Materialpaket 07_STD – Standard Library

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):

Sie können einfache Templates implementieren und Teile der C++-Standardbibliothek verwenden.

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

- anhand eines einfachen Containers nachvollziehen, wie `template`-Klassen eingesetzt werden

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

- die wichtigsten Container der C++-Standardbibliothek verwenden
- einige wenige Algorithmen der C++-Standardbibliothek verwenden

2 Konzepte

Im Folgenden wird das Konzept des *parametric polymorphism* dargestellt, welches es dem Programmierer gestattet, Code zu schreiben, der mit unterschiedlichen (dennoch gleichartigen) Datentypen rechnet.

Des Weiteren werden wichtige Komponenten der C++-Standardbibliothek vorgestellt.

2.1 Implementierung eines Stacks für int-Werte

Die vielfach verwendete Datenstruktur Stack soll anhand einer einfachen Implementierung für `int`-Werte illustriert werden. Zunächst die Typdefinition:

```
1  //  intStack.hpp
2  //  6_STD_stacks
3
4  #ifndef intStack_hpp
5  #define intStack_hpp
6
7  // good: encapsulation
8  // good: separation of interface and implementation
9  // bad: fixed element type
10 // bad: fixed capacity
11
12 struct intStack {
13   static const int capacity=1024;
14   int tos; // top of stack, i.e. index of next write/push
15   int elements[capacity];
16 public:
17   intStack();
18   void push(int element);
19   int pop();
20   int size(); // number of elements pushed
21   bool isEmpty();
22   void clear();
23   void print();
24 };
25
26 #endif
```

Diese Klasse stellt einen Werttypen dar, in dem bis zu 1024 `int`-Werte gespeichert (`push()`) und wieder abgefragt (`pop()`) werden können. Ein Zugriff auf beliebige Stellen ist nicht vorgesehen. Die Membervariable `tos` speichert den Index der beim nächsten `push()` zu beschreibenden Zelle im Array `elements` (beim Beschrieben wird `tos` inkrementiert). In den Kommentaren in obigem Quelltext sind wesentliche Nachteile dieser einfachen Implementierung zu sehen:

- der Typ der Elemente ist auf `int` festgelegt
- die Kapazität ist auf `1024` festgelegt

Die folgende Funktion `use_intStack()` zeigt, wie ein Stapel verwendet werden kann:

```
1  void use_intStack(){
2    intStack s_1;
3    for(int i=0; i<10;i++){
4      s_1.push(i*i);
5    }
6    std::cout << "s_1 : " << std::endl;
7    s_1.print();
8    // replace top element by 4711:
9    s_1.pop();
10   s_1.push(4711);
11   // swap topmost two elements:
12   int tmp1 = s_1.pop();
13   int tmp2 = s_1.pop();
14   s_1.push(tmp1);
15   s_1.push(tmp2);
16   std::cout << "draining s_1 : " << std::endl;
17   while(!s_1.isEmpty()){
18     std::cout << s_1.pop() << std::endl;
19   }
20 }
```

In obigem Quelltext werden die Quadratzahlen von `0..89` auf einem Stack gespeichert. Das oberste Element wird durch `4711` getauscht, wozu eine `pop()/push()`-Kombination notwendig ist. Das anschließende Tauschen der obersten beiden Elemente benötigt sogar zwei lokale Variablen (`tmp1` und `tmp2`).

Die Ausgabe von `use_intStack()` sieht wie folgt aus:

```
use_intStack()
--------------
s_1 :
10 of 1024 allocated.
0: 0
1: 1
2: 4
3: 9
```

```
4: 16
5: 25
6: 36
7: 49
8: 64
9: 81
draining s_1 :
64
4711
49
36
25
16
9
4
1
0
```

2.2 Implementierung eines generischen Stacks

Der folgende Quellcode verdeutlicht, wie Templates helfen können, die Probleme des `intStack` zu vermeiden: Es lassen sich `genericStack`-Objekte anlegen, welche verschiede Elementtypen erlauben und unterschiedliche feste Größen haben (`genericStack<int, 20>` und `genericStack<double, 30>`).

```
1  void use_genericStack(){
2    genericStack<int, 20> s_2;
3    genericStack<double, 30> s_3;
4    for(int i=0; i<10;i++){
5      s_2.push(i*i);
6      s_3.push(i*i * 1.1);
7    }
8    s_2.print();
9    s_3.print();
10 }
```

Die Ausgabe von `use_genericStack()` sieht wie folgt aus:

```
use_genericStack()
------------------
10 of 20 allocated.
0: 0
1: 1
2: 4
3: 9
4: 16
```

```
5: 25
6: 36
7: 49
8: 64
9: 81
10 of 30 allocated.
0: 0
1: 1.1
2: 4.4
3: 9.9
4: 17.6
5: 27.5
6: 39.6
7: 53.9
8: 70.4
9: 89.1
```

Die Definition der Template-Klasse `genericStack`:

```
1  //  genericStack.hpp
2
3  #ifndef genericStack_hpp
4  #define genericStack_hpp
5  #include <iostream>
6
7  // good: encapsulation
8  // good: separation of interface and implementation
9  // good: arbitrary element type
10 // good: arbitrary capacity
11 // bad: implementation in .hpp
12 // bad: compiler error messages are difficult to read
13
14 template <typename ElementType, int capacity>
15 struct genericStack {
16   int tos; // top of stack, i.e. index of next write/push
17   ElementType elements[capacity];
18 public:
19   genericStack();
20   void push(ElementType element);
21   ElementType pop();
22   int size(); // number of elements pushed
23   bool isEmpty();
24   void clear();
25   void print();
26 };
```

Die Methoden sind analog zu denen des `intStacks` implementiert. Die

wesentlichen Unterschiede sind jedoch:

- die Methoden müssen in der Header-Datei vollständig implementiert sein, da der Compiler diese Definitionen an den benutzenden Stellen benötigt
- bei jeder Methodendefinition müssen die Template-Parameter angeführt sein (`template <typename ElementType, int capacity> ... <ElementType, capacity>`)

```
1  //  genericStack.hpp
2  // ============= implementation =============
3
4  template <typename ElementType, int capacity>
5  genericStack<ElementType, capacity>::genericStack()
6  : tos(0)
7  {
8
9  }
10
11 template <typename ElementType, int capacity>
12 void genericStack<ElementType, capacity>::push(ElementType element){
13   elements[tos++] = element;
14 }
15
16 template <typename ElementType, int capacity>
17 ElementType genericStack<ElementType, capacity>::pop(){
18   return elements[--tos];
19 }
20
21 template <typename ElementType, int capacity>
22 int genericStack<ElementType, capacity>::size(){ // number of elements pushed
23   return tos;
24 }
25
26 template <typename ElementType, int capacity>
27 bool genericStack<ElementType, capacity>::isEmpty(){
28   return tos == 0;
29 }
30
31 template <typename ElementType, int capacity>
32 void genericStack<ElementType, capacity>::clear(){
33   tos = 0;
34 }
35
36 template <typename ElementType, int capacity>
37 void genericStack<ElementType, capacity>::print(){
38   std::cout << size() << " of " << capacity << " allocated." << std::endl;
39   for (int i=0; i<tos; i++){
```

```
40      std::cout << i << ": " << elements[i] << std::endl;
41    }
42  }
43  #endif
```

2.3 Wichtige Container der C++-Standardbibliothek

- `std::vector`
- `std::list`
- `std::deque`
- `std::map`

Die Standardcontainer unterscheiden sich hinsichtlich der Zugriffsmöglichkeiten und der zu erwartenden Laufzeit einzelner Operationen (so kann erwartet werden, dass das Einfügen in Listen effizienter vonstatten geht, als das Einfügen in Vektoren).

Der folgende Code zeigt die Verwendung der beiden Standardcontainer `std::vector` und `std::list`:

```
1   void use_std_containers(){
2     std::vector<int> v;
3     v.push_back(17);          // append an element at the end
4     int i = v.back();         // get last element
5     int j = v[0];             // access by subscript operator
6     if ((i==j) && (i==17)){   //
7       v.pop_back();           // remove last element
8       v.clear();              // remove all elements
9     }
10    long s = v.size();        // get size; size() <= max_size()
11    std::list<long> l;
12    l.push_back(17);          // append an element at the end
13    l.push_front(s);          // put an element in front of the list
14  }
```

Es wird ersichtlich, dass es nicht notwendig ist, eine eigene Stack-Klasse (wie `genericStack`) zu implementieren, da die Standardcontainer die notwendigen Operationen mitbringen.

Siehe hierzu auch [PPP] Seite 1146.

2.4 Wichtige Algorithmen der C++-Standardbibliothek

Eine vollständige Auflistung der Algorithmen der C++-Standardbibliothek findet sich hier[2]. Hier sollen nur einige wenige Algortihmen vorgestellt werden:

[2] http://en.cppreference.com/w/cpp/algorithm

- `std::sort`
- `std::transform`

Der folgende Quellcode zeigt die Benutzung dieser beiden Standardalgorithmen:

```
1  void use_std_algorithms(){
2    std::vector<int> v(100);
3    for(int i=0; i<10; i++){
4      v[i*i] = i;
5    }
6    v.push_back(17);
7    printVector(v);
8    std::sort(v.begin(), v.end());
9    printVector(v);
10   std::transform(v.begin(), v.end(), v.begin(),
11                  [](int i) { return i*10; }); // use lambda expression here
12   printVector(v);
13 }
```

An der Ausgabe ist zu erkennen, wie der Inhalt des Vektors sortiert und schließlich mit `10` multipliziert wird:

```
// some output omitted
[0]=0, [1]=0, [2]=0, [3]=0, [4]=0, [5]=0, [6]=0, [7]=0, [8]=0,
[9]=0, [10]=0, [11]=0, [12]=0, [13]=0, [14]=0, [15]=0, [16]=0,
// ...
[73]=0, [74]=0, [75]=0, [76]=0, [77]=0, [78]=0, [79]=0, [80]=0,
[81]=0, [82]=0, [83]=0, [84]=0, [85]=0, [86]=0, [87]=0, [88]=0,
[89]=0, [90]=0, [91]=10, [92]=20, [93]=30, [94]=40, [95]=50,
[96]=60, [97]=70, [98]=80, [99]=90, [100]=170,
```

3 Material zum aktiven Lernen

3.1 Aufgabe: Grundgerüst

Achten Sie darauf, bei den Projekteinstellung den C++14-Standard zu aktivieren (unter Workspace - active project settings - global settings - C++ compiler options - ...).

Nehmen Sie den unten gegebenen Quellcode als Basis und fügen den `BinaryOctet`-Konstruktor hinzu, der ein `int` als Parameter akzeptiert.

Generieren Sie `42` Zufallswerte aus dem Bereich `0..23`. Speichern Sie die Zufallszahlen in einem `int`-Array.

Schreiben sie eine Funktion `int stringSimilarity()`, die für zwei `std::string` Vergleichswerte ausgibt (Größe, <, <=, etc.) und auf einer Skala von 0..100

zurückgibt, wie ähnlich die beiden Strings sind. Vergeben Sie Ähnlichkeitspunkte für gleiche Länge, gemeinsame Zeichen, viele Zeichen gleicher Groß-/Kleinschreibung, etc. Der Wert 100 ist für vollständig gleiche Zeichenketten zurückzugeben. Zeigen Sie die Funktionsfähigkeit von `stringSimilarity()` anhand einiger Testaufrufe in `void unittest_stringSimilarity()`.

```
// based on example from http://en.cppreference.com/w/cpp/numeric/random/rand
#include <cstdlib>
#include <iostream>
#include <ctime>

struct BinaryOctet {
  bool evenParity;
  char bitsAsDigits[bitsPerOctet];
};

int main()
{
    std::srand(std::time(0)); // use current time as seed for random generator
    int random_variable = std::rand();
    std::cout << "Random value on [0 " << RAND_MAX << "]: "
              << random_variable << '\n';
}
```

Wenn Sie den Aufruf von `std::srand()` weglassen, erleichtert sich die Fehlersuche, da Sie dann in jedem Lauf die selbe Sequenz von Zufallszahlen erhalten.

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. Zufallszahlen in einen `vector<>` ohne Duplikate einfügen
2. Funktion schreiben, welche den gefüllten `vector<>` CSV-artig ausgibt (`"7", "9", "1", ...`)
3. Zufallszahlen in `list<>` ohne Duplikate einfügen (`std::find` darf verwendet werden)
4. Zufallszahlen in `vector<>` nacheinander an die richtige Stelle (sortiert) einfügen
5. Zufallszahlen in `list<>` nacheinander an die richtige Stelle (sortiert) einfügen

6. mehrere binaryOctets in einen `vector<>` speichern und dort von `std::sort` sortieren lassen

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. Worin liegen die wesentlichen Unterschiede zwischen `std::vector` und `std::list`?
2. Wie gelingt es `std::sort()` einen Container zu sortieren, in dem sich Objekte befinden, welche Instanzen von Typen sind, die `sort()` nicht bekannt sind?
3. Welchen Vorteil hat die Nutzung von Container-Templates aus der Standardbibliothek gegenüber einer Eigenimplementierung?
4. Sie benötigen für ein Programmierproblem eine Datenspeicher, welche sich wie ein Stapel verhält. Wie lösen Sie dieses Problem?
5. Ein `char` kann implizit in ein `int` umgewandelt werden. Kann eine `std::list<char>` in eine `std::list<int>` implizit umgewandelt werden? Begründung!

4 Nützliche Links

- cxx-prettyprint: A C++ Container Pretty-Printer[3]
- Wikibooks: C++-Programmierung, Die STL Container[4]
- Wikibooks: C++-Programmierung, Die STL Algorithmen[5]
- The C++ Standard Library - A Tutorial and Reference[6]
- Carlos Moreno: An Introduction to the Standard Template Library (STL)[7]
- Chua Hock-Chuan: C++ Standard Libraries and Standard Template Library (STL)[8]
- Learn C++: Overloading the parenthesis operator[9]

5 Literatur

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

[3] http://louisdx.github.io/cxx-prettyprint/
[4] https://de.wikibooks.org/wiki/C%2B%2B-Programmierung/_Die_STL/_Container
[5] https://de.wikibooks.org/wiki/C%2B%2B-Programmierung/_Die_STL/_Algorithmen
[6] http://www.cppstdlib.com
[7] https://cal-linux.com/tutorials/STL.html
[8] https://www3.ntu.edu.sg/home/ehchua/programming/cpp/cp9_STL.html
[9] http://www.learncpp.com/cpp-tutorial/99-overloading-the-parenthesis-operator/

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