1 Grundlagen

"An meine Kinder:
Macht euch nie darüber lustig, mir mit Computerkram helfen zu müssen.
Ich habe euch beigebracht, wie man einen Löffel benutzt."

– Sue Fitzmaurice

Im ersten Kapitel werden wir die grundlegenden Features von C++ einführen. Wie im gesamten Buch werden wir sie aus verschiedenen Blickwinkeln betrachten, aber nicht versuchen, jedes denkbare Detail herauszuarbeiten (was ohnehin nicht machbar ist). Für detailliertere Fragen zu bestimmten Features empfehlen wir die Online-Handbücher, wie *http://en.cppreference.com* und *http://www.cplusplus.com/*.

1.1 Unser erstes Programm

⇒ c++03/hello42.cpp

Als Einführung in die Sprache C++ sehen wir uns das folgende Beispiel an:

```
#include <iostream>

int main ()
{
    std::cout << "Die ultimative Antwort auf die Frage nach dem Leben,\n"
              << "dem Universum und dem ganzen Rest ist:"
              << std::endl << 6 * 7 << std::endl;
    return 0;
}
```

Dies ergibt laut Douglas Adams [2]:

```
Die ultimative Antwort auf die Frage nach dem Leben,
dem Universum und dem ganzen Rest ist:
42
```

Dieses kurze Beispiel illustriert bereits mehrere Features von C++:

- Ein- und Ausgabe sind nicht Teil der Kernsprache, sondern werden von der Standard-Bibliothek bereitgestellt. Sie müssen explizit *"eingebunden"* werden. Sonst können unsere Programme weder lesen noch schreiben.
- Die Standard-I/O hat ein *"Stream-Modell"* und heißt daher `<iostream>`. Um seine Funktionalität zu nutzen, haben wir sie in der ersten Zeile mit `#include` eingebunden (inkludiert).
- Jedes C++-Programm beginnt mit dem Aufruf der Hauptfunktion `main`. Sie gibt einen ganzzahligen Wert zurück, wobei 0 für ein erfolgreiches Ende steht.

- Geschweifte Klammern (engl. braces) markieren einen *"Anweisungsblock"* – auch zusammengesetzte Anweisung, engl. compound statement, genannt.
- `std::cout` und `std::endl` sind in `<iostream>` definiert. Ersteres ist ein Ausgabestrom, der es uns ermöglicht, Text auf dem Bildschirm zu schreiben. `std::endl` beendet eine Zeile. Wir können eine Zeile auch mit dem Sonderzeichen `\n` beenden.
- Der Operator ≪ kann verwendet werden, um Objekte an einen Ausgabe-Stream wie `std::cout` zu übergeben und somit eine Ausgabeoperation durchzuführen. Bitte beachten Sie, dass der Operator in Programmen mit zwei kleiner-als-Zeichen (`<<`) geschrieben wird. Für ein eleganteres Druckbild verwenden wir stattdessen ein (doppeltes) französisches Guillemet in einem einzigen Symbol.
- `std::` bedeutet, dass der Typ oder die Funktion aus dem Standard-*"Namensraum"* (engl. `namespace`) verwendet wird. Namensräume helfen uns, unsere Namen zu organisieren und mit Namenskonflikten zu umgehen; siehe Abschnitt 3.2.1.
- Zeichenkettenkonstanten (genauer gesagt Literale) werden in doppelte Anführungszeichen gesetzt. Im Buch verwenden wir hauptsächlich den englischen Begriff *"String"*.
- Der Ausdruck `6 * 7` wird ausgewertet und als ganze Zahl an `std::cout` übergeben. In C++ hat jeder Ausdruck einen Typ. Manchmal müssen wir als Programmierer den Typ explizit deklarieren und andere Male kann der Compiler ihn für uns ermitteln. 6 und 7 sind literale Konstanten vom Typ `int` und dementsprechend ist auch ihr Produkt `int`.

Bevor Sie weiterlesen, empfehlen wir Ihnen dringend, dass Sie dieses kleine Programm auf Ihrem Computer übersetzen (kompilieren). Sobald es kompiliert und läuft, können Sie ein wenig damit spielen, z.B. weitere Operationen und Ausgaben hinzufügen. Und gegebenenfalls die Fehlermeldungen betrachten. Schließlich ist der einzige Weg, eine Sprache wirklich zu lernen, sie zu benutzen, selbst wenn am Anfang mehr schiefgeht als klappt. Wenn Sie bereits wissen, wie man einen Compiler oder sogar eine C++-IDE benutzt, können Sie den Rest dieses Abschnitts überspringen.

Linux: Jede Distribution liefert zumindest den GNU C++-Compiler – üblicherweise schon installiert (siehe das kurze Intro in Abschnitt B.1). Wenn wir unser Programm `hello42.cpp` aufrufen wollen, ist dies ganz einfach mit dem Befehl:

```
g++ hello42.cpp
```

Einer obskuren Tradition des letzten Jahrhunderts folgend, wird die daraus resultierende Binärdatei standardmäßig "`a.out`" genannt. Spätestens wenn wir mehrere Programme in einem Verzeichnis haben, werden wir der Binärdatei einen aussagekräftigeren Namen geben wollen:

```
g++ hello42.cpp -o hello42
```

Wir können auch das Build-Tool `make` verwenden, das Standardregeln für die Erstellung von Binärdateien hat (Übersicht in Abschnitt 7.2.2.1). Wir müssen nur:

```
make hello42
```

aufrufen und `make` sucht im aktuellen Verzeichnis nach einer ähnlich benannten Programmquelle. Es wird "`hello42.cpp`" finden und den standardmäßigen C++-Compiler des Systems aufgerufen, da "`.cpp`" eine Standarddateiendung für C++-Quellen ist. Sobald wir unser Programm kompiliert haben, können wir es auf der Kommandozeile als aufrufen:

```
./hello42
```

Unsere Binärdatei kann ohne weitere Software ausgeführt werden, und wir können sie auf ein anderes kompatibles Linux-System[1] kopieren und dort lassen laufen.

Windows: Wenn Sie MinGW nutzen, können Sie Ihre Programme auf die gleiche Weise kompilieren wie unter Linux. Verwenden Sie Visual Studio, müssen Sie zuerst ein Projekt erstellen. Zu Beginn ist es das Einfachste, die Projektvorlage für eine Konsolenanwendung zu nutzen, wie bspw. unter *http://www.cplusplus.com/doc/tutorial/introduction/visualstudio* beschrieben. Wenn Sie das Programm ausführen, haben Sie bei bestimmten Konfigurationen nur ein paar Millisekunden Zeit, um die Ausgabe zu lesen, bevor die Konsole geschlossen wird. Um die Lesezeit auf eine Sekunde zu verlängern, können Sie einfach den nicht-portablen Befehl `Sleep(1000)` einfügen (und `<windows.h>` einbinden). Mit C++11 oder höher kann die Wartephase portabel programmiert werden:

```
std::this_thread::sleep_for(std::chrono::seconds(1));
```

Binden Sie dafür `<chrono>` und `<thread>` ein. Microsoft bietet kostenlose Versionen von Visual Studio namens "Express", die die Standardsprache genau so gut unterstützen wie ihre professionellen Äquivalente. Der Unterschied besteht darin, dass die professionellen Editionen mehr Entwickler-Bibliotheken (SDKs) enthalten. Da diese im Buch nicht verwendet werden, können Sie die "Express"-Version zum Testen unserer Beispiele nutzen.

IDE: Kurze Programme können auch mit einem gewöhnlichen Editor geschrieben werden. Vor allem bei größeren Projekten ist es ratsam, eine *"Integrierte Entwicklungsumgebung"* (engl. *"Integrated Development Environment"* , kurz IDE) zu verwenden. Damit kann man sehen, wo eine Funktion definiert oder verwendet wird, die Dokumentation innerhalb des Codes anzeigen, Namen projektweit suchen oder ersetzen und vieles andere mehr. KDevelop ist eine freie, in C++ geschriebene IDE aus der KDE-Community. Es ist vermutlich die effizienteste IDE unter Linux und sowohl `git` als auch `CMake` sind bereits integriert. Eclipse ist in Java geschrieben und merklich langsamer (soll aber besser geworden sein). Trotzdem sind viele Entwickler damit recht produktiv. Visual Studio ist eine sehr solide IDE, die eine produktive Entwicklung unter Windows ermöglicht und in neueren Versionen auch eine Integration von `CMake`-Projekten bietet. Die produktivste Umgebung für sich zu finden, benötigt etwas Zeit und Experimentiererei. Außerdem es ist natürlich abhängig vom persönlichen und kollaborativen Geschmack, der sich im Laufe der Zeit auch weiterentwickeln kann.

1.2 Variablen

C++ ist eine stark typisierte Sprache, im Gegensatz zu vielen Skriptsprachen. Das bedeutet, dass jede Variable einen Typ hat und sich dieser nie ändert. Eine Variable wird durch eine Anweisung deklariert, die mit dem Typ beginnt, gefolgt von einem Variablennamen und einer optionalen Initialisierung:

[1] Oft ist die Standardbibliothek dynamisch verlinkt (vgl. Abschnitt 7.2.1.4) und dann ist ihr Vorhandensein in der gleichen Version auf dem anderen System Teil der Kompatibilitätsanforderungen.

```
int    i1= 2;              // Ausrichtung nur für Lesbarkeit
int    i2, i3= 5;
float  pi= 3.14159;
double x= -1.5e6;          // -1500000
double y= -1.5e-6;         // -0.0000015
char   c1= 'a', c2= 35;
bool   cmp= i1 < pi,       // -> true
       happy= true;
```

Die beiden Schrägstriche `//` sind der Anfang eines einzeiligen Kommentars, d.h. alles ab den doppelten Schrägstrichen bis zum Ende der Zeile wird ignoriert. Im Prinzip ist das alles, was man über Kommentare wirklich wissen muss. Nichtsdestotrotz werden wir in Abschnitt 1.9.1 noch ein bisschen mehr darüber sagen.

1.2.1 Fundamentale Typen

Die grundlegendsten Typen (engl. intrinsic types) in C++ sind die in Tabelle 1.1 aufgeführten fundamentalen Typen. Sie sind Teil der Kernsprache und immer verfügbar.

Tabelle 1.1 Grundlegende Typen

Name	Semantik
`char`	Buchstabe oder sehr kleine Zahl
`short`	recht kleine ganze Zahl
`int`	reguläre ganze Zahl
`long`	große ganze Zahl
`long long`	sehr große ganze Zahl
`unsigned`	vorzeichenlose Versionen der vorangegangenen
`signed`	vorzeichenbehaftete Versionen der vorangegangenen
`float`	Gleitkommazahl mit einfacher Genauigkeit
`double`	Gleitkommazahl mit doppelter Genauigkeit
`long double`	Gleitkommazahl mit mehr als doppelter Genauigkeit
`bool`	logischer Typ

Die ersten fünf Typen sind ganze Zahlen mit nicht abnehmender Länge. Zum Beispiel ist `int` mindestens so lang wie `short`, d.h. es ist normalerweise länger, aber nicht zwangsläufig. Die genaue Länge jedes Typs ist von der Implementierung abhängig; z.B. kann `int` 16, 32 oder 64 Bit sein. Alle diese Typen können als `signed` (vorzeichenbehaftet) oder `unsigned` (vorzeichenlos) gekennzeichnet werden. Ersteres hat keinen Einfluss auf Integer-Zahlen (außer `char`), da sie standardmäßig `signed` sind.

Wenn wir einen ganzzahligen Typ als `unsigned` deklarieren, haben wir keine negativen Werte, dafür jedoch doppelt so viele positive (plus eins, wenn wir Null als weder positiv noch negativ betrachten). `signed` und `unsigned` können als Adjektive für das Substantiv `int` betrachtet werden, wenn das Adjektiv allein verwendet wird. Gleiches gilt auch für die Adjektive `short`, `long` und `long long`.

Der Typ `char` kann auf zwei Arten verwendet werden: für Buchstaben und recht kleine Zahlen. Abgesehen von wirklich exotischen Architekturen hat der Typ fast immer eine Länge von 8 Bit. So können wir entweder Werte von -128 bis 127 (`signed`) oder von 0 bis 255 (`unsigned`) darstellen und alle numerischen Operationen mit ihnen durchführen, die für ganze Zahlen verfügbar sind. Wenn weder `signed` noch `unsigned` deklariert wird, hängt es von der Implementierung des Compilers ab, was verwendet wird. Die Verwendung von `char` oder `unsigned char` für kleine Zahlen kann jedoch nützlich sein, wenn es sehr viele davon gibt.

Logische Werte werden am besten als `bool` dargestellt. Eine boolsche Variable kann die Werte `true` und `false` annehmen.

Die Eigenschaft der nicht abnehmenden Länge gilt in gleicher Weise für Gleitkommazahlen: `float` ist kürzer oder gleich lang wie `double`, was wiederum kürzer oder gleich lang wie `long double` ist. Typische Größen sind 32 Bit für `float`, 64 Bit für `double` und 80 Bit für `long double`.

1.2.2 Characters und Strings

Wie bereits erwähnt, kann der Typ `char` verwendet werden, um Zeichen zu speichern:

```
char c= 'f';
```

Wir können auch jeden Buchstaben darstellen, dessen Code in 8 Bit passt. Es kann sogar mit Zahlen gemischt werden; z.B. führt `'a' + 7` in der Regel zu `'h'`, abhängig von der zugrundeliegenden Kodierung der Buchstaben. Wir raten dringend davon ab, damit zu spielen, da die mögliche Verwirrung wahrscheinlich zu unnötiger Zeitverschwendung führt.

Von C haben wir die Möglichkeit geerbt, Zeichenketten als Arrays von `char` darzustellen.

```
char name[9]= "Herbert";
```

Diese alten C-Strings enden alle mit einer binären `0` als `char`-Wert. Fehlt die `0`, laufen die Algorithmen bis zum nächsten Speicherplatz mit einem `0`-Byte weiter. Eine andere große Gefahr besteht beim Anhängen von Zeichenketten: `name` hat keinen zusätzlichen Platz, zusätzliche Zeichen überschreiben irgendwelche andere Daten. Alle String-Operationen richtig zu implementieren, ohne den Speicher zu beschädigen oder längere Strings abzuschneiden, ist bei diesen alten Zeichenketten alles andere als trivial. Wir empfehlen daher dringend, sie nur für literale Werte zu verwenden.

Der C++-Compiler unterscheidet zwischen einfachen und doppelten Anführungszeichen: `'a'` ist das Zeichen "a" (es hat den Typ `char`) und `"a"` ist ein Array mit einer binären `0` als Abschluss (d.h. sein Typ ist `const char[2]`).

Im Gegensatz dazu erlaubt die Klasse `string` aus der Bibliothek `<string>` einen viel einfacheren und zugleich sichereren Umgang mit Zeichenketten:

```
#include <string>

int main()
{
    std::string name= "Herbert";
}
```

C++-Strings verwenden dynamischen Speicher und verwalten ihn selbst. Wenn wir also mehr Text an einen String anhängen, müssen wir uns keine Sorgen über Speicherzugriffsfehler oder das Abschneiden von Strings machen:

```
name= name + ", unser cooler Antiheld"; // Mehr dazu später
```

Viele aktuelle Implementierungen verwenden auch eine Optimierung für kurze Strings (z.B. bis 16 Byte), die direkt im `string`-Objekt gespeichert werden statt zusätzlichen Speicher anzufordern. Diese Optimierung kann die aufwendige Speicherallokation und -freigabe deutlich reduzieren.

C++14 Da Text in doppelten Anführungszeichen als `char`-Array interpretiert wird, benötigen wir die Möglichkeit, einen Text als String zu bezeichnen. Dies geschieht mit dem Nachsatz `s`, z.B. `"Herbert"s`. Leider hat es bis C++14 gedauert, um dies zu ermöglichen. Eine explizite Konvertierung wie `string("Herbert")` war schon immer möglich. In C++17 wurde eine leichtgewichtige, konstante Sicht auf Strings hinzugefügt, die wir in Abschnitt 4.4.5 vorstellen werden.

1.2.3 Variablen deklarieren

Deklarieren Sie Variablen so spät wie möglich, in der Regel direkt vor der ersten Verwendung und wenn möglich nicht bevor Sie sie initialisieren können.

Dies macht Programme besser lesbar, wenn sie lang werden. Es erlaubt dem Compiler auch, den Speicher mit verschachtelten Bereichen effizienter zu nutzen.

C++11 Wir können auch den Compiler den Typ einer Variablen für uns ermitteln lassen, z.B:

```
auto i4= i3 + 7;
```

Der Typ von `i4` ist der gleiche wie der von `i3 + 7`, nämlich `int`. Auch automatisch ermittelte Typen ändern sich nicht wieder, und was auch immer `i4` noch zugewiesen wird, wird in `int` konvertiert. Wir werden später sehen, wie nützlich `auto` in der fortgeschrittenen Programmierung ist. Für einfache Variablendeklarationen (wie die in diesem Abschnitt) ist es normalerweise besser, den Typ explizit zu deklarieren. `auto` wird in Abschnitt 3.4 ausführlich behandelt.

1.2.4 Konstanten

Syntaktisch gesehen sind Konstanten wie spezielle Variablen in C++ mit dem zusätzlichen Attribut `const`:

```
const int    ci1= 2;
const int    ci3;              // Fehler: kein Wert
const float  pi= 3.14159;
const char   cc= 'a';
const bool   cmp= ci1 < pi;
```

Da diese Objekte nicht geändert werden können, ist es zwingend erforderlich, ihre Werte in der Deklaration festzulegen. Die zweite Konstantendeklaration verletzt diese Regel, und der Compiler wird ein solches Fehlverhalten nicht tolerieren.

Konstanten können überall dort verwendet werden, wo Variablen erlaubt sind, solange sie nicht verändert werden. Andererseits sind Konstanten wie im obigen Beispiel normalerweise bereits während der Kompilierung bekannt. Dies ermöglicht viele Arten von Optimierungen, und die Konstanten können sogar als Argumente von Typen verwendet werden (wir werden darauf später in Abschnitt 5.1.4 zurückkommen).

1.2.5 Literale

Literale wie `2` oder `3.14` werden ebenfalls typisiert. Einfach ausgedrückt, werden ganze Zahlen je nach Anzahl der Stellen als `int`, `long` oder `unsigned long` behandelt. Jede Zahl mit einem Punkt oder einem Exponenten (z.B. `3e12` $\equiv 3 \cdot 10^{12}$) wird als `double` angesehen.

Literale anderer Typen erhalten wir, wenn wir einer Endung aus Tabelle 1.2 hinzufügen:

Tabelle 1.2 Typen von Literalen

Literal	Typ
`2`	`int`
`2u`	`unsigned`
`2l`	`long`
`2ul`	`unsigned long`
`2.0`	`double`
`2.0f`	`float`
`2.0l`	`long double`

In den meisten Fällen ist es nicht notwendig, die Typen der Literale explizit zu deklarieren, da die implizite Konvertierung in gemischten Ausdrücken (engl. coercion) zwischen fundamentalen numerischen Typen normalerweise unseren Erwartungen entspricht.

Es gibt jedoch drei Gründe, warum wir auf den Typ der Literale achten sollten:

Verfügbarkeit: Wenn wir eine Funktion aufrufen wollen, die es für `int` bzw. `double` nicht gibt und der Wert auch nicht implizit in einen Typen, für den es eine Funktion gibt, umgewandelt werden kann. Beispielsweise stellt die Standardbibliothek eine Klasse für komplexe Zahlen zur Verfügung, bei dem der Typ für Real- und Imaginärteil vom Anwender parametrisiert werden kann:

```
std::complex<float> z(1.3, 2.4), z2;
```

Leider werden Operationen nur zwischen dem Typ selbst und dem zugrundeliegenden realen Typ angeboten (und Argumente werden hier nicht konvertiert).[2] Folglich können wir `z` nicht mit einem `int` oder `double`, sondern nur mit `float` multiplizieren:

```
z2= 2 * z;       // Fehler: kein int * complex<float>
z2= 2.0 * z;     // Fehler: kein double * complex<float>
z2= 2.0f * z;    // Okay:  float * complex<float>
```

[2] Gemischte Arithmetik ist jedoch realisierbar, wie in [16] demonstriert wurde.

Mehrdeutigkeit: Wenn eine Funktion für verschiedene Argumenttypen überladen ist (Abschnitt 1.5.4), kann ein Argument wie `0` mehrdeutig sein, während für ein qualifiziertes Argument wie `0u` eine eindeutige Überladung existieren kann.

Genauigkeit: Das Problem der Genauigkeit tritt auf, wenn wir mit `long double` arbeiten. Da ein nicht-qualifiziertes Literal den Typ `double` hat, können wir Ziffern verlieren, bevor wir es einer `long double`-Variable zuweisen:

```
long double third1= 0.3333333333333333333;  // könnte Ziffern verlieren
long double third2= 0.3333333333333333333l; // exakt
```

Ganzzahlige Literale, die mit einer Null beginnen, werden als Oktalzahlen interpretiert, z.B:

```
int o1= 042;          // int o1= 34;
int o2= 084;          // Fehler: keine 8 und 9 in Oktalzahlen
```

Hexadezimale Literale können durch Voranstellen von `0x` oder `0X` geschrieben werden:

```
int h1= 0x42;         // int h1= 66;
int h2= 0xfa;         // int h2= 250;
```

C++14 Mit dem Präfix `0b` or `0B` können wir nun auch binäre Literale schreiben:

```
int b1= 0b11111010;  // int b1= 250;
```

C++14 Um lange Literale lesbarer zu gestalten, können wir die Ziffern durch Apostrophe trennen:

```
long               d=   6'546'687'616'861'129l;
unsigned long      ulx= 0x139'ae3b'2ab0'94f3;
int                b=   0b101'1001'0011'1010'1101'1010'0001;
const long double pi=   3.141'592'653'589'793'238'462l;
```

C++17 Gleitkomma-Literale können jetzt auch hexadezimal geschrieben werden:

```
auto f1= 0x10.1p0f; // 16.0625
auto d2= 0x1ffp10;  // 523264
```

Für sie ist der Exponent obligatorisch – daher brauchten wir im ersten Beispiel `p0`. Durch das Suffix `f` ist `f1` ein `float`, der den Wert $16^1 + 16^{-1} = 16.0625$ speichert. Diese Literale verwenden drei Basen: Die Pseudo-Mantisse ist hexadezimal, skaliert mit Potenzen zur Basis 2, wobei der Exponent als Dezimalzahl angegeben wird. Somit hat `d2` den Wert $511 \times 2^{10} = 523264$. Hexadezimale Literale wirken anfangs zwar etwas seltsam, aber sie erlauben es uns, binäre Gleitkommazahlen ohne Rundungsfehler zu deklarieren.

String-Literale werden als Arrays von `char` geschrieben:

```
char s1[]= "Alter C-Stil"; // lieber nicht
```

Allerdings sind diese Arrays alles andere als nutzerfreundlich und wir fahren besser mit `string`-Objekten, die direkt mit einem String-Literal initialisiert werden können:

```
#include <string>

std::string s2= "In C++ lieber so.";
```

Sehr langer Text kann in mehrere Teilstrings aufgeteilt werden:

```
std::string s3= "Das ist langer und langatmiger Text, "
                "der nicht ganz auf eine Zeile passt.";
```

Für weitere Details zu Literalen siehe [49, §6.2].

1.2.6 Werterhaltende Initialisierung

C++11

Angenommen, wir initialisieren eine `long`-Variable mit einer großen Zahl:

```
long l2= 1234567890123;
```

Dies kompiliert problemlos und funktioniert korrekt – wenn `long` wie auf den meisten 64-Bit-Plattformen 64 Bit nutzt. Wenn `long` nur 32 Bit lang ist (z.B. in Visual Studio oder beim Kompilieren mit dem Flag `-m32`), ist der obige Wert zu lang. Das Programm kompiliert aber trotzdem (eventuell mit einer Warnung) und läuft mit einem anderen Wert, z.B. indem die führenden Bits abgeschnitten werden.

C++11 führt eine *"werterhaltende"* Initialisierung ein, bei der die Genauigkeit nicht verloren geht oder wie es der Standard nennt, dass die Werte nicht *"verengt"* werden; siehe Abschnitt C.2.4 für die genaue Definition. Dies wird mit der vereinheitlichten Initialisierung erreicht, die wir hier einführen möchten und in Abschnitt 2.3.4 weiter vertiefen werden. Werte in geschweiften Klammern dürfen nicht verengt werden:

```
long l= {1234567890123};
```

Nun prüft der Compiler, ob die Variable `l` den Wert auf der Zielplattform speichern kann. Diese Kontrolle des Compilers bewahrt uns auch vor Genauigkeitsverlusten bei der Initialisierung. Eine gewöhnliche Initialisierung eines `int` durch eine Gleitkommazahl ist dagegen aufgrund der impliziten Konvertierung erlaubt:

```
int i1= 3.14;        // kompiliert trotz Verengung (unser Risiko)
int i1n= {3.14};     // Verengungsfehler: nach  Komma abgeschnitten
```

Die neue Initialisierungsform in der zweiten Zeile verbietet dies, da sie den Nachkommateil der Gleitkommazahl abschneiden würde. Ebenso wird die Zuweisung negativer Werte an vorzeichenlose Variablen oder Konstanten bei der traditionellen Initialisierung toleriert, aber in der neuen Form verworfen:

```
unsigned u2= -3;     // kompiliert trotz Verengung (unser Risiko)
unsigned u2n= {-3};  // Fehler durch Verengung: keine negativen Werte
```

In den vorherigen Beispielen haben wir literale Werte in den Initialisierungen verwendet und der Compiler prüft, ob ein bestimmter Wert mit diesem Typ darstellbar ist:

```
float f1= {3.14};    // okay
```

Nun, der Wert 3.14 kann in keinem binären Fließkommaformat absolut genau dargestellt werden, aber der Compiler kann `f1` auf den Wert setzen, der 3.14 am nächsten kommt. Wenn eine `float`-Variable mit einer `double`-Variablen (keinem Literal oder Konstante) initialisiert wird, müssen alle möglichen `double`-Werte berücksichtigt werden und ob sie alle verlustfrei in `float` konvertierbar sind:

```
double d;
...
float f2= {d};       // Fehler durch Verengung
```

Beachten Sie, dass die Verengung zwischen zwei Typen wechselseitig sein kann:

```
unsigned u3= {3};
int      i2= {2};

unsigned u4= {i2};   // Fehler: keine negativen Werte
int      i3= {u3};   // Fehler: nicht alle großen Werte
```

Die Typen `signed int` und `unsigned int` haben die gleiche Größe, aber nicht alle Werte des einen Typs sind im jeweils anderen darstellbar.

1.2.7 Gültigkeitsbereiche

Gültigkeitsbereiche (engl. Scopes) bestimmen die Lebensdauer und Sichtbarkeit von (nichtstatischen) Variablen und Konstanten und etablieren eine Struktur in unseren Programmen.

1.2.7.1 Globale Definitionen

Jede Variable, die wir in einem Programm verwenden wollen, muss mit ihrer Typangabe zuvor im Code deklariert worden sein. Eine Variable kann sowohl im globalen als auch im lokalen Bereich liegen. Eine globale Variable wird außerhalb aller Funktionen deklariert. Nach ihrer Deklaration können globale Variablen von überall im Code genutzt werden, auch innerhalb von Funktionen. Das klingt zunächst sehr praktisch, weil die Variablen leicht verfügbar sind, aber wenn unsere Software wächst, wird es schwieriger und schmerzhafter, alle Änderungen von globalen Variablen nachzuvollziehen. Jede Code-Änderung birgt irgendwann das Potenzial, eine ganze Lawine von Fehlern auszulösen.

Verwenden Sie keine globalen Variablen!

Sie können im gesamten Programm verwendet werden und daher ist es extrem mühsam, den Überblick über ihre Verwendung zu wahren.

Globale Konstanten wie:

```
const double pi= 3.14159265358979323846264338327950288419716939;
```

sind in Ordnung, da sie keine Seiteneffekte verursachen können.

1.2.7.2 Lokale Definitionen

Eine lokale Variable wird innerhalb des Körpers einer Funktion deklariert. Ihre Sicht- und Verfügbarkeit ist auf den in `{ }` eingeschlossenen Block ihrer Deklaration beschränkt. Genauer gesagt, beginnt der Scope einer Variablen mit ihrer Deklaration und endet mit der schließenden Klammer des Deklarationsblocks.

Wenn wir `pi` in der Funktion `main` definieren: