1. EINFÜHRUNG

Inhalt

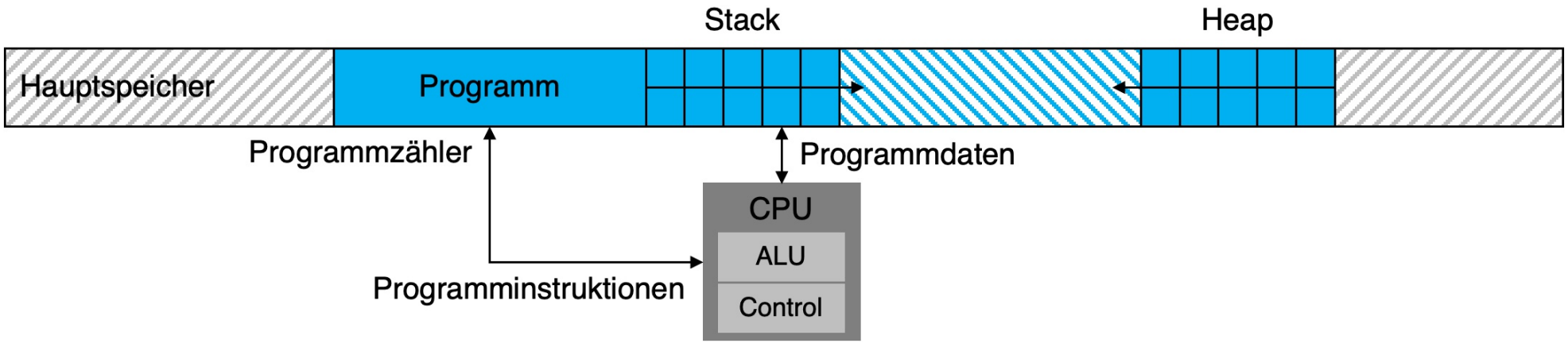
- Die Programmiersprache C++
  - Übersicht
  - Sprachelemente
  - Eigenschaften
  - Compiler und Linker Workflow
  - Minimales Programm und `HelloWorld`
- Ein erstes Programm: `ggT`
  - Erste Version: ganzes Programm in einer Datei
  - Zweite Version: Aufteilung in Hauptprogramm und Funktion
  - Dritte Version: Unit Tests
- Automatisierung mit `make`

Was ist eine Programmiersprache?

- Eine Programmiersprache ist eine **formale** Sprache zur Spezifikation von Algorithmen und Datenstrukturen
- Programmiersprachen **abstrahieren** den Computer, der die Programme ausführen soll
- Ein **Compiler** übersetzt das Programm in eine Rechenvorschrift, die der Computer versteht
- Verschiedene Programmiersprachen sind oft auf verschiedenen **Abstraktionsebenen** angesiedelt
  - Wieviel „sieht“ der Mensch von der Maschine?
  - Welche Aufgaben übernimmt der Mensch?
  - Welche Aufgaben übernimmt der Compiler?

Abstraktionsebenen?

Der Betrachtung der Abstraktionsebene einer Programmiersprache liegt typischerweise die **von-Neumann-Architektur** zugrunde.



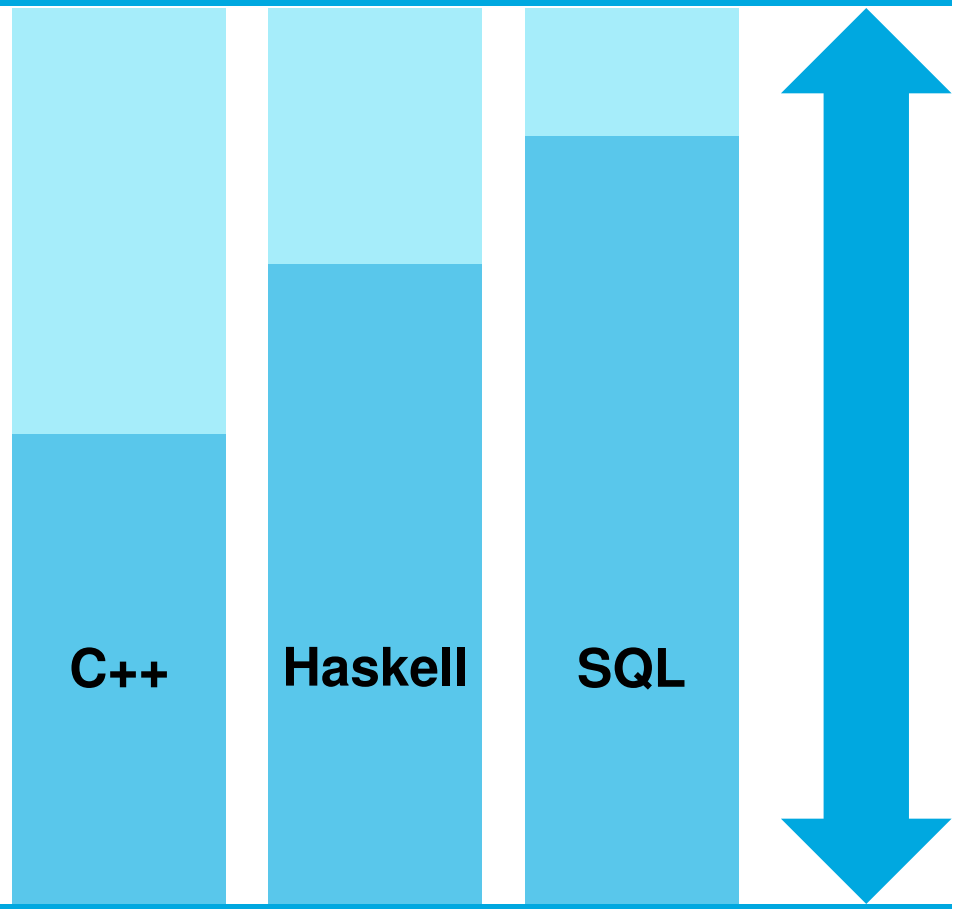
- **Tiefe Abstraktionsebene**: Der Mensch spezifiziert den Algorithmus *und* die Steuerung der Maschine
- **Hohe Abstraktionsebene**: Der Mensch spezifiziert *nur* den Algorithmus, die Programmiersprache generiert und optimiert die Steuerung der Maschine
- Wir kommen später auf konkrete Abstraktionen zurück

Abstraktionsebenen!

Problemlösungen: Algorithmen und Datenstrukturen

Dazwischen liegen „künstliche“ Probleme

- Ausführungsreihenfolge
- Speicherverwaltung
- Parallelisierung
- Verteilung
- Präzision
- Energieeffizienz
- . . .



Computer: von-Neumann-Architektur (CPU, RAM etc.)

C++ Steckbrief

- In C++ kann man auf verschiedenen Abstraktionsebenen programmieren
  - **Imperative Programmierung**: Funktionen, Variablen, Konstanten, Ausdrücke, Kontrollfluss
  - **Objektorientierte Programmierung**: Klassen, Vererbung
  - **Generische Programmierung**: Templates
- Der C++-Standard definiert zwei Arten von Entitäten
  - Konzepte des **Sprachkerns**: Basisdatentypen, Schleifen etc.
  - Komponenten der **Standardbibliothek**: Datenstrukturen, I/O-Operationen etc.
- C++ ist eine **kompilierte**, **statisch typisierte** Sprache
  - Bevor ein Programm ausgeführt werden kann, muss es in Maschinensprache übersetzt werden
  - Die Typen aller Ausdrücke werden vom Compiler während dieser Übersetzung ermittelt

Ein minimales C++-Programm

```
int main() { }
```

- Jedes C++-Programm hat **eine** `main`-Funktion, mit der die Ausführung beginnt
- Die im Beispiel gezeigte `main`-Funktion
  - hat keine Argumente: `()`
  - tut nichts: `{ }`
  - gibt einen Wert vom Datentyp Integer zurück: `int`
- Der **Rückgabewert** der `main`-Funktion zeigt an, ob das Programm erfolgreich beendet wurde
  - `0`: das Programm wurde erfolgreich ausgeführt
  - ungleich `0`: programmspezifischer Fehlercode

  GTK Wenn kein **expliziter** Rückgabewert (z.B. mit `return`) zurück gegeben wird, gibt die `main`-Funktion (und **nur** die `main`-Funktion) **implizit** `0` zurück.

Hello World!

```
#include <iostream>

int main() {
  std::cout << "Hello World!" << std::endl;
}
```

- Das Programm gibt die Zeichenkette „`Hello World!`“ aus
- Dabei wird die **Standardbibliothek** (`std`) genutzt
  - `cout` ist die Standardausgabe auf der Konsole
  - Der Operator << fügt in einen Datenstrom ein
  - `endl` bezeichnet den Zeilenumbruch („end line“)

Erstes C++-Programm

- Wir schreiben ein Programm, um den **größten gemeinsamen Teiler** zweier natürlicher Zahlen zu finden
  - Aus „Konzepte der Informatik“ kennen Sie bereits den **Algorithmus von Euklid**
  - Nun geht es weniger um den Algorithmus selbst, sondern wie man daraus ein C++-Programm macht
- Im Folgenden stellen wir Ihnen das Problem vor und setzen es dann **live** in ein C++-Programm um
  - Wir beginnen einfach und machen das Programm dann Schritt für Schritt raffinierter
  - Die Fehler, die uns dabei begegnen, werden Sie beim Selbermachen wiedertreffen

Größter gemeinsamer Teiler

- Direkte Übersetzung des **Pseudocodes** aus der Vorlesung „Konzepte der Informatik“

```
int ggT(int a, int b) {
  assert(a >= b);
  int t;
  if (b == 0) {
    t = a;
  } else {
    t = ggT(b, a % b);
  }
  return t;
}
```

Algorithm 2: EUCLID(a,b)

Input: $a, b \in \mathbb{N}; a \geq b$

Output: t = ggT(a,b)

begin

  if $b = 0$ **then** $t \leftarrow a$

  else $t \leftarrow EUCLID(b, mod(a, b))$

  return t

- Diese Funktion können wir in C++ noch **schöner**, d.h. kürzer, lesbarer und effizienter, schreiben!

Erste Version

- Im ersten Schritt schreiben wir den gesamten Programmcode in **eine** Textdatei und speichern diese unter dem Namen `main.cpp`
  - Sie können dafür einen beliebigen Texteditor verwenden
  - Ich verwende **nano**, da dieser über Syntaxhighlighting verfügt
- Als nächstes kompilieren wir unseren Programmcode mit dem GNU C++ Compiler **g++**
  - `g++ -o ggT main.cpp`
  - **GTK** Die Option `--version` zeigt die verwendete Version an
  - Im Programmierkurs benötigen wir **Version 8.5** oder neuer
- Der Befehl oben erzeugt Maschinencode, den wir wie folgt ausführen können
  - `./ggT`
  - **GTK** Ohne die Option `-o` würde das ausführbare Programm einfach `a.out` heißen

Manöverkritik

- Initialisierung von Variablen
  - GTK C++ initialisiert Variablen **nicht**!
  - Initialisierung mit cleverem Wert kann beim Debuggen helfen
- Verbesserungen am Programmcode mit **Early Return**
  - Überflüssige lokale Variablen
  - Unnötig verschachtelter Kontrollfluss
- Assertions mit `assert()`
  - Dienen zum Testen und Debuggen, nicht zur Fehlerbehandlung im fertigen Programm
  - Können mit `#define NDEBUG` ausgeschaltet werden
- Die `main`-Funktion kennt zwei Argumente
  - `argc`: Anzahl („count“) der übergebenen Argumente
  - `argv`: „Vektor“ der übergebenen Argumenten als C-Strings
  - GTK Das `0`-te Argument ist der Name des Programms
  - GTK `atoi()` wandelt C-Strings in `int`-Zahlen um

Zweite Version

- Im nächsten Schritt unterteilen wir den Programmcode wie folgt in **zwei** Dateien
  - Die erste Datei `ggt.cpp` enthält nur die Funktion `ggT`
  - Die zweite Datei `main.cpp` enthält das restliche Programm und bindet zu Beginn die erste Datei ein

    ```
    #include "./ggt.cpp"
    ```

  - Beim Thema „Modularität“ werden wir in der fünften Woche sehen, dass man das eigentlich nicht so macht
- Den Programmcode können wir wieder mit dem gleichen Befehl wie vorher kompilieren
  - `g++ -o ggT main.cpp`
  - **GTK**

    Der sogenannte Präprozessor des C++-Compilers **ersetzt** die `#include`-Direktive durch den Inhalt der angegebenen Datei

Dritte Version

- Im dritten und letzten Schritt wollen wir unser Programm mittels **Unit Tests** auf Korrektheit prüfen
- Dazu schreiben wir noch eine **dritte** Datei `test.cpp`
  - Die Datei enthält einen Test für unsere `ggT`-Funktion sowie eine generische `main`-Funktion, die alle Tests ausführt

```
#include <gtest/gtest.h>
#include "./ggt.cpp"

TEST(GGTSuite, checkGGT) { ... }

int main() {
  ::testing::InitGoogleTest();
  return RUN_ALL_TESTS();
}
```

  - **GTK** In diesem Beispiel verwenden wir das Google Test Framework, das zusätzlich installiert werden muss

Überprüfung des Codestils

- Neben korrekter Funktionalität muss guter Programmcode auch noch weitere Kriterien erfüllen
  - **Lesbarkeit**: auch andere Menschen sollen unseren Programmcode verstehen können
  - **Qualität**: unser Programmcode sollte keinen offensichtlichen Quatsch oder allseits bekannte No-Gos beinhalten
  - **Compliance**: in (großen) Projekten müssen oft Guidelines und Vorschriften eingehalten werden
- Mit **cpplint** können wir unseren Programmcode zum Glück automatisch auf die Einhaltung solcher Kriterien überprüfen
  - `cpplint *.cpp`
  - cpplint liefert umfangreiche und selbsterklärende Fehlermeldungen zum Stil des Programmcodes

Makefiles

- Mit dem Werkzeug `make` können wir die Befehle, um unseren Programmcode zu kompilieren, zu testen oder seinen Stil zu überprüfen, **automatisieren**
- Dazu legen wir eine Datei `Makefile` mit folgender Syntax an

```
<Target>:
    <Befehl1>
    <Befehl2>
    ...
```

  - GTK vor jedem Befehl **muss** ein Tabulator (keine Leerzeichen) stehen
  - Führt man im gleichen Verzeichnis „`make` Target“ aus, werden einfach die entsprechenden Befehle ausgeführt.
- Mit `make` und `Makefiles` kann man noch viel mehr machen, wie wir in der fünften Woche lernen werden!

Literatur und Links

- C++
  - `https://www.cplusplus.com/doc/tutorial` (einfacher)
  - `https://en.cppreference.com` (ultimative Referenz)
- GoogleTest Framework
  - `https://github.com/google/googletest`
  - Außerdem Installationsanleitung dazu auf dem Wiki
- cpplint
  - `https://pypi.org/project/cpplint/`
- Git
  - `https://git-scm.com`