Datentypen

by

Dr. Günter Kolousek

Daten

- Analoge Daten
  - Bild, Sprache, Messwerte,…
  - $\rightarrow$ AD und DA Wandler
- Digitale Daten
  - Zeichendaten
    - Text (unformatiert)
    - Formatierte Daten
  - Binärdaten

Digitale Daten

- unstrukturiert
  - Textdaten: Folge von Zeichen
  - Binärdaten: Folge von Bits/Bytes
- strukturiert
  - Textdaten: CSV, XML, JSON, YAML,…
  - Binärdaten: Zahlen, PNG, JPEG, GIF,…
    - Objekte: z.B. Java, C#, Python,…

strukturiert vs. semi-strukturiert

- strukturierte Daten
  - Daten müssen einem definierten Datenbankmodell entsprechen
- semi-strukturierte Daten
  - Daten unterliegen keiner formalen Struktur eines Datenbankmodells
    - z.B. Datenbankschema
  - Daten tragen einen Teil der Strukturinformation in sich tragen
  - gebräuchliche Datenformate
    - XML
    - JSON

Informationsgehalt

- Ein *Bit* ist
  - die Maßeinheit für die Datenmenge digitaler Daten
  - die Stelle einer Binärzahl
- Einheiten

Einheit	Präfix	Abkürzung	Anzahl	Menge
Bit		b		ja/nein
Byte		B	2^3 Bits	ein ASCII-Zeichen
Kibibyte	Kibi	KiB	2^{10} Bytes	halbe Seite Text
Mebibyte	Mebi	MiB	2^{20} Bytes	Buch ohne Bilder
Gibibyte	Gibi	GiB	2^{30} Bytes	2 Musik CDs
Tebibyte	Tebi	TiB	2^{40} Bytes	Textmenge einer großen Bibliothek
Petibyte	Pebi	PiB	2^{50} Bytes	ca. Datenmenge: Augen und Ohren in 100 Jahren
Exibyte	Exbi	EiB	2^{60} Bytes	…

  - Achtung: 100 GB $\neq 100 \times 2^{30} = 107'374'182'400$

Variable

- in Programmiersprache (wie Java, C# oder C++)
- beinhaltet digitale Daten
- Merkmale
  - Bezeichner (identifier)
  - (konkreter) Datentyp ($\rightarrow$ statische Typbindung!)
    - $\rightarrow$ Größe
  - Adresse
- aber: Variable vs. Name ($\rightarrow$ Python)

Datentyp

- Definition: Datentyp (engl. data type), kurz: Typ
  - definiert Menge von Operationen
- Operation … Verhalten
  - in OO Programmiersprachen: Methode (method), in C++: member function
  - $\rightarrow$ Signatur + Spezifikation des Verhaltens
  - Signatur
    - Name der Operation
    - Anzahl und Reihenfolge der Parametertypen
    - Rückgabewert zählt in Java, C, C++ **nicht** zur Signatur ($\rightarrow$ Überladen)
  - Prototyp
    - Begriff in C und C++
    - Rückgabetyp, Name der Funktion, Anzahl und Reihenfolge der Parametertypen ($\rightarrow$ Headerdatei)

Typspezifikation

- Angabe der Signatur
  - unterspezfiziert
  - $\rightarrow$ **Interface**
- Axiomatische oder algebraische Spezifikation des Verhaltens
  - voll spezifiziert
  - $\rightarrow$ **abstrakter Datentyp** (abstract data type, ADT)
    - Achtung: hat nichts mit einer *abstrakten Klasse* zu tun
- Spezifikation der Implementierung
  - überspezifiziert
  - $\rightarrow$ konkreter Datentyp
  - $\rightarrow$ **Klasse** …und vordefinierte Typen wie `int`, `bool`, usw.

Unterscheidungen

- Art des Typs
  - Werttypen: keine Identität, nur Wert
    - primitive oder fundamentale Typen
      – eingebaut, keine Methoden
      – z.B. Java: `int` vs. `Integer`
    - $\rightarrow$ Wertobjekte
  - Objekt- oder Referenztypen
- Eingebaut (built-in) oder benutzerdefiniert (user defined)
- Multpilizität: skalar oder mehrwertig

Skalar vs. mehrwertig

- skalare Datentypen (einwertig, engl. scalar)
- mehrwertige Datentypen (engl. multi-valued)
  - zusammengesetzte DT (engl. compound, composite, structure, aggregate data type, record)
    - z.B. `struct`, `class`, `union`
    - Bitfield, z.B. in C++:

```
struct IOPort {
    unsigned read:4,
    unsigned write:4
};
```

  - Container DT
    - Sequenz: Reihenfolge!
    - mengenwertig: keine Reihenfolge!
    - Abbildungstyp (mapping; assoziatives Array, Dictionary, Map, Multimap)
    - Tree, Graph

Skalare Datentypen

- arithmetischer Typ
  - Integraler Typ (siehe C, C++): rechnen und bitweise Operationen!
    - Ganze Zahlen, wie z.B. `int`, `long`
    - Boolscher Typ: `bool`
    - Zeichentyp: `char`
  - Gleitkommazahl, wie z.B. `float`, `double`, `long double`
  - komplexe Zahl
    - Python: `numbers.Complex`
    - C++: `std::complex`
- Aufzählungstyp, wie z.B. `enum`
- Zeiger, Referenzen
- ordinale Typen (diskrete Werte)

  $\rightarrow$ Integrale und Aufzählungstypen

Sequenztypen

- String: index, nur Zeichen, je nach Implementierung veränderbar oder nicht
- Liste: index, veränderbar
- Tupel: index, nicht veränderbar (zumindest nicht Größe)
- Array (Feld): index, Größe nicht veränderbar, Elemente des *selben* Typs, liegen *hintereinander* im Speicher
  - 2 Arten von mehrdimensionalen Arrays
    - rechteckige sequentielle Arrays
    - Array von Arrays
- Stream: nur sequentieller Zugriff!

2-dim Arrays in C++

- rechteckige sequentielle Arrays

```
char ttt_field[3][3]{
    {'x', 'o', 'x'},  // (0,0), (0,1), (0,2)
    {'o', 'o', 'x'},  // (1,0),...
    {'o', 'x', 'x'}
};
```

- Array von Arrays

```
// e.g. an array of C-strings
char* days[]{"montag", "dienstag", /* ... */ };
```

2-dim Arrays in Java

- ”zweidimensionale” Arrays sind *immer* Arrays von Arrays
- Beispiel 1

```
String[][] day_entries = new String[31][];
day_entries[0] = new String[1];
day_entries[0][0] = ”my first daily log”;
day_entries[1] = new String[3];
day_entries[1][0] = ”my second daily log”;
day_entries[1][1] = ”my third daily log”;
```

- Beispiel 2

```
char[][] chess_field = new int[8][8];
chess_field[0][0] = 'T';
chess_field[0][7] = 'T';
```

2-dim Arrays in C#

- rechteckige sequentielle Arrays

```
//                  rows x columns
int[,] mat=new int[3,3];
mat[0,0] = 1;
mat[0,1] = 2;
mat[0,2] = 3;
```

- non-rectangular (jagged)

```
int[][] nonrect={
    new int[]{0},
    new int[]{1,2},
    new int[]{3,4,5},
    new int[]{6,7,8,9}};
WriteLine(nonrect[2][1]); // -> 4
```

Listen in Python

- ”zweidimensionale” Listen sind immer eine Liste von Listen

```
>>> mat = [[0,0,0],[0,0,0],[0,0,0]]
>>> mat
[[0, 0, 0], [0, 0, 0], [0, 0, 0]]
>>> mat[0][1] = 1
>>> mat
[[0, 1, 0], [0, 0, 0], [0, 0, 0]]
```

- Sequenzmultiplikation

```
>>> ”a” * 3
'aaa'
>>> lst = [1] * 9
>>> lst
[1, 1, 1, 1, 1, 1, 1, 1, 1]
>>> mat2 = [[0, 0, 0]] * 3
>>> mat2
[[0, 0, 0], [0, 0, 0], [0, 0, 0]]
>>> mat2[0][1] = 1
>>> mat2
```

Listen in Python

- ”zweidimensionale” Listen sind immer eine Liste von Listen

```
>>> mat = [[0,0,0],[0,0,0],[0,0,0]]
>>> mat
[[0, 0, 0], [0, 0, 0], [0, 0, 0]]
>>> mat[0][1] = 1
>>> mat
[[0, 1, 0], [0, 0, 0], [0, 0, 0]]
```

- Sequenzmultiplikation

```
>>> ”a” * 3
'aaa'
>>> lst = [1] * 9
>>> lst
[1, 1, 1, 1, 1, 1, 1, 1, 1]
>>> mat2 = [[0, 0, 0]] * 3
>>> mat2
[[0, 0, 0], [0, 0, 0], [0, 0, 0]]
>>> mat2[0][1] = 1
>>> mat2
[[0, 1, 0], [0, 1, 0], [0, 1, 0]]
```

Lexikographisches Vergleichen

- Vergleichen siehe Folien Mengen bzgl. Totalordnung
  - $2.7182818 \leq 3.1415926$ aber $1 + 2.718j \not\leq 1 + 3.141j$
- Und mit Sequenzen?
  - Lexikographisch $\equiv$ Sortieren wie im Lexikon
  - in Python
    - `"abc" < "abd"` $\rightarrow$ `True`
    - `(1, 1) < (1, 2)` $\rightarrow$ `True`
    - `[1, 2, 3] < [1, 2, 4]` $\rightarrow$ `True`

Lexikographisches Vergleichen

- Vergleichen siehe Folien Mengen bzgl. Totalordnung
  - $2.7182818 \leq 3.1415926$ aber $1 + 2.718j \not\leq 1 + 3.141j$
- Und mit Sequenzen?
  - Lexikographisch $\equiv$ Sortieren wie im Lexikon
  - in Python
    - `"abc" < "abd"` $\rightarrow$ `True`
    - `(1, 1) < (1, 2)` $\rightarrow$ `True`
    - `[1, 2, 3] < [1, 2, 4]` $\rightarrow$ `True`
  - in C++
    - `"abc" < "abd"` $\rightarrow$ `true`
    - `std::pair{1, 1} < std::pair{1, 2}` $\rightarrow$ `true`
    - `std::tuple{1, 2, 3} < std::tuple{1, 2, 4}` $\rightarrow$ `true`
    - `std::vector{1, 2, 3} < std::vector{1, 2, 4}` $\rightarrow$ `true`
    - `operator<` überladen!

Mengenwertige und Abbildungs DT

- Mengenwertige DT
  - Keine Reihenfolge!
  - Set: keine Doppelten!
  - Bag: mehrfache Vorkommen!
  - Bitstring (bit set, bit array, bit vector, bit map): Folge von Bits im Speicher mit effizientem Zugriff auf einzelne Bits (setzen, zurücksetzen, abfragen, maskieren)
- Abbildungs DT
  - Key $\rightarrow$ Value
  - *Menge* von Keys
    - Keine doppelten Keys! (außer so etwas wie Multi-Map)
    - aber: Reihenfolge bei gewissen Implementierungen gegegeben
  - Values: keine Einschränkung
- Keys: oft nicht veränderbar oder undefiniertes Verhalten!

Eigenschaften

- Nicht veränderbar
  - wie implementiert?
    - $\rightarrow$ *immutable objects*
- Keine Doppelten $\rightarrow$ keine gleichen Elemente
  - wie ist Gleichheit definiert?
    - Gleichheit der Werte bzw. Gleichheit der Identität
      $\rightarrow$ gleich vs. dasselbe
  - wie wird Gleichheit implementiert?
    - z.B. in Java: Methoden `equals` und `hashCode`
  - Erstellung eines gleichen Objektes: Kopie!
    - Beachte: Übergabe per-value vs. per-reference!
    - $\rightarrow$ seicht vs. tief (engl. shallow vs. deep)!
- Reihenfolge vs. keine Reihenfolge
  - wie ist Reihenfolge definiert?
    - z.B. lexikographische Ordnung bei Strings
  - wie wird diese Reihenfolge implementiert?
    - z.B. in Java: Interface `Comparable`

Immutable objects

- Keine Veränderung *nach* der Initialisierung
- Implementierung entweder
  - Markierung mittels Schlüsselwörter
    - wie `const`, `final` je nach Programmiersprache
  - Datentyp lässt keine Veränderung zu
    - z.B. Klasse `String` in Java, Python, C#
- Warum?
  - kein Kopieren notwendig
  - Referenz (Pointer): ohne Bedenken weitergeben!
  - können gut als Keys in Abbildungs DT verwendet werden
  - automatisch thread-safe

Classes should be immutable unless there's a very good reason to make them mutable….If a class cannot be made immutable, limit its mutability as much as possible. – Joshua Bloch (Effective Java)

Value Object (Wertobjekt)

- Was?
  - Gleichheit basiert nur auf Wert (Inhalt)
    - nicht auf Identität
  - keine Identität
    - u.U. vorhanden, wie z.B. Adresse
    - $\rightarrow$ werden bei Übergabe kopiert (nicht in Java!)
  - sind immutable objects
    - Achtung aber in C#: `System.ValueType`!
- z.B. eine Münze
  - Wert und Währung
  - unabhängig von einer Seriennummer (id)
  - zwei Münzen sind gleich, wenn Wert und Währung gleich
  - kann nicht geändert werden
- z.B. `str`, `tuple` in Python, `String` in Java und C#,...
- Gegenteil: Entity Object (oder kurz Entity)

Parameterübergabe

- per-value
  - es wird kopiert: Achtung bei großen Datentypen
- per-reference
  - als Ein/Ausgabeparameter verwendbar
  - es wird die Adresse kopiert $\rightarrow$ Performance

Parameterübergabe

- per-value
  - es wird kopiert: Achtung bei großen Datentypen
- per-reference
  - als Ein/Ausgabeparameter verwendbar
  - es wird die Adresse kopiert → Performance

```
#include <iostream>
#include <vector>
using namespace std;

void scale(vector<double>& v, const double& factor) {
    for (size_t i{}; i != v.size(); ++i) {
        v[i] /= factor;  }}

int main() {
    vector<double> values{5, 4, 3, 2, 1};
    scale(values, values[0]);
    for (const auto& v : values)
        cout << v << ' ';  }
```

```
1 4 3 2 1
```

Gleich vs. dasselbe

- ”das gleiche Fahrrad” vs. ”dasselbe Fahrrad”
  - gleich: Gleichheit bezüglich Daten
  - dasselbe: Gleichheit bezüglich Identidät
- Beispiel

```
>>> a = [1, 2, 3]; b = [1, 2, 3]
>>> a == b
True
>>> a is b
False
>>> id(a)  # z.B.:
3068807852
>>> id(b)
3068807852
>>> c = a  # Kopie der Referenz!
>>> id(a) == id(c)
True
```

  - Referenztypen in Java defaultmäßig gleich bzgl. Identität!

Kopieren: seicht vs. tief

```
>>> arr1 = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> arr2 = arr1  # Kopie der Referenz!
>>> arr2[0][0] = ”X”
>>> arr1
[['X', 2, 3], [4, 5, 6], [7, 8, 9]]
>>> arr3 = arr1.copy()  # seichte Kopie!
>>> arr3[0] = [1, 2, 3]
>>> arr1
[['X', 2, 3], [4, 5, 6], [7, 8, 9]]
>>> arr3[1][1] = ”Y”
>>> arr1
[['X', 2, 3], [4, 'Y', 6], [7, 8, 9]]
>>> import copy
>>> arr4 = copy.deepcopy(arr1)  # tiefe Kopie!
>>> arr4[2][2] = ”Z”
>>> arr4
[['X', 2, 3], [4, 'Y', 6], [7, 8, 'Z']]
>>> arr1
[['X', 2, 3], [4, 'Y', 6], [7, 8, 9]]
```

Datentyp vs. Datenstruktur

- Datentyp: legt Verhalten fest
- Datenstruktur: legt Struktur fest
  - um (neuen) Datentyp zu *implementieren*
  - wird in der Regel nur für mehrwertige DT verwendet
- Datenstrukturen
  - Array
  - Liste: sll, dll, Array
  - Set: BSB (bst), Hasharray
  - Map: bst, hasharray
  - Stack: Array, sll
  - Queue, Deque: sll, dll, Array
  - Ringbuffer: Array
  - Heap: Array
  - Priority Queue: Heap
  - Graph: Array, Map

Abstrakter Datentyp (ADT)

- Ein ADT…
  - definiert einen Typ
  - definiert eine Menge von Operationen (genannt Interface)
    - beschreibt WAS aber **nicht** WIE (durch formale Definition)
  - beschränkt Zugriff auf Typ über Operationen
    - kein direkter Zugriff auf die Daten
- Formale Beschreibung
  - mathematisch-axiomatisch
  - mathematisch-algebraisch

Was also *ist* ein ADT?

- So etwas ähnliches wie eine Klasse (mit Instanzvariable und Methoden)?
  - NEIN, denn:
    - beschreibt WIE
- dann vielleicht eine abstrakte Klassen?
  - NEIN, denn:
    - beschreibt teilweise WIE
- aha, also so etwas wie Java Interfaces?
  - NEIN, denn:
    - beschreibt weder WAS noch WIE
    - nur Signatur der Operationen!

Stack – Signatur

$$
\begin{aligned}
&\mathit{empty\_stack} : \rightarrow \mathit{Stack} \\
&\mathit{is\_empty} : \mathit{Stack} \rightarrow \mathit{bool} \\
&\mathit{push} : \mathit{Stack} \times \mathit{Element} \rightarrow \mathit{Stack} \\
&\mathit{pop} : \mathit{Stack} \rightarrow \mathit{Stack} \\
&\mathit{top} : \mathit{Stack} \rightarrow \mathit{Element}
\end{aligned}
$$

Stack – Semantik: axiomatisch

$x : \mathit{Element}$

$s : \mathit{Stack}$

$\mathit{is\_empty}(\mathit{empty\_stack}()) = \mathit{true}$

$\mathit{is\_empty}(\mathit{push}(\mathit{empty\_stack}(), x)) = \mathit{false}$

$\mathit{pop}(\mathit{empty\_stack}()) \rightarrow \text{Error}$

$\mathit{pop}(\mathit{push}(s, x)) = s$

$\mathit{top}(\mathit{empty\_stack}()) \rightarrow \text{Error}$

$\mathit{top}(\mathit{push}(s, x)) = x$

$$\mathit{push}(\mathit{pop}(s), \mathit{top}(s)) = \begin{cases} s & \text{falls} \quad \mathit{is\_empty}(s) = \mathit{false} \\ \rightarrow \text{Error} & \text{sonst} \end{cases}$$

Stack – Semantik: algebraisch

$$s \in \{()\} \cup \{(x_1, ..., x_n) | x_i \in \mathit{Element}, n \in N, n \geq 1\}$$

$$\mathit{empty\_stack}() = ()$$

$$\mathit{is\_empty}(s) = (s = ())$$

$$\mathit{push}(s, x) = \begin{cases} (x,) & \text{falls } s = () \\ (x_1, ..., x_n, x) & \text{falls } s = (x_1, ..., x_n) \end{cases}$$

$$\mathit{top}(x) = \begin{cases} x_n & \text{falls } s = (x_1, ..., x_n) \\ \rightarrow \text{Error} & \text{sonst} \end{cases}$$

$$\mathit{pop}(s) = (x_1, ..., x_{n-1}) \quad \text{falls } s = (x_1, ..., x_n)$$

$$\mathit{pop}(s) = \begin{cases} () & \text{falls } s = (x) \\ \rightarrow \text{Error} & \text{sonst} \end{cases}$$

Generische DT

- betrifft statisch getypte Programmiersprachen
  - z.B. Java, C++, C#
- Definition eines DT enthält Typvariable
- Ziel: Verwendung eines DT (Datenstruktur) mit verschiedenen Typen
- prinzipiell 2 Möglichkeiten
  - derselbe Code für jeden konkreten Typ und dynamische Bindung
    - Java (nur Objekttypen!), C#
  - Ersetzung des Typparameters mit dem konkreten Typ
    - C++, eingeschränkt: C# (bei Werttypen)
- Möglichkeiten bzw. Komplexität steigend: Java $\rightarrow$ C# $\rightarrow$ C++

Generische Programmierung

- Definition einer Funktion (oder auch Klasse samt Methoden) enthält Typvariablen
  - aber: unabhängig von Klassen oder Vererbung!
- Ziel: Verwendung einer Funktion mit verschiedenen Typen
- Beispiel: Entwicklen einer Funktion `add(x, y)`
  - Lösung in C++ mit mehreren überladenen Funktionen:

```
int add(int x, int y) {
    return x + y;
}
double add(double x, double y) {
    return x + y;
}
// ...
```

  - $\rightarrow$ immer der ”gleiche” Code!

Generische Programmierung – 2

- Beispiel:
  - Lösung mit einem (Funktions)Template:

```
// Voraussetzung: Operator + ist überladen
// anderenfalls Compilerfehler!
template <typename T>
T add(T x, T y) { return x + y; }

int main() {
    cout << add(1, 2) << endl;
    cout << add(1.0, 2.0) << endl;
    // cout << add("abc", "def") << endl;  // -> error
    cout << add(string{"abc"}, string{"def"}) << endl;
}
```

- $\rightarrow$ Reduzierung der Implementierung

(Template) Meta-Programming

```cpp
#include <iostream>

using namespace std;
using ull = unsigned long long;

template <ull n>
struct Factorial {
    static constexpr ull value{n * Factorial<n - 1>::value};
};

template <>
struct Factorial<0> {
    static constexpr ull value{1};
};

int main() {
    Factorial<0> f0;
    cout << f0.value << endl;  // -> 1
    cout << Factorial<1>::value << endl;  // -> 1
    cout << Factorial<2>::value << endl;  // -> 2
    cout << Factorial<64>::value << endl;  // -> 9223372036854775808
}
```