Einführung in die Softwareentwicklung

SOMMERSEMESTER 2020

Foliensatz #08

Module und der Präprozessor

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

Der C/C++ Preprozessor

Der Präprozessor

Der Präprocessor – ein Relikt aus C

- Der Präprozessor ersetzt Text in der Eingabedatei
  - Reine Textebene („Tokens“)
    - Namespaces und Gültigkeitsbereiche werden ignoriert
    - Vorrang vor allen C/C++ Regeln
  - Benutzt für einfache „Metaprogrammierung“
    - Programmtext selbst kann „umprogrammiert“ werden
- Ursprünglich: separater Übersetzungsschritt
  - Heute im Compiler eingebaut, läuft automatisch.
  - Manueller Aufruf möglich (hier nur zur Illustration):
    - GNU C Compiler: `gcc -E in.c -o out.i`
    - Microsoft MSVC: `CL /P file.c` (Ausgabe: `file.i`)

Der Präprozessor

Der Präprocessor – ein Relikt aus C

- Drei Funktionen
  - Makros ersetzen („`#define`“)
  - Bedingte Übersetzung („`#ifdef`“)
  - Einfügen anderer Dateien („`#include`“)
- Anwendungshinweis
  - Makros sind in C++ weniger wichtig
    - Vermeiden, wann immer möglich
  - Der Rest ist leider immer noch unvermeidbar

Präprozessor #1

„**#define**“

Makros

Beispiele für Makros

```
// Syntax: NAME wird ersetzt durch Text
// Konvention: Makros in Großbuchstaben
#define <NAME> <Text>


// Hier eine beliebte Anwendung (C Standard-Bibliothek definiert so „M_PI“)
#define PI 3.14159265358979323846264338327795
double x = sin(PI*2);

// Eine andere Anwendung
#define NUM_RUNS 128
for (int i = 0; i < NUM_RUNS; i++) {doStuff(i);}

// Eine andere Anwendung
#define ERROR_MSG {cout << "something went wrong!\n";}
if (i != 42) {
   ERROR_MSG
   throw SomeException();
}
```

Makros

Beispiele für Makros

```
// Syntax 2: Parameter in Klammern
#define <NAME>(param1, param2, …) <Text>

// Hier eine beliebte aber sehr gefährliche Anwendung
#define MAX(a,b) (((a)>(b))?(a):(b)) // Klammern optional – nur zur Sicherheit!
double x = MAX(23.0, 42.0);

// viele C Bibliotheken enthalten #define min()…, #define max() …
// Eigener Code mit Bezeichner min, max compiliert dann nicht mehr
// Das gilt sogar für viele C++ Standardbibliotheken…

// Eine andere Anwendung (aus unserer eigenen GeoX-Bibliothek): Codegenerierung
#define MODULE_REG(moduleName) \   // „Backslash“: Zeilenumbruch Teil des Makros
   void init_##moduleName () { \      // „##“: Parameter mit Text zu einem Token
      register_##moduleName(1); \                                  //  verschmelzen
   }; \
   void shutdown_##moduleName () { \
      register_##moduleName(0); \
   };
```

Makros – WCPGW?

What could possibly go wrong?

```
// Beliebter Fehler
#define PI 3.1415

// Merkwürdiger Compilerfehler!
void trigometriy(const double PI) {…}
// Erzeugt Code: void trigometriy(const double 3.1415) {…}
// Compiler is not amused. Fehler oft schwer zu finden!

// Das funktioniert, da Ersetzung „Tokenweise“ (ganze Wörter)
void calc_PI() {…}
```

Textuelle Ersetzung

- Gefährlich – Text wird einfach ersetzt
  - „Token“-weise: Getrennt durch Whitespaces oder Sonderzeichen
- Ersetzung greift ab `#define`
  - Gültigkeitsbereiche (`{}`, `namespace` etc.) wirkungslos

In C++: Makros vermeiden

Makros vermeiden

```
// Bessere Konstanten (sicherer, aber nur in C++ möglich)
const double PI = 3.14159265358979323846264338327795;

// Kein Compilerfehler mehr – voller Scope & Type-Check!
void trigometriy(const double PI) {…}


// Bessere Code-Makros (sicherer, genauso schnell)
inline double max(double a, double b) {return (a>b) ? a : b;}


// Generische Version (beliebige Typen) auch möglich!
// Compiler prüft, dass die beiden Typen für a, b gleich sind!
// Kein Geschwindigkeitsunterschied zum Makro zur Laufzeit!
template <typename T>
inline T max(T a, T b) {return (a>b) ? a : b;}
```

Präprozessor #2

„**#ifdef**“

Bedingte Compilierung

Syntax für bedingte Compilierung

```
// Syntax: #ifdef, #else, #endif
#ifdef <NAME>
  <code> // Falls NAME definiert ist

#else // optional!

  <code> // Falls NAME nicht definiert ist

#endif


// Syntax: #ifdef, #else, #endif
#ifndef <NAME>

  <code> // Falls NAME nicht definiert ist

#endif

// Setzen & löschen des „boolschen“ Makros (auch per Compiler-Option möglich)
#define <NAME> // ab hier sicher definiert
#undef <NAME>  // ab hier sicher nicht mehr definiert
```

Bedingte Compilierung

Beispiel für bedingte Compilierung

```
// Debugging ein/ausschalten
void zero_array(vector<double> a, unsigned numZeros)

   #ifdef _DEBUG  // Die meisten Compiler setzen dies im Debug Modus

      if (numZeros >= a.size()) {
         throw OutOfRangeException("That almost went wrong!");
      }

   #endif

   for (unsigned i=0; i<a.size(); a++) {
      a[i] = 0.0;
   }
}
```

Code nur für „Debug“-Modus

- Nützlich: Stärkere Prüfungen zur Fehlersuche
- Volle Geschwindigkeit bei „Release“-Build

Bedingte Compilierung

Beispiel für bedingte Compilierung

```
#define USING_64_BIT_CPU // switch to 64-Bit Code

void fast_zero_memory(uint8_t *buffer, unsigned size) {
   #ifdef USING_64_BIT_CPU
      uint64_t i;
      for (i = 0; i<size/8; i++) {
         *buffer[i] = 0;
      }
      uint64_t rest = size % 8;
   #else
      uint32_t i;
      for (i = 0; i<size/4; i++) {
         *buffer[i] = 0;
      }
      uint32_t rest = size % 4;
   #endif
   for (unsigned j=0; j<rest; j++) {
      buffer[i+j] = 0;
   }
}
```

Bedingte Compilierung

Beispiel für bedingte Compilierung

```
#define USING_AVX2  // switch to vectorized code using Intel AVX-2 Extensions

void fast_zero_memory(uint8_t *buffer, unsigned size) {
   #ifdef USING_AVX2
      uint64_t i; const __m256i ZERO = _mm256_set_epi64(0,0,0,0,0,0,0,0);
      for (i = 0; i<size/32; i++) {
         *((__m256i*)buffer)[i] = ZERO;
      }
      uint64_t rest = size % 32;
   #else
      uint64_t i;
      for (i = 0; i<size/8; i++) {
         *buffer[i] = 0;
      }
      uint64_t rest = size % 8;
   #endif
   for (unsigned j=0; j<rest; j++) {
      buffer[i+j] = 0;
   }
}
```

Anmerkung:

Auf modernen CPUs macht 32/64Bit Zugriff keinen oder kaum einen Unterschied.

Optimierungen dieser Art sind aber für Vektorisierung mit AVX o.ä. nützlich.

Präprozessor #3

„**#include**“

#include

Datei: definitions.h

```
#define M_PI 3.14159265358979323846264338327795

inline double max(double a, double b) {
   return (a>b) ? a : b;
}

void printHelloWorld() {
   std::cout << "Hello World!";
}
```

Datei: main.cpp

```
#include <iostream>       // <…> = Look in path with standard libraries first
#include "definitions.h"  // "…" = Look in current directory first (use for own libraries)

int main() {
   std::cout << M_PI << "\n";
   std::cout << max(23, 42) << "\n";
   printHelloWorld();
}
```

#include

Includes

- `#include "Datei.h"`
- Präprozessor fügt Datei wortlich ein
- Konvention: Dateiname endet mit „`.h`“

Gleiche Wirkung wie abtippen!

- `#define`s, `#ifdef`s, etc. wirken weiter
- Keine Prüfung von Namensräumen
- Rekursive `#include`s möglich (Compiler-Absturz!)

#include

Datei: MyMathTools.h

```
#include "Definitions.h"

inline double max(double a, double b) {
   return (a>b) ? a : b;
}
```

Datei: Definitions.h

```
#include "MyMathTools.h"

#define M_PI 3.14159265358979323846264338327795

#define USING_AVX2
```

Zyklische Abhängigkeiten nicht erlaubt!

- „`#include`“ einer der beiden obigen Dateien in einem CPP-File führt zu Compier-Crash!
- Vermeidung mit „include-guards“ (ja, auch im Jahr 2018…)

#include-Guards

Datei: MyMathTools.h

```
#ifndef MYMATHTOOLS_H // Hope & pray that nobody #defines this symbol elsewhere!
#define MYMATHTOOLS_H // Typo-risk (I always copy & paste this symbol !!)

#include "Definitions.h"

inline double max(double a, double b) {
   return (a>b) ? a : b;
}

#endif // Do not forget!!! (weirdest compiler errors)
```

Datei: Definitions.h

```
#ifndef DEFINITIONS_H
#define DEFINITIONS_H

// Still impossible to include “Defintions.h” cyclically
// It will just not work / do what its seems to do (no error message from compiler. good luck.)

#define M_PI 3.14159265358979323846264338327795
#define USING_AVX2

#endif
```

#pragma once (nicht Standard)

Datei: MyMathTools.h

```
#pragma once // supported by most compilers, but non-standard

#include "Definitions.h"

inline double max(double a, double b) {
   return (a>b) ? a : b;
}
```

Datei: Definitions.h

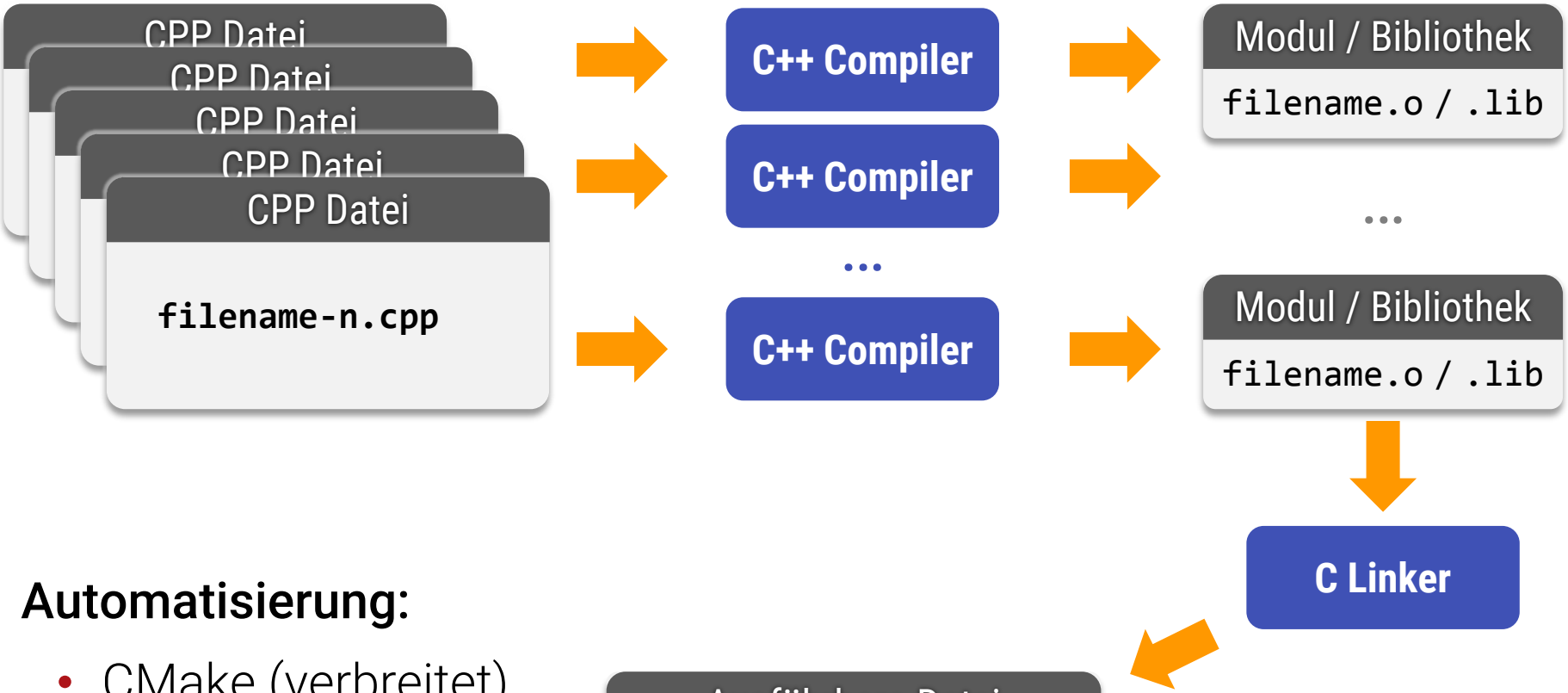
```
#pragma once // supported by most compilers, but non-standard

// Also in this case: still impossible to include “Defintions.h” cyclically
// Same problems as in guarded case.

#define M_PI 3.14159265358979323846264338327795
#define USING_AVX2
```

Erläuterung

- `#pragma` generell: „nicht standartisierte“ Hinweise an Compiler
- `#pragma once` heißt: Datei nur einmal #includen pro Comp.-Lauf

Separate Übersetzung

Übersetzung: Komplexes System



Automatisierung:

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

Hinweis: dynamische Bibliotheken möglich

- DLLs (Windows)
- .so (Linux)

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

Wie man es richtig macht

Sehr einfach

- Interface als eine Datei
- Implementation als gesonderte Datei
- Beides wird übersetzt
  - Übersetztes Interface:
    Für Benutzung durch Compiler
  - Übersetzte Implementation:
    Für Linker (Code zum Zusammenbauen)
- C/C++ kann das leider (noch) nicht :-(

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

Achtung:

In C++ heißen Interfaces „Deklarationen“ und Implementationen „Definitionen“.

Übersetzung

Wenn Modul benutzt wird

- `.sym` Dateien aller benutzter Bibliotheken werden benötigt

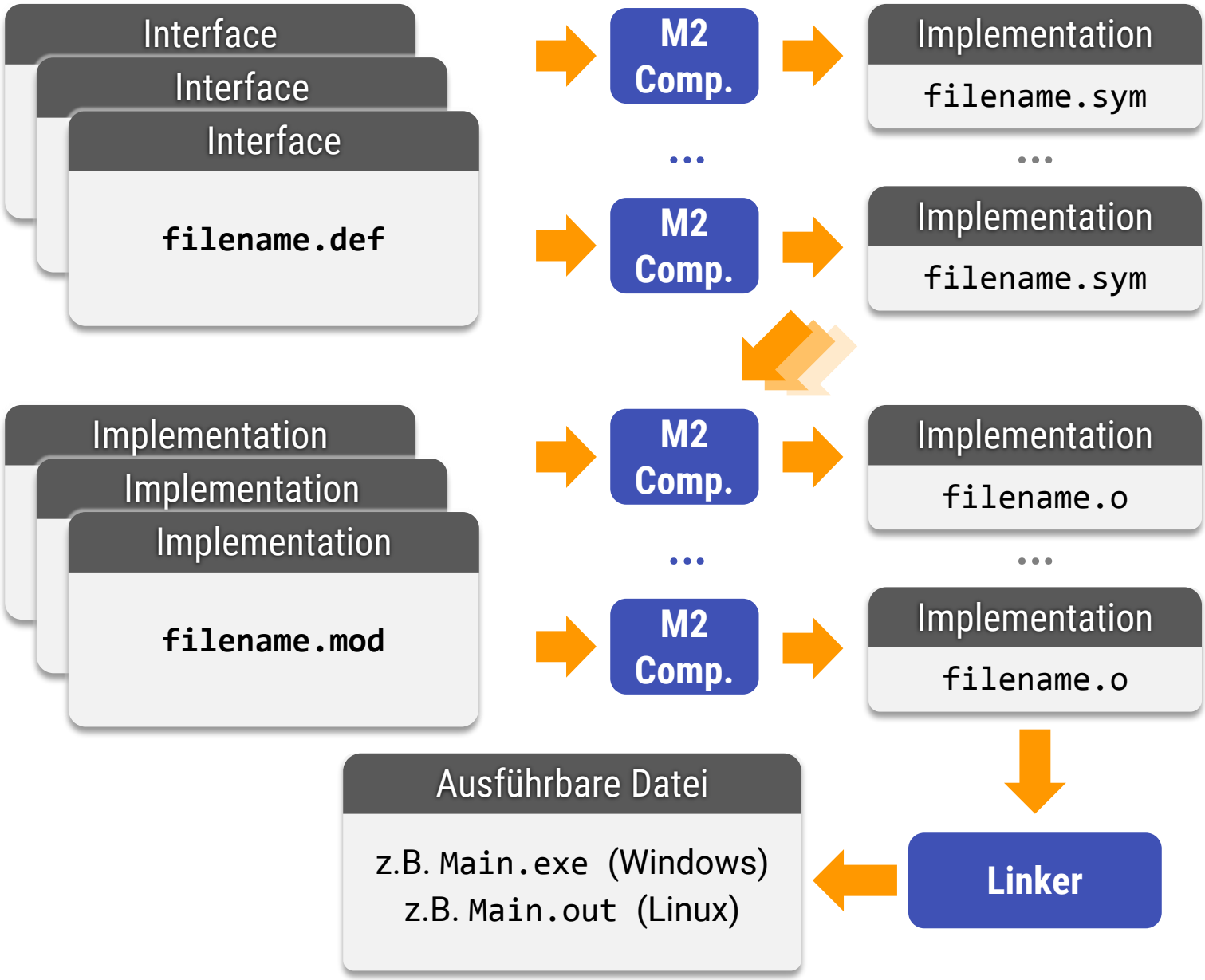
Zum Linken des gesamten Programms

- 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 Sprachen

- Java, 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 automatisch in getrennte Dateien

Vorteil:

- Weniger Tipparbeit, trotzdem schnelle Übersetzung

Nachteil:

- Interface nicht (so) sauber getrennt

Weiterentwicklungen

Java Virtual Machine

- Genormte Module (.class) für alle Platformen und Prozessoren
  - Zwischencode („virtual machine“)
  - Übersetzung in Maschinensprache bei Bedarf (JIT)

Microsoft .net Framework

- Genormte Module wie bei JVM
  - „Assemblies“ in Zwischencode oder Binärcode
  - Übersetzung ahead-of-time oder just-in-time (JIT)
- Versionierung, Metadaten, etc.

Definition vs. Deklaration

Schnittstelle: Deklaration (C++ Sprachgebrauch)

- Wird nur vom Compiler gesehen
- Deklaration von Typen
- Deklaration von Funktionen ohne Code
- Deklaration von globalen Variablen als „extern“

Implementation: Definition (C++ Sprachgebrauch)

- Wird vom Linker geprüft
- Deklaration von Funktionen mit Code
- Globalen Variablen ohne „extern“
- Darf nur 1x auftauchen (sowohl Compiler & Linker!)

„Best Practices“
(es gibt ja keine Module)

Beispiel (C++)

Datei: `MyMathTools.h`

```
#ifndef MYMATHTOOLS_H // Include-guard!
#define MYMATHTOOLS_H

#include <cstdint> // Interfaces anerderer Module – Textuelle Ersetzung!

double max(double a, double b); // Deklaration, keine Definition

uint64_t max(uint64_t a, uint64_t b); // Deklaration, keine Definition

#endif // Include-guard!
```

Wird nicht vorab übersetzt

(bei jeder Benutzung von Neuem)

Datei: `MyMathTools.cpp`

```
#include "MyMathTools.h" // Immer als erstes eigenes .h-File includen!!
#include <stdio> // Interfaces anderer Module, nur für Implementation

double max(double a, double b) { // Definition! (Implementation)
   return (a>b) ? a:b;
}

uint64_t max(uint64_t a, uint64_t b) {…}
```

Compiled Implementation

`IntegerLists.obj`

Deklarationen in C++ (.h Files)

(Wichtige) erlaubte Deklarationen

- // Funktionsdeklaration
```
void someFunction(int a, int b);
```
- // Typdeklaration
```
struct SomeClass {
  int a; int b;
};
```
- // Opake Typdeklaration (Damit nur Zeiger möglich!)
```
struct SomeClass;
```
- // Variablendeklarationen
```
extern int a;
```

Definitionen (.cpp Files)

Dazu passende Definitionen

- // Funktionsdefintion
```
void someFunction(int a, int b) {
  …some code…
}
```
- // Variablendefinitionen
```
int a;
```
- // Mit Wert initialisierte Variablen sind immer Definition
```
int a = 42;
```

Deklarationen von Klassen

Datei: IntList.h

```
// (Include guards dazudenken!)
struct IntList {
    int *memory;
    int size;
    // Konstruktur
    IntList(unsigned initialSize = 0);
    // Destruktor (bei Löschen des Objektes)
    ~IntList(unsigned initialSize = 0);
    // Zugriff auf Elemente
    int &operator[](unsigned index);
};
```

Klassendeklarationen (.h-File)

- Code auslassen!
  - Man kann ihn auch direkt reinschreiben, ist schlechter Stil
  - Probleme mit Abhängigkeiten, außerdem: „implicit inline“

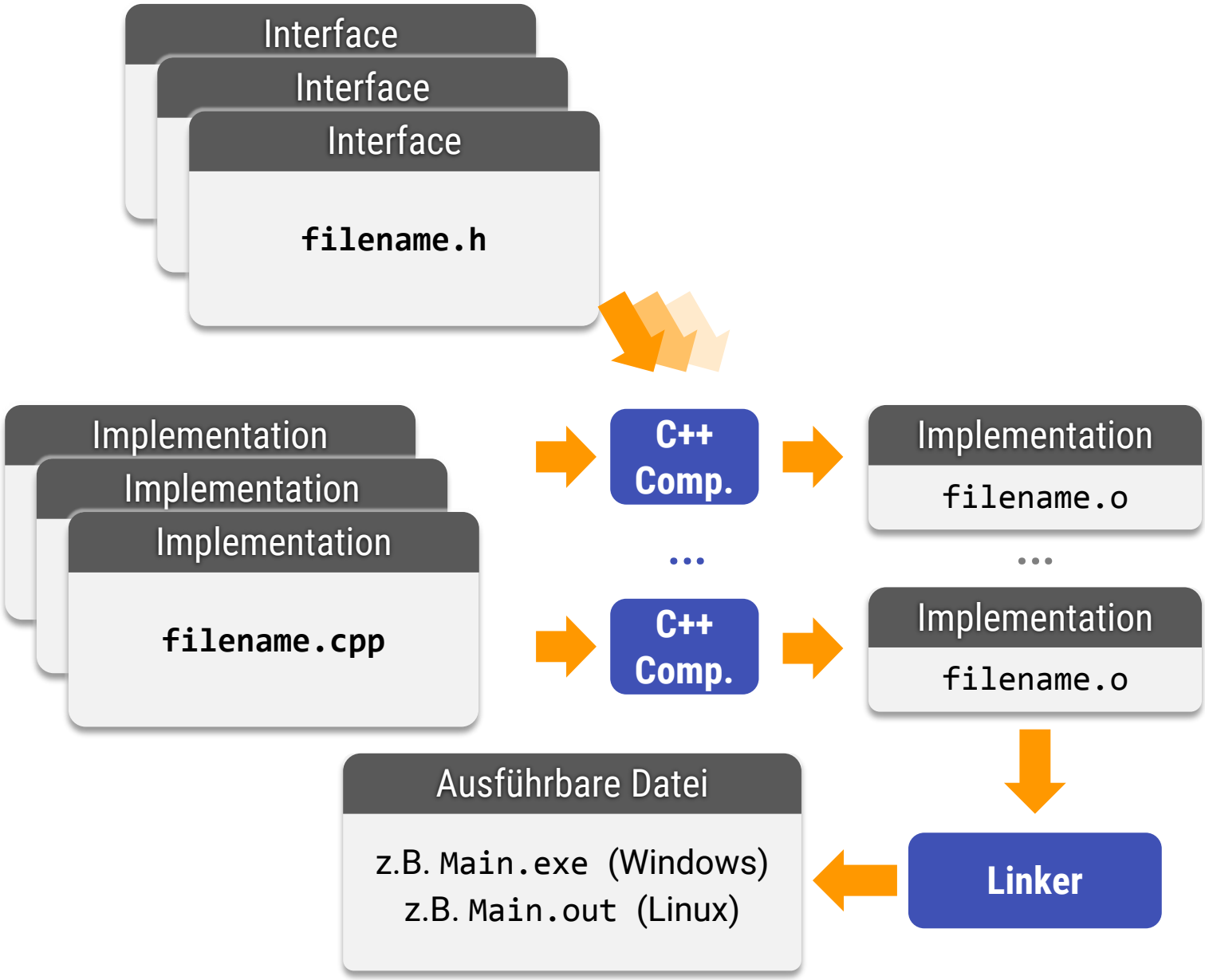
Deklarationen von Klassen

Implementation: IntList.cpp

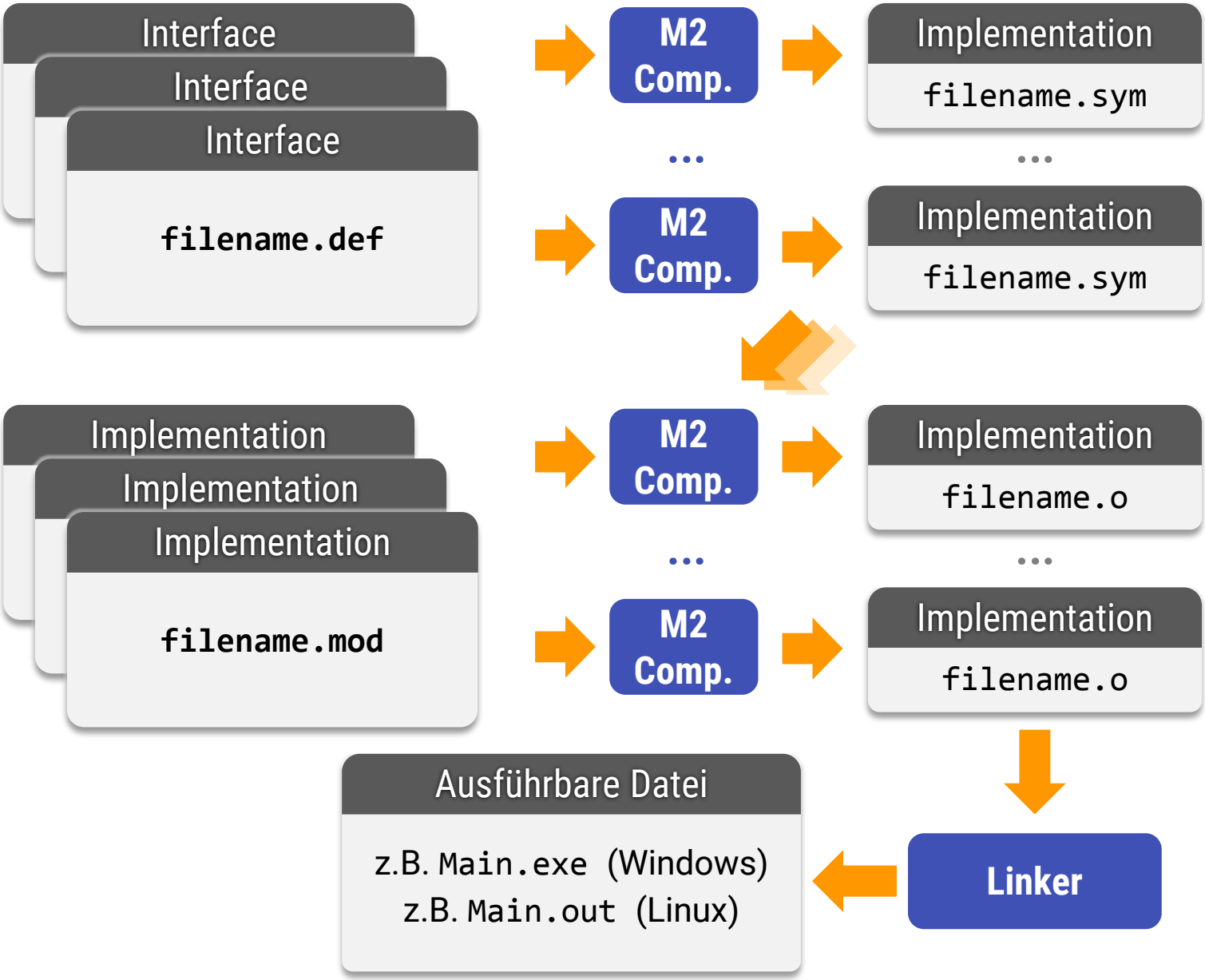
```
IntList::IntList(unsigned initialSize = 0) {
   memory = new int[initialSize]; // Speicher reservieren (0 erlaubt)
   size = initialSize;
}
IntList::~IntList(unsigned initialSize = 0) {
   delete[] memory; // Speicher freigeben
}

int &IntList::operator[](unsigned index) {
   if (index >= size) {
      throw std::out_of_range("out of bounds");
   }
   return memory[index];
}
```

Definitionen getrennt! (.cpp-File)

Übersetzung in C++



Übersetzung in Modula-2 u.ä.



Best Practices

Probleme beim Modulsystem

- C++-Module werden textuell eingefügt
- Durch `#define` können Seiteneffekte auftreten
  - Reihenfolge von `#include` kann eine Rolle spielen
  - Semantik eines Modul ändert sich durch Verwendung eines anderen
  - *<<Schauder>>* – dies sollte eigentlich verboten sein
  - Weil es aber so definiert ist, kann man #includes nicht getrennt übersetzen
- Seiteneffekte unbedingt vermeiden!
  - `#define` sehr sparsam verwenden
  - Global eindeutige Namen (vor allem für include-guards!)

Best Practices

Übersetzungszeiten

- C++ Übersetzt sehr langsam
- Wenn jedes Modul alle anderen #included, steigt der Aufwand quadratisch mit der Projektgröße
- Hauptgrund für lange Übersetzungszeiten!

Milderung

- Abhängigkeiten Minimieren
  - So wenige `#include`s anderer Module wie möglich
- Opake Typdefinition „`struct Typ;`“, „`class Typ;`“ wann immer möglich (statt `#include` der Deklaration)

Best Practices

Zyklische Abhängigkeiten

- Headerfiles können/dürfen nicht zyklisch abhängig sein
- `.cpp`-Files binden nur `.h`-Files ein – Zyklen unmöglich
- Auflösung per Hand
  (z.B. opake Deklaration für Zeigertypen)

inline / templates

Noch eine Schwierigkeit

- inline-Code muss im Header stehen!
- Binarbibliotheken von C++ unterstützen kein inline (da C-kompatibel)
- Das gilt auch für template code aller Art!!!
  - Macht viel Ärger
  - z.B. bei zyklischen Abhängigkeiten (google template-rebinding)

Name Mangeling

Und noch eine… :-)

- Der Linker ist ein C-Linker
  - Linker kennt keine C++-Features
- Daher „Name-Mangeling“ für Überladen + Klassen
  - Namen werden ersetzt für Eindeutigkeit
  - Ersetzungsschema ist nicht „Human-friendly“
  - Linker-Fehlermeldungen oft unverständlich
  - Name Mangeling kann mit
    „`extern "C" {…Deklarationen…}`“
    abgeschaltet werden (auch bei Übersetzung von `.c` files)

(Hätte man doch von Modula-2 gelernt…)

Die Python/C++-Platform

Genormte ABIs für C (nicht C++)

- „`ccall`“ definiert für jede OS+CPU-Kombination
  - Aufruf von Funktionen mit C-features
  - Keine C++-Konstrukte
    - Compiler nutzen C-Abbildung via Name-Mangeling
    - Abbildung nicht genormt
- Statisch (`.lib`) und dynamisch ladbar (`.so`/`.dll`)

Python

- Module im Quelltext
- Interpretation (precompile) bei ersten Laden
- Einbindung von C-Dlls, z.B. „ctypes“(C), „pybind“ (C++)

Aber wie geht es in C++?

C++ kennt keine Module

- Statt dessen: teilweise Übersetzung

Was heißt das?

- Man kann Code weglassen
- Funktionen fehlen dann in Binärdatei
- Linker fügt alles am Ende zusammen

Regel

- Implementation muss genau 1x vorhanden sein
- Sonst Fehler (Linkerfehler / Compilerfehler)

C++ 20 Current Draft

(Ab May '20 gibt es nun doch Module...)

Beispiel (C++20 / CLANG)

Datei: MyMathTools.cppm

```
export module MyMathTools; // Module interface: MyMathTools.cppm

export double max(double a, double b); // Deklaration
export uint64_t max(uint64_t a, uint64_t b); // Deklaration
```

Datei: MyMathTools.cpp

```
module MyMathTools; // Module implementation: MyMathTools.cpp

double max(double a, double b) { // Definition! (Implementation)
   return (a>b) ? a:b;
}
uint64_t max(uint64_t a, uint64_t b) {…}
```

Datei: main.cpp

```
import MyMathTools; // Benutzung z.B. in Hauptprogramm main.cpp

int main() {
   cout << max(23, 42);
}
```

Details tbd.

Fehlende Details

- Genaue Regeln für „`export`“, Interfaces, Implementationen, Teilmodule (parts)
- Kompatibilität mit „Include“
- Übersetzungsprozess

Offene Baustellen

- Compilersupport – noch in Entwicklung
- (Derzeit?) Probleme mit Geschwindigkeit, Parallelität
- C++20 sieht noch kein stabiles ABI (application binary interface) vor – gleiche Compilerversion nötig