

Systemprogrammierung

Teil 6: C++ Standardbibliothek

Templates, Ein-/Ausgabe, Strings, Container,
Algorithmen, Iteratoren, intelligente Zeiger

C++ Standardbibliothek: Überblick

Die C++ Standardbibliothek enthält die C Standardbibliothek und zusätzlich vor allem templatebasierte Erweiterungen, die auf der maßgeblich von Alexander Stepanow entwickelten STL (*Standard Template Library*) beruhen:

- erweiterbare objektorientierte Ein-/Ausgabe mit Streams

`std::istream, std::ostream, ...`

- Zeichenketten

`std::string`

- Container und Iteratoren

`std::vector, std::array, std::list, ...`

- Algorithmen

`std::max, std::find, ...`

- intelligente Zeiger (*smart pointers*)

`std::unique_ptr, std::shared_ptr, std::weak_ptr`

- ...

C++ Templates: Syntax

C++ kennt verschieden Arten von Templates:

- **Klassentemplates** definieren Familien von Klassen 

```
template< Parameterliste > class Klassenname ...
```

- **Funktionstemplates** definieren Familien von Funktionen

```
template< Parameterliste > Typ Funktionsname( ... ) ...
```

- ...

Template-Parameterlisten können verschiedene Arten von Parametern enthalten:

- **Typ-Parameter:** `template<typename Name > ...`

bei der Instanziierung muss ein Typ als Argument angegeben werden

- **Nichttyp-Parameter:** `template< Typ Name > ...`

*bei der Instanziierung muss ein konstanter Ausdruck als Argument angegeben werden,
z.B. ein Literal*


als Typen sind ganzzahlige, Zeiger-, Referenz- und Aufzählungstypen erlaubt

- ...

C++ Templates: Vergleich mit Java

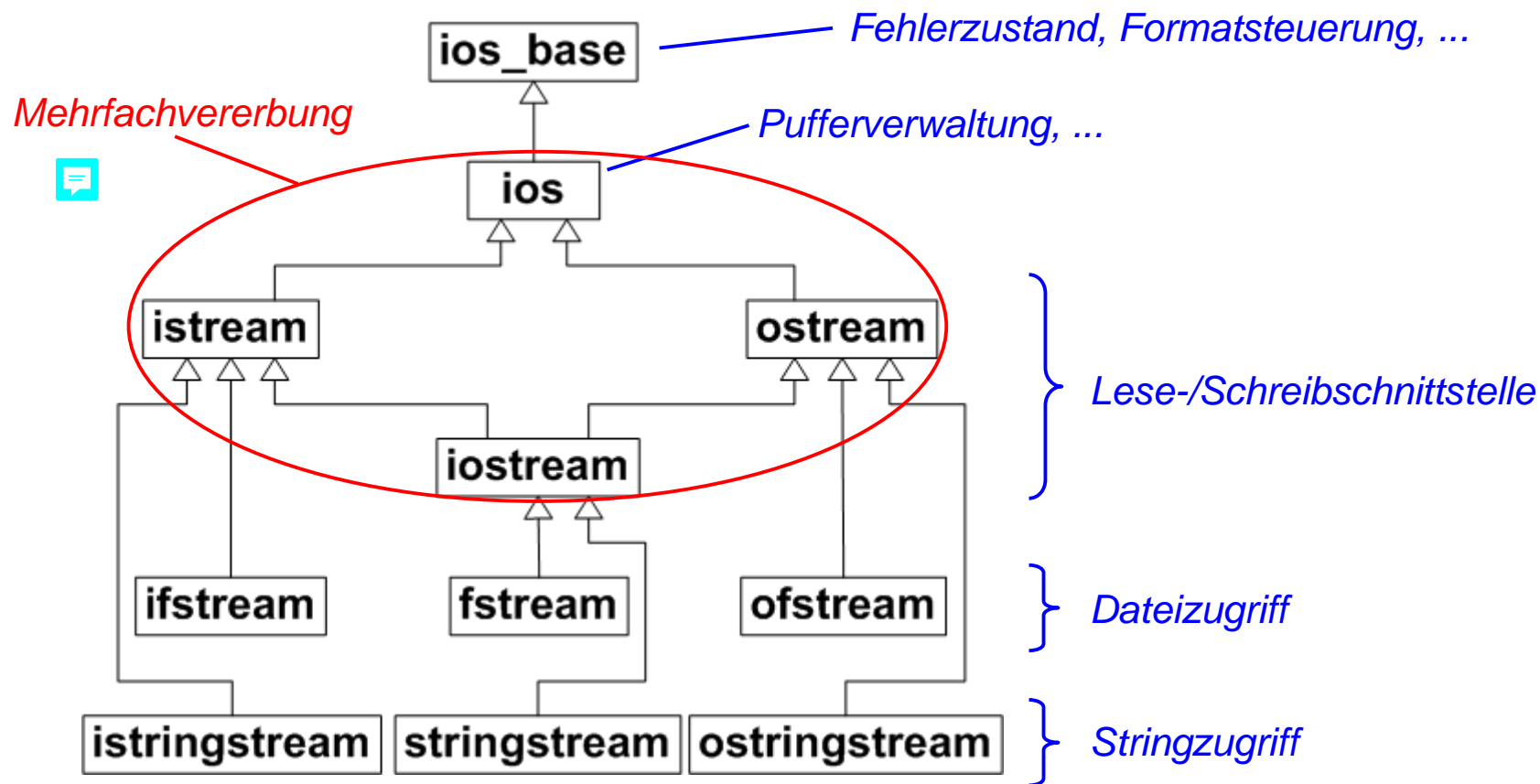


C++ Templates sind sehr viel mächtiger als die Generics von Java:

- in Java gibt es keine Nichttyp-Parameter
- in Java sind nur Klassen als Argumente für Typ-Parameter erlaubt, in C++ sind dagegen alle Typen als Argumente erlaubt, auch Grundtypen wie z.B. `int` und abgeleitete Typen wie z.B. `int*`
- bei Java gibt es nur eine Implementierung eines Generics
der Compiler ersetzt die Typ-Parameter durch die Klasse `Object` und ergänzt bei der Benutzung der Generics entsprechende Up- und Downcasts
- bei C++ erstellt der Compiler für jede Instanziierung eines Templates mit anderen Argumenten per Copy und Paste eine eigene Implementierung 
kommen in einem Programm viele unterschiedliche Argumente für das gleiche Template vor, kann wegen der mehrfachen Vervielfältigung der Implementierung die Übersetzung lange dauern und der ausführbare Code sehr umfangreich werden
bei Fehlern in Templates sind die Fehlermeldungen des Compilers oft sehr umfangreich und schwer zu verstehen

C++ Standardbibliothek: Ein-/Ausgabe (1)

Klassen-Hierarchie der **Ein-/Ausgabe-Streams** (vereinfacht, eigentlich Templates) 



C++ Standardbibliothek: Ein-/Ausgabe (2)

Prinzipieller Aufbau der Entitätsklasse für **Ausgabe-Streams** (vereinfacht):

```
class ostream : virtual public ios {  
public:  
    // Ausgabeoperatoren für u.a. alle Grundtypen:  
    ostream& operator<<(int);  
    ...  
    // Ausgabeoperatoren für Manipulatoren:  
    ostream& operator<<(ostream& (*)(ostream&));  
    ...  
    // Zeichenausgabe und Pufferleerung:  
    ostream& put(char);  
    ostream& flush();  
    ...  
};  
  
// Manipulatoren:  
ostream& endl(ostream&);  
ostream& flush(ostream&);  
...
```

*virtual: Unterklassen von ostream
sollen bei Mehrfachvererbung
nur einmal von ios erben*

*Ein Manipulator ist eine Funktion, die
vom Ausgabeoperator aufgerufen wird*

```
// globale Variablen:  
extern ostream cout;  
extern ostream cerr;  
...
```

C++ Standardbibliothek: Ein-/Ausgabe (3)

- Verwenden von ostream-Funktionen:

```
#include <iostream>
```

```
std::cout.setf(std::ios_base::fixed, std::ios_base::floatfield);
```

```
std::cout.precision(1);
```

```
std::cout << 1.26 << std::endl; // gibt 1.3 aus
```

```
std::cout.width(4);
```

```
std::cout.fill('0');
```

```
std::cout << 1 << std::endl; // gibt 0001 aus
```

gilt nur für die nächste Ausgabe

- das gleiche mit Manipulatoren:

```
#include <iostream> // cout, operator<<, fixed, endl
```

```
#include <iomanip> // setprecision, setw, setfill
```

```
std::cout << std::fixed << std::setprecision(1) << 1.26 << std::endl;
```

```
std::cout << std::setw(4) << std::setfill('0') << 1 << std::endl;
```



C++ Standardbibliothek: Ein-/Ausgabe (4)

Prinzipieller Aufbau der Entitätsklasse für **Eingabe-Streams** (vereinfacht):

```
class istream : virtual public ios {  
public:  
    // Eingabeoperatoren für alle Grundtypen:  
    istream& operator>>(int&);  
    ...  
    // Eingabeoperatoren für Manipulatoren:  
    istream& operator>>(ios_base& (*)(ios_base&));  
    ...  
    // Zeichen- und Zeichenketteneingabe:  
    istream& get(char&);  
    istream& getline(char*, int);  
    ...  
};  
  
// Manipulatoren:  
ios_base& hex(ios_base&);  
...
```

*virtual: Unterklassen von istream
sollen bei Mehrfachvererbung
nur einmal von ios erben*

```
// globale Variablen:  
extern istream cin;  
...
```


C++ Standardbibliothek: Ein-/Ausgabe (5)

- Verwenden von istream-Funktionen

```
#include <iostream>
```

```
std::cin.setf(ios_base::hex, ios_base::basefield);
```

```
int n;
```

```
std::cin >> n;    // liest hexadezimale Zahl
```

```
std::cout << n << std::endl;    // gibt n dezimal aus
```

- das gleiche mit Manipulatoren

```
#include <iostream>    // cin, operator>>, hex
```

```
int n;
```

```
std::cin >> std::hex >> n;    // liest hexadezimale Zahl
```


```
std::cout << n << std::endl;    // gibt n dezimal aus
```

C++ Standardbibliothek: Ein-/Ausgabe (6)

Prinzipieller Aufbau der Entitätsklassen für **File-Streams** (vereinfacht):

```
class ofstream : public ostream
{
public:
    ofstream();
    ofstream(const char*);
    bool is_open();
    void open(const char*);
    void close();
    ...
};
```

Lesezugriff mit ifstream analog

- Verwenden von ofstream-Funktionen:
std::ofstream aFile;
aFile.open("Beispiel.txt");
aFile << "Hallo\n";
... // Schreiben wie bei std::cout
aFile.close();
- das gleiche mit Konstruktor und Destruktor:
std::ofstream aFile("Beispiel.txt");
aFile << "Hallo\n";
...
// Destruktor von aFile sorgt für das close 

C++ Standardbibliothek: std::string (1)



Ausschnitt aus der Wertklasse std::string (vereinfacht): 

```
class string
{
public:
    string(); // Konstruktoren
    string(const string& str);
    string(const char *s);
    ~string(); // Destruktor

    string& operator=(const string& str); // Zuweisungen
    string& operator=(const char *s);
    string& operator+=(const string& str);
    string& operator+=(const char *s);

    const char *c_str() const; // Datenabfragen
    unsigned length() const;

    const char& operator[](unsigned pos) const;
    char& operator[](unsigned pos);
    ...
};
```



C++ Standardbibliothek: std::string (2)

Operatoren außerhalb der Wertklasse std::string (vereinfacht):

```
// Verknüpfungen
string operator+(const string& s1, const string& s2);
...
// Vergleiche
bool operator==(const string& s1, const string& s2);
...
// Ein-/Ausgabe
istream& operator>>(istream& is, string& s);
ostream& operator<<(ostream& os, const string &s);
...
```

Anwendungsbeispiel:

```
#include <string> // damit std::string bekannt ist
char buffer[10];
std::cin >> buffer; // Risiko eines Pufferüberlaufs
std::string s;
std::cin >> s; // string-Objekt und operator>> sorgen für genug Speicher
```

Beispielprogramm std::string

```
#include <iostream>
#include <string>



int main()
{
    std::string a = "halli"; // a("halli")
    std::string s = "hallo"; // s("hallo")
    std::string t; // leerer String

    // compare, copy and concatenate strings
    if (a < s) // operator<(a, s)
    {
        t = a + s; // t.operator=(operator+(a, s))
    }

    // print string values and addresses
    std::cout << a << '\n' << s << '\n' << t << '\n'; // operator<<(..., ...)
    std::cout << sizeof a << '\n' << sizeof s << '\n' << sizeof t << '\n';
    std::cout << a.length() << '\n' << s.length() << '\n' << t.length() << '\n';
}
```

C++ Standardbibliothek: Container `std::vector` (1)

Ausschnitt aus dem Wertklassen-Template `std::vector<>` (vereinfacht):

```
template <typename T> class vector   
{  
public:  
    vector();  
    vector(std::size_t n);  
    vector(const vector& v);  
    ~vector();  
    vector<T& operator=(const vector& v);  
    std::size_t size() const;  
    void resize(std::size_t n, T c = T());  
    T& operator[] (std::size_t i);   
    T& at(std::size_t i);  
    ...  
};  
template <typename T>  
bool operator==(const vector<T>& v, const vector<T>& w);  
...
```

Elementtyp als Template-Parameter

C++ Standardbibliothek: Container `std::vector` (2)

- zu fast jedem Typ kann ein Vektortyp abgeleitet werden:

```
#include <vector> // damit std::vector<> bekannt ist
```

```
// Vektor von vier ganzen Zahlen, alle mit 0 initialisiert:
```

```
std::vector<int> vi(4);
```

```
// Vektor von zwei Strings, mit Leerstrings initialisiert:
```

```
std::vector<std::string> vs(2);
```

- ein Vektor kennt im Gegensatz zum Feld seine Länge:

```
for (unsigned i = 0; i < vi.size(); i++) ...
```

- Vektorzugriff per `[]` ohne oder per `.at()` mit Indexprüfung:


```
vi[2] = 1; // std::vector<int>::operator[](&vi, 2) = 1;
```

```
vi.at(2) = 1; // std::vector<int>::at(&vi, 2) = 1;
```

- ein Vektor kann im Gegensatz zum Feld per Zuweisungs-Operator kopiert und per Vergleichsoperatoren verglichen werden

C++ Standardbibliothek: Container `std::array`


Seit C++11 gibt es zusätzlich zu `std::vector<>` ein vereinfachtes Klassentemplate `std::array<>` für Felder mit statischer Länge:

```
template <typename T, std::size_t N> class array   
{  
public:  
    T _elemente[N];  
    // Konstruktoren, Destruktor, Zuweisungsoperatoren vom Compiler implizit erzeugt ...  
    unsigned size() const;  
    T& operator[ ](std::size_t i);  
    T& at(std::size_t i);  
    ...  
};  
  
template <typename T, std::size_t N>  
bool operator==(const array<T,N>& v, const array<T,N>& w);  
...  

```


Feldlänge als Nichttyp-Parameter


Name des eingebetteten C-Felds im Standard nicht festgelegt



Beispielprogramm `std::vector<>`

```
#include <iostream>
#include <vector> // alternativ: #include <array>

int main()
{
    std::vector<int> v(4); // alternativ: std::array<int,4> v;
    v.at(0) = 3421;
    v.at(1) = 3442; 
    v.at(2) = 3635;
    v.at(3) = 3814;

    // print vector values
    for (std::size_t i = 0; i < v.size(); ++i)
    {
        std::cout << i << ": " << v[i] << '\n'; // v.operator[](i) 
    }

    // print vector size
    std::cout << "sizeof v = " << sizeof v << '\n';
    std::cout << "v.size() = " << v.size() << '\n';
}
```

C++ Standardbibliothek: Algorithmus `std::max<>`

Das Funktions-Template `std::max<>` zur Bestimmung des Maximums zweier Werte ist für jeden Werttyp nutzbar, der `operator<` unterstützt:

```
template<typename T>
const T& max(const T& a, const T& b);
```

- Beispiel:

```
#include <algorithm> // damit std::max<> bekannt ist


int main()
{
    // T = int
    int n = std::max(1, 2);

    // T = std::string
    std::string s = std::max(std::string("abc"), std::string("def"));
}
```

Was würde `std::max("abc", "def")` liefern? 


C++ Standardbibliothek: Algorithmus `std::find<>` (1)


Das Funktions-Template `std::find<>` zur linearen Suche eines Werts ist für jeden Typ mit zugeordnetem Iterator-Typ nutzbar:

```
template<typename I, typename T>
I find(I first, I last, const T& value); 
```

- Beispiel mit einfachem C-Array:

```
#include <algorithm> // damit std::find<> bekannt ist
...
```

```
 int a[] = {3421, 3442, 3635, 3814};
int *begin = a; // Zeiger auf Elementtyp dienen als Iteratoren
int *end = a + 4;
```

```
 auto i = std::find(begin, end, 3442); // I = int* und T = int
if (i != end) {
    std::cout << *i << " ist in a enthalten\n";
}
```

C++ Standardbibliothek: Algorithmus `std::find<>` (2)

- Beispiel mit C++ Container:

```
#include <array>           // alternativ vector, list, ...
#include <algorithm>       // damit std::find<> bekannt ist
...

std::array<int,4> a{3421, 3442, 3635, 3814};
// std::vector<int> a{3421, 3442, 3635, 3814};
// std::list<int> a{3421, 3442, 3635, 3814};
auto begin = a.begin();
auto end = a.end();

auto i = std::find(begin, end, 3442);
if (i != end) {
    std::cout << *i << " ist in a enthalten\n";
}
```

C++ Standardbibliothek: Iteratoren (1)

Damit C++ Algorithmen wie `std::find<>` mit einer Container-Klasse funktionieren, muss die Klasse einen Typ iterator sowie Memberfunktionen begin() und end() bereitstellen (*vereinfacht*)


- Beispiel `std::array<>`: ein einfacher Zeiger auf Elementtyp dient als Iterator

```
template <typename T, std::size_t N >
class array
{
public:
    ...
    typedef T* iterator;
    iterator begin() { return &this->_elemente[0]; }
    iterator end()   { return &this->_elemente[N]; }
};
```

C++ Standardbibliothek: Iteratoren (2)

- Beispiel intlist aus Teil 5: Iterator mit zusätzlichen Typnamen

```
class intlist final {  
    ...  
    class iterator final {  
        ... // operator!=, operator*, operator++ wie in Teil 5  
        typedef std::input_iterator_tag iterator_category;  
        typedef T value_type;  
        typedef std::ptrdiff_t difference_type;  
        typedef T* pointer;  
        typedef T& reference;  
    };  
    ...  
};
```

 Typnamen, die in den Funktions-Templates der C++ Bibliothek benutzt werden, z.B. in `std::find`

Seit C++11 statt *typedef* auch alternative Syntax: `using value_type = T;`

C++ Standardbibliothek: Intelligente Zeiger (1)

Einfache Zeigervariable (*raw pointers*) sind eine regelmäßige Fehlerquelle:



- es kommt zu **Speicherlecks** (*memory leaks*), wenn für mit `new` allokierten Heap-Speicher das zugehörige `delete` fehlt
- es kommt zu **Speicherzugriffsfehlern**, wenn ein Zeiger weiter dereferenziert wird, obwohl der referenzierte Speicher gar nicht mehr allokiert ist (*dangling pointers*)

Intelligente Zeiger (*smart pointers*) sind Wrapper für einfache Zeiger:

- das Klassentemplate **`std::unique_ptr<T>`** bindet die Lebensdauer eines Heap-Speicherstücks vom Typ `T` exklusiv an die Lebensdauer einer Variablen
Der Destruktor der Zeigerklasse garantiert den `delete` -Aufruf.
- das Klassentemplate **`std::shared_ptr<T>`** ergänzt Heap-Speicherstücke um einen Referenzzähler und erlaubt so mehrere Zeiger pro Speicherstück
Konstruktoren, Destruktor und Copy-Zuweisung zählen den Referenzzähler hoch und runter. Bei einem Zählerstand 0 wird `delete` aufgerufen.
Zyklische Referenzierungen müssen mit `std::weak_ptr<T>` aufgelöst werden.

C++ Standardbibliothek: Intelligente Zeiger (2)


Ausschnitt aus dem Klassentemplate `std::unique_ptr<>` (vereinfacht): 

```
template <typename T> class unique_ptr
{
private:
    T *_p;
public:
    unique_ptr();
    explicit unique_ptr(T *p); 
    unique_ptr(unique_ptr&& u); // nur Move-Konstruktor, kein Copy-Konstruktor
     ~ unique_ptr();
    unique_ptr& operator=(unique_ptr&& u); // nur Move-Zuweisung
    T& operator*() const;
    T* operator->() const;
    ...
};
```

Objekte der Klasse `unique_ptr` sind nicht kopierbar. Dadurch wird die Bindung des referenzierten Speicherbereichs an die Lebensdauer genau einer Variablen sichergestellt. 

C++ Standardbibliothek: Intelligente Zeiger (3)

Ausschnitt aus dem Klassentemplate `std::shared_ptr<>` (vereinfacht): 

```
template <typename T> class shared_ptr
{
private:
    T *_M_ptr;
    _control_block *_M_pi; // der Kontrollblock enthält den Referenzzähler
public:
    shared_ptr();
    explicit shared_ptr(T *p);
    shared_ptr(const shared_ptr& s);
    shared_ptr(shared_ptr&& s);
     ~ shared_ptr();
    shared_ptr& operator=(const shared_ptr& s);
    shared_ptr& operator=(shared_ptr&& s);
    T& operator*() const;
    T* operator->() const;
    ...
};
```

Objekte der Klasse
`shared_ptr`
sind kopierbar.
Jede Kopie erhöht den
Referenzzähler um 1.

Beispielprogramm Intelligente Zeiger (1)

- Fabrikfunktion für die Entitätenklasse termin aus Teil 5:

```
class termin final
{
    ...
public:
    static std::unique_ptr<termin> new_instance(const datum&,
                                                const std::string&);
    ...
};
```

- Implementierung der Fabrikfunktion:

```
std::unique_ptr<termin> termin::new_instance(const datum& d, const std::string& s)
{
    return std::unique_ptr<termin>(new termin(d, s));
}
```

Beispielprogramm Intelligente Zeiger (2)

- Objektbenutzung:

...

```
std::list<std::shared_ptr<termin>> > pruefer_kalender;
```

```
std::list<std::shared_ptr<termin>> > kandidaten_kalender;
```

```
std::shared_ptr<termin> pruefung
```

```
 = termin::new_instance(datum::heute(), "Pruefung Systemprogrammierung");
```

```
pruefer_kalender.push_back(pruefung);  // Referenzzähler = 2
```

```
kandidaten_kalender.push_back(pruefung);  // Referenzzähler = 3
```

```
pruefung->verschieben({1, 4, 2040}); // Aufruf operator->( )
```

...

*Die Destruktoraufrufe für pruefung, kandidaten_kalender und pruefer_kalender zählen den Referenzzähler des Termin-Objekts herunter, beim letzten Aufruf wird **delete** aufgerufen*



C++ Standardbibliothek: Index

Ausgabe-Stream 6-5,6-6
dangling pointer 6-22
Eingabe-Stream 6-7,6-8
File-Stream 6-9
Funktionstemplate 6-2
intelligenter Zeiger 6-22
Iterator 6-20,6-21
Klassentemplate 6-2
memory leak 6-22
smart pointer 6-22

std::array 6-15
std::find 6-18,6-19
std::istream 6-7
std::max 6-17
std::ofstream 6-9
std::ostream 6-5
std::shared_ptr 6-22,6-24
std::string 6-10,6-11,6-12
std::unique_ptr 6-22,6-23
std::vector 6-13,6-14,6-16
Stream 6-4
Template 6-2,6-3