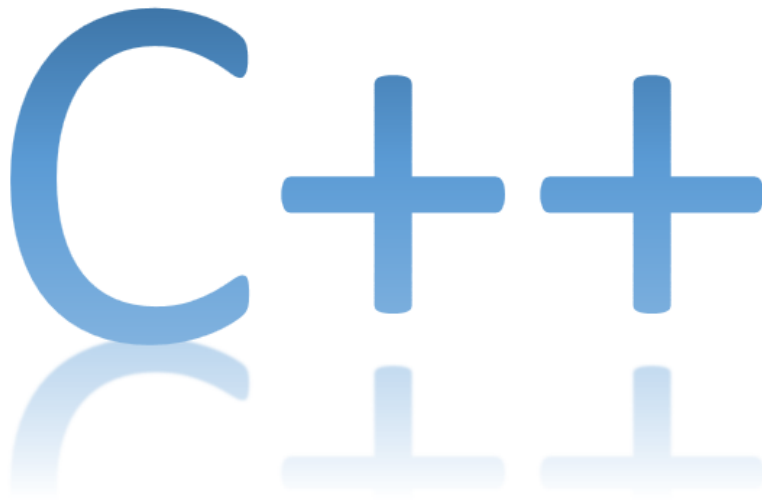


Version 0.02

Objektorientiert Programmieren



Ulrich Schrader

Nur zur Verwendung in der Vorlesung - Nicht weitergeben

Inhaltsverzeichnis

1	Einleitung	5
1.1	Die Arbeitsumgebung	5
1.2	Endlich beginnen	6
1.2.1	Elemente der objektorientierten Programmierung	6
1.3	Hello World!	7
1.4	Ein bisschen Interaktivität	10
1.5	Challenges	11
1.5.1	Freundliches Programm	11
1.5.2	Kalender	11
2	Umgang mit grundlegenden Datentypen	13
2.1	Datentypen	13
2.2	Variablen	14
2.3	Ausdrücke, Zuweisungen, Operatoren	17
2.3.1	Ausdrücke und Zuweisungen	17
2.3.2	Operatoren	18
2.4	Automatische Variablen mit <code>auto</code>	19
2.5	Präprozessor	21
2.6	Challenges	23
2.6.1	Formatierungsoptionen von <code>cout</code>	23

1 Einleitung

1.1 Die Arbeitsumgebung

C++ ist eine der älteren Programmiersprachen, aber auch eine der mächtigsten und weit verbreiteten Programmiersprachen, die heute noch verwendet werden. Die meisten Teile von Betriebssystemen, Datenbanken, Webserver, etc. sind in C++ geschrieben. Diese Vorlesung wird Sie mit den Grundlagen der objektorientierten Programmierung in dieser Sprache vertraut machen und Ihnen die grundlegenden Elemente näher bringen. Es ist keine komplette Einführung in die Sprache C++, da Kenntnisse, die Sie bereits in der C-Programmierung erworben haben, vorausgesetzt werden. Nichts desto trotz wird es gerade am Anfang eine Reihe von Überlappungen geben, wenn es darum geht die Unterschiede aufzuzeigen. Obwohl es nicht erforderlich ist eine integrierte Entwicklungsumgebung (Integrated Development Environment IDE) zu nutzen, macht eine solche Umgebung die Arbeit einfacher, da in ihr der Editor, der Compiler und Linker integriert sind. Ich persönlich nutze

- Visual Studio Community
- CodeBlocks

Beide stehen für nicht-kommerzielle Arbeiten kostenfrei zur Verfügung. Auf den Rechnern der Hochschule ist Eclipse installiert, welches ebenfalls ein kostenfreies IDE anbietet. Hinweise zur Installation unter Windows finden Sie in nachfolgendem Link:

https://www3.ntu.edu.sg/home/ehchua/programming/howto/EclipseCpp_HowTo.html

Für Unix-basierte System ist es noch einfacher. Meine Empfehlung ist daher, diese IDE auch auf Ihren Rechnern zu installieren, damit Sie dann in einer vertrauten Umgebung - gerade auch bei der Klausur - arbeiten können. Abschließend benötigen Sie noch eine Referenz für die C++ Sprache. Ich empfehle hierfür cppreference.com. Hierbei handelt es sich um ein frei zugängliches Wiki, dass als Referenz für C und C++ gilt. Dieses Skript übernimmt viele Inhalte aus dem sehr guten C++ Kurs von Eduardo Corpeño auf LinkedIn-Learning, sowie Inhalte aus den folgenden Quellen:

- Kaiser U.; Guddat M.: C/C++ Das umfassende Lehrbuch. Galileo Computing, 2014.
- Dmitrovic S.; Modern C++ for Absolute Beginners. Apress, 2020.
- Briggs W.; C++20 for Lazy Programmers. Apress, 2021.

1.2 Endlich beginnen

1.2.1 Elemente der objektorientierten Programmierung

Obwohl der Name C++ es nahelegt, ist C++ keine einfache Erweiterung der Programmiersprache C, sondern eine eigenständige Sprache mit einigen deutlichen Unterschieden, so dass nicht einfach C in C++ enthalten ist. Der wesentliche Unterschied ist, dass C++ eine objektorientierte Programmiersprache ist. Was bedeutet das? Zunächst versucht C++ die Realität zu *modellieren*. Dabei ist ein Modell als eine Repräsentation der Wirklichkeit zu verstehen. Modellierung ist ein ganz wesentliches Konzept der objektorientierten Programmierung. Dabei kann alles mehr oder weniger gut modelliert werden: eine Kuh, ein Auto, eine Person, eine Studentin, ... Der trickreiche Teil ist dabei, welche Elemente und Aspekte in dem Modell zu berücksichtigen sind, damit es der Aufgabe des Programms gerecht werden kann.

Nehmen wir an, wir wollen ein Auto modellieren, dann könnte man Eigenschaften aufnehmen, die jeden Aspekt des Autos, den man sich denken kann, berücksichtigen: Hersteller, Gewicht, Eigentümer, Farbe, Alter, Kilometerstand, Diese Liste könnte nahezu unendlich fortgesetzt werden, und wir würden nie mit dem Modellieren fertig werden. Daher besteht die Herausforderung des Modellierens darin, zu entscheiden, welche Eigenschaften relevant sind. So mögen der Preis, Hersteller, Modell, Anzahl der Sitze und Leistungsdaten für einen Händler relevant sein. Für einen Reisenden ist es eher ein Vehikel um von A nach B zu gelangen, daher mag ihn interessieren, wie viele Passagiere und wie viel Gepäck er mitnehmen kann. Der Betreiber eines Parkhauses ist eher an Daten interessiert, die Auskunft über die Größe des Autos geben.

Ein weiterer wesentliche Aspekt der objektorientierten Programmierung ist *Kapselung*. Kapselung bedeutet dabei, dass alle Daten und Operationen innerhalb des Modells gehalten werden. Gut definierte Modelle enthalten daher nur Informationen über sich selber. Die modellbezogenen Operationen sind knapp und präzise. Wenn eine Funktion A und B machen soll, dann macht sie auch nichts anderes. Daher sollte das Modell eines Autos etwa nicht die Kosten zum Füllen des Tanks beinhalten, da dieses eher als Bestandteil des Modells einer Tankstelle zu verstehen ist.

Als weiteres gibt es sogenannte *Klassen*, die in C++ als Konstrukte für Modelle anzusehen sind. Klassen beinhalten zwei Typen von Informationen: Daten und Funktionen, die oft auch als Methoden der Klasse bezeichnet werden. In unserem Beispiel mag unsere Autoklasse ein Datenelement zum Hersteller des Autos beinhalten, zum Modell sowie eine Funktion zum Ändern der Farbe des Autos. Eine spezielle Instanz der Klasse wird als *Objekt* bezeichnet. Wenn wir also drei verschiedenen Variablen der Klasse Auto haben, dann haben wir damit drei Objekt vom Typ Auto, die alle über die *Eigenschaften (Attribute)* Hersteller, Modell und Farbe verfügen und sich darin unterscheiden können.

Darüber hinaus können Elemente der Klasse *public* (öffentlich) oder *private* (privat) sein. Öffentliche Elemente sind aus jeder Stelle des Programms ansprechbar, wohingegen private Element nicht innerhalb der Klasse selber verwendet werden können. Aber es gibt

noch eine dritte Möglichkeit den Zugriff zu regeln, *protected* (geschützte) Elemente einer Klasse können in sogenannten vererbten Klassen verwendet werden. Dieses führt auf die Konzepte der *inheritance* (Vererbung) und der *Polymorphismen*, die in C++ umgesetzt sind. So kann eine *subclass* (Unterklasse) Element der *superclass* erben. Zum Beispiel mag die Klasse Tier ein Datenelement für die Anzahl der Beine enthalten. Dieses wird von der Unterklasse Hund geerbt. Unterklassen können weitere Elemente mit aufnehmen, So könnte die Unterklasse Vogel ein Element für Flügel ergänzen. Eine weiterer interessanter Aspekt im Zusammenhang mit Vererbung ist der Polymorphismus. Dabei definiert die Überklasse eine Funktion, aber die vererbte Funktion wird in einigen Unterklassen anders implementiert, und macht diese Unterklassen damit polymorph. Diese Konzepte mögen Ihnen von anderen Programmiersprachen wie Java oder Python schon vertraut sein. C++ setzt all dieses um, und noch viel mehr.

1.3 Hello World!

Um Ihnen ganz einfach C++ schon einmal nahezubringen, setzen wir wieder das schon bekannte Programm `hello world` um, dass nichts anderes macht, als eine Nachricht auf den Bildschirm der Konsole zu senden. Dieses einfache traditionelle Programm dient dazu, einen ersten Einblick in die Struktur und die Syntax zu bekommen.

Listing 1.1: Hello World

```

1 // HelloWorld1 - ohne Namespace
2 //
3
4 #include <iostream>
5
6 int main()
7 {
8     std::cout << "Hello_World!" << std::endl;
9 }
```

Wie Sie sehen können werden nur wenige Zeilen benötigt. Zuerst wollen wir die Bibliotheken, die wir verwenden wollen, einfügen. Dazu verwenden wir die Präprozessoranweisung `#include`, auf die wir später noch eingehen werden. Wir verwenden eine Bibliothek mit dem Namen `iostream`, die Teil der C++ Standardbibliotheken ist. Sie enthält u.a. Objekte und Funktionen, um Text auf dem Bildschirm darzustellen oder Text von der Tastatur zu erhalten. Präprozessoranweisungen enden nie mit einem Semicolon.

Als nächstes haben wir mit der von C bekannten Hauptfunktion `main()` den Einstiegspunkt des Programms. Jedes C++ Programm beginnt mit der `main()` Funktion und gibt einen Integerwert zurück. In diesem Fall bekommt sie keine Argumente übergeben, ansonsten würde sie ein Array von Strings übergeben bekommen, die beim Aufruf des Programms mit angegeben wurden.

Der Rumpf jeder Funktion ist ein Anweisungsblock, der in geschweiften Klammern ein-

1 Einleitung

geschlossen ist, die den Anfang und das Ende des Blocks angeben. Wie in C muss jetzt entschieden werden, wie wir die geschweiften Klammern setzen wollen. Dafür gibt es viele Möglichkeiten. Ich bevorzuge die öffnende Klammer am Ende des Kontrollblocks und die schließende Klammer in einer eigenen Zeile mit derselben Einrückung wie die öffnende Zeile. Sie werden diese Formatierung bei allen längeren Programmen wiederfinden. Diese Form wird von vielen anderen ebenfalls verwendet.

C++ Programme würde nahezu unleserlich, wenn diese nicht konsistent durch Einrückungen strukturiert werden. Einrückungen sind zwar für den Compiler nicht erforderlich, aber helfen dem Menschen den Überblick zu behalten. Die meisten Entwicklungsumgebungen unterstützen automatisch bei einer konsistenten Einrückung. Andere Programmiersprachen wie etwa Python verlangen Einrückungen, damit das Programm korrekt interpretiert werden kann. Gewöhnen Sie sich daher an eine konsistente Einrückung entweder immer um eine feste Anzahl Leerzeichen oder einen Tabulator.

Aufgabe: Recherchieren Sie mittels Google, welche weiteren Formen der Einrückung es gibt, indem Sie nach "Indentation Styles" suchen, und entscheiden Sie sich für Ihren Stil.

Zurück zum Programm, die erste Zeile der `main()` Funktion ist die Anweisung, die unsere Nachricht ausgibt. Dazu verwenden wir ein Objekt der `iostream` Bibliothek, das Bestandteil der Standardbibliothek von C++ ist. Diese Zugehörigkeit wird durch den Scope-Resolution-Operator `::` ausgedrückt. `std::` bedeutet also, das nachfolgende gehört zum Namensraum der Standardbibliothek von C++. Diese ist eine wichtige Eigenschaft, da es gerade bei umfangreichen Anwendungen bei denen viele beispielsweise Funktionen entwickeln, oder auch extern entwickelte Funktionen eingesetzt werden, es zu Namensüberschneidungen kommen kann. Diese können durch Namensräume aufgelöst werden.

Das hier verwendete Objekt ist `cout`, und steht für Character Out, also Zeichenausgabe. Es handelt sich dabei um eine Instanz der Klasse `ostream`, die generell für Output-Streams entwickelt wurde. Mit dem `<<` Operator können Variablen und Konstante zur Ausgabe an das `cout` Objekt gesandt werden. Dieses kann auch verkettet werden, indem nacheinander weitere Inhalte an des Objekt gesandt werden, wie in diesem Beispiel gezeigt, indem zunächst "Hello World" und dann das Zeilenendezeichen geschickt wird, dieses könnte das bekannte `"\n"` oder die in `iostream` definierte Konstante `endl` sein, die im Namensraum `std` definiert ist. Wie in C wird jede Anweisung mit einem Semikolon beendet.

Zum Schluss, da `main()` eine Funktion, die einen Integerwert zurückgibt, ist, geben wir noch 0 als Rückgabewert an. Dabei kann der Wert in Klammern eingeschlossen werden, dieses muss aber nicht der Fall sein. Es wird 0 zurückgegeben, da dieses traditionell für einen korrekten Ablauf des Programms steht. Ein von 0 verschiedener Wert bedeutet meist einen Fehlercode, den Sie natürlich in der Dokumentation des Programms näher erläutert haben. Damit ist das Programm abgeschlossen.

Um sich häufige Angaben des jeweiligen Namensraumes zu erleichtern, kann man diesen auch einmal festlegen. Dieses werde ich in vielen der Programmbeispiele hier verwenden, da damit die Zeilen etwas kürzer werden und nicht dauernd umgebrochen werden müssen. Unser Programm lautet dann:

Listing 1.2: Hello World unter Verwendung des Namensraumes std

```

1 // HelloWorld2.cpp - Verwendung von namespaces
2 //
3
4 #include <iostream>
5
6 using namespace std;
7
8 int main()
9 {
10     cout << "Hello_World!" << endl;
11 }
```

Da `cout` und `endl` Elemente des `std` Namensraumes sind, wird mit

```
using namespace std;
```

erreicht, dass nicht mehr angegeben muss, dass diese Elemente aus dem Namensraum `std` kommen, und man spart sich die Angabe von `std::`. Von manchen wird diese Lösung als Stützräder für Anfänger bezeichnet, und Sie können sich entscheiden auf diese Zeile zu verzichten, da dieses häufig als schlechter Programmierstil angesehen wird, und es sich eine Reihe von Problemen daraus ergeben können.

Aufgabe: Recherchieren Sie mittels Google, welche Vor- und Nachteile sich aus der obigen Verwendung von `using namespace` ergeben, und entscheiden Sie, ob und wann Sie es verwenden wollen.

Mein Grund für die Verwendung in den Beispielen dieses Skripts ist, dass die Zeilen dann etwas kürzer werden, wenn der Namensraum nicht immer angegeben werden muss, und die Zeilen dann nicht so oft umgebrochen werden müssen. Ansonsten würde ich aber darauf verzichten.

Das Objekt `cout` bietet eine sehr einfache Möglichkeit der Ausgabe in dem Konsolfenster. Dabei werden unterschiedliche Datentypen von dem Objekt sinnvoll behandelt. Das nachfolgende Beispiel zeigt das auf:

Listing 1.3: Verwendung von cout

```

1 // CoutExample.cpp : Umgang mit dem cout Objekt
2 //
3
4 #include <iostream>
5
```

1 Einleitung

```
6 int main()
7 {
8     int i = 42;
9     float a = 3.141;
10    std::string s = "Hi there!";
11    std::cout << "Variable i equals " << i << std::endl;
12    std::cout << "Variable a equals " << a << std::endl;
13    std::cout << "Variable s equals " << s << std::endl;
14    return(0);
15 }
```

Das Objekt `cout` hat die unterschiedlichen Datentypen der Variablen automatisch so in auszugebene Zeichenketten gewandelt, dass sinnvolle Ausgaben erzeugt werden, wie in dem Listing 1.3 gezeigt wird. Damit bietet `cout` eine sehr einfache und komfortable Form der Ausgabe in dem Konsolfenster an. Übrigens ist mit dem Objekt `cerr` dasselbe für Fehlermeldungen möglich, die standardmäßig im Konsolfenster angezeigt werden, aber auch leicht in etwa Dateien zur Protoklierung der Fehlermeldungen umgelenkt werden können.

1.4 Ein bisschen Interaktivität

Die meisten Programme leben davon, dass der Anwender etwas eingibt, worauf das Programm dann in einer vordefinierten Art und Weise reagiert. Da es ein `cout` Objekt zur Konsolausgabe gibt, gibt es auch ein `cin` Objekt zur Eingabe über die Konsole. Das folgende Programm zeigt die Verwendung des `cin` Objekts.

Listing 1.4: Verwendung von `cin`

```
1 // UsingCin.cpp : Holen von Worten von der Tastatur
2 //
3
4 #include <iostream>
5
6 using namespace std;
7
8 int main()
9 {
10    string str;
11    cin >> str;
12    cout << str << endl;
13    return(0);
14 }
```

In diesem Beispiel wird ein String vom Anwender eingegeben und gleich wieder ausgegeben. Auch hier werden die `cin` und `cout` Objekte verwendet, die aus den C++ Standardbibliotheken kommen. Dafür müssen wir durch die `include <iostream>` Anweisung wieder die benötigten Deklarationen einfügen. In Zeile 10 wird dann eine Stringvariable

namens `str` deklariert. Beachten Sie, dass C++ den Datentyp `string` kennt. Wir werden darauf in dem nächsten Abschnitt eingehen. In dieser Variablen wird die Eingabe des Nutzers gespeichert. In der Zeile 11 haben wir jetzt das `cin` Objekt, dass in umgekehrter Richtung `>>` wie `cout` verwendet wird und auf die Zielvariable weist. Nach dem Ausführen dieser Zeile ist die Eingabe in der Variablen `str` gespeichert und kann in der folgenden Zeile über das Objekt `cout` ausgegeben werden.

Beachten Sie, dass `cin` in dieser Form nur bis zum ersten Leerzeichen liest. Wenn also ganze Zeilen, die Leerzeichen enthalten eingelesen werden sollen, muss dazu eine spezielle Funktion verwendet werden.

1.5 Challenges

1.5.1 Freundliches Programm

Schreiben Sie ein Programm, welches zunächst den Anwender nach seinem Namen fragt und ihn dann unter Verwendung des Namens freundlich begrüßt. Verwenden Sie dabei die Objekte `cin` und `cout`.

1.5.2 Kalender

Schreiben Sie ein Programm, dass den aktuellen Monat in dem untenstehenden Format ausgibt:

Oktober 2021						
Mo	Di	Mi	Do	Fr	Sa	So
				1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29	30	31

Verwenden Sie dabei das Objekt `cout`. Suchen Sie nach Möglichkeiten, die `cout` bietet, die Ausgabe zu formatieren.

2 Umgang mit grundlegenden Datentypen

Wie in jeder Programmiersprache ist es immer hilfreich die unmittelbar unterstützten Datentypen kennenzulernen. Einiges davon werden Sie bereits von C her kennen. Wie C unterstützt C++ nur einige wenige Datentypen.

2.1 Datentypen

- **int**

Integerzahlen können als **signed**, d.h. als mit Vorzeichen versehene Ganzzahlen verstanden werden. Ebenso ist es aber auch möglich sie als vorzeichenlos (**unsigned**) zu deklarieren. Sie repräsentieren dann nur positive Zahlen oder den Wert 0. Es muss beachtet werden, dass der Datentyp **int** implementationsabhängig ist. Meist hat er einen Umfang von 4 Bytes, allerdings ist es auch möglich, dass er nur 2 Bytes lang ist. Der Datentyp **long** ist mindestens 4 Byte groß. Darüber hinaus gibt es noch den Datentyp **long long** der mindestens 8 Byte groß sein muss und den Datentyp **short** mit mindestens 2 Byte.

- **char**

Der **char** Datentyp ist immer 8 Bit lang und wurde entwickelt, um den ASCII-Zeichensatz abzubilden. Man kann ihn aber auch wie einen 1-Byte langen Integer typ verwenden, ebenfalls in den Ausprägungen **signed** und **unsigned**. Will man portable Programme schreiben und genau die Bytelänge spezifizieren, dann bietet **stdint.h** die Deklarationen für eine sehr hilfreiche Bibliothek mit genau spezifizierten Datentypen, die genau die Vorzeichenbehandlung und die Länge eines Integerdatentypen spezifizieren. So ist etwa der Datentyp **uint32_t** ein 32 bit langer Integer mit der Eigenschaft **unsigned** und **int8_t** eine 8 bit lange Integerzahl mit Vorzeichen.

- **float**

Wie C unterstützt auch C++ Gleitkommazahlen. Damit können reelle Zahlen wie, pi, 1.33 oder -0.5 mit unterschiedlicher Genauigkeit gemäß dem IEEE 754 Binary Floating Point Standard abgebildet werden. C++ unterscheidet drei verschiedene Genauigkeiten **float**, **double** oder **long double**. Der Unterschied besteht in unterschiedlichen Genauigkeiten und Zahlenumfang, wobei allerdings **float** deutlich performanter ist als **double**. Für viele Anwendungen ist der **float** Datentyp

ausreichend.

- **bool**

Neu ist der Datentyp **bool**, der die Werte **true** und **false** repräsentiert. Dabei sind die Keywords **true** und **false** Bestandteile der Sprache C++, so dass diese direkt einer **bool** Variablen zugewiesen werden können. Ebenso werden weiterhin alle von 0 verschiedenen Werte als **true** und der Wert 0 als **false** interpretiert.

- **string**

Ebenfalls neu gibt es den Datentyp **string**, der zwar nicht direkt in C++ definiert ist, sondern als Klasse in der Standardbibliothek implementiert wurde.. Wie in C kann mit Strings als mit dem Wert 0 terminiertes Zeichen-Array gearbeitet werden. Die **string** Headerdatei deklariert nicht nur die Klasse, sondern auch eine Vielzahl von Funktionen zur Bearbeitung von Strings.

- **Pointer**

Natürlich bietet C++ auch die Möglichkeit wie in C mit Pointern zu arbeiten. Dieser Datentyp wird für Speicheradressen verwendet und kann als Referenz auf bestehende Variablen verwendet werden. Dabei kann ein und dieselbe Pointervariable nacheinander auf verschiedenen Variablen verweisen.

Dieses war sicherlich keine umfassende Beschreibung der grundlegenden Datentypen. Auch hier kann wieder nur auf cppreference.com verwiesen werden. Die detaillierte Beschreibung der Datentypen finden Sie unter **basic concept** und dann **fundamental types**. Sie werden dort noch weitere Varianten vorfinden.

2.2 Variablen

Es gibt eine Reihe wichtiger Aspekte beim Umgang mit Variablen in C++. Variablen sind temporäre Speicherbereiche. Sie müssen deklariert werden, bevor darauf zugegriffen wird. Dabei muss, wie in C, die Deklaration den Datentyp und den Variablennamen festlegen. Eine Deklaration kann zusätzlich noch die Initialisierung der Variablen mit einem bestimmten konstanten Wert umfassen. Im weiteren Verlauf des Programms können dann den Variablen (neue) Werte zugewiesen werden. In dem Listing 2.1 erfolgt die Zuweisung für einige Variablen mit unterschiedlichen Datentypen.

Für den Datentyp **integer** gibt es einige spezielle Formate, die beachtet werden müssen:

- 123 oder -3

Das Dezimalformat ist das Standardformat und entspricht den Werten, so wie wir sie lesen würden.

- 010

Beginnt eine Zahl mit einer führenden 0, dann wird diese als Oktalzahl interpretiert. Diese Zahl entspricht also dem Dezimalwert 8.

- `0x10`
Beginnt eine Zahl mit einem führenden `0x`, dann handelt es sich um eine Hexadezimalzahl. In dem Beispiel also den Dezimalwert 16.
- `0b10`
Beginnt die Zahl mit einem führenden `0b`, dann handelt es sich hier um eine Binärzahl. In dem Beispiel also hat sie den Dezimalwert 2.

Wird die Zahl mit einem kleinen oder großen `U` beendet, dann wird diese als eine Zahl mit der Eigenschaft `unsigned` interpretiert.

Gleitkommazahlen werden mit dem Dezimalpunkt und mindestens einer Ziffer rechts davon angegeben. Auch, wenn es sich um den Wert einer ganzen Zahl handelt, muss die Ziffer rechts des Dezimalpunkts angegeben werden. Soll es sich um einen `float` Wert handeln, dann muss die Zahl mit einem `f` wie in `123.5f` abgeschlossen werden. Wird das `f` weggelassen, dann wird die Zahl als `double` interpretiert. Doubles sind damit der Standarddatentyp für Gleitkommazahlen.

Variablen vom Typ `char` können einerseits Ganzzahlen, solange sie sich mit einem Byte darstellen lassen, zugewiesen werden. Zeichen können aber auch in bekannter Weise wie `'a'`, `'A'` oder mit Backslash `'\n'` oder `'\0'` zugewiesen werden.

Die Zuordnung zu Strings haben wir bereits gesehen, indem wir das Stringliteral `"Hello World"` zugewiesen haben.

Beispiele für die Zuweisung von Werten zu Variablen sehen sie in dem Listing 2.1. Gehen wir das Beispiel einmal Zeile für Zeile durch.

Listing 2.1: Umgang mit Variablen

```

1 // Variablen.cpp : Zuweisung zu Variablen.
2 //
3
4
5 #include <iostream>
6
7 using namespace std;
8
9 int a, b = 5; // single line comment
10
11 /* Multi
12  * line
13  * comment */
14
15 int main() {
16     bool my_flag;
17     a = 7;
18     my_flag = false;
19     cout << "a_=" << a << endl;
20     cout << "b_=" << b << endl;

```

2 Umgang mit grundlegenden Datentypen

```
21     cout << "flag_=" << my_flag << endl;
22     my_flag = true;
23     cout << "flag_=" << my_flag << endl;
24     cout << "a+b_=" << a + b << endl;
25     cout << "b-a_=" << b - a << endl;
26     unsigned int positive;
27     positive = b - a;
28     cout << "b-a(unsigned)_=" << positive << endl;
29     return (0);
30 }
```

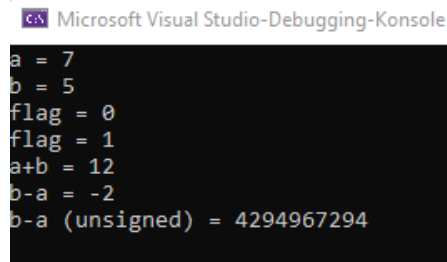
Zunächst sehen wir den Umgang mit Kommentaren. Wie in C gibt es zwei Möglichkeiten alles nach einem `//` wird als einzeiliger Kommentar verstanden, wie in Zeile 1,2 und Zeile 9. Alles zwischen der Zeichenkombination `/*` und `*/` wird ebenfalls als Kommentar angesehen. Dieser kann sich durchaus über mehrere Zeilen erstrecken.

In Zeile 9 werden dann zwei Integervariablen `a` und `b` deklariert. `b` wird dabei mit dem Wert 5 initialisiert. Nicht immer ist es guter Programmierstil mehrere Variablen in einer Zeile zu deklarieren. Da diese Variablen außerhalb der `main()` Funktion deklariert werden, handelt es sich hier um globale Variable, auf die von jeder Stelle des Programms zugegriffen werden kann. Daher werden diese auch statisch durch den Compiler im Datenssegment des Programms angelegt, dass während der gesamten Programmlaufzeit für die Variablen zur Verfügung steht. Erst wenn das Programm beendet wird, wird dieser Speicherplatz wieder freigegeben.

Meist sollten Variablen mit einem begrenzten Geltungsbereich angelegt werden. Damit ist gemeint, dass diese Variablen innerhalb von Funktionen, Anweisungsblöcken oder innerhalb von Schleifen deklariert werden. Diese Variablen sind nur innerhalb ihres Geltungsbereiches sichtbar und zugänglich. Wenn das Programm den Geltungsbereich verlässt, werden die Variablen gelöscht und der Speicherplatz freigegeben. Diese werden durch den Compiler verwaltet und in der Regel temporär im sogenannten Stackbereich des Speichers angelegt.

So deklarieren wir innerhalb des Geltungsbereichs der `main()` Funktion in Zeile 17 eine Bool-Variable namens `my_flag`. Für die Benennung der Variablen gibt es Regeln, natürlich dürfen Variablennamen nicht den Schlüsselworten der C++ Sprache entsprechen. So darf es etwa keine Variable mit dem Namen `return` geben. Kurz gesagt, es muss jeder Variablenname mit einer Nicht-Ziffer beginnen, und darf dann aus Buchstaben, Ziffern und Symbolen, wie etwa Binde- oder Unterstrichen, bestehen. Genaues können Sie unter cppreference.com nachlesen.

In Zeile 17 wird der Variablen `a` in der `main()` Funktion jetzt der Wert 7 zugewiesen. Der Zuweisungsoperator `=` arbeitet dabei immer von rechts nach links. Ebenso wird in Zeile 18 der boolschen Variable `my_flag` der Wert `false` zugewiesen. In den folgenden Zeilen 19-21 wird dann der Wert der Variablen ausgegeben. Wie in Abb, 2.1 gezeigt wird für den Wert des Flags 0, was aber `false` entspricht, ausgegeben.



```
Microsoft Visual Studio-Debugging-Konsole
a = 7
b = 5
flag = 0
flag = 1
a+b = 12
b-a = -2
b-a (unsigned) = 4294967294
```

Abbildung 2.1: Ergebnis der Ausgabe

In Zeile 22 wird dann `my_flag` der Wert `true` zugewiesen. In den folgenden Zeilen 23-25 wird dann wiederum das Flag, diesmal mit dem Wert 1 (`true`) und die Summe und Differenz aus `a` und `b` ausgegeben.

Zum Schluss überprüfen wir noch, was passiert, wenn wir einen negativen Wert einer Variablen vom Typ `unsigned` zuweisen. Dazu wird in Zeile 26 eine Variable `positive` vom Typ `unsigned int` deklariert. In C++ kann an nahezu jeder Stelle des Programmcodes eine Variable deklariert werden. Dieses Eigenschaft wurde erst später von C übernommen.

Dieser Variablen `positive` wird dann die Differenz `b-a`, die den negativen Wert -2 hat, in Zeile 27 zugewiesen. Der Wert von `positive` wird dann in Zeile 28 ausgegeben.

Falls das Ergebnis in der Abbildung 2.1 überrascht, dann kann man nachrechnen, dass es sich um den Wert $2^{32} - 2$, und damit dem 2er Komplement von -2 in Binärdarstellung entspricht. Die Binärdarstellung ist identisch, nur die Interpretation ist verschieden. Aus diesem Grund muss man vorsichtig sein, wenn sich entscheidet Variablen vom Typ `unsigned` oder `signed` zu deklarieren und in Ausdrücken gemischt zu verwenden.

2.3 Ausdrücke, Zuweisungen, Operatoren

2.3.1 Ausdrücke und Zuweisungen

Ausdrücke (expressions) sind in der Regel die symbolische Repräsentation einer Berechnung, genauso wie Sie es etwa von algebraischen Ausdrücken kennen. Bestandteile eines Ausdrucks können dabei Variablen, Konstanten und Operatoren sein. Wichtig ist dabei, dass jeder Ausdruck einen bestimmten Wert hat. Eine Zuweisung ist dann nichts anderes als eine Programmanweisung, die einer Variablen den Wert eines Ausdrucks zuweist, dabei steht die Variable, der der Wert zugewiesen wird immer auf der linken Seite, des Zuweisungsoperators `=` – der den Wert bestimmende Ausdruck immer auf der rechten Seite. Damit die linke Seite den Wert auch zugewiesen bekommen kann, müssen die Datentypen der linken und rechten Seite übereinstimmen. Wichtig ist dabei, dass dieses nicht durch den Compiler überprüft wird, wie wir in Listing 2.1 gesehen haben, wo einer `unsigned int` Variablen der negative Wert einer `int` Variable zugewiesen wurde. Daher liegt es in der Verantwortung des Programmierenden, darauf zu achten, dass der

2 Umgang mit grundlegenden Datentypen

Programmcode konsistent ist.

2.3.2 Operatoren

Hier noch einmal zur Wiederholung, da sich die Operatoren kaum von C unterscheiden. So haben wir arithmetische Operatoren, die bis auf Modulo für Gleitkomma- und Ganzzahlen erlaubt sind:

Tabelle 2.1: Arithmetische Operatoren

+	Addition
-	Substraktion
*	Multiplikation
/	Division
%	Modulo –Rest einer Division (nur Integer)

Es hängt von der Art der Operanden ab, ob eine Gleitkomma- oder ganzzahlige Operation ausgeführt wird.

Eine weitere Form der Operatoren sind bitweise boolsche Operatoren:

Tabelle 2.2: bit-bezogene Operatoren

&	bitwise and
	bitwise or
~	bitwise not
^	bitwise xor
>>	bitshift rechts
<<	bitshift links

Dabei ist zu beachten, dass diese Operationen bit für bit des Wertes einer oder mehrerer Variablen durchgeführt werden. Es sind keine logischen Operationen zwischen Variablen.

Logische Operationen sind solche Operationen, wie sie beispielweise in `if` Entscheidungen oder als Bedingungen in Schleifen verwendet werden, so wie Sie das von C bereits her kennen. Entscheidend ist, dass diese Operationen immer nur die Werte `true` oder `false` liefern. Erinnern Sie sich, dass `false` immer dem Wert 0 entspricht und `true` allen anderen.

Tabelle 2.3: Logische Operatoren

&&	logisch and
	logisch or
!	logisch not

Darüber hinaus gibt es noch relationale Operatoren, um damit Werte zu vergleichen.

Tabelle 2.4: Relationale Operatoren

<code>==</code>	ist gleich
<code>!=</code>	ist ungleich
<code><</code>	ist kleiner
<code>></code>	ist größer
<code><=</code>	ist kleiner oder gleich
<code>>=</code>	ist größer oder gleich

Vorsicht: Ein häufiger Fehler ist beim Vergleich auf Gleichheit nur ein einfaches `=` zu verwenden. Das Problem ist, dass dieses als Zuweisung verstanden wird, und der zugewiesene Wert dann als `true` oder `false` verstanden wird. Der Compiler kann nicht entscheiden, ob hier ein Fehler vorliegt, oder ob eine solche Anweisung gewollt ist.

Hier muss noch der sogenannte Spaceship-Operator `<=>` mit aufgenommen werden, der mit C++ 20 neu aufgenommen wurde und die relationalen Operatoren ergänzt. Dieser dient ebenfalls dem Größenvergleich zweier Objekte, resultiert aber in einen Wert kleiner 0, wenn eine `<` Beziehung erfüllt ist, dem Wert 0 bei Gleichheit und einem Wert größer 0, wenn eine `>` Beziehung gilt.

Abschließend gibt es noch die drei von C bekannten Pointer-Operatoren.

Tabelle 2.5: Relationale Operatoren

Präfix <code>*</code>	Indirektion. Hierdurch wird ein Pointer dereferenziert und erlaubt den Zugriff auf den Inhalt der Variablen, auf die er zeigt
Präfix <code>&</code>	Gibt die Adresse des Speicherplatzes einer Variablen zurück
<code>-></code>	Lässt auf Elemente einer Struktur oder Klasse zugreifen

Aufgabe: Da Ausdrücke ein ganz wesentliches Konzept der Sprache C++ sind, macht es Sinn ein wenig mehr darüber zu lernen. Googlen Sie beispielsweise `cpp reference operator precedence`, um zu sehen, in welcher Priorität komplexe Ausdrücke ausgewertet werden.

2.4 Automatische Variablen mit `auto`

Mit dem C++ 11 Standard wurden sogenannte automatische Variablen eingeführt. Hierbei handelt es sich um eine sehr bequeme Möglichkeit Variablen zu deklarieren, die an den

2 Umgang mit grundlegenden Datentypen

Datentyp angepasst sind, der ihnen zugewiesen werden soll. Allerdings geht dieses nur für die Basisdatentypen. Damit der Compiler entscheiden kann, um was für eine Variable es sich handeln soll, muss die Variable bei der Deklaration bereits initialisiert werden, das heißt, ihr muss ein Wert zugeordnet werden. Das Listing 2.2 zeigt die Verwendung automatischer Variablen.

Listing 2.2: Automatische Variablen mit `auto`

```
1 // AutoType.cpp : Beispiele für Auto Variablen
2 //
3
4 #include <iostream>
5 #include <typeinfo>
6
7 int main() {
8     auto a = 8;
9     auto b = 12345678901;
10    auto c = 3.14f;
11    auto d = 3.14;
12    auto e = true;
13    auto f = 'd';
14
15    std::cout << typeid(a).name() << std::endl;
16    std::cout << typeid(b).name() << std::endl;
17    std::cout << typeid(c).name() << std::endl;
18    std::cout << typeid(d).name() << std::endl;
19    std::cout << typeid(e).name() << std::endl;
20    std::cout << typeid(f).name() << std::endl;
21
22    return (0);
23 }
```

Das Schlüsselwort `auto` ersetzt dabei die Angabe des Datentyps. In dem Programm werden in Zeile 10 bis 15 sechs Variablen deklariert und mit Werten jeweils unterschiedlichen Datentyps initialisiert. So wird der Variablen `a` ein Integer zugeordnet. Die Variable `b` wird mit einer sehr großen Ganzzahl initialisiert, die sich nicht mehr in einer normalen `int`-Variablen ausdrücken lässt. Die Variable `c` bekommt einen `float`-Wert zugewiesen und `d` einen `double`-Wert. Die Variable `e` den booleschen Wert `true`. Zu Schluss wird noch der Variablen `f` das Zeichen `'d'` zugewiesen.

Im folgenden wird dann jeweils der Datentyp, den jede Variable automatisch erhalten hat, in einer eigenen Zeile ausgegeben. Hierzu wird der `typeid()`-Operator, der in der `typeinfo`-Headerdatei der Standardbibliothek definiert ist, verwendet. Dieser gibt ein `type_info`-Objekt zurück. Dessen `name()` Funktion wird verwendet, um dann den Namen des Datentyps der Variablen auszugeben. Es kann sein, dass bei einigen Implementationen lediglich ein Kürzel für den Datentyp ausgegeben wird. Daher wundern Sie sich nicht, wenn Sie andere Ergebnisse bekommen.

2.5 Präprozessor

C++ ist eine kompilierte Sprache, das bedeutet, dass der gesamte Programmcode nacheinander wie auf einem Fließband - einer Pipeline - eine Folge von Werkzeugen durchläuft, die versuchen die semantischen Elemente des Programms zu extrahieren. Im Gegensatz dazu stehen interpretative Sprachen, bei denen jede Anweisung für sich übersetzt und sofort ausgeführt wird. Dabei ist der Präprozessor einer der ersten Werkzeuge, die den gesamten Sourcecode verarbeiten. Man kann ihn sich wie ein automatisiertes Textverarbeitungsprogramm vorstellen, das nichts anderes macht, als bestimmte Textteile zu ersetzen, Texte aus anderen Dateien zu ergänzen oder, wenn bestimmte Bedingungen erfüllt sind, ganze Textteile zu löschen. Der Präprozessor wandelt den ursprünglichen Sourcecode, den Programmtext, in einen veränderten Sourcecode um. Dieser vorverarbeitete Programmcode wird dann an den eigentlichen Compiler weitergereicht.

Alle Präprozessoranweisungen starten mit dem Beginn der Zeile mit dem # Zeichen. Eine Präprozessoranweisung haben Sie bereits kennengelernt.

```
#include <...>
```

Die `#include`-Anweisung dient dazu den gesamten Inhalt der angegebenen Datei in das Programm anstelle der Anweisung einzufügen.

Listing 2.3: Einfache Präprozessoranweisungen

```
1 // PreprocessorExample.cpp : Example for simple preprocessor instructions
2 //
3 #include <iostream>
4 #include <string>
5 #include <cstdint>
6
7 #define CAPACITY 5000
8 #define DEBUG
9
10 using namespace std;
11
12 int main() {
13     int32_t large = CAPACITY;
14     uint8_t small = 37;
15 #ifdef DEBUG
16     cout << "[About to perform the addition]" << endl;
17 #endif
18     large += small; // shorthand for large = large + small
19     cout << "The large integer is " << large << endl;
20     return (0);
21 }
```

So wird in Zeile 3 in dem Listing 2.3 der gesamte Inhalt der Datei `iostream`, gefolgt von dem Inhalt der Dateien `string` und `cstdint` (Zeile 4 und 5) eingefügt.

2 Umgang mit grundlegenden Datentypen

Beachten Sie, dass der Dateiname in `<` und `>` eingeschlossen ist. Das hat zur Folge, dass nach diesen Dateien an einer vordefinierten Stelle gesucht wird. Es braucht also kein Pfad zu der Datei angegeben werden. Dieser wird meist durch die Entwicklungsumgebung vorgegeben. Auf diese Weise können alle Header-Dateien der Standardbibliotheken eingefügt werden. Diese Dateien benötigen auch nicht die Dateierweiterung `.h` oder `.hpp`.

Die in Zeile 5 eingefügte Datei `cstdint` ist eine Header-Datei der Standard-C-Bibliothek, und stellt Integer Datentypen mit fest definierten Längen zur Verfügung, so dass man von der Abhängigkeit von der jeweiligen Implementierung des `|int|` Datentyps befreit ist. Diese Header-Datei beinhaltet C Code. Daher beginnt der Dateiname mit einem vorangestellten kleinen `c`. Hierbei handelt es sich um eine für C++ aufbereitete Variante der C-Header-Datei `stdint.h`.

Die nächste Präprozessoranweisung ist `#define`. Diese Anweisung definiert Symbole, die jeweils das rechts davon stehende bedeuten. Diese Symbole werden auch als Makros bezeichnet. Aber eigentlich macht die Anweisung nichts anderes, als ein bestimmter Text in dem Sourcecode gefunden und ersetzt wird. So wird in Zeile 7 ein Symbol namens `CAPACITY` mit der Bedeutung 5000 definiert.

Üblich ist es für Makros ausschließlich Großbuchstaben zu verwenden, obwohl das nicht zwingend erforderlich ist. Aber es hat sich als guter Programmierstil etabliert.

Diese Anweisung bewirkt, dass an jeder Stelle im Sourcecode, wo der Präprozessor das Symbol `CAPACITY` findet, dieses durch 5000 ersetzt wird. Beachten Sie, dass Präprozessoranweisungen nicht mit einem Semikolon abgeschlossen werden.

In dem Hauptprogramm werden zwei Integer-Variablen unterschiedlicher Länge definiert. Die erste ist eine signed integer Variable der Länge 32 bit, so wie sie in `stdint` festgelegt ist. Würde der Datentyp mit einem `u` beginnen, dann wäre es eine unsigned Variable. Diese Variable `large` wird nun mit `CAPACITY` initialisiert. In Zeile 14 wird dann eine 8 bit lange unsigned Variable `small` deklariert und mit dem Wert 37 initialisiert. Diese wird dann in Zeile 18 zu dem Wert der Variablen `large` hinzu addiert, und das Ergebnis in Zeile 19 ausgegeben.

Der dritte Typ einer Präprozessoranweisung erlaubt das bedingte Einfügen von Programmcode. Dabei sind es einfache if-else Bedingungen, die in der Regel auf das Vorhandensein einer `#define`-Anweisung testen. Dabei handelt es sich natürlich nicht um bedingte Verzweigungen im Programmcode selber, sondern sie sind eher als Anweisungen zu sehen, die Programmcode unter bestimmten Bedingungen im Sourcecode stehen lassen, oder, die ihn entfernen, bevor der durch den Präprozessor vorverarbeitete Sourcecode compiliert wird. In dem Listing 2.3 gibt es in Zeile 16 eine Anweisung einen Text auszugeben, damit man weiß, wo das Programm sich gerade in der Verarbeitung befindet. Diese Ausgabe mag während der Entwicklungs- und Testphase sehr hilfreich sein, ist aber bei dem fertigen Programm eher überflüssig. Da man oft nicht weiß, ob die Ausgabe bei der Suche nach später auftretenden Problemen vielleicht noch einmal hilfreich sein kann, möchte man die Anweisung nicht einfach löschen oder auskommentieren. Insbesondere bei umfangreichen Programmen hat man oft eine Vielzahl solcher

die Fehlersuche unterstützenden Anweisungen. Diese im Fehlerfalle alle nach und nach wieder einzufügen oder auszukommentieren, ist einfach nur lästig. Das heißt, ich möchte den Code in Abhängigkeit von der Entwicklungsphase einfügen oder auch nicht. Dieses geschieht in Zeile 15-17. Die fragliche Ausgabeanweisung wird in `#ifdef DEBUG` und `#endif` eingeschlossen. Immer wenn das Makro `DEBUG` definiert ist, wie in Zeile 8, dann bleibt die Ausgabeanweisung in Sourcecode stehen. Gibt es aber die Definition des Makros `DEBUG` nicht, dann wird die Programmzeile vom Präprozessor entfernt. Dabei muss das Makro bei der Definition keinen Wert zugeordnet bekommen.

Weitere Einsatzbereiche dieser bedingten Präprozessoranweisung sind Programmteile, die je nach Betriebssystem unterschiedlich ausfallen. Hierbei wird ausgenutzt, dass je nach Betriebssystem unterschiedliche Makros bereits vordefiniert sind oder über entsprechende Header-Dateien eingefügt werden müssen (siehe Tabelle 2.6).

Tabelle 2.6: Makros, die das Betriebssystem identifizieren

Windows 32 bit + 64 bit	<code>_WIN32</code>	alle Windows OS
Windows 64 bit	<code>_WIN64</code>	nur 64 bit Windows
Apple	<code>__APPLE__</code>	all Apple OS
MacOS	<code>TARGET_OS_MAC</code>	include TargetConditionals.h
Android	<code>__ANDROID__</code>	subset of linux
Unix based OS	<code>__unix__</code>	
Linux	<code>__linux__</code>	subset of unix

2.6 Challenges

2.6.1 Formatierungsoptionen von `cout`

Erstellen Sie ein C++ Programm, welches eine Handelskalkulation nach dem folgenden Beispiel durchführt. Die Bildschirmeingabe und -ausgabe soll dabei wie in Tabelle 2.7 aussehen. Die Ausgabe soll mit 2 Nachkommastellen rechtsbündig formatiert werden. Außerdem soll die Bezeichnung EUR hinter den Beträgen stehen.

Die in Tabelle 2.8 Folgende Variablen sollen zur Eingabe und für die Berechnungen verwendet werden:

Hinweis: Entwickeln Sie das Programm schrittweise. Testen Sie erst die Eingabe, dann schrittweise die Ausgabe mit den Berechnungen. Formatierungen am besten erst am Ende, wenn die Logik stimmt. Verwenden Sie die Manipulatoren aus der Bibliothek `iomanip.h`.

Tabelle 2.7: Handelskalkulation: Ein- und Ausgabe

Eingabe	Ausgabe
Listeneinkaufspreis: 250	Listeneinkaufspreis: 250,00 EUR
Rabatt (in %): 10	Rabatt: 25,00 EUR
Skonto(in %): 2	Zieleinkaufspreis: 225,00 EUR
Bezugskosten: 30	Skonto: 4,50 EUR
Handlungskostensatz (in %): 30	Bareinkaufspreis: 220,50 EUR
Gewinnzuschlag (in %): 5	Bezugskosten: 30,00 EUR
	Bezugspreis: 250,50 EUR
	Handlungskosten: 75,15 EUR
	Selbstkostenpreis: 325,65 EUR
	Gewinn: 16,28 EUR
	Barverkaufspreis: 341,91 EUR
	Mehrwertsteuer (19%): 54,71 EUR
	Bruttoverkaufspreis: 396,62 EUR

Tabelle 2.8: Handelskalkulation: Variablen

Variable	Bedeutung	Verwendung	Typ
listenp	Listenpreis	Eingabe	Float
rabatt	Rabattsatz	Eingabe	Float
skonto	Skontosatz	Eingabe	Float
bk	Bezugskosten	Eingabe	Float
hks	Handlungskostensatz	Eingabe	Float
gkz	Gewinnzuschlag	Eingabe	Float
zek	Zieleinkaufspreis	Berechnung	Float
bek	Bareinkaufspreis	Berechnung	Float
bp	Bezugspreis	Berechnung	Float
hk	Handlungskosten	Berechnung	Float
sk	Selbstkostenpreis	Berechnung	Float
gw	Gewinn	Berechnung	Float
nvk	Barverkaufspreis	Berechnung	Float
mw	Mehrwertsteuer	Berechnung	Float
bvk	Bruttoverkaufspreis	Berechnung	Float

Listings

1.1	Hello World	7
1.2	Hello World unter Verwendung des Namensraumes <code>std</code>	9
1.3	Verwendung von <code>cout</code>	9
1.4	Verwendung von <code>cin</code>	10
2.1	Umgang mit Variablen	15
2.2	Automatische Variablen mit <code>auto</code>	20
2.3	Einfache Präprozessoranweisungen	21