Beispiel `roman2dec`

Dr. Günter Kolousek

14. Juli 2016

1 Allgemeines

- Drucke dieses Dokument **nicht** aus!
- Halte **unbedingt** die Coding Conventions ein! Zu finden am edvossh!
- Lege für dieses Beispiel wieder ein entsprechendes Unterverzeichnis mit Beispielnummer und Beispielname an!
- Verwende den vorgegebenen Texteditor!

2 Aufgabenstellung

Schreibe ein `C++` Programm `roman2dec`, das eine beliebige römische Zahl in das dezimale Äquivalent umwandeln kann.
Der Zeichenvorrat umfasst die folgenden Zeichen:

`M` 1000

`D` 500

`C` 100

`L` 50

`X` 10

`V` 5

`I` 1

Die römischen Ziffern werden hintereinnander angeordnet und ihre Werte aufaddiert. D.h. `III` entspricht der dezimalen Zahl 3.
Die Reihenfolge der römischen Ziffern muss so sein, dass die Werte der römischen Ziffern monoton fallend angeordnet sind. Das bedeutet, dass die Werte nicht größer werden dürfen. D.h. `IIVX` ist ungültig.
Von dieser Regel gibt es eine Ausnahme: Genau eine kleinere Ziffer darf vor einer höheren Ziffer stehen. D.h. `MCDVIII` ist 1408 oder `MCDIX` ist 1409, aber `MCCDVIII` ist keine gültige römische Zahl.
Allerdings muss auch immer die kleinste Schreibweise gewählt werden. D.h. `VV` ist als Darstellung für 10 ungültig, da dafür `X` geschrieben werden muss.
Dazu ist eine Funktion `unsigned int roman2dec(string roman)` zu schreiben, die die römische Zahl als String erhält und das dezimale Äquivalent zurückliefert.

3 Anleitung

1. Schreibe eine Funktion `int value(char digit)`, die für eine *gültige* römische Ziffer das dezimale Äquivalent zurückliefert. Verwende dazu eine `switch` - Anweisung mit einer entsprechenden Anzahl an `case` Klauseln.

2. Weiters nehmen wir an, dass wir die Funktion `value` allgemein verwenden wollen. In diesem Fall soll für eine *ungültige* römische Ziffer eine Exception `std::logic_error` mit einem sinnvollen Text geworfen werden.

   Natürlich sehen wir diesen Fall in unserem Programm eigentlich gar nicht gerne und genau aus diesem Grund werfen wir eine Exception (Ausnahme).

   Das Werfen der Ausnahme mit einem entsprechenden Text sollte so erfolgen:

   ```
   throw logic_error("invalid roman digit: " + string(1, digit));
   ```

   und **nicht** so:

   ```
   throw logic_error("invalid roman digit: " + digit;
   ```

   Warum die erste Version funktioniert aber die zweite Version nicht, werden wir später noch lernen.

3. Schreibe die Funktion `unsigned int roman2dec(string roman)`, die für eine *gültige* römische Zahl gemäß den Regeln das dezimale Äquivalent berechnet. D.h. eine Fehlerüberprüfung ist nicht notwendig! Allerdings kann die römische Zahl prinzipiell beliebig lang sein.

   Erweitere dein Programm, dass der Benutzer eine römischen Zahl eingeben kann und die gerade programmierte Funktion aufgerufen wird als auch danach das Ergebnis dem Benutzer in einer ansprechenden Form präsentiert wird.

   Gut, wenn du jetzt zu der Überzeugung gekommen bist, dass dein Programm soweit funktioniert, dann mache mit dem nächsten Punkt weiter.

4. Testen kannst du dein Programm jetzt mit Hilfe der folgenden Testdaten:

```
vector<pair<string, unsigned int>> valid_data{
    {"", 0},
    {"I", 1}, {"II", 2}, {"III", 3}, {"IV", 4}, {"V", 5},
    {"VI", 6}, {"VII", 7}, {"VIII", 8}, {"IX", 9}, {"X", 10},
    {"XI", 11}, {"XII", 12}, {"XIII", 13}, {"XIV", 14}, {"XV", 15},
    {"XVI", 16}, {"XVII", 17}, {"XVIII", 18}, {"XIX", 19}, {"XX", 20},
    {"XXI", 21}, {"XXII", 22}, {"XXIII", 23}, {"XXIV", 24}, {"XXV", 25},
    {"XXVI", 26}, {"XXVII", 27}, {"XXVIII", 28}, {"XXIX", 29}, {"XXX", 30},
    {"XXXI", 31}, {"XXXII", 32}, {"XXXIII", 33}, {"XXXIV", 34}, {"XXXV", 35},
    {"XXXVI", 36}, {"XXXVII", 37}, {"XXXVIII", 38}, {"XXXIX", 39}, {"XL", 40},
    {"XLI", 41}, {"XLII", 42}, {"XLIII", 43}, {"XLIV", 44}, {"XLV", 45},
    {"XLVI", 46}, {"XLVII", 47}, {"XLVIII", 48}, {"IL", 49}, {"L", 50},
    {"LI", 51}, {"LII", 52}, {"LIII", 53}, {"LIV", 54}, {"LV", 55},
    {"LVI", 56}, {"LVII", 57}, {"LVIII", 58}, {"LIX", 59}, {"LX", 60},
    {"LXI", 61}, {"LXII", 62}, {"LXIII", 63}, {"LXIV", 64}, {"LXV", 65},
    {"LXVI", 66}, {"LXVII", 67}, {"LXVIII", 68}, {"LXIX", 69}, {"LXX", 70},
    {"LXXI", 71}, {"LXXII", 72}, {"LXXIII", 73}, {"LXXIV", 74}, {"LXXV", 75},
    {"LXXVI", 76}, {"LXXVII", 77}, {"LXXVIII", 78}, {"LXXIX", 79}, {"LXXX", 80},
    {"LXXXI", 81}, {"LXXXII", 82}, {"LXXXIII", 83}, {"LXXXIV", 84}, {"LXXXV", 85},
    {"LXXXVI", 86}, {"LXXXVII", 87}, {"LXXXVIII", 88}, {"LXXXIX", 89}, {"XC", 90},
    {"XCI", 91}, {"XCII", 92}, {"XCIII", 93}, {"XCIV", 94}, {"XCV", 95},
    {"XCVI", 96}, {"XCVII", 97}, {"XCVIII", 98}, {"IC", 99}, {"C", 100},

    {"CXLVIII", 148}, {"CXCIII", 193}, {"CCCXC", 390}, {"CCCIC", 399},
    {"ID", 499}, {"D", 500}, {"MMMCM", 3900}, {"MMMCMXLIV", 3944}
};
```

Damit wird ein `vector` (so etwas wie eine Liste in Python) angelegt, der lauter Paare (`pair`, so etwas wie ein zweielementiges Tupel in Python) enthält. Jedes Paar besteht zuerst aus einem `string` und aus einer ganzen, positiven Zahl.

Der folgende Code zeigt dir an, ob deine Funktion auch ”funktioniert”:

```
for (auto p : valid_data) {
     assert(roman2dec(p.first) == p.second);
}
```

Diese Schleife funktioniert wie eine `for p in valid_data` Schleife in Python und `assert` überprüft zur Laufzeit, ob die angegebene Bedingung erfüllt ist (wenn das Programm im Debug-Modus übersetzt wird). Wenn nicht, dann wird das Programm abgebrochen.

Dafür ist das Inkludieren von `<utility>`, `<vector>` und `<cassert>` zwingend notwendig.

Wenn du diesen Testcode in dein Programm einbaust und es danach startest, darf dies keinerlei Abbruch deines Programmes zur Folge haben! Dann funktioniert (mit hoher Wahrscheinlichkeit) deine Funktion mit gültigen römischen Zahlen.

5. Ok, bis jetzt war es der einfache Teil dieses Beispieles. In weiterer Folge sollen die *ungültigen* römischen Zahlen erkannt werden und als Fehler gemeldet werden.

   Zur Überprüfung deines Erfolges stelle ich folgende Variable und nachfolgenden Code zur Verfügung:

```
vector<string> invalid_data{
    "A", "IIII", "VV", "VIIII", "IIV", "XIIII", "XIIIII", "IIX",
    "XXXX", "XXXXX", "XXIIIV", "LIVX", "LXVIV", "VIV", "LXL", "DD", "DCDII",
    "IVI", "LC", "DM"
};

for (auto item : invalid_data) {
    bool caught{};
    try {
        roman2dec(item);
    } catch (logic_error& e) {
        caught = true;
        cout << e.what() << endl;
    }
    assert(caught);
}
```

   Funktioniert dein Programm, dann wird für jede ungültige Zahl aus `invalid_data` eine aussagekräftige Fehlermeldung ausgegeben. Das Programm darf aber **nicht** abbrechen.

4 Übungszweck dieses Beispiels

- iterativen Algorithmus gemäß Anforderungen entwerfen
- mehrere Funktionen definieren, `return`
- globale Variablen definieren
- `switch`, `case`, `break`, `default`
- `string::operator[]`
- `char`, `unsigned int`
- `throw` mit Exceptions aus `std`
- `++`
- Verwendung eines Texteditors