Materialpaket 04_UDEF – User-defined Types

C/C++, Autor: Prof. Dr.-Ing. Carsten Link

Version 1.3.1 March 3, 2019

Contents

1 Kompetenzen und Lernegebnisse

Durch das Bearbeiten dieses Materialpaketes erwerben Sie diese Kompetenzen (Wissen, Fähigkeiten, 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

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

- 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:

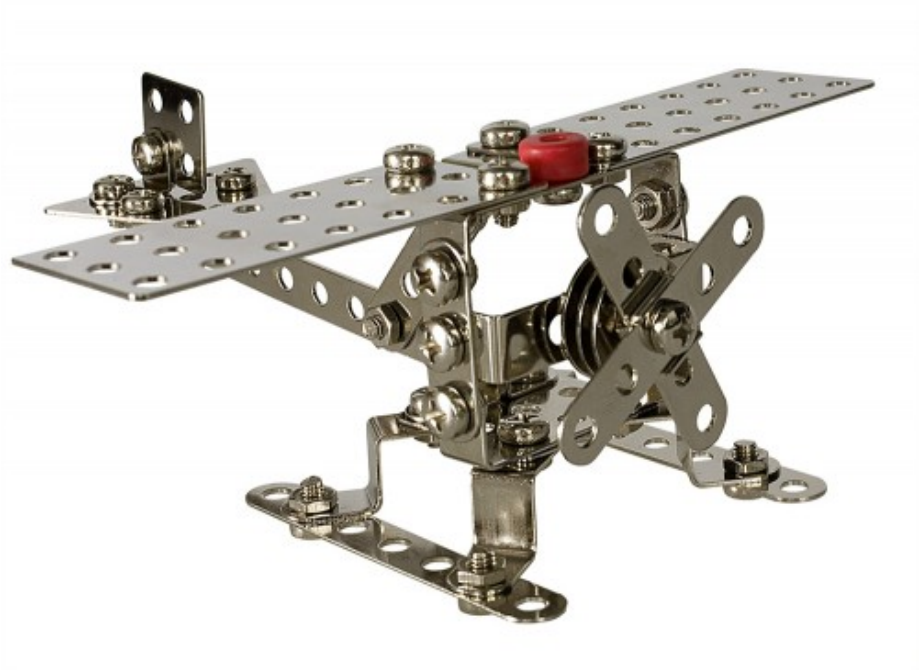
- der Begriff Maschine abstrahiert von den Details einer konkreten Maschine
- das Abstraktionsniveau von *Maschine* ist höher als das von *Multitransfunktionsdeformator*

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

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.

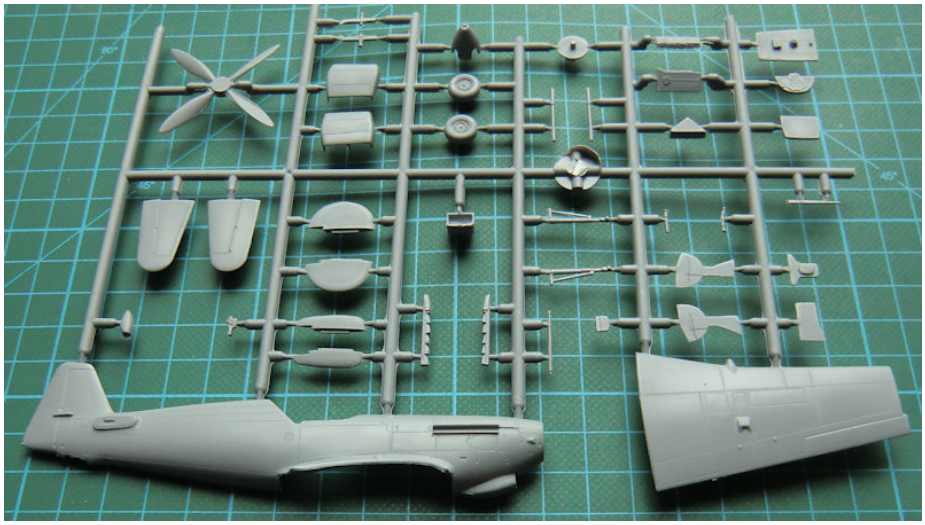
[2] http://www.ndr.de/info/Multitransfunktionsdeformator,audio31606.html



Aus universellen Bauteilen lässt sich vieles bauen - auch ein Flugzeug. Quelle: STEMfinity https://www.stemfinity.com/Eitech-Basic-Mini-Helicopter-and-Airplane-Construction-Set

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 Fehler-anfällig.

Ein Modellbausatz einer Mig-3. Quelle: IPMS-Deutschland http://www.ipmsdeutschland.de/FirstLook/Revell/Rev_MiG-3/Rev_MiG-3.html



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 der Propeller und die Räder. Quelle: IPMS-Deutschland http://www.ipmsdeutschland.de/FirstLook/RS_models/RS_Doflug_D-3802/RS_Doflug_D-3802.html

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 Pro-

gramm 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 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 folgenden Beispiel veranschaulicht:

```
1  // file: main_04_UDEF_a.cpp
2  #include "../helpers/println.hpp"
3  #include "RationalNumber.hpp"
4  #include <iostream>
5
6  // location 1
7
8  // this declares an alias for type <see below>, which is called calctype
9  //typedef unsigned int calctype;
10 //typedef int calctype;
11 //typedef double calctype;
12 typedef RationalNumber calctype;
13
14 // location 2
15
16 void doCalculation(calctype a, calctype b){
17     calctype c = a + b;
18   // here is some advanced computation on arguments a and b
19   // for(){ ...
20   //   if(){ ...
21   //     for(){ ...
```

```
22    println("a = ", a);
23    println("b = ", b);
24    println("c = ", c);
25  }
26
27  int main(int argc, char** argv, char** envp){
28      calctype a;
29      calctype b;
30      doCalculation(a,b);
31      return 0;
32  }
```

Zunächst wird in Zeile 14 die Funktion `void doCalculation(calctype a, calctype b)` definiert, welche irgendeine Berechnung durchführt. Durch die Verwendung des Typalias `typedef int calctype;` kann die Definition von `doCalculation` 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 `unsigned int`, `int` und `double` verwendet.

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

```
1  // file: RationalNumber.hpp
2  #include <iostream>
3
4  struct RationalNumber{
5      int zaehler;
6      int nenner;
7  };
8
9  RationalNumber addRationalNumbers(RationalNumber left, RationalNumber right);
10 RationalNumber 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  // file: RationalNumber.cpp
2  #include "RationalNumber.hpp"
3
4  RationalNumber addRationalNumbers(RationalNumber left, RationalNumber right){
5      RationalNumber result;
```

```
6       // add left and right
7       return result;
8   }
9
10  RationalNumber operator+ (RationalNumber left, RationalNumber right){
11      return addRationalNumbers(left, right);
12  }
13
14  // for println()
15  std::string as_string(RationalNumber r){
16    std::string result = "(";
17    result += std::to_string(r.zaehler);
18    result += "/";
19    result += std::to_string(r.nenner);
20    result += ")";
21    return result;
22  }
```

Wird nun an der Stelle `// location 2` der zu verwendende Datentyp umgestellt per `typedef RationalNumber calctype;` so wird für die Berechnung in `doCalculation()` unser selbstdefinierter problemangepasster Datentyp `RationalNumber` verwendet.

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 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.

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 `string as_string(RationalNumber r)` definieren, wie in diesem Beispiel mit `struct stru`:

```
1   struct stru {
2     int a,b;
```

```
};

std::string as_string(stru s){
  return std::string("str(") + as_string(s.a) + "," + as_string(s.b) + ")";
}
```

Diese Funktion erlaubt es nun, einen Wert `stru s` mit `println(s)` auszugeben.

2.3 Überladung von Operatoren

Zunächst muss der Begriff "überladen" geklärt werden. 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)[3]. Beispiel:

```
int foo(double d){
  return 0;
}

int foo(int i){
  return 0;
}

void bar(){
  foo(0);     // invokes int foo(int i);
  foo(0.0);   // invokes int foo(double d);
}
```

In obigem Beispiel wird der Compiler an der Aufrufstelle diejenige Funktion aufrufen, die zu den Typen der Parameter passt, die zur Übersetzungszeit ermittelt werden.

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>`

Vorübung: Nehmen Sie `main_mp3a.cpp` und aktivieren dort `typedef RationalNumber calctype;`. Fügen Sie nun in `doCalculation` eine Berech-

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

nung mit dem --Operator hinzu. Was geschieht? Wie können Sie das Problem beheben?

3 Material zum aktiven Lernen

3.1 Aufgabe: Grundgerüst

Ab diesem Materialpaket können Sie die im VirtualBox-VM-Image vorinstallierte integrierte Entwicklungsumgebung (IDE) CodeLite oder Code::Blocks verwenden. Einführung dazu Siehe

Achten Sie darauf, bei den Projekteinstellung den C++14-Standard zu aktivieren (unter Workspace - active project settings - global settings - C++ compiler options - ...; auf der Kommandozeile mit `clang++ -std=c++14 ...`).

Nehmen Sie den nachfolgenden Quelltext (Datei `main_04_UDEF_e.cpp`) und compilieren ihn. Fügen Sie dem Datentyp `BinaryOctet` die fehlenden Operatorüberladungen hinzu, so dass der Quelltext ohne Fehler (und Warnungen!) compiliert.

Instanzen des Typs `BinaryOctet` stellen jeweils ein Byte (8 Bits) mitsamt einem Paritätsbit dar. Dabei werden die 8 Bits des Octets[4] in 8 `char`s gespeichert (vgl. PascalString) und das Feld wird auf `true` gesetzt, falls die Anzahl der `1`-Bits gerade ist, andernfalls auf `false`.

```
1  // file main_04_UDEF_e.cpp
2  #include "../helpers/println.hpp"
3  #include <iostream>
4
5  const int bitsPerOctet=8;
6
7  struct BinaryOctet {
8    bool evenParity;   // set to true if number of '1' in bitsAsDigits is even, false otherwi
9    char bitsAsDigits[bitsPerOctet]; // bit values as chars
10   BinaryOctet(int x=0); // "x=0": if no parameter is proved, use zero as parameter => may s
11   BinaryOctet(const BinaryOctet&) = default;
12 };
13
14 BinaryOctet::BinaryOctet(int x){
15   // TODO: implement
16 }
17
18 BinaryOctet doCalculation(BinaryOctet a, BinaryOctet b){
19   BinaryOctet result;
```

[4] Die Begriffe Octet und Parität stammen aus der Netzwerkwelt.

```
20    for(; a != b; b--){
21      a = a + 1;
22      a = a / b;
23    }
24    result = a + b;
25    return result;
26  }
27
28  // for println();
29  std::string as_string(BinaryOctet a){
30    std::string result = "(";
31    // TODO: implement
32    result += ")";
33    return result;
34  }
35
36  int main(int argc, char **argv)
37  {
38    BinaryOctet a = 0b00001111;
39    BinaryOctet b = 0b00000110;
40    println("result = ", doCalculation(a,b));
41    return 0;
42  }
```

3.2 Aufgabe: Modifikationen

Stellen Sie sicher, dass Sie jede einzelne der nachfolgenden Modifikationen innerhalb weniger Minuten (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.

Modifikationen:

1. fügen Sie dem Typ `BinaryOctet` die Operatoren `+=` und `-=` hinzu und nutzen diese
2. fügen Sie den Typumwandlugsoperator hinzu, der für den Aufruf von `void foobar(double)` mit einer Variable des Typs `BinaryOctet` benötigen (d.h. `BinaryOctet b; foobar(b);`)
3. freiwillige Zusatzmodifikationen für besonders Interessierte:
   - fügen Sie einen Literal-Operator hinzu, der ein `BinaryOctet`-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, Nutzfahrzeug, PKW, Fahrzeug, LKW.
2. Begründen Sie Ihre Einordnung.
3. Welchen Zweck haben benutzerdefinierte Datentypen? Beziehungsweise: warum begnügt man sich bei der Implementierung von Programmierproblem nicht mit den in C++ eingebauten Datentypen?
4. Welche Schritte sind notwendig, um zwei nullterminierte Zeichenketten an einander zu hängen? Geben Sie mindestens zwei der möglichen Antworten.
5. Welche Aussagen können Sie bezüglich des Ausdrucks `a = b + c` machen?
6. Welchen Nutzen hat `typedef`?

4 Nützliche Links

- Benjamin Buch, C++ Programmierung, wikibooks.de: Operatoren überladen[5]
- Wikibooks C++ Programming: Increment and decrement operators[6]
- Wikipedia, Operators in C and C++[7]
- CppReference.com: user-defined conversion[8]
- CppReference.com: operator overloading[9]
- CppReference.com: user-defined literals[10]
- Andrzej's C++ blog: User-defined literals — Part I[11]

5 Literatur

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

[5] https://de.wikibooks.org/wiki/C%2B%2B-Programmierung/_Eigene_Datentypen_definieren/_Operatoren_überladen

[6] https://en.wikibooks.org/wiki/C%2B%2B_Programming/Operators/Operator_Overloading#Increment_and_decrement_operators

[7] https://en.wikipedia.org/wiki/Operators_in_C_and_C%2B%2B

[8] http://en.cppreference.com/w/cpp/language/cast_operator

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

[10] http://en.cppreference.com/w/cpp/language/user_literal

[11] https://akrzemi1.wordpress.com/2012/08/12/user-defined-literals-part-i/