Einführung in die Softwareentwicklung

SOMMERSEMESTER 2020

Institut für Informatik

JOHANNES GUTENBERG UNIVERSITÄT MAINZ

```
// This is C++
void main() {
   int var = 42;
   std::string s = "Hello World!\n";
   s += "The answer is still: ";
   s += string(var);
   std::cout << s;
}
```

Foliensatz #02

Grundlagen von C++

Michael Wand · Institut für Informatik · michael.wand@uni-mainz.de

Historie & Einordnung

Historie

Programmiersprache C

- Die Sprache „C“ war/ist sehr erfolgreich
- Zuerst 1972 erschienen
  - Maschinennah, aber komfortabler zu benutzen als Maschinensprache (Assembler)
  - Größtenteils „portabel“ (Prozessorunabhängig)
- Unix war in C geschrieben worden
  - Schnittstellen („Application Binary Interfaces“, ABIs) in C
  - Auch in Windows (und vielen anderen OS)
  - Erfolg unaufhaltsam

Historie

Programmiersprache C

- Hauptkonkurrent in der Praxis: Pascal
  - Später Modula-2 / Borland Pascal
  - Gemeinsame Abstammung: Algol
  - Größtenteils identische Funktionalität
- C ist „Industriestandard“
  - Vorteil: überall verfügbar, ausgereift
  - Nachteil: Kompatibilität hemmt Weiterentwicklung
- Beispielproblem: fehlendes Modulsystem
  - In Modula-2 seit 1978
  - Stundenlangen Übersetzungszeiten bei großen Systemen
  - (Erst) seit C++20 behoben

Idee für C++

Entwicklung

- Erfinder: Bjarne Stroustrup
- Erste Phase: 1979-83
  - Verbesserung von C
  - Objektorientierte Programmierung in C
  - „C with classes“
  - Precompiler für C („CFront“)
  - Standardisiert in C++98
- Hauptvorteil
  - Fast 100% kompatibel zu C
  - Auch ein Hauptnachteil…
  - Rasche Verbreitung!

Neuere Versionen

Weitere Entwicklung

- 2003: „C++03“ – Bugfixes zur 1998 Spezifikation
- 2011: „C++11“ – wesentliche Modernisierung
  - Bessere Unterstützung für funktionale Programmierung
  - Typinferenz (Umwidmung von „auto“)
  - Diverse Verbesserungen
- 2014: „C++14“ – Was das Kommittee 2011 nicht geschafft hat
- 2017: „C++17“ – Neuste Version
- 2020: Modulsystem, Concepts, diverse Details.
  - Reflection wieder verschoben

Vor- und Nachteile

Vorteil

- Schnell, viele verschiedene Featuers
- Alternativlos
  - Keine andere Sprache mit diesem Profil ist ähnlich weit verbreitet und unterstützt

Nachteil

- Altlasten und Inkonsistenzen
- Diskussion / kritische Analyse z.B. in:

  Yossi.Kreinin, C++ frequently questioned answers, https://yosefk.com/c++fqa/

Warum C++?

Python & C++ als Plattform

- Ökosystem für „Scientific Computing“
  - Schwerpunkt in Mainz (naturwiss. Informatik)
- Integration C++ & Python
  - Python als Middleware, C/C++ als Low-Level Plattform
  - Aufteilung in Python Module behebt wichtige Schwäche
    - Kompilierzeiten (kleine, unabhängige Module)
    - Scripting in Python
    - Dynamische Metaprogrammierung in Python
- Vertierungsvorlesungen
  - Modellierung, Computer Graphik, Parallele Algorithmen, Verteilte Systeme, Bioinformatik, …

Softwarelandschaft

Einschätzung: Ökosysteme

- „Virtual machines“ wie `JVM` oder `.net`
  - z.T. / demnächst auch JS/WebAssembly
  - Viele Sprachen werden interoperabel
    - z.B. Scala, Kotlin, JAVA (JVM)
    - z.B. F#, C#, managed C++ (.net)
  - Standard bei Unternehmenssoftware
- C/C++ mit Skriptsprache (Lua, Python, Bash)
  - Anbindung anderer Sprachen über C-ABIs (shared libraries)
  - (Älterer) Unix-Standard
  - Performancekritische Anwendungen
  - Python & C++ heute Standard in Wissenschaft / Datenanalyse

Warum eine zweite Sprache?

Was fehlt in Python?

- Statische Typisierung
- Hardwarenähe (Speed!)

Lernziele C++

- Statisch typisierte Programmiersprache(n)
- Abstraktionen für effiziente, hardware-nahe Programmierung

C++ vs. Python Beispiel

Python – dynamisch typisiert

```python
# ganze Zahl
a = 42

# Fließkommazahl (auch double prec.)
pi = 3.14

# String (eingebaut)
text = "Hello World!"

# Kein Fehler
text = 42;  // Why not? Typ ändert sich!
a = pi;     // Typ ändert sich

# Laufzeitfehler (Übersetzung ok)
text = "Hello World!"  # noch ok
text = text * 42   # Laufzeitfehler
```

C++ - statisch typisiert

```cpp
// ganze Zahl
int a = 42;

// Fließkommazahl (double precision)
double pi = 3.14; // ungefähr

// String (Klasse aus Bibliothek “std”)
std::string text = "Hello World!";

// Diese Fehler werden erkannt!
text = 42;  // Fehler beim Übersetzen!
a = pi;     // Warnung beim Übersetzen!

// Auch Übersetzungsfehler
text = "Hello World!";    // soweit ok
text = text * 42; // Übersetzungsfehler
```

Typsysteme

In C/C+ haben Daten immer einen festen Typ

- In C/C++ legt der Typ fest
  - Semantik (z.B. „zwei ganze Zahlen mit Vorzeichen“)
  - Speicherlayout (z.B. 32bit integer, 32bit Fließkomma)
- C/C++ - Beispiel einer Typdefinition

```
struct ZweiZahlen {
    int   ganzeZahl1;   // i.d.R. 32-bit integer
    float ganzeZahl2;   // 32-bit floating point
};
ZweiZahlen a; // a besteht aus zwei Zahlen
int b;        // b besteht ist eine „Integer“ Zahl
```

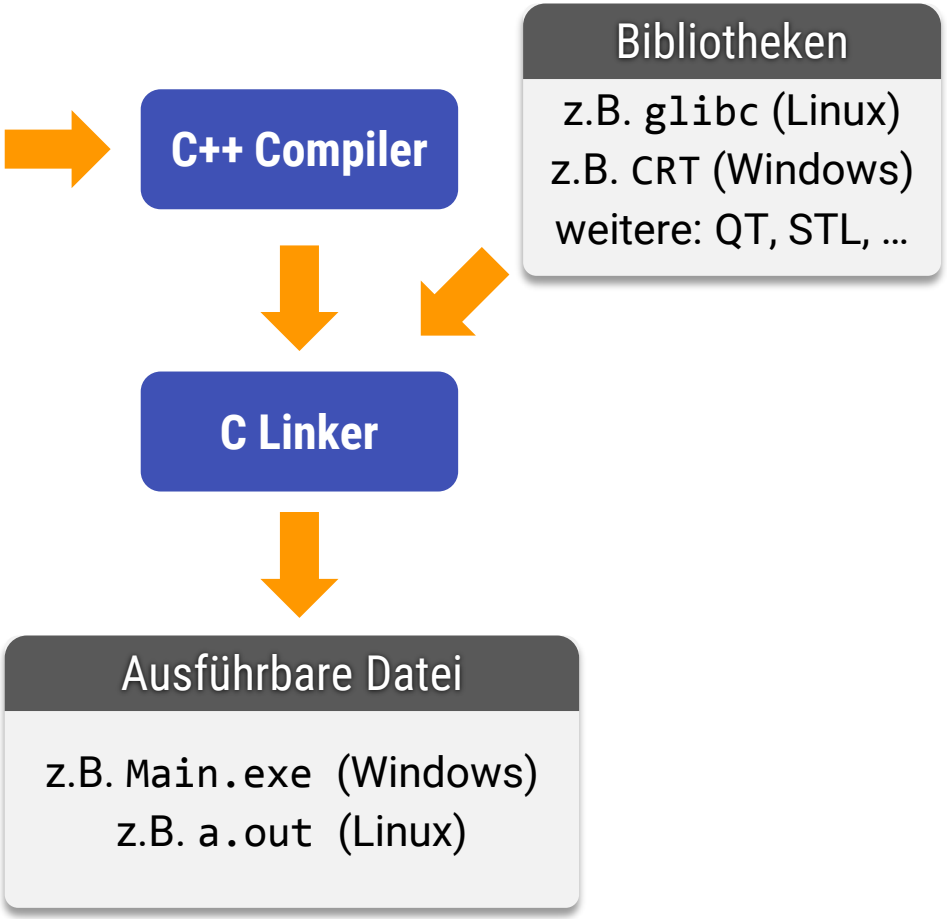
„Pipeline“

Übersetzung: Einzelne Datei

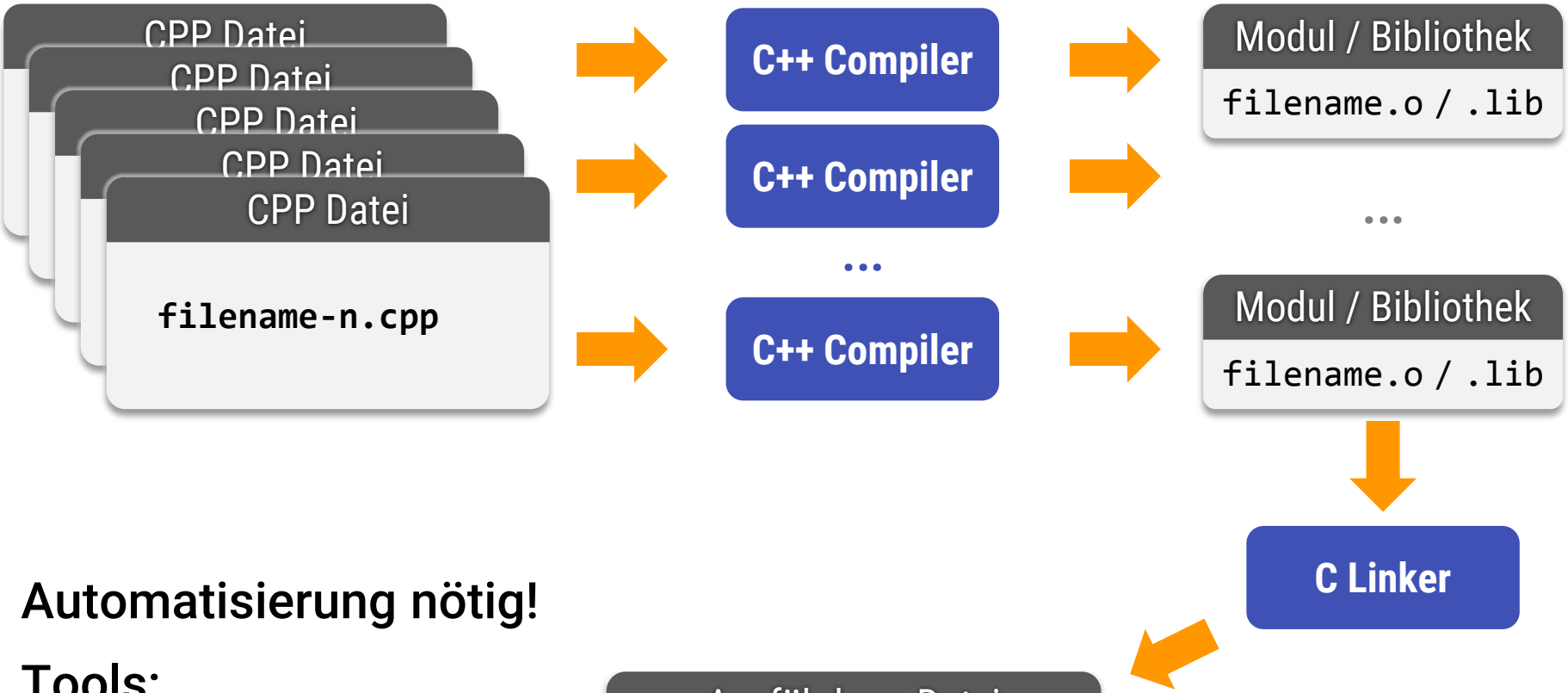
Datei: main.cpp

```
#include <iostream>

void main() {
    int var = 42;
    std::cout << "Hello World!";
}
```



Übersetzung: Komplexes System



Automatisierung nötig!

Tools:

- CMake (verbreitet)
- QMake (QTCreator)
- Make (veraltet?)

Hinweis: dynamische Bibliotheken möglich

- DLLs (Windows)
- .so (Linux)

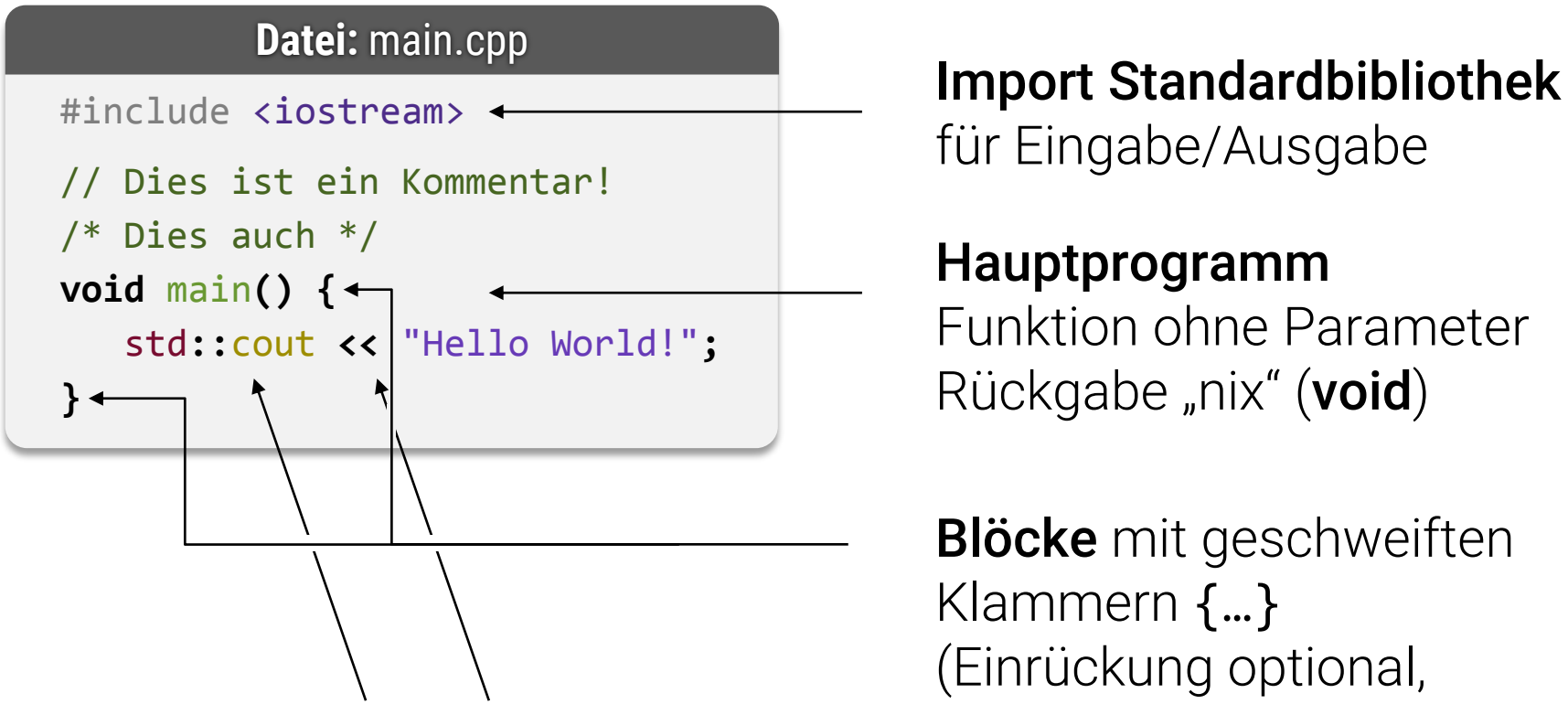
Minimalbeispiel

Das erste C++ Program

Datei: main.cpp

```
#include <iostream>

// Dies ist ein Kommentar!
/* Dies auch */
void main() {
   std::cout << "Hello World!";
}
```



Import Standardbibliothek
für Eingabe/Ausgabe

Hauptprogramm
Funktion ohne Parameter
Rückgabe „nix“ (**void**)

Blöcke mit geschweiften Klammern **{…}**
(Einrückung optional, keine Funktion)

Ausgabe-Objekt
für Konsole „cout“
Namensraum „std“
(Standardbibliothek)

Ausgabe
Zeichenkette mit "…"
Ausgabeoperator „**<<**“

Erstes Problem

Datei: main.cpp

```
#include <iostream>

// Dies ist ein Kommentar!
/* Dies auch */
int main() {
   std::cout << "Hello World!";
   return 0; // kein Fehler
}
```


Hauptprogramm

muss (eigentlich) einen Rückgabewert haben!

Rückgabe als Fehlercode (`0` = alles ok)

Unter Windows/MVC++ ok

GCC / CLANG unterstützen Ausnahme

Vergleich C++ - Python

Datei: main.cpp

```cpp
#include <math>
#include <iostream>


// Definition einer Funktion
void compute_angles() {
    double val = std::sin(42);
    std::cout << val;
}

// Hauptprog. ist eine Funktion!
int main() {
    compute_angles();
    return 0;
}
```

Datei: main.py

```python
import math
# print ist fest eingebaut

# Definition einer Funktion
def compute_angles():
    val = math.sin(42)
    print(val)


# Hauptprogramm
compute_angles()
```

Namensräume

Datei: main.cpp

```cpp
#include <math>
#include <iostream>

using std::cout;
using std::sin;

// Definition einer Funktion
void compute_angles() {
    double val = sin(42);
    cout << val;
}

// Hauptprog. ist eine Funktion!
int main() {
    compute_angles();
    return 0;
}
```

Datei: main.py

```python
from math import sin
# print ist fest eingebaut


# Definition einer Funktion
def compute_angles():
    val = sin(42)
    print(val)


# Hauptprogramm
compute_angles()
```

Namensräume

Datei: main.cpp

```cpp
#include <math>
#include <iostream>

using namespace std;

// Definition einer Funktion
void compute_angles() {
    double val = sin(42);
    cout << val;
}

// Hauptprog. ist eine Funktion!
int main() {
    compute_angles();
    return 0;
}
```

Datei: main.py

```python
from math import *
# print ist fest eingebaut


# Definition einer Funktion
def compute_angles():
    val = sin(42)
    print(val)


# Hauptprogramm
compute_angles()
```

Typen, Ausdrücke, Befehle

Variablendeklaration

In C++ müssen Variablen deklariert werden!

- Vor Benutzung!
- Gültigkeit nur im selben {…}-Block
- Typ muss angegeben werden

Beispiele

```
double x;  // Fließkommazahl (64 Bit / doppelte Genauigkeit)
double y;  // Übrigens: Das gleiches Format wie in Python
x = 42.0;  // Zuweisung eines Wertes
y = 23.0;  // Ditto.
double z = x + y;  // Deklaration und sofortige Initialisierung
int w = 1337;      // Ditto, aber Typ „ganze Zahl“ (meist 32 Bit).
```

Befehle und Blöcke

Blöcke & Befehle

- Mit Klammern gekennzeichnet: {…}
- Einrückungen sind dem compiler völlig egal
  - Dem Menschen nicht → immer einrücken!
- Befehle mit Semikolon „;“ abgeschlossen
  - (Zeilenumbruch egal; wegen Lesbarkeit immer verwenden!)

Beispiel:

```
double x = 42.0;
cout << x;  // Ergibt „42“
{
   double x = 23.0;
   cout << x;  // Ergibt „23“
}
cout << x;  // Ergibt „42“
```

Typen in C++

Primitive Typen (repräsentieren Prozessortypen)

- **`int`**: ganze Zahlen
- **`float`** / **`double`**: Fließkommazahlen
- **`char`**: Zeichen (Bytes)
- **`void`**: nix (Typ für leere Menge)
- **`bool`**: Wahrheitswert (urspr. **`int`**; nicht-null ≡ **`true`**)
- `<type>`**`*`** Zeiger (später)

Modifizierer

- **`long`** / **`short`** – Verschiedene Größen
- **`signed`** / **`unsigned`** – mit/ohne Vorzeichen

Primitive Typen in C++

Beispiele (typische Formate)

- **`signed char`**: 8 Bit integer, mit Vorzeichen
- **`short int`**: 16 Bit integer, mit Vorzeichen
- **`int`**: 32 Bit integer, mit Vorzeichen
- **`long int`**: 64 Bit integer, mit Vorzeichen

Ohne Vorzeichen

- **`unsigned char`**: 8 Bit integer, ohne Vorzeichen
- **`short unsigned [int]`**: 16 Bit integer, ohne Vorz.
- **`unsigned [int]`**: 32 Bit integer, ohne Vorzeichen
- **`unsigned long int`**: 64 Bit integer, ohne Vorzeichen

Primitive Typen in C++

Fließkomma

- **`float`**: 32Bit Fließkomma
- **`double`**: 64Bit Fließkomma
- **`[long] long double`**:
  80Bit Fließkomma auf einigen x86 Compilern

Achtung

- Genaue Formate sind Compiler+OS+CPU abhängig!
- Sicherheit: `#include <cstdint>`
  - Definiert **`uint32_t`**, **`int64_t`**, **`uint8_t`** und weitere
- Fließkommaformate **`float`**, **`double`** fast immer 32/64Bit

Primitive Typen

Beispiele

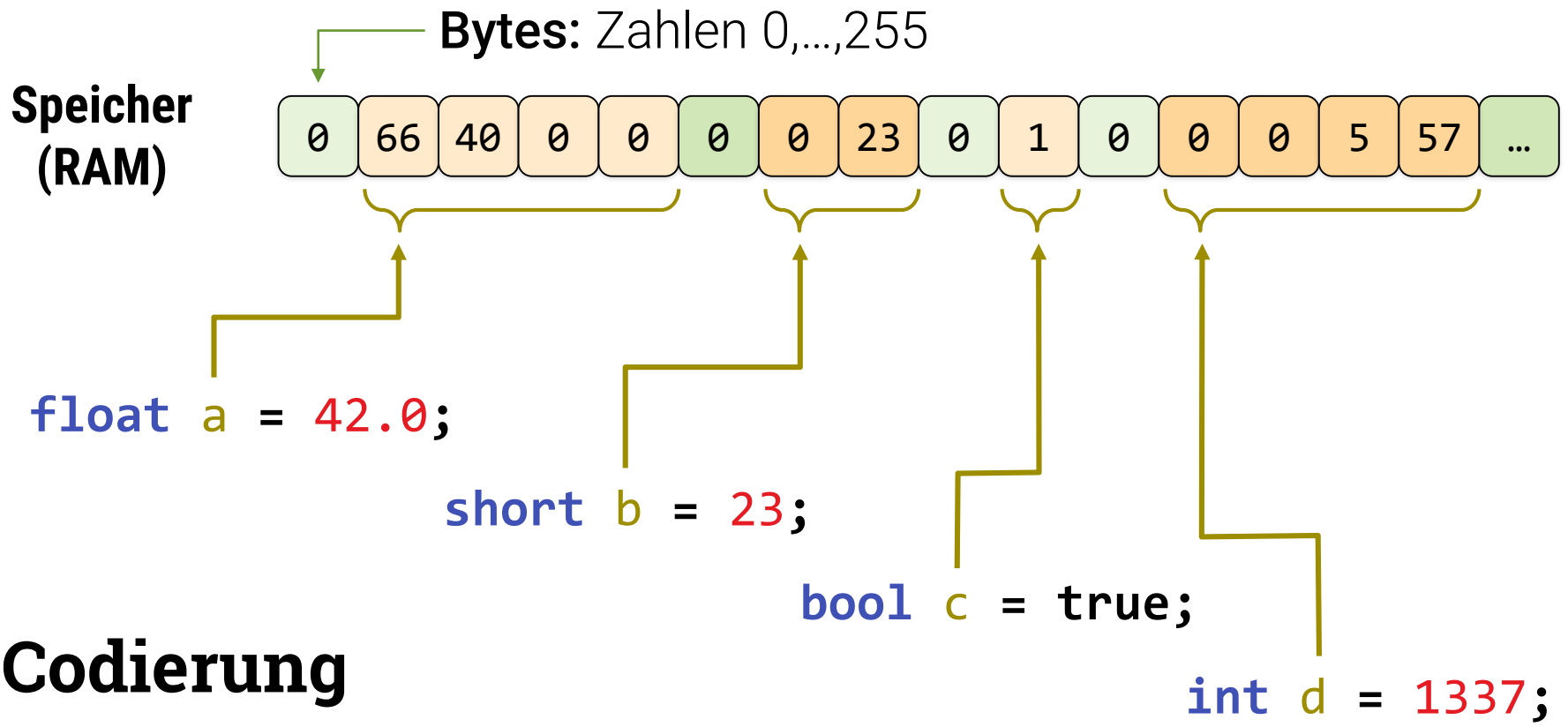
```
float    a =  42.0f; // Suffix „f“ für einfache Genauigkeit
                     // Manche Compiler geben sonst „Warnings“
double   b =  23.0;  // Doppelte Genauigkeit

unsigned c =  1337;  // Ganze Zahl
int      d = -1337;  // Negative Zahl
unsigned e =    -1;  // Viel Glück! (mehr dazu später)

short    f = -1234;  // in der Regel eine 16-Bit Zahl
int16_t  g = -1234;  // jetzt mit Sicherheit eine 16-Bit Zahl

char     h =   255;  // Sollte meist ein Byte sein (Vorzeichen? Größe?)
uint8_t  i =   255;  // Jetzt mit Sicherheit ein Byte (8-Bit, unsigned)
```

Maschinenrepräsentation



Codierung

- Aufeinanderfolgende Bytes
- Variable = Zeiger auf erstes Byte (Länge implizit)

Zwei Besonderheiten

Konstanten

```
const int fixed =  42; // Kann mann nicht mehr ändern
fixed =  23; // Compilerfehler!
```

- Prefix „const“ erzeugt Konstante
  - Ändern nur bei Inititalisierung
- Auch bei Parametern für Unterprogramme
  - Keine Änderung nach Übergabe des Wertes
  - Erlaubt Optimierungen

Zwei Besonderheiten

Referenzen

```
int variable = 42; // Eine Variable
int &ref = variable; // Eine zweite Refernez auf die gleiche Variable

variable =  23;   // variable ändern
cout << variable;      // gibt nun 23 aus
cout << ref;      // gibt nun auch 23 aus

ref =  1337;      // gleiche variable ändern
cout << variable; // gibt nun 1337 aus
cout << ref; // gibt nun auch 1337 aus
```

- „<Typ> &“ ist eine Referenz (Alias) auf eine Variable
  - Unterschied zu Pointern (später): Muss auf Variable zeigen
  - Auch bei Parametern für Unterprogramme (wichtig)

C++

Ausdrücke, Seiteneffekte

Elementare Anweisungen

Beispiele

```
// Variablen deklarieren (immer vorab!)
int a; int b; int c;

// Wert zuweisen
a = 42;

// Arithmetische Ausdrücke
b = 2 + 2;

// Ablauf: Immer erst rechte Seite auswerten, dann zuweisen!
c = a + b;
```

Ausdrücke mit Seiteneffekten

Besonderheit in C++

- Ausdrücke
  - Arithmetische Terme
  - z.B. `2 + 4 - a / (5*a + b)`
  - Operatoren explizit (`5*a` statt „`5a`“)
  - Prioritäten wie gewohnt
- Anweisungen
  - Ausdrücke mit „Seiteneffekt“
  - durch Semikolon „ `;` “ getrennt
  - z.B. `a = … ; b = … ;`
- Ausdrücke ohne Seiteneffekt in C++ legal
  - „`42;`“ hat aber keinen Effekt (in Java verboten).

Zulässige Ausdrücke

Beispiele

```
// Variablen deklarieren (immer vorab!)
int a; int b; int c = 0;

// Wert zuweisen
a = 42;

// Ausdruck b = 42 gibt Wert der rechten Seite zurück
a = b = 42;

// Zählt c um eins hoch
c = c + 1;
c += 1; // das auch
c++; // kürzer! (c++ gibt orginal zurück, ++c erhöht erst)
```

Die wichtigsten Operatoren

Operator	Beschreibung,	
+, -, *, /	Grundrechenarten	**Rechnen**
%	Modulo (Rest der Division: 42 % 10 ist 2)	
==, !=	Gleichheit, Ungleicheit	**Vergleich**
<, <=, >=, >	Kleiner(-gleich) / größer(-gleich)	
!	Logisches „nicht“ (bool)	**Logische Operationen**
&&, &	Logisches „und“ (bool)	
||, |	Logisches „oder“ (bool)	
^	„Exklusiv-oder“ (XOR)	
=	Zuweisung	**Operatoren mit Seiteneffekt**
+=, *=, &=, …	Operation und Zuweisung	
++, --	Inkrement/Dekrement	

Ein = oder zwei ==

Achtung

- Vergleichsoperator `a == b`
  - Liefert **`true`** / **`false`**.
- Zuweisungsoperator `a = b`
  - Führt Zuweisung aus.
  - Gibt `b` zurück
- Nicht verwechseln!

Priorität

Unterschiedliche Priorität

- Punkt vor Strichrechnung

```
    2 * a + b;
⇒ (2 * a) + b;
```

- Arithmetik vor Zuweisungen

```
   a = 2 * a + b;
⇒ a = (2 * a + b);
```

  Rechte Seite zuerst auswerten!

- Klammern `()` können jederzeit genutzt werden um Prioritäten zu klären

Priorität

Operators	Precedence (high to low)
postfix	*expr++ expr--*
unary	*++expr --expr +expr -expr* ~ !
multiplicative	* / %
additive	+ -
shift	<< >> >>>
relational	< > <= >= instanceof
equality	== !=
bitwise AND	&
bitwise exclusive OR	^
bitwise inclusive OR	|
logical AND	&&
logical OR	||
ternary	? :
assignment	= += -= *= /= %= &= ^= |= <<= >>= >>>=

höher

niedriger

Priorität

Precedence	Operator	Description	Associativity
1	`::`	Scope resolution	Left-to-right
2	`a++ a--`	Suffix/postfix increment and decrement	
	type`()` *type*`{}`	Functional cast	
	`a()`	Function call	
	`a[]`	Subscript	
	`. ->`	Member access	
3	`++a --a`	Prefix increment and decrement	Right-to-left
	`+a -a`	Unary plus and minus	
	`! ~`	Logical NOT and bitwise NOT	
	(*type*)	C-style cast	
	`*a`	Indirection (dereference)	
	`&a`	Address-of	
	`sizeof`	Size-of[note 1]	
	`new new[]`	Dynamic memory allocation	
	`delete delete[]`	Dynamic memory deallocation	
4	`.* ->*`	Pointer-to-member	Left-to-right
5	`a*b a/b a%b`	Multiplication, division, and remainder	
6	`a+b a-b`	Addition and subtraction	
7	`<< >>`	Bitwise left shift and right shift	
8	`<=>`	Three-way comparison operator (since C++20)	
9	`< <=`	For relational operators < and ≤ respectively	
	`> >=`	For relational operators > and ≥ respectively	
10	`== !=`	For relational operators = and ≠ respectively	
11	`&`	Bitwise AND	
12	`^`	Bitwise XOR (exclusive or)	
13	`|`	Bitwise OR (inclusive or)	
14	`&&`	Logical AND	
15	`||`	Logical OR	
16	`a?b:c`	Ternary conditional[note 2]	Right-to-left
	`throw`	throw operator	
	`=`	Direct assignment (provided by default for C++ classes)	
	`+= -=`	Compound assignment by sum and difference	
	`*= /= %=`	Compound assignment by product, quotient, and remainder	
	`<<= >>=`	Compound assignment by bitwise left shift and right shift	
	`&= ^= |=`	Compound assignment by bitwise AND, XOR, and OR	
17	`,`	Comma	Left-to-right

höher

niedriger

http://en.cppreference.com/w/cpp/language/operator_precedence

Typumwandlungen (Casting)

Alter Stil (C/C++ 98/JAVA):

- Syntax: `(<neuer Typ>)<variable>`
- Beispiel: `(double)42`
- Beispiel:
```
double pi = 3.14;
int p = (int)pi;
```
- Änderung des Typs; Anpassung falls Repräsentation anders (z.B. int → double)

Typumwandlungen (Casting)

Neuer Stil (modernes C++)

- `const_cast<neuer Typ>(variable)`
  - add / remove const attribute
- `static_cast<neuer Typ>(variable)`
  - Konvertierung des Typs
- `reinterpret_cast<neuer Typ>(variable)`
  - Bytefolge anders interpretieren
  - (Alter Stil: Mit Pointern)
- `dynamic_cast<neuer Typ>(variable)`
  - Mit dynamischen Typcheck (OOP/ nur für Klassentypen mit virtuelle Methoden / mehr dazu später)

Typumwandlungen (Casting)

Beispiel

- ```
  const int x = 23;
  const_cast<int>(x) = 42;
  ```
  - Gemogelt (häufig bei schlecht designten „const methods“ im OOP-Kontext)
- ```
  static_cast<int>(3.14) // ergibt 3
  ```

Alter C-Cast „(Typ)var“:

- Versucht const_cast,
- danach static_cast,
- und danach reinterpret_cast

C++

Kontrollfluss

Kontrollfluss in Python

Sequenz

```
# nacheinander ausführen
x = 42
y = 23
x = x + y
print(x)
```

Wiederholung

```
# Beispielschleife
x = 23
while x <= 42:
    print(x)
    x = x + 1
```

Fallunterscheidung

```
# Bsp.-Fallunterscheidung
if x == 42:
    print("very meaningful")
else:
    print("don‘t care")
```

Unterprogramme

```
# Unterprogram
def comp_sum(x, y):
    z = x + y
    return(z)
```

Kontrollfluss in C++

Sequenz

```
// nacheinander ausführen
int x = 42;
int y = 23;
x = x + y;
cout << x;
```

Wiederholung

```
// Beispielschleife
x = 23;
while (x <= 42) {
    cout << x;
    x = x + 1;
}
```

Fallunterscheidung

```
// Bsp.-Fallunterscheidung
if (x == 42) {
   cout << "very meaningful";
} else {
   cout << "don't care";
}
```

Unterprogramme

```
// Unterprogram
int compSum(int x, int y)
{
    int z = x + y;
    return z;
}
```