SWE Zusammenfassung

Elias Leonhardsberger

7. Juli 2025, Hagenberg

Inhaltsverzeichnis

1 C++

1.1 Standard Library

Der Namensraum *std* ist für die Standardbibliothek von C++ und C reserviert. Alle C Header-Dateien sind in C++ verfügbar, ihr Name ist immer mit einem *c* vorangestellt, z.B. *cstdio* für *stdio.h* in C.

1.1.1 Strings

Im gegensatz zu C, wo Strings als char-Arrays implementiert sind, gibt es in C++ die Klasse *std::string*, die es ermöglicht mit Templates Strings aus *char*, *wchar_t* oder einem beliebigen eigenen Typ zu erstellen. Die ersten beiden Varianten sind bereits vordefiniert.

```
1 typedef basic_string< char >   string;
2 typedef basic_string<wchar_t> wstring;
```

1.1.2 Ein-/Ausgabe

In C und C++ gibt es keine in die Sprache eingebauten Ein-/Ausgabefunktionen. In C werden die Funktionen aus *stdio.h* verwendet, in C++ die Klassen aus *iostream*.

Der Unterscheid ist, dass C++ mit Streams arbeitet, diese sind

- typsicher
- implementierbar für eigene Klassen
- effizienter da man nicht auf interpretierte Formatzeichenketten angewiesen ist(z.B. *%d* für int)
- auf Zeichenebene Thread-sicher

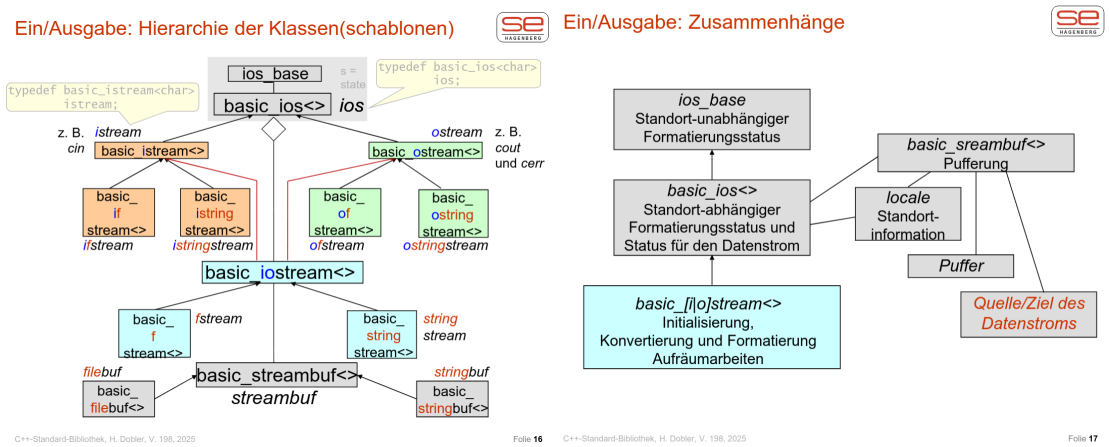

Abbildung 1: Klassenhierarchie der Ein-/Ausgabeklassen in C++

Diese Streams sind in mehreren Header-Dateien aufgeteilt:

- **iosfwd** für Vorwärtsdeklarationen(Sehr klein)

- **streambuf** für die Pufferung von Ein-/Ausgabe
- **istream** für die Eingabe
- **ostream** für die Ausgabe
- **iostream** für die Standard-Ein-/Ausgabe
- **fstream** für Dateiein-/ausgabe
- **sstream** für String-Ein-/Ausgabe
- **strstream** für für Char*-Ein-/Ausgabe
- **iomanip** für Ein-/Ausgabeformatierung

Flush kann auch mit Stream Aufrufen verkettet werden, z.B. *cout « std::flush;*. *std::endl* flusht den Stream auch nachdem er einen Zeilenumbruch ausgegeben hat. Achtung kann Performanceprobleme verursachen, da es den Puffer jedes Mal leert! Wenn mehrere Zeilenumbrüche ausgegeben werden eher den folgenden Ausschnitt verwenden.

```
1 stream << "\n"
```

Es gibt auch diverse Manipulatoren, die das Verhalten von Streams beeinflussen.

- *std::dec* für Dezimalzahlen
- *std::hex* für Hexadezimalzahlen
- *std::oct* für Oktalzahlen
- *std::setbase(n)* für die Basis der Zahlendarstellung
- *std::setfill(c)* für das Füllzeichen
- *std::setprecision(n)* für die Anzahl der Nachkommastellen
- *std::setw(n)* für die Breite des Feldes
- ...

4 Syntaxvergleich

Um den Syntax von C++ und Java zu vergleichen, hab ich ein kleines Beispielprogramm geschrieben mit:

- einem Interface
- einer Basisklasse, die das Interface implementiert
- einer abgeleiteten Klasse, die von der Basisklasse erbt
- einer Iteratorimplementierung in der abgeleiteten Klasse
- einer Main-Methode, die ein Objekt der abgeleiteten Klasse verwendet

4.1 C++

4.1.1 Interface

```cpp
template <typename T> // Generic programming with templates
class interface // in C++ there is no special keyword for interfaces, so we use a class
{
public:
    virtual ~interface() = default; // Virtual destructor for proper cleanup of derived classes

    virtual T getValue(int index) const = 0; // Pure virtual function, must be implemented by derived classes
    virtual T operator[](int index) const = 0; // operator overload for indexing, must be implemented by derived classes
    virtual void addValue(T value) = 0; // Pure virtual function, must be implemented by derived classes
    virtual void operator+=(T value) = 0; // operator overload for adding a value, must be implemented by derived classes
    virtual int getSize() const = 0; // Pure virtual function, must be implemented by derived classes
};
```

4.1.2 Basisklasse

```cpp
#include "interface.h"

template <typename T>
class baseclass : public interface<T>
{
private:
    int _size;

public:
```

```
    baseclass(int size = 10) : _size(size) {}// Constructor with default
    ↪ parameter

    virtual int getSize() const override
    {
        return _size;
    }
};
```

4.1.3 Abgeleitete Klasse

```
#include "baseclass.h"
#include <iostream>

template <typename T>
class derivedclass : public baseclass<T>
{
    friend std::ostream& operator<<(std::ostream& os, const
    ↪ derivedclass<T>& obj)
    {
        for (int i = 0; i < obj._currentIndex; ++i)
        {
            os << obj._data[i] << " "; // Access _data directly since
            ↪ it's a friend
        }

        return os;
    }

private:
    T* _data;
    int _currentIndex;

public:
    // Constructor calls base constructor with size parameter and
    ↪ initializes _currentIndex
    derivedclass(int size = 10) : baseclass<T>(size), _currentIndex(0)
    {
        // Allocate memory for _data based on the size provided
        _data = new T[size];
    }

    // Destructor to clean up allocated memory
    virtual ~derivedclass()
    {
        delete[] _data; // Free the allocated memory for _data
    }

```

```
35      // copy constructor to create a deep copy of the derivedclass
36      derivedclass(const derivedclass& other) :
        ↪  baseclass<T>(other.getSize()),
        ↪  _currentIndex(other._currentIndex)
37      {
38          // Allocate memory for _data and copy the values from the other
            ↪  instance
39          _data = new T[other.getSize()];
40          for (int i = 0; i < other.getSize(); ++i)
41          {
42              _data[i] = other._data[i];
43          }
44      }
45
46      // move constructor to transfer ownership of resources
47      derivedclass(derivedclass&& other) noexcept :
        ↪  baseclass<T>(other.getSize())
48      {
49          // Transfer ownership of the data pointer and current index
            ↪  from the other instance
50          _data = other._data;
51          _currentIndex = other._currentIndex;
52
53          // Set the other instance's data pointer to nullptr to avoid
            ↪  double deletion
54          other._data = nullptr;
55          other._currentIndex = 0; // Reset the index of the moved-from
            ↪  instance
56      }
57
58      // copy assignment operator to create a deep copy of the
        ↪  derivedclass
59      derivedclass& operator=(const derivedclass& other)
60      {
61          if (this != &other) // Check for self-assignment
62          {
63              // Clean up existing resources
64              delete[] _data;
65
66              // Allocate new memory and copy the values from the other
                ↪  instance
67              _data = new T[other.getSize()];
68              for (int i = 0; i < other.getSize(); ++i)
69              {
70                  _data[i] = other._data[i];
71              }
72
73              _currentIndex = other._currentIndex;
```

```
74          }
75
76          return *this;
77      }
78
79      // move assignment operator to transfer ownership of resources
80      derivedclass& operator=(derivedclass&& other) noexcept
81      {
82          if (this != &other) // Check for self-assignment
83          {
84              // Clean up existing resources
85              delete[] _data;
86
87              // Transfer ownership of the data pointer and current index
                ↪  from the other instance
88              _data = other._data;
89              _currentIndex = other._currentIndex;
90
91              // Set the other instance's data pointer to nullptr to
                ↪  avoid double deletion
92              other._data = nullptr;
93              other._currentIndex = 0; // Reset the index of the
                ↪  moved-from instance
94          }
95
96          return *this;
97      }
98
99      // Override getValue to return the value at the current index
100     virtual T getValue(int index) const override
101     {
102         if (index < 0 || index >= this->getSize())
103         {
104             throw std::out_of_range("Index out of range");
105         }
106
107         return _data[index];
108     }
109
110     // Override operator[] to provide access to the value at the
        ↪  current index
111     virtual T operator[](int index) const override
112     {
113         return getValue(index); // Use getValue to access the value at
            ↪  the index
114     }
115
```

```
116     // Override addValue to add a value at the current index and
        ↪ increment the index
117     virtual void addValue(T value) override
118     {
119         if (_currentIndex < 0 || _currentIndex >= this->getSize())
120         {
121             throw std::out_of_range("Index out of range");
122         }
123 
124         _data[_currentIndex] = value; // Store the value at the current
            ↪ index
125         _currentIndex++; // Increment the index for the next value
126     }
127 
128     // Override operator+= to add a value at the current index and
        ↪ increment the index
129     virtual void operator+=(T value) override
130     {
131         addValue(value); // Use addValue to handle the addition and
            ↪ index increment
132     }
133 
134     class iterator
135     {
136     private:
137         T* _ptr; // Pointer to the data
138         int _index; // Current index in the data array
139 
140     public:
141         // Constructor to initialize the iterator with a pointer
142         iterator(T* ptr, int index) : _ptr(ptr), _index(index) {}
143 
144         // Overload the dereference operator to return the value at the
            ↪ current index
145         T& operator*() const
146         {
147             return _ptr[_index];
148         }
149 
150         // Overload the increment operator to move to the next index
151         iterator& operator++()
152         {
153             _index++;
154             return *this;
155         }
156 
157         // Overload the equality operator to compare two iterators
158         bool operator==(const iterator& other) const
```

```
        {
            return _index == other._index && _ptr == other._ptr;
        }

        // Overload the inequality operator to compare two iterators
        bool operator!=(const iterator& other) const
        {
            return !(*this == other);
        }

    };

    iterator begin()
    {
        return iterator(_data, 0); // Return an iterator pointing to
        ↪  the start of the data
    }

    iterator end()
    {
        return iterator(_data, _currentIndex); // Return an iterator
        ↪  pointing to the end of the data
    }
};
```

4.1.4 Main

```
#include "derivedclass.h"
#include <iostream>

int main()
{
    try
    {
        // Create an instance of derivedclass with a size of 5
        derivedclass<int> myDerived(5);

        // Add some values to the derived class
        myDerived.addValue(10);
        myDerived.addValue(20);
        myDerived += 30;

        // Access values using operator[]
        std::cout << "Value at index 0: " << myDerived[0] << std::endl;
        std::cout << "Value at index 1: " << myDerived[1] << std::endl;
        std::cout << "Value at index 2: " << myDerived.getValue(2) <<
        ↪  std::endl;

```

```
        // Demonstrate copy constructor
        derivedclass<int> copiedDerived(myDerived);
        std::cout << "Copied: " << copiedDerived << std::endl;

        // Demonstrate move constructor
        derivedclass<int> movedDerived(std::move(myDerived));
        std::cout << "Moved: " << movedDerived << std::endl;

        // Iterate through the values using the iterator
        std::cout << "Iterating through values: ";
        for (auto it = movedDerived.begin(); it != movedDerived.end();
        ↪  ++it)
        {
            std::cout << *it << " ";
        }
        std::cout << std::endl;
    }
    catch (const std::exception& e)
    {
        std::cerr << "Exception: " << e.what() << std::endl;
    }

    return 0;
}
```

4.2 Java

4.2.1 Interface

```java
package javademo;

// Java supports interfaces and generics but not operator overloading
public interface Interface<T> {
    T getValue(int index);

    void addValue(T value);

    int getSize();
    // No operator[] or operator+= in Java
}
```

4.2.2 Basisklasse

```java
package javademo;

public abstract class BaseClass<T> implements Interface<T> {
    private int size;

    BaseClass(int size) {
        this.size = size;
    }

    @Override
    public int getSize() {
        return size;
    }

    // Java supports not implementing interfaces in abstract classes
}
```

4.2.3 Abgeleitete Klasse

```java
package javademo;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.NoSuchElementException;

public class DerivedClass<T> extends BaseClass<T> implements
 ↪  Iterable<T> {
    private ArrayList<T> data;
    private int currentIndex;

```

```
    public DerivedClass(int size) {
        super(size);
        data = new ArrayList<>(size); // generic arrays in Java are
        ↪ semi-supported
        currentIndex = 0;
    }

    @Override
    public T getValue(int index) {
        if (index < 0 || index >= getSize()) {
            throw new IndexOutOfBoundsException();
        }

        return data.get(index);
    }

    @Override
    public void addValue(T value) {
        if (currentIndex < 0 || currentIndex >= getSize()) {
            throw new IndexOutOfBoundsException();
        }

        data.add(value);
        currentIndex++;
    }

    @Override
    public String toString() {
        StringBuilder sb = new StringBuilder();

        for (int i = 0; i < currentIndex; i++) {
            sb.append(data.get(i));
            sb.append(" ");
        }

        return sb.toString().trim();
    }

    // Iterable implementation for for-each loop
    @Override
    public Iterator<T> iterator() {
        return new Iterator<T>() {
            private int index = 0;

            @Override
            public boolean hasNext() {
                return index < currentIndex;
            }
```

```
58
59              @Override
60              public T next() {
61                  if (!hasNext()) {
62                      throw new NoSuchElementException();
63                  }
64
65                  return data.get(index++);
66              }
67          };
68      }
69
70      // No operator overloading in Java, so no operator[] or operator+=
71      // No need for destructors or manual memory management
72  }
```

4.2.4 Main

```
1   package javademo;
2
3   public class Demo {
4       public static void main(String[] args) {
5           DerivedClass<Integer> myDerived = new DerivedClass<>(5);
6           myDerived.addValue(10);
7           myDerived.addValue(20);
8           myDerived.addValue(30);
9
10          System.out.println("Value at index 0: " +
            ↪  myDerived.getValue(0));
11          System.out.println("Value at index 1: " +
            ↪  myDerived.getValue(1));
12          System.out.println("Value at index 2: " +
            ↪  myDerived.getValue(2));
13
14          System.out.println("toString: " + myDerived.toString());
15
16          System.out.print("Iterating through values: ");
17          for (int value : myDerived) {
18              System.out.print(value + " ");
19          }
20          System.out.println();
21      }
22  }
```