

Einführung in C++: Strings, Vektoren, ...

Thomas Hausberger, Matthias Panny
Ursprünglich erstellt von Sebastian Stabinger

SS2022

Strings

Strings in C

- Ein Array vom Typ `char`
- Wir müssen uns selbst um die Größe kümmern
- Ende des Strings ist durch das Zeichen `\0` gekennzeichnet

Beispiel für einen C String im Speicher

```
char str[] = "PROGRAM";
```

Index	0	1	2	3	4	5	6	7
Characters	P	R	O	G	R	A	M	\0
Address	1000	1001	1002	1003	1004	1005	1006	1007

Strings in C: Ein Beispiel

Wir wollen den Inhalt von zwei Strings `str1` und `str2` aneinanderhängen und das Ergebnis in `str3` speichern.

Beispiel

```
#include <stdio.h>
#include <string.h>

// Wir müssen uns selbst darum kümmern, dass str3 groß genug ist
char str1[100], str2[100], str3[200];
strcpy(str1, "Hello ");
// str1 = "Hello " funktioniert nicht!
strcpy(str2, "World");
// Hänge str1 und str2 zusammen und speichere in str3
strcpy(str3, str1);
strcat(str3, str2);
// Gib str3 auf Bildschirm aus
printf("%s", str3); // Wir müssen den Typ von str3 angeben (%s)
```

Man sieht: Die Verwendung ist **umständlich und unnatürlich**

Strings in C++

- Ein eigener Datentyp names `std::string`
- Die Größe wird dynamisch angepasst
- Verhält sich wie man es erwarten würde!

Beispiel

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

string str1, str2, str3; // Platz für beliebig viele Zeichen
str1 = "Hello ";        // Wir können einfach zuweisen
str2 = "World";
// Hänge str1 und str2 zusammen und speichere in str3
str3 = str1 + str2;
// Gib str3 auf Bildschirm aus
cout << str3 << endl; // Der Compiler kennt den Typ!
```

- Falls wir z.B. `str3` als C-String benötigen: `str3.c_str()`

Vergleichen von Strings

Um zu testen, ob zwei Strings gleich sind muss in C eine spezielle Funktion (`strcmp`) verwendet werden. In C++ erfolgt der Vergleich ganz natürlich mit dem Vergleichsoperator `==`, wie bei allen anderen Datentypen auch.

C

```
if (strcmp(str1, str2) == 0) {  
    printf("String 1 und 2 sind gleich");  
}
```

C++

```
if (str1 == str2) {  
    std::cout << "String 1 und 2 sind gleich";  
}
```

... in einen String

Zahlen können mittels der Funktion `to_string` in einen `string` konvertiert werden.

```
string s = to_string(42); // s enthält den String "42"
```

... von einem String

Ein String welcher eine Zahl enthält kann mit folgenden Funktionen in eine Zahl konvertiert werden:

- `stoi`, `stol`, `stoll` für Integer, Long und Long Long
- `stof`, `stod`, `stold` für Float, Double und Long Double

```
int i = stoi("42"); // i enthält die Zahl 42
```

Für komplexere Umwandlungen verwendet man einen `stringstream`.

Daten formatieren mit `stringstream` [optional]

- Man erzeugt einen `String Stream` mittels `stringstream` (benötigt `sstream` als Include)
- Schreiben wie in `cout`
- Um daraus einen String zu erzeugen: `.str()`

Beispiel

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

int main() {
    stringstream str_stream;
    str_stream << "Die Antwort ist " << 42 << " !";
    string str = str_stream.str();

    cout << str << endl;
} // Ausgabe: Die Antwort ist 42 !
```


Daten extrahieren mit stringstream [optional]

- `stringstream ss(string)` erzeugt einen Stringstream namens `ss` aus einem bereits existierenden String.
- Aus einem `stringstream` können wie mittels `cin` Daten ausgelesen werden

Beispiel

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

int main() {
    string s = "23 42 47";
    stringstream str_stream(s); // oder direkt str_stream("23 42 47")
    int a, b, c;
    str_stream >> a >> b >> c; // Einlesen der Daten

    cout << "a=" << a << " b=" << b << " c=" << c << endl;
} // Ausgabe: a=23 b=42 c=47
```

vector als Array-Alternative

Größe ist nicht Teil des Datentyps

```
int sum(int *a, int length) {  
    // Länge muss explizit mitgegeben werden  
    int sum = 0;  
    for (int i = 0; i < length; i++) {  
        sum += a[i];  
    }  
    return sum;  
}
```

Keine Überprüfung von Indexfehlern

```
int a[10];  
a[20] = 47; // Ouch! Kein Fehler!!
```

- Die Größe muss vorab bekannt sein wenn man nicht dynamische Speicherverwaltung verwenden will!

Verwendbar mittels `#include <vector>`

Vorteile:

- Automatisches Speichermanagement
- Dynamische Größe
- Überprüft auf Indexfehler (wenn man das will)
- Bietet viele komfortable Funktionen

Arraybeispiel mit `vector` implementiert

```
vector<int> a(100); // Vektor mit Platz für 100 Integer Werte
// Vektor mit den Zahlen 1 bis 5
vector<int> b = {1, 2, 3, 4, 5};
// Ausgabe des ersten Elements von a und fünften von b
cout << "a[0]=" << a[0] << " b[4]=" << b[4] << endl;
// Ausgabe: a[0]=0 b[4]=5
```

vector mit bekannter Größe

Ein Vektor mit bekannter Größe kann folgendermaßen deklariert werden: `vector<typ> name(größe);`

Beispiel

```
vector<int> a(100); vector<double> bla(400);
```

Im Gegensatz zu einem Fundamentalen Array kann die Größe aber auch erst **zur Laufzeit** festgelegt werden. D.h. **als Größe kann auch eine Variable** angegeben werden (in C seit C99 auch möglich) und die **Größe** kann problemlos **zur Laufzeit verändert** werden.

Beispiel

Angenommen `get_num` fragt den Benutzer nach einer Zahl und gibt diese zurück

```
int size = get_num();  
vector<int> vec(size);  
// vec hat die Größe welche der Benutzer eingegeben hat
```

Der leere Vektor

Für die Verwendung von `vector` muss die benötigte **Größe nicht von Anfang an bekannt sein**. Wir lassen die Größe einfach weg und erzeugen damit einen leeren Vektor: `vector<typ> name;`

Beispiel

```
vector<int> a; vector<double> bla;
```

Vorsicht!

Sie dürfen nicht auf Elemente eines Vektors zugreifen wenn diese nicht existieren!

```
vector<int> a; // Vektor hat Größe 0  
a[0] = 10; // Im besten Fall ein Speicherfehler ...  
cout << a[0] << endl; // ... im ungünstigsten Fall unvorhersehbar
```

Wofür ein leerer Vektor?

Wenn man nicht auf Elemente eines leeren Vektors zugreifen kann, wofür ist er dann nützlich?

Späteres Festlegen der Größe mit `resize`

```
vector<int> a;  
a.resize(100); // Vektor a hat jetzt Platz für 100 Integer  
a.resize(200); // Vektor a hat jetzt Platz für 200 Integer
```

Anhängen von Elementen mit `push_back`

```
vector<int> a;  
a.push_back(12);  
a.push_back(23);  
a.push_back(42);  
// a enthält nun {12, 23, 42}
```

Zugriff auf Elemente

```
vector<int> vec;  
vec.push_back(23); vec.push_back(42); vec.push_back(7);
```

Ohne Bounds Checking (schneller)

Funktioniert wie bei Arrays mittels `[index]`

```
vec[0]; // == 23  
vec[2]; // == 7  
vec[5]; // == ?? keinerlei Garantien
```

VisualStudio macht bei Debug Builds auch hier ein Bounds Checking! Kann bei den meisten Compilern eingestellt werden (z.B. für gcc mit dem Compilerflag `-D_GLIBCXX_DEBUG`)

Mit Bounds Checking (sicherer)

Mit Hilfe der Funktion `.at(index)`

```
vec.at(0); // == 23  
vec.at(2); // == 7  
vec.at(5); // Wirft zur Laufzeit zuverlässig eine Exception
```

Beide Versionen können auch für die Zuweisung von Werten verwendet werden: `vec[1] = 10;` bzw. `vec.at(1) = 10;`

Löschen von Elementen

- Es ist möglich **einzelne Elemente** aus einem Vektor mit folgender Funktion zu **löschen**:

`vektorname.erase(vektorname.begin() + position)`

- Falls ein anderes Element als das letzte im Vektor gelöscht wird ist die Operation **relativ langsam** (**list** ist für solche Fälle eine bessere Alternative zu **vector**)
- In den allermeisten Fällen aber trotzdem **schnell genug**!

Löschen von Elementen — Beispiel

Beispiel

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

int main() {
    vector<int> vec = {2, 6, 1, 28, 42, 23, 47, 7};

    vec.erase(vec.begin() + 3);

    for (int &i : vec)
        cout << i << " ";
}
```

Ausgabe

2 6 1 42 23 47 7

Weitere Nützliche Funktionen

- `.size()` Gibt die Anzahl an Elementen zurück
- `.empty()` Gibt `true` zurück falls der Vektor keine Elemente enthält
- `.data()` Gibt ein C-Array auf die Daten des Vektors zurück. Wichtig falls man mit C-Code interagieren muss.
- `.pop_back()` Löscht das letzte Element vom Vektor. Gegenstück zu `.push_back(element)`.
- `.back()` Gibt letztes Element des Vektors zurück ohne es zu löschen
- `==` Zwei Vektoren können mittels `==` verglichen werden. Die Vektoren gelten als gleich falls sie die gleiche Größe haben und die selben Elemente enthalten. `if(vec1 == vec2);`

Vector — Beispiel

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

int main() {
    vector<int> vec;
    cout << "Größe von vec: " << vec.size() << endl;
    vec.push_back(23);
    vec.push_back(13);
    vec.push_back(42);
    vec.push_back(7);
    cout << "Größe von vec: " << vec.size() << endl;
    // For-each
    for (auto e : vec)
        cout << e << " ";
}
```

Ausgabe

```
Größe von vec: 0
Größe von vec: 4
23 13 42 7
```

for-each

for - each (neu in C++11)

Iteriert automatisch über alle Elemente eines "Containers" z.B. Array, Vector, etc.

Beispiel

```
std::vector<int> vec = {23, 12, 42, 13, -40}; // Ein Array

for (int i : vec) {
    std::cout << i << " ";
}
```

Ausgabe

```
23 12 42 13 -40
```

for - each — Zuweisung

Um in einer for-each Schleife Werte in ein Array zu schreiben, muss der Laufvariable ein `&` vorangestellt werden. Dies kennzeichnet die Variable als eine sogenannte **Referenz** (eine Alternative zu Zeigern in C++)

Beispiel 1

```
std::vector<int> vec(100); // Ein Array der Größe 100

// Wir füllen das ganze Array mit dem Wert 23
for (int &i : vec) {
    i = 23;
}
```

Beispiel 2

```
std::vector<int> vec = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

```
// Quadriert alle Einträge in arr
```

```
for (int &i : vec) {  
    i = i * i;  
}
```

```
for (int i : vec) {  
    std::cout << i << " ";  
}
```

```
1 4 9 16 25 36 49 64 81 100
```


Funktionsüberladung

Funktionsüberladung

- Es können mehrere Funktionen mit **gleichem Namen** deklariert werden so lange die **Datentypen der Argumente eindeutig** sind
- Der **Compiler wählt die korrekte Funktion** anhand der Datentypen der an die Funktion übergebenen Argumente aus

OK

```
int f(int a);  
double f(double a);  
int f(short a);  
int f(int a, int b); // Parameteranzahl ist auch wichtig
```

Fehlerhaft

```
int f(int a);  
int f(int b); // Error: nur der Typ zählt  
double f(int a); // Error: Der Rückgabetyt wird ignoriert
```

Funktionsüberladung — Beispiel

```
#include <iostream>

// int wird quadriert
int f(int a) { return a * a; }
// Bei double wird 10 hinzugezählt
double f(double a) { return a + 10; }
// short wird verdoppelt
short f(short a) { return a * 2; }

int main() {
    int a = 5;
    double b = 10;
    short c = 4;

    std::cout << "int: " << f(a) << " ";
    std::cout << "double: " << f(b) << " ";
    std::cout << "short: " << f(c) << " ";
}
```

Ausgabe

```
int: 25 double: 20 short: 8
```

auto

auto (neu in C++11)

- Seit C++11 unterstützt C++ eine einfache Form der sogenannten **Typinferenz**¹
- Bedeutet: Falls der Compiler den Typ einer Variable selbst herausfinden kann, muss man ihn nicht angeben.
- Um Typinferenz zu verwenden schreibt man **auto** statt des eigentlichen Datentyps.

Beispiel

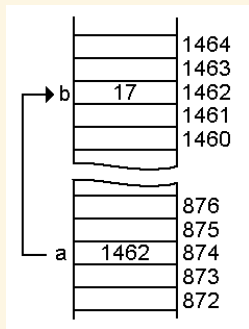
```
auto i = 10;    // i ist vom Typ int
auto tmp = f(); // tmp hat den Typ welcher von f
                // zurückgegeben wird
```

¹<https://de.wikipedia.org/wiki/Typinferenz>

Zeiger

Zeiger in C

- Zeiger sind Variablen welche Adressen speichern können (also normalerweise 64bit Integer)
- Zeiger haben einen Typ
 - Der Typ eines Zeigers gibt an, was an der Adresse auf die gezeigt wird gespeichert ist.
 - Im Zeiger selbst wird unabhängig vom Typ immer das gleiche gespeichert (eine Adresse)



Beispiel

```
int intVar = 10;
int *intPtr = &intVar;
cout << intVar << endl;
cout << *intPtr << endl;
intVar = 20;
cout << *intPtr << endl;
```

Probleme

```
int *intPtr2;
// zufaelliger speicherzugriff (pointer wird nicht initialisiert)
cout << *intPtr2 << endl;
int *intPtr3 = NULL;
// zugriff auf speicheradresse 0 (ueblicherweise segmentation fault)
cout << *intPtr3 << endl;
```


Probleme mit Zeigern

Probleme

- 1 Ein Zeiger muss nicht auf ein gültiges Objekt im Speicher zeigen
- 2 Ein Zeiger kann auch auf nichts zeigen (wenn `NULL` als Adresse gespeichert ist)
- 3 Die Syntax von Zeigern ist am Anfang verwirrend (`*`, `&`, `...`)

Vorteile

Nachdem Zeiger ein sehr einfaches Konzept sind (eine Variable welche eine Adresse speichert) ist es auch ein extrem mächtiges Werkzeug

Referenzen

Referenzen verweisen wie Zeiger auf Objekte im Speicher

Vorteile von Referenzen

- Referenzen lösen einige der Probleme mit Pointern
- Typprüfung des Compilers kann nicht mehr so leicht übergangen werden
- Referenzen können wie normale Variablen verwendet werden

Nachteile von Referenzen

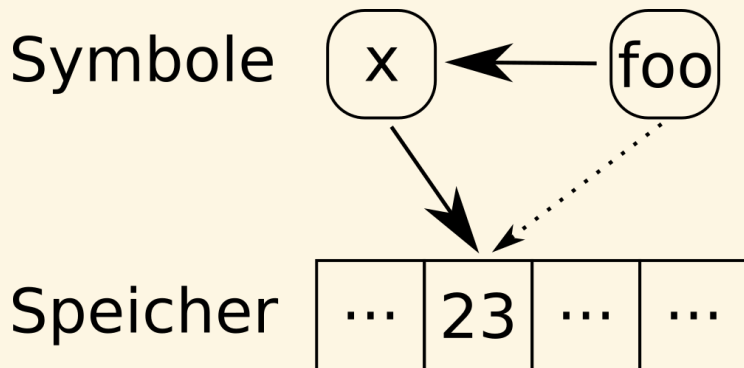
- Referenzen sind weniger flexibel als Zeiger
 - Keine Zeigerarithmetik
 - Keine Möglichkeit direkt auf die Adresse zuzugreifen

- Um eine Referenz zu erzeugen stellt man ein `&` an den Anfang eines Variablen- oder Parameternamens (Äquivalent zu dem `*` bei einem Zeiger)
- Bei Variablen muss zudem in der gleichen Zeile eine Referenz auf eine andere Variable zugewiesen werden!
- Um eine Referenz auf eine Variable zeigen zu lassen muss nicht wie bei Zeigern zuerst die Variable mittels `&` referenziert werden

```
int x = 23;  
int &foo = x;  
// foo ist jetzt eine Referenz auf x  
// (foo und x enthalten immer den gleichen Wert)
```

- Auf eine Referenz wird genauso zugegriffen wie auf eine gewöhnliche Variable (es ist kein Dereferenzieren mit einem `*` notwendig wie bei einem Zeiger)

```
foo = 42;  
std::cout << foo << " " << x << std::endl;
```



Vergleich zwischen Zeiger und Referenz

```
// Erzeugen von Variable, Zeiger und Referenz
int intVar = 10;
int *intPtr = &intVar;
int &intRef = intVar;
// Auslesen von Variable, Zeiger und Referenz
cout << intVar << endl;
cout << *intPtr << endl;
cout << intRef << endl;
// Zuweisen an Variable, Zeiger und Referenz
intVar = 20;
*intPtr = 30;
intRef = 40;

cout << intVar << endl; // Ausgabe = 40
```

Man sieht also, dass sich Referenzen genauso verwenden lassen wie Variablen, aber zu großen Teilen die Funktionalität eines Zeigers haben

- Referenzen als Variablen in "normalem" Code sind eher unüblich
- Referenzen werden am häufigsten bei Parametern von Funktionen verwendet:
 - Die übergebenen Parameter müssen dadurch nicht kopiert werden was gerade bei großen Klassen schneller ist
 - Innerhalb der Funktion können Änderungen an den Parametern vorgenommen werden welche auch ausserhalb der Funktion sichtbar sind. Normalerweise funktioniert das nicht, weil die Änderungen nur an einer Kopie vorgenommen werden.

Swap mit normalen Parametern, Zeigern, Referenzen

Normale Parameter

```
void swap(int p1, int p2) {  
    int temp = p1; p1 = p2; p2 = temp;  
}  
int a = 2, b = 3; swap(a, b);    // Verwendung
```

Sieht einfach aus, funktioniert aber auch einfach nicht ...

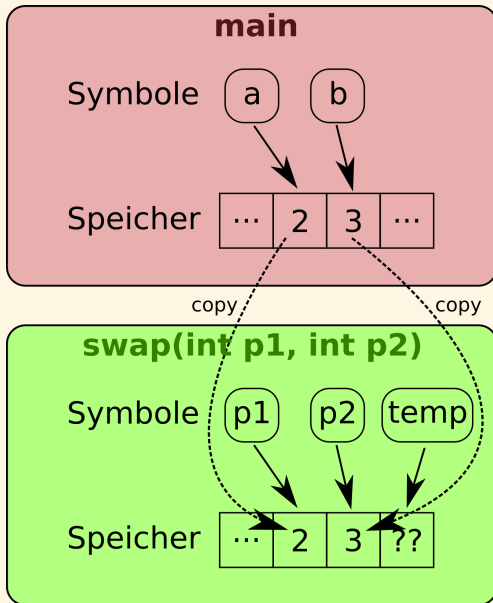
Zeiger

```
void swap(int *p1, int *p2) {  
    int temp = *p1; *p1 = *p2; *p2 = temp;  
}  
int a = 2, b = 3; swap(&a, &b); // Verwendung
```

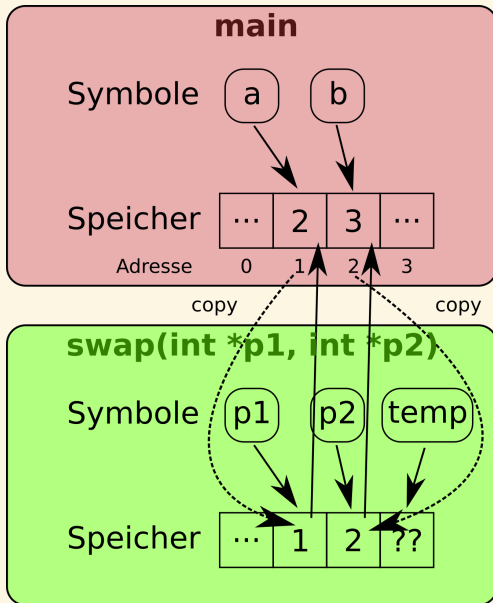
Referenzen

```
void swap(int &p1, int &p2) {  
    int temp = p1; p1 = p2; p2 = temp;  
}  
int a = 2, b = 3; swap(a, b);    // Verwendung
```

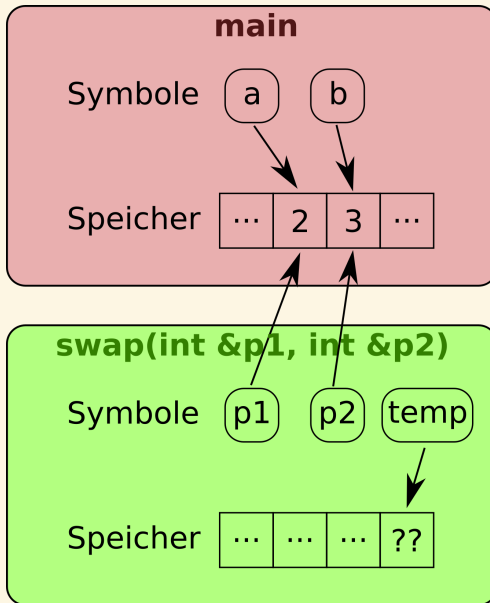

Swap – Graphische Visualisierung



Swap – Graphische Visualisierung



Swap – Graphische Visualisierung



Casten

Casten "normaler" Datentypen

Casten einer "normalen" Variable konvertiert (so gut wie möglich) den Inhalt einer Variable in einen anderen Datentyp

```
int i1 = 23;  
double d1 = (double)i; // Konvertiert i explizit nach double  
double d2 = 23.42;  
int i2 = d2; // Hier wird implizit von double nach int konvertiert
```