Teil II

Die Programmiersprache C++

Lernziele

- Entstehung von C++ *kennen*
- Aufbau eines Programms in C++ *kennen*
- Einfache Programme mit Ein- und Ausgabe über <iostream> ohne Kontrollstrukturen *schreiben können*
- Variablen vom Typ double anlegen und verwenden können
- Einfache mathematische Operatoren (+ - * /) sowie den Zuweisungsoperator '=' in eigenen Programmen anwenden können

Geschichtlicher Hintergrund:

- C++ entstand ca. 1980 als „C with classes“[16] von Bjarne Stroustrup
- Erste ISO Standardisierung 1998 (International Standards Organisation)
- Nationale Standards (ANSI und DIN) haben sich dem angeschlossen
- Ursprüngliche Version war nahe an C, mit allen Vor- und Nachteilen
- Kontinuierliche Verbesserungen, sodass es 2003, 2011 (C++11), 2014 (C++14), 2017 (C++17), neue Standards gab (C++20, C++23/26 in Planung)
- `http://boost.org` stellt einen Pool an Code, der zur Aufnahme in neue Standards diskutiert wird
- Wir orientieren uns an C++14, da diese mittlerweile einheitlich verfügbar ist und viele neue Konzepte einführt.

[16] classes (engl.) = Klassen, i.d.R. Objekte

Obwohl C++ standardisiert ist, ist

- C++ ist eine Sprache mit wenigen Vorgaben
  $\Rightarrow$ Es ist einfach, schlechten Code zu schreiben!
- Es besteht jedoch immer mehr der Konsens, was „schöner“ C++ Code ist
- September 2015: Vorstellung von Guidelines[17] zu schönem C++-Code, inklusive von Hinweisen, wie der Code verbessert werden kann (Ziel: Integration in die Compiler)

[17] https://isocpp.org/blog/2015/09/bjarne-stroustrup-announces-cpp-core-guidelines

Weiterführende Literatur

Ergänzende Literatur zur Vorlesung:

- `http://cppreference.com`
- C++ Die Programmiersprache (Bjarne Stroustroup)
- `http://www.cprogramming.com/tutorial/c++-tutorial.html`
- `http://www.tutorialspoint.com/cplusplus/cpp_if_else_statement.htm`

Bücher:

- Der C++ Programmierer, 4. Auflage(!) Hanser-Verlag
- Nachschlagewerk: Die C++ Programmiersprache, Bjarne Stroustrup, z.B.: Hanser Verlag

Für Fortgeschrittene:

- C++ Guidelines: `https://github.com/isocpp/CppCoreGuidelines/`

C++-Entwicklungsumgebungen I

- Visual C++ (Windows)
  `http://www.microsoft.com/germany/express/default.aspx`
- Code::Blocks (Windows, Linux, Mac)
  `codeblocks-16.01-setup.exe` von
  `http://www.codeblocks.org`
- Dev-C++ (Windows)
  `devcpp-4.9.9.2_setup.exe` von
  `http://www.bloodshed.net`
- Eclipse-Plattform (Windows, Linux, Mac)
  Enwicklungsumgebung von `http://www.eclipse.org`

C++-Entwicklungsumgebungen II

- Android: CppDroid
  siehe: Play Store.
- Compiler: g++-Compiler
  - Windows gcc-Compiler von `http://www.mingw.org` Installation ist mit Aufwand verbunden!
  - Mac: über homebrew oder macports verfügbar (jedoch nicht empfohlen)
  - Linux: In der Regel vorinstalliert, über Paket installierbar
- clang++-Compiler
  - `http://clang.llvm.org/get_started.html`
  - Mac: Teil der Command line tools von XCode (Apple LLVM version 8.1.0 (clang-802.0.42))
  - Linux: In der Regel als Paket (clang) installierbar
  - Windows: `http://llvm.org/releases/download.html`

C++-Entwicklungsumgebungen III

- Microsoft Visual C++: Windows: als Teil von Visual Studio

Quelltext

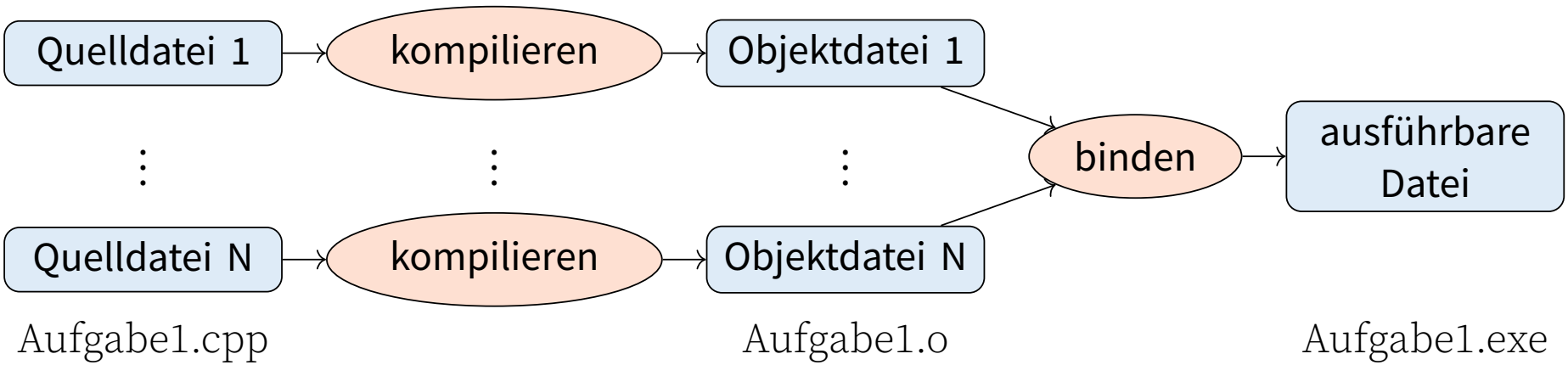
Unter **Quelltext** oder **Programmquelltext** versteht man eine textuelle Repräsentation eines Programms in einer bestimmten Programmiersprache.
C++ basiert auf eine Beschreibung mittels Quelltext. Dementsprechend können C++-Programme mit jedem Texteditor eingegeben werden.[18]

[18] Microsoft Word kann zwar auch Dateien als Text abspeichern, ist jedoch für die Programmierung eher ungeeignet. Mehr dazu in 1

Das einfachste C++ Programm

```
1  int main() { } // Das minimale C++ Programm
```

Programmquelltexte sind Textdateien, die kompiliert (übersetzt) und gelinkt (gebunden, zusammengefügt) werden müssen.



Compiler

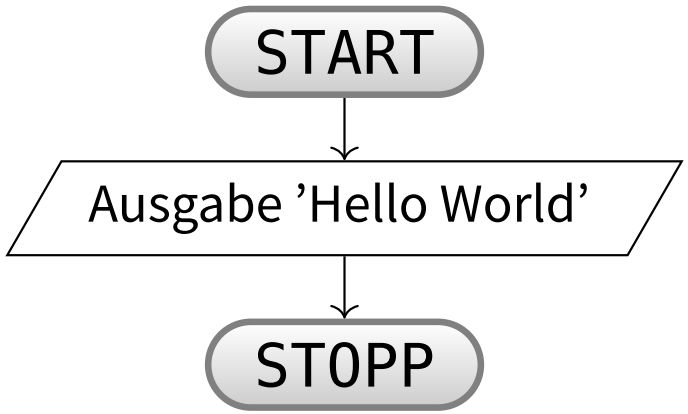
GNU C++ Compiler

```
# 1) erzeugt die Objektdatei programm.o
g++ -Wall -std=c++14 -c programm.cpp
# 2) erzeugt programm.exe aus der angegebenen Objektdatei
g++ -Wall -std=c++14 -o programm.exe programm.o

# Fuer einfache Programme reicht auch der eine Schritt für Kompilieren und Binden
g++ -Wall -std=c++14 -o programm.exe programm.cpp
```

clang C++ Compiler

```
# 1) erzeugt die Objektdatei programm.o
clang++ -Weverything -std=c++14 -c programm.cpp
# 2) erzeugt programm.exe aus der angegebenen Objektdatei
clang++ -Weverything -std=c++14 -o programm.exe programm.o

# Fuer einfache Programme reicht auch der eine Schritt für Kompilieren und Binden
clang++ -Weverything -std=c++14 -o programm.exe programm.cpp
```

Das Programm „Hallo Welt“



Das Programm „Hello World“

```
1  // einbinden externer Definitionen (hier std::cout)
2  #include <iostream>
3
4  int main()
5  { // Start des Programms
6
7      std::cout << "Hello World!\n";
8
9      return 0; // Rückgabe an das aufrufende Programm
10 } // Ende des Programms
```

Alles, was in einer Zeile *//* folgt ist ein Kommentar. Kommentare werden nicht interpretiert, sondern vom Computer überlesen. Eine andere Art Kommentare zu schrieben ist, den Kommentar mit */* Kommentar */* zu klammern. Kommentare werden verwendet um:

- Das Programm zu dokumentieren
- Den Gedankengang zum Quelltext zu kommentieren
- Hinweise an den Leser des Quelltextes zu geben

Selten auch um

- den Quelltext zu strukturieren
- Bilder als ASCII-Grafik zu zeichnen, um Dinge zu erklären

Und im Rahmen dieser Vorlesung

- Quelltextzeilen für Einsteiger in lesbarer Form zu übersetzen

Quelltexte bestehen aus verschiedenen Teilen, die vom **Compiler** von oben nach unten **gelesen** werden.
Die Reihenfolge im Programm ist dabei nicht fest, man hat sich aber auf bestimmte Vorgehensweisen geeinigt :

Einbinden von Bibliotheken `#include <iostream>`
: Einbinden anderer Bibliotheken, hier einer Bibliothek zur Ausgabe auf der Konsole.
Bibliotheken stellen verschiedene vorgefertigte Problemlösungsstrategien bereit, die im eigenen Programm verwendet werden können.

Globaler Teil
: Hier können globale, das heißt überall verwendbare Programmteile stehen (mehr später)

Hauptprogramm `int main()`
: Die Funktion `main` ist der **Einstiegspunkt** in ein Programm. Hier beginnt die Abarbeitung der Befehle bei der Ausführung.

Der Funktionsrumpf

```
1 {
2     // ...
3 }
```

Dieser beschreibt die Befehle, die beim Programmstart „sequenziell“ von oben nach unten abgearbeitet werden. Kontrollstrukturen (mehr dazu später) erlauben es, diese Abarbeitungsreihenfolge zu ändern.

[Wertrückgabe (optional)] **return** 0;

Springt aus ihrem Programmteil heraus und beendet diesen. Hier wird der Wert „0“ an das Betriebssystem zurückgegeben (Bedeutung: Programm wurde korrekt ausgeführt). Die Wertrückgabe im Hauptprogramm ist Optional.

Das Programm „Addition“

```cpp
#include <iostream>

int main()
{   // hier beginnt der Programmablauf

    // reserviere Speicher für eine Ganzzahl, nenne diese a
    int a;

    // lies eine Zahl von der Konsole ein und speichere das Ergebnis in a
    std::cin >> a;

    // reserviere Speicher für eine Ganzzahl, nenne diese b
    int b;

    // lies eine Zahl von der Konsole ein und speichere das Ergebnis in b
    std::cin >> b;

    // reserviere Speicher für eine Ganzzahl, nenne diesen c
    // berechne die Summe aus a und b und
    // weise das Ergebnis der Rechnung dem Speicher, den wir c genannt haben zu.
    int c = a + b;

    // gib die Zeichenfolge "Die Summe ist ",
    // gefolgt von dem Wert der Zahl, die in dem Speicher c steht,
    // gefolgt von einem Punkt,
    // gefolgt von einem Zeilenendezeichen aus und wartet bis alles angezeigt wurde.
    std::cout << "Die Summe ist " << c << "." << std::endl;

    // (Optional) bestätigt dem aufrufenden Programm den erfolgreichen Abschluss
    // und beendet die Ausführung des Programms
    return 0;  // hiermit endet der Programmablauf
}
```

Compiler habe verschiedene Optionen, mit denen der Kompiliervorgang beeinflusst werden kann. Eine Option ist hierbei zusätzliche Warnungen einzuschalten.

Verwendung des Compilers

```
# GNU C++ Compiler (g++)
-pedantic -Wall -Wextra -Weffc++ -Wcast-align -Wcast-qual -Wctor-dtor-privacy
-Wdisabled-optimization -Wformat=2 -Winit-self -Wlogical-op -Wmissing-declarations
-Wmissing-include-dirs -Wnoexcept -Wold-style-cast -Woverloaded-virtual
-Wredundant-decls -Wshadow -Wsign-conversion -Wsign-promo -Wstrict-null-sentinel
-Wstrict-overflow=5 -Wswitch-default -Wundef -Werror -Wno-unused

# clang++
-Weverything
```

Warnungen sind häufig sinnvoll, um Programmierfehler im Vorfeld ausschließen zu können.

Quadrieren einer Fließkommazahl (quadrat.cpp)

```cpp
#include <iostream>

int main()
{
    std::cout << "Geben Sie eine Zahl ein: ";

    double zahl;
    std::cin >> zahl;

    double quadrat;
    quadrat = zahl * zahl;

    std::cout << "Das Quadrat der Zahl ist: " << quadrat << '\n';
}
```

Kompilieren g++

```
# Übersetzen mit:
g++ -Wall quadrat.cpp -o quadrat
# Ausführen mit:
./quadrat
```

Kompilieren clang++

```
# Übersetzen mit:
clang++ -Weverything quadrat.cpp -o quadrat
# Ausführen mit:
./quadrat
```

Symbole

Anhand der **Symbole** prüft der Rechner die formale Korrektheit eines Programms und setzt es in Maschinensprache um. Zu den Symbolen zählen

- Schlüsselwörter
- Bezeichner
- Literale
- Operatoren
- Kommentare
- Separatoren („Whitespace“ (Zwischenräume), Klammern)

Schlüsselwörter

Schlüsselwörter sind in einer Programmiersprache häufig **reservierte Wörter**[19], die eine fest vorgegebene Bedeutung haben.
Bisher kennen Sie die Schlüsselwörter in C++: **int**, **double**, **return**

[19] Reservierte Wörter sind Wörter, die vom Programmierer nicht neu definiert werden können. Sie haben immer die von der Sprache vorgegebene, gleiche Bedeutung im Programm.

Variablen

Variablen sind Repräsentanten für Speicherbereiche. In diesen können Daten abgelegt werden. C++ fordert, dass jeder Speicherbereich einen speziellen Typ an Daten beinhalten (z. B. Zahlen, Buchstaben, etc.). Die **Werte** in den Speicherbereichen sind variabel, das heißt, sie können sich über die Ausführung des Programms ändern. Wir sprechen von dem **Wert** einer Variablen, wenn wir das darin gespeicherte meinen und von dem **Bezeichener der Variablen**, wenn wir den Namen der Variablen im Programm meinen.

Bezeichner

Bezeichner identifizieren Speicherbereiche, die Datenobjekte beinhalten.
Bezeichner sind „symbolische Namen“ für Variablen, für benutzerdefinierte Datentypen, Funktionen, Klassen und Objekte.
Bezeichner folgen bestimmten Regeln, z. B. sind in C++ Schlüsselworte als Bezeichner verboten.

Bezeichner

```
1  int a;                // definiert a als Bezeichner für eine Variable vom Typ int
2  double meineZahl;     // definiert meineZahl als Bezeichner für eine Variable vom Typ double
3  std::string meinName; // definiert meinName als Bezeichner für eine Variable vom Typ
       Zeichenkette
```

Literale

Literale sind konstante Größen in einem Programm. Sie beschreiben die konkrete Werte für einen Datentyp.

- Ganzzahlkonstante, reele Zahlenkonstante, …
- Zeichenkettenkonstante
- benutzerdefinierte Konstante

Beispiele sind:

Beispiele für Literale

```
1  42                  // Ganzzahlkonstante (Typ int)
2  -42                 // Ganzzahlkonstante (Typ int)
3  1e-10               // Fließkommakonstante (Typ double)
4  123.456             // Fließkommakonstante (Typ double)
5  'b'                 // Buchstabenkonstante (Typ char)
6  "Mario Hlawitschka" // Zeichenkettenkonstante (Typ char*)
7  "Ende\r\n"          // Zeichenkettenkonstante mit Steuerzeichen
```

Anweisungen

Eine **Anweisung** in C++ kann unter anderem sein

- eine Deklarationsanweisung
- eine Ausdrucksanweisung
- eine Schleifenanweisung
- eine Auswahlanweisung
- eine Verbundsanweisung (auch Block genannt).

Eine **Deklarationsanweisung** führt einen Namen in das Programm ein. Nach der Deklaration ist das Objekt im Programm bekannt und kann genutzt werden.

Deklarationsanweisung

```
1  int c;
```

Strukturiertes Programmieren

Beim Programmieren ist Übersichtlichkeit wichtig. Die gelernten Kontrollstrukturen dienen dazu, den Code zu strukturieren.
Die Befehle eines Programms liegen immer in einer bestimmten Reihenfolge im Speicher. Kontrollstrukturen ermöglichen, diese Reihenfolge zu durchbrechen und darin „umherzuspringen“. Zur besseren Nachvollziehbarkeit wollen wir jedoch beliebige Sprünge vermeiden und lassen daher nur bestimmte Sprünge zu.

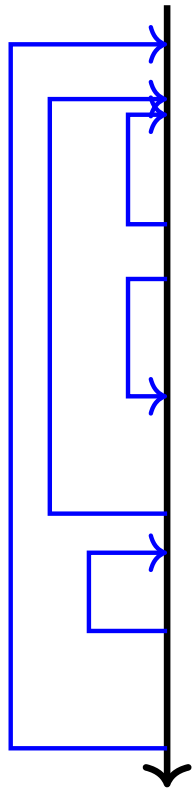


Abb. Strukturierter Code mit gut erkennbaren Programmteilen

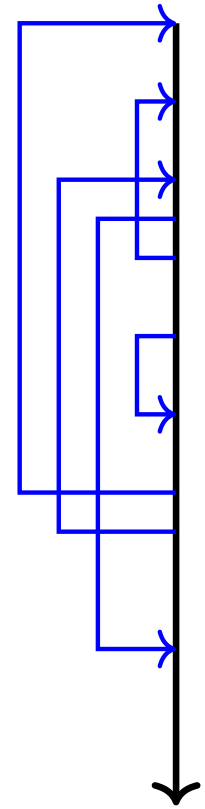


Abb. Unstrukturierter Code, keine offensichtliche Systematik

Variablen

Variablen sind Platzhalter im Speicher des Rechners („Merkzettel“) in denen Information gespeichert werden kann. C++ ist eine streng typisierte Sprache, das heißt, dass jeder Variablen neben einem Namen und ein eindeutiger Datentyp zuzuweisen ist.
Ein Datentyp definiert

- die Größe der Daten
  (das heißt die Menge des verwendeten Speichers)
- die Art der Speicherung
  (das heißt das exakte Bitmuster eines Datenwertes im Speicher).

Bezeichner

Ein **Bezeichner** ist der Name einer Variablen, wie sie im Programm benannt wird. Der Name dient dazu, eindeutig auf den Speicher zuzugreifen um somit den Inhalt der Variablen lesen oder verändern zu können.
Für die Benennung von Variablen gibt es in den meisten Programmiersprachen Einschränkungen. Für C++ gelten diese Einschränkungen für alle Namen, die im Programm vorkommen.

Variablennamen: Bezeichner in C++

Definition der Syntax über Backus-Naur-Form wp

- <Buchstabe> ::= a-z,A-Z
- <Zeichen> ::= <Buchstabe> | _
- <Ziffer> ::= 0–9
- <Bezeichner> ::= <Zeichen>
  <Bezeichner> ::= <Bezeichner> <Ziffer>
  <Bezeichner> ::= <Bezeichner> <Zeichen>

Umlaute sind in C++ nicht vorgesehen.

Variablendeklaration und Initialisierung

Variablendeklaration und Initialisierung

```
1  // http://en.cppreference.com/w/cpp/language/value_initialization
2
3  double d;           // reserviert den Speicher für einen double-Wert
4                      // der Speicher ist uninitialisiert!
5
6  double e = 1.0;     // reserviert den Speicher für einen double-Wert
7                      // und initialisiert ihn mit 1
8
9  double f( 1.0 );    // reserviert den Speicher für einen double-Wert
10                     // und initialisiert ihn mit 1
11
12 double g{ 1.0 };    // reserviert den Speicher für einen double-Wert
13                     // und initialisiert ihn mit 1 (C++11)
14
15 double h{ g };      // reserviert den Speicher für einen double-Wert
16                     // und initialisiert ihn mit dem Wert von g
17
18 double i = double();// reserviert den Speicher für einen double-Wert
19                     // und initialisiert ihn mit dem Wert 0.0 (C++11)
20
21 double j{};         // reserviert den Speicher für einen double-Wert
22                     // und initialisiert ihn mit dem Wert 0.0 (C++11)
```

Variablendeklaration: Konstanten

Alle Variablen können als konstant markiert werden.

Variablendeklaration von Konstanten

```
1  double d{ 1.0 }; // definiert eine Variable namens d vom Typ double
2                   // und initialisiert diese mit dem Wert 1,0
3  d = 2.0;         // weist d den neuen Wert 2,0 zu
4
5  constexpr double e{ 0.5/3.1 }; // initialisiert einen konstanten Audsruck
6                   // vom Typ double und setzt diesen auf den Wert von 0,5/3,1.
7                   // e ist auf double-Präzision gerundet!
8                   // e kann nicht modifiziert werden
9  e = 2.0;         // Fehler! Zuweisung an constexpr-lvalue nicht erlaubt
10
11 double const f{ 0.5/3.1 };
12 const double f{ 0.5/3.1 }; // initialisiert einen konstante Variable
13                  // vom Typ double und setzt diesen auf den Wert von 0,5/3,1.
14                  // f ist auf double-Präzision gerundet!
15                  // f kann nicht modifiziert werden
16 f = 2.0;         // Fehler! Zuweisung an const-lvalue nicht erlaubt
17
18 constexpr double g{ e }; // ok, da e constexpr
19 constexpr double h{ f }; // ok, da f aus Literal ableitbar
20 constexpr double i{ d }; // Fehler! d ist keine constexpr
```

Konstanten sind Variablen, die nur zur Initialisierung einen Wert zugewisen

Das Schlüsselwort **const** gibt an, dass sich der Wert einer Variablen nach der Zuweisung nicht mehr ändert.
Das Schlüsselwort **constexpr** gibt an, dass der initiale Wert der Variablen schon zur Zeit des Compilierens feststeht.

const bezieht sich immer auf das, was direkt davor steht. Im Falle, dass **const** am Anfang einer Anweisung steht, bezieht es sich auf das, was direkt dahinter steht.

```
1  const double a;
2  // ist äquivalent zu
3  double const a;
```

constexpr steht am Anfang einer Deklaration und bezieht sich auf die gesamte Deklaration.

```
1  constexpr double i{ 1.0 };
```

Konstanten können helfen, Fehler in Programmen schneller zu finden:

Typische Fehler

```
1  double pi = 3.14;                    // Setzen der Kreiszahl
2
3  // ... viel Code ...
4
5  pi = computeOsmoticPressurePi(...); // Berechne osmotischen Druck Pi
6
7  // ... viel Code ...
8
9  double kreisflaeche = pi * r * r;   // Oops...
```

Konstanten vs. Zahlenliterale

- Konstanten sind lesbarer als die Zahl jedes mal neu zu verwenden
- Konstanten können leichter gewartet werden
- Konstanten belegen bei der gleichen Verwendung ebenso keinen Speicher

```
1  const double pi{ 3.14 };
2  const double circleCircumferencePerRadius{ 2.0 * pi };
3  const double circleAreaPerRadiusSquared{ pi };
4
5  double radius1        = 42.0;
6  double circumference1 = circleCircumferencePerRadius * radius1;
7  double area1          = circleAreaPerRadiusSquared * radius1 * radius1;
```

Hinweis: Faustregel: Alle feststehenden Zahlen außer 0 und 1 als Konstanten deklarieren. Hinweis: Viele Naturkonstanten sind in `<cmath>` bereits definiert, so zum Beispiel die Zahl `M_PI`.

Konstanten vs. Variablen

- Konstanten können vom Compiler leichter optimiert werden
- Konstanten schützen vor versehentlichen Änderungen

Hinweis: Faustregel: Alle unveränderlichen Variablen als Konstanten deklarieren.

Literale sind Zeichenketten, die dazu dienen, Variablen Werte zuzuweisen.
Fließkommazahlen können auf unterschiedliche Arten angegeben werden:

double Literale

```
1  double a{ 1.0 };    // 1
2  double a{ 1.  };    // 1 (unschön, da schnell zu übersehen)
3  double f{ .1 };     // 0.1 (unschön, da schnell zu übersehen)
4  double b{ 1.E-8 };  // 0.00000001
5  double c{ .1E4 };   // 1000 = 0.1 * 10^4
6  double d{ 4e2 };    // 400 = 4 * 10^2
```

float literale

```
1  float a{ 1.0f };   // Regeln wie bei double, jedoch mit Suffix f oder F
2  float b{ 1.E-8f };
3  float c{ .1E8f };
```

long double Literale

```
1  long double a{ 1.0l };   // Regeln wie bei double, jedoch mit Suffix
2  long double b{ 1.E-8L }; // l oder L
3  long double c{ .1E8l };
```

e bzw. E ist hierbei ein „Infix“, f bzw. F und l bzw. L sind „Suffixe“.

Mathematische Operatoren

Tab. Arithmetische Operatoren[22]

C++	Beispiel	Bedeutung
`+`	`+i`	unäres Plus
`-`	`-i`	unäres Minus
`++`	`++i`	vorherige Inkrementierung um eins
`++`	`i++`	nachfolgende Inkrementierung um eins
`--`	`--i`	vorherige Dekrementierung um eins
`--`	`i--`	nachfolgende Dekrementierung um eins
`+`	`i + 2`	binäres Plus
`-`	`5 - i`	binäres Minus
`*`	`5 * i`	Multiplikation
`/`	`i / 2`	Division
`%`	`i % 4`	Modulo (Rest mit Vorzeichen von i)

Ganzzahl-Datentypen

Tab. Übliche Datentypgrößen

Typ	**übliche Größe**	**geforderte Größe** (C++ Standard)
`short` oder `short int`	16 bit	
`int`	32 bit	16 bit
`long` oder `long int`	64 bit	32 bit
`long long` oder `long long int`	$>$ 64 bit	

Analog dazu gibt es vorzeichenlose Typen. Ihnen wird das Schlüsselwort `unsigned` vorangestellt.

Garantierte Größenbeziehungen in C++

Folgende Größen sind in C++ garantiert:

Größenvorgaben

```
1  1 =  sizeof( char ) <= sizeof( short ) <= sizeof( int )
2    <= sizeof( long ) <= sizeof( long long )
3
4  1 <= sizeof( bool ) <= sizeof( long )
5
6  sizeof( float ) <= sizeof( double ) <= sizeof( long double )
7
8  // Für jeden Datentyp N aus char, short, int, long int, long long int gilt:
9  sizeof( N ) == sizeof( signed N ) == sizeof( unsigned N )
```

Ganzzahl-Suffixe geben den Datentyp an

Suffixe

```
42                          // Zahl vom Typ int
42u                         // Zahl vom Typ unsigned int
42l                         // Zahl vom Typ long int
42ul oder 1lu               // Zahl vom Typ unsigned long int
42llu                       // Zahl vom Typ unsgned long long int
```

Generell gilt: Eine Zahl wird mit dem ersten Typ dargestellt, der groß genug ist, diese Zahl darzustellen.

Ganzzahl-Präfixe sind 0 für Oktalzahlen und 0x für Hexadezimalzahlen und 0b für Binärzahlen (C++14)

Präfixe

```
1  int a{ 0x20 };                // a hat den Wert 32
2  int b{ 0xA };                 // b hat den Wert 10
3  int c{ 020 };                 // c hat den Wert 16
4  int d{ 20 };                  // d hat den Wert 20
5  unsigned int e{ 0b01100101 }  // e hat den Wert 101 (C++14)
6  unsigned int f{ 0b0110_0101 } // f hat den Wert 101 (C++14)
```

Vor allem in der hardwarenahen Programmierung ist es sinnvoll, direkten Zugriff auf die interne Darstellung ganzer Zahlen zu haben.

Tab. Bitoperatoren

C++	Beispiel	Bedeutung
`<<`	`i << n`	Linksschieben[23]
`>>`	`i >> n`	Rechtsschieben
`&`	`i & j`	bitweises UND
`^`	`i ^ j`	bitweises XOR
`|`	`i | j`	bitweises ODER
`~`	`~i`	bitweise Negation
`<<=`	`i <<= 3`	`i = i << 3`
`>>=`	`i >>= 3`	`i = i >> 3`
`&`	`i &= 3`	`i = i & 3`
`|=`	`i |= 3`	`i = i | 3`

[23] Linksschieben entspricht der Multiplikation mit 2^n wenn kein Überlauf auftritt.