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 #07

Typsysteme

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

Typsysteme

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
```

Typsysteme

Was ist allgemein ein Typsystem?

- Daten haben einen Typ
- Typ = Eigenschaften, mögliche Ausprägungen
  - z.B. „ganze Zahl“
  - z.B. „Zeiger zeigt auf gültigen Speicherbereich“
  - z.B. „Daten können addiert werden“
  - z.B. „Datum ist eine Funktion die ein Array als Eingabe nimmt und aufsteigend sortiert, wobei die Laufzeit maximal quadratisch mit der Eingabelänge wächst“
- Typsystem = Verwaltung von Dateneigenschaften
  - Komplexere Eigenschaften denkbar
  - Leider schnell unentscheidbar, daher i.d.R. stark limitiert

Typsysteme

Starke & Schwache, Statische & Dynamische

Stark vs. Schwach

Schwache Typisierung

- Wenig Unterscheidung von Typen/Eigenschaften
- Untypisiert: Alles geht (z.B. reine Maschinensprache)

Starke Typisierung

- Eigenschaften genau festgelegt / eingeschränkt
- Keine (gefährlichen) impliziten Umwandlungen oder Fehlinterpretationen von Daten
- „Programming Language“-Research Literatur:
  - Starke Typisierung = gar keine Laufzeittypfehler möglich
  - Das heißt: statische Typisierung Voraussetzung für starke Typis.
  - Wir nutzen hier die laxere Definition

Statisch vs. Dynamisch

Statische Typisierung

- Compiler prüft, ob Typen passen
  - Eigenschaften werden garantiert
  - Prüfung, bevor das Programm überhaupt läuft!
    - „Statische Analyse“
  - Stimmt in jeder beliebigen Ausführung des Codes!
- Compiler liefert allgemeinen „Beweis“ der Korrektheit
  - Nur einfache Tests möglich (Satz von Rice / Halteproblem!)

Statisch vs. Dynamisch

Dynamische Typisierung

- Typen (Eigenschaften) werden zur Laufzeit geprüft
- Vorteile: Einfacher zu implementieren, komplexe Tests möglich
  - Zum Beispiel:
    - Ergebnis der Berechnung ist positiv
    - Summe der Arrayelemente ist immer 42
  - Statisch wäre dies im Allg. nicht automatisch beweisbar
- Nachteile:
  - Keine allgemein Garantien:
    Fehler werden erst bei Lauf mit speziellen Eingaben bemerkt
  - Langsam (Typprüfung zur Laufzeit)

Faustregeln

Faustregeln: Speed

- Effiziente Sprachen immer statisch typisiert!
  - Beispiele: Algol, C, Pascal, Modula, C++, OCamel
  - Das Gegenteil gilt nicht unbedingt
- Beispiel C/C++:

```
int a = 40;     // Reserviere &a, &b, &temp auf dem Stack
int b = 2;      // mov 40, &a
cout << a+b;    // mov  2, &b
                // add &a, &b, &temp
                // out &temp
```

  - Repräsentation im Speicher steht bei Übersetzung fest
  - Maschinenbefehle stehen fest
  - Sehr schneller Code

Faustregeln

Faustregeln: Speed

- Beispiel Python:

```
a = 40
b = 2;
print(a+b)
```

- Langsame Ausführung
  - `a` und `b` sind allgemeine Python objekte
  - Addition: Suche nach Methode „`__add__()`“ in Methodentabelle der Klasse von Objekt `a`
  - Aufruf nach Typ zur Laufzeit
- Flexibler [auch für Fehler :-( ], aber langsamer
  - Auch mit Optimierungen (hash-table) ca. 30x langsamer

Faustregel

Faustregeln: Fehleranfälligkeit

- Statische Typisierung unterstützt das Schreiben korrekter Programme
  - Sehr hilfreich, aber nicht notwendig
    - Wissenschaftliche Studien zeigen keinen signifikanten Unterschied zu dynamisch typisierten Sprachen
  - Dynamische Typisierung erforderte besondere Methoden
    - „Test-Driven-Development“
    - Auch mit statischer Typisierung nützlich/wichtig
- Sehr sichere Systeme: formalen Beweisen
  - Compilerunsterstützung möglich (z.B. COQ, Agda, Isabelle)
  - Trotzdem sehr viel Handarbeit
  - Typisch: Flugzeugsteuerung, Stellwerk im Bahnhof

Unterschied

C++ ist

- Statisch typisiert
- Stark*) typisiert
  - Nicht „sehr stark“, tatsächlich sind noch viele Fehler möglich

Python ist

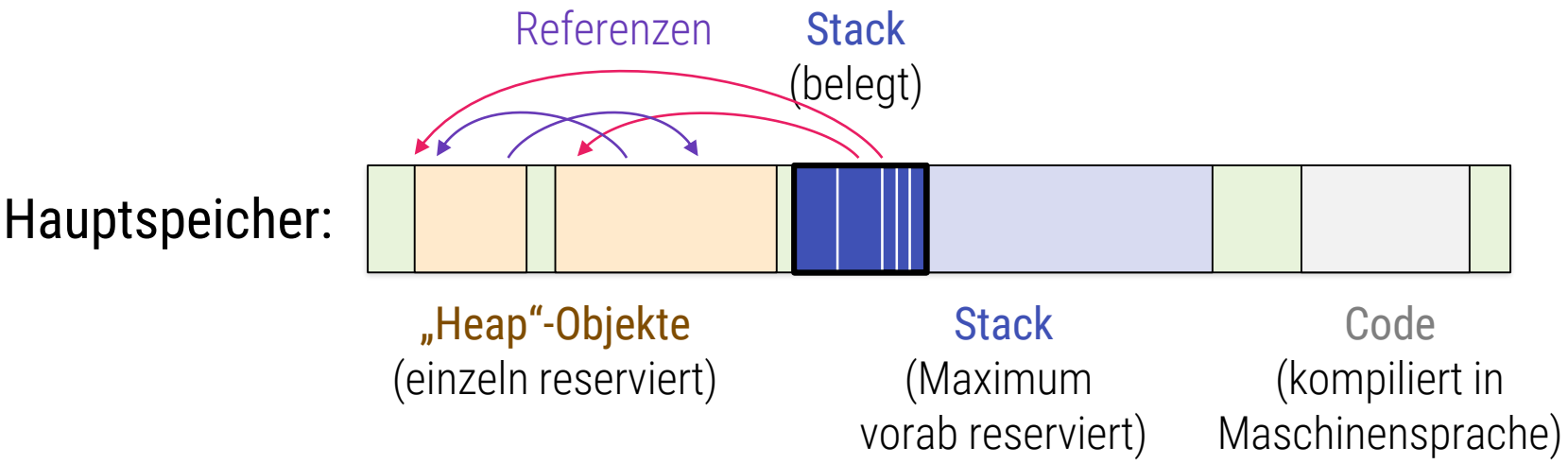
- Dynamisch typisiert
- Stark*) typisiert
  - Stärker als C++, aber auch nicht „sehr stark“

Java-Skript, PHP: dynamisch & schwach*) typisiert

*) laxere Definition

Hardwarenahe Programmierung -
Hilfe durch das Typsystem?

Datenobjekte im Speicher



Probleme

- Ungültige Zeiger, z.B.:
  - Uninitialisierte Zeigervariablen
  - Buffer overruns `ptr[offset]` oder `*(ptr + offset)`
  - Nullpointer dereferentiation

Typsystem

Mögliche Garantien eines Typsystems

- Typsicher („type save“):
  - Datentypen werden nicht ungewollt vermischt bzw. falsch interpretiert, z.B. Zeichenkette als Zahl
  - Statisch (compile-time) oder dynamisch (run-time checks)
- Speicher-sicher („memory save“):
  - Kein Zugriff auf ungültigen Speicher
  - Z.B. Buffer-overruns, Stack-Overflow, Invalid Pointer Dereferentiation
- Abwesenheit von Laufzeittypfehlern
- Weitergehendes (Invarianten)
  - Oft unberechenbar, User-Support nötig (z.B. COQ, Agda, Isabelle)

JAVA in one Slide

JAVA ist ein C++ Nachfolger

- Weniger Fehleranfällig
- (Viel) einfacher – weniger Features
- Nicht (hauptsächlich) auf Performance optimiert

Wesentliche Änderungen (Speichersicherheit)

- Keine Zeigerarithmetik erlaubt
- Zeiger nur durch „new“ zu erzeugen
- Arrays für variablen Zugriff andere, Laufzeitprüfung
- Garbage-collection: Kein delete nötig

Unterschiede

C, C++ und JAVA sind

- Statisch typisiert
- Stark*) typisiert
  - Nicht „sehr stark“, tatsächlich sind noch viele Fehler möglich

Python ist

- Dynamisch typisiert
- Stark*) typisiert
  - Stärker als C++, aber auch nicht „sehr stark“

Python, Java: Zusätzlich Garbage-Collection

- Kein „delete“; delete wird automatisch durchgeführt

Typsystem

Unsere Sprachen

- **Typsicher („type save“)**
  - Python: ja (dynamische Tests)
  - Java: ja (statische und dynamische Tests)
  - C++: i.d.R. ja (kann explizit umgangen werden!)
- **Speicher-sicher („memory save“)**
  - Python: ja (ggF. Laufzeitfehler)
  - Java: ja (Laufzeitfehler; Trick: garbage collector)
  - C++: nein (dangeling references, NULL-pointer deref.)
- **Abwesentheit von Laufzeitfehlern**
  - Alle drei Sprachen produzieren Laufzeitfehler
  - Nur C/C++ Programme können unkontrolliert abstürzen

Warum der Ärger?

C++ ist unsicher

- Ungewolltes Überschreiben von Speicher möglich
  - Andere Prozesse / OS:
    - „Segmentation Fault“ (Unix)
    - „General protection fault“ (Windows)
    - Systemcrash (schwache embedded Systeme, z.B. 68K)
  - Daten im eigenen Programm
    - Sehr schwer zu findende Fehlerquelle
    - Fehlerursache und Absturz liegen u.U. weit auseinander
- Warum tuen wir uns das an?
  - JAVA garantiert, dass das nie passieren kann
  - Python sowieso

It‘s not a Bug, it‘s a Feature!

Kosten für Speichersicherheit

- Lösung 1: Garbage Collection
  - Freigabe von ungenutzten Datenobjekten immer automatisch/implizit
  - Langsam (Overhead)
  - Schwer zu kontrollieren
  - Latenzen (GC startet plötzlich)
- Lösung 2: Komplexe Typsysteme
  - Z.B. Sprache „Rust“
  - Schwer zu handhaben (Programmierung)
  - Deckt nicht alle denkbaren Fälle ab (Architekturmuster)

C++ ist universell & schnell – dafür gefährlicher