          ziel.put(ch);  // zeichenweise kopieren
}  // Dateien werden am Programmende automatisch geschlossen.
```

Anmerkung zu ios::binary

Ohne `ios::binary` wären nur Textdateien kopierbar. Der Schalter `ios::binary` verhindert, dass die Umwandlung der Zeilenendekennung `\n` in `CR/LF` (= 0x0d 0x0a) automatisch beim Schreiben beziehungsweise zurück beim Lesen erfolgen soll (nur bei MS DOS/Windows und OS/2 von Bedeutung). `ios::binary` muss bei `ifstreams` mit `ios::in` verknüpft werden.

Dateien beliebigen Inhalts (binäre Dateien) haben keine Zeilenstruktur. Die Funktion `c_str()` stellt den Quelldateinamen in einer für `open` verträglichen Form dar, nämlich als C-String. C-Strings werden ab Seite 193 behandelt. Falls das Programm nach Abschluss des Kopierens fortgesetzt werden sollte, wäre die Ergänzung um die Anweisungen `quelle.close();` und `ziel.close();` sinnvoll (vergleiche obige Diskussion). In den Abfragen `if(!quelle)` und `if(!ziel)` wird im Programmbeispiel scheinbar auf die Negation des Dateiobjekts geprüft, um einen Fehler festzustellen. Die Erklärung dafür muss auf den Abschnitt 10.5 verschoben werden.

Tipp

Wenn ein Objekt für eine Eingabedatei *mehrfach* benutzt wird, etwa um mehrere Dateien einzulesen, sind nach dem `close()` die objektinternen Statusbits mit `clear()` zu löschen.

Der Grund: Beim Aufruf von `close()` wird `clear()` leider nicht von jedem Compiler automatisch ausgeführt, sodass das Objekt, hier `quelle` genannt, sich möglicherweise noch am Ende einer Datei wähnt (EOF = end of file), obwohl es schon ein neues `open()` gab. Beispiel:

```
quelle.open("ersteDatei.txt");
// ... hier die Datei verarbeiten
quelle.close();  // Datei schließen
quelle.clear();  // Statusbits löschen nicht vergessen!
```

```
quelle.open("zweiteDatei.txt");
// ... hier die nächste Datei verarbeiten usw.
```

Übungen

2.1 (Fast) Jeder braucht mal einen Kredit. Schreiben Sie ein Programm zur Berechnung eines Tilgungsplans für einen Ratenkredit, damit Sie gegebenenfalls Ihre Bank kontrollieren können. Es sei vorausgesetzt, dass die Rate monatlich gezahlt wird. Das Programm verlange die folgenden Eingaben: Kreditumfang (in Euro), nominaler Zinssatz in Prozent (pro Jahr), Anfangsmonat, Anfangsjahr, Höhe der monatlichen Rate, Laufzeit des Kredits in Jahren. Der Tilgungsplan soll in eine Datei *tilgungsplan.txt* ausgegeben werden. Von der Rate werden zunächst die Zinsen beglichen, der Rest wird zur Tilgung der Restschuld verwendet. Daraus ergibt sich, dass die Rate größer als die anfänglichen Zinsen sein muss, wenn die Restschuld kleiner werden soll. Die Laufzeit ist die Zinsbindungsfrist, nach deren Ablauf die Restschuld zurückgezahlt oder ein neuer Zinssatz festgelegt wird. Damit das Programm in der Praxis nutzbar sein kann, muss kaufmännisch auf ganze Cent-Beträge gerundet werden. Die auszugebende Datei könnte wie folgt aussehen:

```
Anfangsschulden : 20000.00  Zinssatz nominal : 5.00

Zahlmonat    Rate   Zinsen  Tilgung     Rest
  1.2012   500.00    83.33   416.67 19583.33
  2.2012   500.00    81.60   418.40 19164.93

                                      ⋮
```

usw. bis zum letzten Jahr:

```
Zahlmonat    Rate   Zinsen  Tilgung     Rest
  1.2015   500.00    16.05   483.95  3368.84

                                      ⋮

  7.2015   500.00     3.83   496.17   422.53
  8.2015   424.29     1.76   422.53     0.00
Summen:   3924.29    71.50  3852.79 pro Jahr

Gesamt:  21924.29  1924.29 20000.00
```

Bei kleinen Raten endet die Tabelle mit dem Ende der Laufzeit, in diesem Beispiel jedoch wegen der großen Raten schon früher. Hinweis: Die Spaltenbreite kann vor jeder Ausgabe mit `ausgabe.width(10)` eingestellt werden. `ausgabe` sei der zur Datei gehörige `ofstream`. Mit

```
ausgabe.setf(ios::showpoint|ios::fixed, ios::floatfield);
ausgabe.precision(2); // Nachkommastellen
```

wird die Ausgabe mit Dezimalpunkt und zwei Nachkommastellen eingestellt.

Mehr dazu lesen Sie in Kapitel 10

2.2 Schreiben Sie ein Progamm, das den Inhalt einer Datei hexadezimal ausdruckt.

2.3 Erweiterung zur vorigen Aufgabe: Erst 16 Buchstaben und dann die zugehörigen 16 Hex-Codes pro Zeile ausgeben.

2.4 Schreiben Sie ein Programm *stat.cpp*, das eine Statistik für eine Textdatei ausgibt, deren Name eingegeben werden soll. Das Ergebnis soll eine Ausgabe folgender Art hervorbringen:

Anzahl der Zeichen = 16437
Anzahl der Worte = 2526
Anzahl der Zeilen = 220

Ein Wort sei hier als ununterbrochene Folge von Buchstaben definiert, wobei Umlaute hier nicht als Buchstaben zählen sollen, weil sie nicht Teil des ASCII-Zeichensatzes sind. In der Anzahl der Zeichen soll die Zeilenendekennung nicht enthalten sein.