C++

8. Januar 2023

Inhaltsverzeichnis

1 Hallo Welt! 2

2 Elemente des Programmierens 2

2.1 Ausdrücke (Expressions) . . . 2
2.2 Namen . . . 3
2.3 Funktionen . . . 4
2.4 Bedingte Ausdrücke . . . 6
2.5 Datentypen . . . 7
2.6 Beispiel: Quadratwurzeln nach der Newton-Methode . . . 8
2.7 Coding Dojo Römische Zahlen . . . 9
2.8 Abstraktion von Daten . . . 10

1 Hallo Welt!

Hallo Welt! Programm in C++.

```
#include <iostream>

int main() {
    std::cout << "Hallo Welt!";
    return EXIT_SUCCESS;
}
```

Jedes ausführbare C++ Programm muss eine **main** Funktion besitzen, (**int main()**) die als Einstiegspunkt dient, d.h. hier beginnt das Programm. Am Ende jedes C++ Programms wird zurück geliefert, ob das Programm erfolgreich durchgelaufen ist (**return EXIT_SUCCESS;**). Die Zeile **std::cout « "Hallo Welt!";** gibt den Text **Hallo Welt!** auf der Konsole aus. Damit **std::cout** verwendet werden kann, muss das dem aktuellen Programm bekannt gemacht werden, das funktioniert über das Einbinden von **iostream** mit (**#include <iostream>**).

Compiler Explorer Beispiel: Hallo Welt!

2 Elemente des Programmierens

- primitive Ausdrücke
- Kombinationsmöglichkeiten
- Abstraktionsmöglichkeiten

2.1 Ausdrücke (Expressions)

Ein einfacher Ausdruck kann zum Beispiel eine Zahl sein oder eine Kombination von Zahlen mit Operatoren wie + oder *. Kombinierte Ausdrücke werden wieder zu einfachen Ausdrücken ausgewertet (siehe unten).

```
    486                 ->      486

    137 + 349           ->      486
    1000 - 334          ->      666
    5 * 99              ->      495

    21 + 35 + 12 + 7    ->      75
    3 * 5 + 10 - 6      ->      19
```

Mit der globalen Objekt **std::cout** und dem Streamoperator « lassen sich Ausdrücke auf der Konsole ausgeben. Zum Beispiel die Zahl **5**:

```
#include <iostream>

int main(){
    std::cout << 5;
    return EXIT_SUCCESS;
}
```

Oder ein kombinierter Ausdruck.

```
    #include <iostream>

    int main(){
        std::cout << 2 * 4 * (3 + 5) - 21 * (3 - 1);
        return EXIT_SUCCESS;
    }
```

2.2 Namen

Ein essenzieller Aspekt des Programmierens ist die Möglichkeit, Namen für Berechnungsobjekte zu vergeben. Um die Intention einer Berechnung ausdrücken zu können und Berechnungsobjekte an mehreren Stellen zu verwenden. *Beispiel für die Verwendung eines Namens:*

```
    #include <iostream>

    int main(){
        const auto size{2};
        std::cout << size;
        return EXIT_SUCCESS;
    }
```

Die Zeile **const auto size{2};** verbindet den Name **size** mit der Zahl **2**. Wir können auf den Wert **2** über den Namen **size** zugreifen und diesen für die Ausgabe nutzen (**std::cout « size;**).

Vergleicht man die beiden nachstehenden Beispiele, ist es dem Leser bei der Verwendung von Namen viel leichter möglich zu erkennen, was berechnet wird. Durch das Vergeben von Namen kann ein Wert an mehreren Stellen benutzt werden.

```
    #include <iostream>

    int main(){
        std::cout << 3.14159 * 10.0 * 10.0 << '\n';
        std::cout << 2 * 3.14159 * 10.0 << '\n';
        return EXIT_SUCCESS;
    }
```

```
#include <iostream>

int main(){
    const auto pi{3.14159};
    const auto radius{10.0};

    std::cout << pi * radius * radius << '\n';

    const auto circumference{2 * pi * radius};
    std::cout << circumference << '\n';

    return EXIT_SUCCESS;
}
```

- Compiler Explorer Beispiel: Namen

2.3 Funktionen

Bis jetzt haben wir die fundamentalen Bestandteile jeder Programmiersprache kennengelernt.

- Zahlen und arithmetische Operatoren
- Kombinationsmöglichkeiten von Operatoren
- eine limitierte Abstraktionsmöglichkeit durch das Vergeben von Namen für Werte

Im nächsten Schritt lernen wir Funktionen kennen, eine sehr leistungsfähige Abstraktionsmöglichkeit, bei der einer Kombination von Operationen ein Name gegeben werden kann, um diese als Einheit wiederzuverwenden.

Als Beispiel gucken wir uns die Idee des Quadrierens an. Um Etwas zu quadrieren, multiplizieren wir es mit sich selbst. In C++ wird das so ausgedrückt:

```
auto square(auto x)
{
    return x * x;
}
```

square ist der Name der Funktion und **x** ist der Übergabeparameter, der quadriert werden soll. Mit **return** wird das Ergebnis des Ausdrucks $x * x$ zurück an die Aufrufstelle übergeben. Benutzt werden kann diese Funktion nach der Definition mit verschiedenen Werten:

```
square(21)          ->      441
square(2 + 5)       ->      49
square(square(3))   ->      81
```

Wir können die Funktion **square** auch benutzen um weitere Funktionen zu definieren. Zum Beispiel der Ausdruck $x^2 + y^2$ kann ausgedrückt werden als:

```
1     square(x) + square(y)
```

Daraus lässt sich einfach eine neue Funktion bilden, die die Summe der Quadrate bildet.

```
1     auto sumOfSquares(auto x, auto y)
2     {
3         return square(x) + square(y);
4     }
```

In einem vollständigen C++ Programm, welches das Ergebnis der Summe der Quadrate von 3 und 4 auf der Konsole ausgibt, sieht das dann so aus:

```
1     #include <iostream>
2
3     auto square(auto x)
4     {
5         return x * x;
6     }
7
8     auto sumOfSquares(auto x, auto y)
9     {
10        return square(x) + square(y);
11    }
12
13    int main(){
14        std::cout << sumOfSquares(3, 4) << '\n';
15
16        return EXIT_SUCCESS;
17    }
```

Ablauf des Programms:

```
1     std::cout << sumOfSquares(3, 4) << '\n';
2  ->  std::cout << square(3) + square(4) << '\n';
3  ->  std::cout << 3 * 3 + 4 * 4 << '\n';
4  ->  std::cout << 9 + 16 << '\n';
5  ->  std::cout << 25 << '\n';
```

Die Funktion **sumOfSquares** wird mit den Parametern **3** und **4** aufgerufen. Innerhalb der Funktion hat jetzt der Parameter **x** den Wert **3** und **y** ist gleich **4**. Mit **x** und **y** wird jetzt jeweils die Funktion **square** aufgerufen, d.h. der Parameter **x** der Funktion **square** ist einmal **3** und einmal **4**, sodass als Ergebnis einmal **9** und einmal **16** zurück gegeben wird. Diese Ergebnisse werden addiert und an die Funktion **main** zurück gegeben.

- Compiler Explorer Beispiel: Funktionen

Aufgabe 1 Schreibe eine Funktion, die drei Zahlen als Parameter übergeben bekommt und die Summe der Quadrate als Ergebnis liefert. (Tipp: Erweitere das Beispiel für Funktionen)

2.4 Bedingte Ausdrücke

Wollen wir eine Funktion erstellen, die den absoluten Wert einer Zahl zurück gibt, brauchen wir die Möglichkeit, zwei Fälle zu unterscheiden.

$$|x| = \begin{cases} x, & \text{if } x \geq 0 \\ -x, & \text{if } x < 0 \end{cases}$$

Ist der Eingangsparameter negativ, muss das Vorzeichen umgedreht werden, bei einer positiven Zahl muss diese einfach zurück gegeben werden.

```
1 auto abs(auto x)
2 {
3     return x < 0 ? -x : x;
4 }
```

Mögliche Vergleichsoperatoren, die zusammen mit dem **?-Operator** benutzt werden können, sind in der folgenden Tabelle aufgelistet.

Name	Operator
gleich	==
ungleich	!=
größer als	>
kleiner als	<
größer gleich	>=
kleiner gleich	<=

Vor dem Fragezeichen muss ein Ausdruck stehen, der zu einem Wahrheitswert (Wahr oder Falsch) ausgewertet wird. Ist der Ausdruck **Wahr**, wird der Gesamte **?-Operator** zu dem Code links vom Doppelpunkt ausgewertet, ist der Ausdruck **Falsch**, der Rechte.

Aufgabe 2 Schreibe eine Funktion, die drei Zahlen als Parameter übergeben bekommt und von den beiden größten Zahlen die Summe der Quadrate als Ergebnis liefert.

2.5 Datentypen

In C++ haben alle Variablen einen festen Typ. Beispiele für Typen sind: Ganzzahl **x** mit dem Wert **-3**

```
1    const int x{-3};
```

Positive Ganzzahl **y** mit dem Wert **2**

```
1    const unsigned int y{2};
```

Gleitkommazahl **z** mit dem Wert **3,7**

```
1    const double y{3.7};
```

Wahrheitswert **b** mit dem Wert **true (Wahr)**

```
1    const bool b{true};
```

Buchstabe **c** mit dem Wert **H**

```
1    const char c{'H'};
```

Datentype für Text aus der "C++ Standard Library". Beispiel Text **t** mit dem Wert **Hallo**

```
1    const std::string t{"Hallo"};
```

Mit dem Schlüsselwort **auto** wird der Typ automatisch vom Wert abgeleitet, die Variable **size** aus dem Beispiel **const auto size{2};** ist beispielsweise vom Typ **int**. Würde man im Vergleich dazu **const auto size{2.};** schreiben, wäre die Variable **size** vom Typ **double**, da es sich jetzt um eine Gleitkommazahl handelt.

Bei Funktionen ist es meist sinnvoll, den Typ genau vorzugeben, um explizit anzugeben für welche Typen die Funktion vorgesehen ist (Hier ist das Schlüsselwort **auto** bei älteren Versionen von C++ noch gar nicht erlaubt.) Im nächsten Beispiel ist das Schlüsselwort **auto** für die Rückgabewerte und die Parameter der Funktionen durch den speziellen Typ ersetzt worden (double und bool).

2.6 Beispiel: Quadratwurzeln nach der Newton-Methode

Aufgabe 3 Erkläre den Ablauf des nachfolgenden Programms.

```
1  #include <iostream>
2
3  double square(double x)
4  {
5      return x * x;
6  }
7
8  double sumOfSquares(double x, double y)
9  {
10     return square(x) + square(y);
11 }
12
13 double abs(double x)
14 {
15     return x < 0 ? -x : x;
16 }
17
18 double average(double x, double y)
19 {
20     return (x + y) / 2.;
21 }
22
23 double improve(double guess, double x)
24 {
25     return average(guess, x/guess);
26 }
27
28 bool isGoodEnough(double guess, double x)
29 {
30     return abs(square(guess) - x) < 0.001;
31 }
32
33 double squareRootIter(double guess, double x)
34 {
35     if(isGoodEnough(guess, x))
36     {
37         return guess;
38     }
39     return squareRootIter(improve(guess,x),x);
40 }
41
42 double squareRoot(double x)
43 {
44     return squareRootIter(1.0, x);
45 }
46
```

```
47
48 int main(){
49     std::cout << squareRoot(9.0) << '\n';
50     std::cout << squareRoot(100. + 37.) << '\n';
51
52     std::cout << square(squareRoot(1000.)) << '\n';
53
54     return EXIT_SUCCESS;
55 }
```

Compiler Explorer Beispiel: Quadratwurzeln nach der Newton-Methode

Aufgabe 4 (***) Die Funktion **isGoodEnough** ist nicht sehr effektiv beim finden der Wurzeln von sehr kleinen Zahlen. Dadurch, das Gleitkommaberechnungen im Computer immer nur mit einer gewissen Genauigkeit durchgeführt werden können, ist unser Test (isGoodEnough) auch für große Zahlen unzureichend. Beschreibe mit Beispielen, warum die Testfunktion für kleine und große Zahlen nicht funktioniert. Compiler Explorer: Aufgabe 2

2.7 Coding Dojo Römische Zahlen

Die Aufgabe ist es, eine Funktion zu schreiben, die zu einer gegebenen Zahl (arabisch) die römische Zahl liefert. Wenn zum Beispiel eine "7"übergeben wird, ist das Ergebnis "VII". Die Signatur der Funktion könnte zum Beispiel so aussehen:

```
1     std::string toRoman(int x)
2     {
3         return "";   // TODO
4     }
```

Um sicherzustellen das die Funktion wie erwartet funktioniert, müssen passende Unittests erstellt werden. Gute Praxis ist es testgetrieben zu entwickeln (TDD: Test-driven development). Hierbei wird wie folgt vorgegangen:

1. schreibe einen **einzigen** Unittest
2. führe den Test aus (Red)
3. schreib gerade genug Code damit der Test erfolgreich ist (Green)
4. überarbeite/vereinfache den Code (Refactor)
5. starte wieder bei Punkt 1

Compiler Explorer: Coding Dojo Römische Zahlen

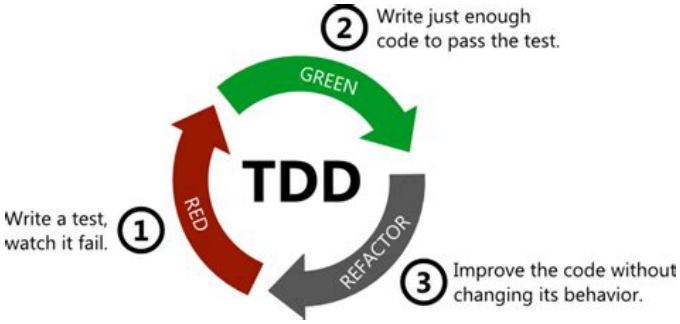

Abbildung 1: TDD

2.8 Abstraktion von Daten

Mit Funktionen haben wir gelernt, eine Abstraktion für eine Kombination von Operationen einzuführen. In C++ gibt es auch die Möglichkeit neue Typen zu definieren und somit eine neue Abstraktion aus der Kombination von Typen zu bilden. Wollen wir zum Beispiel ein Geometrie Programm implementieren, das den Abstand zwischen zwei Punkten **A** und **B** berechnen kann, könnten wir einen Punkt durch zwei **double**-Variablen abbilden (x und y Koordinate).

```
1  #include <iostream>
2
3  double square(double x)
4  {
5      return x * x;
6  }
7
8  double distance(double x1, double y1, double x2, double y2)
9  {
10     const double dx{x2-x1};
11     const double dy{y2-y1};
12     return std::sqrt(square(dx) +square(dy));
13 }
14
15
16 int main(){
17     //missing abstraction for points
18     const double xPointA{2.5};
19     const double yPointA{3.0};
20
21     const double xPointB{5.5};
22     const double yPointB{7.0};
23
24
25     std::cout << distance(xPointA, yPointA, xPointB, yPointB) << '\n';
26
```

```
27      return EXIT_SUCCESS;
28  }
```

Compiler Explorer: Beispiel mit fehlender Abstraktion für Punkte

Mit dem Schlüsselwort **struct** lassen sich die beiden Koordinaten eines Punktes zusammenfassen und mit einem Namen versehen, um eine Abstraktion zu erzeugen. Neue Objekte vom Typ Punkt lassen sich wie bei anderen Typen mit Hilfe von geschweiften Klammern anlegen. Da Punkte aus zwei Koordinaten bestehen, müssen auch beide Werte Komma getrennt übergeben werden. Zum Beispiel erzeugt **const Point p{3.5, 2.3};** einen Punkt mit dem Namen **p** und den Koordinaten **x=3.5** und **y=2.3**. Eigene Typen können genauso wie eingebaute Typen (z.B. **int**, **double**,...) verwendet werden (z.B. als Funktionsparameter, Konstanten, Variablen, Bestandteile von neuen Typen, ...)

```
1   #include <iostream>
2
3   double square(double x)
4   {
5       return x * x;
6   }
7
8   //new abstraction for points
9   struct Point{
10      double x;
11      double y;
12  };
13
14  double distance(Point a, Point b)
15  {
16      const double dx{b.x-a.x};
17      const double dy{b.y-a.y};
18      return std::sqrt(square(dx) +square(dy));
19  }
20
21
22  int main(){
23      const Point a{2.5, 3.0};
24      const Point b{5.5, 7.0};
25      const Point c{5.5, 3.0};
26
27      std::cout << distance(a, b) << '\n';
28      std::cout << distance(a, c) << '\n';
29      std::cout << distance(b, c) << '\n';
30
31      return EXIT_SUCCESS;
32  }
```

Es lassen sich auch Standartwerte für **structs** vorgeben, die verwendet werden wenn zum Beispiel ein Punkt ohne explizite Koordinaten erzeugt wird.

```
1
2       struct Point{
3           double x{0.0};
4           double y{0.0};
5       };
6
7       int main(){
8           const Point a{};          // -> a.x = 0.0 , a.y = 0.0
9           return EXIT_SUCCESS;
10      }
```

Compiler Explorer: Beispiel mit Abstraktion für Punkte