

Modernes C++

...für Programmierer

Unit 04: Arrays, Zeiger, Referenzen

by

Dr. Günter Kolousek

Überblick

- ▶ Arrays
- ▶ C-Strings
- ▶ Kommandozeilenargumente
- ▶ Zeiger
- ▶ Referenzen

Arrays

- ▶ Speicherbereich *fester* Größe
- ▶ enthält hintereinander Elemente *eines* Typs
- ▶ keine Längeninformaton zur Laufzeit vorhanden!
- ▶ Name des Arrays als Zeiger auf erstes Element!

```
#include <iostream> // array.cpp
using namespace std;
int main() {
    int odd[4];
    cout << sizeof(odd) / sizeof(int) << endl;
}
```

4

Arrays – 2

```
#include <iostream> // array2.cpp
using namespace std;
int main() {
    int odd[] {1, 3, 5, 7};
    cout << odd[0] << ' ' << odd[4] << endl;
}
???
```

Arrays – 2

```
#include <iostream> // array2.cpp
using namespace std;
int main() {
    int odd[] {1, 3, 5, 7};
    cout << odd[0] << ' ' << odd[4] << endl;
}
```

??? z.B.:

1 -1077502000

Arrays – 2

```
#include <iostream> // array2.cpp
using namespace std;
int main() {
    int odd[]{1, 3, 5, 7};
    cout << odd[0] << ' ' << odd[4] << endl;
}
```

??? z.B.:

1 -1077502000

→ keine Überprüfung

Arrays – 2

```
#include <iostream> // array2.cpp
using namespace std;
int main() {
    int odd[]{1, 3, 5, 7};
    cout << odd[0] << ' ' << odd[4] << endl;
}
```

??? z.B.:

1 -1077502000

→ keine Überprüfung

→ Absturz möglich

std::string vs. C-String

▶ std::string

- ▶ benutzerdefinierter Typ der Standardbibliothek
- ▶ kennt seine Länge!
- ▶ Neue Syntax für Literale ab C++ 14:

```
using namespace std::literals;  
// or more specific: std::string_literals  
auto name{"Maxi Muster"s};
```

▶ C-String

- ▶ Array von char
- ▶ mit Nullzeichen ('\0') abgeschlossen
 - ▶ → keine Längeninfo!
- ▶ C-String Literal "abc"
 - ▶ Typ: const char[4]

C-String

```
#include <iostream> // cstring.cpp
using namespace std;
int main() {
    const char cstr[4]{"abc"};
    // also possible: const char cstr[]{"abc"};
    cout << cstr[0] << ' '
         << static_cast<int>(cstr[3]) << endl;
}

a 0
```

C-String – 2

```
#include <iostream> // cstring2.cpp
using namespace std;
int main() {
    char cstr[10]{'a', 'b', 'c'};
    cstr[5] = 'e';
    cout << cstr[0] << ' '
         << static_cast<int>(cstr[3]) << ' '
         << static_cast<int>(cstr[4]) << ' '
         << cstr[5] << ' '
         << static_cast<int>(cstr[6]) << endl;
}
```

a 0 0 e 0

C-String-Funktionen: String endet mit erstem '\0'!

C-String-Literal

- ▶ `"abc" ... const char[4]`
- ▶ `u8"äüö" ... UTF-8 const char[7]`
- ▶ `u"äüö" ... UTF-16 const char16_t[4]`
- ▶ `U"äüö" ... UTF-32 const char32_t[4]`
- ▶ `L"äüö" ... wide const wchar_t[4]`
- ▶ `"Hello," " World" ... äquivalent zu "Hello, World"`

Kommandozeilenargumente

- ▶ Parameter vs. Argument einer Funktion
 - ▶ Parameter: Funktion hat Parameter
 - ▶ auch: formaler Parameter
 - ▶ Argument: Funktion erhält Argument
 - ▶ auch: aktueller Parameter
- ▶ beim Aufruf des Programmes: Argumente
- ▶ alternative Form von main:
`int main(int argc, char* argv[])`
- ▶ argc Anzahl der Argumente inkl. Programmname
- ▶ argv wird mit 0 abgeschlossen, d.h.:
`argv[argc] == 0`

Kommandozeilenargumente – 2

```
#include <iostream> // sort.cpp
using namespace std;
// argc ... number of arguments
// argv ... array of "char*"
// char* ... char pointer
int main(int argc, char* argv[]) {
    for (int i{0}; i < argc; ++i) {
        cout << argv[i] << endl;
    }
}
```

```
$ sort a b c
sort
a
b
c
```

Arrays – 3

```
#include <iostream> // sort2.cpp
using namespace std;
// argc ... number of arguments
// argv ... array of "char*"
// char* ... char pointer
int main(int argc, char* argv[]) {
    for (int i{0}; i < argc; ++i) {
        cout << argv[i] << endl;
    }
    cout << sizeof(argv) / sizeof(char*) << endl;
}
???
```

Arrays – 4

Funktioniert nicht!

sort2.cpp: In Funktion »int main(int, char**)«:

sort2.cpp:8:24: Warnung: »sizeof« on array function

```
    cout << sizeof(argv) / sizeof(char*) << endl;  
                      ^
```

sort2.cpp:4:31: Anmerkung: hier deklariert

```
int main(int argc, char* argv[]) {
```

Arrays – 4

Funktioniert nicht!

sort2.cpp: In Funktion »int main(int, char**)«:

```
sort2.cpp:8:24: Warnung: »sizeof« on array function  
    cout << sizeof(argv) / sizeof(char*) << endl;  
                   ^
```

sort2.cpp:4:31: Anmerkung: hier deklariert
int main(int argc, char* argv[]) {

Standardmäßig ist bei g++ die Warnung
-Wsizeof-array-argument aktiviert...

Arrays – 4

Funktioniert nicht!

```
sort2.cpp: In Funktion »int main(int, char**)«:  
sort2.cpp:8:24: Warnung: »sizeof« on array function  
    cout << sizeof(argv) / sizeof(char*) << endl;  
                   ^
```

```
sort2.cpp:4:31: Anmerkung: hier deklariert  
    int main(int argc, char* argv[]) {
```

Standardmäßig ist bei g++ die Warnung
-Wsizeof-array-argument aktiviert...

mit -Wno-sizeof-array-argument deaktivieren...

Arrays – 4

Funktioniert nicht!

```
sort2.cpp: In Funktion »int main(int, char**)«:  
sort2.cpp:8:24: Warnung: »sizeof« on array function  
    cout << sizeof(argv) / sizeof(char*) << endl;  
                  ^
```

```
sort2.cpp:4:31: Anmerkung: hier deklariert  
    int main(int argc, char* argv[]) {
```

Standardmäßig ist bei g++ die Warnung
-Wsizeof-array-argument aktiviert...

mit -Wno-sizeof-array-argument deaktivieren... NEIN!

Arrays – 5

- ▶ → Arrays werden immer als Pointer auf das erste Element übergeben bzw. implizit konvertiert
 - ▶ decay (verfallen, zerfallen)
- ▶ → Arrays können *nicht* mittels Zuweisung kopiert werden:
`arr2 = arr1;`
- ▶ → Längenberechnung *nur* wenn Definition vorhanden (→ Compiler!)
 - ▶ ab C++17 auch mit `size()`!
- ▶ Ende der Kommandozeilenargumente kann an 0 erkannt werden
 - ▶ → Laufzeit!... daher `argc`
- ▶ mehrdimensionale Arrays werden als Arrays von Arrays dargestellt
- ▶ → (Immer) `std::vector` (oder `std::array`) verwenden

Zeiger (engl. pointer)

```
#include <iostream> // pointer.cpp
using namespace std;
int main(int argc, char* argv[]) {
    int age{42};
    cout << age << ' '; // access by name
    int* p{&age};
    // access by pointer:
    cout << p << ' ' << *p << endl;
    p = new int{3}; cout << *p << ' ';
    delete p; // don't forget → memory leak
    cout << *p << endl; // dangling pointer!
    //p = nullptr; cout<< *p<< endl; // segfault!
}
```

42 0x7ffef5453cfc 42

3 0

Zeiger – 2

```
#include <iostream> // pointer2.cpp
using namespace std;
int main(int argc, char* argv[]) {
    int age{42};
    int* p{nullptr}; // formerly: int* p{0};
    // shorter: int* p{};
    // age = nullptr; // error!
    p = &age;
    p = new int[10]{}; // initialized!
    cout << p[5] << endl; // 0
    int* q; // not initialized
    q = p; // assignement
    cout << *q << endl; // 0
    p = 0; // possible but not recommended
    delete[] q; // it's an array!
}
```

Zeiger – 3

```
#include <iostream> // pointer3.cpp
using namespace std;
int main(int argc, char* argv[]) {
    char name[]{"Maxi"};
    char* p{name}; // implicit conversion
    p = name; p = &name[0];
    cout << *p << endl;
    p = name + 4;
    cout << static_cast<int>(*p) << endl;
    // undefined: arbitrary value or termination
    cout << *(p + 500000) << endl;
}

M
0
... terminated by signal SIGSEGV (Adressbereichsf..
```

Zeiger – 4

```
int main() { // pointer4.cpp
    char mini[]{"x"}; char* p{mini};
    char s[]{"Maxi"};

    const char* pc{s}; // pointer to const char
    // pc[0] = 'm'; // error
    pc = p; // ok

    char* const cp{s}; // constant pointer
    cp[0] = 'm';
    // cp = p; // error

    const char* const cpc{s};
    // cpc[0] = 'm'; // error
    // cpc = p; // error
}
```

Zeiger – 5

```
#include <iostream> // pointer5.cpp
#include <string>
using namespace std;

struct Person {
    string first_name;  string last_name;
    int year_of_birth;
};

int main() {
    Person* p{new Person{"Max", "Mustermann", 90}};
    cout << (*p).first_name << endl; //parentheses!
    cout << p->last_name << endl;
    delete p;
    p = nullptr;  delete p;  // safe!
}
```


Zeiger – 6

Probleme mit "rohen" Zeigern:

- ▶ mehrmaliges Freigeben (mit `delete`) ist nicht definiert!
 - ▶ außer für Nullpointer (kein Effekte)!
 - ▶ d.h. nach jedem `delete`: auf `nullptr` setzen!
- ▶ Vergessen des Freigebens: → Speicherleck (engl. memory leak)
- ▶ Hängende Zeiger (engl. dangling pointer)
 - ▶ verweist auf nicht mehr gültiges Objekt

Referenzen

```
#include <iostream> // reference.cpp
#include <string>
using namespace std;
int main() {
    int x{1};
    int& r{x}; // other name for x!
    r = 2;
    cout << "x = " << x << endl;
    int* p{nullptr};
    p = &x;
    *p = 3;
    cout << "x = " << x << " r = " << r << endl;
}
```

x = 2

x = 3 r = 3

Referenzen – 2

Unterschiede zu Zeigern:

- ▶ Syntax unterschiedlich
 - ▶ $r = 2$ vs. $*p = 2$
- ▶ Pointer kann zu unterschiedlichen Objekten zeigen
 - ▶ Referenz wird bei Definition initialisiert!
- ▶ Pointer kann einen Nullwert haben.
- ▶ Zugriff über Pointer hat immer eine Indirektion
 - ▶ Referenz unter bestimmten Umständen nicht
- ▶ Kein Pointer auf Referenz!
- ▶ Kein Array von Referenzen!

Referenzen – 3

- ▶ lvalue - Referenz
 - ▶ Referenz auf einen lvalue
 - ▶ ohne `const`
 - ▶ mit `const`
 - ▶ implizite Konvertierung, sodass Typen übereinstimmen
 - ▶ Wert in temporäre Variable
 - ▶ temporäre Variable wird zur Initialisierung verwendet.
Lebenszeit endet, wenn Referenz den Scope verlässt.
- ▶ rvalue - Referenz
 - ▶ Referenz auf einen rvalue

Referenzen – 4

```
#include <iostream>    // reference2.cpp
#include <string>
using namespace std;
int main() {
    int* q{new int{1}};
    // int& r1{0};    // error: no lvalue
    // int& r2{q};    // error: wrong type
    {
        const char& r{65};
        cout << r << endl;
    }
}
```

A

Referenzen – 5

```
#include <iostream> // reference3.cpp
#include <string>
using namespace std;
int main() {
    string long_names[]{"maxi", "mini", "otto"};
    // find the appropriate type yourself
    // no change and no copy
    // → useful for long strings!
    for (const auto& name : long_names) {
        cout << name << endl;
    }
}
```

Referenzen – 6

```
#include <iostream> // rreference.cpp
using namespace std;
string f() {
    return "f()";
}
int main() {
    // at least one copy possible! (up to C++14)
    string res{f()};
    cout << res << endl;
}
```

Referenzen – 7

Welche Objekte werden bei der Rückgabe des Rückgabewertes erzeugt?

- ▶ `return`: Aus C-String-Literal ein `string` Objekt
- ▶ Rückgabe an Aufrufer: Kopie dieses Objektes
- ▶ Kopie des (temporären) Objektes bei der Initialisierung von `res`

Compiler?

- ▶ Compiler kann mittels Optimierungen temporäre Objekte vermeiden
- ▶ ab C++ 17 gibt es unter gewissen Umständen keine temporären Objekte mehr!

Referenzen – 8

```
#include <iostream> // rreference2.cpp
using namespace std;
string f() {
    return "f()";
}
int main() {
    // string& res{f()}; // error
    string&& res{f()};
    cout << res << endl;
}
```

Referenzen – 9

Compiler kann eine Kopieraktion gegen eine Verschiebeaktion austauschen!

- ▶ lvalue - Referenz: bezieht sich auf ein Objekt, das vom Benutzer beschrieben werden kann.
- ▶ konstante lvalue - Referenz: bezieht sich auf konstantes Objekt
- ▶ rvalue - Referenz: bezieht sich auf ein temporäres Objekt, das verändert werden kann, da es nicht mehr benützt wird.

Referenzen – 10

```
#include <iostream> // rreference3.cpp
using namespace std;
```

```
void swap(string& a, string& b) {
    string tmp{a};
    a = b;
    b = tmp;
}
```

```
int main() {
    string s1{"foo"}; string s2{"bar"};
    swap(s1, s2);
    cout << s1 << " " << s2 << endl;
}
```

```
bar foo
```

Referenzen – 11

```
#include <iostream> // rreference4.cpp
```

```
using namespace std;
```

```
void swap(string& a, string& b) {  
    string tmp{static_cast<string&&>(a)}; // move  
    a = static_cast<string&&>(b); // move assignme  
    b = static_cast<string&&>(tmp); // move assign  
}
```

```
int main() {  
    string s1{"foo"}; string s2{"bar"};  
    swap(s1, s2);  
    cout << s1 << " " << s2 << endl;  
}
```

```
bar foo
```

Referenzen – 12

```
#include <iostream> // rreference5.cpp
#include <utility>
using namespace std;

void swap(string& a, string& b) {
    string tmp{move(a)}; // the same as the cast!
    a = move(b);
    b = move(tmp);
}

int main() {
    string s1{"foo"}; string s2{"bar"};
    swap(s1, s2);
    cout << s1 << " " << s2 << endl;
}
```

Aber in `<utility>` gibt es schon eine generische swap Funktion!

Referenzen – 13

```
#pragma once
```

```
#include <iostream>
```

```
struct String { // string.h
    String(std::string s) : s{s} {};
    std::string s;
    String(const String& o) {
        s = o.s;
        std::cout << "copy" << std::endl;
    }
    String(String&& o) {
        s = o.s;
        std::cout << "move" << std::endl;
    }
};
```

Referenzen – 14

```
#include <iostream> // rreference6.cpp
#include <utility>
#include "string.h"
using namespace std;

void printit(String&& msg) {
    cout << "rvalue ref: " << msg.s << endl; }
void printit(const String& msg) {
    cout << "const lvalue ref: " << msg.s << endl; }

int main() {
    const String s{"abc"};
    printit(move(s));
}
```

const lvalue ref: abc

weil?

Referenzen – 14

```
#include <iostream> // rreference6.cpp
#include <utility>
#include "string.h"
using namespace std;

void printit(String&& msg) {
    cout << "rvalue ref: " << msg.s << endl; }
void printit(const String& msg) {
    cout << "const lvalue ref: " << msg.s << endl; }

int main() {
    const String s{"abc"};
    printit(move(s));
}
```

const lvalue ref: abc

weil? move belässt const, d.h. konstante rvalue ref kann nicht an rvalue ref gebunden werden!

Referenzen – 15

```
#include <iostream> // rreference7.cpp
#include <utility>
#include "string.h"
using namespace std;

void printit(String&& msg) {
    cout << "rvalue ref: " << msg.s << endl; }
void printit(String msg) {
    cout << "copy: " << msg.s << endl; }

int main() {
    const String s{"abc"};
    printit(move(s));
}
```

copy
copy: abc

?

Referenzen – 15

```
#include <iostream> // rreference7.cpp
#include <utility>
#include "string.h"
using namespace std;

void printit(String&& msg) {
    cout << "rvalue ref: " << msg.s << endl; }
void printit(String msg) {
    cout << "copy: " << msg.s << endl; }

int main() {
    const String s{"abc"};
    printit(move(s));
}
```

copy
copy: abc

? wie vorhin, jetzt (klarerweise) Kopie!