C++ By Example

Sebastian Stabinger*

March 21, 2018

Contents

*sebastian@stabinger.name

1 Einführung

Dieses Skriptum soll es Ihnen erleichtern dem Stoff der Lehrveranstaltung *Programmieren 3* zu folgen. Es ist umfangreicher als die Folien und enthält zudem mehr Beispiele als in der Lehrveranstaltung selbst besprochen werden können. Zudem enthält das Skriptum auch Teile die Ihnen bereits aus Ihren Lehrveranstaltungen zu C bekannt sein sollten und für die in *Programmieren 3* keine Zeit zur Wiederholung ist.

Ich bin der Auffassung, dass gerade Programmieranfänger von einer großen Anzahl an präsentierten Beispielen profitieren. Ich versuche daher zu jedem theoretisch eingeführten Konzept eine Vielzahl an ausführlich kommentierten Beispielen zu präsentieren.

Bedenken Sie bitte, dass dieses Skriptum kein Buch zum Thema C++ ersetzen kann. Aus zeitlichen Gründen muss ich viele Themenbereiche entweder komplett auslassen, oder bin nur dazu in der Lage sie kurz anzuschneiden. Sollten Sie Interesse an der Programmierung in C++ haben ist es sinnvoll sich weiterführende Literatur zum Thema zu besorgen.

1.1 Geschichte von C++

1979 Bjarne Stroustrup beginnt mit der Arbeit an "C with classes" (C mit Klassen). Es war als eine Erweiterung zu C gedacht welche die objektorientierte Programmierung in C erleichtert.

1983 Umbenennung in C++

1998 Erster ISO C++ Standard. C++ ist eine der wenigen Sprachen die offiziell standardisiert ist. Andere Beispiele sind C und Fortran. Es gibt keinen offiziellen Compiler für C++ (so wie es z.B. eine offizielle Version von Matlab, Java oder Python gibt). Daher war es wichtig einen Standard zu schaffen an den sich unterschiedliche Compiler halten können.

2011 Ein neuer ISO Standard wird veröffentlicht, der recht viel an der Sprache ändert. **Der C++11 Standard ist der Standard den wir in dieser Lehrveranstaltung verwenden werden.**

2014 ISO C++ 14. Kleine Änderungen am Standard. Betreffen uns in dieser Lehrveranstaltung nicht.

2017 ISO C++17. Größere Änderungen, aber nichts was uns in dieser Lehrveranstaltung berührt.

1.2 Verwandschaft zu anderen Sprachen

C++ ist ursprünglich als eine Erweiterung zu C entwickelt worden um die objektorientierte Programmierung zu erleichtern. Es ist daher immer noch fast jeder C Code auch gültiger C++ Code. Über die Jahre hat sich die Sprache C++ aber stark weiterenwickelt während die Sprache C relativ unverändert geblieben ist. Heute ist es so, dass C Code zwar prinzipiell auch gültiger C++ Code ist (mit kleinen Ausnahmen), aber schön geschriebener C++ Code hat nicht mehr viel mit C gemeinsam. Wenn C++ Code also aussieht wie C Code, dann ist er in der Regel schlecht geschrieben.

1.3 Warum C++

Sehr schnell, geringer Speicherverbrauch C++ basiert auf dem Konzept, dass ein Programm keine Dinge tut die man nicht explizit angibt. Dadurch sind Programme die in C++ geschrieben sind generell relativ schnell und verbrauchen wenig Arbeitsspeicher.

Hardwarenahe Programmierung C++ eignet sich genauso wie C für sehr hardwarenahe Programmierung. Man kann z.B. sehr genau bestimmen was bei der Ausführung in der CPU passiert, etc.

Verwenden von C-Bibliotheken Durch die historische Nähe von C++ zu C lassen sich C-Bibliotheken genauso einfach verwenden wie in C selbst. Das ist von Vorteil, da sehr viele C-Bibliotheken existieren. Speziell wenn man mit Hardware interagieren will, existiert üblicherweise eine C-Bibliotheken für diesen Zweck vom Hersteller.

Flexibel C++ ist vermutlich die flexibelste Sprache hinsichtlich des Einsatzgebietes. Man kann vom Mikroprozessor, über PCs/Konsolen, bis hin zu High Performance Computing auf riesigen Clustern mit zehntausenden CPUs alles programmieren. Und C++ wird in all diesen Bereichen auch tagtäglich eingesetzt.

Hohes Maß an Abstraktion Im Gegensatz zu C ist ein hohes Maß an Abstraktion möglich (OOP, Templates, Meta Template Programming, ...). Wir werden auf diese Dinge noch im Verlauf des Semesters genauer eingehen. Dieses höhere Maß an Abstraktion macht zwar die Sprache selbst komplexer (da es mehr Konzepte gibt die man lernen und verstehen muss), aber das Programmieren selbst wird dadurch letztlich einfacher wenn man die Konzepte verstanden hat.

Grössere Standardbibliothek Im Gegensatz zu C inkludiert C++ eine wesentlich größere Standardbibliothek. Wir werden im Laufe des Semesters einige nützliche Bereiche der Standardbibliothek kennen lernen. Im Vergleich zu den Standardbibliotheken von Sprachen wie Java, Python, C#, ... ist sie aber immer noch relativ klein.

Lange erprobt C++ existiert seit 35 Jahren und hat sich im industriellen Einsatz bewährt. Die Sprache hat ihre Probleme, aber man hat gelernt damit umzugehen. Es ist daher anzunehmen, dass die Sprache auch in Zukunft weiterhin ihren Platz haben wird. Im TIOBE Index welcher Sprachen nach ihrer popularität bewertet ist C++ z.B. seit Jahrzehnten in den Top-5 (C ist seit Jahrzehnten abwechselnd auf Platz 1 oder 2)

Sehr große Code Base Sehr viele Programme sind in C++ geschrieben. Zum Beispiel sind faktisch alle Anwendungen (Office, Webbrowser, ...) in C++ geschrieben. Spiele auf PC oder Konsolen werden heutzutage auch fast durchgehend in C++ entwickelt. Es gibt also sehr viel C++ Code der gewartet und weiter entwickelt werden muss.

2 C vs C++

Ich will Ihnen hier einen kleinen Vorgeschmack geben, wie einem C++ das Leben gegenüber C leichter machen kann. Dieses Kapitel ist als Vorschau gedacht und soll Ihnen C++ schmackhaft machen. Sie müssen noch nicht verstehen, was hier im Detail passiert.

2.1 Funktionsüberladung

Angenommen Sie wollen eine Funktion schreiben mit der man das Quardat einer Zahl bestimmen kann und Sie wollen diese Funktion für `int` (Ganzzahlen) als auch für `double` (Fließkommazahlen) implementieren.

2.1.1 In C

In C ist es nicht erlaubt zwei Funktionen mit dem gleichen Namen zu definieren, selbst wenn die Datentypen der Parameter unterschiedlich sind oder sie unterschiedliche Anzahl an Parametern haben. Wir müssen also für unsere Funktion zwei verschiedene Namen wählen und bei der Verwendung selbst daran denken die richtige Version zu verwenden.

```
int quadriere_int(int a) { return a * a; }
double quadriere_double(double a) { return a * a; }

int main() {
  int i = 2;
  double d = 2.0;

  int qi = quadriere_int(i);
  double qd = quadriere_double(d);
}
```

2.1.2 In C++

In C++ können wir mehreren Funktionen den gleichen Namen geben, so lange die Datentypen der Parameter unterschiedlich sind. Der C++ Compiler wählt dann anhand der Datentypen automatisch die richtige Version aus. Man bezeichnet das als **Funktionsüberladung** (function overloading) . In unserem Fall können wir also für die Quadrierfunktion für `int` und für `double` den gleichen Namen wählen und der Compiler verwendet automatisch die richtige Version.

```
int quadriere(int a) { return a * a; }
double quadriere(double a) { return a * a; }

int main() {
  int i = 2;
  double d = 2.0;

  int qi = quadriere(i);
  double qd = quadriere(d);
}
```

2.1.3 In C++ mit Templates

Wir haben aber immer noch das Problem, dass wir für jeden Datentyp eine eigene Version unserer Quadrierfunktion implementieren müssen obwohl ja unabhängig vom Typ immer das gleiche passiert (es wird einfach der übergebene Parameter mit sich selbst multipliziert). C++ ermöglicht es mittels sogenannter Templates, Funktionen zu schreiben welche automatisch für alle Typen funktionieren welche z.B. die Multiplikation unterstüzen.

```
template <typename T> T quadriere(T a) { return a * a; }

int main() {
  int i = 2;
  unsigned int ui = 2;
  short d = 2;
  double d = 2.0;

  int qi = quadriere(i);
```

```
  unsigned int qui = quadriere(ui);
  short qs = quadriere(s);
  double qd = quadriere(d);
}
```

Das `T` dient hier als ein Platzhalter für beliebige Datentypen.

2.2 Verwendung komplexer Zahle

Angenommen wir wollen das Ergebnis dieser Formel berechnen und müssen daher mit komplexen Zahen rechnen:

$$d = \frac{(1.3 + 5i) + 5}{(3.4 + 1i)^2}$$

Sowohl C als auch C++ bieten über ihre Standardbibliothek das Rechnen mit komplexen Zahlen an. **Achtung:** Die folgenden Beispiele verwenden nicht die Implementierungen wie sie von C und C++ zur Verfügung gestellt werden. Die Beispiele verdeutlichen nur, wie eine Implementierung aussehen könnte.

2.2.1 In C

Da C nur sehr wenige Möglichkeiten zur Abstraktion bietet müssen alle Operationen über Funktionen realisiert werden. Es ist in C z.B. nicht möglich eigenes Verhalten für `+` zu implementieren. Wir müssen daher zum Addieren von zwei komplexen Zahlen die Funktion `complex_add` verwenden, etc. Das macht den Code sehr unleserlich.

```
struct complex d =
    complex_div(complex_add(create_complex(1.3, 5), create_complex(5, 0)),
                complex_pow(create_complex(3.4, 1.0), 2));
```

2.2.2 In C++

C++ erlaubt die Implementierung von eigenen Versionen von `+`, `*`, ... Man bezeichnet dies als Operatorüberladung (operator overloading). Dadurch wir Code häufig leichter lesbar.

```
complex d = (complex(1.3, 5.0) + 5.0) / complex(3.4, 1.0) ^ 2.0;
```

2.3 Typinferenz

In C müssen wir den Datentyp einer Variable angeben, auch wenn der Compiler eigentlich selbst herausfinden könnte welchen Datentyp die Variable haben muss. Angenommen wir haben folgende Funktionen gegeben:

```
struct some_long_name get_struct();
void do_something_with_struct(struct some_long_name a);
```

Wir wollen nun mittels `get_struct` eine Variable befüllen und diese dann mittels `do_something_with_struct` weiterverarbeiten.

2.3.1 Verwendung in C

In C müssen wir den Datentyp der Variable `mystruct` explizit angeben, obwohl der Compiler ja selbst herausfinden könnte welchen Datentyp `get_struct` zurück gibt und den selben Typ für die Variable verwenden könnte.

```
struct some_long_name mystruct = get_struct();
// ...
do_something_with_struct(mystruct);
```

2.3.2 Verwendung in C++(11)

Seit C++11 unterstützt C++ eine einfache Form der Typinferenz. Dadurch muss der Datentyp einer Variable nicht angegeben werden, falls der Compiler den Datentyp selbst bestimmen kann. Um den Compiler zu veranlassen den Datentyp selbst zu bestimmen, wird statt des Datentyps das Schlüsselwort `auto` verwendet.

```
auto mystruct = get_struct();
// ...
do_something_with_struct(mystruct);
```

`auto` findet den Typ einer Variable also automatisch.

```
auto i = 42;   // i ist vom Typ int
auto d = 23.4; // d ist vom Typ double
auto c = 'A';  // c ist vom Typ char
```

3 Hello World

Heute ist es üblich, als erstes Beispiel einer Programmiersprache das sogenannte "Hello World" Programm zu implementieren. Es handelt sich dabei um ein Programm welches einfach den Text "Hello World" auf dem Bildschirm ausgibt. Begonnen hat diese Tradition im Buch The C Programming Language (DAS Standardwerk zu C) von Kernighan und Ritchie. Es folgt ein Vergleich dieses Programms in C (welches Sie bereits kennen sollten) und C++.

3.1 C

```c
#include <stdio.h>
int main() {
    printf("Hello World\n");
}
```

Die Headerdatei `stdio.h` wird eingebunden um `printf` verwenden zu können. Die Funktion `main` ist die Funktion die bei Ausführung eines Programms als erstes aufgerufen wird. Die Funktion `printf` ist der Standardweg um Text in C auf dem Bildschirm auszugeben.

3.2 C++

```cpp
#include <iostream>
int main() {
    std::cout << "Hello World" << std::endl;
}
```

Hier wird der Header für `iostream` eingebunden um `std::cout` verwenden zu können. Bei C++ ist es üblich die Dateierweiterung `.h` bei Headerdateien wegzulassen. In C++ ist `std::cout` der Standardweg um Text auf dem Bildschirm auszugeben. Der Bezeichner `std::endl` bewirkt das gleiche wie `\n` in C (ein Zeilenumbruch wird eingefügt).

4 Datentypen

Alle Datentypen die in C verfügbar sind können und werden auch in C++ verwendet.

4.1 Ganzzahlen

Ganzzahlen werden verwendet um ganze Zahlen (ohne Kommastellen) zu speichern. Die unterschiedlichen Datentypen für Ganzzahlen unterscheiden sich hinsichtlich der Wertebereiche welche in ihnen gespeichert werden können und wie viel Speicher eine einzelne Zahl belegt.

Leider ist die Größe dieser Datentypen in C nicht standardisiert (es ist nur die minimale Größe spezifiziert). Sollte man garantierte Größen benötigen kann man (seit C99) den Header `stdint.h` verwenden (weitere Info). Auf einem aktuellen PC mit 64 Bit Architektur sind folgende Größen üblich. Auf Mikroprozessoren und Embedded Devices können die Größen stark von dieser Tabelle abweichen.

Bezeichner	Übliche Größe auf PC	Üblicher Wertebereich auf PC
`char`	8 Bit	-128 ... 127
`unsigned char`	8 Bit	0 ... 255
`short`	16 Bit	-32768 ... 32767
`unsiged short`	16 Bit	0 ... 65536
`int`	32 Bit	-2147483648 ... 2147483647
`unsigned int`	32 Bit	0 ... 4294967296
`long`	32/64 Bit	-9223372036854775808 ... 9223372036854775807
`unsigned long`	32/64 Bit	0 ... 18446744073709551616
`long long`	64 Bit	-9223372036854775808 ... 9223372036854775807
`unsigned long long`	64 Bit	0 ... 18446744073709551616

Ganzzahlen mit mehr als 64 Bit werden üblicherweise nicht direkt von C unterstützt (einige Compiler unterstützen noch 128 Bit Ganzzahlen). Falls man größere Zahlen benötigt muss eine externe Bibliothek, z.B. GMP , verwendet werden. Mit dieser Bibliothek sind beliebig große Ganzzahlen möglich.

4.2 Fließkommazahlen

Fließkommazahlen sind eine Möglichkeit Zahlen mit Nachkommastellen in einem Computer darzustellen. Die Anzahl der Nachkommastellen ist nicht fix vorgegeben. Daher der Name **Fließ**-kommazahlen. C kennt die folgenden Datentypen für Fließkommazahlen.

Bezeichner	Größe	Wertebereich	Genauigkeit
`float`	32 Bit	$\pm 3.4 * 10^{38}$	ca. 7 Stellen
`double`	64 Bit	$\pm 2.3 * 10^{308}$	ca. 16 Stellen
`long double`	128 Bit	$\pm 3.4 * 10^{4932}$	ca. 19 Stellen

```
auto pi = 3.14159265358979323846;
auto tau = 2 * pi;
auto ganz = 3.0; // Auch 3.0 ist eine Kommazahl (mit .0000 als Kommastellen)
auto ganz2 = 4.; // Ein . ist ausreichend um eine Fließkommazahl zu erzeugen
auto expo = 3e3; // Exponentialschreibweise. Entspricht 3*10^3
double kein_integer = 23; // Wird automatisch in 23.0 umgewandelt
```

4.3 Welche Datentypen sollte man verwenden?

Generell faktisch immer `int` und `double`

C und C++ unterstützen sehr viele unterschiedliche Datentypen für Zahlen, aber nur wenige sind ohne spezielle Gründe sinnvoll. Falls Sie keine

SEHR guten Gründe haben, sollten Sie generell `int` und `double` verwenden. Gute Gründe sind fast ausschließlich die Interaktion mit Hardware welche bestimmte Datentypen benötigen.

Das Rechnen mit `short` und `float` ist auf PCs **nicht generell schneller** als das Rechnen mit `int` und `double` und häufig sogar langsamer.

Der Datentyp `char` stellt eine Ausnahme dar da er zum Speichern von Zeichen verwendet wird.

4.4 Boolsche Werte

Wie auch in C werden Wahrheitswerte (true/false) in C++ durch Ganzzahlen dargestellt. Die Zahl `0` repräsentiert `false` und alle anderen Zahlen (positiv wie auch negativ) repräsentieren `true`.

```
if(0) printf("true"); else printf("false");     // false
if(-0) printf("true"); else printf("false");    // false
if(23-23) printf("true"); else printf("false"); // false
if(42) printf("true"); else printf("false");    // true
if(-255) printf("true"); else printf("false");  // true

int a = 0;
int b = 47;
if(a) printf("true"); else printf("false");  // false
if(b) printf("true"); else printf("false");  // true
```

In C++ gibt es allerdings einen eigenen Datetyp zur Speicherung von Wahrheitswerten namens `bool`. Intern wird immer noch eine Ganzzahl gespeichert (die Größe ist nicht spezifiziert, aber häufig 1 Byte), aber die Verwendung von `bool` hat einige Vorteile:

1. Die Verwendung von `bool` macht klar, dass die Variable, der Parameter, der Rückgabewert nur Wahrheitswerte berücksichtigt. Bei Verwendung von z.B. `int` ist das häufig nicht offensichtlich.

2. Es gibt die Bezeichner `true` und `false` mit denen verglichen werden kann und die als Rückgabewert verwendet werden können, was den Code ebenfalls lesbarer macht.

3. Der Datentyp `bool` garantiert, dass intern nur `0` oder `1` gespeichert wird, selbst wenn andere Zahlen zugewiesen werden (was gültig ist)

In C ist etwas sehr ähnliches durch das Einbinden der Headerdatei `stdbool.h` verfügbar (Info). In C++ ist `bool` allerdings fixer Bestandteil der Sprache selbst.

4.4.1 Beispiel 1

Wir sehen uns die Verwendung von `bool` als Rückgabewert einer Funktion an die zurück gibt ob ein Integer gerade ist oder nicht.

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

bool ist_gerade(int zahl) {
  if (zahl % 2 == 0)
    return true;
  else
    return false;
}

int main() {
  int a = 23;
  if (ist_gerade(a)) // bool kann direkt als Bedingung verwendet werden
    cout << a << " ist gerade" << endl;
  else
    cout << a << " ist nicht gerade" << endl;
  // Ausgabe:
  // 23 ist nicht gerade

  int b = 42;
  bool ist_b_gerade =
      ist_gerade(b); // Wir können bool in einer Variable speichern
  if (b == true)     // Wir können explizit mit == vergleichen
    cout << b << " ist gerade" << endl;
  else
    cout << b << " ist nicht gerade" << endl;
  // Ausgabe:
  // 42 ist nicht gerade
}
```

4.4.2 Beispiel 2

Verwendung von `bool` als Parameter

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

void print(int zahl, bool mit_text) {
  if (mit_text)
    cout << "Zahl ist " << zahl << endl;
  else
    cout << zahl << endl;
}

int main() {
  int i = 42;
  print(i, false);
  print(i, true);
}
```

```
// Ausgabe:
// 42
// Zahl ist 42
```

4.4.3 Beispiel 3

Wir sehen, dass eine Variable vom Typ `bool` mit `cout` als Zahl (`0` oder `1`) ausgegeben wird und dass `bool` nur `0` oder `1` speichert, selbst wenn eine andere Zahl zugewiesen wird.

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

int main() {
  bool b1 = true;
  bool b2 = false;

  cout << "b1 = " << b1 << ", b2 = " << b2 << endl;
  // Ausgabe:
  // b1 = 1, b2 = 0

  bool b3 = 500;
  cout << "b3 = " << b3 << endl;
  // Ausgabe:
  // b3 = 1

  cout << "false = " << false << endl;
  cout << "true = " << true << endl;
  // Ausgabe:
  // false = 0
  // true = 1
}
```

4.5 auto

Seit C++11 unterstützt C++ eine einfache Form der sogenannten Typinferenz. Das bedeutet, dass wir den Datentyp einer Variable nicht explizit angeben müssen, wenn der Compiler einen eindeutigen Typ automatisch bestimmen kann. Der Compiler wird angewiesen den Datentyp selbst zu bestimmen indem das Schlüsselwort `auto` statt des Datentyps verwendet wird.

Das Schlüsselwort `auto` ändert nichts an der Tatsache, dass C++ den Datentyp einer Variable bereits beim Compilieren eindeutig wissen muss und dass sich dieser Datentyp während der Ausführung auch nicht ändern kann!

4.5.1 Vorteile

- Man kann sich teilweise das Schreiben langer Datentypen sparen. Dies ist insbesondere hilfreich wenn man Templateparameter verwendet, da

die Typnamen dadurch sehr lange werden können. Templates und Templateparameter werden später noch besprochen.

- Falls man den Datentyp an einer Stelle im Code ändert, ändern sich automatisch alle davon betroffenen Variablen mit `auto` automatisch. Ohne `auto` muss man möglicherweise an vielen Stellen im Code die Datentypen manuell anpassen. Siehe späteres Beispiel.

4.5.2 Nachteile

- Der Code kann durch Verwendung vieler Variablen mit `auto` unleserlich werden, da man nicht mehr auf einen Blick sieht was in einer Variable gespeichert ist.
- Falls bei einer Variable klar ist, dass sie für das korrekte Funktionieren des Codes einen bestimmten Datentyp haben muss (z.B. `int` für einen Index eines Arrays) kann es ungünstig sein für diese Variable `auto` zu verwenden, da es schwieriger wird Fehler zu finden.

4.5.3 Beispiel 1

Ein paar Beispiele von `auto` bei Zuweisungen

```
auto a = 23;       // a ist int
auto b = 42.0;     // b ist double
auto c = 'A';      // c ist char
auto d = 23 / 7;   // d ist int
auto e = 23 / 3.4; // e ist double
```

4.5.4 Beispiel 2

Hier ein Beispiel wie `auto` nicht verwendet werden kann:

```
auto i;
i = 20;
// Gibt einen Compilerfehler
```

Prinzipiell könnte der Compiler herausfinden, dass an `i` nach der Deklaration ein `int` zugewiesen wird und deshalb `i` ein `int` sein muss. Die Typinferenz in C++ ist aber nicht mächtig genug um das aufzulösen. `auto` funktioniert nur, falls in der gleichen Zeile etwas zugewiesen wird.

4.5.5 Beispiel 3

Ein großer Vorteil von `auto` ist, dass sich definierte Typen automatisch durch ein ganzes Programm durchziehen, ohne dass der Datentyp an jeder Stelle geändert werden muss. Angenommen wir haben den folgenden Code mit einem `int` als Anfangsdatentyp geschrieben:

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

int main() {
  int start = 23;
  int division = start / 3;
  int addition = division + 4;
  int ergebnis = division * addition;
  cout << "Ergebnis = " << ergebnis << endl;
}
// Ausgabe:
// Ergebnis = 77
```

Wollen wir nun alle Berechnungen mit `double` durchführen, so müssen wir alle daran beteiligten Datentypen ändern. Bei diesem einfachen Beispiel ist das kein großes Problem, aber wenn sich die ganzen Werte über ein größeres Programm verteilen, kann es schwierig werden alle Variablen zu finden und korrekt zu ändern.

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

int main() {
  double start = 23;
  double division = start / 3;
  double addition = division + 4;
  double ergebnis = division * addition;
  cout << "Ergebnis = " << ergebnis << endl;
}
// Ausgabe:
// Ergebnis = 89.4444
```

Wenn wir statt eines konkreten Typs `auto` verwenden, müssen wir nur den Typ des Anfangswerts `start` ändern und alle anderen Typen ändern sich automatisch mit.

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

int main() {
  auto start = 23;      // start ist int
  // auto start = 23.0; // start ist double
  auto division = start / 3;
  auto addition = division + 4;
  auto ergebnis = division * addition;
```

```
  cout << "Ergebnis = " << ergebnis << endl;
}
// Ausgabe bei start = 23:
// Ergebnis = 77
// Ausgabe bei start = 23.0:
// Ergebnis = 89.4444
```

4.6 Strings

4.6.1 Strings in C

In C sind Strings nichts anderes als Arrays vom Datentyp `char`. Jeder Eintrag im Array entspricht also einem Zeichen. Das Ende des Strings ist erreich, sobald ein Element des Arrays die Zahl `0` enthält. (Achtung: Nicht das Zeichen ~'0'~). Dies bedeutet also, dass die Länge des Arrays und die Länge des Strings nicht zwingend etwas miteinander zu tun haben. Z.B. erzeugt der Befehl `char str[200] = "Hallo";` einen String der Länge 5, welcher 6 Plätze des Arrays einnimmt (5 Zeichen des Strings und die `0` für das Ende des Strings). Das Array selbst ist aber 200 Elemente groß. Im Speicher sieht der String folgendermaßen aus:

Index	0	1	2	3	4	5	6	7	8	...	199
Zeichen	`H`	`a`	`l`	`l`	`o`	`\0`	undef	undef	undef	...	undef
Zahl	`72`	`97`	`108`	`108`	`111`	`0`	undef	undef	undef	...	undef

Die Art und Weise wie Strings in C implementiert sind hat einige Nachteile:

- Wir müssen uns selbst darum kümmern, dass ein Array groß genug ist um Platz für den String zu haben. Eine Ausnahme ist die Zuweisung eines fixen Strings direkt bei der Initialisierung eines Strings (z.B. ~char str[] = "Hallo"~). In diesem Fall kann der Compiler selbst bestimmen wie groß der String sein muss.

- Falls wir das Array zu klein wählen, bekommen wir keine Fehlermeldung. Das System schreibt einfach über das reservierte Array hinaus und überschreibt möglicherweise wichtige Speicherbereiche unseres Programms

- Falls aus irgend einem Grund die abschließende `0` des Strings fehlt, so lesen alle Stringfunktionen (z.B. `printf("%s",str)` einfach über das Stringende hinaus bis irgendwo im Speicher eine `0` steht)

Strings verhalten sich in C (weil es einfach Arrays sind) oft nicht so wie man es erwarten würde. Ein Beispiel: Wir wollen z.B. zwei Strings `str1` und `str2` aneinanderhängen und in einem String `str3` speichern. In C sieht das folgendermaßen aus:

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

int main() {
```

Wir müssen uns selbst darum kümmern, dass `str1`, `str2`, und `str3` groß genug ist. Aus diesem Grund wählt man die Arrays häufig größer als eigentlich notwendig.

```
char str1[100], str2[100], str3[200];
```

Nach der Initialisierung, können wir einem String in C nichts mehr mit `=` zuweisen. str1 = "Hello " funktioniert also nicht! Wir müssen eine spezielle Funktion namens `strcpy` (string copy) verwenden. Diese Funktion überprüft nicht, ob das Array auch groß genug für den String ist!

```
strcpy(str1, "Hello ");
strcpy(str2, "World");
```

Man könnte meinen, dass man zwei Strings aneinanderhängt indem man z.B. `str1 + str2` schreibt. Das ist in C aber nicht möglich. Um die Strings aneinanderzuhängen müssen wir zuerst `str1` nach `str3` kopieren.

```
strcpy(str3, str1);
```

Anschließend können wir die Funktion `strcat` (string concatenation) verwenden um einen zusätzlichen String an `str3` anzuhängen. Auch hier wird nicht überprüft ob `str3` groß genug ist um den ganzen String aufzunehmen!

```
  strcat(str3, str2);
  // Ausgabe von str3
  printf("%s", str3);
}
```

Man sieht also, dass Strings in C relativ kompliziert sind.

4.6.2 Strings in C++

In C++ gibt es einen eigenen Datentyp für Strings namens `std::string`. Dieser verhält sich viel mehr wie ein normaler Datentyp (`int`, `double`, ...). Um C++ Strings verwenden zu können müssen Sie diese String Bibliothek mittels `#include <string>` einbinden. Das letzte Beispiel sieht mit Verwendung von C++ Strings folgendermaßen aus:

```cpp
#include <iostream>
#include <string>

int main() {
  std::string str1, str2, str3; // Platz für beliebig viele Zeichen
  str1 = "Hello ";              // Wir können einfach mit "=" zuweisen
  str2 = "World";
  // Hänge str1 und str2 zusammen und speichere in str3
  str3 = str1 + str2;
  // Gib str3 auf Bildschirm aus
  std::cout << str3;
}
```

4.6.3 Vergleich von C++Strings

In C++ können Strings einfach mittels `==` verglichen werden. (In C funktioniert das nicht, hier muss die Funktion `strcmp` verwendet werden und das Ergebnis auf `0` überprüft werden. Noch ungünstiger ist, dass z.B. `str1 == str2` in C eine gültige Anweisung ist, aber nicht die Strings vergleicht sondern die Adressen an denen `str1` und `str2` liegen!). In C++ funktionieren auch die Vergleichsoperatoren `<`, `>`, `<=`, ... Sie testen die alphabetische Reihenfolge zweier Strings.

In C:

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

In C++:

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

4.6.4 Umwandlung von C++String nach C-String

Falls wir aus irgend welchen Gründen einen klassischen C-String aus einem C++ String erzeugen müssen (z.B. weil wir eine C-Bibliothek verwenden wollen) erreichen wir das mittels `.c_str()`. Wollen wir z.B. einen C-String aus `str3` erzeugen, schreiben wir `str3.c_str()`.

4.6.5 Beispiel 1

Wir können C++ Strings wie ganz normale Variablen behandeln und Sie problemlos als Parameter an eine Funktion schicken, oder sie als Rückgabewert einer Funktion verwenden.

Zum Beispiel: Eine Möglichkeit, `bool` mit `cout` als Text `true` und `false` auszugeben ist es den Wert mit Hilfe einer Funktion von `bool` nach `string` zu konvertieren und diesen String dann mittels `cout` auszugeben.

```
#include <iostream>
#include <string>
using namespace std; // Verhindert, dass wir überall std:: schreiben müssen

// Nimmt einen bool Wert entgegen (true, false) und liefert je nach
// übergebenem Wert des String "true" oder "false" zurück
string bool_to_string(bool b) {
  if (b == true)
    return "true";
  else
    return "false";
}

int main() {
  bool b1 = false;
  bool b2 = true;

  cout << "b1 = " << bool_to_string(b1) << endl;
  cout << "b2 = " << bool_to_string(b2) << endl;
  cout << "false = " << bool_to_string(false) << endl;
  cout << "true = " << bool_to_string(true) << endl;
}

// Ausgabe:
// b1 = false
// b2 = true
// false = false
// true = true
```

4.6.6 Beispiel 2

Da wir Strings sehr einfach auf Gleichheit überprüfen können und auch einfach als Parameter einer Funktion übernehmen können, ist es auch sehr einfach die Strings ~"true"~ und ~"false"~ wieder in einen `bool` Wert zu konvertieren.

```
#include <iostream>
#include <string>
using namespace std; // Verhindert, dass wir überall std:: schreiben müssen

// Nimmt einen String entgegen ("true", "false") und liefert je nach
// übergebenem String die bool werte true oder false zurück
bool string_to_bool(string s) {
  if (s == "true")
    return true;
  else
    return false;
}
```

```cpp
int main() {
  string s1 = "false";
  string s2 = "true";
  // Konvertieren der Strings nach bool
  bool b1 = string_to_bool(s1);
  bool b2 = string_to_bool(s2);

  cout << "b1 = " << b1 << endl;
  cout << "b2 = " << b2 << endl;
  // Das ganze geht natürlich auch ohne zusätzliche Variable
  cout << "false = " << string_to_bool("false") << endl;
  cout << "true = " << string_to_bool("true") << endl;
}

// Ausgabe:
// b1 = 0
// b2 = 1
// false = 0
// true = 1
```

5 Ein-/Ausgabe

Um die folgende Funktionalität für die Ein- und Ausgabe verwenden zu können muss die Headerdatei `iostream` mittels `#include <iostream>` eingebunden werden.

5.1 Ausgabe

Zur Ausgabe von Daten auf dem Bildschirm wird in C++ üblicherweise `std::cout` (für character out) verwendet. Variablen, Konstanten, Ausdrücke, etc. werden mittels `<<` nach `std::cout` geschickt. Das System erkennt automatisch den Typ der Daten die ausgegeben werden sollen und verwendet die korrekte Formatierung (man kann sich also die Formatplatzhalter von `printf` wie z.B. `%d`, `%f`, `%s`, etc. sparen)

```cpp
#include <iostream>
#include <string>

int main() {
  // Variablen
  int i = 23;
  double d = 42.47;
  std::string s = "Hello World!";
  std::cout << i;
  std::cout << d;
  std::cout << s;
  // Konstanten
  std::cout << 45;
  std::cout << 34.5;
  std::cout << "Noch ein Hello!";
```

```
  // Ausdrücke
  std::cout << 2 * i;
  std::cout << (i * d) / 3.4;
  std::cout << s + " World2!";
}
// Ausgabe:
// 2342.47Hello World!4534.5Noch ein Hello!46287.297Hello World! World2!
```

Die gleichzeitige Ausgabe mehrerer Daten kann durch wiederholtes aneinanderhängen von `<<` erzielt werden. Will man z.B. die Variablen `a`, `b` und `c` ausgeben, dann schreibt man `std::cout << a << b << c;`.

```
#include <iostream>
#include <string>

int main() {
  int i = 23;
  double d = 42.47;
  std::string s = "Hello World!";
  // Variablen
  std::cout << i << d << s;
  // Konstanten
  std::cout << 45 << 34.5 << "Noch ein Hello!";
  // Ausdrücke
  std::cout << 2 * i << (i * d) / 3.4 << s + " World2!";
}
// Ausgabe:
// 2342.47Hello World!4534.5Noch ein Hello!46287.297Hello World! World2!
```

Es werden bei Ausgabe mehrerer Daten in einer Zeile keine automatischen Leerzeichen eingefügt. Falls man Leerzeichen hab will, müssen diese manuell zwischen den Daten ausgegeben werden.

```
#include <iostream>
#include <string>

int main() {
  int i = 23;
  double d = 42.47;
  std::string s = "Hello World!";
  // Variablen
  std::cout << i << " " << d << " " << s;
  // Konstanten
  std::cout << 45 << " " << 34.5 << " " << "Noch ein Hello!";
  // Ausdrücke
  std::cout << 2 * i << " " << (i * d) / 3.4 << " " << s + " World2!";
}
// Ausgabe:
// 23 42.47 Hello World!45 34.5 Noch ein Hello!46 287.297 Hello World! World2!
```

An vorherigen Beispielen sieht man, dass `cout`, genauso wie auch `printf`, keinen automatischen Zeilenumbruch einfügt. Um einen Zeilenumbruch zu

erzwingen gibt man entweder `std::endl` (für endline) oder wie in C ein `\n` aus.

```
#include <iostream>
#include <string>

int main() {
  int i = 23;
  double d = 42.47;
  std::string s = "Hello World!";
  // Variablen
  std::cout << i << " " << d << " " << s << "\n";
  // Konstanten
  std::cout << 45 << " " << 34.5 << " " << "Noch ein Hello!" << std::endl;
  // Ausdrücke
  std::cout << 2 * i << " " << (i * d) / 3.4 << " " << s + " World2!\n";
}
// Ausgabe:
// 23 42.47 Hello World!
// 45 34.5 Noch ein Hello!
// 46 287.297 Hello World! World2!
```

5.2 Eingabe

Analog zur Ausgabe von Daten mittels `std::cout` werden mittels `std::cin` (character in) Daten vom Benutzer eingelesen. Um Daten einlesen zu können muss bereits eine Variable deklariert sein in welche die Daten geschrieben werden können. Das eigentliche Einlesen geschieht mit dem Befehl `std::cin >> variablenname;`. Beachten Sie, dass die "Pfeile" von `cin` in die entgegengesetze Richtung weisen wie bei `cout`: `std::cin >> bla;` vs `std::cout << bla;`.

Wir lesen z.B. einen Integer ein und geben ihn wieder aus:

```
#include <iostream>

int main() {
  // Ausgabe der Eingabeaufforderung
  std::cout << "Bitte geben Sie einen Integer ein: ";
  // Deklarieren der Variable in der die Eingabe gespeichert wird
  int i = 0;
  // Einlesen vom Benutzer in die Variable
  std::cin >> i;
  // Zu diesem Zeitpunkt ist die Variable i mit dem Wert befüllt, den
  // der Benutzer eingegeben hat und wir können den Inhalt ausgeben
  std::cout << "Sie haben die Zahl " << i << " eingegeben!\n";
}
```

`cin` liest immer bis zum nächsten Leerzeichen und man kann auch hier mehrere Daten gleichzeitig einlesen indem mehrere Variablen mittels `>>` aneinanderhängt werden.

Wir lesen z.B. drei `double` Werte ein und geben die Summe dieser aus:

```cpp
#include <iostream>

int main() {
  double a = 0, b = 0, c = 0;
  std::cout << "Bitte geben Sie drei Zahlen ein: ";
  std::cin >> a >> b >> c;
  std::cout << "Die Summe der eingegebenen Zahlen = " << a + b + c << std::endl;
}
```

Bei der Aufforderung `Bitte geben Sie drei Zahlen ein:` kann nun z.B. `2.25 13.2 5.25` eingegeben werden und die Variablen werden der Reihe nach befüllt: `a = 2.25`, `b = 13.2` und `c = 5.25`.

5.2.1 getline

Wie erwähnt, liest `cin` jeweils bis zum nächsten Leerzeichen. Insbesondere für das Einlesen eines Strings kann das unpraktisch sein, weil man mit `std::cin` dadurch nur ein einziges Wort einlesen kann. Um längere Texte einlesen zu können gibt es einen speziellen Befehl namens `std::getline` welcher alles bis zum nächsten Zeilenumbruch einliest und in einer Stringvariable abspeichert.