Benutzerdefinierte und zusammengesetzte Datentypen

- Aufzählungstypen
- Felder: Der C++-Standardtyp `vector`
- Zeichenketten: Der C++-Standardtyp `string`
- Strukturierte Datentypen
    - Strukturen
    - Unions
    - Bitfelder

Benutzerdefinierte und zusammengesetzte Datentypen

Aufzählungstypen

Häufig gibt es nicht-numerische Wertebereiche. So kann etwa ein Wochentag nur die Werte Sonntag, Montag, ... Samstag annehmen. Oder ein Farbwert soll nur den Werten rot, grün, blau und gelb entsprechen. Eine mögliche Hilfskonstruktion wäre die Abbildung auf den Datentyp `int`:

```
int farbe;      // rot = 0, grün = 1, blau = 2, gelb = 3
int Wochentag;  // Sonntag = 0, Montag = 1, ...
```

Diese Vorgehensweise hätte einige Nachteile:

- Die Bedeutung muss im Programm als Kommentar notiert werden.
- Zugeordnete Zahlenwerte sind nicht eindeutig:
  0 kann rot, aber auch Sonntag bedeuten usw.
- Der Dokumentationswert ist schlecht:
  ```
  if(farbe == 3) ...
  ```
  Es ist nicht zu ersehen, welche Farbe gemeint ist.
  ```
  if(farbe == 7) ...
  ```
  Der Wert 7 ist undefiniert.

Benutzerdefinierte und zusammengesetzte Datentypen

Aufzählungstypen

Die Lösung für solche Fälle sind die *Aufzählungs- oder Enumerationstypen*, die eine Erweiterung der vordefinierten Typen durch den Programmierer darstellen. Die Syntax einer Deklaration ist:

enum [*Typname*] { *Aufzählung* } [*Variablenliste*];

Die neuen Datentypen `Farbtyp` und `Wochentag` können jetzt folgendermaßen deklariert werden:

```
enum Farbtyp   {rot, gruen, blau, gelb};
enum Wochentag {sonntag, montag, dienstag,
                mittwoch, donnerstag,
                freitag, samstag} heute;
```

Wenn der Datentyp erst einmal bekannt ist, können weitere Variablen definiert werden:

```
Farbtyp eimer, klotz;
Wochentag feiertag, werktag,
          morgen = freitag;
```

Benutzerdefinierte und zusammengesetzte Datentypen

Aufzählungstypen

Falls der Aufzählungstyp nur ein einziges Mal in einer Variablendeklaration benötigt wird, kann der Typname auch weggelassen werden. Man erhält dann eine *anonyme Typdefinition*:

```
enum {fahrrad, mofa, lkw, pkw} fahrzeug;
```

Den mit Hilfe von Aufzählungstypen definierten Variablen können ausschließlich Werte aus der zugehörigen Liste zugewiesen werden, Mischungen sind nicht erlaubt.

Aufzählungstypen sind eigene Datentypen, werden aber intern auf die natürlichen Zahlen abgebildet, beginnend bei 0. Eine Voreinstellung mit anderen Zahlen ist möglich.

```
enum Farbtyp {rot=0, gruen=1, blau=2, gelb=4};

enum Palette {weiss=0, grau=1, braun=2,
              umbra=4, lila=8} mischung;

enum Bitmaske {wert1 = 1<<0, wert2 = 1<<1,
               wert3 = 1<<2, wert4 = 1<<3,
               wert5 = 1<<4, // usw.
};
```

Benutzerdefinierte und zusammengesetzte Datentypen

Aufzählungstypen

Eine Umwandlung von Aufzählungs-Konstanten in `int` ist möglich, aber nicht die Umwandlung einer `int`-Zahl in einen Aufzählungstyp.

Als Operation auf `enum`-Typen ist nur die Zuweisung erlaubt, bei allen anderen Operatoren wird vorher in `int` gewandelt.

Welche Anweisungen möglich oder aber unzulässig sind, zeigen die folgenden Zeilen:

```
int i = dienstag;

heute = montag;

heute = i;

montag = heute;

i = rot + blau;

mischung = weiss + lila;

mischung++;

if(mischung > grau)
    mischung = lila;
```

Benutzerdefinierte und zusammengesetzte Datentypen

Felder: Der C++-Standardtyp `vector`

Ein *Vektor* ist eine Tabelle von Elementen desselben Datentyps. Mit Ausnahme von Referenzen kann der Datentyp beliebig sein; er kann auch selbst wieder zusammengesetzt sein. Auf ein Element in der Tabelle wird über eine Positionsangabe zugegriffen.

Die Anweisung

```
vector <int> v(10);
```

stellt einen Vektor v bereit, der 10 Elemente des Typs `int` enthält. Alle Elemente werden mit dem Wert 0 initialisiert. Die Vektorelemente sind stets von 0 bis (Anzahl der Elemente - 1) durchnummeriert.

Ein Vektor „kennt“ die Zahl seiner Elemente, d.h. seine Größe (englisch *size*):

```
cout << v.size() << endl;          // 10
```

Der Zugriff auf ein spezielles Element wird mit dem Indexoperator `[]` realisiert. Zum Beispiel zeigt

```
cout << v[0] << endl;
```

das erste Element des Vektors an. Der zwischen den eckigen Klammern stehende Wert heißt *Index*.

Benutzerdefinierte und zusammengesetzte Datentypen

Felder: Der C++-Standardtyp `vector`

Achten Sie sorgfältig darauf, dass Indexwerte die Vektorgrenzen nicht unter- oder überschreiten!

> Es gibt *keine* Überprüfung auf Bereichsüber- oder Bereichsunterschreitung!

Zugriffe auf nicht existierende Elemente wie z.B.

```
int j = v[-1];
```

erzeugen *keine* Fehlermeldung, weder durch den C++-Compiler noch zur Laufzeit, sofern nicht der zulässige Speicherbereich für das Programm überschritten wird.

Man kann auf den Indexoperator verzichten und stattdessen einen Vektor nach einem Wert *an* (englisch *at*) einer Position fragen. Diese Art des Zugriffs wird geprüft.

```
// alles bestens, dasselbe wie v[0]
cout << v.at(0) << endl;

// Der Index 10000 ist zu groß;
// => Programmabbruch mit Fehlermeldung.
cout << v.at(10000) << endl;
```

Benutzerdefinierte und zusammengesetzte Datentypen

Felder: Der C++-Standardtyp `vector`

Beispiel:

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

int main()
{
   vector<int> meineDaten;    // anfängliche Größe ist 0
   int wert;

   cout << "Bitte Werte eingeben" << endl;
   do
   {
      cout << "Wert (0 = Ende der Eingabe): ";
      cin >> wert;
      if(wert != 0)
         meineDaten.push_back(wert);
   } while(wert != 0);

   cout << "Es wurden die folgenden Werte eingegeben:"
        << endl;
   for(size_t i = 0; i < meineDaten.size(); ++i)
      cout << i << ". Wert : "
           << meineDaten[i] << endl;

    return 0;
}
```

Benutzerdefinierte und zusammengesetzte Datentypen

Felder: Der C++-Standardtyp `vector`

Beispiel:

```
#include <iostream>
#include <algorithm>  // enthält Sortierfunktion sort()
#include <vector>     // Standard-Vektor einschließen
using namespace std;


// Programm mit typischen Vektor-Operationen
int main()
{
   // Tabelle mit 12 float-Werten anlegen
   vector<float> kosten(12);
   // Schleifenvariable zum Durchlaufen des Vektors
   size_t i;

   // Füllen der Tabelle mit beliebigen Daten,
   // dabei Typumwandlung int => float
   for(i = 0; i < kosten.size(); ++i)
      kosten[i] = static_cast<float>(150-i*i)/10.0;

   // Tabelle ausgeben
   for(i = 0; i < kosten.size(); ++i)
      cout << i << ": " << kosten[i] << endl;
```

Benutzerdefinierte und zusammengesetzte Datentypen

Felder: Der C++-Standardtyp `vector`

Beispiel (Fortsetzung):

```
   // Berechnung und Anzeige der Summe
   float sum = 0.0;
   for(i = 0; i < kosten.size(); ++i)
      sum += kosten[i];
   cout << "Summe = " << sum << endl;

   // Mittelwert anzeigen
   cout << "Mittelwert = " << sum/kosten.size() << endl;

   // Maximum anzeigen
   float max = kosten[0];
   for(i = 1; i < kosten.size(); ++i)
      if(max < kosten[i])
         max = kosten[i];
   cout << "Maximum = " << max << endl;

   // Tabelle aufsteigend sortieren
   sort(kosten.begin(), kosten.end());

   // sortierte Tabelle ausgeben
   for(i = 0; i < kosten.size(); ++i)
      cout << i << ": " << kosten[i] << endl;

   return 0;
}
```

Benutzerdefinierte und zusammengesetzte Datentypen

Zeichenketten: Der C++-Standardtyp `string`

Eine Zeichenkette, *String* genannt, ist aus Zeichen des Datentyps `char` zusammengesetzt. Im Grunde kann eine Zeichenkette wie ein eindimensionaler Vektor aufgefasst werden. Dennoch wird nicht ein `vector<char>`, sondern eine andere Standardklasse mit dem Namen `string` als Baustein verwendet.

Beispiel:

```
#include<iostream>
#include<string>     // Standard-String einschließen
using namespace std;

// Programm mit typischen String-Operationen
int main()
{
    size_t i;
    // String-Objekt einString anlegen
    // und mit "hallo" initialisieren.
    string einString("hallo");

    // String ausgeben
    cout << einString << endl;

    // Beim Vektor waere stattdessen für die Ausgabe
    // eine Schleife notwendig, etwa der folgenden Art:
    // String zeichenweise ausgeben,
    // ungepruefter Zugriff wie bei vector:
    for(i = 0; i < einString.size(); ++i)
       cout << einString[i];
    cout << endl;
```

Benutzerdefinierte und zusammengesetzte Datentypen

Zeichenketten: Der C++-Standardtyp `string`

Beispiel (Fortsetzung):

```
// String zeichenweise mit Indexpruefung ausgeben.
// Die Anzahl der Zeichen kann bei Strings auch mit
// length() ermittelt werden.
for(i = 0; i < einString.length(); ++i)
   cout << einString.at(i);
cout << endl;

// Die o.g. Pruefung geschieht wie beim Vektor.
// Ein Versuch, einString.at(i)
// mit i >= einString.size() abzufragen,
// fuehrt zum Programmabbruch mit Fehlermeldung.

// Kopie des Strings einString erzeugen
string eineStringKopie(einString);
cout << eineStringKopie << endl;     // hallo

// Kopie durch Zuweisung
string diesIstNeu("neu!");
eineStringKopie = diesIstNeu;
cout << eineStringKopie << endl;     // neu!

// Zuweisung einer Zeichenketten-Literalkonstanten
eineStringKopie = "Buchstaben";
cout << eineStringKopie << endl;     // Buchstaben

// Zuweisung nur eines Zeichens vom Typ char
einString = 'X';
cout << einString<< endl;            // X
```

Benutzerdefinierte und zusammengesetzte Datentypen

Zeichenketten: Der C++-Standardtyp `string`

Beispiel (Fortsetzung):

```
    // Strings mit dem +=-Operator verketten
    einString += eineStringKopie;
    cout << einString << endl;          // XBuchstaben

    // Strings mit dem +-Operator verketten
    einString = eineStringKopie + " ABC";
    cout << einString< < endl;          // Buchstaben ABC

    einString = "123" + eineStringKopie;
    cout << einString << endl;          // 123Buchstaben

    // Eine Erklaerung gibt es erst im
    // Kapitel Ueberladen von Operatoren,
    // aber Folgendes geht nicht:
    //    einString = "123" + "ABC";    // Fehler!

    string a("Albert"), z("Alberta"), b = a;

    if(a == b)
        cout << a << " == " << b << endl;
    else
        cout << a << " != " << b << endl;

    if(a < z)
        cout << a << " < " << z << endl;
    if(z > a)
        cout << z << " > " << a << endl;
    if(z != a)
        cout << z << " != " << a << endl;

    return 0;
}
```

Benutzerdefinierte und zusammengesetzte Datentypen

Strukturierte Datentypen

Ein Vektor hat nur Elemente desselben Datentyps. Häufig möchte man logisch zusammengehörige Daten zusammenfassen, die *nicht* vom selben Datentyp sind. Für diese Zwecke stellt C++ die *Struktur* bereit, die einen Datensatz zusammenfasst.

Strukturen

Strukturen werden mit einer Syntax definiert, die bis auf den inneren Teil der Syntax von Aufzählungstypen ähnelt:

```
struct [Typname] { Aufbau } [Variablenliste];
```

Die strukturinternen Daten heißen *Elemente* (auch *Felder* oder *Komponenten*).

Benutzerdefinierte und zusammengesetzte Datentypen

Strukturen

Beispiel: Daten eines Punktes auf dem Bildschirm

```
enum Farbtyp {rot, gelb, gruen};

struct Punkt
{
   int x;
   int y;
   bool sichtbar;
   Farbtyp farbe;
} p;                        // p ist ein Punkt

Punkt q;                    // q auch

// Zugriff
p.x = 123;                  // Koordinaten von p
p.y = 45;
p.sichtbar = true;
p.farbe = gruen;
```

Benutzerdefinierte und zusammengesetzte Datentypen

Strukturierte Datentypen

Unions

Unions sind Strukturen, in denen verschiedene Elemente *denselben* Speicherplatz bezeichnen.

`union` [*Typname*] `{` *Aufbau* `}` [*Variablenliste*]`;`

Unions sollten nur in Sonderfällen eingesetzt werden.

Der gesamte Speicherplatz einer Variablen vom Typ `union` ist identisch mit dem Speicherplatzbedarf des jeweils größten internen Elements.

Im folgenden Beispiel werden eine `int`-Zahl und ein `char`-Array überlagert. Eine Variable vom Typ `char` belegt genau ein Byte. Das Array hat hier genauso viel Elemente wie ein `int`-Zahl Bytes hat.

```
union HiLo
{
  int zahl;
  unsigned char c[sizeof(int)];
};
```

Benutzerdefinierte und zusammengesetzte Datentypen

Unions

Beispiel:

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

union HiLo
{
  int zahl;
  unsigned char c[sizeof(int)];
};

int main()
{
  HiLo was;
  do
  {
    cout << "Bitte Zahl eingeben (0=Ende): ";
    cin >> was.zahl;

    cout << "Byte-weise Darstellung von "
         << was.zahl << endl;

    for(int i = sizeof(int) - 1; i >= 0; i--)
      cout << "Byte Nr. " << i << " = "
           << int(was.c[i]) << endl;
  } while(was.zahl != 0);

  return 0;
}
```

Benutzerdefinierte und zusammengesetzte Datentypen

Unions

Beispiel für die Ausgabe des Programms:

```
Bitte Zahl eingeben (0=Ende): 1234567890
Byte-weise Darstellung von 1234567890
Byte Nr. 3 = 73
Byte Nr. 2 = 150
Byte Nr. 1 = 2
Byte Nr. 0 = 210
Bitte Zahl eingeben (0=Ende): 65536
Byte-weise Darstellung von 65536
Byte Nr. 3 = 0
Byte Nr. 2 = 1
Byte Nr. 1 = 0
Byte Nr. 0 = 0
Bitte Zahl eingeben (0=Ende): 65535
Byte-weise Darstellung von 65535
Byte Nr. 3 = 0
Byte Nr. 2 = 0
Byte Nr. 1 = 255
Byte Nr. 0 = 255
Bitte Zahl eingeben (0=Ende): 0
Byte-weise Darstellung von 0
Byte Nr. 3 = 0
Byte Nr. 2 = 0
Byte Nr. 1 = 0
Byte Nr. 0 = 0
```

Benutzerdefinierte und zusammengesetzte Datentypen

Strukturierte Datentypen

Bitfelder

In der hardwarenahen Programmierung oder der Systemprogrammierung ist es oft wünschenswert, Bitfelder verschiedener Längen anzulegen, in denen die einzelnen Bitpositionen unterschiedliche Bedeutungen haben.

Die Deklaration eines Bitfelds hat die Syntax

Datentyp Name `:` *Konstante* `;`

wobei *Datentyp* einer der Datentypen `char`, `int`, `long` (mit den vorzeichenlosen `unsigned`-Varianten) oder ein Aufzählungstyp sein kann.

Beispiel:

```
int bitfeld : 13;
```

Es dürfen keine Annahmen darüber gemacht werden, wo die 13 Bits innerhalb eines 16- oder 32-Bit-Worts angeordnet sind – dies ist *implementierungsabhängig*.

Benutzerdefinierte und zusammengesetzte Datentypen

Bitfelder

Beispiel: Zugriff auf Bitfelder

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

struct Bitfeld
{
  unsigned int a : 4;
  unsigned int b : 3;
};

int main()
{
  Bitfeld x;
  x.a = 006;
  x.b = x.a | 03;
  cout << x.b << endl;
  return 0;
}
```