PEARSON Education

ADDISON-WESLEY

Stanley B. Lippman
Josée Lajoie
Barbara E. Moo

C++ Primer

Vierte Auflage

KAPITEL 3

TYPEN DER BIBLIOTHEK

INHALT

Neben den elementaren Typen, die in Kapitel 2 behandelt wurden, definiert C++ eine umfangreiche Bibliothek abstrakter Datentypen. Die wichtigsten sind `string` und `vector` für Zeichenstrings und Sammlungen unterschiedlicher Länge. Mit ihnen sind Begleittypen verknüpft, so genannte Iteratoren, die zum Zugriff auf die Zeichen eines Strings oder die Elemente eines Vektors benutzt werden. Diese Bibliothekstypen sind Abstraktionen elementarer, in die Sprache integrierter Typen – Arrays und Zeiger.

Ein weiterer Bibliothekstyp mit der Bezeichnung `bitset` bietet eine abstrakte Möglichkeit, eine Sammlung von Bits zu bearbeiten, die bequemer ist als die der bitweise agierenden Operatoren für Integerwerte.

In diesem Kapitel werden die Bibliothekstypen `vector`, `string` und `bitset` eingeführt, im nächsten Arrays und Zeiger und in Kapitel 5 die bitweise arbeitenden integrierten Operatoren.

Die in Kapitel 2 eingeführten Typen sind maschinennah: Sie stehen für Abstraktionen wie Zahlen oder Zeichen und sind dadurch definiert, wie sie diese auf dem Rechner darstellen.

Außer den in der Sprache definierten Typen enthält die Standardbibliothek eine Reihe höherer **abstrakter Datentypen**, die komplexere Konzepte wiedergeben. Sie sind abstrakt, weil wir uns nicht darum zu kümmern brauchen, wie sie dargestellt werden, wenn wir sie benutzen, sondern nur darum, welche Operationen sie unterstützen.

Zwei wichtige Bibliothekstypen sind `string` und `vector`. Der Typ `string` unterstützt Zeichenstrings variabler Länge, der Typ `vector` nimmt eine Reihe von Objekten des angegebenen Typs auf. Gegenüber den einfacheren von der Sprache definierten Typen stellen sie eine Verbesserung dar. In Kapitel 4 betrachten wir die integrierten Konstrukte, die ihnen ähneln, aber weniger flexibel und fehleranfälliger sind.

Ein weiterer Bibliothekstyp, der eine bequemere und angemessen effiziente Abstraktion eines integrierten Merkmals darstellt, ist die Klasse `bitset`, mit der die Behandlung eines Werts als Bitsammlung möglich ist, also eine direktere Bearbeitung der Bits als die Operatoren, um die es in Abschnitt 5.3 (S. 194) geht.

Bevor wir mit den Bibliothekstypen fortfahren, sehen wir uns einen Mechanismus an, der den Zugriff auf die in der Bibliothek definierten Namen vereinfacht.

3.1 Using-Deklarationen

Die bisher vorgestellten Programme verweisen auf Namen aus der Bibliothek, indem sie den Namensbereich `std` explizit erwähnen. Wenn wir zum Beispiel von der Standardeingabe lesen wollen, schreiben wir `std::cin`. Namen dieser Art verwenden den Bereichsoperator `::` (Abschnitt 1.2.2, S. 26), der besagt, dass wir den Namen des rechten Operanden im Gültigkeitsbereich des linken finden. Die Formulierung `std::cin` bedeutet also, dass wir den Namen `cin` meinen, der im Namensbereich `std` definiert ist. Durch diese Schreibweise auf Bibliotheksnamen zu verweisen kann lästig sein.

Glücklicherweise gibt es einfachere Methoden, Elemente aus Namensbereiche zu verwenden. In diesem Abschnitt zeigen wir die einfachste: **Using-Deklarationen**. Andere finden Sie in Abschnitt 17.2 (S. 830).

Eine Using-Deklaration ermöglicht den Zugriff auf einen Namen aus einem Namensbereich ohne das Präfix `namespace-name::` und hat folgende Form:

```
using namespace::name;
```

Nach der Deklaration können wir direkt auf *name* zugreifen, ohne auf den Namensbereich zu verweisen:

```
#include <string>
#include <iostream>
// Die Using-Deklarationen erklären unsere Absicht,
// die folgenden Namen aus dem Namensbereich std
// zu verwenden.
using std::cin;
using std::string;
int main()
{
   string s; // O.K.: string ist jetzt Synonym
             // für std::string.
```

```
    cin >> s; // O.K.: string ist jetzt Synonym für std::string.
    cout << s; // Fehler: Keine Using-Deklaration. Wir müssen
               // den vollständigen Namen benutzen.
    std::cout << s; // O.K.: Explizite Verwendung von cout
                    // aus dem Namensbereich std.
}
```

Die Verwendung einer nicht qualifizierten Version eines Namens aus einem Namensbereich ohne Using-Deklaration stellt einen Fehler dar, obwohl manche Compiler ihn nicht entdecken.

Eine eigene Using-Deklaration für jeden Namen

Eine Using-Deklaration stellt jeweils nur ein Element eines Namensbereichs vor. Sie ermöglicht uns, sehr genau festzulegen, welche Namen wir in unseren Programmen benutzen. Wenn wir mehrere Namen aus `std` – oder einem anderen Namensbereich – verwenden wollen, müssen wir für jeden eine Using-Deklaration vornehmen. Wir können das Additionsprogramm von Seite 26 zum Beispiel folgendermaßen umschreiben:

```
#include <iostream>
// Using-Deklarationen für Namen aus der
// Standardbibliothek
using std::cin;
using std::cout;
using std::endl;
int main()
{
   cout << "Enter two numbers:" << endl;
   int v1, v2;
   cin >> v1 >> v2;
   cout << "The sum of " << v1
      << " and " << v2
      << " is " << v1 + v2 << endl;
   return 0;
}
```

Die Using-Deklarationen für `cin`, `cout` und `endl` bedeuten, dass wir diese Namen ohne das Präfix `std::` verwenden können, was den Code leichter lesbar macht.

Von hier an gehen unsere Beispiele davon aus, dass Using-Deklarationen für die im Code verwendeten Namen aus der Standardbibliothek vorliegen. Wir schreiben also in den Text- und Codebeispielen nicht mehr `std::cin`, sondern `cin`. Um die Beispiele kurz zu halten, zeigen wir die für die Kompilierung erforderlichen Deklarationen nicht. Außerdem verzichten wir auch auf die erforderlichen `#include`-Direktiven. In Tabelle A.1 (S. 936) in Anhang A sind die Bibliotheksnamen und die dazugehörigen Header für Namen aus der Standardbibliothek aufgeführt, die wir in dieser Einführung verwenden.

Sie müssen daran denken, die jeweiligen `#include`-Direktiven und Using-Deklarationen in die Beispiele einzufügen, bevor Sie sie kompilieren.

Klassendefinitionen mit Typen der Standardbibliothek

Es gibt einen Fall, in dem wir *grundsätzlich* die vollständig qualifizierten Bibliotheksnamen verwenden sollten: in Headerdateien. Der Inhalt eines Headers wird ja vom Präprozessor in unseren Programmtext kopiert. Wenn wir eine Headerdatei einbinden, wird der Text des Headers zum Bestandteil unserer Datei. Steht nun im Header eine Using-Deklaration, ist es so, als ob wir diese in jedes Programm schrieben, das den Header einbindet, *ob das Programm sie benötigt oder nicht*.

Im Allgemeinen ist es angebracht, in Headern nur das zu definieren, was wirklich erforderlich ist.

ÜBUNGEN ZUM ABSCHNITT 3.1

Übung 3.1: Schreiben Sie das Programm aus Abschnitt 2.3 (S. 69), das die Potenz einer bestimmten Zahl berechnet, so um, dass es nicht über das Präfix `std::` auf Bibliotheksnamen zugreift, sondern entsprechende Using-Deklarationen verwendet.

3.2 Der Bibliothekstyp string

Der Typ `string` unterstützt Zeichenstrings variabler Länge. Die Bibliothek verwaltet den zum Speichern der Zeichen zugewiesenen Platz und stellt verschiedene hilfreiche Operationen bereit. Der Typ ist effizient genug, um allgemein verwendet zu werden.

Wie bei jedem Bibliothekstyp müssen Programme, die Strings verwenden, zuerst den entsprechenden Header einbinden. Unsere Programme werden kürzer, wenn wir außerdem eine Using-Deklaration bereitstellen:

```
#include <string>
using std::string;
```

3.2.1 Strings definieren und initialisieren

Tabelle 3.1: Methoden zur Initialisierung von Strings

`string s1;`	Standardkonstruktor; `s1` ist der leere String.
`string s2(s1);`	Initialisiert `s2` als Kopie von `s1`.
`string s3("value");`	Initialisiert `s3` als Kopie des Stringliterals.
`string s4(n, 'c');`	Initialisiert `s4` mit `n` Kopien des Zeichens `'c'`.

Die Stringbibliothek stellt mehrere Konstruktoren (Abschnitt 2.3.3, S. 74) bereit, besondere Elementfunktionen, die definieren, wie Objekte des betreffenden Typs initialisiert werden können. In Tabelle 3.1 finden Sie die häufigsten Stringkonstruktoren. Der Standardkonstruktor (Abschnitt 2.3.4, S. 77) wird »standardmäßig« verwendet, wenn kein Initialisierer angegeben ist.

WARNUNG: DER BIBLIOTHEKSTYP STRING UND STRINGLITERALE

Aus historischen Gründen und zur Kompatibilität mit C sind Zeichenstringliterale *nicht* derselbe Typ wie der Typ `string` der Standardbibliothek. Dieser Umstand kann Verwirrung verursachen und muss bei der Verwendung von Stringliteralen oder des Datentyps `string` berücksichtigt werden.

ÜBUNGEN ZUM ABSCHNITT 3.2.1

Übung 3.2: Was ist ein Standardkonstruktor?

Übung 3.3: Nennen Sie die drei Möglichkeiten, den Datentyp `string` zu initialisieren.

Übung 3.4: Wie lauten die Werte von `s` und `s2`?

```
string s;
int main() {
    string s2;
}
```

3.2.2 Strings lesen und schreiben

Wie wir in Kapitel 1 gesehen haben, benutzen wir die `iostream`-Bibliothek zum Lesen und Schreiben von Werten integrierter Typen wie `int`, `double` usw. In gleicher Weise können wir die `iostream`- und die `String`-Bibliothek verwenden, um mit Hilfe des Standardein- und Ausgabeoperators Strings zu lesen und zu schreiben:

```
// Hinweis: #include und using-Deklarationen
// müssen eingefügt werden, um diesen Code
// kompilieren zu können.
int main()
{
   string s; // Leerer String
   cin >> s; // Liest den durch Leerraum getrennten
             // String in s ein.
   cout << s << endl; // Schreibt s an die Ausgabe.
   return 0;
}
```

Dieses Programm beginnt mit der Definition des Strings `s`. Die nächste Zeile liest die Standardeingabe und legt das Gelesene in `s` ab:

```
cin >> s;   // Liest den durch Leerraum getrennten String
            // in s ein.
```

Der Stringeingabeoperator geht wie folgt vor:

- Er liest und verwirft führenden Leerraum (zum Beispiel Leerzeichen, Zeilenumbrüche, Tabulatoren).
- Anschließend liest er Zeichen ein, bis er auf das nächste Leerraumzeichen trifft.

Wenn die Eingabe für dieses Programm " Hello World! " lautet (beachten Sie die führenden und die angehängten Leerzeichen), wird also "Hello" ohne zusätzliche Zeichen ausgegeben.

Die Ein- und Ausgabeoperationen verhalten sich genauso wie die Operatoren für die integrierten Typen. Insbesondere geben sie ihren linken Operanden als Ergebnis zurück. Daher können wir mehrere Lese- oder Schreibvorgänge verketten:

```
string s1, s2;
cin >> s1 >> s2;            // Liest erst die Eingabe in s1
                            // dann die in s2.

cout << s1 << s2 << endl;   // Schreibt beide Strings.
```

Wenn wir dieser Version des Programms dieselben Eingaben liefern wie im vorherigen Absatz, lautet die Ausgabe wie folgt:

```
Hello World!
```

Um dieses Programm kompilieren zu können, müssen Sie `#include`-Direktiven für die `iostream`- und die `string`-Bibliothek sowie Using-Deklarationen für alle verwendeten Namen aus der Bibliothek einfügen, also für `string`, `cin`, `cout` und `endl`.
Von jetzt an gehen wir in den Beispielpogrammen davon aus, dass die benötigten Deklarationen vorgenommen wurden.

Eine unbekannte Anzahl von Strings lesen

Wie die Eingabeoperatoren für die integrierten Typen gibt der Stringeingabeoperator den Stream zurück, den er gelesen hat. Deshalb können wir ihn als Bedingung verwenden, wie wir es getan haben, als wir in das Programm auf Seite 39 Integerwerte eingelesen haben. Das folgende Programm liest eine Reihe von Strings von der Standardeingabe und schreibt das Gelesene, jeweils einen String pro Zeile, an die Standardausgabe:

```
int main()
{
   string word;
   // Liest bis EOF und schreibt jedes Wort in eine neue Zeile.
   while (cin >> word)
      cout << word << endl;
   return 0;
}
```

In diesem Fall lesen wir mit dem Eingabeoperator Daten in einen String ein. Der Operator gibt den gelesenen Eingabestream zurück und die `while`-Bedingung testet ihn, nachdem der Lesevorgang abgeschlossen ist. Wenn der Stream gültig ist – also weder

auf EOF noch auf eine ungültige Eingabe gestoßen ist –, wird der Rumpf der `while`-Schleife ausgeführt und der eingelesene Wert an die Standardausgabe gegeben. Sobald wir EOF erreichen, verlassen wir die Schleife.

Eine komplette Zeile mit getline lesen

Es gibt eine weitere sinnvolle E/A-Operation für Strings: `getline`. Diese Funktion übernimmt einen Eingabestream und einen String. Sie liest die nächste Eingabezeile vom Stream und legt das Gelesene *ohne* den Zeilenumbruch in ihrem Stringargument ab. Anders als der Eingabeoperator ignoriert sie führende Zeilenumbrüche nicht. Sobald sie auf einen Zeilenumbruch trifft – selbst, wenn es sich um das erste Zeichen der Zeile handelt – hört sie auf zu lesen und kehrt zurück. Ein Zeilenumbruch als erstes Zeichen der Eingabe bewirkt, dass das Stringargument auf den leeren String gesetzt wird.

Die Funktion `getline` gibt ihr `istream`-Argument zurück, so dass sie wie der Eingabeoperator als Bedingung benutzt werden kann. Wir können das vorhergehende Programm, das ein Wort pro Zeile ausgab, so umschreiben, dass es stattdessen jeweils eine Zeile ausgibt:

```
int main()
{
   string line;
   // Liest bis EOF jeweils eine Zeile.
   while (getline(cin, line))
      cout << line << endl;
   return 0;
}
```

Da `line` keinen Zeilenumbruch enthält, müssen wir einen einfügen, wenn die Strings jeweils in eine eigene Zeile geschrieben werden sollen. Wie üblich benutzen wir `endl`, um einen Zeilenumbruch einzugeben und den Ausgabepuffer zu leeren.

Der Zeilenumbruch, der `getline` veranlasst, die Steuerung zurückzugeben, wird verworfen und nicht im Stringobjekt abgelegt.

ÜBUNGEN ZUM ABSCHNITT 3.2.2

Übung 3.5: Schreiben Sie ein Programm, das die Standardeingabe zeilenweise einliest. Ändern Sie es so, dass es wortweise liest.

Übung 3.6: Erläutern Sie, wie Leerraumzeichen im Stringeingabeoperator und in der Funktion `getline` gehandhabt werden.

3.2.3 Operationen mit Strings

In Tabelle 3.2 sind die häufigsten Stringoperationen aufgeführt.

Die Stringoperationen size und empty

Die Länge eines Strings ist die Anzahl seiner Zeichen. Sie wird von der Operation `size` zurückgegeben:

```
int main(
{
   string st("The expense of spirit\n");
   cout << "The size of " << st << "is " << st.size()
      << " characters, including the newline" << endl;
   return 0;
}
```

Tabelle 3.2: Stringoperationen

`s.empty()`	Gibt `true` zurück, wenn `s` leer ist, sonst `false`.
`s.size()`	Gibt die Anzahl der Zeichen in `s` zurück.
`s[n]`	Gibt das Zeichen an Position `n` in `s` zurück; Start bei `0`.
`s1 + s2`	Gibt einen String zurück, der aus der Verkettung von `s1` und `s2` besteht.
`s1 = s2`	Ersetzt die Zeichen in `s1` durch eine Kopie von `s2`.
`v1 == v2`	Gibt `true` zurück, wenn `v1` und `v2` gleich sind, sonst `false`.
`!=`, `<`, `<=`, `>` und `>=`	Haben ihre normale Bedeutung.

Wenn wir dieses Programm kompilieren und ausführen, gibt es Folgendes aus:

```
The size of the expense of spirit
is 22 characters, including the newline
```

Häufig ist es sinnvoll zu wissen, ob ein String leer ist. Eine Möglichkeit bietet der Vergleich von `size` mit `0`:

```
if (st.size() == 0)
   // O.K.: leer
```

In diesem Fall brauchen wir nicht wirklich zu wissen, wie viele Zeichen der String enthält, sondern sind nur daran interessiert, ob die Größe gleich null ist. Mit dem Element `empty` können wir die Frage direkter beantworten:

```
if (st.empty())
   // O.K.: leer
```

Die Funktion `empty` gibt den Boole'schen Wert (Abschnitt 2.1, S. 58) `true` zurück, wenn der String keine Zeichen enthält, sonst `false`.

string::size_type

Es wäre logisch zu erwarten, dass die Operation `size` einen Wert vom Typ `int` oder, wenn man an den Hinweis auf Seite 63 denkt, einen vorzeichenlosen Typ zurückgibt. Tatsächlich benutzt sie aber den Typ `string::size_type`, der der Erläuterung bedarf.

Die Klasse `string` definiert – wie viele andere Bibliothekstypen – mehrere Begleittypen, die die rechnerunabhängige Verwendung der Bibliothekstypen möglich machen. Der Typ `size_type` gehört dazu. Er ist als Synonym für einen Typ ohne Vorzeichen definiert, entweder `unsigned int` oder `unsigned long`, der garantiert groß genug für die Größe jeden beliebigen Strings ist. Um den von `string` definierten Typ `size_type` zurückzugeben, verwenden wir den Bereichsoperator, der kennzeichnet, dass dieser Typ in der Klasse `string` definiert ist.

Jede Variable, in der das Ergebnis der Operation `size` für einen String gespeichert werden soll, muss vom Typ `string::size_type` sein. Besonders wichtig ist es, den Rückgabewert *nicht* einer Variablen vom Typ `int` zuzuweisen.

Auch wenn wir den genauen Typ von `string::size_type` nicht kennen, wissen wir, dass es sich um einen Typ ohne Vorzeichen (Abschnitt 2.1.1, S. 58) handelt. Außerdem wissen wir, dass die vorzeichenlose Version eines bestimmten Typs doppelt so große positive Werte aufnehmen kann wie der entsprechende Typ mit Vorzeichen, woraus sich ergibt, dass der größte Wert, den eine Variable vom Typ `string` aufnehmen kann, doppelt so groß sein kann wie bei einer Variablen vom Typ `int`.

Ein weiteres Problem bei der Verwendung einer Variablen vom Typ `int` liegt darin, dass diese Variablen auf manchen Rechnern zu klein sind, um selbst Strings von plausibler Länge aufzunehmen. Wenn sie auf einem Rechner beispielsweise 16 Bit lang sind, können sie höchstens einen String von 32.767 Zeichen aufnehmen. Ein String mit dem Inhalt einer Datei kann ohne Weiteres länger sein. Die sicherste Möglichkeit besteht darin, den Typ zu benutzen, den die Bibliothek für diesen Zweck definiert: `string::size_type`.

Relationale Operatoren von String

Die Klasse `string` definiert mehrere Operatoren, die zwei Werte dieses Typs vergleichen. Alle vergleichen dazu die einzelnen Zeichen.

Vergleiche von Strings berücksichtigen Groß- und Kleinschreibung – die große und die kleine Version eines Buchstaben sind unterschiedliche Zeichen. Auf den meisten Rechnern kommt der Großbuchstabe zuerst: Jeder Großbuchstabe ist kleiner als ein Kleinbuchstabe.

Der Gleichheitsoperator vergleicht zwei Strings und gibt `true` zurück, wenn sie gleich sind. Dies ist der Fall, wenn sie gleich lang sind und dieselben Zeichen enthalten. Außerdem definiert die Bibliothek den Operator `!=` zum Prüfen der Ungleichheit.

Die relationalen Operatoren `<`, `<=`, `>` und `>=` prüfen, ob ein String kleiner, kleiner oder gleich, größer oder größer oder gleich einem anderen ist:

```
string big = "big", small = "small";
string s1 = big;    // s1 ist eine Kopie von big.
if (big == small)   // Falsch
                    // ...
if (big <= s1)      // Wahr, sie sind gleich, big ist also
kleiner oder gleich s1.
                    // ...
```

Die relationalen Operatoren vergleichen Strings nach demselben Verfahren wie ein Wörterbuch (das Groß-/Kleinschreibung berücksichtigt):

- Wenn zwei Strings verschieden lang sind und jedes Zeichen im kürzeren gleich dem entsprechenden im längeren ist, ist der kürzere String kleiner als der längere.
- Wenn sich die Zeichen zweier Strings unterscheiden, vergleichen wir sie, indem wir das erste unterschiedliche Zeichen vergleichen.

Nehmen wir beispielsweise folgende Strings:

```
string substr = "Hello";
string phrase = "Hello World";
string slang = "Hiya";
```

Dort ist `substr` kleiner als `phrase` und `slang` größer als `substr` und als `phrase`.

Zuweisung für Strings

Im Allgemeinen sollen Bibliothekstypen so einfach zu verwenden sein wie integrierte. Dazu unterstützen die meisten Bibliothekstypen die Zuweisung. Was Strings betrifft, können wir ein Objekt dieses Typs einem anderen zuweisen:

```
// st1 ist ein leerer String, st2 eine Kopie des Literals.
string st1, st2 = "The expense of spirit";
st1 = st2;      // st1 wird durch eine Kopie von st2 ersetzt.
```

Nach der Zuweisung enthält `st1` eine Kopie der Zeichen in `st2`.

Die meisten Implementierungen der Stringbibliothek haben einige Mühe, um effiziente Operationen wie die Zuweisung bereitzustellen, wobei erwähnt werden muss, dass die Zuweisung von der Idee her einiges an Arbeit erfordert. Der Speicher, in dem die Zeichen von `st1` stehen, muss geleert werden, der für die Kopie der Zeichen von `st2` benötigte Speicher muss zugewiesen werden und dann müssen die Zeichen in den neuen Speicher kopiert werden.

Zwei Strings addieren

Die Addition von Strings ist als Verkettung definiert, d.h. zwei oder mehr Strings lassen sich mit Hilfe des Additions- (+) oder des zusammengesetzten Zuweisungsoperators (+=) addieren (Abschnitt 1.4.1, S. 32). Aus den beiden folgenden Strings ...

```
string s1("hello, ");
string s2("world\n");
```

... können wir durch Verkettung den folgenden String bilden:

```
string s3 = s1 + s2;   // s3 lautet hello, world\n
```

Wenn wir `s2` direkt an `s1` anhängen wollen, benutzen wir +=:

```
s1 += s2;   // Äquivalent mit s1 = s1 + s2
```

Strings und Stringliterale addieren

Die Strings `s1` und `s2` enthielten die Satzzeichen direkt. Dasselbe Ergebnis können wir erreichen, indem wir Stringobjekte und Stringliterale wie folgt mischen:

```
string s1("hello");
string s2("world");
string s3 = s1 + ", " + s2 + "\n";
```

Dabei muss mindestens ein Operand für jeden Additionsoperator vom Typ `string` sein:

```
string s1 = "hello";      // Keine Satzzeichen
string s2 = "world";
string s3 = s1 + ", ";    // O.K.: Addition eines
                          // Strings und eines Literals
string s4 = "hello" + ", ";         // Fehler: Kein Operand
                                    // vom Typ string
string s5 = s1 + ", " + "world";    // O.K.: Jeder
                                    // Additionsoperator hat
                                    // einen Operanden vom Typ
                                    // string.
string s6 = "hello" + ", " + s2;    // Fehler: Stringliterale
                                    // können nicht addiert
                                    // werden.
```

Die Initialisierung von `s3` und `s4` umfasst jeweils nur eine Operation. In diesen Fällen ist leicht festzustellen, dass die Initialisierung von `s3` zulässig ist: Wir initialisieren `s3` durch Addition eines Strings und eines Stringliterals. Die Initialisierung von `s4` versucht, zwei Stringliterale zu addieren, und ist damit unzulässig.

Die Initialisierung von `s5` mag überraschend wirken, funktioniert aber weitgehend genauso wie die Verkettung von Ein- und Ausgabeoperationen (Abschnitt 1.2, S. 26). In diesem Fall legt die Stringbibliothek fest, dass eine Addition einen Wert vom Typ `string` zurückgibt. Wenn wir `s5` initialisieren, gibt der Teilausdruck `s1 + ","` daher den Typ `string` zurück, der sich mit dem Literal `"world\n"` verketten lässt, als ob wir Folgendes geschrieben hätten:

```
string tmp = s1 + ", ";   // O.K.: + hat einen Operanden
                          // vom Typ string
s5 = tmp + "world ";      // O.K.: + hat einen Operanden
                          // vom Typ string
```

Die Initialisierung von `s6` ist dagegen nicht zulässig. Wenn wir die einzelnen Teilausdrücke betrachten, sehen wir, dass der erste zwei Stringliterale addiert. Das darf nicht sein, und deshalb stellt die Anweisung einen Fehler dar.

Ein Zeichen aus einem String abrufen

Der Typ `string` benutzt den **Indexoperator** (`[]`), um auf einzelne Zeichen eines Strings zuzugreifen. Er übernimmt einen Wert des Typs `size_type`, der die Position des abzurufenden Zeichens angibt. Der enthaltene Wert wird häufig als »**Index**« bezeichnet.

Indizes für Strings beginnen mit 0. Wenn `s` ein String ist, ist `s[0]`, falls der String nicht leer ist, das erste, `s[1]` das zweite und `s[s.size() - 1]` das letzte Zeichen des Strings.
Einen Index außerhalb dieses Bereichs anzugeben ist ein Fehler.

Wir können den Indexoperator einsetzen, um jedes Zeichen des Strings in einer eigenen Zeile auszugeben:

```
string str("some string");
for (string::size_type ix = 0; ix != str.size(); ++ix)
   cout << str[ix] << endl;
```

Bei jedem Schleifendurchlauf rufen wir das nächste Zeichen aus `str` ab und geben es aus, worauf ein Zeilenumbruch folgt.

Indizierung führt zu einem Lvalue

Erinnern Sie sich daran, dass eine Variable ein Lvalue ist (Abschnitt 2.3.1, S. 70) und dass die linke Seite einer Zuweisung ein Lvalue sein muss. Wie eine Variable ist auch der vom Indexoperator zurückgegebene Wert ein Lvalue. Daher kann ein Index auf beiden Seiten einer Zuweisung verwendet werden. Die folgende Schleife setzt jedes Zeichen in `str` auf ein Sternchen:

```
for (string::size_type ix = 0; ix != str.size(); ++ix)
   str[ix] = '*';
```

Indexwerte berechnen

Jeder Ausdruck, der einen Integerwert ergibt, kann als Index für den Indexoperator verwendet werden. Vorausgesetzt, dass `someval` und `someotherval` Integerobjekte sind, können wir Folgendes schreiben:

```
str[someotherval * someval]= someval;
```

Auch wenn jeder Integertyp als Index benutzt werden kann, ist der tatsächliche Typ des Index `string::size_type`, also ein vorzeichenloser Typ.

Dieselben Gründe, aus denen `string::size_type` als Typ der Variablen für den Rückgabewert von `size` verwendet wird, gelten auch für die Definition einer Variablen für einen Index. Eine Variable zum Indizieren eines Strings sollte vom Typ `string::size_type` sein.

Wenn wir einen String indizieren, müssen wir dafür sorgen, dass er »im Bereich« liegt. Damit meinen wir, dass der Index eine Zahl ist, die bei Zuweisung an eine Variable vom Typ `size_type` im Bereich von 0 bis zur Länge des Strings minus 1 liegt. Durch Verwendung von `string::size_type` oder einen anderen vorzeichenlosen Typ stellen wir sicher, dass der Index nicht negativ sein kann. Solange unser Index zu einem vorzeichenlosen Typ gehört, brauchen wir nur zu prüfen, ob er kleiner ist als die Stringlänge.

Die Bibliothek ist nicht verpflichtet, den Wert des Index zu prüfen. Die Verwendung eines bereichsüberschreitenden Werts ist undefiniert und führt normalerweise zu einem schweren Laufzeitfehler.

3.2.4 Umgang mit den Zeichen eines Strings

Häufig wollen wir die einzelnen Zeichen eines Strings verarbeiten. Wir wollen zum Beispiel wissen, ob ein bestimmtes Zeichen ein Leerraumzeichen ist oder ob es sich um einen Buchstaben oder eine Zahl handelt. In der Tabelle 3.3 finden Sie die Funktionen, die auf die Zeichen eines Strings (oder auf einen anderen Wert vom Typ `char`) angewendet werden können. Sie sind im Header `cctype` definiert.

Meistens prüfen diese Funktionen das angegebene Zeichen und geben einen Wert vom Typ `int` zurück, der als Wahrheitswert fungiert. Alle Funktionen geben beim Fehlschlag des Tests 0 zurück, andernfalls einen (bedeutungslosen) Wert ungleich null, der besagt, dass das Zeichen zur geprüften Art gehört.

Für diese Funktionen ist ein druckbares Zeichen ein Zeichen, das sichtbar dargestellt wird. Zu den Leerraumzeichen gehören Leerzeichen, Tabulator, Vertikaltabulator, Umbruch, Zeilenwechsel und Seitenwechsel. Satzzeichen sind druckbare Zeichen, die keine Zahl, kein Buchstabe und kein (druckbares) Leerraumzeichen wie das Leerzeichen sind.

Tabelle 3.3: `cctype`-Funktionen

`isalnum(c)`	`true`, wenn `c` ein Buchstabe oder eine Ziffer ist.
`isalpha(c)`	`true`, wenn `c` ein Buchstabe ist.
`iscntrl(c)`	`true`, wenn `c` ein Steuerzeichen ist.
`isdigit(c)`	`true`, wenn `c` eine Ziffer ist.
`isgraph(c)`	`true`, wenn `c` kein Leerzeichen, aber druckbar ist.
`islower(c)`	`true`, wenn `c` ein Kleinbuchstabe ist.
`isprint(c)`	`true`, wenn `c` ein druckbares Zeichen ist.
`ispunct(c)`	`true`, wenn `c` ein Satzzeichen ist.
`isspace(c)`	`true`, wenn `c` ein Leerraumzeichen ist.
`isupper(c)`	`true`, wenn `c` ein Großbuchstabe ist.
`isxdigit(c)`	`true`, wenn `c` eine Hexadezimalziffer ist.
`tolower(c)`	Gibt den entsprechenden Kleinbuchstaben zurück, wenn `c` ein Großbuchstabe ist, andernfalls das unveränderte `c`.
`toupper(c)`	Gibt den entsprechenden Großbuchstaben zurück, wenn `c` ein Kleinbuchstabe ist, andernfalls das unveränderte `c`.

Wir können diese Funktionen beispielsweise benutzen, um die Anzahl der Satzzeichen in einem bestimmten String zurückzugeben:

```
string s("Hello World!!!");
string::size_type punct_cnt = 0;
// Anzahl der Satzzeichen in s
for (string::size_type index = 0; index != s.size(); ++index)
   if (ispunct(s[index]))
      ++punct_cnt;
cout << punct_cnt
   << " punctuation characters in " << s << endl;
```

Die Ausgabe sieht folgendermaßen aus:

```
3 punctuation characters in Hello World!!!
```

Die Funktionen `tolower` und `toupper` geben keinen Wahrheitswert zurück, sondern ein Zeichen – entweder das unveränderte Argument oder die kleine oder große Version des Zeichens. Wir können `s` mit Hilfe von `tolower` in Kleinbuchstaben umwandeln:

```
// Wandelt in Kleinbuchstaben um.
for (string::size_type index = 0; index != s.size(); ++index)
    s[index] = tolower(s[index]);
cout << s <<endl;
```

... gibt die folgende Zeile aus:

```
hello world!!!
```

RATSCHLAG: VERWENDEN SIE DIE C++-VERSIONEN DER HEADER AUS DER BIBLIOTHEK

Außer den eigens für C++ geschriebenen Funktionen enthält die C++-Bibliothek auch die C-Bibliothek. Der Header `cctype` macht die in der C-Headerdatei `ctype.h` definierten C-Bibliotheksfunktionen verfügbar.

Die Standardheadernamen von C verwenden die Form *`name.h`*. Die C++-Versionen dieser Header heißen *`cname`* – sie lassen die Endung `.h` weg und setzen den Buchstaben `c` vor den Namen, der besagt, dass der Header aus der C-Bibliothek stammt. Daher weist `cctype` denselben Inhalt auf wie `ctype.h`, aber in einer für C++-Programme geeigneten Form. Insbesondere sind die in den *`cname`*-Headern definierten Namen im Namensbereich `std` definiert, die in den `.h`-Versionen dagegen nicht.

Normalerweise sollten C++-Programme die *`cname`*-Versionen der Header benutzen, nicht die *`name`*`.h`-Versionen. Auf diese Weise stehen die Namen aus der Standardbibliothek konsistent im Namensbereich `std`. Wenn die *`name`*`.h`-Header verwendet werden, müssen die Programmierer daran denken, welche Bibliotheksnamen von C geerbt und welche eigens für C++ geschrieben sind.

ÜBUNGEN ZUM ABSCHNITT 3.2.4

Übung 3.7: Schreiben Sie ein Programm, das zwei Strings einliest und meldet, ob sie gleich sind. Falls nicht, soll es feststellen, welcher größer ist. Ändern Sie das Programm dann so, dass es ermittelt, ob die Strings gleich lang sind, und ggf. ausgibt, welcher länger ist.

Übung 3.8: Schreiben Sie ein Programm, das Strings von der Standardeingabe liest und zu einem langen String verkettet. Geben Sie den verketteten String aus. Ändern Sie das Programm dann so, dass es aufeinander folgende Eingabestrings durch ein Leerzeichen trennt.

Übung 3.9: Was macht das folgende Programm? Ist es gültig? Wenn nicht, warum?

```
string s;
cout << s[0] << endl;
```

Übung 3.10: Schreiben Sie ein Programm, das die Satzzeichen aus einem String entfernt. Die Eingabe soll ein String mit Satzzeichen sein, die Ausgabe ein String, in dem die Satzzeichen fehlen.

3.3 Der Bibliothekstyp vector

Ein Vektor ist eine Sammlung von Objekten desselben Typs, die jeweils mit einem Integerindex verknüpft sind. Wie bei Strings verwaltet die Bibliothek den zugewiesenen Speicher. Wir bezeichnen den Vektor als **Container**, weil er andere Objekte enthält. Alle Objekte in einem Container müssen zum selben Typ gehören. Wesentlich mehr über Container erfahren Sie in Kapitel 9.

Um einen Vektor benutzen zu können, müssen wir den betreffenden Header einbinden. In unseren Beispielen gehen wir außerdem davon aus, dass eine passende Using-Deklaration vorgenommen wurde:

```
#include <vector>
using std::vector;
```

Ein Vektor ist ein **Klassentemplate**. Templates ermöglichen es, eine Klassendefinition zu schreiben, die von einer Vielzahl von Typen benutzt werden kann. Daher können wir einen Vektor definieren, der Objekte vom Typ `string`, `int` oder einem von uns definierten Klassentyp wie `Sales_item` enthält. Wie wir eigene Klassentemplates definieren, sehen wir in Kapitel 16. Glücklicherweise brauchen wir nur wenig darüber zu wissen, um sie benutzen zu können.

Um Objekte eines Typs zu deklarieren, der aus einem Klassentemplate stammt, müssen wir je nach Template zusätzliche Informationen bereitstellen. Bei Vektoren müssen wir den Typ der Objekte angeben, die der Container enthalten soll. Dazu setzen wir den Typ in spitze Klammern, die auf den Namen des Templates folgen:

```
vector<int> ivec;     // ivec enthält Objekte vom Typ int.
vector<Sales_item> Sales_vec;   // Enthält Objekte vom Typ
                                // Sales_item.
```

Wie bei jeder Variablendefinition geben wir einen Typ und eine Liste mit einer oder mehreren Variablen an. In der ersten Definition heißt der Typ `vector<int>`, ist also ein Vektor, der Objekte vom Typ `int` enthält. Der Name der Variablen lautet `ivec`. In der zweiten Zeile definieren wir die Variable `Sales_vec` für Objekte vom Typ `Sales_item`.

Bei `vector` handelt es sich nicht um einen Typ, sondern um ein Template, mit dessen Hilfe wir eine beliebige Anzahl von Typen definieren können. Jeder dieser Typen legt einen Elementtyp fest. `vector<int>` und `vector<string>` sind also Typen.

3.3.1 Vektoren definieren und initialisieren

Die Klasse `vector` definiert mehrere Konstruktoren (Abschnitt 2.3.3, S. 74), die wir zur Definition und Initialisierung ihrer Objekte benutzen. Sie sind in Tabelle 3.4 aufgeführt.

Tabelle 3.4: Möglichkeiten der Initialisierung eines Vektors

`vector<T> v1;`	Vektor für Objekte vom Typ `T`; Standardkonstruktor `v1` ist leer.
`vector<T> v2(v1);`	`v2` ist eine Kopie von `v1`.
`vector<T> v3(n, i);`	`v3` hat `n` Elemente mit dem Wert `i`.
`vector<T> v4(n);`	`v4` hat `n` Exemplare eines mit einem Wert initialisierten Objekts.

Eine festgelegte Anzahl von Elementen erstellen

Wenn wir einen nicht leeren Vektor anlegen, müssen wir einen oder mehrere Werte zur Initialisierung der Elemente angeben. Kopieren wir dagegen einen Vektor in einen anderen, wird jedes Element des neuen Vektors als Kopie des entsprechenden Elements im Ursprungsvektor initialisiert. Die beiden Vektoren müssen Elemente desselben Typs enthalten:

```
vector<int> ivec1;        // ivec1 enthält Objekte vom Typ int.
vector<int> ivec2(ivec1);       // O.K.: Kopiert Elemente von
                                // ivec1 in ivec2.
vector<string> svec(ivec1);     // Fehler: svec enthält
                                // Objekte vom Typ string,
                                // nicht vom Typ int.
```

Wir können einen Vektor mit einem Zähler und einem Elementwert initialisieren. Der Konstruktor ermittelt mit Hilfe des Zählers, wie viele Elemente der Vektor haben soll, und legt als deren Wert den angegebenen fest:

```
vector<int> ivec4(10, -1);        // 10 mit -1 initialisierte
                                  // Elemente
vector<string> svec(10, "hi!");   // 10 mit "hi!"
                                  // initialisierte Strings
```

SCHLÜSSELKONZEPT: DYNAMISCHES VEKTORWACHSTUM

Eine zentrale Eigenschaft der Vektoren (und der übrigen Bibliothekscontainer) ist das Erfordernis, sie so zu implementieren, dass zur Laufzeit effizient Elemente hinzugefügt werden können: Da Vektoren effizient wachsen, ist es normalerweise am günstigsten, sie dynamisch nach und nach durch Hinzufügen von Elementen zu vergrößern, sobald die Elementwerte jeweils bekannt sind.

Wie wir in Kapitel 4 sehen werden, unterscheidet sich dieses Verhalten wesentlich von dem der integrierten Arrays in C und den meisten anderen Sprachen. Insbesondere könnten an C oder Java gewöhnte Leser, davon ausgehen, dass es günstig wäre, von vornherein Vektoren mit der erwarteten Größe zuzuweisen, weil die Vektorelemente zusammenhängend gespeichert werden. Tatsächlich ist das Gegenteil der Fall, und zwar aus Gründen, die wir in Kapitel 9 erläutern.

Wir können einem Vektor zwar eine bestimmte Anzahl von Elementen zuweisen, aber normalerweise ist es effizienter, einen leeren Vektor zu definieren und die Elemente später hinzuzufügen (was wir in Kürze lernen werden).

Wertinitialisierung

Wenn wir keinen Initialisierer für ein Element angeben, legt die Bibliothek einen **mit einem Wert initialisierten** Elementinitialisierer an, der die einzelnen Elemente im Container initialisiert. Sein Wert hängt vom Typ der gespeicherten Elemente ab.

Wenn der Vektor Elemente eines integrierten Typs enthält, beispielsweise `int`, legt die Bibliothek einen Initialisierer mit dem Wert 0 an:

```
vector<string> fvec(10);   // 10 mit 0 initialisierte Elemente
```

Wenn er Werte eines Klassentyps wie zum Beispiel `string` enthält, der eigene Konstruktoren definiert, verwendet die Bibliothek den Standardkonstruktor des Werttyps zum Anlegen des Elementinitialisierers:

```
vector<string> svec(10);   // 10 Elemente, alle leere Strings
```

Wie wir in Kapitel 12 sehen werden, definieren einige Klassen zwar eigene Konstruktoren, aber keinen Standardkonstruktor. Einen Vektor eines solchen Typs können wir nicht durch ausschließliche Angabe einer Größe initialisieren, sondern müssen auch einen Anfangswert für die Elemente angeben.

Es gibt noch eine dritte Möglichkeit: Der Elementtyp kann ein Klassentyp sein, der überhaupt keine Konstruktoren definiert. In diesem Fall legt die Bibliothek trotzdem ein mit einem Wert initialisiertes Objekt an, indem sie jedes Element des Objekts mit einem Wert initialisiert.

ÜBUNGEN ZUM ABSCHNITT 3.3.1

Übung 3.11: Welche der folgenden Vektordefinitionen sind unzulässig?

```
(a) vector< vector<int> > ivec;
(b) vector<string> svec = ivec;
(c) vector<string> svec(10, "null");
```

Übung 3.12: Wie viele Elemente haben die folgenden Vektoren? Wie lauten deren Werte?

```
(a) vector<int> ivec1;
(b) vector<int> ivec2(10);
(c) vector<int> ivec3(10, 42);
(d) vector<string> svec1;
(e) vector<string> svec2(10);
(f) vector<string> svec3(10, "hello");
```

3.3.2 Operationen mit Vektoren

Die Vektorbibliothek stellt verschiedene Operationen bereit, von denen viele den Stringoperationen ähneln. In Tabelle 3.5 sind die wichtigsten Vektoroperationen aufgeführt.

Tabelle 3.5: Vektoroperationen

`v.empty()`	Gibt `true` zurück, wenn `v` leer ist, sonst `false`.
`v.size()`	Gibt die Anzahl der Elemente in `v` zurück.
`v.push_back(t)`	Hängt ein Element mit dem Wert `t` an das Ende von `v` an.
`v[n]`	Gibt das Element an der Position `n` in `v` zurück.
`v1 = v2`	Ersetzt die Elemente in `v1` durch eine Kopie der Elemente in `v2`.
`v1 == v2`	Gibt `true` zurück, wenn `v1` und `v2` gleich sind.
`!=, <, <=, >` und `>=`	Haben ihre normale Bedeutung.

Die Größe eines Vektors

Die Operationen `empty` und `size` laufen ähnlich ab wie die gleichnamigen Stringoperationen (Abschnitt 3.2.3, S. 114). Das Element `size` gibt einen Wert des `size_type`-Typs zurück, der vom entsprechenden Vektortyp definiert ist.

Um `size_type` verwenden zu können, müssen wir den Typ angeben, in dem er definiert ist. Ein Vektortyp schließt *immer* den Elementtyp des Vektors ein:

```
vector<int>::size_type   // O.K.
vector::size_type        // Fehler
```

Elemente zu einem Vektor hinzufügen

Die Operation `push_back` übernimmt einen Elementwert und hängt ihn als neues Element an das Ende des Vektors an. Sie »schiebt« also sozusagen ein Element »hinten« in den Vektor hinein:

```
// Liest Wörter von der Standardeingabe und speichert sie
// als Elemente in einem Vektor.
string word;
vector<string> text;       // Leerer Vektor
while (cin >> word) {
   text.push_back(word);   // Hängt word an text an.
}
```

Diese Schleife liest eine Reihe von Strings von der Standardeingabe und hängt sie nacheinander an das Ende des Vektors an. Zuerst definieren wir `text` als leeren Vektor. Jeder Schleifendurchlauf fügt dem Vektor ein neues Element hinzu und gibt ihm den Wert des eingelesenen Worts. Nach Abschluss der Schleife hat `text` so viele Elemente, wie eingelesen wurden.

Einen Vektor indizieren

Die Objekte im Vektor sind nicht benannt, sondern der Zugriff erfolgt über ihre Position im Vektor. Wir können ein Element mit Hilfe des Indexoperators abrufen. Die Indizierung eines Vektors erfolgt ähnlich wie bei einem String (Abschnitt 3.2.3, S. 114).

Der Indexoperator für Vektoren übernimmt einen Wert und gibt das Element an der entsprechenden Position zurück. Die Elemente werden mit 0 beginnend nummeriert. Das folgende Beispiel setzt die einzelnen Elemente des Vektors mit Hilfe einer `for`-Schleife auf 0 zurück:

```
// Setzt die Elemente des Vektors auf 0 zurück.
for (vector<int>::size_type ix = 0; ix != ivec.size(); ++ix)
    ivec[ix] = 0;
```

Wie der Indexoperator für Strings führt auch der für Vektoren zu einem Lvalue, so dass wir in ihn schreiben dürfen, wie es im Rumpf der Schleife geschieht. Außerdem verwenden wir wie bei Strings den `size_type-Typ` des Vektors als Typ für den Index.

Auch wenn `ivec` leer ist, wird diese `for`-Schleife korrekt ausgeführt. Der Aufruf von `size` gibt dann 0 zurück, und der Test in der Schleife vergleicht `ix` mit `0`. Da `ix` beim ersten Durchlauf gleich 0 ist, schlägt der Test fehl, weshalb der Schleifenrumpf nicht ein einziges Mal ausgeführt wird.

Indizierung führt nicht zu neuen Elementen

Neue C++-Programmierer meinen manchmal, die Indizierung eines Vektors füge Elemente hinzu, was aber nicht stimmt:

```
vector<int> ivec;    // Leerer Vektor
for (vector<int>::size_type ix = 0; ix != 10; ++ix)
    ivec[ix] = ix;   // Katastrophe: ivec hat keine Elemente.
```

SCHLÜSSELKONZEPT: SICHERE, GENERISCHE PROGRAMMIERUNG

Programmierer, die von C oder Java zu C++ kommen, sind möglicherweise überrascht, dass unsere Schleife nicht < verwendet, sondern `!=`, um den Index mit der Vektorgröße zu vergleichen. C-Programmierer sind wahrscheinlich auch überrascht, dass wir das Element `size` innerhalb der `for`-Schleife aufrufen, anstatt es einmal vor der Schleife aufzurufen und uns dann den Wert zu merken.

C++-Programmierer schreiben Schleifen gewohnheitsmäßig lieber mit `!=` als mit <. In diesem Fall gibt es keinen besonderen Grund, den einen oder den anderen Operator zu bevorzugen. Wir verstehen den Hintergrund für diese Gewohnheit, wenn wir in Teil II die generische Programmierung behandeln.

`size` `aufzurufen`, anstatt sich den Wert zu merken, ist in diesem Fall ähnlich unnötig, aber wiederum eine gute Gewohnheit. In C++ können Datenstrukturen wie Vektoren dynamisch wachsen. Unsere Schleife liest nur Elemente ein, fügt sie aber nicht hinzu. Eine Schleife kann jedoch ohne Weiteres Elemente hinzufügen. Wenn sie es täte, schlüge die Prüfung eines gespeicherten Werts von `size` fehl – unsere Schleife hätte die neu hinzugefügten Elemente nicht berücksichtigt. Da eine Schleife Werte hinzufügen kann, schreiben wir sie lieber so, dass sie bei jedem Durchlauf die aktuelle Größe prüft, anstatt eine Kopie der ursprünglichen Größe zu speichern.

Wie wir in Kapitel 7 sehen werden, lassen sich in C++ Funktionen als `inline` deklarieren. Der Compiler erweitert den Code einer solchen Funktion wenn möglich direkt, anstatt tatsächlich einen Funktionsaufruf durchzuführen. Kleine Bibliotheksfunktionen wie `size` sind nahezu sicher als `inline` definiert, so dass wir für den Aufruf bei jedem Schleifendurchlauf nur geringe Laufzeitkosten erwarten.

Dieser Code sollte 10 neue Elemente in `ivec` einfügen und ihnen die Werte von 0 bis 9 zuweisen. Der Vektor ist jedoch leer, und mit Hilfe der Indizierung können nur vorhandene Elemente abgerufen werden.

Richtig wird die Schleife wie folgt geschrieben:

```
for (vector<int>::size_type ix = 0; ix != 10; ++ix)
    ivec.push_back(ix);   // O.K.: Fügt ein neues Element mit
                          // dem Wert ix hinzu.
```

Ein Element muss vorhanden sein, damit es indiziert werden kann. Bei der Zuweisung über einen Index werden keine Elemente hinzugefügt.

3.4 Einführung in Iteratoren

Wir können zwar über Indizes auf die Elemente eines Vektors zugreifen, aber die Bibliothek stellt noch eine andere Methode zu ihrer Untersuchung bereit: Wir können einen **Iterator** einsetzen, also einen Typ, mit dem wir die Elemente in einem Container untersuchen und von einem Element zum nächsten gehen können.

Die Bibliothek definiert für jeden Standardcontainer, so auch für Vektoren, einen Iterator. Sie sind allgemeiner als Indizes: Alle Bibliothekscontainer definieren Iteratortypen, aber nur wenige unterstützen Indizes. Da alle Container Iteratoren haben, neigen moderne C++-Programme dazu, sie anstelle von Indizes für den Zugriff auf Containerelemente zu verwenden, selbst bei solchen Typen wie Vektoren, die Indizes unterstützen.

VORSICHT: GREIFEN SIE ÜBER DEN INDEX NUR AUF ELEMENTE ZU, VON DEREN EXISTENZ SIE WISSEN!

Es ist von entscheidender Bedeutung, dass der Indexoperator `[]` nur zum Abrufen tatsächlich vorhandener Elemente eingesetzt werden kann. Ein Beispiel:

```
vector<int> ivec;          // Leerer Vektor
cout << ivec[0];           // Fehler: ivec hat keine Elemente!
vector<int> ivec2(10);     // Vektor mit 10 Elementen
cout << ivec[10];  // Fehler: ivec hat die Elemente 0, ...9.
```

Der Versuch, ein nicht vorhandenes Element abzurufen, führt zu einem Laufzeitfehler. Wie bei den meisten Fehlern dieser Art gibt es keine Garantie, dass die Implementierung ihn bemerkt. Das Ergebnis der Programmausführung ist ungewiss. Die Folgen des Abrufs eines nicht vorhandenen Elements sind undefiniert – was geschieht, hängt von der Implementierung ab, aber nahezu sicher schlägt das Programm zur Laufzeit auf irgendeine bemerkenswerte Art fehl.

Diese Warnung gilt für jede Benutzung von Indizes, beispielsweise bei der Indizierung von Strings und, wie wir demnächst sehen werden, bei der Indizierung von Arrays.

Der Versuch der Indizierung nicht vorhandener Elemente ist leider ein häufiger und gefährlicher Programmierfehler. Fehler durch so genannten »Pufferüberlauf« sind das Ergebnis. Sie bilden die häufigsten Ursachen für Sicherheitsprobleme in PC- und anderen Anwendungen.

ÜBUNGEN ZUM ABSCHNITT 3.3.2

Übung 3.13: Lesen Sie eine Reihe von Integerzahlen in einen Vektor ein. Berechnen Sie die Summe von jeweils zwei aufeinander folgenden Elementen und geben Sie sie aus. Wenn die Anzahl der Elemente ungerade ist, teilen Sie dies dem Benutzer mit und geben den letzten Wert ohne Addition aus. Ändern Sie das Programm nun so, dass es zuerst die Summe des ersten und des letzten, dann die Summe des zweiten und des vorletzten Elements ausgibt usw.

Übungen zum Abschnitt 3.3.2 (Forts.)

Übung 3.14: Lesen Sie Text in einen Vektor ein und speichern Sie jedes Wort der Eingabe als Vektorelement. Verwandeln Sie die Wörter in Großbuchstaben. Geben Sie die umgewandelten Elemente in Zeilen zu je acht Wörtern aus.

Übung 3.15: Ist das folgende Programm zulässig? Wenn nicht, wie lässt es sich korrigieren?

```
vector<int> ivec;
ivec[0] = 42;
```

Übung 3.16: Nennen Sie drei Möglichkeiten, einen Vektor mit 10 Elementen zu definieren, die alle den Wert 42 haben. Geben Sie an, ob eine Methode zu bevorzugen ist, und begründen Sie dies.

Die Einzelheiten der Funktionsweise von Iteratoren werden in Kapitel 11 behandelt, wir können sie jedoch benutzen, ohne sie in ihrer ganzen Komplexität zu verstehen.

Iteratortypen von Containern

Jeder Containertyp, beispielsweise `vector`, definiert seinen eigenen Iteratortyp:

```
vector<int>::iterator iter;
```

Diese Anweisung definiert eine Variable `iter` mit dem von `vector<int>` definierten Typ `iterator`. Jeder Bibliothekscontainertyp definiert ein Element `iterator` als Synonym für den eigentlichen Typ seines Iterators.

Terminologie: Iteratoren und Iteratortypen

Bei der ersten Begegnung kann die Nomenklatur rund um Iteratoren verwirrend wirken, insbesondere dadurch, dass der Begriff *Iterator* für zwei Dinge verwendet wird. Wir sprechen sowohl von dem allgemeinen Begriff des Iterators als auch von dem konkreten Typ `iterator`, den ein Container wie `vector<int>` definiert.

Verstehen müssen Sie dabei vor allem, dass es eine Reihe von Typen gibt, die als Iteratoren dienen und konzeptionell verwandt sind. Wir bezeichnen einen Typ als Iterator, wenn er eine bestimmte Gruppe von Aktionen unterstützt, mit denen wir uns zwischen den Elementen eines Containers bewegen und auf ihre Werte zugreifen können.

Jede Containerklasse definiert einen eigenen Iteratortyp, über den der Zugriff auf die Elemente des Containers erfolgt, d.h. jeder Container definiert einen Typ mit dem Namen `iterator`, der die Aktionen eines (konzeptuellen) Iterators unterstützt.

Die Operationen begin und end

Jeder Container definiert ein Funktionspaar `begin` und `end`, das Iteratoren zurückgibt. Der von `begin` zurückgegebene Iterator verweist auf das erste Element des Containers, falls es vorhanden ist:

```
vector<int>::iterator iter = ivec.begin();
```

Diese Anweisung initialisiert `iter` mit dem Wert, den die Vektoroperation `begin` zurückgegeben hat. Vorausgesetzt, der Vektor ist nicht leer, verweist `iter` nach der Initialisierung auf dasselbe Element wie `ivec[0]`.

Der von der Operation `end` zurückgegebene Iterator steht an der »ersten Stelle hinter dem Ende« des Vektors. Er wird häufig als **endüberschreitender Iterator** bezeichnet, was andeutet, dass er auf ein nicht existierendes Element verweist. Wenn der Vektor leer ist, geben `begin` und `end` denselben Iterator zurück.

Der von der Operation `end` zurückgegebene Iterator verweist nicht auf ein echtes Vektorelement, sondern dient als **Wächter**, der angibt, wann alle Elemente des Vektors verarbeitet sind.

Dereferenzierung und Inkrementierung von Vektor-Iteratoren

Mit Hilfe der Operationen auf Iteratortypen können wir das Element abrufen, auf das der Iterator verweist, und den Iterator von einem Element zu einem anderen verschieben.

Iteratortypen greifen über den **Dereferenzierungsoperator** (`*`) auf das Element zu, auf das der Iterator verweist:

```
*iter = 0;
```

Der Dereferenzierungsoperator gibt das Element zurück, das der Iterator aktuell bezeichnet. Wenn `iter` auf das erste Element des Vektors verweist, bezeichnet `*iter` dasselbe Element wie `ivec[0]`. Mit dieser Anweisung wird dem Element 0 zugewiesen.

Mit dem Inkrementoperator (`++`) (Abschnitt 1.4.1, S. 32) wird der Iterator zum nächsten Element des Vektors verschoben. Diese Operation ähnelt logisch der an Objekten vom Typ `int`, bei denen der Wert »um 1 erhöht« wird. Bei Iteratoren findet eine »Verschiebung im Container um eine Stelle nach vorn« statt. Verweist `iter` auf das erste Element, bezeichnet `++iter` also das zweite.

Da der von `end` zurückgegebene Iterator kein Element bezeichnet, kann er nicht inkrementiert oder dereferenziert werden.

Andere Iteratoroperationen

Ein weiteres sinnvolles Operationspaar für Iteratoren stellt der Vergleich dar: Zwei Iteratoren können mit == oder `!=` verglichen werden. Iteratoren sind gleich, wenn sie auf dasselbe Element verweisen, sonst ungleich.

Ein Programm mit Iteratoren

Nehmen wir an, wir haben einen Vektor namens `ivec` mit Elementen vom Typ `int`, die wir auf 0 zurücksetzen wollen. Das lässt sich durch Indizierung erreichen:

```
// Setzt alle Elemente in ivec auf 0 zurück.
for (vector<int>::size_type ix = 0; ix != ivec.size(); ++ix)
      ivec[ix] = 0;
```

Dieses Programm benutzt eine `for`-Schleife, um alle Elemente von `ivec` zu durchlaufen. Sie definiert einen Index, den sie bei jedem Durchlauf inkrementiert. Der Schleifenrumpf setzt jedes Element auf 0.

Die üblichere Methode, diese Schleife zu schreiben, verwendet Iteratoren:

```
// Äquivalente Schleife mit Iteratoren zum Zurücksetzen
// aller Elemente von ivec auf 0.

for (vector<int>::iterator iter = ivec.begin();
                           iter != ivec.end(); ++iter)
    *iter = 0;   // Setzt das Element, auf das iter verweist,
                 // auf 0.
```

Zuerst definiert die `for`-Schleife den Iterator `iter` und initialisiert ihn mit dem ersten Element von `ivec`. Die Bedingung der Schleife prüft, ob `iter` ungleich dem von der Operation `end` zurückgegebenen Iterator ist. Jeder Durchlauf inkrementiert `iter`. Die Schleife beginnt mit dem ersten Element und verarbeitet alle Elemente des Vektors nacheinander. Zum Schluss verweist `iter` auf das letzte Element von `ivec`. Nach der Verarbeitung des letzten Elements und dem Hochsetzen von `iter` ist der Iterator gleich dem von `end` zurückgegebenen Wert. An dieser Stelle wird die Schleife beendet.

Die Anweisung im Rumpf der `for`-Schleife greift über den Dereferenzierungsoperator auf den Wert des aktuellen Elements zu. Wie der Indexoperator gibt auch der Dereferenzierungsoperator einen Lvalue zurück. Wir können den Wert des Elements durch eine Zuweisung ändern. Die Wirkung der Schleife besteht darin, jedem Element von `ivec` den Wert null zuzuweisen.

Wenn wir den Code im Einzelnen untersuchen, stellen wir fest, dass dies genau der Version mit Indizes entspricht: Wir fangen mit dem ersten Element des Vektors an und setzen alle Elemente auf null.

Dieses Programm ist wie das auf Seite 125 auch dann sicher, wenn der Vektor leer ist. Dann bezeichnet der von `begin` zurückgegebene Iterator kein Element, weil es keine Elemente gibt. Er ist derselbe wie der von `end` zurückgegebene, so dass die Prüfung in der `for`-Schleife sofort fehlschlägt.

const_iterator

Das vorstehende Programm hat die Werte im Vektor mit Hilfe von `vector:: iterator` geändert. Außerdem definiert jeder Container einen Typ `const_iterator`, der zum Lesen, aber nicht zum Schreiben von Containerelementen benutzt werden sollte.

Dereferenzieren wir einen einfachen Iterator, erhalten wir eine nicht konstante Referenz (Abschnitt 2.5, S. 86) auf das Element. Beim Dereferenzieren des Typs `const_iterator` wird dagegen eine Referenz auf ein konstantes Objekt (Abschnitt 2.4, S. 83) zurückgegeben. Wie bei jeder konstanten Variablen können wir den Wert des Elements nicht überschreiben.

Wenn `text` beispielsweise ein Vektor mit Elementen vom Typ `string` ist, wollen wir ihn durchlaufen und die Elemente ausgeben. Dazu können wir folgendermaßen vorgehen:

```
// Verwendet const_iterator, weil wir die Werte
// nicht ändern wollen.
```

```
for (vector<string>::const_iterator iter = text.begin();
                                    iter != text.end(); ++iter)
   cout << *iter << endl;   // Gibt jedes Element von text aus.
```

Diese Schleife ähnelt der vorherigen mit der Ausnahme, dass wir den Wert aus dem Iterator lesen, anstatt ihm etwas zuzuweisen. Da wir den Iterator nicht überschreiben, definieren wir `iter` als `const_iterator`. Wenn wir einen Iterator vom Typ `const_iterator` dereferenzieren, wird ein konstanter Wert zurückgegeben. Mit einem solchen Iterator sind keine Zuweisungen an ein Element möglich:

```
for (vector<string>::const_iterator iter = text.begin();
                                    iter != text.end(); ++iter)
    *iter = " ";  // Fehler: *iter ist konstant.
```

Wenn wir den Typ `const_iterator` verwenden, bekommen wir einen Iterator, dessen eigener Wert geändert werden kann, der aber seinerseits den Wert des zugrunde liegenden Elements nicht ändern kann. Wir können ihn inkrementieren und mit Hilfe des Dereferenzierungsoperators den Wert auslesen, aber keinen Wert zuweisen.

Ein Iterator vom Typ `const_iterator` darf nicht mit einem konstanten Iterator verwechselt werden. Wenn wir einen Iterator als konstant deklarieren, müssen wir ihn initialisieren. Anschließend können wir seinen Wert nicht mehr ändern:

```
vector<int> nums(10);   // nums ist nicht konstant.
const vector<int>::iterator cit = nums.begin();
*cit = 1;          // O.K.: cit kann das zugrunde liegende
                   // Element ändern.
++cit;             // Fehler: Der Wert von cit kann nicht
                   // geändert werden.
```

Der Typ `const_iterator` kann in Verbindung mit konstanten und nicht konstanten Vektoren benutzt werden, weil er keine Elemente überschreiben kann. Ein konstanter Iterator ist weitgehend nutzlos: Nach der Initialisierung können wir dafür sorgen, dass er das Element überschreibt, auf das er verweist, aber nicht dafür, dass er auf ein anderes Element verweist.

```
const vector<int> nines(10, 9);   // Elemente in nines
                                  // können nicht geändert
                                  // werden.
// Fehler: cit2 könnte das Element ändern, auf das es verweist,
// nines ist jedoch konstant.
const vector<int>::iterator cit2 = nines.begin();
// O.K.: it kann keinen Elementwert ändern, so dass es
// in Verbindung mit einem konstanten vector<int>
// verwendet werden kann.
vector<int>::const_iterator it = nines.begin();
*it = 10; // Fehler: *it ist konstant.
++it;     // O.K.: it ist nicht konstant,
          // so dass wir seinen Wert ändern können.
```

Der folgende Iterator kann keine Elemente überschreiben:

```
vector<int>::const_iterator
```

Der Wert des folgenden Iterators kann nicht geändert werden:

```
const vector<int>::iterator
```

ÜBUNGEN ZUM ABSCHNITT 3.4

Übung 3.17: Wiederholen Sie die Übungen zu Abschnitt 3.3.2 (S. 124) und verwenden Sie dabei zum Zugriff auf die Elemente von `vector` Iteratoren statt Indizes.

Übung 3.18: Schreiben Sie ein Programm, das einen Vektor mit 10 Elementen anlegt. Weisen Sie jedem Element mit Hilfe eines Iterators einen Wert zu, der doppelt so groß ist wie der aktuelle.

Übung 3.19: Prüfen Sie das Programm aus Übung 3.18, indem Sie den Vektor ausgeben.

Übung 3.20: Erläutern Sie, welchen Iterator Sie in den Programmen der vorhergehenden Übungen verwendet haben, und begründen Sie Ihre Wahl.

Übung 3.21: Wann verwenden Sie einen konstanten Iterator, wann einen Iterator vom Typ `const_iterator`? Erläutern Sie den Unterschied.

3.4.1 Iteratorarithmetik

Außer dem Inkrementoperator, der den Iterator um jeweils ein Element weiterschiebt, unterstützen Vektoriteratoren (aber nur wenige Iteratoren anderer Bibliothekscontainer) auch andere arithmetische Operationen, die als **Iteratorarithmetik** bezeichnet werden:

- `iter + n`
  `iter - n`

  Wir können einen Integerwert zu einem Iterator addieren oder von ihm subtrahieren. Damit erhalten wir einen neuen Iterator, der `n` Elemente weiter vorn (Addition) oder hinten (Subtraktion) steht als das Element, auf das `iter` verweist. Das Ergebnis muss auf ein Element im Vektor oder auf eins über ihn hinaus verweisen. Der Typ des addierten oder subtrahierten Werts sollte normalerweise der `size_type` oder der `difference_type` (siehe Folgendes) des Vektors sein.

- `iter1 - iter2`

  Berechnet die Differenz zwischen zwei Iteratoren als Wert eines vorzeichenbehafteten Integertyps mit dem Namen `difference_type`, der wie `size_type` von der Klasse `vector` definiert wird. Der Typ hat ein Vorzeichen, weil eine Subtraktion zu negativen Ergebnissen führen kann. Er ist garantiert groß genug, um die Differenz zwischen zwei beliebigen Iteratoren aufzunehmen. Sowohl `iter1` als auch `iter2` müssen auf ein Element innerhalb des Vektors oder auf das Element verweisen, das um eins über den Vektor hinausgeht.

Mit Iteratorarithmetik können wir einen Iterator direkt auf ein Element setzen. Die Mitte des Vektors lässt sich zum Beispiel wie folgt ermitteln:

```
vector<int>::iterator mid = vi.begin() + vi.size() / 2;
```

Dieser Code initialisiert `mid` mit dem Element, das der Mitte von `ivec` am nächsten liegt. Es ist effizienter, diesen Iterator direkt zu berechnen, als ein äquivalentes Programm zu schreiben, das den Iterator inkrementiert, bis er das mittlere Element erreicht hat.

Jede Operation, die die Größe eines Vektors ändert, macht vorhandene Iteratoren ungültig. Nach dem Aufruf von `push_back` sollten Sie sich zum Beispiel nicht auf den Wert eines Iterators für den Vektor verlassen.

ÜBUNGEN ZUM ABSCHNITT 3.4.1

Übung 3.22: Was geschieht, wenn wir `mid` folgendermaßen berechnen:

```
vector<int>::iterator mid = (vi.begin() + vi.end()) / 2;
```

3.5 Der Bibliothekstyp bitset

Manche Programme handhaben Sammlungen von Bits. Jedes Bit enthält entweder den Wert 0 (aus) oder den Wert 1 (ein). Die Verwendung von Bits stellt eine kompakte Methode für Ja/Nein-Informationen zu mehreren Punkten oder Bedingungen dar (manchmal Flags genannt). Die Standardbibliothek macht mit Hilfe der Klasse `bitset` den Umgang mit Bits einfach. Um ein Bitset zu benutzen, müssen wir die entsprechende Headerdatei einbinden. In unseren Beispielen gehen wir außerdem davon aus, dass eine Using-Deklaration für `std::bitset` vorgenommen wurde:

```
#include <bitset>
using std::bitset
```

3.5.1 Bitsets definieren und initialisieren

In Tabelle 3.6 sind die Konstruktoren von `bitset` aufgeführt. Wie `vector` ist auch `bitset` ein Klassentemplate. Anders als bei `vector` unterscheiden sich die Objekte von `bitset` aber nicht im Typ, sondern in der Größe. Bei der Definition eines Bitsets legen wir fest, wie viele Bits es enthält, indem wir die Größe in spitze Klammern schreiben:

```
bitset<32> bitvec; // 32 Bits, alle 0
```

Die Größe muss ein konstanter Ausdruck sein (Abschnitt 2.7, S. 90). Sie kann wie hier als integrale Literalkonstante oder als konstantes Objekt eines integralen Typs definiert werden, der mit einer Konstante initialisiert wird.

Die Anweisung definiert `bitvec` als Bitset mit 32 Bits. Wie die Elemente eines Vektors sind auch die Bits in einem Bitset nicht benannt, sondern werden über ihre Position angesprochen, wobei die Nummerierung mit 0 beginnt. Das Bitset `bitvec` enthält also die Bits 0 bis 31. Die Bits, die bei 0 anfangen, werden als **niederwertige**, diejenigen, die mit 31 enden, als **höherwertige Bits** bezeichnet.

Tabelle 3.6: Möglichkeiten zur Initialisierung eines Bitsets

`bitset<n> b;`	`b` hat `n` Bits mit dem Wert 0.
`bitset<n> b(u);`	`B` ist eine Kopie des Werts `u` vom Typ `unsigned long`.
`bitset<n> b(s);`	`b` ist eine Kopie der im String `s` enthaltenen Bits.
`bitset<n> b(s, pos, n);`	`b` ist eine Kopie der Bits in `n` Zeichen von `s`, beginnend mit Position `pos`.

Initialisierung eines Bitsets mit einem vorzeichenlosen Wert

Wenn wir einen Wert des Typs `unsigned long` als Initialisierer für ein Bitset benutzen, wird dieser Wert als Bitmuster behandelt. Die Bits im Bitset sind eine Kopie dieses Musters. Wenn das Bitset größer ist als die Anzahl der Bits in einem Wert dieses Typs, werden die verbleibenden höherwertigen Bits auf null gesetzt. Ist das Bitset kleiner, werden nur die niederwertigen Bits des Werts verwendet, und die höherwertigen, die die Größe überschreiten, verworfen.

Auf einem Rechner mit Werten des Typs `unsigned long` von 32 Bit wird der Hexadezimalwert `0xffff` in Bitform als Folge von 16 Einsen dargestellt, auf die 16 Nullen folgen. (Die Ziffer `0xf` wird jeweils als 1111 dargestellt.) Wir können ein Bitset mit `0xffff` initialisieren:

```
// bitvec1 ist kleiner als der Initialisierer.
bitset<16> bitvec1(0xffff); // Die Bits 0 bis 15 werden
                            // auf 1 gesetzt.
// bitvec2 hat dieselbe Größe wie der Initialisierer.
bitset<32> bitvec2(0xffff); // Die Bits 0 bis 15 werden auf
                            // 1 gesetzt, die Bits 16 bis
                            // 31 auf 0.
// Die Bits 0 bis 31 werden auf einem 32-Bit-Rechner mit 0xffff
initialisiert.
bitset<128> bitvec3(0xffff); // Die Bits 32 bis 127 werden mit
                             // 0 initialisiert.
```

In allen drei Fällen werden die Bits 0 bis 15 auf 1 gesetzt. Für `bitvec1` werden die höherwertigen Bits des Initialisierers verworfen; das Bitset hat weniger Bits als ein Wert vom Typ `unsigned long`. Das Bitset `bitvec2` ist genauso groß wie ein Objekt vom Typ `unsigned long`, so dass alle Bits zur Initialisierung verwendet werden. `bitvec3` ist größer als ein Wert vom Typ `unsigned long`, so dass die höherwertigen Bits über 31 mit null initialisiert werden.

Initialisierung eines Bitsets mit einem String

Wenn wir ein Bitset mit einem String initialisieren, stellt diese Initialisierung das Bitmuster direkt dar. Die Bits werden *von rechts nach links* vom String gelesen:

```
string strval("1100")
bitset<32> bitvec4(strval);
```

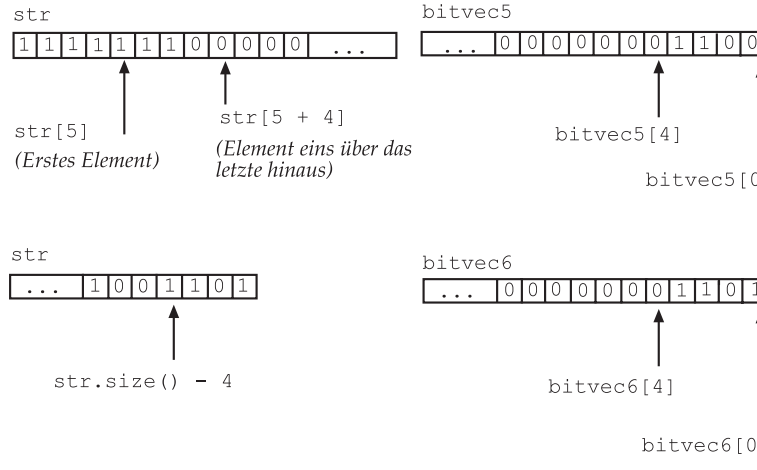
Im Bitmuster in `bitvec4` sind die Bits 2 und 3 auf 1 gesetzt, die übrigen dagegen auf 0. Wenn der String weniger Zeichen enthält, als das Bitset lang ist, werden die höherwertigen Bits auf null gesetzt.

Die Nummerierungskonventionen für Strings und Bitsets sind umgekehrt verknüpft: Das Zeichen ganz rechts im String (das mit dem höchsten Index) dient zur Initialisierung des niedrigsten Bits im Bitset – des Bits mit dem Index 0. Bei der Initialisierung eines Bitsets mit einem String müssen Sie unbedingt an diesen Unterschied denken.

Wir brauchen nicht den gesamten String als Anfangswert für das Bitset zu verwenden, sondern können auch einen Teilstring benutzen:

```
string str("11111110000000011001101");
bitset<32> bitvec5(str, 5, 4);      // 4 Bits von str[5] an,
                                    // also 1100
bitset<32> bitvec6(str, str.size() - 4); // Verwendet die 4
                                         // letzten Zeichen.
```

Hier wird `bitvec5` mit einem Teilstring von `str` initialisiert, der bei `str[5]` anfängt und vier Zeichen lang ist. Wie üblich beginnen wir bei der Initialisierung mit dem rechten Ende des Teilstrings, d.h. wir setzen die Bitpositionen `0` bis `3` auf `1100` und die übrigen auf `0`. Lässt man den dritten Parameter weg, werden die Zeichen von der Startposition bis zum Ende des Strings verwendet. Dann werden die vier niedrigsten Bits von `bitvec6` mit den Zeichen initialisiert, die an der vierten Position von hinten beginnen, die übrigen mit null. Diese Initialisierungen sehen aus wie folgt:



3.5.2 Operationen mit Bitsets

Die Bitsetoperationen (Tabelle 3.7) definieren verschiedene Möglichkeiten zum Testen des Bitsets und zum Setzen einzelner Bits.

Tabelle 3.7: Bitsetoperationen

`b.any()`	Ist ein Bit in `b` auf ein?
`b.none()`	Ist kein Bit in `b` ein?
`b.count()`	Anzahl der Bits in `b`, die ein sind.
`b.size()`	Anzahl der Bits in `b`.
`b[pos]`	Zugriff auf Bit von `b` an Position `pos`.
`b.test(pos)`	Ist das Bit in `b` an Position `pos` ein?
`b.set()`	Alle Bits in `b` auf ein
`b.set(pos)`	Bit in `b` an Position `pos` auf ein.
`b.reset()`	Alle Bits in `b` auf aus.
`b.reset(pos)`	Bit in `b` an Position `pos` auf aus.
`b.flip()`	Zustand aller Bits in `b` umkehren.
`b.flip(pos)`	Zustand des Bits in `b` an Position `pos` umkehren.
`b.to_ulong`	Gibt einen Wert vom Typ `unsigned long` mit denselben Bits wie in `b` zurück.
`os << b`	Gibt die Bits in `b` in den String `os` aus.

Das gesamte Bitset testen

Die Operation `any` gibt `true` zurück, wenn ein oder mehrere Bits des Bitsetobjekts ein – d.h. gleich 1 – sind. Umgekehrt gibt die Operation `none` den Wert `true` zurück, wenn alle Bits des Objekts auf null gesetzt sind.

```
bitset<32> bitvec;                // 32 Bits, alle 0
bool is_set = bitvec.any();       // false, alle Bits auf null
bool is_not_set = bitvec.none();  // true, alle Bits auf null
```

Wenn wir wissen müssen, wie viele Bits gesetzt sind, können wir die Operation `count` benutzen, die diese Anzahl zurückgibt:

```
size_t bits_set = bitvec.count(); // Gibt die Anzahl der Bits
                                  // zurück, die gesetzt sind.
```

Der Rückgabetyp dieser Operation ist ein Bibliothekstyp mit dem Namen `size_t`, der im Header `cstddef` definiert ist, der C++-Version des Headers `stddef.h` aus der C-Bibliothek. Es handelt sich um einen rechnerspezifischen Typ *ohne Vorzeichen*, der garantiert lang genug für die Größe eines Objekts im Speicher ist.

Die Operation `size` gibt wie die gleichnamige Operation für Vektoren und Strings die Gesamtzahl der Bits im Bitset zurück. Der Rückgabewert ist vom Typ `size_t`:

```
size_t sz = bitvec.size();   // Gibt 32 zurück.
```

Auf die Bits eines Bitsets zugreifen

Der Indexoperator lässt uns das Bit an der indizierten Position lesen oder beschreiben. Wir können damit also den Wert eines bestimmten Bits prüfen oder setzen:

```
// Weist Bits mit gerader Nummer den Wert 1 zu.
for (int index = 0; index != 32: index += 2)
            bitvec[index] = 1;
```

Diese Schleife schaltet die Bits mit gerader Nummer ein.
Wie beim Indexoperator können wir mit Hilfe der Operationen `set`, `test` und `reset` einen bestimmten Bitwert prüfen oder setzen:

```
// Äquivalente Schleife, die die Operation set benutzt
for (int index = 0; index != 32: index += 2)
            bitvec.set(index);
```

Um zu prüfen, ob ein Bit ein ist, können wir die Operation `test` einsetzen oder den vom Indexoperator zurückgegebenen Wert untersuchen:

```
if (bitvec.test(i))
    // bitvec[i] ist an.
for (int index = 0; index != 32: index += 2) bitvec.set(index);
    // Äquivalenter Test mit Indizierung
if (bitvec [i])
    // bitvec[i] ist an.
```

Das Testergebnis für den von einem Index zurückgegebenen Wert ist `true`, wenn das Bit 1 ist, und `false`, wenn es 0 ist.

Das gesamte Bitset festlegen

Die Operationen `set` und `reset` können auch benutzt werden, um das gesamte Bitsetobjekt ein- oder auszuschalten:

```
bitvec.reset();   // Setzt alle Bits auf 0.
bitvec.set();     // Setzt alle Bits auf 1.
```

Die Operation `flip` kehrt den Wert eines einzelnen Bits oder des gesamten Bitsets um:

```
bitvec.flip(0);     // Kehrt den Wert des ersten Bits um.
bitvec[0].flip();   // Kehrt ebenfalls das erste Bit um.
bitvec.flip();      // Kehrt den Wert aller Bits um.
```

Den Wert eines Bitsets abrufen

Die Operation `to_ulong` gibt eine Variable des Typs `unsigned long` zurück, die dasselbe Bitmuster enthält wie das Bitsetobjekt. Sie kann nur benutzt werden, wenn das Bitset kleiner oder gleich der Größe der Variablen ist:

```
unsigned long ulong = bitvec3.to_ulong();
cout << "ulong = " << ulong << endl;
```

Die Operation `to_ulong` ist für den Fall gedacht, dass wir einem C- oder C++-Programm vor Standard-C++ ein Bitset übergeben müssen. Wenn das Bitset mehr Bits enthält, als eine Variable vom Typ `unsigned long` aufnehmen kann, wird eine Laufzeitausnahme signalisiert. Ausnahmen werden in Abschnitt 6.13 (S. 265) eingeführt und in Abschnitt 17.1 (S. 804) näher betrachtet.

Die Bits ausgeben

Mit dem Ausgabeoperator können wir die Bits eines Bitsetobjekts ausgeben:

```
bitset<32> bitvec2(0xffff);  // Die Bits 0 bis 15 werden auf 1
                             // gesetzt, die Bits 16 bis 31
                             // auf 0.
cout << "bitvec2: " << bitvec2 << endl;
```

Die Ausgabe sieht aus wie folgt:

```
bitvec2: 00000000000000001111111111111111
```

Bitweise Operatoren verwenden

Außerdem unterstützt die Klasse `bitset` die integrierten bitweise vorgehenden Operatoren. Wie von der Sprache definiert gelten sie für integrale Operanden. Sie ähneln den in diesem Abschnitt beschriebenen Bitsetoperationen und sind in Abschnitt 5.3 (S. 194) beschrieben.

ÜBUNGEN ZUM ABSCHNITT 3.5.2

Übung 3.23: Geben Sie an, welche Bitmuster die folgenden Bitsetobjekte enthalten:

```
(a) bitset<64> bitvec(32);

(b) bitset<32> bv(1010101);

(c) string bstr; cin >> bstr; bitset<8>bv(bstr);
```

Übung 3.24: Betrachten Sie die Zahlenfolge 1, 2, 3, 5, 8, 13, 21. Initialisieren Sie ein `bitset<32>`-Objekt, das ein Bit in jeder Position aufweist, die einer Zahl dieser Folge entspricht. Schreiben Sie alternativ ein kurzes Programm für ein leeres Bitset, das jedes der genannten Bits einschaltet.

ZUSAMMENFASSUNG DES KAPITELS

Die Bibliothek definiert mehrere höhere abstrakte Datentypen, unter anderem Strings und Vektoren. Die Klasse `string` stellt Zeichenstrings variabler Länge bereit, der Typ `vector` verarbeitet eine Sammlung von Objekten eines bestimmten Typs.

Iteratoren ermöglichen den indirekten Zugriff auf Objekte, die in einem Container gespeichert sind. Außerdem werden sie benutzt, um sich zwischen den Elementen von Strings und Vektoren zu bewegen.

Im nächsten Kapitel geht es um Arrays und Zeiger, in die Sprache integrierte Typen. Sie stellen maschinennahe Analogien der Vektor- und der Stringbibliothek bereit. Im Allgemeinen sollten Sie jedoch vorrangig die Bibliothekstypen benutzen.

TERMINOLOGIE

<< Ausgabeoperator Die Bibliothekstypen `string` und `bitset` definieren einen Ausgabeoperator. Der Stringoperator gibt die Zeichen eines Strings aus, der Bitsetoperator das Bitmuster des Bitsets.

:: Bereichsoperator Sucht den Namen seines rechten Operanden im Gültigkeitsbereich des linken. Dient zum Zugriff auf Namen in einem Namensbereich, beispielsweise in `std::cout`, was für den Namen `cout` aus dem Namensbereich `std` steht. Wird auch eingesetzt, um Namen aus einer Klasse zu bezeichnen, zum Beispiel in `string::size_type`, was den in der Klasse `string` definierten Typ `size_type` bedeutet.

*** Dereferenzierungsoperator** Die Iteratortypen definieren ihn so, dass er das Objekt zurückgibt, auf das der Iterator verweist. Die Dereferenzierung gibt einen Lvalue zurück, so dass wir diesen Operator als linken Operanden einer Zuweisung benutzen können. Eine Zuweisung an das Ergebnis einer Dereferenzierung weist dem indizierten Element einen neuen Wert zu.

>> Eingabeoperator Die Bibliothekstypen `string` und `bitset` definieren einen Eingabeoperator. Der Stringoperator liest durch Leerraumzeichen getrennte Zeichenblöcke ein und speichert das Gelesene im rechten (String-)Operanden. Der Bitsetoperator liest eine Bitfolge in seinen Bitsetoperanden ein.

[] Indexoperator Ein von den Typen `string`, `vector` und `bitset` definierter überladener Operator, der zwei Operanden übernimmt: Der linke ist der Name des Objekts, der rechte ein Index. Er ruft das Element ab, dessen Position dem Index entspricht. Indizes werden von null an gezählt – das erste Element ist das Element `0`, das letzte wird mit `obj.size() - 1` indiziert. Der Indexoperator gibt einen Lvalue zurück, so dass wir einen Index als linken Operanden einer Zuweisung einsetzen können. Eine Zuweisung an das Ergebnis einer Indizierung weist dem indizierten Element einen neuen Wert zu.

++ Inkrementoperator Die Iteratortypen definieren ihn so, dass er »1 addiert«, indem er den Iterator auf das nächste Element setzt.

Abstrakter Datentyp Ein Typ, dessen Darstellung verborgen ist. Um einen abstrakten Typ benutzen zu können, müssen wir nur wissen, welche Operationen er unterstützt.

Bitset Klasse aus der Standardbibliothek, die eine Sammlung von Bits entgegennimmt und Operationen zum Prüfen und Setzen der Bits dieser Sammlung bereitstellt.

cctype-Header Von C geerbter Header, der Routinen zum Prüfen von Zeichenwerten enthält. Auf Seite 119 finden Sie eine Liste der häufigsten Routinen.

Container Ein Typ, dessen Objekte eine Sammlung von Objekten eines bestimmten Typs enthalten.

difference_type Ein von `vector` definierter Integertyp mit Vorzeichen, der die Distanz zwischen zwei beliebigen Iteratoren aufnehmen kann.

empty Von den Typen `string` und `vector` definierte Funktion, die einen Boole'schen Wert dafür zurückgibt, ob der String Zeichen bzw. der Vektor Elemente enthält. Gibt `true` zurück, wenn `size` gleich null ist, sonst `false`.

Endüberschreitender Iterator Der von `end` zurückgegebene Iterator auf ein nicht existierendes Element, das das Ende des Containers um eins überschreitet.

getline Im Header von `string` definierte Funktion, die einen Eingabestream und einen String übernimmt. Sie liest den Stream bis zum nächsten Zeilenumbruch, speichert das Gelesene im String und gibt den Eingabestream zurück. Der Zeilenumbruch wird gelesen und verworfen.

Höherwertig Die Bits eines Bitsets mit den größten Indizes.

Index Vom Indexoperator benutzter Wert zur Bezeichnung des Elements, das aus einem String oder Vektor abgerufen werden soll.

Iterator Ein Typ, der benutzt werden kann, um die Elemente eines Containers zu untersuchen und sich zwischen ihnen zu bewegen.

Iteratorarithmetik Die arithmetischen Operationen, die auf einige, aber nicht alle Iteratortypen angewendet werden können. Ein Integertyp kann zu einem Iterator addiert oder von ihm subtrahiert werden, was dazu führt, dass der Iterator um entsprechend viele Elemente hinter oder vor seiner ursprünglichen Position steht. Zwei Iteratoren lassen sich subtrahieren, was die Distanz zwischen ihnen ergibt. Iteratorarithmetik ist nur für Iteratoren gültig, die auf Elemente im selben Container verweisen, sowie für den endüberschreitenden Iterator.

Klassentemplate Eine Vorlage, aus der sich zahlreiche potenzielle Klassentypen erstellen lassen. Um ein Klassentemplate benutzen zu können, müssen wir angeben, welche Typen oder Werte verwendet werden sollen. Ein Vektor ist zum Beispiel ein Template, das Objekte eines bestimmten Typs enthält. Wenn wir einen Vektor anlegen, müssen wir sagen, welchen Typ *dieser* Vektor enthalten soll. `vector<int>` nimmt Objekte vom Typ `int` auf, `vector<string>` Objekte vom Typ `string` usw.

Niederwertig Die Bits eines Bitsets mit den niedrigsten Indizes.

push_back Von `vector` definierte Funktion, die Elemente an das Ende eines Vektors anhängt.

size Von den Bibliothekstypen `string`, `vector` und `bitset` definierte Funktion, die die Anzahl der Zeichen, Elemente oder Bits zurückgibt. Der Rückgabewert der String- und der Vektorfunktion ist vom Typ `size_type`. Für die Stringfunktion heißt er zum Bei-

spiel `string::size_type`. Die Operation `bitset` gibt einen Wert vom Typ `size_t` zurück.

size_t Rechnerabhängiger, im Header `cstddef` definierter Integertyp ohne Vorzeichen, der für die Größe des größtmöglichen Arrays ausreicht.

size_type Von den Klassen `string` und `vector` definierter Typ, der die Größe eines jeden Strings oder Vektors aufnehmen kann. Bibliotheksklassen definieren diesen Typ ohne Vorzeichen.

Using-Deklarationen Deklarationen, die einen Namen aus einem Namensbereich direkt zugänglich machen.

```
using Namensbereich::Name
```

Diese Deklaration macht *`Name`* ohne das Präfix *`Namensbereich::`* zugänglich.

Wächter Programmiertechnik, die einen Wert als Wächter zur Verarbeitungssteuerung einsetzt. In diesem Kapitel haben wir die Verwendung des von `end` zurückgegebenen Iterators als Wächter gezeigt, der die Verarbeitung von Vektorelementen stoppt, wenn alle Elemente verarbeitet sind.

Wertinitialisierung Initialisierung von Containerelementen, wenn zwar die Containergröße angegeben ist, aber kein expliziter Elementinitialisierer. Die Elemente werden mit der Kopie eines vom Compiler erzeugten Werts initialisiert. Wenn der Container integrierte Typen enthält, lautet dieser Wert null. Für Klassentypen wird der Wert durch Ausführen des Standardkonstruktors der Klasse gebildet. Eine Wertinitialisierung kann für Containerelemente vom Klassentyp nur dann stattfinden, wenn die Klasse einen Standardkonstruktor hat.