Einführung in die Programmierung mit C++

Nutzerdefinierte Statische Felder

Uwe Naumann

Informatik 12:
Software and Tools for Computational Engineering (STCE)

RWTH Aachen

Inhalt

1D-Felder

2D-Felder

*n*D-Felder

Nutzerdefinierte Statische Felder und ...

- ... Zeiger
- ... Referenzen
- ... **auto**

Inhalt

Nutzerdefinierte Statische 1D-Felder

Deklaration und Initialisierung


```
1  #include<iostream>
2
3  int main() {
4    int v[3]={1,2};
5    for (size_t i=0;i<3;i++)
6      std::cout << v[i] << std::endl;
7    return 0;
8  }
```

resultiert, z.B., in der Ausgabe

1
2
0

→ Tafelbild: Hauptspeicherlayout
→ Live: Inspektion mit gdb

- Kollektion konstanter Anzahl von Variablen des gleichen Typs
- Elemente werden an einer Basisadresse beginnend konsekutiv und ohne Lücken im Speicher abgelegt
- Zugriff auf i+1-tes Element eines Feldes v durch v[i]
- Initialisierung mittels Werteliste in geschweiften Klammern
  - Übersetzungsfehler bei zu vielen Werten
  - nicht initialisierter Speicher bei zu wenigen Werten

Nutzerdefinierte Statische 1D-Felder

Alternativen für Deklaration

- Aliasname

```
int main() { const size_t n=3; int v[n]; ... }
```

- Präprozessorvariable

```
#define N 3
int main() { int v[N]; ... }
```

- Compileroption

```
int main() { int v[N]; ... }
```

und `g++ -DN=3 ...`

- Übrigens: g++ unterstützt auch

```
int main() { size_t n; std::cin >> n; int v[n]; ... }
```

generiert jedoch eine Warnung, dass dies nicht dem C++ Standard entspricht.

Nutzerdefinierte Statische 1D-Felder

Beispiel: Skalarprodukt

Das Skalarprodukt $z = \mathrm{x}^T \cdot \mathrm{y} \in \boldsymbol{R}$ zweier Vektoren $\mathrm{x} = (x_i), \mathrm{y} = (y_i) \in \boldsymbol{R}^n$ is definiert als

$$z = \sum_{i=0}^{n-1} x_i \cdot y_i \ .$$

```
1  #include<iostream>
2
3  int main() {
4    const size_t n=4;
5    float x[n],y[n],xTy=0;
6    for (size_t i=0;i<n;i++) { x[i]=i%2; y[i]=1/(i+1); } // init
7    for (size_t i=0;i<n;i++) xTy+=x[i]*y[i]; // scalar product
8    std::cout << xTy << std::endl;
9    return 0;
10 }
```

AUFWACHEN: Warum wird hier 0 ausgegeben?

Nutzerdefinierte Statische 1D-Felder

Unzulässiger Elementzugriff

Es findet keine Überprüfung der Feldgrenzen statt, d.h. man kann auf Elemente von f zugreifen, die nicht zu f gehören, z.B.

```
1  #include<iostream>
2
3  int main() {
4    int f[2]={1,2}, n;
5    std::cin >> n;
6    for (int i=0;i<n;i++)
7      std::cout << f[i] << " ";
8    return 0;
9  }
```

z.B.

```
2000
1 2 2125285888 -1071335646 4196752 0
...
6649Segmentation fault (core dumped)
```

oder auch f[-1], ...

Bei Zugriff auf einen unerlaubten (Speicher-)Bereich wird durch das Betriebssystem eine Schutzverletzung (*segmentation fault*) ausgelöst.

Der prinzipiell erlaubte (damit jedoch noch lange nicht algorithmisch sinnvolle) Speicherbereich ist jedoch typischerweise deutlich größer als der geplante Gültigkeitsbereich von f.

Nutzerdefinierte Statische 1D-Felder

Alternativer Elementzugriff

- Statische Felder werden durch den Compiler als konstante Zeiger (nicht per $\rightarrow$ Zeigerarithmetik modifizierbar), welche die Basisadresse des ersten Elements referenzieren, implementiert.
- Auf den Inhalt des ersten Elements von, z.B.

```
float x[3]={1.1,2.2,3.3};
```

kann alternativ mittels `x[0]` oder `*x` zugegriffen werden.

- Durch `x[i]` wird die Adresse `&x+i*sizeof(float)` dereferenziert. Alternativ kann man auch `*(x+i)` schreiben, z.B.

```
1  #include<iostream>
2
3  int main() {
4    const size_t n=3; int iv[n]={41,42,43};
5    for (size_t i=0;i<n;i++)
6      std::cout << *(iv+i) << std::endl;
7    return 0;
8  }
```

Nutzerdefinierte Statische 1D-Felder

Zeigerarithmetik

Nach Definition eines nichtkonstanten Zeigers auf das erste Element eines Feldes können dessen Einträge auch mittels Zeigerarithmetik besucht werden, z.B. illustriert das folgende Programm alle bisher diskutierten Elementzugriffsoptionen.

```
1  include<iostream>
2  #include<cmath>
3
4  int main() {
5    const size_t n=3; float iv[n];
6    size_t i;
7    for (i=0;i<n;i++) iv[i]=cos(i);
8    for (i=0;i<n;) --*(iv+i++); i=0;
9    for (float* ivp=iv;i<n;i++)
10     std::cout << *ivp++ << std::endl;
11   return 0;
12 }
```

Beachte: Gültigkeitsbereiche und Prä-/Postfixnotation

AUFWACHEN!

Was wird ausgegeben?

```
1  #include<iostream>
2
3  int main() {
4          int a[3]={1,2,3};
5          std::cout << a[0]+*(a+1)-a[a[*a]] << std::endl;
6          return 0;
7  }
```

Wach?

Was wird ausgegeben?

```
1  #include<iostream>
2
3  int main() {
4          int a[3]={1,2,3};
5          std::cout << a[0]+*(a+1)-a[a[*a]] << std::endl;
6          return 0;
7  }
```

0

Inhalt

Statische 2D-Felder sind statische 1D-Felder von statischen 1D-Feldern.

Betrachtet man sie als Matrizen $(a_{i,j}) \in \boldsymbol{R}^{m \times n}$ dann folgt:

- Elemente werden an einer Basisadresse beginnend konsekutiv und ohne Lücken zeilenweise im Speicher abgelegt (1D-Feld der Länge m von 1D-Feldern der Länge n)
- Zugriff auf Elemente mittels a[i][j]=*(*(a+i)+j) bzw. mittels Zeigerarithmetik auf nichtkostanten Zeiger auf erstes Element
- Initialisierung mittels Werteliste in geschweiften Klammern der Länge $m \cdot n$
- unzulässiger Elementzugriff möglich

Nutzerdefinierte Statische 2D-Felder

Speicherlayout (siehe auch → gdb)


```
1  #include<iostream>
2
3  int main() {
4    using namespace std;
5    int m[2][3]={1,2,3,4,5,6};
6    for (size_t i=0;i<2;i++)
7      for (size_t j=0;j<3;j++)
8    cout << *(m+i)+j << ": " << *(*(m+i)+j) << endl;
9        // äquivalent: cout << &m[i][j] << ": " << m[i][j] << endl;
10   return 0;
11 }
```

generiert (z.B.) die Ausgabe

```
0x7ffff36d0e40: 1
0x7ffff36d0e44: 2
0x7ffff36d0e48: 3
0x7ffff36d0e4c: 4
0x7ffff36d0e50: 5
0x7ffff36d0e54: 6
```

Nutzerdefinierte Statische 2D-Felder

Beispiel: Matrix-Vektor Produkt

Das Produkt $y = (y_i) = A \cdot x \in \boldsymbol{R}^m$ einer Matrix $A = (a_{i,j}) \in \boldsymbol{R}^{m \times n}$ mit einem Vektor $x = (x_j) \in \boldsymbol{R}^n$ is definiert als

$$y_i = \sum_{j=0}^{n-1} a_{i,j} \cdot x_j \ .$$

```
1  #include<iostream>
2  #include<cmath>
3
4  int main() {
5    const size_t m=2,n=3;
6    int A[m][n]={1,2,3,4,5,6},x[n]={-3,-2,-1},Ax[m]={0,0};
7    for (size_t i=0;i<m;i++)
8      for (size_t j=0;j<n;j++)
9        Ax[i]+=A[i][j]*x[j];
10   for (size_t i=0;i<m;i++)
11     std::cout << Ax[i] << std::endl;
12   return 0;
13 }
```

AUFWACHEN!

Was wird ausgegeben?

```
1  #include<iostream>
2
3  int main() {
4          const int m=2,n=3;
5          int a[m][n]={1,2,3,4,5,6};
6          std::cout << *(a[0]) << " ";
7          std::cout << (*a)[0] << " ";
8          std::cout << *(a[1]) << " ";
9          std::cout << (*a)[1] << std::endl;
10         return 0;
11 }
```

Wach?

Was wird ausgegeben?

```
1  #include<iostream>
2
3  int main() {
4          const int m=2,n=3;
5          int a[m][n]={1,2,3,4,5,6};
6          std::cout << *(a[0]) << " ";
7          std::cout << (*a)[0] << " ";
8          std::cout << *(a[1]) << " ";
9          std::cout << (*a)[1] << std::endl;
10         return 0;
11 }
```

1 1 4 2

WACH BLEIBEN!

Was wird ausgegeben?

```
1  #include<iostream>
2
3  int main() {
4          const int m=2,n=3;
5          int a[m][n]={1,2,3,4,5,6};
6          for (int i=0;i<m;i++)
7            for (int j=0;j<n;j++)
8              std::cout << a[0][i*m+j] << " ";
9          std::cout << std::endl;
10         return 0;
11 }
```

Noch Wach?

Was wird ausgegeben?

```
1  #include<iostream>
2
3  int main() {
4          const int m=2,n=3;
5          int a[m][n]={1,2,3,4,5,6};
6          for (int i=0;i<m;i++)
7            for (int j=0;j<n;j++)
8              std::cout << a[0][i*m+j] << " ";
9          std::cout << std::endl;
10         return 0;
11 }
```

1 2 3 3 4 5

Inhalt

Statische nD-Felder sind rekursiv als statische 1D-Felder von statischen $(n-1)$D-Feldern definiert.

Betrachtet man sie als Tensoren, z.B. $(a_{i,j,k}) \in \boldsymbol{R}^{m \times n \times p}$ dann folgt:

- Elemente werden an einer Basisadresse beginnend konsekutiv und ohne Lücken im Speicher abgelegt (z.B. 1D-Feld der Länge m von 1D-Feldern der Länge n von 1D-Feldern der Länge p)
- Zugriff auf Elemente mittels `a[i][j][k]=*(*(*(a+i)+j)+k)` bzw. mittels Zeigerarithemtik auf nichtkostanten Zeiger auf erstes Element
- Initialisierung mittels Werteliste in geschweiften Klammern, z.B. der Länge $m \cdot n \cdot p$
- unzulässiger Elementzugriff möglich

Nutzerdefinierte Statische *n*D-Felder

Speicherlayout (siehe auch → gdb)


```
1  #include<iostream>
2  #include<limits>
3
4  int main() {
5    const size_t m=2,n=3,p=2;
6    int t[m][n][p]={1,2,3,4,5,6,7,8,9,10,11,12};
7    for (size_t i=0;i<m;i++)
8      for (size_t j=0;j<n;j++)
9        for (size_t k=0;k<p;k++)
10         std::cout << *(*(t+i)+j)+k << ": " << *(*(*(t+i)+j)+k) << std::endl;
11         // äquivalent: std::cout << &t[i][j][k] << ": " << t[i][j][k] << std::endl;
12   return 0;
13 }
```

generiert (z.B.) die (gekürzte) Ausgabe

```
0x7ffc9dbabd00: 1
0x7ffc9dbabd04: 2
...
0x7ffc9dbabd28: 11
0x7ffc9dbabd2c: 12
```

Das folgende Program findet das Minimum über alle Einträge eines 3-Tensors mithilfe von Zeigerarithmetik.

```
1  #include<iostream>
2  #include<limits>
3
4  int main() {
5    const size_t m=2,n=2,p=3;
6    float A[m][n][p]={1,-2,3,-4,5,-6,-1,2,-3,4,-5,6}, *z=&A[0][0][0];
7    float min_entry=std::numeric_limits<float>::max();
8    for (size_t i=0;i<m*n*p;i++)
9      min_entry=std::min(*z++,min_entry);
10   std::cout << min_entry << std::endl;
11   return 0;
12 }
```

Ausgabe: -6

AUFWACHEN!


```
1  double t[2][3][2];
2
3  *(**(t+1)+1) = ???
4
5  **(*(t+1)+1) = ???
6
7  ***(t+1)+1 = ???
```

Wach?

```
1  double t[2][3][2];
2
3  *(**(t+1)+1) = t[1][0][1]
4
5  **(*(t+1)+1) = t[1][1][0]
6
7  ***(t+1)+1 = t[1][0][0]+1
```

→ Tafelbild: Hauptspeicherlayout
→ Optional: Inspektion mit gdb

WACH BLEIBEN!

Was wird ausgegeben?

```
1  int m[2][3]={1,2,3,4,5,6};
2  int *p=m[0]+2;
3  cout << p[0] << endl;
4  cout << p[1] << endl;
```

Noch Wach?

```
1  int m[2][3]={1,2,3,4,5,6};
2  int *p=m[0]+2;
3  cout << p[0] << endl;
4  cout << p[1] << endl;
```

3
4

→ Tafelbild

Inhalt

Nutzerdefinierte Statische Felder

... und Zeiger


```
1  #include<iostream>
2
3  int main() {
4    const unsigned int n=3;
5    int m[n][n]={0,1,2,3,4,5,6,7,8};
6    int* const mp0=&m[1][0];
7    std::cout << *(mp0+1) << *(mp0-1) << std::endl;
8    int* mp1=m[0]+1;
9    std::cout << *(mp1+1)+2 << *(mp0+5)/(*(mp1+3)) << std::endl;
10   return 0;
11 }
```

erzeugt Ausgabe

42
42

Übrigens: Klammern um *(mp1+3) essentiell. Warum?

Nutzerdefinierte Statische Felder

... und Referenzen


```
1  #include<iostream>
2
3  int main() {
4    const unsigned int n=3;
5    int m[n][n]={0,1,2,3,4,5,6,7,8};
6    int& m11=m[1][1];
7    int*const& mp0=&m[0][0];
8    std::cout << m11 << *(mp0+2) << std::endl;
9    int*const& mp1=m[0]+1;
10   std::cout << *(mp1+1)+2 << *(mp0+8)/(*(mp1+3)) << std::endl;
11   return 0;
12 }
```

erzeugt Ausgabe

```
42
42
```

... (2x) klar!

Nutzerdefinierte Statische Felder

... und **auto**


```
1  #include<iostream>
2
3  int main() {
4    const unsigned int n=3;
5    int m[n][n]={0,1,2,3,4,5,6,7,8};
6    auto& mr=m; // auto mr=m yields copy
7    auto mp=&m;
8    std::cout << mr[1][2]; mr[1][2]+=1;
9    std::cout << mp[0][1][2] << std::endl;
10   return 0;
11 }
```

erzeugt Ausgabe 56 ... und nicht etwa 42!

Übrigens: mp[0][1][2] $\Leftrightarrow$ (*mp)[1][2]

Nutzerdefinierte Statische Felder

... und mehr **auto**


```
1  #include<iostream>
2
3  int main() {
4    const unsigned int n=3;
5    int m[n][n]={0,1,2,3,4,5,6,7,8};
6    auto& mr=m;
7    auto mp=&m;
8    std::cout << mr[2][2] << std::endl;
9    std::cout << (*(++mp))[0][-1] << std::endl;
10   return 0;
11 }
```

erzeugt Ausgabe

8
8

aufgrund von Modifikation des nichtkonstanten Zeigers mp.

Nutzerdefinierte Statische Felder

... und noch mehr **auto**


```
1  #include<iostream>
2
3  int main() {
4    const unsigned int n=3;
5    int m[n][n]={0,1,2,3,4,5,6,7,8};
6    const auto& mr=m;
7    auto const mp=&m;
8    std::cout << mr[2][2]++ << std::endl;
9    std::cout << (*(++mp))[0][-1] << std::endl;
10   return 0;
11 }
```

kann nicht übersetzt werden, da

```
8:23: error: increment of read-only location ‘mr[2][2]’
   8 |   std::cout << mr[2][2]++ << std::endl;
     |                ~~~~~~~^
9:21: error: increment of read-only variable ‘mp’
   9 |   std::cout << (*(++mp))[0][-1] << std::endl;
     |                     ^~
```

AUFWACHEN!

Schreiben sie NIEMALS solchen Code:

```
1  int m[2][3]={1,2,3,4,5,6};
2  int *p=&m[1][1];
3  cout << *(m[0]+5)-*p+*++p << endl;
```

Trotzdem ... was wird (wahrscheinlich) ausgegeben?

Wach?

Schreiben sie NIEMALS solchen Code:

```
1  int m[2][3]={1,2,3,4,5,6};
2  int *p=&m[1][1];
3  cout << *(m[0]+5)-*p+*++p << endl;
```

... ausgegeben wird

7

→ Tafelbild

Zusammenfassung

1D-Felder

2D-Felder

*n*D-Felder

Nutzerdefinierte Statische Felder und ...

- ... Zeiger
- ... Referenzen
- ... **auto**