C/C++ Materialpaket (Level AB)
04b_VAL – Value Types

Prof. Dr. Carsten Link

Zusammenfassung

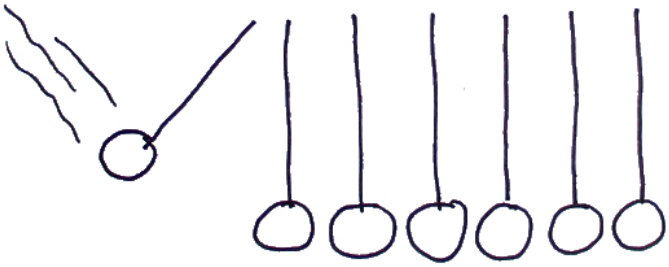

Abbildung 1: In der Newton-Wiege wird der Impuls durch elastische Stöße weitergegeben – auch dessen Wert.

Inhaltsverzeichnis

1 Kompetenzen und Lernegebnisse **2**

2 Konzepte **2**
2.1 Ausdrücke und Werte 2
2.2 Reguläre Werttypen (value types that “behave like int”) 3
2.3 Werttypen mit Konstruktoren zur Initialisierung 3
2.4 Member Initializer 3
2.5 Typumwandlungen 4
2.6 Beispiel für einen vollständigen Werttypen 4
2.7 (Level C) Rule of 0-3-5 7

3 Material zum aktiven Lernen **7**
3.1 Aufgabe: Grundgerüst 7
3.2 Aufgabe: Modifikationen 7
3.3 Verständnisfragen 8

4 Nützliche Links **9**

5 Literatur **9**

1 Kompetenzen und Lernegebnisse

Durch das Bearbeiten dieses Materialpaketes erwerben Sie diese Kompetenzen (Wissen, Fähigkeiten und Fertigkeiten zur Problemlösung):

Sie können mit Hilfe von speziellen Funktionen beeinflussen, wie Werte von benutzerdefinierten Datentypen erzeugt, geändert, kopiert und gelöscht werden.

Die oben genannten Kompetenzen erwerben Sie, indem Sie Lernziele erreichen, welche sich prüfen lassen. Lernegebnisse: Sie können nachweislich[1]:

- Konstruktoren und Destruktoren definieren
- Member Initializer verwenden
- Operatoren definieren für Typumwandlung und Kopien
- einen Datentypen derart gestalten, dass er sich in der Benutzung ähnlich verhält wie `int`

2 Konzepte

Werte sind Ausprägungen von Werttypen und stellen eine wichtige Grundlage für C++-Programme dar. Im folgenden wird dargelegt, wie benutzerdefinierte Datentypen ausgestattet werden müssen, damit sie einfach und ohne das Prinzip der geringsten Überraschung zu verletzen zu verwenden sind.

2.1 Ausdrücke und Werte

Ausdrücke sind Sprachkonstrukte, die einen Wert liefern. Werte können zugewiesen und kopiert werden.

```
1 int k=0;
2 int m=1;
```

Die Ausdrücke `k`, `m = k`, `k+1` und `m = k + 1` liefern jeweils einen Wert. Hierbei ist zu beachten, dass `k+1` einen anonymen temporären Wert liefert, der wieder gelöscht wird, sobald er nicht mehr benötigt wird.

[1] Sie können das Erzielen der einzelnen Lernergebnisse beispielsweise bei einem Testat im Praktikum oder einer Aufgabe in der Modulprüfung nachweisen.

2.2 Reguläre Werttypen (value types that "behave like int")

Das C++-Konzept *Regular Type* fordert, dass Ausprägungen von Typen kopierbar, ==-vergleichbar und default-konstruierbar sein müssen.

```
1 bool equal(int i, int k){
2   int zero;
3   zero = 0;
4   return  i == k + zero;
5 }
```

Werttypen, die dem Konzept Regular Type folgen, können wie `int`s verwendet werden: es lassen sich Variablen davon anlegen, Werte können als Parameter oder Rückgabewerte verwendet werden, Werte können in Arrays oder `std::`-Containern gespeichert werden etc. Im Code oben sind diese Eigenschaften in den Zeilen 1, 2 und 4 zu sehen.

2.3 Werttypen mit Konstruktoren zur Initialisierung

Neben der Regular-Type-Eigenschaft default konstruierbar zu sein, ist es sinnvoll, Werttypen mit Konstruktoren zur Initialisierung auszustatten. Beispiele hierfür finden sich bei dem Typen `RationalNumber` (in `src-cpp-student/04a_UDEF/RationalNumber.hpp`):

```
1 struct RationalNumber{
2     int zaehler;
3     int nenner;
4   RationalNumber();
5   RationalNumber(int z);
6   RationalNumber(int z, int n);
7   // ...
8 };
```

Neben dem default-Konstruktor sind hier zwei weitere Konstruktoren angegeben, die zur Initialisierung verwendet werden können. Beispielsweise `RationalNumber r(5);` oder `RationalNumber r2(355,113)`.

2.4 Member Initializer

Mit der mit `:` beginnenden *member initializer list* kann der Compiler angewiesen werden, bestimmte Konstruktoren mit bestimmte Parametern für die Initialisierung von Member Variablen zu verwenden:

```
1 RationalNumber::RationalNumber()
2 : zaehler(0), nenner(1)
3 { }
4
```

```
RationalNumber::RationalNumber(int z)
: zaehler(z), nenner(1)
{ }

RationalNumber::RationalNumber(int z, int n)
: zaehler(z), nenner(n)
{ }
```

Da nun die Memberwerte direkt mit dem vorgesehenen Wert konstruiert werden, entfallen die beiden Schritte default-Konstruktion mit nachfolgender Zuweisung (siehe Kapitel `05c_OO_CYCL`).

2.5 Typumwandlungen

Konstruktoren, die nur einen Parameter erwarten, können vom Compiler für implizite Typumwandlungen genutzt werden (beispielsweise `RationalNumber r; r = 17;`). Meist ist dieses Verhalten nicht erwünscht, daher sollten solche Konstruktoren mit dem `explicit`-Spezifizierer markiert werden.

Die Umwandlung in die umgekehrte Richtung ist auch möglich:

```
struct RationalNumber{
  // ...
  explicit RationalNumber(int z);
  explicit operator double();
  // ...
};
```

2.6 Beispiel für einen vollständigen Werttypen

```
#include "println.hpp"
#include <random>

//typedef int Byte;       /** prove "struct Byte behaves like int" */
// or
#include "Byte.hpp"   /** struct Byte { uint8_t bits; // ...  }; */

/// invokes copy ctor for creation of b
void printByte(Byte b){
  println("printByte()  b = ", b);
}

/// copy ctor for return or
/// copy ellision / RVO return value optimization
Byte randomByte(){
  /**std::uniform_int_distribution<> d(0, 255);
```

```
17    std::random_device r;
18    std::mt19937 gen(r());
19    return Byte(d(gen));
20    */
21    return Byte(17);   /// chosen by fair dice roll --
22                       /// guaranteed to be random
23                       /// see https://xkcd.com/221/
24  }
25
26  int main(){
27    println("111111111111111111111111");
28    Byte b1;            // default ctor
29
30    println("\n222222222222222222222222");
31    Byte b2(0xfe);      // ctor unsigned int
32
33    println("\n333333333333333333333333");
34    Byte b3=b2;         // copy ctor
35
36    println("\n444444444444444444444444");
37    Byte b4 = 0xfc;     // ctor unsigned int, copy ctor
38
39    println("\n555555555555555555555555");
40    b1 = b2;            // operator =
41
42    println("\n666666666666666666666666");
43    b4 = 23;            // implicit cast (ctor unsigned int), then copy ctor
44
45    println("\n777777777777777777777777");
46    if(b1 == b2){       // operator ==
47      println("operator== seems to work");
48    }
49
50    println("\n888888888888888888888888");
51    if(b4 < b3){        // operator <
52      println("operator< seems to work");
53    }
54
55    println("\n999999999999999999999999");
56    Byte b5 = randomByte();                 // ctor uint, copy return value, copy to b5
57
58    println("\naaaaaaaaaaaaaaaaaaaaaaaa");
59    printByte(b5);
60
61    println("\nbbbbbbbbbbbbbbbbbbbbbbbb");
```

```
62    Byte arr[7];
63
64    return 0;
65  }
```

Wenn `#include "Byte.hpp"` aktiv ist und mit `-DTRACE` übersetzt wird, zeigt sich folgende Ausgabe:

```
bash$ ./compile.sh main.cpp
bash$ ./a.out_Byte
111111111111111111111111
Byte::Byte()

222222222222222222222222
Byte::Byte(uint8_t u) u=254

333333333333333333333333
Byte::Byte(const Byte& src.bits=254

444444444444444444444444
Byte::Byte(uint8_t u) u=252
Byte::Byte(const Byte& src.bits=252

555555555555555555555555
Byte& Byte::operator=(Byte& src) src.bits=254

666666666666666666666666
Byte::Byte(uint8_t u) u=23
Byte& Byte::operator=(Byte& src) src.bits=23

777777777777777777777777
operator== seems to work

888888888888888888888888
operator< left=.bits23 right=.bits254
operator< seems to work

999999999999999999999999
Byte::Byte(uint8_t u) u=17
Byte::Byte(const Byte& src.bits=17
Byte::Byte(const Byte& src.bits=17

aaaaaaaaaaaaaaaaaaaaaaaa
Byte::Byte(const Byte& src.bits=17
printByte()  b = 0b10001000
```

```
bbbbbbbbbbbbbbbbbbbbbbbbb
Byte::Byte()
Byte::Byte()
Byte::Byte()
Byte::Byte()
Byte::Byte()
Byte::Byte()
Byte::Byte()
```

2.7 (Level C) Rule of 0-3-5

TBD

https://en.cppreference.com/w/cpp/language/rule_of_three

3 Material zum aktiven Lernen

Regelmäßiger Hinweis: Da eine Programmiersprache nur durch aktive Verwendung erlernt werden kann, werden im Folgenden Aufgaben zum praktischen Üben vorgestellt. Zunächst wird ein Grundgerüst (C/C++Programm) erstellt, welches dann auf mehrere Arten modifiziert wird. Insbesondere die Modifikationen ermöglichen es dem Lernenden (und auch dem Lehrenden), die Qualität des Kompetenzerwerbs bzgl. dieses Materialpakets bewerten zu können.

3.1 Aufgabe: Grundgerüst

Bereiten Sie ein Verzeichnis mit Quelldateien vor, mit welchen Sie die Modifikationen üben können.

3.2 Aufgabe: Modifikationen

Regelmäßiger Hinweis: Weiter unten ist eine Liste mit Modifikationen gegeben, die zwei Zwecken dienen: 1) Sie dienen als Richtschnur für das Praktizieren und Üben der Inhalte dieses Materialpakets. 2) Die Modifikationen können im Rahmen eines Testats als Aufgabe verwendet werden, durch deren Lösung Studierende nachweisen können, dass sie den Stoff dieses Materialpakets beherrschen. Stellen Sie sicher, dass Sie jede einzelne der nachfolgenden Modifikationen innerhalb weniger Minuten (ca. 5 - 10) vor Zuschauern (Testatsituation) umsetzen können. Konkret sollen Sie im Testat in der Lage sein, das gegebene Grundgerüst um mindestens eine zufällig ausgewählte Modifikation zu erweitern. Bereiten Sie dazu auf ihrer Arbeitsumgebung ein Verzeichnis vor, welches ausschließlich das Grundgerüst enthält. **Arbeiten Sie also auf einer Kopie des Verzeichnisses `src-cpp-student/`!** Achten Sie darauf, dass der Text auf Ihrem Bildschirm in heller Umgebung aus einem Meter Abstand heraus gut lesbar ist (light mode, große Schrift).

Modifikationen:

1. Implementieren Sie in `struct Byte` einen der bit shift-Operatoren `<<` oder `>>` selbst mit Iteration.
2. In struct Byte eine Methode hiervon selbst implementieren:
   - `setBit(int n)`
   - `clearBit(int n)`
   - `invertBit(int n)`
3. Erweitern Sie PascalStr zu einem vollständigen Werttypen und fügen die beiden Konstruktoren für `char` und `std::string` hinzu.
4. (Level C) Aufgabe zuvor, aber mit dynamischer Verwaltung des `char`-Arrays (also mit `new` und `delete`).

3.3 Verständnisfragen

Nach Bearbeitung des Kapitels "Konzepte", der Erstellung des Grundgerüsts sowie dem Üben der Modifikationen sollten Sie in der Lage sein, die folgenden Fragen zu beantworten.

1. Warum sollten Konstruktoren, die genau einen Parameter erwarten, mit `explicit` markiert werden?
2. Welchen Vorteil hat die Verwendung der Member Initializer List bei der Konstruktion von Werten?
3. Analysieren Sie die nachfolgenden Zeilen. Was macht der Compiler und was passiert zur Laufzeit? Nehmen Sie an, dass der Quelltext Fehlerfrei compiliert.

```
1 RgbColor c1;
2 RgbColor c2(2);
```

4. (Level C) Analysieren Sie die nachfolgenden Zeilen. Was macht der Compiler und was passiert zur Laufzeit? Nehmen Sie an, dass der Quelltext Fehlerfrei compiliert.

```
1 RgbColor c1;          // siehe oben
2 RgbColor c2(2);       // siehe oben
3 RgbColor c3=3;
4 c3=4;
5 RgbColor c5=c3;
6 RgbColor carr[7];
7 carr[0] = c5;
8 FancyColor f; // derived from RgbColor
9 RgbColor c6=f;
```

4 Nützliche Links

- The Standard C++ Foundation, FAQ, Constructors: https://isocpp.org/wiki/faq/ctors
- C++ Core Guidelines (by Bjarne Stroustrup and Herb Sutter), Make concrete types regular: https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-regular

5 Literatur

- [PPP] Stroustrup, Bjarne: Programming - Principles and Practice using C++
- [TCPL] Stroustrup, Bjarne: The C++ Programming Language, Fourth Edition