Programmierkurs: Typen

Manfred Hauswirth | Open Distributed Systems | Einführung in die Programmierung, WS 25/26

Rückblick

VL 0 „Organisation und Inhalt“: Ablauf der Vorlesung, Termine
VL 1 „Hello World“: „Lebenswichtiges“, Programablauf, Programmierablauf, Kompilierung und Ausführung von Programmen
VL 2 „Die ersten Schritte“: Erstes C-Programm, Elementare C-Strukturen, Datentypen, Operatoren, Schleifen
VL 3 „Kontrollstrukturen & Funktionen“: Syntax, Semantik, bedingte Anweisungen, Blöcke, Sichtbarkeit
VL 4 „Rekursive Funktionen & Bibliotheken“: rekursive Funktionsaufrufe, Modularisierung
VL 5 „Typen“: Einfache und strukturierte Datentypen, Wertebereiche, Typendefinition
VL 6 „Speicher und Adressen“: Speicher, Pointer, Funktionsaufrufe „call by value“ vs. „call by reference“
VL 7 „Speicher und Arrays“: Speicher, Arrays, mehrdimensionale Arrays, Arrays und Pointer
VL 8 „Dynamische Speicherverwaltung“: Speicherallokation, Fehlerbehandlung, Rückgabewerte, Arrays/Pointer/Adressen
VL 9 „Strings, Kanäle, Git“: Strings und Arrays, Zeichensätze, Stringlänge, Ein- und Ausgabe, Arbeiten mit git
VL 10 „Debugging und Stack“: Fehlverhalten/Bugs, Fehlersuche Strategien und Werkzeuge

Variablen haben Typen

- Bisher haben wir für Variablen und Argumente nur Integer-Werte verwendet:

```
int n;   // n ist ein Integer-Wert
```

- Unsere bisherigen Programme können nur mit ganzen Zahlen umgehen
- Unsere Variablen haben den **Typ** `int`

Typen

- Typen definieren
  - die Art der Werte, die eine Variable annehmen kann
    - z.B.: `int i` $\in \mathbb{Z}$ (hier ganze Zahlen)
    - `int i = 10;`
  - einen (darstellbaren) Wertebereich
    - z.B.: `int i` $\in [-2^{31}, 2^{31}-1] \in \mathbb{Z}$
    - Wird bestimmt vom Speicherplatz, der für einen Typ festgelegt ist, z.B. 4 Byte = 32 Bit
  - Operationen, die auf den Werten ausgeführt werden können
    - z.B.: +, -, *, /, %, etc.

Einfache Typen

- Typen helfen dem Compiler zu entscheiden, ob Source-Code korrekt ist:

```
int i = 0;          // gültige Operation

int i = "hello";    // ungültige Operation

int i = 7 + 14;     // gültige Operation

int i = 7 + "hi";   // ungültige Operation
```

Wertebereich

- Wird bestimmt vom Speicherplatz, der für einen Typ festgelegt wird:
  - 1 Byte = 8 Bit ⇨ 256 unterschiedliche Bitmuster
  - 00000000, 00000001, ……., 11111111
- Typ legt fest, was diese Bitmuster bedeuten sollen (die Semantik):
  - $[0, 2^8-1]$ = [0, 255] ⇨ ganze Zahlen von 0 bis 255 oder aber
  - $[-2^7, 2^7-1]$ = [-128, 127] ⇨ ganze Zahlen von -128 bis 127 oder
  - etwas ganz anderes (irrelevant für uns)

Welcher Wertebereich?

- Keine Angabe: Sprachdefinition prüfen
- Expliziter Typ:

  `uint8_t`
  - `u`: Unsigned
  - `int`: Integer
  - `8`: 8 Bit = 1 Byte Speicherplatz
  - `_t`: „ich bin ein Typ“ (Schreibkonvention)

- `uint8_t i = 17;` `// gültige Operation`
- `uint8_t i = -100;` `// ungültige Operation`
- `uint8_t i = -1024;` `// ungültige Operation`

Integer-Typen

- `uint8_t` : unsigned int, 8 Bit, 0…255 (0…`UINT8_MAX`)
- `int8_t` : signed int, 8 Bit, -128…127(`INT8_MIN`…`INT8_MAX`)

- `uint16_t` : unsigned int, 16 Bit, 0…65535 (0…`UINT16_MAX`)
- `int16_t` : signed int, 16 Bit, -32768…32767 (`INT16_MIN`…`INT16_MAX`)

- `uint32_t` …

Kleiner Test mit Typen

```c
#include <stdio.h>
#include <stdint.h>          // notwendig um int-Typen wie uint8_t nutzen
                             // zu können!
int main() {
    uint8_t my_counter = 254;          // statt "int my_counter"

    my_counter = my_counter + 1;       // alle bekannten Operatoren
                                       // funktionieren: +, -, *, /, %
    printf("%d\n", my_counter);        // gibt 255 aus
    my_counter = my_counter + 1;
    printf("%d\n", my_counter);        // gibt 0 aus, da Überlauf
                  // vgl. 999 + 1 wenn man nur die letzten 3 Stellen
                  // der Lösung anschaut
    return 0;
}
```

Wieviel Platz braucht ein Typ?

- `sizeof` gibt Größe eines Typs in Byte aus

- `sizeof(uint32_t)` liefert 4 als Ergebnis, da 32 Bit = 4 Byte

- Was ist nun `int`?
  - Nicht eindeutig festgelegt!
  - Ein `int` ***kann*** mindestens dieselben Zahlen wie ein `int16_t` darstellen, ***darf*** aber auch mehr darstellen können.

Fangfrage: Was tut das folgende Programm?

```c
#include <stdio.h>
#include <stdint.h>

int main() {
    int x = INT16_MAX;
    x = x + 1;
    printf("%d\n", x); // Was ist `x`?
    return 0;
}
```

Antwort

- Hängt vom Computer ab, auf dem das Programm ausgeführt wird.
- Unangenehme Fehlerquelle

- Wir wollen lieber über Programme als solche nachdenken statt über konkrete Computer, denn davon gibt es viel zu viele unterschiedliche.
- Darum lieber `int16_t` statt `int` verwenden.

Und … andere Typen?

- Unbegrenzt viele!
- Diese kommen nun im Anschluss!
  - **Einfache** Datentypen
  - **Strukturierte** Datentypen

Wahrheitswerte

- Für Schleifen oder If-Statements testet C, ob ein Ausdruck zu 0 auswertet oder nicht
  - 0 ⇒ „falsch“
  - alle anderen Werte ⇒ „wahr“

- Schöner: Warum können wir nicht im Source-Code direkt über „wahr“ und „falsch“ („true“ und „false“) sprechen.

- Der Typ (d.h., der Wertebereich), der nur „true“ und „false“ enthält, wird Programmiersprachen-übergreifend als „Boolean“ bezeichnet (siehe Boolsche Algebra).

Boolean: Beispiel

```
#include <stdbool.h>
void practice() { /* do something */ };

int main() {
    bool is_programming_mysterious = true;
   if (is_programming_mysterious) {
      practice();
      is_programming_mysterious = false;
   }
      return 0;
}
```

Boolean ist eindeutiger

- `int is_equal(int8_t x, int8_t y);`
  - Kann hier wohl auch 2 returned werden? Oder -123? Was passiert hier?
- `bool is_equal(int8_t x, int8_t y);`
  - Klar verständlicher Code: „true“ oder „false“, sonst nichts!

- Allgemeines Prinzip: **Möglichst präzise Typen machen Code besser verständlich** (und weniger fehleranfällig).
  - Und der Compiler kann besser helfen
    `bool is_awesome = -123` gibt eine Warnung
    `int is_awesome = -123` gibt keine Warnung

Kommazahlen

- Typ `float` („floating point numbers“ - Gleitkommazahlen)

```
int main() {
    float speed = 2;
    float acceleration = 1.2;
    speed = speed * acceleration; // 2.4
    return 0;
}
```

Gleitkommazahlen – float

- `float` hat (meist) 32 bit ⇒ nicht unendlich viele, reelle Zahlen darstellbar
  - selbst zwischen 0 und 1 gibt es schon unendlich viele Zahlen
    ⇒ manchmal muss gerundet werden ⇒ Ungenauigkeiten!
    z.B.: 0.2 + 0.1 = 0.30000000000000004
- Gut geeignet, wenn kleine Rundungsfehler akzeptabel sind, z.B., um in Echtzeit ein Computerspiel anzuzeigen
- Nicht geeignet wenn Rundungsfehler inakzeptabel sind, z.B., um Geldbeträge zu verwalten ⇒ spezielle numerische Bibliotheken
- Besondere Werte, welche zu keiner reellen Zahl korrespondieren:
  - 1.0 / 0.0 = `INFINITY`, -1.0 / 0.0 == `-INFINITY`, 0.0 / 0.0 = `NAN` (not a number)

Ausgabe mit println: Formatzeichen

- Wichtige Formatzeichen:

%d	Integer	int, int8_t, int16_t
%u	Unsigned Integer	unsigned int, int8_t, …
%f	Gleitkommazahl	float
%lf	Gleitkommazahl	double
%c	Einzelzeichen	char
%s	Zeichenkette	char *

Aufzählungen

- Nicht alles lässt sich mit Zahlen gut darstellen
  - Oft braucht man „nur“ wohlunterscheidbare Werte
  - z.B. Verkehrsampel: grün, gelb, rot

- Enumerations („enums“) erlauben es eigene Typen als Liste von möglichen Werten („variants“) zu definieren

Enum: Beispiel

```
enum TrafficLightColor {
    Red,
    Yellow,
    Green
};
enum TrafficLightColor MyTrafficLight;
```

Der neue (Daten-) Typ heißt dann

`enum TrafficLightColor`

nicht nur `TrafficLightColor` !

Enum: Beispiel

```
enum TrafficLightColor next_color(enum TrafficLightColor c) {
    if (c == Red) {
        return Yellow;
    } else if (c == Yellow) {
        return Green;
    } else {
        return Red;
    }
}
```

Typendefinition: „syntactic sugar“

Alternative **Namen** für Typen:

```
typedef float Speed;
typedef float Acceleration;

Speed apply_acceleration(Speed s, Acceleration a) {
    return s * a;
}

// Achtung: Erzeugt keine neuen Typen, bloß neue Namen!
Speed x = 2.9;
Acceleration y = 5.2;
x = y;                    // erlaubt, jetzt ist x = 5.2
```

Typedef: Praktisch für enum

```
enum CoinFace_ {
    Heads,
    Tails
};
typedef enum CoinFace_ CoinFace;
CoinFace x = Heads;     // äquivalent zu `enum CoinFace_ x = Heads;`

// Kompaktere Syntax:
typedef enum CoinFace_ {
    Heads,
    Tails
} CoinFace;
```

⇒ **`enum`** muss nicht mehr geschrieben werden

Einfache Datentypen reichen nicht

- Bis jetzt: **Einfache** Datentypen
  - int, float, double, Aufzählungen
- Aber das reicht nicht
  - z.B. die Distanz zweier Punkte in einem Koordinatensystem:

```
// ziemlich unpraktisch
float distance_3d(float x1, float y1, float z1,
                  float x2, float y2, float z2)
        { /* Berechne Distanz */ }
```

⇒ **Strukturierte** Datentypen

Strukturierte Datentypen

- Schöner wäre

```
float distance_3d(Point3d p1, Point3d p2) { /* Berechne Distanz */ }
```

⇒ Datentypen `Point3d` mit x-, y-, und z-Koordinaten des Punktes, **d.h. der Datentyp hat eine Struktur**

```
struct Point3d {
    float x;
    float y;
    float z;
}; // erstellt neuen Typ namens `struct Point3d`
```

- Dieser neue Datentyp kann dann wie jeder andere verwendet werden

```
// Wert erstellen
struct Point3d my_point = { .x = 0.5, .y = 3.14, .z = -123.4 };

float a = 5.0 + my_point.y;       // Felder auslesen
my_point.z = 1.0                  // Felder zuweisen
```

Arbeiten mit struct

- struct-Werte können genauso kopiert werden wie andere Werte auch:

```
struct Point3d my_other_point = my_point;
my_other_point.x = 80.6;
```

`my_point.x` ist noch 0.5, aber `my_other_point.x` ist nun 80.6

Funktionen mit struct

- Funktionen können mit structs arbeiten:

```
struct Point3d add(struct Point3d p1, struct Point3d p2) {
        struct Point3d result;
        result.x = p1.x + p2.x;
        result.y = p1.y + p2.y;
        result.z = p1.z + p2.z;
        return result;
}
```

Typedef macht das Leben leichter

```
typedef struct MyNewType_ {
    int32_t anInt;
    float   aFloat;
} MyNewType;

MyNewType x = {.anInt = 8, .aFloat = 9.99};
```

struct in einer struct

```
typedef struct Point2d_ {          // Punkt in 2D-Koordinatensystem
    float x;
    float y;
} Point2d;

typedef struct LineSegment2d_ { // Linie in 2D-Koordinatensystem
    Point2d p1;                    // Anfangspunkt
    Point2d p2;                    // Endpunkt
} LineSegment2d;

LineSegment2d line = {             // Linie von (0.0/1.0) nach (6.23/3.3)
    .p1 = { .x = 0.0, .y = 1.0 },
    .p2 = { .x = 6.23, .y = 3.3 }
};
```

Mehr „struct in einer struct“

```
float x1 = line.p1.x;

Point2d endpoint = line.p2;

line.p2.y = 5.5; // endpoint.y hingegen ist
                 // immer noch 3.3
```

Rekursive struct nicht möglich!

ACHTUNG !!!

```
struct Weird {
    int8_t        i;
    struct Weird w;
};

Weird x = {.i = 0, .w = { .i = 1, .w = {... /*oh no!*/}}};
```

Ausblick

VL 0 „Organisation und Inhalt“: Ablauf der Vorlesung, Termine
VL 1 „Hello World“: „Lebenswichtiges“, Programablauf, Programmierablauf, Kompilierung und Ausführung von Programmen
VL 2 „Die ersten Schritte“: Erstes C-Programm, Elementare C-Strukturen, Datentypen, Operatoren, Schleifen
VL 3 „Kontrollstrukturen & Funktionen“: Syntax, Semantik, bedingte Anweisungen, Blöcke, Sichtbarkeit
VL 4 „Rekursive Funktionen & Bibliotheken“: rekursive Funktionsaufrufe, Modularisierung
VL 5 „Typen“: Einfache und strukturierte Datentypen, Wertebereiche, Typendefinition
VL 6 „Speicher und Adressen“: Speicher, Pointer, Funktionsaufrufe „call by value“ vs. „call by reference“
VL 7 „Speicher und Arrays“: Speicher, Arrays, mehrdimensionale Arrays, Arrays und Pointer
VL 8 „Dynamische Speicherverwaltung“: Speicherallokation, Fehlerbehandlung, Rückgabewerte, Arrays/Pointer/Adressen
VL 9 „Strings, Kanäle, Git“: Strings und Arrays, Zeichensätze, Stringlänge, Ein- und Ausgabe, Arbeiten mit git
VL 10 „Debugging und Stack“: Fehlverhalten/Bugs, Fehlersuche Strategien und Werkzeuge