Fachhochschule Nordwestschweiz
Hochschule für Technik

Programmieren in C++

Christian Lang (Lac)

20. September 2019

Arrays und Pointerarithmetik

Inhalt

- C-Arrays
- C-String
- `std::string`
- Array von Arrays
- Mehrdimensionale C-Arrays
- Pointer-Arithmetik
- `void`-Pointer auf Array
- `std::array`
- `std::vector`

C-Arrays

- Länge wird nicht im Array gespeichert
- Compiler kennt Länge, aber nur im aktuellen Sichtbarkeitsbereich
- grosse Arrays sollten auf dem Heap angelegt werden
- Wenn Länge zur Compiletime bekannt: statisch, sonst dynamisch

```
1  // statisch auf Stack
2  int static_array[100];
3
4  // dynamisch auf Heap
5  size_t length = ...;
6  int* dynamic_array = new int[length];
7  delete[] dynamic_array;
```

C-Strings

- Eindimensionales `char`-Array mit 0-Terminierung
- 0-Terminierung benötigt zusätzliches Byte
- Vereinfachte Initialisierung mittels String-Literal
- `sizeof` funktioniert nur im Sichtbarkeitsbereich wie erwartet.

```
1  // s zeigt auf eine Kopie des String-Literals
2  char s[] = "Test";
3  // Anzahl Zeichen im String + 1
4  size_t slength = sizeof(s);
5
6  // t zeigt direkt auf konstantes String-Literal
7  const char *t = "Test";
8  // Anzahl Bytes von Typ
9  size_t type_size = sizeof(t);
```

std::string

- Container-Typ aus der Standard-Library

```
1  #include <string>
2
3  std::string name = "clang";
4  name.size();         // Anzahl Zeichen
5  name.length();       // Anzahl Zeichen
6  name[2];             // Direkter Zeichen-Zugriff
7  name.c_str();        // Konverter zu C-String
8  name.begin();        // Iteratoren
9  name.substr(2,2);    // Sub-Strings
10 name.find("la");     // Such-Algorithmen
```

C-Array als Funktionsparameter

- Funktioniert als `char*` oder `char[]`
- Jeweils nicht ganz klar ob Länge bekannt. Darum wird üblicherweise `char*` mit zusätzlichem Länge-Parameter verwendet.
- Bei C-Strings kann die Länge mittels `std::strlen` evaluiert werden

```
1  void FillRandom(int* data, size_t data_size) {
2    ...
3  }
4
5  void FillString(char* data) {
6    const size_t data_size = std::strlen(data);
7    ...
8  }
```

Array von Arrays: Pascalsches Dreieck

- Entspricht mehrdimensionalen Arrays in Java
- Eigentlich dann Pointer auf Pointer auf Integer

```
1  void Create(int hoehe) {
2    int** dreieck = new int*[hoehe];
3
4    for (int i=0; i < hoehe; i++) {
5      dreieck[i] = new int[i+1];  // erzeuge Array von int
6
7      dreieck[i][0] = 1;
8      dreieck[i][i] = 1;
9      for (int j=1; j < i; j++) {
10       dreieck[i][j] =
11         dreieck[i-1][j-1] + dreieck[i-1][j];
12 } } }
```

Mehrdimensionale C-Arrays

- mehrdimensionale C-Arrays werden intern als eindimensional gespeichert
- Auch hier Sichtbarkeits-Problematik der Längen
- Nur erste Dimension kann dynamisch sein

```
1  const int dim1 = 2;
2  const int dim2 = 3;
3  int matrix[dim1][dim2]; // matrix ist nicht initialisiert
4
5  int matrix2[dim1][dim2] = {{ 1,2,3 }, { 4,5,6 }};
6
7  // die Länge der 1. Dimension ergibt sich
8  int matrix3[][dim2] = { 1,2,3,4,5,6 };
9
10 int m = matrix2[0][2];  // m == 3
11 matrix[1][0] = m;
```

Pointer-Arithmetik

Idee: Auf Basis einer bestehenden Memory-Adresse wird eine neue Adresse berechnet.

```
1 int x = *(p + 2);
2 // anstatt
3 int x = p[2];
```

Erlaubte Operationen

- +, +=, ++
- −, −=, −−
- Ergebnis ist vom gleichen Pointertyp
- +1 bedeutet nicht + 1 Byte, sondern + Anzahl Bytes des Zielobjekts des Pointer

Beispiel: Array-Initialisierung

```
1  static constexpr size_t kArraySize = 1024;
2  int* array = new int[kArraySize];
3
4  const int* end = array + kArraySize;  // Pointerarithmetik
5  for (; array != end; ++array) {       // Pointerarithmetik
6    *array = 0;
7  }
8
9  delete[] array;
```

Beispiel: Raster-Bild

```
1  using Byte = unsigned char;
2  const size_t width = 41;
3  const size_t height = 31;
4
5  Byte gray_image[width * height];
6  Byte* row = gray_image;
7
8  for(size_t v = 0; v < height; ++v) {
9    for(size_t u = 0; u < width; ++u) {
10     row[u] = static_cast<Byte>(u + v);
11   }
12   row += width;      // Pointerarithmetik
13 }
```

`void`-Pointer auf Array

Pointerarithmetik kann mit gleicher Syntax wie Iteratoren verwendet werden.

```
1  void* memset(void* dest, int byte_value, size_t count) {
2    // nicht geprüfter down-cast
3    uint8_t* const begin = static_cast<uint8_t*>(dest);
4
5    uint8_t* const end = begin + count;
6
7    for (auto* it = begin; it != end; ++it) {
8      *it = byte_value;
9    }
10 }
```

std::array

- Klassen-Template für Arrays mit fixer Grösse
- Speichert Länge und hat diverse andere Helper-Methoden
- Intern ein rohes C-Array

```
1  #include <array>
2
3  std::array<double, 4> storage = { 1.1, 2.2, 3.3, 4.4 };
4  std::cout << "value[2]=" << storage[2] << std::endl;
5  std::cout << "size=" << storage.size() << std::endl;
6  for(const auto& s : storage) {
7    std::cout << s << ", ";
8  }
9
10 storage.fill(1.0);
11 storage.data();     // roher Pointer
```

std::vector

- Klassen-Template für Arrays mit dynamischer Grösse
- Speichert Länge und hat diverse andere Helper-Methoden
- kann alles was auch `std::array` kann
- Intern dynamisches Memory auf Heap

```cpp
#include <vector>

// mit "a" initialisieren
std::vector<std::string> storage(32, "a");
storage.reserve(128);

// perfect forwarding für inplace Construction
storage.emplace_back("inplace");
storage.push_back("copy");
storage.erase(storage.begin());
```