Advanced Software Development

Container und Iteratoren

Übersicht

- Boost Iteratoren
- Boost Container
- Implementierung von eigenen Containern und Iteratoren

Boost Iteratoren (1)

- counting_iterator: Iteration über einen Zahlenbereich

```
// Vektor mit Zahlen von 0 bis 9 befüllen
copy(counting_iterator<int>(0), counting_iterator<int>(10),
     back_inserter(v));
// Alternativ:
copy(make_counting_iterator(0), make_counting_iterator(10),
     back_inserter(v));
```

- indirect_iterator: Implizite Dereferenzierung

```
vector<int*> v;
…
copy(make_indirect_iterator(v.begin()), make_indirect_iterator(v.end()),
     ostream_iterator<int>(cout, " "));
```

Boost Iteratoren (2)

- transform_iterator: Anwendung eines Funktionsobjekts

```
vector<MyObject*> objects;
int sum = accumulate(make_transform_iterator(objects.begin(),
                       mem_fn(&MyObject::getValue)),
                     make_transform_iterator(objects.end(),
                       mem_fn(&MyObject::getValue)), 0);
```

- filter_iterator: Iteration über alle Elemente, die eine Bedingung erfüllen

```
vector<int> v;
// Ausgabe aller Werte größer als 0
copy(make_filter_iterator(bind2nd(greater<int>(), 0), v.begin(), v.end()),
     make_filter_iterator(bind2nd(greater<int>(), 0), v.end(), v.end()),
     ostream_iterator<int>(cout, " "));
```

Boost Iteratoren (3)

- permutation_iterator: Iteration gemäß Indizes in einem anderen Container

```
vector<int> v;
vector<size_t> indices;
…
// Ausgabe der Elemente von v entsprechend den Indizes in indices
copy(make_permutation_iterator(v.begin(), indices.begin()),
     make_permutation_iterator(v.begin(), indices.end()),
     ostream_iterator<int>(cout, " "));
```

Boost Iteratoren (4)

- zip_iterator: Iteration über mehrere Container gleichzeitig
- shared_container_iterator: Iterator über Shared Container (Container in shared_ptr)
- function_output_iterator: Output-Iterator, der Funktion aufruft

Boost Iterator Ranges

- durch zwei Iteratoren definierter Bereich
- kann für Iteration verwendet werden (begin/end Methoden)
- Beispiel:

```
vector<int> v;
...
auto range = boost::make_iterator_range(v.begin() + 1, v.end());
for (int x : range) {
  ...
}
```

Boost Iterator Ranges (2)

- Weitere Methoden (u.a.)
  - empty(), size()
  - Indexoperator
  - Vergleichsoperatoren: elementweiser Vergleich der beiden Bereiche
  - Ausgabeoperator <<: Ausgabe aller Elemente des Bereichs (ohne Trennzeichen)

Boost Range Adaptors

- zur Transformation von Ranges
- Anwendung als Funktion oder über Operator |
- Beispiel:

```
using namespace boost::adaptors;
vector<int> v;
...
for (int x : reverse(v)) { // alternativ: for (int x : v | reversed)
  ...
}
```

Boost Range Adaptors (2)

- filter(range, pred), filtered(pred): Elemente, die Bedingung erfüllen
- indirect(range), indirected: dereferenzierte Elemente
- reverse(range), reversed
- slice(range, n, m), sliced(n, m): Indexbereich [n, m[
- stride(range, n), strided(n): Schrittweite n

Boost Container

- boost::container::static_vector
- boost::container::flat_map/flat_set
- boost::multi_array
- boost::circular_buffer (Ringpuffer)
- boost::dynamic_bitset (dynamische Variante von std::bitset)
- boost::bimap (bidirektionale Map)
- Lock Free Container (boost::lockfree::…)

boost::container::static_vector

- Vektor mit konstanter maximaler Kapazität
- Deklaration z.B.: `static_vector<int, 10> v;`
- Bietet die gleichen Methoden wie std::vector
- Exception bad_alloc bei Versuch, zu viele Elemente einzufügen

boost::container::flat_map/flat_set

- Verwendung wie std::map und std::set
- Interne Darstellung als sortierter Vektor
- Elementsuche in O(log n)
- Einfügen in O(n)
- Weniger Overhead, schnellere Iteration

boost::multi_array

- Dynamisches mehrdimensionales Array
- Beispiel:

```
size_t m = 3;
size_t n = 3;
multi_array<int, 2> matrix(extents[m][n]);
for (size_t i = 0; i < matrix.shape()[0]; i++) {
  for (size_t j = 0; j < matrix.shape()[1]; j++) {
    matrix[i][j] = …;
  }
}
for (auto &line : matrix) {
  for (auto x : line) {
    ...
  }
}
```

STL-konforme Container (1)

- Allocator als Template-Parameter (standardmäßig std::allocator)
- Konstruktoren
  - Default Constructor: leerer Container
  - Copy Constructor und Move Constructor
  - Alle Konstruktoren (außer Copy Constructor) nehmen einen Allocator als optionalen Parameter
- Destruktor

STL-konforme Container (2)

- Typdefinitionen
  - value_type
  - reference, const_reference
  - iterator, const_iterator
  - difference_type, size_type
- Operatoren
  - Zuweisungsoperator (Copy und Move)
  - Vergleichsoperatoren: ==, !=, <, >, <=, >=

STL-konforme Container (3)

- Methoden (konstante Komplexität)
  - begin() und end(): iterator
  - begin() const und end() const: const_iterator
  - cbegin() und cend(): const_iterator
  - swap(x)
  - size(), max_size()
  - empty()
  - get_allocator()

Zusatzanforderungen (1)

- Reversible Container
  - reverse_iterator, const_reverse_iterator
  - Methoden rbegin() und rend()
- Sequence
  - Konstruktor mit Wert und Anzahl
  - Konstruktor mit 2 Iteratoren
  - Methode insert in 3 Varianten (vgl. std::vector)
  - Methode erase in 2 Varianten (vgl. std::vector)
  - Methode clear()

Zusatzanforderungen (2)

- Optionale Sequence-Operationen (nur falls in konstanter Laufzeit möglich)
  - Methoden front() bzw. back()
  - Methoden push_front() bzw. push_back()
  - Methoden pop_front() bzw. pop_back()
  - Indexoperator (const und nicht const)
  - Methode at (const und nicht const)
- Weitere Variante: Assoziative Container

Iterator-Kategorien

- Input: nur Lesen, nur ein einziger Durchlauf
- Output: nur Schreiben, nur ein einziger Durchlauf
- Forward: Lesen und Schreiben, mehrere sequentielle Durchläufe möglich
- Bidirectional: Sequentieller Durchlauf in beide Richtungen möglich
- Random Access: Direktzugriff möglich

Iterator-Anforderungen (1)

- Allgemeine Anforderungen
  - Konstruktor
  - Copy Constructor
  - Zuweisungsoperator
  - Dereferenzierungsoperator *
  - Operator ++ (Präfix und Postfix)
- Zusätzlich für Input und Forward Iterator
  - Vergleichsoperatoren == und !=
  - Operator ->

Iterator-Anforderungen (2)

- Zusätzlich für Bidirectional Iterator
  - Operator -- (Präfix und Postfix)
- Zusätzlich für Random Access Iterator
  - Operatoren += und -=
  - Operator +
  - Operator - (auch für zwei Iteratoren)
  - Indexoperator []
  - Vergleichsoperatoren <, >, <=, >=

Iterator Traits (1)

- Typdefinitionen in std::iterator_traits<TIter>
  - value_type
  - difference_type
  - pointer
  - reference
  - iterator_category: input_iterator_tag, output_iterator_tag, forward_iterator_tag, bidirectional_iterator_tag oder random_access_iterator_tag

Iterator Traits (2)

- Implementierungsvarianten
  - Deklaration aller benötigten Typen in Iterator-Klasse
  - Ableitung von Template-Klasse std::iterator:

```
template<typename Category, typename T, typename Distance=ptrdiff_t,
         typename Pointer=T*, typename Reference=T&>
struct iterator {
  typedef T         value_type;
  typedef Distance  difference_type;
  typedef Pointer   pointer;
  typedef Reference reference;
  typedef Category  iterator_category;
};
```

Beispiel: dynamic_array (1)

- Implementierung als Reversible Container mit Random Access Iteratoren
- Unterstützte Sequence-Operationen:
  - Konstruktoren
  - Methode clear()
  - Methoden front() und back()
  - Indexoperator, Methode at()
- Keine vollständig konforme Sequence

Beispiel: dynamic_array (2)

```
template <typename T, typename Alloc = std::allocator<T> >
class dynamic_array {
public:
  typedef Alloc allocator_type;
  typedef T value_type;
  typedef typename Alloc::pointer pointer;
  typedef typename Alloc::const_pointer const_pointer;
  typedef typename Alloc::reference reference;
  typedef typename Alloc::const_reference const_reference;
  typedef typename Alloc::size_type size_type;
  typedef typename Alloc::difference_type difference_type;
  typedef T *iterator;
  typedef T const *const_iterator;
  typedef std::reverse_iterator<iterator> reverse_iterator;
  typedef std::reverse_iterator<const_iterator> const_reverse_iterator;
  …
private:
  Alloc mAlloc;
  T *mElems;
  size_type mSize;
};
```

Beispiel: dynamic_array (3)

```
explicit dynamic_array(Alloc const &alloc = Alloc())
  : mAlloc(alloc), mElems(nullptr), mSize(0) {}

dynamic_array(dynamic_array<T, Alloc> const &other)
: mSize(other.mSize), mAlloc(other.mAlloc) {
  if (mSize > 0) {
    mElems = mAlloc.allocate(mSize);
    std::uninitialized_copy(other.mElems, other.mElems + mSize, mElems);
  }
  else {
    mElems = nullptr;
  }
}

dynamic_array(dynamic_array<T, Alloc> &&other)
 : mSize(other.mSize), mElems(other.mElems), mAlloc(other.mAlloc) {
  other.mSize = 0;
  other.mElems = nullptr;
}
```

Beispiel: dynamic_array (4)

```
explicit dynamic_array(size_type size, T const &val = T(),
                       Alloc const &alloc = Alloc()) : mAlloc(alloc) {
  initialize(size, val);
}

template <typename TIter>
dynamic_array(TIter begin, TIter end, Alloc const &alloc = Alloc()) {
  initialize(begin, end, std::is_integral<TIter>());
}

void initialize(size_type size, T const &val, std::true_type = std::true_type()) {
  mSize = size;
  if (size > 0) {
    mElems = mAlloc.allocate(size);
    std::uninitialized_fill(mElems, mElems + size, val);
  }
  else {
    mElems = nullptr;
  }
}
```

Beispiel: dynamic_array (5)

```
template <typename TIter> void initialize(TIter begin, TIter end, std::false_type) {
  mSize = std::distance(begin, end);
  if (mSize > 0) {
    mElems = mAlloc.allocate(mSize);
    std::uninitialized_copy(begin, end, mElems);
  }
  else {
    mElems = nullptr;
  }
}

~dynamic_array() {
  destroy();
  mAlloc.deallocate(mElems, mSize);
}

void destroy() {
  for (T *p = mElems; p < mElems + mSize; p++) {
    mAlloc.destroy(p);  // oder: p->~T();
  }
}
```

Beispiel: dynamic_array (6)

```
dynamic_array<T, Alloc> &operator=(dynamic_array<T, Alloc> const &other) {
  if (&other != this) {
    destroy();
    if (mSize != other.mSize) {
      mAlloc.deallocate(mElems, mSize);
      mSize = other.mSize;
      mElems = mSize > 0 ? mAlloc.allocate(mSize) : nullptr;
    }
    if (mSize > 0) {
      std::uninitialized_copy(other.mElems, other.mElems + mSize, mElems);
    }
  }
  return *this;
}
```

Beispiel: dynamic_array (7)

```
dynamic_array<T, Alloc> &operator=(dynamic_array<T, Alloc> &&other) {
  if (&other != this) {
    mSize = other.mSize;
    mElems = other.mElems;
    other.mSize = nullptr;
    other.mElems = nullptr;
  }
  return *this;
}

bool operator==(dynamic_array<T, Alloc> const &other) const {
  return mSize == other.mSize && std::equal(mElems, mElems + mSize, other.mElems);
}

// Implementierung Operator !=: return !operator==(other);

bool operator<(dynamic_array<T, Alloc> const &other) const {
  return std::lexicographical_compare(mElems, mElems + mSize,
                                      other.mElems, other.mElems + other.mSize);
}
```

Beispiel: dynamic_array (8)

```
Alloc get_allocator() const { return mAlloc; }

iterator begin() { return mElems; }
iterator end() { return mElems + mSize; }
const_iterator begin() const { return mElems; }
const_iterator end() const { return mElems + mSize; }

reverse_iterator rbegin() { return reverse_iterator(end()); }
reverse_iterator rend() { return reverse_iterator(begin()); }
const_reverse_iterator rbegin() const { return const_reverse_iterator(end()); }
const_reverse_iterator rend() const { return const_reverse_iterator(begin()); }
```

Beispiel: dynamic_array (9)

```
void swap(dynamic_array<T, Alloc> &other) {
  // nicht definiert was für verschiedene Allokatoren passieren soll
  assert(mAlloc == other.mAlloc);
  std::swap(mElems, other.mElems);
  std::swap(mSize, other.mSize);
}

size_type size() const { return mSize; }
size_type max_size() const { return mAlloc.max_size(); }
bool empty() const { return mSize == 0; }

void clear() {
  destroy();
  mSize = 0;
  mElems = nullptr;
}

T &front() { return mElems[0]; }
T const &front() const { return mElems[0]; }
T &back() { return mElems[mSize - 1]; }
T const &back() const { return mElems[mSize - 1]; }
```

Beispiel: dynamic_array (10)

```
T &operator[](size_type i) { return mElems[i]; }
T const &operator[](size_type i) const { return mElems[i]; }

T &at(size_type i) {
  check_index(i);
  return mElems[i];
}

T const &at(size_type i) const {
  check_index(i);
  return mElems[i];
}

void check_index(size_type i) const {
  if (i < 0 || i >= mSize) {
    throw std::out_of_range("invalid dynamic_array index");
  }
}
```

Beispiel: Iterator-Klassen (1)

- Implementierung von iterator und const_iterator als Klassen
- Basierend auf boost::iterator_facade
  - Stellt alle benötigten Operationen mittels CRTP zur Verfügung
  - Implementierung folgender Methoden nötig
    - dereference
    - equal (falls unterstützt)
    - increment, decrement (sofern unterstützt)
    - advance, distance_to (sofern unterstützt)

Beispiel: Iterator-Klassen (2)

```
template <typename TValue, typename TIter>
class iterator_base: public boost::iterator_facade<TIter, TValue,
                                 std::random_access_iterator_tag> {
  template <typename, typename> friend class iterator_base;
  friend class boost::iterator_core_access;
public:
  iterator_base(TValue *ptr) : mPtr(ptr) {}

  template <typename T2, typename TIter2>
  iterator_base(iterator_base<T2, TIter2> const &iter) : mPtr(iter.mPtr) {}

private:
  TValue *mPtr;

  TValue &dereference() const { return *mPtr; }

  template <typename T2, typename TIter2>
  bool equal(iterator_base<T2, TIter2> const &other) const {
    return mPtr == other.mPtr;
  }
```

Beispiel: Iterator-Klassen (3)

```
  void increment() { mPtr++; }
  void decrement() { mPtr--; }
  void advance(std::ptrdiff_t n) { mPtr += n; }

  template <typename T2, typename TIter2>
  std::ptrdiff_t distance_to(iterator_base<T2, TIter2> const &other) const {
    return other.mPtr - mPtr;
  }
};

class iterator : public iterator_base<T, iterator> {
  friend class dynamic_array;
public:
  iterator() : iterator_base<T, iterator>(nullptr) {}
private:
  iterator(T *ptr) : iterator_base<T, iterator>(ptr) {}
};
```

Beispiel: Iterator-Klassen (4)

```
class const_iterator: public iterator_base<T const, const_iterator> {
  friend class dynamic_array;
public:
  const_iterator() : iterator_base<T const, const_iterator>(nullptr) {}
  const_iterator(iterator iter) : iterator_base<T const, const_iterator>(iter) {}
private:
  const_iterator(T const *ptr) : iterator_base<T const, const_iterator>(ptr) {}
};


iterator begin() { return iterator(mElems); }
iterator end() { return iterator(mElems + mSize); }
const_iterator begin() const { return const_iterator(mElems); }
const_iterator end() const { return const_iterator(mElems + mSize); }
```