IV FELDER, ZEIGER UND ABGELEITETE DATENTYPEN

S. BARTELS, 4.6.2018

1. Statische Felder

In vielen EDV-Anwendungen liegen numerierte Daten vor, wie beispielsweise die Temperaturen an den 365 Tagen eines Jahres. Diese lassen sich in Programmen mit Listen beziehungsweise *Feldern* darstellen. Ein (eindimensionales) Feld der Länge N mit Einträgen eines bestimmten Datentyps wird folgendermaßen deklariert:

```
type feld[N];
```

Dabei ist zu beachten, dass die Länge N zum Zeitpunkt der Übersetzung des Programms bereits definiert (bei einigen Compilern sogar als konstant) sein muss. Initiale Einträge können wie bei einfachen Variablen direkt mit der Deklaration festgelegt werden:

```
type feld[N] = {val0,val1,...,valN-1};
```

Alternativ können Einträge einzeln durch Angabe der Position definiert werden. Dabei ist zu beachten, dass die Indizierung in C++ mit dem Index 0 beginnt und im Fall von N Einträgen bis $N-1$ läuft:

```
feld[0] = val0; feld[1] = val1; ... feld[N-1] = valN-1;
```

Allgemeiner können mehrdimensionale Felder benutzt werden, die im Fall eines zweidimensionalen Feldes mittels der Deklaration und zeilenweisen Initialisierung

```
type feld[M][N] = {{val00,...},{val10,...},...};
```

definiert werden. Bei der direkten Initialisierung muss die Größe des Feldes nicht explizit angegeben werden. Auf einzelne Einträge eines zweidimensionalen Feldes wird mittels `feld[j][k]` zugeriffen. Abbildung 1 zeigt die entsprechende Realisierung einer Matrix-Vektor-Multiplikation.
Die Einträge eines Feldes werden hintereinander, in den dem Datentyp entsprechenden Abständen im Speicher abgelegt. Felder werden häufig auch als *Arrays* bezeichnet. Sollen die Inhalte eines Feldes einem anderen zugewiesen werden, so muss dies eintragsweise geschehen, da eine Zuweisung $a = b$ für Felder nicht zulässig ist.

2. Übergabe von Feldern

Felder können als Argumente ohne Spezifikation ihrer Größe in Unterfunktionen verwendet werden, allerdings ist die Behandlung solcher Argumente fundamental unterschiedlich von der Behandlung von Variablen einfachen

```
1  // kompilieren: g++ mat_vek_mult.cc -o mat_vek_mult.out
2  #include <iostream>
3  using namespace std;
4  int main(){
5    const int M = 2, N = 3;
6    double A[M][N] = {{1.0,2.0,3.0},{4.0,5.0,6.0}};
7    double x[N] = {1.0,2.0,3.0};
8    double b[M];
9    for (int j=0; j<M; j++){
10     b[j] = 0.0;
11     for (int k=0; k<N; k++){
12       b[j] += A[j][k]*x[k];
13       }
14     }
15   cout << "Das Matrix-Vektor-Produkt A*x ist: " << endl;
16   for (int j=0; j<M; j++){
17     cout << b[j] << ", ";
18   }
19   cout << endl;
20 }
```

ABBILDUNG 1. Realisierung einer Matrix-Vektor-Multiplikation mit Feldern fester Größe. Man beachte die Initialisierung der Einträge des Feldes b mit Nullwerten.

Typs als Funktionsargument. Bei einfachen Typen kopiert die Unterfunktion lediglich den Wert der Variablen in eine lokale Variable, die denselben Namen haben darf, der Wert der Variable im aufrufenden Programmteil bleibt davon aber unberührt. Dies wird als *Call-by-Value* bezeichnet, ein Beispiel ist in Abbildung 2 gezeigt.
Wird hingegen die Variable als Feld definiert, wie in Abbildung 3 gezeigt, so werden die Werte des Feldes durch den Funktionsaufruf verändert. Hier wird der Funktion die Speicheradresse des Beginns des Feldes übergeben und die Funktion überschreibt die dort abgelegten Werte. Diese Art des Funktionsaufrufs wird als *Call-by-Reference* bezeichnet.

3. Verwendung von Speicheradressen (Zeigern)

Eine sehr Hardware-nahe Programmierung ist mit der direkten Verwendung von Speicheradressen, die auch als *Zeiger* beziehungsweise *Pointer* bezeichnet werden, möglich. Dieses Prinzip ermöglicht zum Beispiel die Übergabe einfacher Variablen an Funktionen mittels *Call-by-Reference*. Ein Zeiger auf eine Variable eines bestimmten Typs wird deklariert durch

```
type *ptr;
```

In diesem Fall kann die Variable `ptr` eine Speicheradresse enthalten, während der Ausdruck `*ptr` wie eine normale Variable verwendet werden kann und

```
1  // kompilieren: g++ call_by_val.cc -o call_by_val.out
2  #include <iostream>
3  using namespace std;
4  void addiere_eins(int a){
5    a += 1;
6    cout << a << endl;
7  }
8  int main(){
9    int a = 1;
10   cout << a << endl;
11   addiere_eins(a);
12   cout << a << endl;
13 }
```

ABBILDUNG 2. Aufruf einer Unterfunktion mit einer einfachen Variable; das Programm führt zur Ausgabe der Werte 1, 2, 1.

```
1  // kompilieren: g++ call_by_ref.cc -o call_by_ref.out
2  #include <iostream>
3  using namespace std;
4  void addiere_eins(int a[]){
5    a[0] += 1;
6    cout << a[0] << endl;
7  }
8  int main(){
9    int N = 1;
10   int a[N] = {1};
11   cout << a[0] << endl;
12   addiere_eins(a);
13   cout << a[0] << endl;
14 }
```

ABBILDUNG 3. Aufruf einer Unterfunktion mit einem Feld; das Programm führt zur Ausgabe der Werte 1, 2, 2.

den Wert hat, der an der durch `ptr` definierten Speicheradresse abgelegt ist. Der Typ eines Zeigers muss festgelegt werden, damit klar ist, wie viele Bits im Speicher zum Wert der Variable gehören. Bevor der Variable `*ptr` ein Wert zugeordnet werden kann, muss dem Zeiger `ptr` eine Adresse zugeordnet worden sein. Ist `var` eine Variable vom Typ des Pointers, so erhält man die Adresse des zugehörigen Speicherbereichs durch `&var`. Diese kann dann einem Zeiger zugewiesen werden:

```
ptr = &var;
```

In diesem Zusammenhang bezeichnet man den Operator `*` als *Dereferenzierungs-* und `&` als *Referenzierungsoperator*. Das in Abbildung 4 gezeigte Programm illustriert die Verwendung und führt zur Ausgabe der Adresse des Zeigers und des dort abgelegten Werts. Der Einsatz von Zeigern ist mit gewissen Gefahren bei der Speicherverwaltung verbunden und sollte möglichst vermieden werden.

```
1  // kompilieren: g++ zeiger_test.cc -o zeiger_test.out
2  #include <iostream>
3  using namespace std;
4  int main(){
5    int *ptr;
6    int j = 1, k = 2;
7    ptr = &k;
8    cout << "Adresse: " << ptr << ", Wert: " << *ptr << endl;
9    *ptr = 5;
10   cout << "Adresse: " << ptr << ", Wert: " << *ptr << endl;
11   ptr = &j;
12   cout << "Adresse: " << ptr << ", Wert: " << *ptr << endl;
13   //
14   cout << "Beachte veraenderten Wert von k, k = " <<  k << endl;
15 }
```

Abbildung 4. Verwendung eines Zeigers und Ausgabe seiner Adresse sowie des dort abgelegten Werts.

4. Vektoren variabler Länge

Die Länge einer Liste oder die Größe eines Feldes ergibt sich häufig erst im Laufe der Berechnungen eines Programms. Die C++-Bibliothek `vector` stellt mächtige Werkzeuge für das Arbeiten mit Listen beziehungsweise Vektoren zur Verfügung. Ein Vektor der Länge N mit Einträgen eines bestimmten Typs wird mit folgendem Befehl deklariert:

```
std::vector<type> vec(N);
```

Die Einträge können bei Deklaration mit einem konstanten Wert initialisiert werden, wenn `vec(N,val)` statt `vec(N)` verwendet wird. Ähnlich können nach der Deklaration die Einträge eines Vektors mit dem Befehl `fill` auf einen konstanten Wert gesetzt werden. Die Länge wird mit der Methode `resize` geändert, wobei der Aufruf mittels `vec.resize(M)` geschieht. Eine Übersicht verschiedener Befehle und weiterer Methoden findet sich in Tabelle 1, ein einfaches Beispielprogramm ist in Abbildung 5 gezeigt. Man beachte dabei, dass die Übergabe dieser Vektoren mittels *call-by-reference* erfolgt. Für das Arbeiten mit zweidimensionalen Feldern variabler Länge und fester Anzahl von Spalten steht in C++ die Bibliothek `array` zur Verfügung.

`std::vector`	Deklaration eines Vektors
`std::fill`	Initialisierung eines Vektors
`std::copy`	Kopieren von Einträgen eines Vektors
vec`.size`	Länge eines Vektors
vec`.resize`	Veränderung der Länge eines Vektors
vec`.begin`	Adressierung des Anfangs eines Vektors
vec`.end`	Referenzierung des Endes eines Vektors
vec`.push_back`	Anhängen eines Elements an einen Vektor
vec`.pop_back`	Entfernen des letzten Elements eines Vektors

TABELLE 1. C++-Routinen zur Bearbeitung eines Vektors variabler.

```
1  // kompilieren: g++ dynam_vektor.cc -o dynam_vektor.out
2  #include <iostream>
3  #include <vector>
4  using namespace std;
5  int main(){
6    int N = 2;
7    vector<double> u(N);
8    fill(u.begin(),u.end(),1.);
9    N = 4;
10   u.resize(N);
11   u[2] = 5.0; u[3] = 7.0;
12   for (int j=0; j<(int)u.size(); j++)
13     cout << "u[" << j << "] = " << u[j] << endl;
14 }
```

ABBILDUNG 5. Anlegen eines Vektors und Änderung seiner Länge.

5. ZEICHENKETTEN

Variablen, die Zeichenketten wie beispielsweise Personennamen als Werte annehmen sollen, können als Felder über Literalen das heißt über dem Typ `char` deklariert werden. Einer Variable wird dann mittels Anführungsstrichen ein Wert zugewiesen:

```
char wort[6] = "hallo";
```

Man beachte, dass für eine Zeichenkette mit n Zeichen ein Feld der Länge $n + 1$ benötigt wird, da implizit das Endsymbol `\0` mit gespeichert wird. Der Zugang über Felder stößt schnell an Grenzen, da eine maximale Länge gewählt werden muss, was mit Hilfe einer globalen Konstante geschehen kann. Ein Beispiel ist in Abbildung 7 gezeigt. Dabei wird die maximale Länge über ein Präprozessor-Makro festgelegt.

```
// kompilieren: g++ vektor_funktion.cc -o vektor_funktion.out
#include <iostream>
#include <vector>
using namespace std;
vector<double> lin_komb(vector<double> x, vector<double> y,
          double a, double b){
  int N = (int)x.size();
  vector<double> z(N);
  for (int j=0; j<N; j++)
    z[j] = a*x[j]+b*y[j];
  return z;
}
int main(){
  int n = 2;
  vector<double> u(n,1.0);
  vector<double> v(n,0.0);
  fill(u.begin(),u.end(),5.0);
  copy(u.begin(),u.end(),v.begin());
  u = lin_komb(u,v,1.0,2.0);
  for (int j=0; j<(int)u.size(); j++)
    cout << "u[" << j << "] = " << u[j] << endl;
}
```

ABBILDUNG 6. Berechnung der Linearkombination zweier Vektoren in einer Unterfunktion.

```
// kompilieren: g++ char_feld.cc -o char_feld.out
#include <iostream>
#define max_laenge 20
using namespace std;
int main(){
  char zeichen_kette[max_laenge] = "lange kette ...";
  cout << zeichen_kette << endl;
}
```

ABBILDUNG 7. Deklaration einer Zeichenkette als Feld über Literalen mit global definierter maximaler Länge.

Kompatibel mit Feldern über Literalen und komfortabel in seiner Verwendung ist der von der Bibliothek beziehungsweise Klasse `string` bereitgestellte Typ `string`. Das Speichermanagement erfolgt hier automatisch und das Arbeiten mit Variablen vom gleichnamigen Typ `string` ist im Hinblick auf Vergleiche ähnlich dem von Zahlentypen. Man kann Zeichenketten dieses Typs alphabetisch vergleichen und mit dem Operator `+` aneinanderhängen. Zudem sind Funktionen wie `length` implementiert, die mit dem Selektionsoperator ausgeführt werden, wie in Abbildung 8 beispielhaft gezeigt ist.

```
1  // kompilieren: g++ string_klasse.cc -o string_klasse.out
2  #include <iostream>
3  #include <string>
4  using namespace std;
5  int main(){
6    string wort_1 = "fussball";
7    string wort_2 = "spiel";
8    string wort_3 = "null";
9    if (wort_1<wort_2)
10     wort_3 = wort_1+wort_2;
11   cout << wort_3 << endl;
12   cout << wort_3.length() << endl;
13 }
```

ABBILDUNG 8. Verwendung einer Bibliothek zur Definition und Bearbeitung von Zeichenketten.

6. ABGELEITETE DATENTYPEN

Sogenannte *Strukturen* oder *Verbunde* erlauben die Zusammenfassung von Variablen verschiedenen Typs zu einer und werden als abgeleiteter Datentyp bezeichnet. Anwendungsbeispiele sind Positionen im Raum, die drei Koordinaten besitzen, oder Adressbücher bei denen für jeden Eintrag verschiedene Informationen erfasst werden. Die Definition einer Struktur und die Deklaration einer Variable dieses Verbunds erfolgt über die Befehle

```
struct name{type var1; ...; type varn;};
name struct_var;
```

Der Zugriff auf die verschiedenen Komponenten einer Verbund-Variable erfolgt durch den Selektionsoperator `.` und die Angabe des Namens der Untervariable, das heißt eine Zuweisung erfolgt über:

```
struct_var.var1 = val1;
```

Eine Variable eines Verbunds kann wie eine normale Variable in Funktionen verwendet werden, siehe Abbildung 9.

7. DATEIOPERATIONEN

Das Lesen und Schreiben von Daten in Dateien ähnelt der Ein- und Ausgabe von Daten auf dem Monitor. Vereinfachend formuliert wird die Ausgabe umgeleitet in die dafür geöffnete Datei. Zum Verwalten der Öffnungs- und Schließprozesse wird der Typ `fstream` aus der gleichnamigen Bibliothek verwendet. Das in Abbildung 10 gezeigte Programm schreibt eine Textzeile in eine Datei beziehungsweise fügt zu vorhandenen Inhalten hinzu. Die Variable `f` ermöglicht Operationen wie das Öffnen und Schließen über einen Selektionsoperator. Bei der Verwendung des Befehls

```
f.open("test.dat",ios::out);
```

```
1  // kompilieren: g++ struct.cc -o struct.out
2  #include <iostream>
3  struct punkt{
4    double x, y, z;
5    char farbe[10];
6  };
7  punkt mod_punkt(punkt q){
8    q.x = q.x+1.0;
9    return q;
10 }
11 using namespace std;
12 int main(){
13   punkt p = {1.0,2.0,3.0,"gelb"};
14   p = mod_punkt(p);
15   cout << "p = ( " << p.x << " , " << p.y
16        << " , " << p.z << " ) " << endl;
17   cout << p.farbe << endl;
18 }
```

ABBILDUNG 9. Definition eines Verbunds und Verwendung in einer Funktion.

werden bereits vorhandene Inhalte gelöscht und es wird eine Textdatei erzeugt. Jedem Öffnungsbefehl sollte ein Schließbefehl folgen. Sollen Inhalte an bereits bestehende Inhalte angehängt werden, ist die zusätzliche, mit einem senkrechten Strich abgetrennte Option `ios::app` zu verwenden.

```
1  // kompilieren: g++ datei_schreiben.cc -o datei_schreiben.out
2  #include <fstream>
3  using namespace std;
4  int main(){
5    fstream f;
6    f.open("test.dat",ios::out|ios::app);
7    f << "Hinzufuegen von Text in eine Datei" << endl;
8    f.close();
9  }
```

ABBILDUNG 10. Öffnen einer Datei, Hinzufügen einer Textzeile und abschließendes Schließen der Datei.

Bei der Arbeit mit Dateien kann es leicht zu Fehlern kommen, wenn etwa die Datei schreibgeschützt ist. Um solche Fehler abzufangen, kann die Methode `good` verwendet werden:

```
if (f.good()){
   // Schreiben beginnt
}
```

Für wissenschaftliche Zwecke ist meist das Lesen von Daten aus Dateien erforderlich. Dazu muss ein geeignetes Format der Daten in der Datei vorliegen, um sie korrekt aus der Datei auslesen zu können. Mit dem Befehl

```
input >> var
```

wird die nächste Information aus der Datei eingelesen, deren Format durch den Typ der Variable `var` spezifiziert ist. Dies ähnelt sehr der Verwendung des Befehls `cin`.

```
1  // kompilieren: g++ datei_lesen.cc -o datei_lesen.out
2  #include <iostream>
3  #include <fstream>
4  using namespace std;
5  int main(){
6    ifstream input("koordinaten.txt");
7    if(!input){
8      cerr << "Datei nicht gefunden" << endl;
9      return 0;
10   }
11   int j, N = 4;
12   double V[N][3];
13   double v_x, v_y, v_z;
14   for(j=0; input >> v_x >> v_y >> v_z; j++){
15     V[j][0] = v_x; V[j][1] = v_y; V[j][2] = v_z;
16   }
17   for(j=0; j<N; j++){
18     cout << V[j][0] << "  " << V[j][1];
19     cout << "  " << V[j][2] << endl;
20   }
21 }
```

Abbildung 11. Lesen der Inhalte der in Abbildung 12 gezeigten Datei.

```
1  1.0     2.2     3.3
2  4.0     5.0     6.0
3  7.0     8.0     9.0
4  10.0   11.0    12.0
```

Abbildung 12. Daten einer Textdatei.