

Systemnahe Programmierung in C/C++

Peter Piehler

`peter.piehler@symetrion.com`

Lehrstuhl für Datenbanken und Informationssysteme
Fakultät für Mathematik und Informatik

2006-12-13

Agenda

- 1 Iteratoren
- 2 C++ Speicherverwaltung
- 3 Aufgabe

1 Iteratoren

2 C++ Speicherverwaltung

3 Aufgabe

iterator Ein iterator ist ein Objekt, welches benutzt wird um eine Menge von Objekten zu durchlaufen. (Header: `<iterator>`)

Iteratoren ermöglichen

- einen einheitlichen Zugriff auf Containerdaten
- es, dass Algorithmen keine Kenntnisse über den Aufbau der Container-Datenstrukturen bedürfen
- es, dass Container von der Bereitstellung zahlreicher Elementfunktionen befreit werden

Alles, was sich wie ein Iterator verhält, ist auch ein Iterator.

iterator Ein iterator ist ein Objekt, welches benutzt wird um eine Menge von Objekten zu durchlaufen. (Header: `<iterator>`)

Iteratoren ermöglichen

- einen einheitlichen Zugriff auf Containerdaten
- es, dass Algorithmen keine Kenntnisse über den Aufbau der Container-Datenstrukturen bedürfen
- es, dass Container von der Bereitstellung zahlreicher Elementfunktionen befreit werden

Alles, was sich wie ein Iterator verhält, ist auch ein Iterator.

iterator Ein iterator ist ein Objekt, welches benutzt wird um eine Menge von Objekten zu durchlaufen. (Header: `<iterator>`)

Iteratoren ermöglichen

- einen einheitlichen Zugriff auf Containerdaten
- es, dass Algorithmen keine Kenntnisse über den Aufbau der Container-Datenstrukturen bedürfen
- es, dass Container von der Bereitstellung zahlreicher Elementfunktionen befreit werden

Alles, was sich wie ein Iterator verhält, ist auch ein Iterator.

- es gibt drei Anforderungsgruppen:
 - 1 Zugriff auf Elemente (z.B. mittels operator*)
 - 2 Bewegung durch eine Menge von Elementen (z.B. mittels operator++)
 - 3 Vergleich zweier Iteratoren (z.B. mittels operator!=)
- Für jede dieser Gruppe werden von den Iteratoren Operationen oder Ausdrücke (Kombination von Operationen) bereitgestellt:

Gruppe	Operation	Ausdruck
Vergleich	==, !=, <, >, <=, >=	
Bewegung	++(prefix/postfix), --, +, -, +=, -=	
Zugriff	[], ->	*i, = *i

- es gibt drei Anforderungsgruppen:
 - 1 Zugriff auf Elemente (z.B. mittels operator*)
 - 2 Bewegung durch eine Menge von Elementen (z.B. mittels operator++)
 - 3 Vergleich zweier Iteratoren (z.B. mittels operator!=)
- Für jede dieser Gruppe werden von den Iteratoren Operationen oder Ausdrücke (Kombination von Operationen) bereitgestellt:

Gruppe	Operation	Ausdruck
Vergleich	==, !=, <, >, <=, >=	
Bewegung	++(prefix/postfix), --, +, -, +=, -=	
Zugriff	[], ->	*i =, = *i

- es gibt drei Anforderungsgruppen:
 - 1 Zugriff auf Elemente (z.B. mittels operator*)
 - 2 Bewegung durch eine Menge von Elementen (z.B. mittels operator++)
 - 3 Vergleich zweier Iteratoren (z.B. mittels operator!=)
- Für jede dieser Gruppe werden von den Iteratoren Operationen oder Ausdrücke (Kombination von Operationen) bereitgestellt:

Gruppe	Operation	Ausdruck
Vergleich	==, !=, <, >, <=, >=	
Bewegung	++(prefix/postfix), --, +, -, +=, -=	
Zugriff	[], ->	*i, = *i

Iteratoranforderung

- es gibt drei Anforderungsgruppen:
 - 1 Zugriff auf Elemente (z.B. mittels operator*)
 - 2 Bewegung durch eine Menge von Elementen (z.B. mittels operator++)
 - 3 Vergleich zweier Iteratoren (z.B. mittels operator!=)
- Für jede dieser Gruppe werden von den Iteratoren Operationen oder Ausdrücke (Kombination von Operationen) bereitgestellt:

Gruppe	Operation	Ausdruck
Vergleich	==, !=, <, >, <=, >=	
Bewegung	++(prefix/postfix), --, +, -, +=, -=	
Zugriff	[], ->	*i =, = *i

- es gibt drei Anforderungsgruppen:
 - 1 Zugriff auf Elemente (z.B. mittels operator*)
 - 2 Bewegung durch eine Menge von Elementen (z.B. mittels operator++)
 - 3 Vergleich zweier Iteratoren (z.B. mittels operator!=)
- Für jede dieser Gruppe werden von den Iteratoren Operationen oder Ausdrücke (Kombination von Operationen) bereitgestellt:

Gruppe	Operation	Ausdruck
Vergleich	==, !=, <, >, <=, >=	
Bewegung	++(prefix/postfix), --, +, -, +=, -=	
Zugriff	[], ->	*i, = *i

- Iteratoren werden in fünf Kategorien unterteilt

① Input

② Output

③ Forward (genügt auch den Anforderungen von Input und Output)

④ Bidirectional (genügt auch den Anforderungen von Forward)

⑤ Random-Access (genügt auch den Anforderungen von Bidirectional)

- Ein Iterator ist einer Kategorie zugeordnet und sichert damit zu, dass Operationen und Ausdrücke (aus den Anforderungsgruppen) verwendbar sind:

Input = $*i$, $->$, $++$, $==$, $!=$

Output $*i$, $++$

Forward Vereinigung aus Input und Output

Bidirectional Forward und $--$

Random-Access Bidirectional und $[]$, $- =$, $+ =$, $+$, $-$

- Iteratoren werden in fünf Kategorien unterteilt

① Input

② Output

③ Forward (genügt auch den Anforderungen von Input und Output)

④ Bidirectional (genügt auch den Anforderungen von Forward)

⑤ Random-Access (genügt auch den Anforderungen von Bidirectional)

- Ein Iterator ist einer Kategorie zugeordnet und sichert damit zu, dass Operationen und Ausdrücke (aus den Anforderungsgruppen) verwendbar sind:

Input = $*i$, $->$, $++$, $==$, $!=$

Output = $*i$, $++$

Forward = Vereinigung aus Input und Output

Bidirectional = Forward und $--$

Random-Access = Bidirectional und $[], - =, + =, +, -$

- Iteratoren werden in fünf Kategorien unterteilt
 - 1 Input
 - 2 Output
 - 3 Forward (genügt auch den Anforderungen von Input und Output)
 - 4 Bidirectional (genügt auch den Anforderungen von Forward)
 - 5 Random-Access (genügt auch den Anforderungen von Bidirectional)
- Ein Iterator ist einer Kategorie zugeordnet und sichert damit zu, dass Operationen und Ausdrücke (aus den Anforderungsgruppen) verwendbar sind:

Input = $*i$, $->$, $++$, $==$, $!=$

Output = $*i$, $++$

Forward = Vereinigung aus Input und Output

Bidirectional = Forward und $--$

Random-Access = Bidirectional und $[], -=, +=, +, -$

- Iteratoren werden in fünf Kategorien unterteilt
 - 1 Input
 - 2 Output
 - 3 Forward (genügt auch den Anforderungen von Input und Output)
 - 4 Bidirectional (genügt auch den Anforderungen von Forward)
 - 5 Random-Access (genügt auch den Anforderungen von Bidirectional)
- Ein Iterator ist einer Kategorie zugeordnet und sichert damit zu, dass Operationen und Ausdrücke (aus den Anforderungsgruppen) verwendbar sind:

Input = $*i$, $->$, $++$, $==$, $!=$

Output = $*i$, $++$

Forward = Vereinigung aus Input und Output

Bidirectional = Forward und $--$

Random-Access = Bidirectional und $[], - =, + =, +, -$

- Iteratoren werden in fünf Kategorien unterteilt
 - 1 Input
 - 2 Output
 - 3 Forward (genügt auch den Anforderungen von Input und Output)
 - 4 Bidirectional (genügt auch den Anforderungen von Forward)
 - 5 Random-Access (genügt auch den Anforderungen von Bidirectional)
- Ein Iterator ist einer Kategorie zugeordnet und sichert damit zu, dass Operationen und Ausdrücke (aus den Anforderungsgruppen) verwendbar sind:

Input $= *i, ->, ++, ==, !=$

Output $*i =, ++$

Forward Vereinigung aus Input und Output

Bidirectional Forward und $--$

Random-Access Bidirectional und $[], - =, + =, +, -$

- Iteratoren werden in fünf Kategorien unterteilt
 - 1 Input
 - 2 Output
 - 3 Forward (genügt auch den Anforderungen von Input und Output)
 - 4 Bidirectional (genügt auch den Anforderungen von Forward)
 - 5 Random-Access (genügt auch den Anforderungen von Bidirectional)
- Ein Iterator ist einer Kategorie zugeordnet und sichert damit zu, dass Operationen und Ausdrücke (aus den Anforderungsgruppen) verwendbar sind:

Input $= *i, ->, ++, ==, !=$

Output $*i =, ++$

Forward Vereinigung aus Input und Output

Bidirectional Forward und $--$

Random-Access Bidirectional und $[], -=, +=, +, -$

- Iteratoren werden in fünf Kategorien unterteilt
 - 1 Input
 - 2 Output
 - 3 Forward (genügt auch den Anforderungen von Input und Output)
 - 4 Bidirectional (genügt auch den Anforderungen von Forward)
 - 5 Random-Access (genügt auch den Anforderungen von Bidirectional)
- Ein Iterator ist einer Kategorie zugeordnet und sichert damit zu, dass Operationen und Ausdrücke (aus den Anforderungsgruppen) verwendbar sind:

Input = $*i$, $->$, $++$, $==$, $!=$

Output $*i$, $++$

Forward Vereinigung aus Input und Output

Bidirectional Forward und $--$

Random-Access Bidirectional und $[], -=, +=, +, -$

- Iteratoren werden in fünf Kategorien unterteilt
 - ① Input
 - ② Output
 - ③ Forward (genügt auch den Anforderungen von Input und Output)
 - ④ Bidirectional (genügt auch den Anforderungen von Forward)
 - ⑤ Random-Access (genügt auch den Anforderungen von Bidirectional)
- Ein Iterator ist einer Kategorie zugeordnet und sichert damit zu, dass Operationen und Ausdrücke (aus den Anforderungsgruppen) verwendbar sind:

Input $= *i, - >, ++, ==, !=$

Output $*i =, ++$

Forward Vereinigung aus Input und Output

Bidirectional Forward und $--$

Random-Access Bidirectional und $[], - =, + =, +, -$

- Iteratoren werden in fünf Kategorien unterteilt
 - ① Input
 - ② Output
 - ③ Forward (genügt auch den Anforderungen von Input und Output)
 - ④ Bidirectional (genügt auch den Anforderungen von Forward)
 - ⑤ Random-Access (genügt auch den Anforderungen von Bidirectional)
- Ein Iterator ist einer Kategorie zugeordnet und sichert damit zu, dass Operationen und Ausdrücke (aus den Anforderungsgruppen) verwendbar sind:

Input $= *i, - >, ++, ==, !=$

Output $*i =, ++$

Forward Vereinigung aus Input und Output

Bidirectional Forward und $--$

Random-Access Bidirectional und $[], - =, + =, +, -$

- Iteratoren werden in fünf Kategorien unterteilt
 - ① Input
 - ② Output
 - ③ Forward (genügt auch den Anforderungen von Input und Output)
 - ④ Bidirectional (genügt auch den Anforderungen von Forward)
 - ⑤ Random-Access (genügt auch den Anforderungen von Bidirectional)
- Ein Iterator ist einer Kategorie zugeordnet und sichert damit zu, dass Operationen und Ausdrücke (aus den Anforderungsgruppen) verwendbar sind:

Input $= *i, - >, ++, ==, !=$

Output $*i =, ++$

Forward Vereinigung aus Input und Output

Bidirectional Forward und $--$

Random-Access Bidirectional und $[], - =, + =, +, -$

- Iteratoren werden in fünf Kategorien unterteilt
 - ① Input
 - ② Output
 - ③ Forward (genügt auch den Anforderungen von Input und Output)
 - ④ Bidirectional (genügt auch den Anforderungen von Forward)
 - ⑤ Random-Access (genügt auch den Anforderungen von Bidirectional)
- Ein Iterator ist einer Kategorie zugeordnet und sichert damit zu, dass Operationen und Ausdrücke (aus den Anforderungsgruppen) verwendbar sind:

Input $= *i, - >, ++, ==, !=$

Output $*i =, ++$

Forward Vereinigung aus Input und Output

Bidirectional Forward und $--$

Random-Access Bidirectional und $[], - =, + =, +, -$

- Für jeden Iterator sind die Eigenschaften über die Templatestruktur `iterator_traits` abfragbar:

```
template<typename _Iterator>
struct iterator_traits
{
    typedef typename _Iterator::iterator_category iterator_category;
    typedef typename _Iterator::value_type value_type;
    typedef typename _Iterator::difference_type difference_type;
    typedef typename _Iterator::pointer pointer;
    typedef typename _Iterator::reference reference;
};

template<typename _Tp>
struct iterator_traits<_Tp*>
{
    typedef random_access_iterator_tag iterator_category;
    typedef _Tp value_type;
    typedef ptrdiff_t difference_type;
    typedef _Tp* pointer;
    typedef _Tp& reference;
};

template<typename _Tp>
struct iterator_traits<const _Tp*>
{
    typedef random_access_iterator_tag iterator_category;
    typedef _Tp value_type;
    ...
};
```

- Für jeden Iterator sind die Eigenschaften über die Templatestruktur `iterator_traits` abfragbar:

```
template<typename _Iterator>
struct iterator_traits
{
    typedef typename _Iterator::iterator_category iterator_category;
    typedef typename _Iterator::value_type value_type;
    typedef typename _Iterator::difference_type difference_type;
    typedef typename _Iterator::pointer pointer;
    typedef typename _Iterator::reference reference;
};

template<typename _Tp>
struct iterator_traits<_Tp*>
{
    typedef random_access_iterator_tag iterator_category;
    typedef _Tp value_type;
    typedef ptrdiff_t difference_type;
    typedef _Tp* pointer;
    typedef _Tp& reference;
};

template<typename _Tp>
struct iterator_traits<const _Tp*>
{
    typedef random_access_iterator_tag iterator_category;
    typedef _Tp value_type;
    ...
};
```

value_type Elementtyp, auf den über den Iterator zugegriffen werden kann.

difference_type Der Typ, der geliefert wird wenn die Distanz zweier Iteratoren berechnet wird.

```
template< typename InputIterator >  
typename iterator_traits< InputIterator >::difference_type  
distance(InputIterator first, InputIterator last);
```

pointer Zeigertyp zu value_type

reference Referenztyp zu value_type

value_type Elementtyp, auf den über den Iterator zugegriffen werden kann.

difference_type Der Typ, der geliefert wird wenn die Distanz zweier Iteratoren berechnet wird.

```
template< typename InputIterator >
typename iterator_traits< InputIterator >::difference_type
distance(InputIterator first, InputIterator last);
```

pointer Zeigertyp zu value_type

reference Referenztyp zu value_type

value_type Elementtyp, auf den über den Iterator zugegriffen werden kann.

difference_type Der Typ, der geliefert wird wenn die Distanz zweier Iteratoren berechnet wird.

```
template< typename InputIterator >
typename iterator_traits< InputIterator >::difference_type
distance(InputIterator first, InputIterator last);
```

pointer Zeigertyp zu value_type

reference Referenztyp zu value_type

value_type Elementtyp, auf den über den Iterator zugegriffen werden kann.

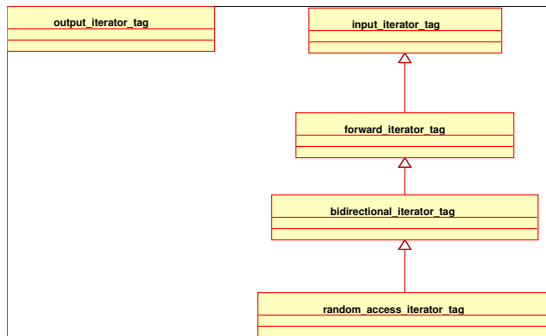
difference_type Der Typ, der geliefert wird wenn die Distanz zweier Iteratoren berechnet wird.

```
template< typename InputIterator >
typename iterator_traits< InputIterator >::difference_type
distance(InputIterator first, InputIterator last);
```

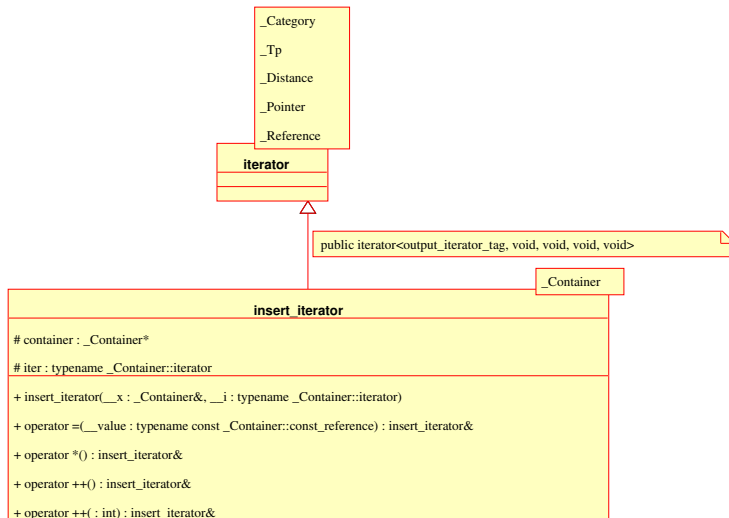
pointer Zeigertyp zu value_type

reference Referenztyp zu value_type

`iterator_category` Gibt an zu welcher Kategorie der Iterator gehört. Ist vom Typ `input_iterator_tag`, `output_iterator_tag`, `forward_iterator_tag`, `bidirectional_iterator_tag` oder `random_iterator_tag`.



insert_iterator



Iterator - Algorithmus - Container

Iteratorkategorie	Algorithmen	Container
Random	sort, random_shuffle	vector, deque
Bidirectional	copy_backward, reverse	list, map
Forward	search, replace, fill, remove, unique, partition	
Input, Output	for_each, transform, find, count, copy	

- Neben dem Typ `iterator` existiert der Typ `const_iterator`.
- Er findet für den lesenden Zugriff Verwendung.
- Mit Hilfe des `const_iterator` wird sichergestellt, dass es nicht zu unbeabsichtigt Elementen neue Werte zugewiesen werden.

```
container_type::const_iterator itEnd( container.end() );  
for( container_type::const_iterator it( container.begin() ), it != itEnd, ++it )  
{  
    *p= Value;  
}
```

- Die Containermethoden `begin` und `end` liefern einen normalen Iterator. Der nächste C++-Standard wird voraussichtlich die Methoden `cbegin` und `cend` definieren, welche einen `const_iterator` als Rückgabewert haben.

- Ein Reversiterator ist ein Iteratoradapter.
- Bidirectional- und Random-Access-Iteratoren können vorwärts als auch rückwärts durchlaufen werden – > für diese steht dieser Iteratoradapter bereit.
- Mit einem Reversiterator werden die Elemente in umgekehrter Richtung durchlaufen.
- Die Ausführung von `operator++` auf einen Reverseiterator entspricht der Ausführung des `operator--` auf einen Iterator.
- Reversible Container stellen die Typdefinitionen `reverse_iterator` und `const_reverse_iterator` bereit.
- Reversible Container besitzen die Methoden `rbegin` und `rend`, welche Reversiteratoren liefern.
- Der kommende C++-Standard wird voraussichtlich für diese Container die Methoden `crbegin` und `crend`, welche `const_reverse_iterator` zurückliefern, beinhalten.

- Ein Reversiterator ist ein Iteratoradapter.
- Bidirectional- und Random-Access-Iteratoren können vorwärts als auch rückwärts durchlaufen werden – $\>$ für diese steht dieser Iteratoradapter bereit.
- Mit einem Reversiterator werden die Elemente in umgekehrter Richtung durchlaufen.
- Die Ausführung von `operator++` auf einen Reverseiterator entspricht der Ausführung des `operator--` auf einen Iterator.
- Reversible Container stellen die Typdefinitionen `reverse_iterator` und `const_reverse_iterator` bereit.
- Reversible Container besitzen die Methoden `rbegin` und `rend`, welche Reversiteratoren liefern.
- Der kommende C++-Standard wird voraussichtlich für diese Container die Methoden `crbegin` und `crend`, welche `const_reverse_iterator` zurückliefern, beinhalten.

- Ein Reversiterator ist ein Iteratoradapter.
- Bidirectional- und Random-Access-Iteratoren können vorwärts als auch rückwärts durchlaufen werden – $>$ für diese steht dieser Iteratoradapter bereit.
- Mit einem Reversiterator werden die Elemente in umgekehrter Richtung durchlaufen.
- Die Ausführung von `operator++` auf einen Reverseiterator entspricht der Ausführung des `operator--` auf einen Iterator.
- Reversible Container stellen die Typdefinitionen `reverse_iterator` und `const_reverse_iterator` bereit.
- Reversible Container besitzen die Methoden `rbegin` und `rend`, welche Reversiteratoren liefern.
- Der kommende C++-Standard wird voraussichtlich für diese Container die Methoden `crbegin` und `crend`, welche `const_reverse_iterator` zurückliefern, beinhalten.

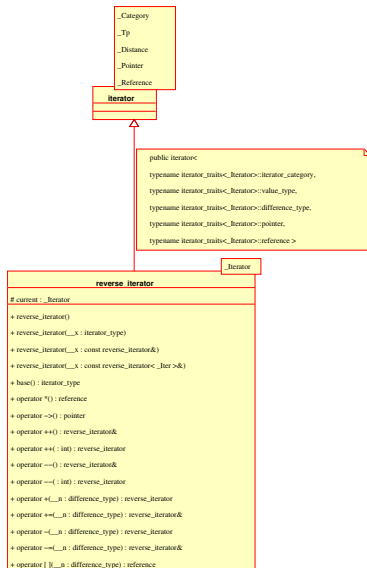
- Ein Reversiterator ist ein Iteratoradapter.
- Bidirectional- und Random-Access-Iteratoren können vorwärts als auch rückwärts durchlaufen werden – $>$ für diese steht dieser Iteratoradapter bereit.
- Mit einem Reversiterator werden die Elemente in umgekehrter Richtung durchlaufen.
- Die Ausführung von `operator++` auf einen Reverseiterator entspricht der Ausführung des `operator--` auf einen Iterator.
- Reversible Container stellen die Typdefinitionen `reverse_iterator` und `const_reverse_iterator` bereit.
- Reversible Container besitzen die Methoden `rbegin` und `rend`, welche Reversiteratoren liefern.
- Der kommende C++-Standard wird voraussichtlich für diese Container die Methoden `crbegin` und `crend`, welche `const_reverse_iterator` zurückliefern, beinhalten.

- Ein Reversiterator ist ein Iteratoradapter.
- Bidirectional- und Random-Access-Iteratoren können vorwärts als auch rückwärts durchlaufen werden – $\>$ für diese steht dieser Iteratoradapter bereit.
- Mit einem Reversiterator werden die Elemente in umgekehrter Richtung durchlaufen.
- Die Ausführung von `operator++` auf einen Reverseiterator entspricht der Ausführung des `operator--` auf einen Iterator.
- Reversible Container stellen die Typdefinitionen `reverse_iterator` und `const_reverse_iterator` bereit.
- Reversible Container besitzen die Methoden `rbegin` und `rend`, welche Reversiteratoren liefern.
- Der kommende C++-Standard wird voraussichtlich für diese Container die Methoden `crbegin` und `crend`, welche `const_reverse_iterator` zurückliefern, beinhalten.

- Ein Reversiterator ist ein Iteratoradapter.
- Bidirectional- und Random-Access-Iteratoren können vorwärts als auch rückwärts durchlaufen werden – $>$ für diese steht dieser Iteratoradapter bereit.
- Mit einem Reversiterator werden die Elemente in umgekehrter Richtung durchlaufen.
- Die Ausführung von `operator++` auf einen Reverseiterator entspricht der Ausführung des `operator--` auf einen Iterator.
- Reversible Container stellen die Typdefinitionen `reverse_iterator` und `const_reverse_iterator` bereit.
- Reversible Container besitzen die Methoden `rbegin` und `rend`, welche Reversiteratoren liefern.
- Der kommende C++-Standard wird voraussichtlich für diese Container die Methoden `crbegin` und `crend`, welche `const_reverse_iterator` zurückliefern, beinhalten.

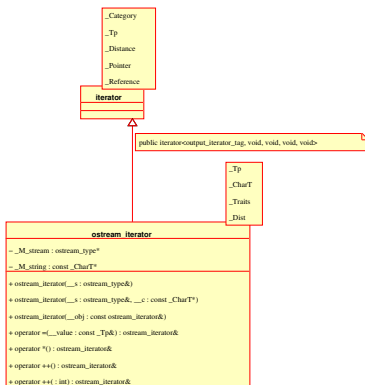
- Ein Reversiterator ist ein Iteratoradapter.
- Bidirectional- und Random-Access-Iteratoren können vorwärts als auch rückwärts durchlaufen werden – $\>$ für diese steht dieser Iteratoradapter bereit.
- Mit einem Reversiterator werden die Elemente in umgekehrter Richtung durchlaufen.
- Die Ausführung von `operator++` auf einen Reverseiterator entspricht der Ausführung des `operator--` auf einen Iterator.
- Reversible Container stellen die Typdefinitionen `reverse_iterator` und `const_reverse_iterator` bereit.
- Reversible Container besitzen die Methoden `rbegin` und `rend`, welche Reversiteratoren liefern.
- Der kommende C++-Standard wird voraussichtlich für diese Container die Methoden `crbegin` und `crend`, welche `const_reverse_iterator` zurückliefern, beinhalten.

Reverseiterator (UML)



Beispiel ostream_iterator

- Ein ostream_iterator gehört zu der Kategorie der Output-Iteratoren.
- Er gibt auf einen ostream Werte per operator<< aus.



Beispiel ostream_iterator

```
template<typename _Tp, typename _CharT = char,
        typename _Traits = char_traits<_CharT> >
class ostream_iterator
: public iterator<output_iterator_tag, void, void, void, void>
{
public:
    typedef _CharT char_type;
    typedef _Traits traits_type;
    typedef basic_ostream<_CharT, _Traits> ostream_type;
private:
    ostream_type* _M_stream;
    const _CharT* _M_string;
public:
    ostream_iterator(ostream_type& __s) : _M_stream(&__s), _M_string(0) {}

    ostream_iterator(ostream_type& __s, const _CharT* __c)
    : _M_stream(&__s), _M_string(__c) { }

    ostream_iterator(const ostream_iterator& __obj)
    : _M_stream(__obj._M_stream), _M_string(__obj._M_string) { }

    ostream_iterator& operator=(const _Tp& __value){
        *_M_stream << __value;
        if (_M_string) *_M_stream << _M_string;
        return *this;
    }

    ostream_iterator& operator*()           { return *this; }
    ostream_iterator& operator++()         { return *this; }
    ostream_iterator& operator++(int)     { return *this; }
};
```

- Insert-Iteratoren fügen an eine bestimmte Stelle zugewiesene Werte ein.
- Überblick:

Name	benutzt	fügt ein
back_insert_iterator	push_back	am Ende
front_insert_iterator	push_front	am Anfang
insert_iterator	insert	an beliebiger Stelle

Durch Einfügen neuer Elemente können bestehende Iteratoren ihre Gültigkeit verlieren.

- Insert-Iteratoren fügen an eine bestimmte Stelle zugewiesene Werte ein.
- Überblick:

Name	benutzt	fügt ein
back_insert_iterator	push_back	am Ende
front_insert_iterator	push_front	am Anfang
insert_iterator	insert	an beliebiger Stelle

Durch Einfügen neuer Elemente können bestehende Iteratoren ihre Gültigkeit verlieren.

- Insert-Iteratoren fügen an eine bestimmte Stelle zugewiesene Werte ein.
- Überblick:

Name	benutzt	fügt ein
back_insert_iterator	push_back	am Ende
front_insert_iterator	push_front	am Anfang
insert_iterator	insert	an beliebiger Stelle

Durch Einfügen neuer Elemente können bestehende Iteratoren ihre Gültigkeit verlieren.

- 1 Iteratoren
- 2 C++ Speicherverwaltung
- 3 Aufgabe

Der Ausdruck `MyClass p = new MyClass()` arbeitet zwei Schritte ab:

- 1 Stelle ausreichend dynamischen Speicher für ein Objekt der Klasse `MyClass` zur Verfügung.
- 2 Rufe den Konstruktor auf (die Instanz richtet sich auf diesem Speicherbereich ein).

Um die durch `new` und den darauf folgenden Konstruktoraufruf angeforderten Ressourcen wieder frei zu geben wird `delete p` aufgerufen. Dieser arbeitet folgende zwei Schritte ab:

- 1 Rufe den Destruktor für das Objekt auf.
- 2 Gebe den angeforderten Speicher zurück.

Der Ausdruck `MyClass p = new MyClass()` arbeitet zwei Schritte ab:

- 1 Stelle ausreichend dynamischen Speicher für ein Objekt der Klasse `MyClass` zur Verfügung.
- 2 Rufe den Konstruktor auf (die Instanz richtet sich auf diesem Speicherbereich ein).

Um die durch `new` und den darauf folgenden Konstruktoraufruf angeforderten Ressourcen wieder frei zu geben wird `delete p` aufgerufen. Dieser arbeitet folgende zwei Schritte ab:

- 1 Rufe den Destruktor für das Objekt auf.
- 2 Gebe den angeforderten Speicher zurück.

new und delete[]

Wird ein Array von Objekten dynamisch angelegt, erfolgt dies mit

```
string* pString = new string[50].
```

Die Arbeitsweise ist:

- 1 Stelle ausreichend großen Block dynamischen Speicher für 50 Objekte der Klasse `string` zur Verfügung.
- 2 Rufe den Konstruktor für jedes der 50 Objekte auf.

`delete` muß mitgeteilt werden ob ein Zeiger auf ein einzelnes Objekt oder auf ein Array zeigt. Um das Array wieder freizugeben, muß `delete[] pString;` aufgerufen werden.

Die Arbeitsweise ist:

- 1 Rufe den Destruktor für jedes Objekt (50) im Array auf.
- 2 Gebe den angeforderten Speicher zurück.

Das Verhalten von `delete` ohne `[]` auf ein Array ist undefiniert.

Das Verhalten von `delete[]` auf ein einzelnes Objekt ist undefiniert.

new und delete[]

Wird ein Array von Objekten dynamisch angelegt, erfolgt dies mit

```
string* pString = new string[50].
```

Die Arbeitsweise ist:

- 1 Stelle ausreichend großen Block dynamischen Speicher für 50 Objekte der Klasse `string` zur Verfügung.
- 2 Rufe den Konstruktor für jedes der 50 Objekte auf.

`delete` muß mitgeteilt werden ob ein Zeiger auf ein einzelnes Objekt oder auf ein Array zeigt. Um das Array wieder freizugeben, muß `delete[] pString;` aufgerufen werden.

Die Arbeitsweise ist:

- 1 Rufe den Destruktor für jedes Objekt (50) im Array auf.
- 2 Gebe den angeforderten Speicher zurück.

Das Verhalten von `delete` ohne `[]` auf ein Array ist undefiniert.

Das Verhalten von `delete[]` auf ein einzelnes Objekt ist undefiniert.

new und delete[]

Wird ein Array von Objekten dynamisch angelegt, erfolgt dies mit

```
string* pString = new string[50].
```

Die Arbeitsweise ist:

- 1 Stelle ausreichend großen Block dynamischen Speicher für 50 Objekte der Klasse `string` zur Verfügung.
- 2 Rufe den Konstruktor für jedes der 50 Objekte auf.

`delete` muß mitgeteilt werden ob ein Zeiger auf ein einzelnes Objekt oder auf ein Array zeigt. Um das Array wieder freizugeben, muß `delete[] pString;` aufgerufen werden.

Die Arbeitsweise ist:

- 1 Rufe den Destruktor für jedes Objekt (50) im Array auf.
- 2 Gebe den angeforderten Speicher zurück.

Das Verhalten von `delete` ohne `[]` auf ein Array ist undefiniert.

Das Verhalten von `delete[]` auf ein einzelnes Objekt ist undefiniert.

new und delete[]

Wird ein Array von Objekten dynamisch angelegt, erfolgt dies mit

```
string* pString = new string[50].
```

Die Arbeitsweise ist:

- 1 Stelle ausreichend großen Block dynamischen Speicher für 50 Objekte der Klasse `string` zur Verfügung.
- 2 Rufe den Konstruktor für jedes der 50 Objekte auf.

`delete` muß mitgeteilt werden ob ein Zeiger auf ein einzelnes Objekt oder auf ein Array zeigt. Um das Array wieder freizugeben, muß `delete[] pString;` aufgerufen werden.

Die Arbeitsweise ist:

- 1 Rufe den Destruktor für jedes Objekt (50) im Array auf.
- 2 Gebe den angeforderten Speicher zurück.

Das Verhalten von `delete` ohne `[]` auf ein Array ist undefiniert.

Das Verhalten von `delete[]` auf ein einzelnes Objekt ist undefiniert.

Wenn die Situation eintritt, dass am System nicht die angeforderte Menge Speicher zur Verfügung steht, kann sich `new` wie folgt verhalten:

- standardmäßig wird eine Ausnahme (`std::bad_alloc`) ausgeworfen,
- alternativ ist es möglich `new` keine Ausnahme werfen zu lassen, sondern einfach einen `NULL`-Zeiger zu liefern.

```
char *p = new (nothrow) char [size]; //array nothrow new
string *pString = new (nothrow) string; //scalar nothrow new
```

- das Verhalten kann selbst definiert werden, indem ein `new_handler` installiert wird

Wenn die Situation eintritt, dass am System nicht die angeforderte Menge Speicher zur Verfügung steht, kann sich `new` wie folgt verhalten:

- standardmäßig wird eine Ausnahme (`std::bad_alloc`) ausgeworfen,
- alternativ ist es möglich `new` keine Ausnahme werfen zu lassen, sondern einfach einen `NULL`-Zeiger zu liefern.

```
char *p = new (nothrow) char [size]; //array nothrow new
string *pString = new (nothrow) string; //scalar nothrow new
```

- das Verhalten kann selbst definiert werden, indem ein `new_handler` installiert wird

Wenn die Situation eintritt, dass am System nicht die angeforderte Menge Speicher zur Verfügung steht, kann sich `new` wie folgt verhalten:

- standardmäßig wird eine Ausnahme (`std::bad_alloc`) ausgeworfen,
- alternativ ist es möglich `new` keine Ausnahme werfen zu lassen, sondern einfach einen `NULL`-Zeiger zu liefern.

```
char *p = new (nothrow) char [size]; //array nothrow new
string *pString = new (nothrow) string; //scalar nothrow new
```

- das Verhalten kann selbst definiert werden, indem ein `new_handler` installiert wird

Wenn die Situation eintritt, dass am System nicht die angeforderte Menge Speicher zur Verfügung steht, kann sich `new` wie folgt verhalten:

- standardmäßig wird eine Ausnahme (`std::bad_alloc`) ausgeworfen,
- alternativ ist es möglich `new` keine Ausnahme werfen zu lassen, sondern einfach einen `NULL`-Zeiger zu liefern.

```
char *p = new (nothrow) char [size]; //array nothrow new
string *pString = new (nothrow) string; //scalar nothrow new
```

- das Verhalten kann selbst definiert werden, indem ein `new_handler` installiert wird

- Um einen `new_handler` installieren zu können, wird `new.h` inkludiert.
- Ein `new_handler` ist eine Funktion, die keine Parameter übernimmt und kein Ergebnis liefert (`typedef void (*new_handler)();`).
- Ein `new_handler` wird durch den Funktionsaufruf `set_new_handler` installiert. (`new_handler set_new_handler(new_handler p) throw()`)
- Wenn `new` eine Speicheranforderung nicht befriedigen kann, ruft es den installierte `new_handler`-Funktion solange auf, bis genügend Speicher gefunden wird.

Daraus ergibt sich das Verhalten des `new_handler`:

- Mehr Speicher zur Verfügung stellen.
- Einen anderen `new_handler` installieren.
- Deinstallieren (Nullzeiger an `set_new_handler` übergeben, damit ist Standardverhalten wieder hergestellt).
- Eine Ausnahme vom Typ `std::bad_alloc` (oder einem abgeleiteten Typ) auswerfen.
- Nicht zurückkehren.

- Um einen `new_handler` installieren zu können, wird `new.h` inkludiert.
- Ein `new_handler` ist eine Funktion, die keine Parameter übernimmt und kein Ergebnis liefert (`typedef void (*new_handler)();`).
- Ein `new_handler` wird durch den Funktionsaufruf `set_new_handler` installiert. (`new_handler set_new_handler(new_handler p) throw()`)
- Wenn `new` eine Speicheranforderung nicht befriedigen kann, ruft es den installierte `new_handler`-Funktion solange auf, bis genügend Speicher gefunden wird.

Daraus ergibt sich das Verhalten des `new_handler`:

- Mehr Speicher zur Verfügung stellen.
- Einen anderen `new_handler` installieren.
- Deinstallieren (Nullzeiger an `set_new_handler` übergeben, damit ist Standardverhalten wieder hergestellt).
- Eine Ausnahme vom Typ `std::bad_alloc` (oder einem abgeleiteten Typ) auswerfen.
- Nicht zurückkehren.

- Um einen `new_handler` installieren zu können, wird `new.h` inkludiert.
- Ein `new_handler` ist eine Funktion, die keine Parameter übernimmt und kein Ergebnis liefert (`typedef void (*new_handler)();`).
- Ein `new_handler` wird durch den Funktionsaufruf `set_new_handler` installiert. (`new_handler set_new_handler(new_handler p) throw()`)
- Wenn `new` eine Speicheranforderung nicht befriedigen kann, ruft es den installierte `new_handler`-Funktion solange auf, bis genügend Speicher gefunden wird.

Daraus ergibt sich das Verhalten des `new_handler`:

- Mehr Speicher zur Verfügung stellen.
- Einen anderen `new_handler` installieren.
- Deinstallieren (Nullzeiger an `set_new_handler` übergeben, damit ist Standardverhalten wieder hergestellt).
- Eine Ausnahme vom Typ `std::bad_alloc` (oder einem abgeleiteten Typ) auswerfen.
- Nicht zurückkehren.

- Um einen `new_handler` installieren zu können, wird `new.h` inkludiert.
- Ein `new_handler` ist eine Funktion, die keine Parameter übernimmt und kein Ergebnis liefert (`typedef void (*new_handler)();`).
- Ein `new_handler` wird durch den Funktionsaufruf `set_new_handler` installiert. (`new_handler set_new_handler(new_handler p) throw()`)
- Wenn `new` eine Speicheranforderung nicht befriedigen kann, ruft es den installierte `new_handler`-Funktion solange auf, bis genügend Speicher gefunden wird.

Daraus ergibt sich das Verhalten des `new_handler`:

- Mehr Speicher zur Verfügung stellen.
- Einen anderen `new_handler` installieren.
- Deinstallieren (Nullzeiger an `set_new_handler` übergeben, damit ist Standardverhalten wieder hergestellt).
- Eine Ausnahme vom Typ `std::bad_alloc` (oder einem abgeleiteten Typ) auswerfen.
- Nicht zurückkehren.

- Um einen `new_handler` installieren zu können, wird `new.h` inkludiert.
- Ein `new_handler` ist eine Funktion, die keine Parameter übernimmt und kein Ergebnis liefert (`typedef void (*new_handler)();`).
- Ein `new_handler` wird durch den Funktionsaufruf `set_new_handler` installiert. (`new_handler set_new_handler(new_handler p) throw();`)
- Wenn `new` eine Speicheranforderung nicht befriedigen kann, ruft es den installierte `new_handler`-Funktion solange auf, bis genügend Speicher gefunden wird.

Daraus ergibt sich das Verhalten des `new_handler`:

- Mehr Speicher zur Verfügung stellen.
- Einen anderen `new_handler` installieren.
- Deinstallieren (Nullzeiger an `set_new_handler` übergeben, damit ist Standardverhalten wieder hergestellt).
- Eine Ausnahme vom Typ `std::bad_alloc` (oder einem abgeleiteten Typ) auswerfen.
- Nicht zurückkehren.

- Um einen `new_handler` installieren zu können, wird `new.h` inkludiert.
- Ein `new_handler` ist eine Funktion, die keine Parameter übernimmt und kein Ergebnis liefert (`typedef void (*new_handler)();`).
- Ein `new_handler` wird durch den Funktionsaufruf `set_new_handler` installiert. (`new_handler set_new_handler(new_handler p) throw();`)
- Wenn `new` eine Speicheranforderung nicht befriedigen kann, ruft es den installierte `new_handler`-Funktion solange auf, bis genügend Speicher gefunden wird.

Daraus ergibt sich das Verhalten des `new_handler`:

- Mehr Speicher zur Verfügung stellen.
- Einen anderen `new_handler` installieren.
- Deinstallieren (Nullzeiger an `set_new_handler` übergeben, damit ist Standardverhalten wieder hergestellt).
- Eine Ausnahme vom Typ `std::bad_alloc` (oder einem abgeleiteten Typ) auswerfen.
- Nicht zurückkehren.

- Um einen `new_handler` installieren zu können, wird `new.h` inkludiert.
- Ein `new_handler` ist eine Funktion, die keine Parameter übernimmt und kein Ergebnis liefert (`typedef void (*new_handler)();`).
- Ein `new_handler` wird durch den Funktionsaufruf `set_new_handler` installiert. (`new_handler set_new_handler(new_handler p) throw()`)
- Wenn `new` eine Speicheranforderung nicht befriedigen kann, ruft es den installierte `new_handler`-Funktion solange auf, bis genügend Speicher gefunden wird.

Daraus ergibt sich das Verhalten des `new_handler`:

- Mehr Speicher zur Verfügung stellen.
- Einen anderen `new_handler` installieren.
- Deinstallieren (Nullzeiger an `set_new_handler` übergeben, damit ist Standardverhalten wieder hergestellt).
- Eine Ausnahme vom Typ `std::bad_alloc` (oder einem abgeleiteten Typ) auswerfen.
- Nicht zurückkehren.

- Um einen `new_handler` installieren zu können, wird `new.h` inkludiert.
- Ein `new_handler` ist eine Funktion, die keine Parameter übernimmt und kein Ergebnis liefert (`typedef void (*new_handler)();`).
- Ein `new_handler` wird durch den Funktionsaufruf `set_new_handler` installiert. (`new_handler set_new_handler(new_handler p) throw()`)
- Wenn `new` eine Speicheranforderung nicht befriedigen kann, ruft es den installierte `new_handler`-Funktion solange auf, bis genügend Speicher gefunden wird.

Daraus ergibt sich das Verhalten des `new_handler`:

- Mehr Speicher zur Verfügung stellen.
- Einen anderen `new_handler` installieren.
- Deinstallieren (Nullzeiger an `set_new_handler` übergeben, damit ist Standardverhalten wieder hergestellt).
- Eine Ausnahme vom Typ `std::bad_alloc` (oder einem abgeleiteten Typ) auswerfen.
- Nicht zurückkehren.

- Um einen `new_handler` installieren zu können, wird `new.h` inkludiert.
- Ein `new_handler` ist eine Funktion, die keine Parameter übernimmt und kein Ergebnis liefert (`typedef void (*new_handler)();`).
- Ein `new_handler` wird durch den Funktionsaufruf `set_new_handler` installiert. (`new_handler set_new_handler(new_handler p) throw()`)
- Wenn `new` eine Speicheranforderung nicht befriedigen kann, ruft es den installierte `new_handler`-Funktion solange auf, bis genügend Speicher gefunden wird.

Daraus ergibt sich das Verhalten des `new_handler`:

- Mehr Speicher zur Verfügung stellen.
- Einen anderen `new_handler` installieren.
- Deinstallieren (Nullzeiger an `set_new_handler` übergeben, damit ist Standardverhalten wieder hergestellt).
- Eine Ausnahme vom Typ `std::bad_alloc` (oder einem abgeleiteten Typ) auswerfen.
- Nicht zurückkehren.

- `new` allokiert Speicher am Heap,
- es kann sinnvoll sein, pre-allocierten Speicher benutzen zu wollen
– `>placement new`
- Beim `placement new` wird kein Speicher angefordert, sondern von Benutzer ein Zeiger auf den Anfang des Speicherbereiches angegeben, an dem das Objekt konstruiert werden soll. Bsp:

```
char *buf = new char[1000]; //pre-allocated buffer
string *p = new (buf) string("hi"); //placement new
string *q = new string("hi"); //ordinary heap allocation
```

- `placement new` ist beispielsweise für die Realisierung von Memory-Pools oder für die Implementierung eines garbage collector nützlich.

- `new` allokiert Speicher am Heap,
- es kann sinnvoll sein, pre-allocierten Speicher benutzen zu wollen
– `>placement new`
- Beim `placement new` wird kein Speicher angefordert, sondern von Benutzer ein Zeiger auf den Anfang des Speicherbereiches angegeben, an dem das Objekt konstruiert werden soll. Bsp:

```
char *buf = new char[1000]; //pre-allocated buffer
string *p = new (buf) string("hi"); //placement new
string *q = new string("hi"); //ordinary heap allocation
```

- `placement new` ist beispielsweise für die Realisierung von Memory-Pools oder für die Implementierung eines garbage collector nützlich.

- `new` allokiert Speicher am Heap,
- es kann sinnvoll sein, pre-allocierten Speicher benutzen zu wollen
– `>placement new`
- Beim `placement new` wird kein Speicher angefordert, sondern von Benutzer ein Zeiger auf den Anfang des Speicherbereiches angegeben, an dem das Objekt konstruiert werden soll. Bsp:

```
char *buf = new char[1000]; //pre-allocated buffer
string *p = new (buf) string("hi"); //placement new
string *q = new string("hi"); //ordinary heap allocation
```

- `placement new` ist beispielsweise für die Realisierung von Memory-Pools oder für die Implementierung eines garbage collector nützlich.

Das Zeigerkonzept von C/C++ kann durch den Benutzer fehlerhaft angewandt werden:

- 1 Zeigerarithmetik: Zeiger verweist auf außerhalb liegenden Speicher.
- 2 memory leak: der letzte, auf ein dynamisch angelegtes Objekt verweisende Zeiger gibt das Objekt nicht wieder frei.
- 3 ungültig: ein Zeiger wird ungültig, wenn das Objekt, auf das verwiesen wird, freigegeben wird.

`smart pointer` sind Datentypen, die sich wie Zeiger verhalten können (`operator*`, `operator->`, `operator++`, ...) aber zusätzliche Logik mitbringen. Bsp: `std::auto_ptr`

Das Zeigerkonzept von C/C++ kann durch den Benutzer fehlerhaft angewandt werden:

- 1 Zeigerarithmetik: Zeiger verweist auf außerhalb liegenden Speicher.
- 2 memory leak: der letzte, auf ein dynamisch angelegtes Objekt verweisende Zeiger gibt das Objekt nicht wieder frei.
- 3 ungültig: ein Zeiger wird ungültig, wenn das Objekt, auf das verwiesen wird, freigegeben wird.

smart pointer sind Datentypen, die sich wie Zeiger verhalten können (operator*, operator->, operator++,...) aber zusätzliche Logik mitbringen. Bsp: `std::auto_ptr`

Das Zeigerkonzept von C/C++ kann durch den Benutzer fehlerhaft angewandt werden:

- 1 Zeigerarithmetik: Zeiger verweist auf außerhalb liegenden Speicher.
- 2 memory leak: der letzte, auf ein dynamisch angelegtes Objekt verweisende Zeiger gibt das Objekt nicht wieder frei.
- 3 ungültig: ein Zeiger wird ungültig, wenn das Objekt, auf das verwiesen wird, freigegeben wird.

`smart pointer` sind Datentypen, die sich wie Zeiger verhalten können (operator*, operator->, operator++,...) aber zusätzliche Logik mitbringen. Bsp: `std::auto_ptr`

Das Zeigerkonzept von C/C++ kann durch den Benutzer fehlerhaft angewandt werden:

- 1 Zeigerarithmetik: Zeiger verweist auf außerhalb liegenden Speicher.
- 2 memory leak: der letzte, auf ein dynamisch angelegtes Objekt verweisende Zeiger gibt das Objekt nicht wieder frei.
- 3 ungültig: ein Zeiger wird ungültig, wenn das Objekt, auf das verwiesen wird, freigegeben wird.

smart pointer sind Datentypen, die sich wie Zeiger verhalten können (operator*, operator->, operator++,...) aber zusätzliche Logik mitbringen. Bsp: `std::auto_ptr`

Das Zeigerkonzept von C/C++ kann durch den Benutzer fehlerhaft angewandt werden:

- 1 Zeigerarithmetik: Zeiger verweist auf außerhalb liegenden Speicher.
- 2 memory leak: der letzte, auf ein dynamisch angelegtes Objekt verweisende Zeiger gibt das Objekt nicht wieder frei.
- 3 ungültig: ein Zeiger wird ungültig, wenn das Objekt, auf das verwiesen wird, freigegeben wird.

`smart pointer` sind Datentypen, die sich wie Zeiger verhalten können (operator*, operator->, operator++,...) aber zusätzliche Logik mitbringen. Bsp: `std::auto_ptr`

- 1 Iteratoren
- 2 C++ Speicherverwaltung
- 3 Aufgabe**

- Schauen Sie sich die Templateklasse `XSNamedMap` des `xerces-c` Parsers an (<http://xml.apache.org/xerces-c/apiDocs/classXSNamedMap.html>).
- Implementieren Sie eine Template-Iterator-Klasse für `XSNamedMap`, die mindestens der Kategorie `Forward-Iterator` genügt.
- Erweitern Sie die Templateklasse `XSNamedMap` um die Typdefinition `iterator` und die Methoden `begin()` und `end()`
- Kopieren Sie sich das Beispiel `SCMPrint`, implementieren Sie die Funktion `void processElements(XSNamedMap<XSObject> *xsElements) ()` mit Hilfe ihrer Iterator-Implementation neu und übersetzen sie das Programm.

Hinweise:

- xerces-c ist im Linux-Pool installiert (ppc201 ..215)
- die Dokumentation ist lokal in dem Verzeichnis `/usr/share/doc/libxerces26-doc/html/index.html` installiert
- alle Beispiele (auch SCMPrint) liegen unter `/usr/share/doc/libxerces26-dev/examples/samples/`
- eine Beschreibung wie die Beispiele zu benutzen sind finden Sie unter <http://xml.apache.org/xerces-c/samples.html>