C/C++ Materialpaket (Level C)
04a_UDEF – User-defined Types

Prof. Dr. Carsten Link

Zusammenfassung

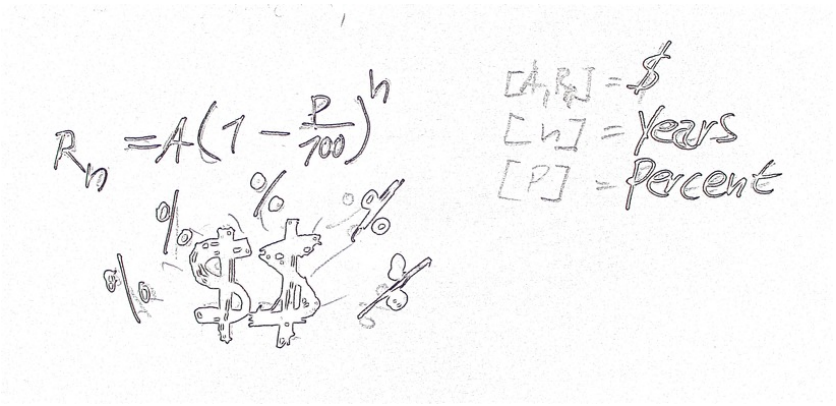

Abbildung 1: Problemangepasste Datentypen

Inhaltsverzeichnis

1 Kompetenzen und Lernegebnisse

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

Problemangepasste Datentypen implementieren und verwenden, um den Abstraktionsgrad von Quellcode anzuheben.

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

- den Zweck der Abstraktionsbildung erklären
- mittels benutzerdefinierter Datentypen die Abstraktionen, auf denen ihre Programme arbeiten, auf ein höheres Niveau bringen
- einen benutzerdefinierten Datentyp implementieren, so dass er in bestehendem Quelltext ohne wesentliche Änderungen verwendet werden kann (incl. operator overloading)
- implizite Typumwandlungen durch den Compiler nutzen und wissen um deren Seiteneffekte

2 Konzepte

Im Folgenden wird zunächst auf die Begriffe *Abstraktion* und *Abstraktionsbildung* eingegangen. Daraufhin wird das C++-Sprachmittel *benutzerdefinierte Datentypen* vorgestellt, das zur Abstraktionsbildung eingesetzt werden kann. Benutzerdefinierte Datentypen erlauben es dem Programmierer, Berechnungen auf Datentypen durchzuführen, welche in C++ an sich nicht enthalten sind.

2.1 Abstraktion und Abstraktionsbildung

Abstrakt ist das Gegenteil von *konkret*. Etwas Konkretes hat viele Details (Eigenschaften, welche benannt und ggf. beziffert werden können). Beispielsweise weist eine technische Beschreibung eines Multitransfunktionsdeformators (Sendung[2] vom 1. April) viele Eigenschaften und Zusammenhänge auf – eine Beschreibung des Begriffs *Maschine* hingegen wenige. Ein Multitransfunktionsdeformator ist also eine konkrete Ausprägung einer abstrakten Maschine. Anders gesagt:

- der Begriff Maschine abstrahiert von den Details einer konkreten Maschine
- das Abstraktionsniveau von *Maschine* ist höher als das von *Multitransfunktionsdeformator*

Die Verwendung von Abstraktionen hat den wesentlichen Vorteil, dass sich um viele Details keine Gedanken gemacht werden muss, da diese schlicht nicht vorhanden sind.

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

[2] http://www.ndr.de/info/Multitransfunktionsdeformator,audio31606.html

Um den Nutzen höherer Abstraktionsniveaus bei der Programmierung zu illustrieren wird hier ein Beispiel aus dem Modellbau gegeben.

In einem Metallbaukausten befinden sich Schrauben, Muttern und gelochte Blechstreifen. Aus diesen können (insbesondere von Kindern) Modelle von Baggern, Kränen und Flugzeugen gebaut werden. Die Bauteile sind also universell einsetzbar, ihr Abstraktionsniveau ist niedrig. Die zusammengebauten Resultate sind nur sehr grobe Abbilder ihrer Originale.

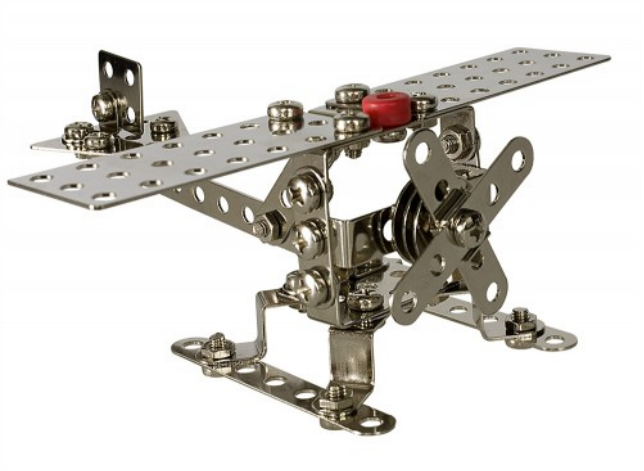

Abbildung 2: Aus universellen Bauteilen lässt sich vieles bauen – auch ein Flugzeug[4]

Im Gegensatz zu einem Metallbaukasten enthält ein Modellbausatz speziell gefertigte Teile, die genau auf das zu konstruierende Modell angepasst sind. Beispielsweise sind für ein Flugzeugmodell Flügel- und Rumpfteile vorhanden. Aufgrund der speziellen Anpassung fällt der Zusammenbau leichter; geht also schneller, mit geringerem Einsatz kognitiver Ressourcen und ist weniger Fehleranfällig.

[4]Quelle: STEMfinity https://www.stemfinity.com/Eitech-Basic-Mini-Helicopter-and-Airplane-Construction-Set

[5]Quelle: Bill Abbod, Flickr ,CCBY-SA2.0, https://www.flickr.com/photos/wba-iv/4828625203/

Abbildung 3: Modelbausatz einer Boeing 737-800[5]

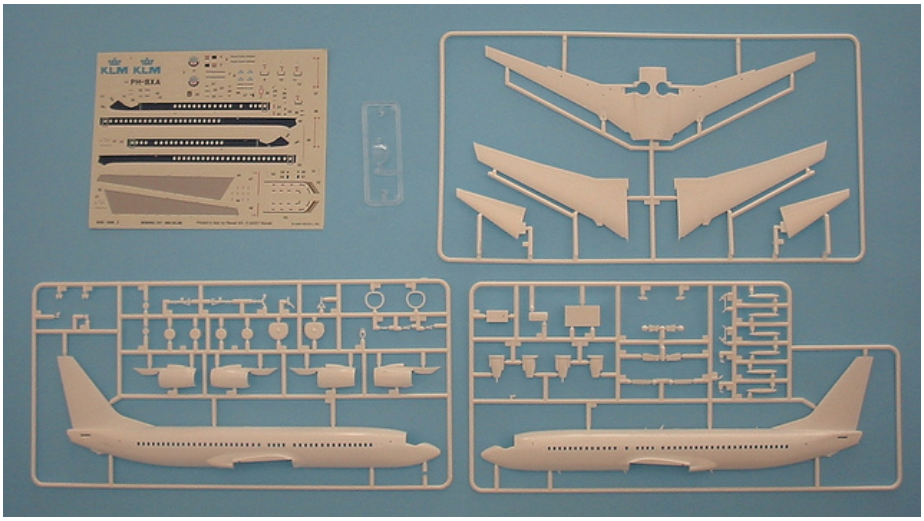

Abbildung 4: Die Bauteile, welche sich ausschließlich zum Bau eines Mig-3-Modells eignen. Für diesen Zweck sind sie besonders angepasst und geeignet. Die meisten Teile sind für den Bausatz Mig-3 bestimmt. Einige Teile sind jedoch in mehreren Bausätzen wiederverwendbar – wie beispielsweise die Turbinen und die Räder[7].

Bezogen auf die C++-Programmierung bedeutet dies: Eingebaute Datentypen sind universell einsetzbar und stellen streng genommen keine Abstraktionen dar. Erst durch die Namensgebung und die Beschreibung wird aus einem `int` eine konkrete Abstraktion `int age; // the students' age`, die in einem Programm ein kleines Bauteil darstellt. Mit den primitiven Datentypen läßt sich jedes Programmierbare umsetzen, wobei der resultierende Quellcode typischerweise schwer lesbar und fehleranfällig ist.

Mit dem C++-Sprachmittel der benutzerdefinierten Datentypen wird Programmierern die Möglichkeit gegeben, Abstraktionen für ihr gegebenes Programmierproblem zu bilden, die ein hohes Abstraktionsniveau aufweisen. Daraus resultiert gut lesbarer Quelltext und durch die damit einhergehende leichtere Verständlichkeit eine geringere Fehlerdichte.

Der Programmierer nimmt hier oftmals zwei Rollen ein: 1) Nutzer von Abstraktionen, die andere Programmierer erstellt haben (aus Bibliotheken) und 2) Ersteller von Abstraktionen zur späteren Verwendung im eigenen Programm.

2.2 Funktionen vs. Operatoren

Es wurden in vorangegangenen Codebeispielen bereits Operatoren für Berechnungen verwendet. So wurde in Materialpaket 01a_SIMPL eine Polynomfunktion der Art `y = (x-1) * (x-2) * (x-1)` verwendet, in der die Operatoren `=`, `-` und `*` vorkommen.

Berechnungen mit Funktionen: Eine Programmiersprache könnte aber durchaus auch ohne Operatoren auskommen. Um die Vorteile der Unterstützung von Operatoren in Programmiersprachen deutlich zu machen, zunächst ein Beispiel, wie ohne Operatoren programmiert werden kann:

```
1  // y = a+b*c
2  // step 1: use functions
3  int            addIntegers(int left, int right);
4  double         addDoubles(double left, double right);
5  RationalNumber addRational(RationalNumber left, RationalNumber right);
6  int            multiplyIntegers(int left, int right);
7  double         multiplyDoubles(double left, double right);
8  RationalNumber multiplyRational(RationalNumber left, RationalNumber right);
9
10 void add_int() {
11   int y, a, b, c;
12   y = addIntegers(a, multiplyIntegers(b, c))
13 }
14
15 void add_double() {
```

[7] Quelle: Bill Abbod, Flickr ,CCBY-SA2.0, https://www.flickr.com/photos/wbaiv/4828625203/

```
16    double y, a, b, c;
17    y = addDoubles(a, multiplyDoubles(b, c))
18  }
19
20  void add_RationalNumber() {
21    RationalNumber y, a, b, c;
22    y = addRationalNumber(a, multiplyRationalNumber(b, c))
23  }
```

Oben ist zu sehen, dass für jeden Typen, jeweils Funktionen zur Addition und Multiplikation zur Verfügung gestellt werden müssen. Für die drei Typen `int`, `double` und `RationalNumber` sowie die zwei Rechenoperationen `+` und `*` sind sechs Funktionen notwendig, die jeweils verschieden Namen und Parameter haben.

Berechnungen mit überladenen Funktionen: Im nächsten Schritt, werden Funktionen verwendet, welche zwar verschiedene Parameterlisten haben, jedoch dieselben Namen haben (genannt: überladene Funktionen):

```
1   // y = a+b*c
2   // step 2: use overloaded functions
3   int             add(int left, int right);
4   double          add(double left, double right);
5   RationalNumber  add(RationalNumber left, RationalNumber right);
6   int             multiply(int left, int right);
7   double          multiply(double left, double right);
8   RationalNumber  multiply(RationalNumber left, RationalNumber right);
9
10  void add_int() {
11    int y, a, b, c;
12    y = add(a, multiply(b, c));
13  }
14
15  void add_double() {
16    double y, a, b, c;
17    y = add(a, multiply(b, c));
18  }
19
20  void add_RationalNumber() {
21    RationalNumber y, a, b, c;
22    y = add(a, multiply(b, c));
23  }
```

Oben sind für die zwei Rechenoperationen `+` und `*` und die drei Typen weiterhin sechs Funktionen notwendig, allerdings werden nur zwei Funktionsnamen benötigt. Der Ausdruck `y = add(a, multiply(b, c))` ist nun für alle Typen gleich.

C++ erlaubt es, eine Funktion mehrfach zu deklarieren und zu definieren, wobei jeweils der Funktionsname gleich, die Signatur jedoch unterschiedlich ist (bzgl. der formalen Parameter)[8]. Beispielsweise können in einem Programm die Funktionen `int foo(int)`, `double foo(double)` und `RationalNumber foo (RationalNumber)` nebeneinander verwendet werden. Dabei wird der Compiler an den jeweiligen Aufrufstellen diejenige Funktion aufrufen, die zu den Typen der Parameter passt, die zur Übersetzungszeit ermittelt werden.

Berechnungen mit überladenen Operatoren: In C++ ist es darüber hinaus möglich, Operatoren für benutzerdefinierte Datentypen zu überladen. Daher wird der für die Berechnung von `y = (x-1) * (x-2) * (x-1)` mit drei verschiedenen Typen nur dieser Code benötigt:

```
1  // y = a+b*c
2  //  step 3: use overloaded operators
3  RationalNumber  operator+(RationalNumber left, RationalNumber right);
4
5  void add_int() {
6    int y, a, b, c;
7    y = a + b * c;
8  }
9
10 void add_double() {
11   double y, a, b, c;
12   y = a + b * c;
13 }
14
15 void add_RationalNumber() {
16   RationalNumber y, a, b, c;
17   y = a + b * c;
18 }
```

Nun ist der Ausdruck `y = a + b * c` ist nun für alle Typen gleich und sieht genauso aus, wie der Programmierer es erwarten würde – auch für den benutzerdefnierten Datentypen `RationalNumber`.

Ein überladener Operator ist eine Funktion, die einen speziellen Namen hat – nämlich den eines Operators (z.B. `+=`). Die Operatoren, die überladen werden können und für dieses Materialpaket relevant sind, sind diese:

- arithmetische Operatoren: `+`, `-`, `*`, `/`
- Zuweisungsoperator und kombinierte Zuweisungen: `=`, `+=`, `-=`, `*=`, `/=`
- Vergleichsoperatoren: `<`, `<=`, `>`, `>=`, `==`, `!=`
- subscript operator (Indexoperator): `[]`
- cast operator (Typumwandlungsoperator): `operator <conversion-type-id>` und `explicit operator <conversion-type-id>`

[8] Überladen anhand des Typs des Rückgabewertes allein ist nicht möglich.

Weitere Details und Standardimplementierungen sind auf CppReference operator overloading[9] angegeben.

2.3 Berechnungen mit verschiedenen eingebauten Datentypen

Um später die Verwendung von benutzerdefinierte Datentypen vorstellen zu können wird das folgende Beispielprogramm verwendet, welches zunächst die eingebauten Datentypen `int` und `double` verwendet:

```
#include "println.hpp"

typedef int calctype;  // this declares an alias for a typename
#define CREATE(x) x    // needed for initialization of new values

//typedef double calctype;
//#define CREATE(x) (x*1.0)

//#include "RationalNumber.hpp"     // location 1
//typedef RationalNumber calctype;  // location 2
//#define CREATE(x) {x,1}

void printMinMax(calctype values[], int length){
  calctype minimum = values[0];
  calctype maximum = values[0];
  for(int i=1; i<length; i++){
    if (values[i] < minimum){
      minimum = values[i];
    }
    if (values[i] > maximum){
      maximum = values[i];
    }
  }
  println("minimum=", minimum, " maximum=", maximum);
}

int main(){
  calctype values[numValues];
  for(int i=0; i<numValues; i++){
    values[i] = CREATE(5+i*7);
  }
  printMinMax(values, numValues);
  return 0;
}
```

[9] https://en.cppreference.com/w/cpp/language/operators

Zunächst wird in Zeile 13 die Funktion `printMinMax()` definiert, welche in einem gegebenen Array den kleinsten und den größten Wert berechnet und jeweils ausgibt. Durch die Verwendung des Typalias `typedef int calctype;` kann die Definition von `printMinMax()` unabhängig von konkreten Typen gemacht werden. Auf diese Weise wird das Abstraktionsniveau von dieser Funktion erhöht. Sie kann auf verschiedenen Datentypen operieren (hier `calctype` genannt), solange sichergestellt ist, dass die verwendeten Operatoren und Funktionsaufrufe für die tatsächlichen Typen (im Beispiel `int`) definiert ist. In dem oben angegebenen Code werden lediglich die eingebauten primitive Datentypen `int` und `double` verwendet.

2.4 Berechnungen mit benutzerdefinierte Datentypen

In C++ können Datentypen mit Strukturen (`struct`) vom Programmierer definiert werden. Wie oben motiviert wurde, sind solche benutzerdefinierten Datentypen sinnvoll, um das Abstraktionsniveau der im Quelltext verwendeten Datentypen an die Problemdomäne anzupassen.

Eine besonders nützliche Eigenschaft von benutzerdefinierten Datentypen in C++ ist: sie können derart gestaltet werden, dass für deren Benutzung im Quelltext keine Umgewöhnung bzgl. der Syntax notwendig ist. Dieser Sachverhalt wird im oben gegeben Beispiel veranschaulicht.

Es wird der Benutzerdefinierte Datentyp `RationalNumber` eingeführt, der der Darstellung von rationalen Zahlen (als Verhältnis ganzer Zahlen darstellbare Zahlen) dienen soll.

```
1  #include <iostream>
2
3  struct RationalNumber{
4      int zaehler;
5      int nenner;
6  };
7
8  RationalNumber addRationalNumbers(RationalNumber left, RationalNumber right);
9  RationalNumber operator+ (RationalNumber left, RationalNumber right);
10 bool operator< (RationalNumber left, RationalNumber right);
11 std::string as_string(RationalNumber);  // for println()
```

Obiger Header kann mit `#include "RationalNumber.hpp"` in den ersten Quelltext an der Stelle `// location 1` eingebunden werden. Die Implementierung der `struct RationalNumber` findet sich in `RationalNumber.cpp`:

```
1  #include "RationalNumber.hpp"
2
3  RationalNumber addRationalNumbers(RationalNumber left, RationalNumber right){
4      RationalNumber result;
5      // add left and right
```

```
6       return result;
7   }
8
9   RationalNumber operator+ (RationalNumber left, RationalNumber right){
10      return addRationalNumbers(left, right);
11  }
12
13  bool operator< (RationalNumber left, RationalNumber right){
14    return (1.0*left.zaehler/left.nenner)<(1.0*right.zaehler/right.nenner);
15  }
16
17  // for println()
18  std::string as_string(RationalNumber r){
19    return std::to_string(r.zaehler) + "/" + std::to_string(r.nenner);
20  }
```

Wird nun im Beispielcode an der Stelle `// location 2` der zu verwendende Datentyp per `typedef RationalNumber calctype;` umgestellt, so wird für die Berechnung in `printMinMax()` unser selbstdefinierter problemangepasster Datentyp `RationalNumber` verwendet. Hinweis: die Umschaltung geschieht über Präprozessordefinitionen, die dem Compiler mitgegeben werden (siehe `build.sh`).

Da der Typ `RationalNumber` benutzerdefiniert und nicht in den Compiler eingebaut ist, kann der Compiler nicht wissen, wie Operatoren auf Werten dieses neuen Datentyps anzuwenden sind. Beispielsweise 'weiß' der Compiler nicht, wie zwei Werte addiert werden sollen, also welchen Assembler-Code er für `a + b` generieren soll – dies weiß nur der Programmierer des Datentyps (Programmierer ist der Benutzer in "benutzerdefinierter Datentyp"). Glücklicherweise muss der Programmierer keinen Assembler-Code angeben, sondern kann im C++-Code definieren, wie `a + b` berechnet werden soll: die in Zeile 10 angegebene Überladung des `+`-Operators definiert, wie die Addition zweier Werte des Typs `RationalNumber` auszuführen ist. Beachten Sie hier, dass der `+`-Operators mittels der Funktion `addRationalNumbers()` definiert ist. Letztere existiert lediglich, um zu illustrieren: gäbe es keine Möglichkeit Operatoren zu überladen, so müssten benutzerdefinierte Datentypen im Quelltext anders addiert werden als eingebaute Datentypen. Statt `c = a + b` müsste geschrieben werden `c = addRationalNumbers(a, b)`. Programmierer wären gezwungen, abhängig vom jeweiligen Datentyp einen anderen Programmierstil anzuwenden und dem Compiler wäre es nicht möglich, zu erraten, welche Funktion er zur Addition aufrufen kann.

Hinweis: Die Funktion `operator<<` ist nötig, um Werte des Typs `RationalNumber` an Streams (z.B. `std::cout`) ausgeben zu können. Falls Sie die Komfortfunktion `println()` verwenden, müssen Sie eine Funktion `std::string as_string(RationalNumber r)` definieren, wie in diesem Beispiel mit `struct twoInts`:

```
struct twoInts {
  int a, b;
};

std::string as_string(twoInts t){
  return std::string("twoInts(") + as_string(t.a) + "," + as_string(t.b) + ")";
}
```

Diese Funktion erlaubt es nun, einen `twoInts`-Wert `t` mit `println(t)` auszugeben (z. B. als `twoInts(1,2)`).

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

Nehmen Sie die Dateien `RgbColor.cpp` sowie `main_rgbColor.cpp` und compilieren diese (Instanzen des Typs `RgbColor` stellen Farbwerte im Rot-Grün-Blau-Farbraum dar). Setzen Sie folgendes um:

1. Fügen Sie dem Datentyp `RgbColor` die fehlenden Konstruktoren hinzu.
2. Fügen Sie dem Datentyp `RgbColor` die fehlenden Operatorüberladungen hinzu, so dass der Quelltext ohne Fehler (und Warnungen!) compiliert.
3. Bauen Sie die Methode `uint8_t RgbColor::luminosity() const;` zur Berechnung der Helligkeit ein, wobei diese mit `0.21 R + 0.72 G + 0.07 B` berechnet wird. Diese Methode kann später von Vergleichsoperatoren verwendet werden

Beachten Sie bei der Implementierung der Methoden und Operatoren folgendes:

- beim Addieren sollte jeweils der Mittelwert der Grundfarben gebildet werden
- (Level C) die beiden `operator++` schieben die Bits der Grundfarbwerte um eine Position nach links und füllen Einsen nach

```

```

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. Aktivieren Sie `typedef RationalNumber T_calc;`. Fügen Sie nun in `printMinMax()` die Berechnung des Durchschnittswertes hinzu. Was geschieht? Wie können Sie das Problem beheben? Beheben Sie das Problem!
2. Verwenden Sie in `printMinMax()` den Typ `RgbColor`. Hinweis: dazu sollten Sie auch `build.sh` analog zu `RationalNumber` erweitern und für `FROM_INT(x)` das Konstrukt `RgbColor( (x) | (x)<<8 | (x)<<16)` verwenden.
3. (Level C) fügen Sie dem Typ `RgbColor` den Präfix-Decrementoperator hinzu und nutzen diesen in `main_rgbColor.cpp`
4. (Level C) fügen Sie den Typumwandlungsoperator hinzu (mit `explicit` markiert), der für den Aufruf von `void foobar(double)` mit einer Variable des Typs `RgbColor` benötigen (d.h. `RgbColor b; foobar(static_cast<double>(b));`)
5. (Level C) implementieren Sie die sechs notwendigen Funktionen, um zwischen `double`, `int` und `std::string` hin- und herwandeln zu können. Dabei ist der Funktionsname immer `convertInto` und der erste Parameter wird per Referenz übergeben, da er als Rückgabewert fungiert (da kein Überladen per Typ des Rückgabewertes möglich ist)
6. (Level C), freiwillig
   - fügen Sie einen Literal-Operator hinzu, der ein `RgbColor`-Objekt zurückgibt
   - fügen Sie einen Konstruktor hinzu, der ein Objekt vom Typ `std::initializer_list<>` akzeptiert

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. Sortieren Sie die folgenden Begriffe in die Reihenfolge von hohem Abstraktionsniveau zu konkret: Ford Mustang, Sprinter, Nutzfahrzeug, PKW, Fahrzeug, LKW. Begründen Sie Ihre Einordnung.
2. Welchen Zweck haben benutzerdefinierte Datentypen? Beziehungsweise: warum begnügt man sich bei der Implementierung von Programmierproblem nicht mit den in C++ eingebauten Datentypen?
3. Welche Aussagen können Sie bezüglich des Ausdrucks `a = b + c` machen?
4. Welchen Nutzen hat `typedef`?
5. Freiwillig: Zeigen Sie auf, inwiefern das Piktogram auf der Titelseite dieses Materialpaketes den Inhalt zusammengefasst darstellt.
6. (Level C) Spielen Sie in Gedanken das Szenario durch, in dem Sie die interne Darstellung von `RgbColor` umstellen auf drei `double`-Werte. Welche Code-Änderungen an dem Code, der den Datentyp `RgbColor` verwendet, sind notwendig? Wären diese Änderungen von vorn herein vermeidbar gewesen?

4 Nützliche Links

- Benjamin Buch, C++ Programmierung, wikibooks.de: Operatoren überladen[10]
- Wikibooks C++ Programming: Increment and decrement operators[11]
- Wikipedia, Operators in C and C++[12]
- CppReference.com: user-defined conversion[13]
- CppReference.com: operator overloading[14]
- CppReference.com: user-defined literals[15]
- Andrzej's C++ blog: User-defined literals — Part I[16]
- Bartosz Ciechanowski: Color Spaces[17]

5 Literatur

- [PPP] Stroustrup, Bjarne: Programming - Principles and Practice using C++

[10] https://de.wikibooks.org/wiki/C%2B%2B-Programmierung/_Eigene_Datentypen_definieren/_Operatoren_überladen
[11] https://en.wikibooks.org/wiki/C%2B%2B_Programming/Operators/Operator_Overloading#Increment_and_decrement_operators
[12] https://en.wikipedia.org/wiki/Operators_in_C_and_C%2B%2B
[13] http://en.cppreference.com/w/cpp/language/cast_operator
[14] https://en.cppreference.com/w/cpp/language/operators
[15] http://en.cppreference.com/w/cpp/language/user_literal
[16] https://akrzemi1.wordpress.com/2012/08/12/user-defined-literals-part-i/
[17] https://ciechanow.ski/color-spaces/

- [TCPL] Stroustrup, Bjarne: The C++ Programming Language, Fourth Edition