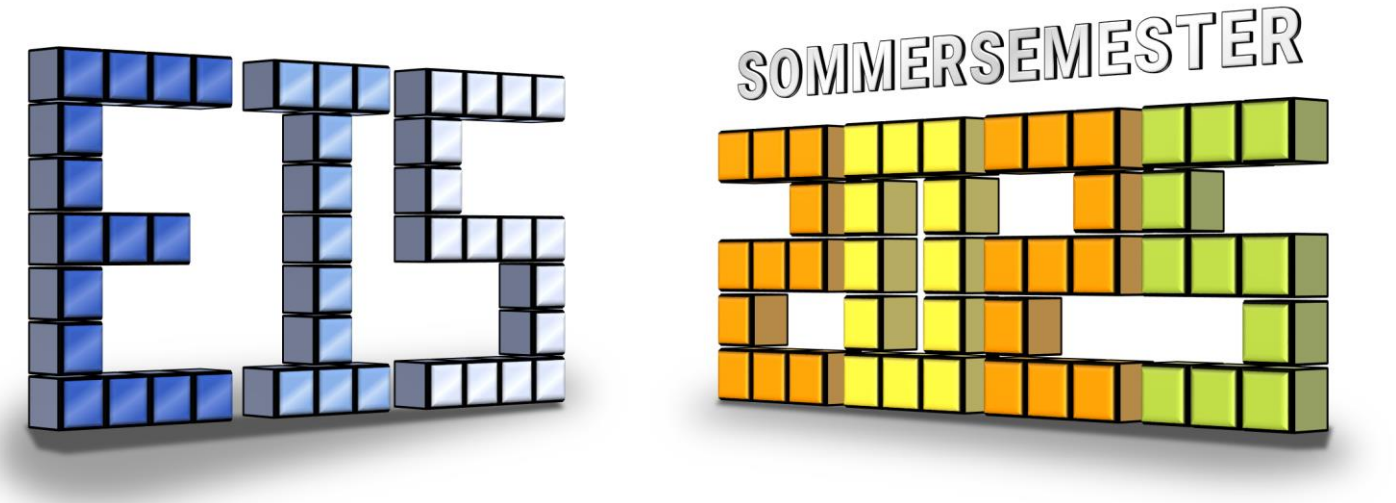
EINFÜHRUNG IN DIE SOFTWAREENTWICKLUNG

Sommersemester 2025



Foliensatz #2b

Programmiersprachen: C und C++

Michael Wand
Institut für Informatik
Michael.Wand@uni-mainz.de

Übersicht

Inhalt heute

- Programmiersprachen
  - Python + MyPy
  - C/C++
  - Java/Scala

Diskussion: Programmiersprache

Vorlage zur Diskussion

- Daten + Variablen
- Ausdrücke und Berechnungen
- Befehle
- Abstraktionen
- Systemumgebung

Programmiersprachen

(b) C, C++

Historie

Programmiersprachen C/C++

- Maschinennahe, effiziente Programmiersprache(n)
- Gemeinsamer Urahne: Algol
  - Mehrere Linien von Nachfolgern C, C++, C#
    und Pascal, Modula-2, Oberon, Delphi / Object-Pascal, ADA
  - C und Pascal/Modula sehr ähnlich, aber C-Linie am Ende beliebter
- Unix, Windows (u.v.a.) in C geschrieben
  - C ist bis heute das Standard „ABI“ für die meisten OSs
- C++ = C mit mehr Features
  - Varianten: CFront, C++98, C++11/14/17/20, aktuell C++23
  - Oft kritisiert wg. Komplexität, Altlasten und Inkonsistenzen
  - Immer noch „Industriestandard“ für effizienten Code

Diskussion: Programmiersprache

Vorlage zur Diskussion

- Daten + Variablen
- Ausdrücke und Berechnungen
- Befehle
- Abstraktionen
- Systemumgebung

C++ vs. Python Beispiel

Python

```
# ganze Zahl
a = 42

# Fließkommazahl (double prececision)
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++

```
// Variablen müssen mit Typ deklariert werden
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
```

Typen in C++

Primitive Typen (repräsentieren Prozessortypen)

- `int`: ganze Zahlen
- `float` / `double`: Fließkommazahlen
- `char`: Zeichen (Bytes)
- `void`: nix (Typ entspricht „leere Menge“)
- `bool`: Wahrheitswert (urspr. `int`; nicht-null ≡ **`true`**)
- `<type>*` Zeiger auf Typ („Referenz“, aber low-level)

Modifizierer

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

Typen in C++

Garantierter Speicherrepräsentation:

- `u_int_8` – ein Byte, kein Vorzeichen
- `u_int_32` – 32-Bit, kein Vorzeichen
- `int_16` – 16-Bit, mit Vorzeichen
- etc…

Definiert in extra Modul

- `#include <cstdint>`

Zusammengesetzte Datentypen

Structs (entspr. ungefähr Python Klassen)

```
struct Bruch {
    int zaehler;
    int nenner;
};
Bruch drei_viertel;
drei_viertel.zaehler = 3;
drei_viertel.nenner = 4;
```

Zusammengesetzte Datentypen

Arrays (entspr. ungefähr Python Listen)

```
#include <vector>   // am Anfang der Datei


std::vector<int> vec(2); // enthält Zahlen (int),
vec[0] = 3;              // hat Länge 2 am Anfang
vec[1] = 4;

vec.resize(3); // Änderung zu Länge 3

vec[2] = 5;

std::cout << vec[1]; // Ausgabe: „4“

std::cout << vec.size(); // Ausgabe: „3“
```

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 ist eine „Integer“ Zahl
```

Immutable Variables

Immutable Variables (Unveränderliche Variablen)

- `const int x = 23;`
- Werte, die nicht mehr verändert werden können/dürfen
  - Nur bei Initialisierung
- In Python
  - Es gibt immutable Types (`int`, `float`, `str`, `tuple`, …),
  - Aber keine Möglichkeit, unveränderlichen Variablen beliebiger Typen zu erzeugen

Diskussion: Programmiersprache

Vorlage zur Diskussion

- Daten + Variablen
- Ausdrücke und Berechnungen
- Befehle
- Abstraktionen
- Systemumgebung

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;
```

Kleine Unterschiede zu Python, z.B.

- „Power“ Operator ** (a**b = a hoch b) gibt es nicht
- Statt dessen `pow(basis, exp)` via `#include <cmath>`

Ausdrücke mit Seiteneffekten

Besonderheit in C++

- Ausdrücke
  - Arithmetische Terme, fast genau wie in Python
- 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).
- „Wertsemantik“ statt „Referenzsemantik“
  - Zuweisungen `a = …` kopieren Speicherinhalt
  - Referenzen über Zeigertypen (Adressen im Speicher)

Zulässige Ausdrücke

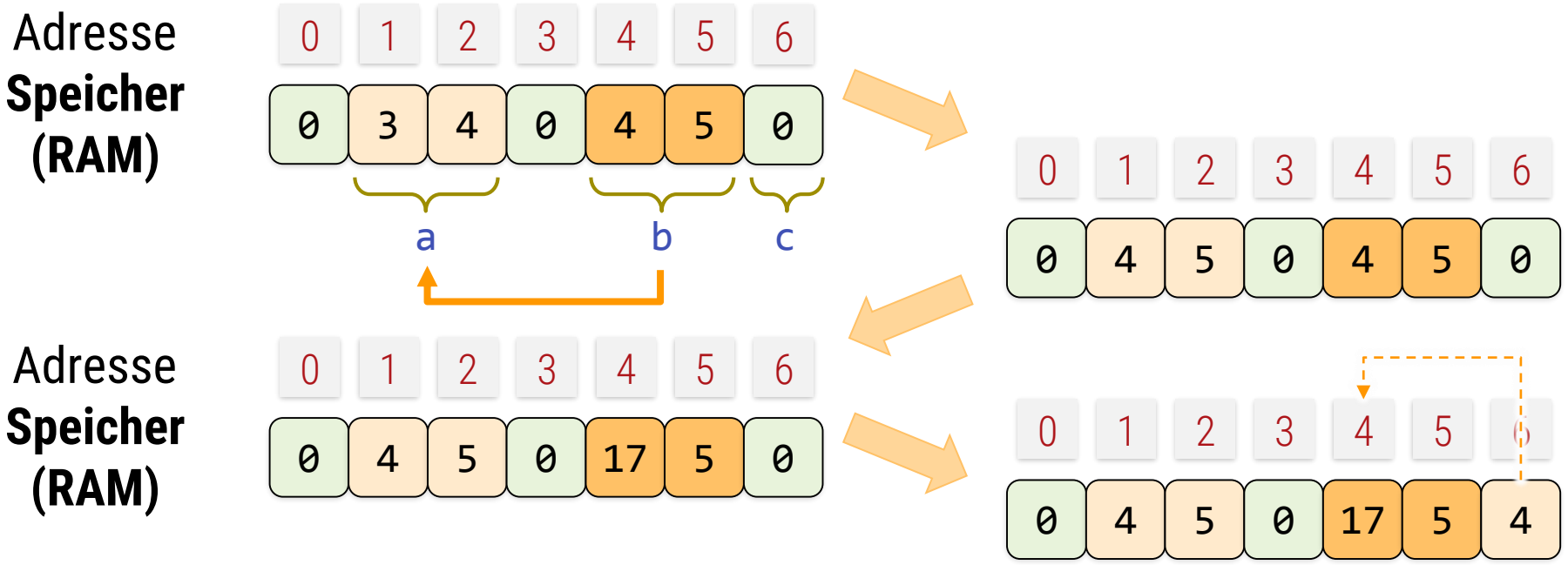
Beispiele

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

// 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)
```

Wert- vs. Referenzsemantik



Werttypen vs. Referenzen

```
a = ZweiZahlen(); a.ganzeZahl1 = 2; a.ganzeZahl2 = 3;
b = ZweiZahlen(); b.ganzeZahl1 = 4; b.ganzeZahl2 = 5;
a = b; // Wertsemantik: Speicherbereich von d mit Inhalt von e überschrieben
b.ganzeZahl1 = 17;
cout << a.ganzeZahl1; // Immer noch 4!
int *c = &b; // Zeigertypen („int*“) für explizite Referenzen
```

Die wichtigsten Operatoren

Operator	Beschreibung,	
`+, -, *, /`	Grundrechenarten	**Rechnen**
`%`	Modulo (Rest der Division: 42 % 10 ist 2)	
`==, !=`	Gleichheit, Ungleichheit	**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!

Im Prinzip

- Alles fast wie in Python
  - Python vermeidet aber manche Fehler, wie „`if a=b:`“…

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)
  - Typkonvertierung, soweit möglich
  - Reinterpretation des Speicherinhaltes möglich via Zeiger
- Neuerer Syntax seit C++03 – hier nicht diskutiert

Diskussion: Programmiersprache

Vorlage zur Diskussion

- Daten + Variablen
- Ausdrücke und Berechnungen
- Befehle
- Abstraktionen
- Systemumgebung

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;
}
```

Muster

Allgemeines Muster

- ```
  if ( <logischer Ausdruck> ) {
      <Anweisungen1>
  } else if ( <logischer Ausdruck> ) {
      <Anweisungen1>
  } else {
    <Anweisungen2>
  }
  ```
- Man kann die „else“ / “else if“-Teile natürlich auch weglassen
  ```

Muster

Allgemeines Muster

```
while ( <logischer Ausdruck> ) {
    <Anweisungen>
}
```

- Überprüfung am Anfang

Muster

Überprüfung am Ende

```
do {
    <Anweisungen>
} while ( <logischer Ausdruck> )
```

- Anweisungen werden immer einmal ausgeführt

break / continue

Vorzeitiger Abbruch

```
while ( <logischer Ausdruck> ) {
    ...
    break;
    ...
}
```

„break“ verläßt Schleife

break / continue

Encore une fois!

```
while ( <logischer Ausdruck> ) {
    ...
    continue;
    ...
}
```

„continue“ geht direkt zurück an Anfang

Muster

Spezialfall

```
for (<init>; <Bedingung>; <amEnde>) {
    <Anweisungen>
}
```

- Äquivalent zu:

```
<init>;
while (<Bedingung>) {
    <Anweisungen>;
    <amEnde>;
}
```

100% Äquivalent:

for-Schleife

```
for (int countDown = 10; countDown >= 1; countDown--) {
    cout << countDown;
    cout << ", ";
}
cout << "0 - liftoff!";
```

while-Schleife

```
{int countDown = 10;
while (countDown >= 1) {
    cout << countDown;
    cout << ", ";
    countDown--;
}}
cout << "0 - liftoff!";
```

Unterprogramme

Einfaches Programm mit zwei Funktionen

```
#include <iostream>
using std::cout;

int addNumbers(int a, int b) {
   // Aufruf: Wertsemantik, Werte werden in a, b kopiert
   int c = a + b;
   return c;
   // Bei return auch (Werte werden kopiert)
}
int main() {
      cout << "23 + 42 ergibt: ";
      // Aufruf des Unterprogramms
      cout << addNumbers(23, 42);
      return 0; // Hauptprogramm endet ohne Fehler
}
```

„Const“-Paramter

Lokale Variablen

```
// „Const“-Parameter können nicht mehr geändert werden
// Erlaubt Compiler mehr Optimierungen
// und vermeidet Fehler
int addNumbers(const int a, const int b) {
   int c = a + b;
   a += 42; // nicht erlaubt – Compilerfehler!
   return c;
}
void main() {
   int d = addNumbers(1, 2);
}
```

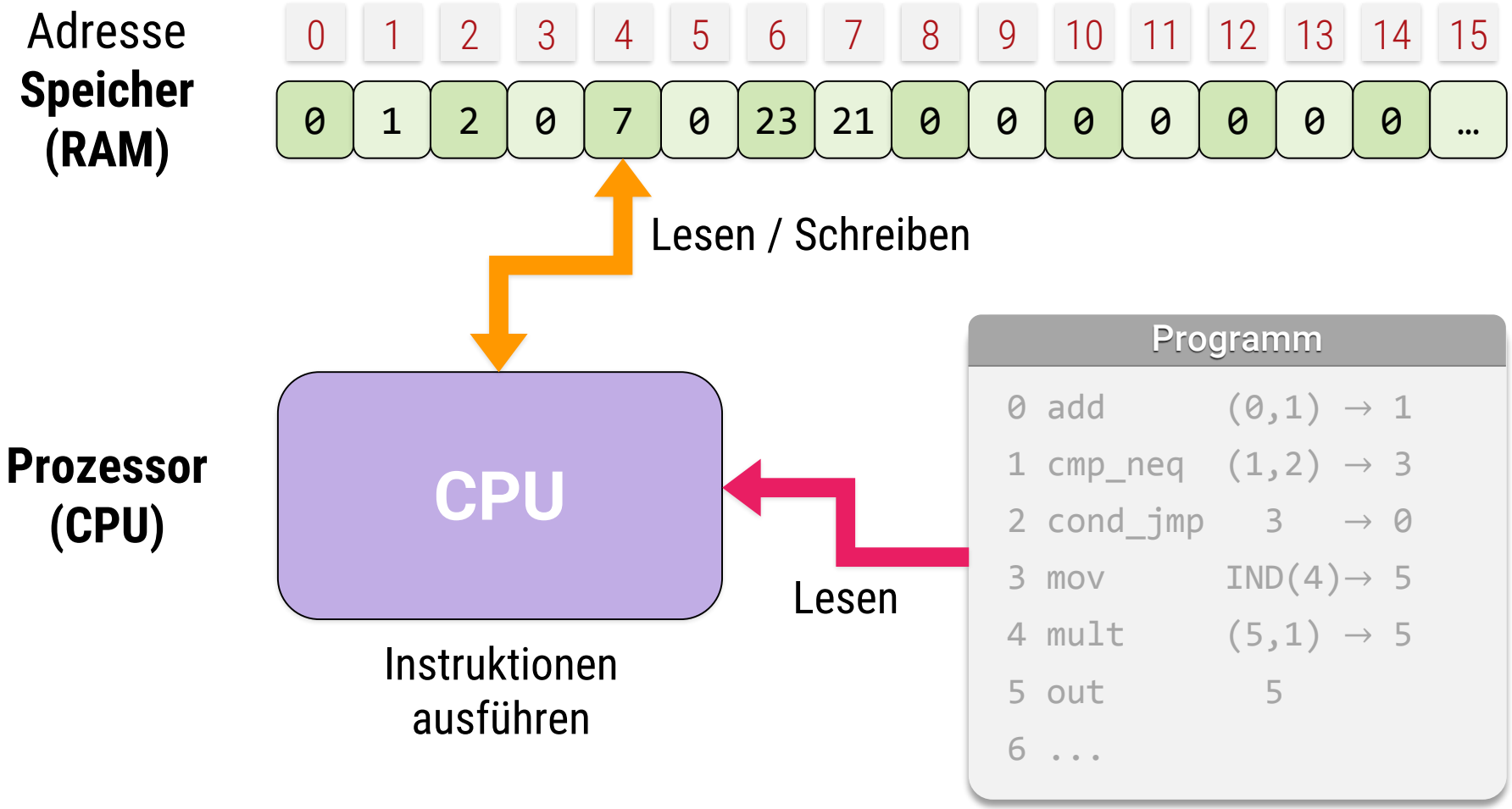
Referenzparameter

Lokale Variablen

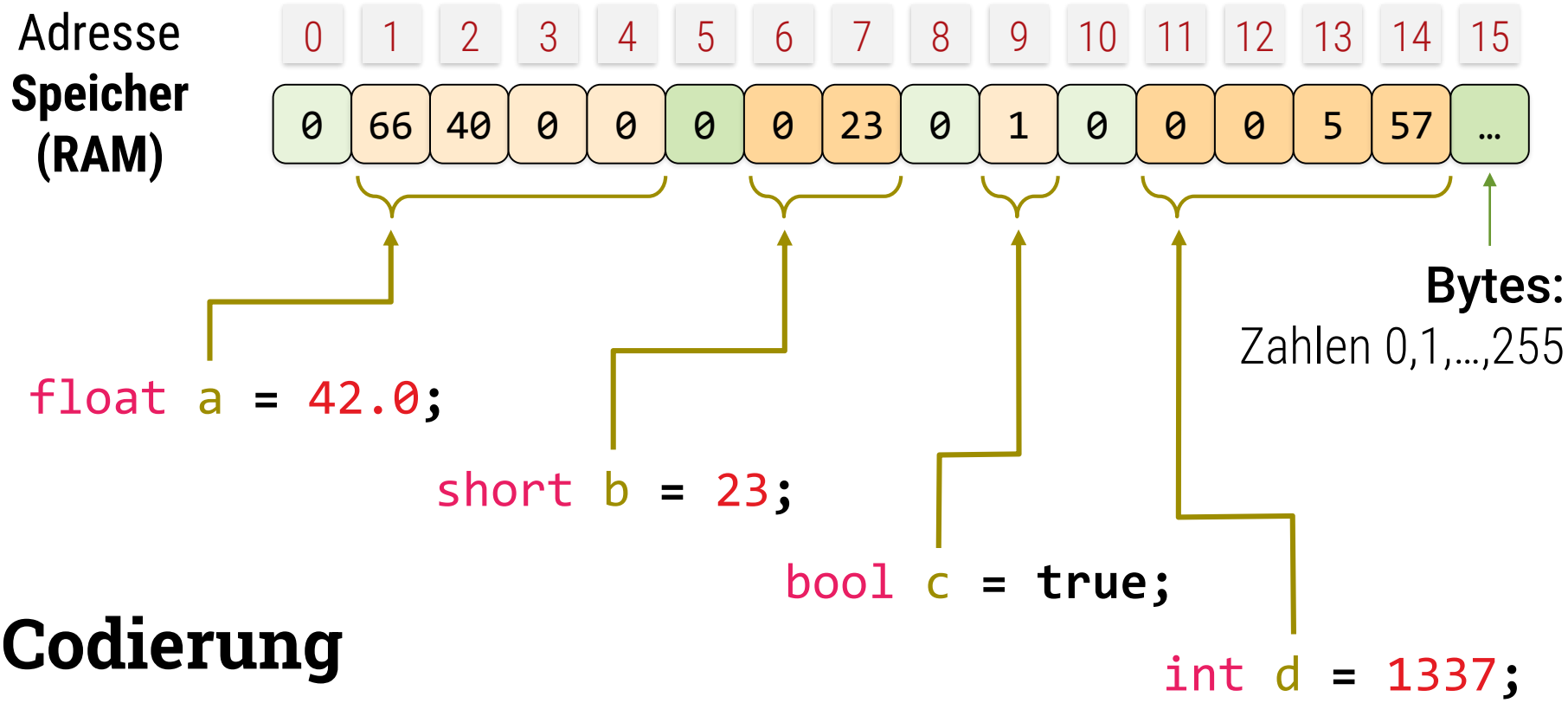
```
// Referenzzarameter <Typ>& werden als Referenzen übergeben
// Erlaubt Ein- und Ausgabe (beides möglich)
// Äußere Variable wird an Stelle von result verwendet (und ggf. verändert)
void addNumbers(const int a, const int b, int &result) {
    result = a + b;
}
void main() {
    int theResult;
    addNumbers(1, 2, theResult); // letztes Arg. muss eine Variable sein!
    cout << "1+2 ist " << theResult;
    addNumbers(1, 2, 3); // Compilerfehler! Referenzparameter keine Variable
}
```

Zeiger

Ein abstrakter Computer



Maschinenrepräsentation



Codierung

- Aufeinanderfolgende Bytes
- Variable = Zeiger auf erstes Byte (Länge implizit)
- Wertsemantik in C/C++: Variable „ist“ Speicherbereich

Zeiger in C++

Zeiger:

- Integraler Bestandteil der Sprache
- Wichtig für Effizienz und Maschinennähe
- Auch wichtig für OOP („Identity Objects“ wie in Python)
  - Gleicher Code für verschiedene Typen: Zeiger, nicht Werte

Drei Regeln

- **&**`<var>`: Addresse (Zeiger auf) `<var>`
- ***`<var>`**: Inhalt von Zeiger `<var>`
- `<typ>`***: Zeigertyp auf `<typ>`

nicht verwechseln mit **Referenzen <Typ> &**

Zeiger in C/C++

Beispiele für Zeiger

```
int a;
int *b; // Man schreibt den Stern an die Variable (Konvention weil: int *a,b nur ein Zeiger)

a = 42;
b = &a;

cout << a;  // Ergebnis 42
cout << *b; // Ergebnis ist auch 42

a = 23;

cout << a;  // Ergebnis 23
cout << *b; // Ergebnis ist auch 23

double *c = &a; // Typfehler! (Beim Übersetzen; zeigt nicht auf double)

uint8_t *c = (uint8_t*)&a; // Explizite Typkonvertierung (use at your own risk)

cout << *(c) <<" "<< *(c+1) <<" "<< *(c+2) <<" "<< *(c+3);
// ↑ „Pointerartithmetik“ – Erlaubt, aber evtl. riskant (z.B. crash falls int kleiner als 32 Bit) ↑
// Für 32Bit ints wird die Kodierung in einzelnen Bytes im Speicher ausgegeben (try it)
```

Referenzen (nur C++) vs. Zeiger

Beispiele für Zeiger

```
int a = 42;
int *b = nullptr;  // Zeigervariable, Initialisierung optional

b = &a;   // Zeiger auf a an Zeigervariable b zuweisen
*b = 23;  // a indirekt verändern

int &c = a;  // Referenzvariable, Initialisierung vorgeschrieben!
c = 1337;    // a indirekt verändern
```

Referenzen

- Initialisierung wird erzwungen
  - „Non-nullable“ Pointer
  - Erzeugt „Alias“ – gleicher Speicherplatz
  - Auch nicht 100% sicher; Aliase auf dyn. allokierten Speicher erlaubt

Wege ins Nirvana

Zeiger ins Nichts

```
int *iWillCrashSoHard() {
    int localVariable = 42;
    return &localVariable; // WTF!
}

int &goingSouth() {
    int localVariable = 42;
    return localVariable; // subtil, aber auch tödlich
}

int *deadRef = iWillCrashSoHard();     // bis hierher noch ok
*deadRef = 23;                         // NOPE. Das geht schief.

goingSouth() = 23;                     // Syntaktisch richtig, aber tödlich
```

Manuelle Speicherverwaltung erfordert Sorgfalt…

Wege ins Nirvana

Zeiger ins Nichts

```
int *iWillCrashSoHard() {
    int localVariable = 42;
    return &localVariable; /
}

int &goingSouth() {
    int localVariable = 42;
    return localVariable; //
}

int *deadRef = iWillCrashSoH
*deadRef = 23;

goingSouth() = 23;
```

MSVC 2013
warning: C4172: returning address of local variable or temporary

MinGW (gcc) 4.9.2
warning: address of local variable 'localVariable' returned

MSVC 2013
warning: C4172: returning address of local variable or temporary

MinGW (gcc) 4.9.2
warning: reference to local variable 'localVariable' returned

Manuelle Speicherverwaltung erfordert Sorgfalt…

Regeln für Zeiger

Null-Zeiger

- Alle Variablen anfangs uninitialisiert
  - Auch Zeiger: Beliebige Adresse – sehr gefährlich!
- Null Zeiger (entspr. in etwa „**None**“ in Python):
  - C: `int *a = 0;` // das passiert wirklich!
  - C++98: `int *a = NULL;` // NULL ist 0 (eine Konstante)
  - C++11: `int *a = nullptr;` // moderne Version (empfohlen)
  - Alles abwärtskompatibel!
- Lesen/Schreiben auf Null-Pointer stürzt auch ab
  - Aber weniger schlimm, die meisten OS erkennen dies und beenden das Programm

Regeln für Zeiger

Operationen auf Zeigern

- „*****“ Dereferenzierung (von links)
- Arithmetik: **+**, **-**, **++**, **--**, **+=**, **-=** erlaubt
- Achtung: „Bezugstyp“
- Zeiger vom typ **<Typ*>** wird um **sizeof(Typ)** Bytes verschoben:

```
uint32_t *a = …;
uint8_t  *b = …;
double   *c = …;

a += 1; // erhöht Adresse um 4 Bytes (32Bit-Zahl)
b += 1; // erhöht Adresse um 1 Byte
b += 7; // erhöht Adresse um 7*8=56 Bytes (64Bit-double)
```

Der Zeiger ins Nichts…

Für den Notfall: void*

- Zeiger auf Typ „`void`“ ist erlaubt
  - „`void*`“ kann auf irgendetwas zeigen
- Zuweisungskompatibel zu allen Zeigertypen
  - `<Typ>*` zu `void*` aber nicht umgekehrt (cast erforderlich)
  - Dereferenzierung nicht möglich
- Bezugstyp für `void*` Zeigerarithmetik (+/-) ist ein „Maschinenwort“ (typ. 4 Bytes; Compiler-abhängig)
  - Empfehlung: Konvertierung in `uint8_t*` (dann ist es klar)

Const-Zeiger

Const-Schlüsselwort

- Veränderlicher Zeiger auf veränderliches Objekt
  ```
  MyClass* ptr = …;
  ```
- Veränderlicher Zeiger auf unveränderliches Objekt
  ```
  const MyClass* ptr = …;
  ```
- Unveränderlicher Zeiger auf veränderliches Objekt
  ```
  MyClass* const ptr = …;
  ```
- Unveränderlicher Zeiger auf unveränderliches Objekt
  ```
  const MyClass* const ptr = …;
  ```

Das taucht in Scala wieder auf

- C++-Syntax egal, aber logische Unterscheidung wichtig!

Zeiger: Wo gibt‘s Speicher?

(Manuelle Speicherverwaltung)

Dynamische Allokation (C/alt)

```
#include <stdlib.h> // alte C-Bibliothek für malloc, free, …

int *a = malloc(sizeof(int)); // Speicher ein int

*a = 42;
cout << (*a);

free(a); // Speicher freigeben!
```

Speicher reservieren in C (alt)

- `void *pointer = malloc(<size in bytes>);`
  - Reserviert Anzahl Bytes im Speicher
  - Rückgabetyp „**`void*`**“ (Kompatibel zu allen Zeigertypen)
- `free(pointer);`
  - Gibt Speicher wieder frei (manueller Aufruf nötig!)

Dynamische Allokation (C/alt)

```
#include <stdlib.h> // alte C-Bibliothek für malloc, free, …

int *a = malloc(sizeof(int)*42); // Speicher für 42 ints

for (int count = 0; count < 42; count++) {
   *(a+count) = count;
}

for (int count = 0; count < 42; count++) {
   cout << *(a+count);
}

free(a); // Speicher freigeben!
```

Neue Schreibweise (C++)

```
int *a = new int(); // Speicher ein int

*a = 42;
cout << (*a);

delete a; // Speicher freigeben!
```

Speicher reservieren in C++ (neu)

- `<Typ> *pointer = new <Typ>;`
  - Reserviert `sizeof(<Typ>)` Bytes im Speicher
  - Rückgabetyp „**`<Typ>*`“ (sicherer!)
- `delete pointer;`
  - Gibt Speicher wieder frei (manueller Aufruf nötig!)

Neue Schreibweise (C++)

```
int *a = new int[42]; // Speicher für 42 ints

for (int count = 0; count < 42; count++) {
   *(a+count) = count;
}

for (int count = 0; count < 42; count++) {
   cout << *(a+count);
}

delete[] a; // Speicher freigeben! (Klammern „[]“ wichtig!!!)
```

Array Allokation

- **new** <Typ>**[**<Zahl>**]** reserviert <Zahl> Einheiten
- **delete[]** <Zeiger> gibt Array frei (**[ ]** wichtig!)

Low-level Arrays = Pointer

```
int *a = new int[42]; // Speicher für 42 ints

for (int count = 0; count < 42; count++) {
   *(a+count) = count;
}

for (int count = 0; count < 42; count++) {
   cout << *(a+count);
}

delete[] a; // Speicher freigeben! (Klammern „[]“ wichtig!!!)
```

`vector<>` ist sicherer und einfacher!

Array Allokation

- **new** `<Typ>[<Zahl>]` reserviert `<Zahl>` Einheiten
- **delete[]** `<Zeiger>` gibt Array frei (**[]** wichtig!)

Arrays fester Größe

```
int *a  = …; // Zeiger auf Speicher
int a[8]; // Array fester Größe – Speicher wird reserviert!
int a[8] = {16,23,32,42,56,64,72,83}; // Initialisierung möglich
*(a+2) += 1; // Array sind weiterhin Zeiger!
a[2] += 1;    // Äquivalente Schreibweise
int *b  = a; // Zeiger wird kopiert, nicht Inhalt!
int b[8]; *b = *a; // Achtung: kopiert nur die erste Zahl! (unerwartet: Bezugstyp int)
```

Arrays fester Größe

- Syntax: `<Typ> <var>[<Größe>];`
- Speicher wird mit angelegt
- Falle: Bezugstype ist Elementtyp (keine ganzen Arrays)

Zeiger auf Strukturen

Zugriff auf Strukturen via Zeiger

- Beispiel

```
struct A {
    int a; int b;
    A *next;
};
A* ptr = new A();
```

- Zugriff via `(*ptr).a` umständlich
  - Vor allem bei tieferer Schachtelung:
    `(*(*(*ptr).next).next).a`
- Alternativ: `(*ptr).a` ≡ `ptr->a`
- Pfeiloperator „`->`“ dereferenziert und wählt Member aus

Strings

Wo sind die Strings?

- Lange Geschichte…
- Mindestens drei Möglichkeiten

Stringtypen

- „C-Strings“
- `std::string`
- Bibliotheken (z.B. QT `QString`)

C-Strings

C-Strings

- Stringtyp: `char*`
- Zeiger auf Array von Zeichen (i.d.R. Bytes)
- Länge? Nullterminiert
  - Letztes Zeichen ist das 0-Byte
  - Reine Konvention
  - Länge immer ein Byte mehr
  - Geht gerne schief…

std::string

Standard-Strings

- Klasse `std::string`
  - Array von Bytes
  - Möglichst UTF-8 Kodierung verwenden (ASCII möglich)
- Überladene Operatoren
  - Verhält sich wie eine Wert (Zuweisung, Vergleich, etc.)
  - Wir bauen das gleich nach!
- Konvertierung zu anderen Libraries via Methode „`c_str()`“
  - Gibt einen Zeiger auf Zeichen (Typ `char*`) zurück (Array)
  - Achtung – Zeiger auf internen Puffer (mit String gelöscht!)

Diskussion: Programmiersprache

Vorlage zur Diskussion

- Daten + Variablen
- Ausdrücke und Berechnungen
- Befehle
- Abstraktionen
- Systemumgebung

Wichtige Sprachfeatures

Abstraktionen (Details: später)

- Funktionen (auch höherer Ordnung und F.‘ Ref. in Variablen)
- Klassen und Objekte
  - C++: Polymorphie durch Subtyping
  - Python: Polymorphie durch Duck-Typing
- Generics (Parametrische Polymorphie): „templates“
  - Der selbe Code für verschiedene Typen
  - Kann auch für Metaprogrammierung genutzt werden
    - Code automatisch erzeugen
- Sichere Kapselung von Low-Level Konstrukten (z.B. „RAII“)
- Andere Features wie überladene (selbstdefinierte) Operatoren, Module (oder sowas ähnliches) etc.

Diskussion: Programmiersprache

Vorlage zur Diskussion

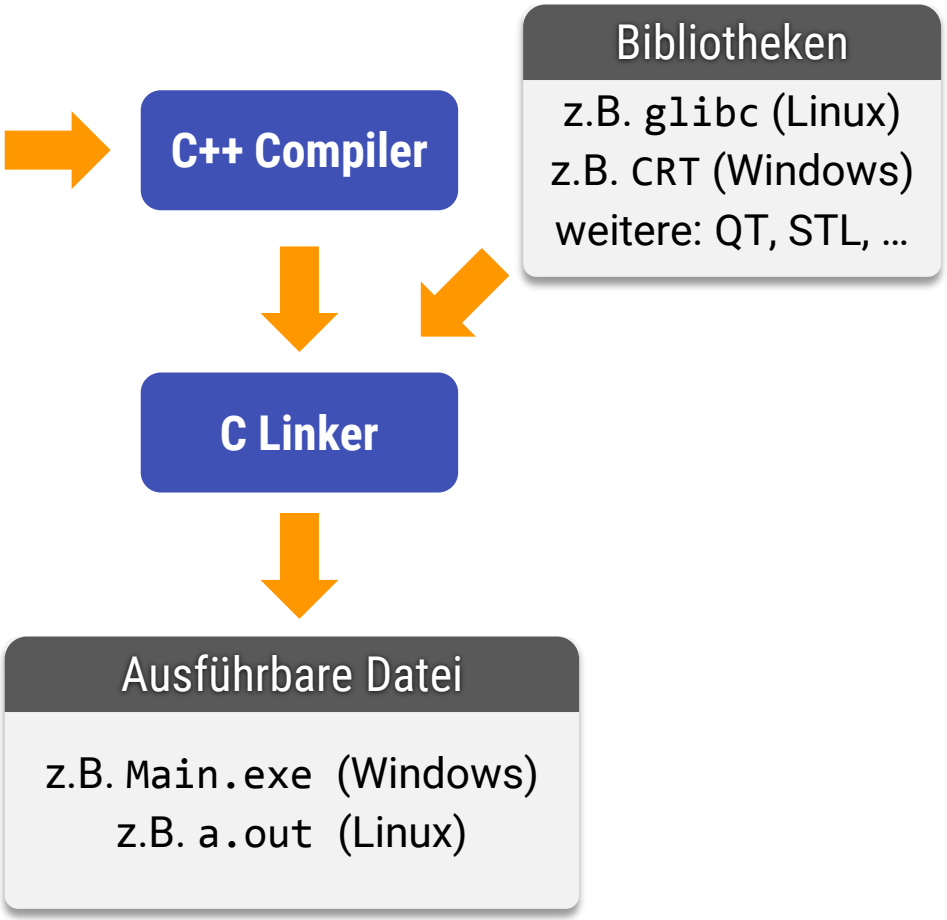
- Daten + Variablen
- Ausdrücke und Berechnungen
- Befehle
- Abstraktionen
- Systemumgebung

Übersetzung: Einzelne Datei

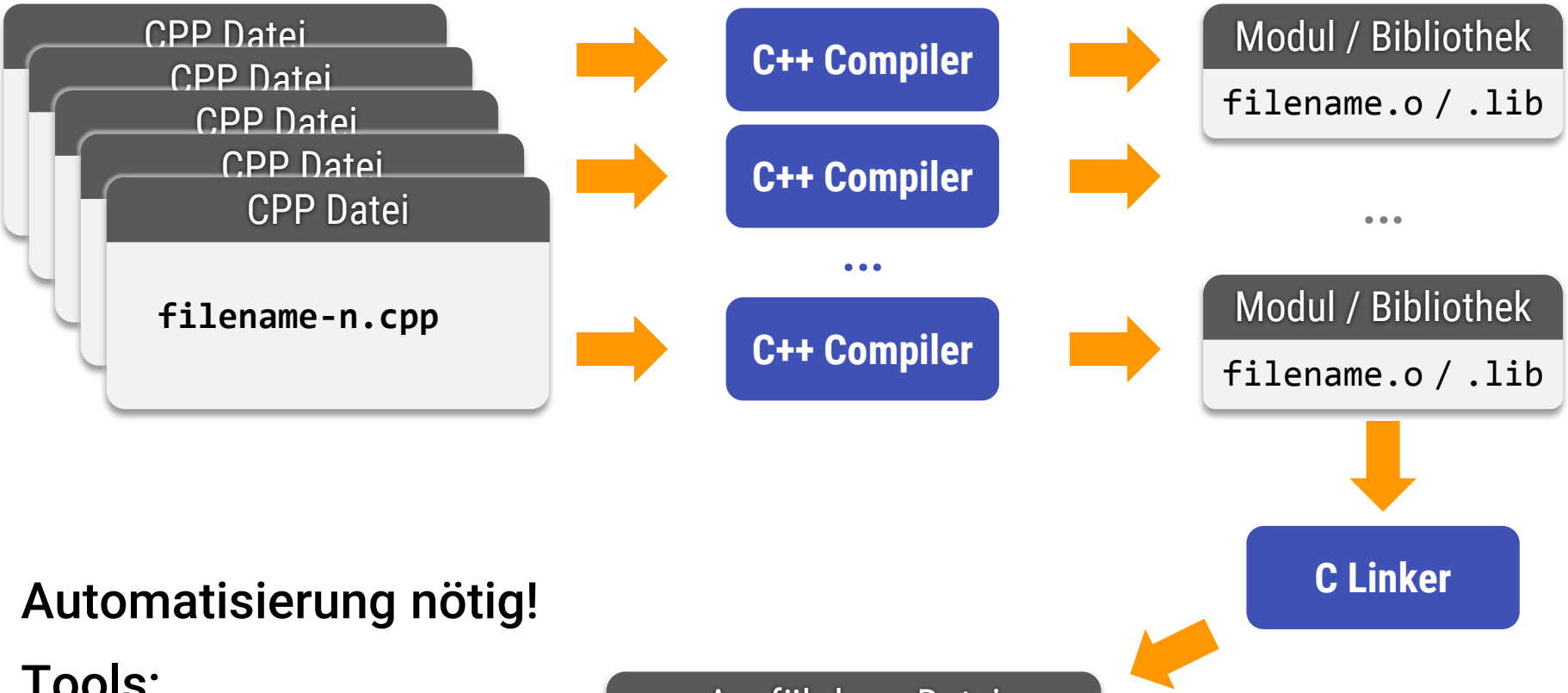
Datei: main.cpp

```
#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)

Module & Bibliothken:
Separate Übersetzung

Prinzipien

Zwei Teile einer Bibliothek

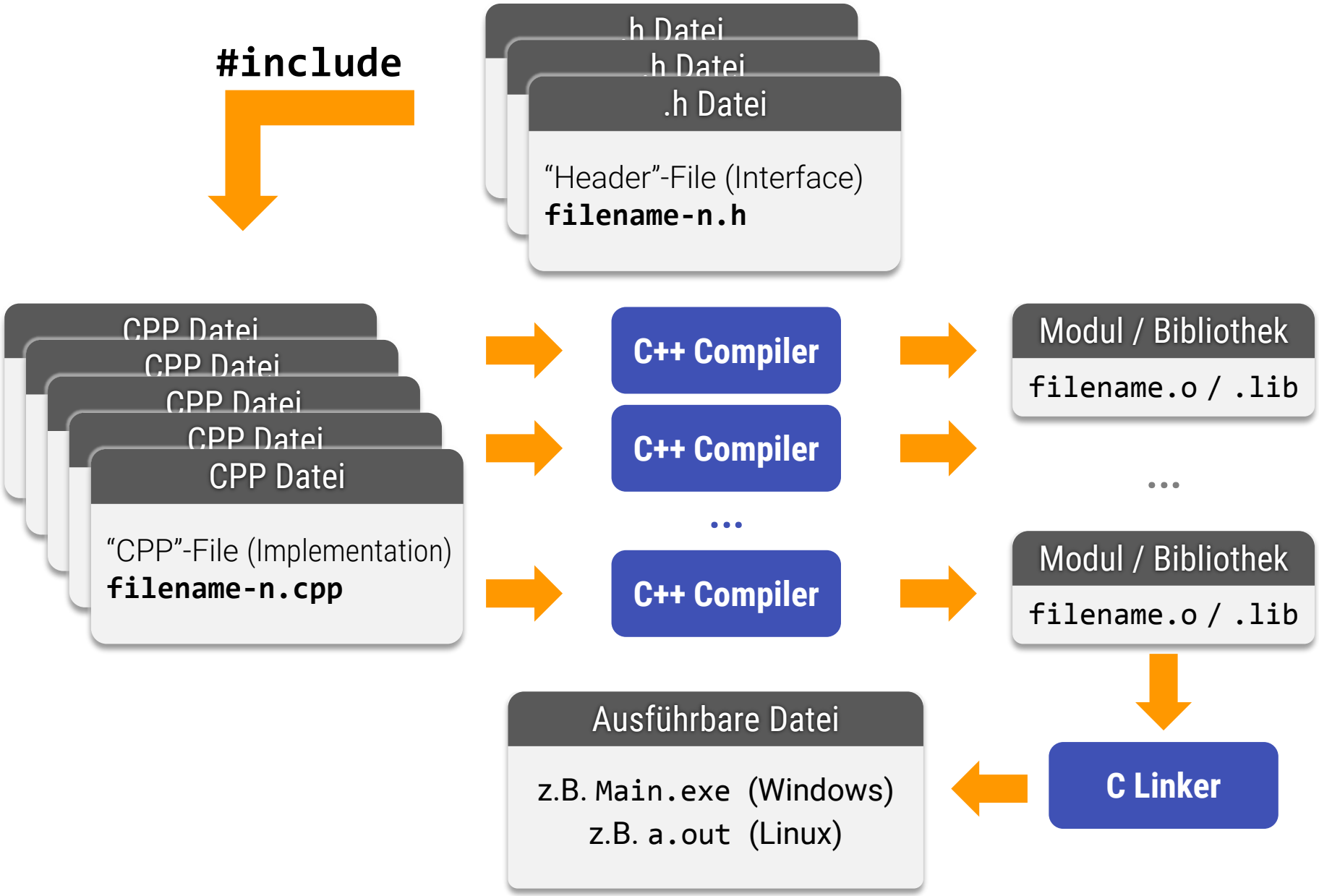
- **Interface („Schnittstelle“)**
  Alle Informationen, die zum Verwenden einer Bibliothek gebraucht werden
  - Signaturen von Funktionen
  - Typinformationen (teilweise)
    - Name, Membersignaturen, ggF. Details
- **Implementation**
  Weitere Informationen, die zum Übersezten/Erstellen der Bibliothek selbst gebraucht werden
  - Code aller Funktionen
  - Genaue Typinformationen

Module

C/C++ (vor C++ 20) hat(te) keine Module

- Simuliert wie folgt
  - Sourcecode wird in Schnittstellen (*.h) und Code (*.cpp) aufgeteilt
  - `#include` fügt Schnittstellencode ein (copy-paste)
  - *.cpp Dateien werden getrennt übersetzt in *.obj-Dateien
  - Beim „Linken“ werden alle *.obj Dateien zusammengefaßt
- Berüchtigt für geringe Effizienz, Fehleranfälligkeit; schwer zu benutzen
  - Für unsere Übungen unwichtig, fast nur Std-Bibliothek
- Seit C++20 gibt es Module
  - In der Praxis wohl immer noch etwas holprig (Stand 2024/25)

(Historische) C/C++ Module...



Wie man es richtig macht

Modulkonzept für separate Übersetzung

- Interface als eine Datei
- Implementation, typ. als gesonderte Datei
- Beides wird übersetzt
  - Übersetztes Interface:
    Für Benutzung durch Compiler
  - Übersetzte Implementation:
    Für Linker (Code zum Zusammenbauen)

Allgemeines Prinzip

- Erfunden in den 1970ern (z.B. MESA, MODULA-2)

Beispiel (Modula-2)

Datei: IntegerLists.def

```
DEFINITION MODULE IntegerLists;

TYPE IntList; (* Opaker Typ – Details nicht öffentlich bekannt *)

FUNCTION createList() : POINTER TO IntList;
EXPORTS createList, IntList;

END.
```

Compiled Interface

IntegerLists.sym

Datei: IntegerLists.mod

```
MODULE IntegerLists;

FROM Storage IMPORT ALLOCATE, DEALLOCATE;

TYPE IntList = RECORD
   memory: POINTER TO Integer;
   size:   Integer;
END;

FUNCTION createList() : POINTER TO IntList;
VAR result: POINTER TO IntList;
BEGIN
   ALLOCATE(result, TSIZE(IntList));
   result^.size = 0; result^.memory = NIL;
   RETURN result;
END;

BEGIN END. (* Hauptprogramm ist hier leer *)
```

Compiled Implementation

IntegerLists.obj

Übersetzung in Modula-2

Wenn Modul benutzt wird

- `.sym` Dateien aller benutzter Bibliotheken werden benötigt

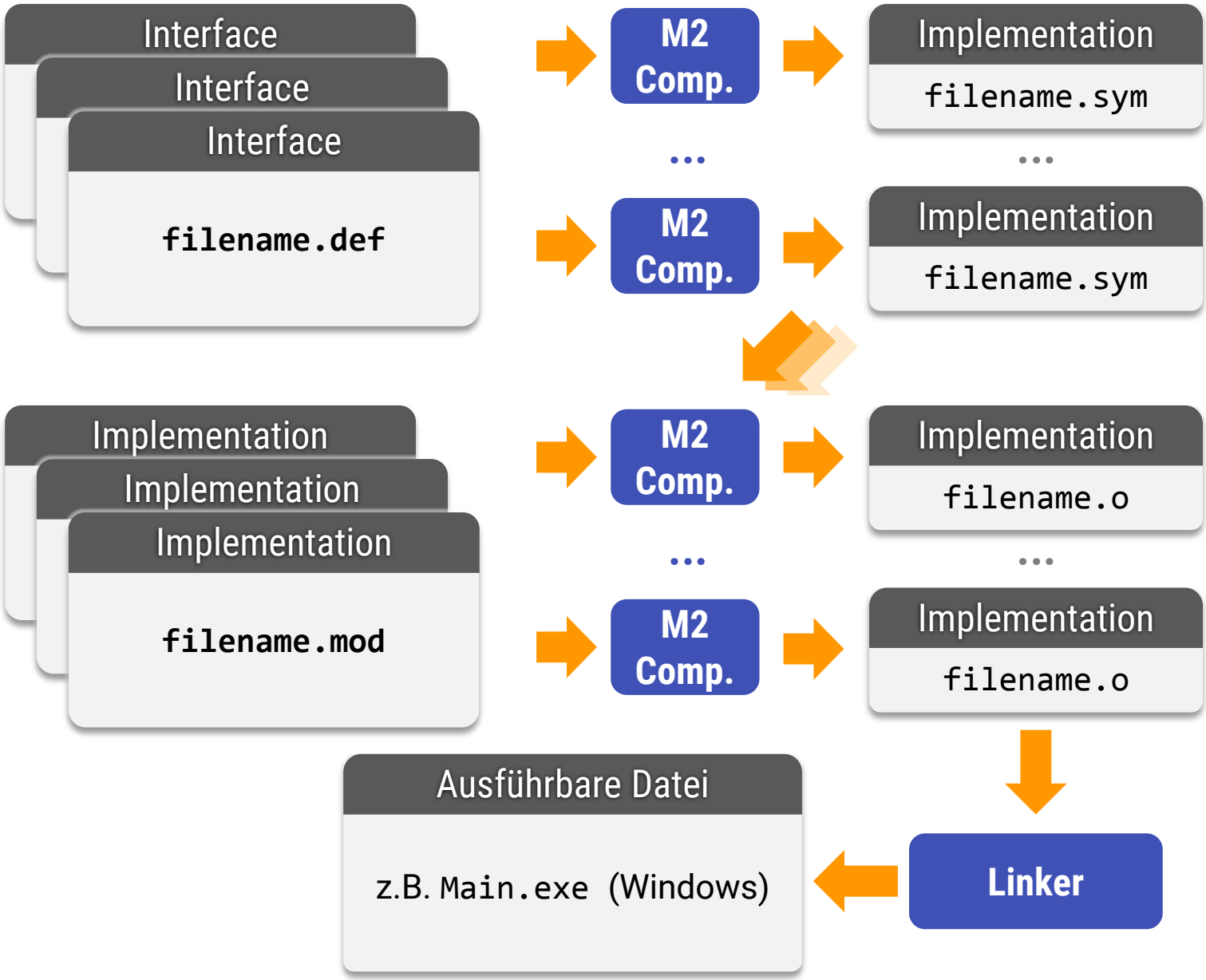
Zum Linken des gesamten Programms

- Alle `.obj` Dateien werden benötigt

Jedes Modul wird getrennt übersetzt

- (sehr!) schnell
- Saubere Trennung von Schnittstelle & Implementation
- Keine Seiteneffekte, expliziter Export/Import

Übersetzung Komplexer Systeme



JAVA / Python

Andere (moderne) Sprachen

- Java/Scala, Python, Borland / Object Pascal
- Eine Datei für Interface und Implementation
  - Bei Pascal mit zwei Abschnitten
  - Bei Python und Java nur volle Implementation
- Compiler übersetzt Interfaceanteil und Implementation ggf. automatisch in getrennte Dateien

Vorteil

- Weniger Tipparbeit, trotzdem schnelle Übersetzung

Nachteil

- Interface nicht (so) sauber getrennt