Kapitel 11 - Toolchain

Peter Ulbrich

AG Systemsoftware

Veranstaltungswebseite

Einleitung

- Was benötigen wir für diese Programm?

```c
#include <stdio.h>

int main() {
    printf("Hello World!\n");

    return 0;
}
```

- `printf()` wird benötigt, bereitgestellt durch *libc*

  → Bibliotheken, die einen Baukasten von Funktionen anbieten

- Aber: Wie genau geschieht die Bereitstellung?
- Dies wird heute beleuchtet!😀

Relevante Compiler

- Zunächst einmal: Neben GCC gibt es natürlich noch andere Compiler für C/C++

GNU C Compiler (GCC)

- Standard für Linux und UNIX
- Sehr ausgereift, unterstützt auch Nischenarchitekturen
  - Beispiel: Diverse Mikrocontroller von Texas Instruments (z.B. MSP-430)

Microsoft Visual C++

- Standard für Windows (Teil von Visual Studio)
- Für Microsoft-Produkte optimiert, interessiert sich kaum für andere Plattformen

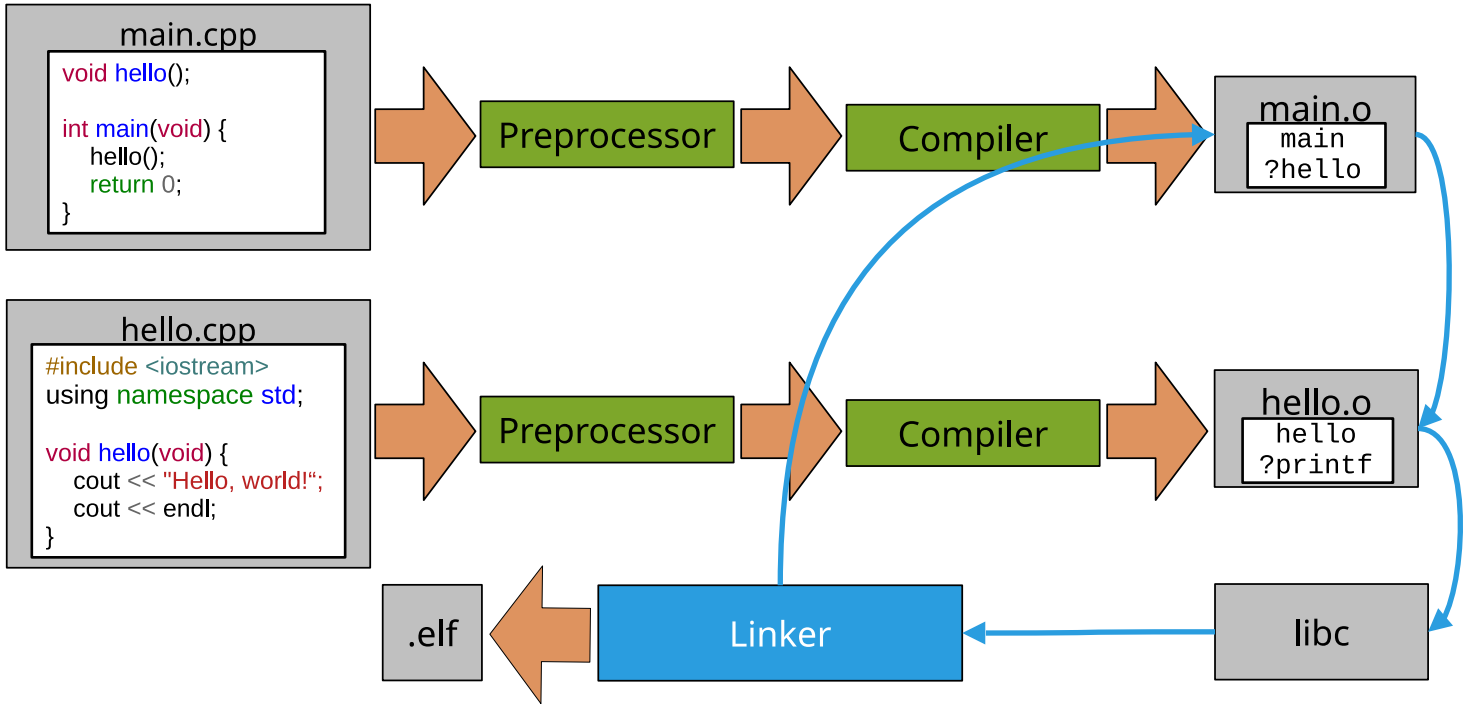
Clang (*C Lang*uage)

- Standard für MacOS
- Plattformunabhängig, strebt möglichst große Kompatibilität an
- Teil des LLVM-Projekts

- Ökosystem um *Clang* herum hat viele nützliche Werkzeuge
  - Clang Static Analyzer, Clang Tidy, ClangD, …
  - Viele sehr nützliche Warnungen und Hinweise
  - Können auch mit fremden Compilern verwendet werden

Ablauf einer Übersetzung (C/C++)

- Übersetzungsprozess
  1. Präprozessor ✅
  2. Das eigentliche Kompilieren (Erzeugung von Assembly)
  3. Assembly: Übersetzung in Binärcode
  4. Linking: Verknüpfen einzelner Objektdateien (*Object Files*) zu einer kohärenten Binärdatei



Allgemeiner Ablauf einer Übersetzung

- Übersetzungsprozess kann allgemein grob in zwei Abschnitte unterteilt werden:
  - Frontend (zuständig für die Sprache)
  - Backend (Übersetzung zu *systemspezifischen* Binäranweisungen)

Frontend: Analyse und Verarbeitung des Quellcodes

- *Lexing*: Zerlegen des Quellcodes in Einzelstücke (*Tokens*) und *Whitespace*
  - Präprozessor ist Teil dieses Schritts
- *Parsing*: Analyse der Syntax/grammatikalischen Korrektheit
  - Resultat: *abstrakter Syntaxbaum* (*Abstract Syntax Tree*, *AST*)
- Semantische Analyse: Logik- und Datenkorrektheit
  - Beispiel: Typfehler, undeklarierte Variablen

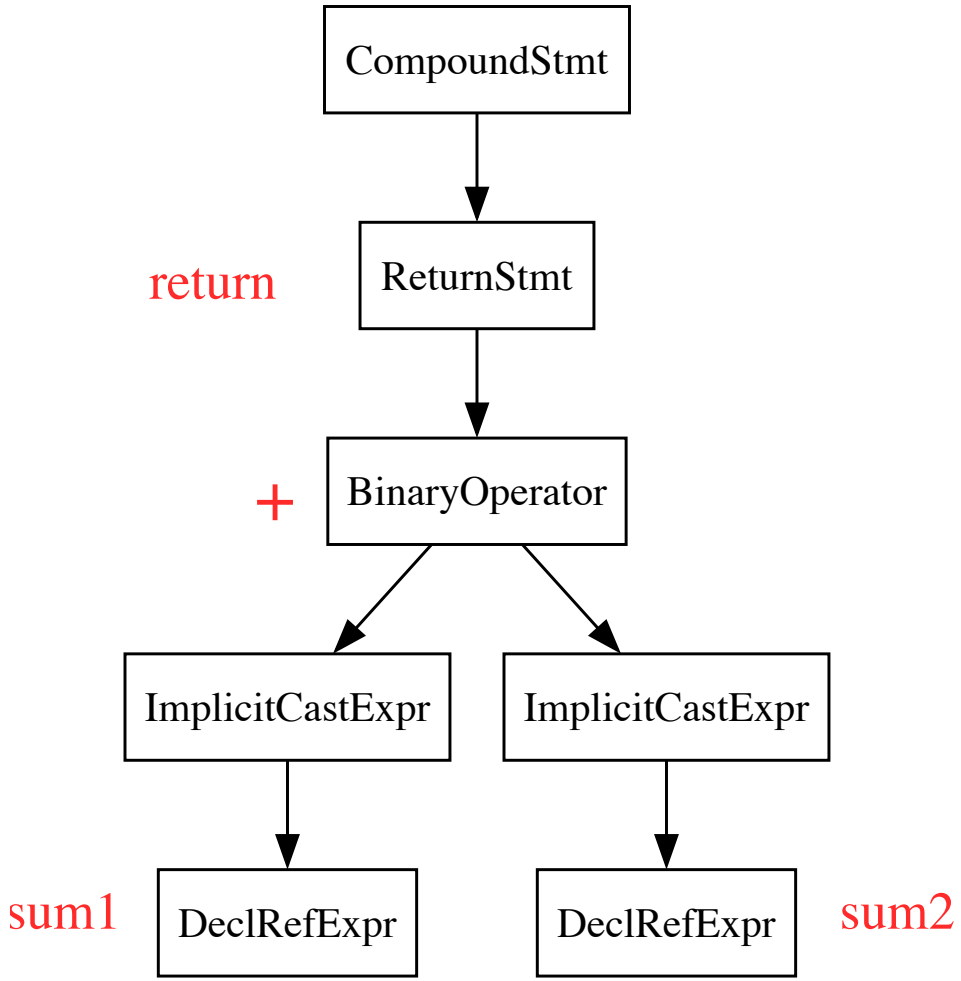
Abstract Syntax Tree

Beispiel für einen (sehr einfachen) AST

- Addieren zweier Integers

```
int add(int sum1, int sum2) {
    return sum1 + sum2;
}
```

- Umwandlung in interne Datentypen des Compilers
- Mit dem AST wird anschließend automatisiert weitergearbeitet



Middle-End

- Zwischen Frontend und Backend gibt es oft noch eine weitere Schicht: Middle-End
- AST wird in eine interne, **einheitliche** Programmiersprache übersetzt

  → *Intermediate Representation*
- Grund: Erleichtert Arbeit
  - Ansonsten: ähnliche Konstrukte (z.B. Schleife: `for` und `while`) separat im AST behandeln
- Weiterer Vorteil: ASTs müssen „nur“ zu IR übersetzt werden
  - Backend ist losgelöst von der Sprache
  - Einfaches Hinzufügen neuer Programmiersprachen → Erzeugung von AST aus neuer Programmiersprache
  - Sahnehäubchen: Performance hauptsächlich abhängig vom Backend

    → automatisch gut optimierte Programme für alle Sprachen

Beispiel – Middle-End

- Positiv-Beispiel für Middle-End: LLVM-Projekt
- Unterstützt viele Sprachen: C/C++, Rust, Fortran, ...
- *Clang* übersetzt C/C++ zu *LLVM IR*

▼ Compiler-Aufrufe für IR-Ausgabe

- Ausgabe der IR:
  - Clang: `clang -S -emit-llvm foo.c`
  - GCC: `gcc -fdump-tree-gimple foo.c`

Beispiel: LLVM-IR

```c
#include <stdio.h>

int main() {
    int i = 127;
    printf("A: %d\n", i);
}
```

```
define dso_local i32 @main() #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  store i32 0, ptr %1, align 4
  store i32 127, ptr %2, align 4
  %3 = load i32, ptr %2, align 4
  %4 = call i32 (ptr, ...)
     @printf(ptr noundef @.str,
     i32 noundef %3)
  ret i32 0
}
```

Backend

- Aufgabe im Backend: Analyse und Verarbeitung des Quellcodes
- Umwandlung in Assembly-Code und Erzeugung des Binärprogramms
- Assembly: Programm mit Maschineninstruktionen für Zielplattform in Textform

```c
// Generische C-Funktion
int add(int i) {
    return i + 1;
}
```

```asm
; Für x86-Plattform
add:
    lea eax, [rdi+1]
    ret
```

- Anschließend: Übersetzung in Binärinstruktionen

→ Ergebnis: Reihe von *Object Files* (Dateiendung: *.o*)

Binden (*Linking*)

- Zusammenführen mehrerer Objektdateien zu Programm: *Linking*
- Außerdem: Einbinden von libc-Funktionen → `printf`
- Linking führt folgende Schritte durch:
  - Sammelt alle *Object Files* ein
  - Durchsucht diese nach fehlenden Symbolen und dann die Bibliotheken
  - „Vergibt" Adressen für Variablen und Funktionen
- **Wichtig:** Lediglich Standardbibliotheken werden **automatisch durchsucht**
  - andere Bibliotheken müssen explizit aufgeführt werden
  - *libc* für C
  - *libstdc++* für C++

Dynamisches Binden

- Alle erforderlichen Bibliotheken werden beim Starten (Laden) des Programms geladen
- Welche Nebeneffekte hat dies?
- Vorteile:
  - Programm ist erheblich kleiner
  - Updates müssen nur an einer einzigen Stelle eingebracht werden
- Nachteile:
  - Alle Bibliotheke müssen ggf. nach Namenssymbolen durchsucht werden (viele Dateioperationen)
  - Etwas langsamer durch vieles Hin- und Herspringen
  - Alle benötigten Bibliotheken müssen vorhanden sein

    → Aufgabe des OS bzw. des Programmierers

Beispiel dynamisches Binden

Beispiel für die Liste an dynamisch-geladenen Bibliotheken:

```
al@ganymed:~/coding$ ldd dyn.elf
        linux-vdso.so.1 (0x00007f50f06a2000)
        libstdc++.so.6 => /lib/x86_64-linux-gnu/libstdc++.so.6 (0x00007f50f0400000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f50f020a000)
        libm.so.6 => /lib/x86_64-linux-gnu/libm.so.6 (0x00007f50f0124000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f50f06a4000)
        libgcc_s.so.1 => /lib/x86_64-linux-gnu/libgcc_s.so.1 (0x00007f50f00f7000)
```

Statisches Binden

- **Alle** verwendeten Bibliotheken werden beim Übersetzen „dazu gelegt”
- Dazu zählt auch der Start-Code für ein Programm
- Vorteile:
  - Schneller: kein Nachschauen in allen Bibliotheken
  - Läuft auf kompatiblen Betriebssystemen/CPUs ohne Probleme
- Nachteile:
  - Updates erfordern Neukompilieren → besonders kritisch bei Sicherheitslücken
  - Dateien sind erheblich größer

Fehler beim Linking

- Typische Fehlermeldung des Linkers:

```
al@ganymed:~/coding$ gcc -Wall -o hello.elf hello.c
/usr/bin/ld: /usr/lib/gcc/x86_64-linux-gnu/14/../../../x86_64-linux-gnu/Scrt1.o:
    in function `_start':
(.text+0x17): undefined reference to `main'
collect2: error: ld returned 1 exit status
```

- Hier fehlt eine `main()`-Funktion im Programm
- `main()` steht in der Datei `main.c`, wird aber nicht mit angegeben

Vergleich statisches und dynamisches Binden

Ein Beispiel:

```c
// hello.c
#include <stdio.h>


void hello() {
    printf("Hello World!\n");
}
```

```c
// main.c
extern void hello();

int main() {
    hello();
    return 0;
}
```

Vergleich statisches und dynamisches Binden

Ergebnisse

Datei	Größe (Bytes)	Größe (k/M Bytes)
`dyn.elf`	16568	~16kBytes
`static.elf`	785424	~785 KBytes
`static.lto`	706576	~707 KBytes

▼ Compiler-Aufrufe

- Dynamisch Binden: `gcc -Wall -o dyn.elf hello.c main.c`
- Statisches Binden: `gcc -Wall -static -o static.elf hello.c main.c`
- Statisches Binden (LTO + Strip): `gcc hello.c -o static-lto -static -O2 -flto -s hello.c main.c`

Build-Systeme

- Je größer das Softwareprojekt, desto aufwändiger Eintippen der Compilerbefehle
- Je mehr involvierte Dateien und Compiler-Flags, desto schwerer:
  - 2-5 Dateien: Händisch gut machbar
  - 10-20 Dateien und eine Bibliothek: Schwierig
  - 100+ Dateien und 5 Bibliotheken: Nahezu unmöglich

→ Die Lösung dafür sind *Build*-Systeme

- Das erste universelle Programm dafür war `make` (seit ca. 1976)
- Am häufigsten verwendete Implementierung: `GNU make`

Build-Systeme

- Textdatei beschreibt Rezepte zur Erzeugung von Dateien → *Makefile*
- Regeln (*Rules*) beschreiben Anweisung (*Recipe*) um Dateien (*Targets*) zu erzeugen
- Anweisungen würden sonst händisch eingetippt werden
- Ausführung: `make <target-name>`
  - Ausnutzung mehrerer CPU-Kerne ebenfalls möglich
  - Beispiel für 8 Kerne: `make -j8`
- Prinzipiell geeignet für beliebige Projektumgebungen

Beispiel:

```
# ... Setup code

# Targets
foo.elf: foo.c
        gcc -Wall -o foo.elf foo.c

bar.elf: bar.c foo.c
        gcc -Wall -o $@ $<
```

Build-Systeme

- Build-Systeme speziell für C/C++ sind etwas problematisch:
- Aufgrund des Alters der Sprachen und zeitweise fehlender Funktionalität gibt es eine Reihe von verschiedenen Lösungen
- Unter anderem: make, CMake, Conan
- Keine davon ist wirklich perfekt (besonders im Vergleich zu anderen Programmiersprachen)
- Bekannte Probleme: komplizierte Konfiguration, sehr komplex, externe Pakete (Bibliotheken, Programme) können nicht automatisch installiert werden

Leider gibt es hier keine gute Lösung 😕

Compiler-Optimierungen

- Compiler können automatisch Programm-Code optimieren
- Viele Optimierungen sind gratis und beschleunigen das kompilierte Programm!
- Anschalten der Optimierungen über die Compiler-Option `gcc -0` :
  - `gcc -00` : Standard, keine Optimierungen
  - `gcc -01` : Einige Optimierungen
  - `gcc -02` : Sehr viele Optimierungen (beinhaltet 01)
  - `gcc -03` : Sehr aggressive Optimierungen (können ggf. das Programm auch langsamer machen)
- Liste mit allen GCC-Optimierungen ist online verfügbar

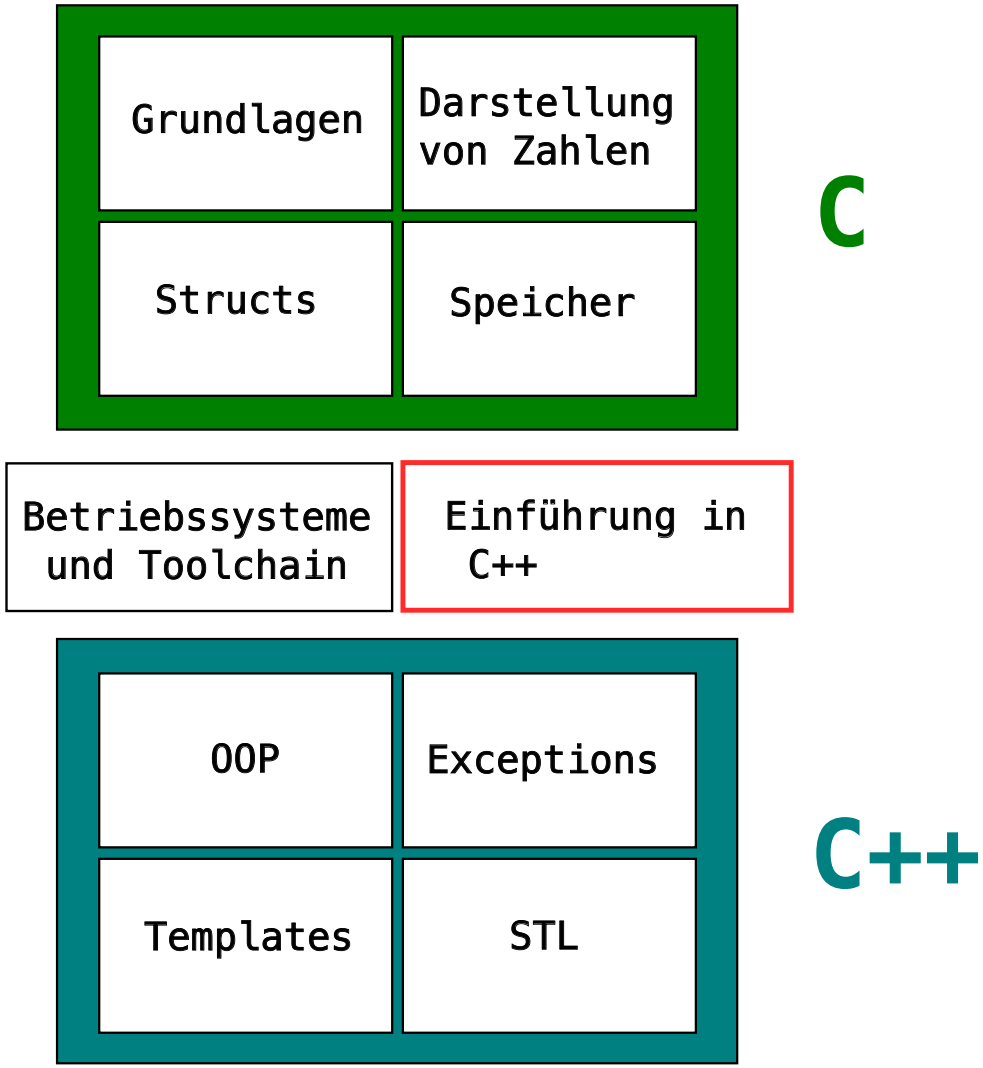
(Sehr gutes Online-Tool für anschaulichen Vergleich ist *Compiler Explorer*)

Zwischenstand

- Wir können jetzt programmieren ✅
  - Gilt insbesondere für die kniffligen Bereiche (*Speicherverständnis/Umgang mit Speicher*) ✅
- Die „Werkbank“ (→ Betriebssystem) und Übersetzer kennen wir auch ✅
- **Einmal durchatmen:** 😮‍💨
- Das Schwierigste ist jetzt geschafft
- Wir verlassen jetzt den C-Teil der Vorlesung
- Ab hier wird das Gelernte meist nur neu angeordnet oder erweitert!

What's next?

- Weiterer Ablauf der Vorlesung:
  1. Einführung in C++ (nächstes Kapitel)
     - Wichtigste Unterschiede und Stolpersteine
  2. Objektorientierte Programmierung (OOP)
     - Bündelung von Funktionen, Structs und konsistenter Initialisierung
  3. Ausnahmenbehandlung (*Exceptions*)
  4. Generische Programmierung (*Templates*)
  5. Kurzer Ausflug in die *Standard Template Library* (STL)



Zusammenfassung

- Weg von der C-Datei zum Programm
  - Präprozessor
  - Compiler
  - Linker
- Interner Aufbau eines Compilers
- Compiler-Optimierungen
- Statisches vs. dynamisches Binden

Kapitel 11 - Toolchain