21. Dynamische Datenstrukturen II

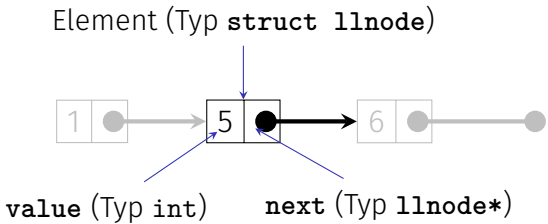
Verkettete Listen, Vektoren als verkettete Listen

Anderes Speicherlayout: Verkettete Liste

- **Kein** zusammenhängender Speicherbereich und **kein** wahlfreier Zugriff
- Jedes Element zeigt auf seinen Nachfolger
- Einfügen und Löschen **beliebiger** Elemente ist einfach

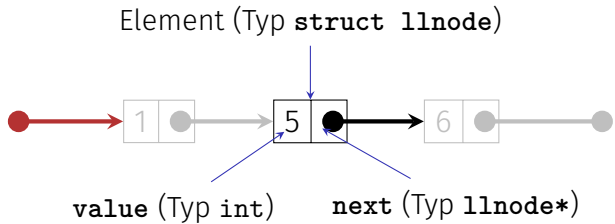
⇒ Unser Vektor kann als verkettete Liste realisiert werden

Verkettete Liste: Zoom




```
struct llnode {
  int value;
  llnode* next;

  llnode(int v, llnode* n): value(v), next(n) {} // Constructor
};
```

Vektor = Zeiger aufs erste Element




```
class llvec {
  llnode* head;
public: // Public interface identical to avec's
  llvec(unsigned int size);
  unsigned int size() const;
  ...
};
```

Funktion `llvec::print()`

```
struct llnode {
  int value;
  llnode* next;
  ...
};
```

```
void llvec::print(std::ostream& sink) const {
  for (llnode* n = this->head;
       n != nullptr;
       n = n->next)
  {
    sink << n->value << ' ';
  }
}
```

Zeiger auf erstes Element

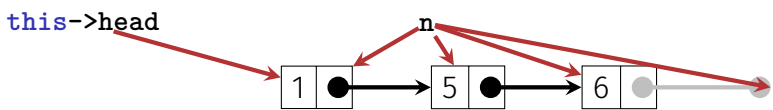
Abbruch falls Ende erreicht

Zeiger elementweise voranschieben

Aktuelles Element ausgeben

Funktion `llvec::print()`

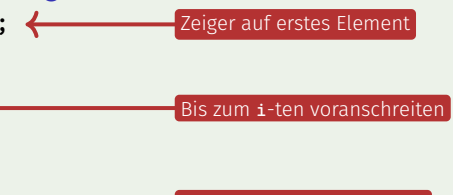
```
void llvec::print(std::ostream& sink) const {
  for (llnode* n = this->head;
       n != nullptr;
       n = n->next)
  {
    sink << n->value << ' ';  // 1 5 6
  }
}
```



Funktion `llvec::operator[]`

Zugriff auf i-tes Element ähnlich implementiert wie **`print()`**:

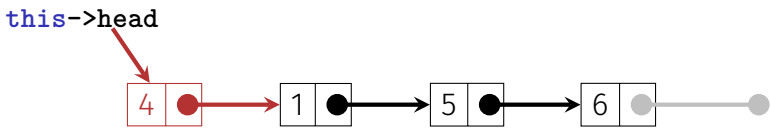
```
int& llvec::operator[](unsigned int i) {
  llnode* n = this->head;

  for (; 0 < i; --i)
    n = n->next;

  return n->value;
}
```



Funktion `llvec::push_front()`

Vorteil **`llvec`**: Elemente am Anfang anfügen ist sehr einfach:

```
void llvec::push_front(int e) {
  this->head =
    new llnode{e, this->head};
}
```



Achtung: Wäre der neue **`llnode`** nicht *dynamisch* alloziert, dann würde er am Ende von **`push_front`** sofort wieder gelöscht (= Speicher deallokiert)

Funktion `llvec::llvec()`

Konstruktor kann mittels **`push_front()`** implementiert werden:

```
llvec::llvec(unsigned int size) {
  this->head = nullptr;

  for (; 0 < size; --size)
    this->push_front(0);
}
```

`head` zeigt zunächst ins Nichts

`size` mal **0** vorne anfügen

Anwendungsbeispiel:

```
llvec v = llvec(3);
std::cout << v; // 0 0 0
```

Funktion `llvec::push_back()`

Einfach, aber ineffizient: Verkettete Liste bis ans Ende traversieren und neues Element anhängen

```
void llvec::push_back(int e) {
  llnode* n = this->head;

  for (; n->next != nullptr; n = n->next);

  n->next =
    new llnode{e, nullptr};
}
```

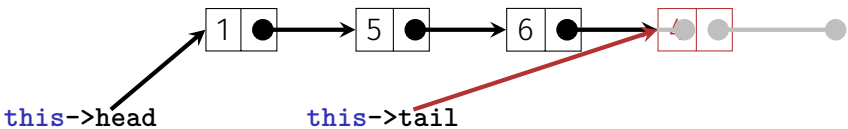
Beim ersten …

… und bis zum letzten Element gehen

Neues Element an bisher letztes anhängen

Funktion `llvec::push_back()`

- Effizienter, aber auch etwas komplexer:
  1. Zweiter Zeiger, der auf das letzte Element zeigt: **`this->tail`**
  2. Mittels dieses Zeigers kann direkt am Ende angehängt werden



- Aber: Verschiedene Grenzfälle, z.B. Vektor noch leer, müssen beachtet werden

Funktion `llvec::size()`

Einfach, aber ineffizient: Grösse durch abzählen *berechnen*

```
unsigned int llvec::size() const {
  unsigned int c = 0;

  for (llnode* n = this->head;
       n != nullptr;
       n = n->next)
    ++c;

  return c;
}
```

Zähler initial 0

Länge der Kette abzählen

Zähler zurückgeben

Funktion `llvec::size()`

Effizienter, aber etwas komplexer: Grösse als Membervariable *nachhalten*

1. Membervariable **`unsigned int count`** zur Klasse **`llvec`** hinzufügen
2. **`this->count`** muss nun bei *jeder* Operation, die die Grösse des Vektors verändert (z.B. **`push_front`**), aktualisiert werden

Effizienz: Arrays vs. Verkettete Listen

- Speicher: Unser **`avec`** belegt ungefähr n ints (Vektorgrösse n), unser **`llvec`** ungefähr $3n$ ints (ein Zeiger belegt i.d.R. 8 Byte)

- Laufzeit (mit **`avec`** = **`std::vector`**, **`llvec`** = **`std::list`**):

22. Container, Iteratoren und Algorithmen

Container, Mengen, Iteratoren, const-Iteratoren, Algorithmen, Templates

Vektoren sind Container

- Abstrakt gesehen ist ein Vektor
  1. Eine Ansammlung von Elementen
  2. Plus Operationen auf dieser Ansammlung
- In C++ heissen **`vector<`**T**`>`** und ähnliche „Ansammlungs“-Datenstrukturen *Container*
- In manchen Sprachen, z.B. Java, *Collections* genannt

Container-Eigenschaften

- Jeder Container hat bestimmte *charakteristische Eigenschaften*
- Ein Array-basierter Vektor z.B. die folgenden:
  - Effizienter, index-basierter Zugriff (`v[i]`)
  - Effiziente Speichernutzung: Nur die Elemente selbst belegen Platz (plus Elementezähler)
  - Einfügen/Entfernen an beliebigem Index ist potenziell ineffizient
  - Suchen eines bestimmten Elements ist potenziell ineffizient
  - Kann Elemente mehrfach enthalten
  - Elemente sind in Einfügereihenfolge enthalten (geordnet aber unsortiert)

Container in C++

- Fast jede Anwendung erfordert die Verwaltung und Manipulation von beliebig vielen Datensätzen
- Aber mit unterschiedlichen Anforderungen (z.B. Elemente nur hinten anhängen, fast nie entfernen, oft suchen, ...)
- Deswegen enthält die Standardbibliothek von C++diverse Container mit unterschiedlichen Eigenschaften, siehe
  `https://en.cppreference.com/w/cpp/container`
- Viele weitere sind über Bibliotheken Dritter verfügbar, z.B.
  `https://www.boost.org/doc/libs/1_68_0/doc/html/container.html`,
  `https://github.com/abseil/abseil-cpp`

Beispiel-Container: `std::unordered_set<`T`>`

- Eine *mathematische Menge* ist eine ungeordnete, duplikatfreie Zusammenfassung von Elementen:

$$\{1, 2, 1\} = \{1, 2\} = \{2, 1\}$$

- In C++: **`std::unordered_set<`**T**`>`**
- Eigenschaften:
  - Kann kein Element doppelt enthalten
  - Elemente haben keine bestimmte Reihenfolge
  - Kein indexbasierter Zugriff (**`s[i]`** nicht definiert)
  - Effiziente „Element enthalten?“-Prüfung
  - Effizientes Einfügen und Löschen von Elementen
- Randbemerkung: Implementiert als Hash-Tabelle

Anwendungsbeispiel `std::unordered_set<`T`>`

Problem:

- Gegeben eine Sequenz an Paaren *(Name, Prozente)* von Code-Expert-Submissions . . .

```
// Input: file submissions.txt
Friedrich 90
Schwerhoff 10
Lehner 20
Schwerhoff 11
```

- . . . bestimme die Abgebenden, die mindestens 50% erzielt haben

```
// Output
Friedrich
```

Anwendungsbeispiel `std::unordered_set<`T`>`

```
std::ifstream in("submissions.txt");
std::unordered_set<std::string> names;

std::string name;
unsigned int score;

while (in >> name >> score) {
  if (50 <= score)
    names.insert(name);
}

std::cout << "Unique submitters: "
          << names << '\n';
```

Öffne `submissions.txt`

Namen-Menge, initial leer

Paar *(Name, Punkte)*

Nächstes Paar einlesen

Namen merken falls Punkte ausreichen

Gemerkte Namen ausgeben

Beispiel-Container: `std::set<`T`>`

- Fast gleich wie **`std::unordered_set<`**T**`>`**, aber die Elemente sind *geordnet*

$$\{1, 2, 1\} = \{1, 2\} \neq \{2, 1\}$$

- Elemente suchen, einfügen und löschen weiterhin effizient (besser als bei **`std::vector<`**T**`>`**), aber weniger effizient als bei **`std::unordered_set<`**T**`>`**
- Denn das Beibehalten der Ordnung zieht etwas Aufwand nach sich
- Randbemerkung: Implementiert als Rot-Schwarz-Baum

Anwendungsbeispiel `std::set<`T`>`

```
std::ifstream in("submissions.txt");
std::set<std::string> names;

std::string name;
unsigned int score;

while (in >> name >> score) {
  if (50 <= score)
    names.insert(name);
}

std::cout << "Unique submitters: "
          << names << '\n';
```

`set` statt `unsorted_set` ...

... und die Ausgabe erfolgt alphabetisch sortiert

Container Ausgeben

- Bereits gesehen: **`avec::print()`** und **`llvec::print()`**
- Wie sieht's mit der Ausgabe von **`set`**, **`unordered_set`**, ... aus?
- Gemeinsamkeit: Über Container-Elemente iterieren und diese ausgeben

Ähnliche Funktionen

- Viele weitere nützliche Funktionen können mittels Container-Iteration implementiert werden:
- **`contains(c, e)`**: wahr gdw. Container **`c`** Element **`e`** enthält
- **`min/max(c)`**: Gibt das grösste/kleinste Element zurück
- **`sort(c)`**: Sortiert die Elemente von **`c`**
- **`replace(c, e1, e2)`**: Ersetzt alle **`e1`** in **`c`** mit **`e2`**
- **`sample(c, n)`**: Wählt zufällig **`n`** Elemente aus **`c`** aus
- ...

Zur Erinnerung: Iterieren mit Zeigern

- Iteration über ein *Array*:
  - Auf Startelement zeigen: `p = this->arr`
  - Auf aktuelles Element zugreifen: `*p`
  - Überprüfen, ob Ende erreicht:
    `p == this->arr + size`
  - Zeiger vorrücken: `p = p + 1`

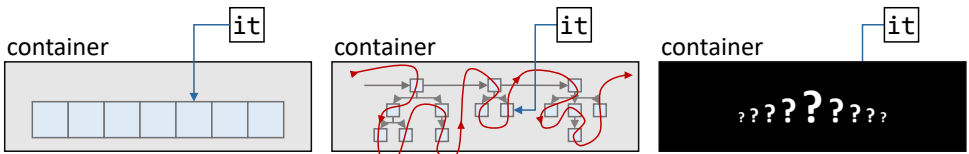
- Iteration über eine *verkettete Liste*:
  - Auf Startelement zeigen: `p = this->head`
  - Auf aktuelles Element zugreifen: `p->value`
  - Überprüfen, ob Ende erreicht: `p == nullptr`
  - Zeiger vorrücken: `p = p->next`

Iteratoren

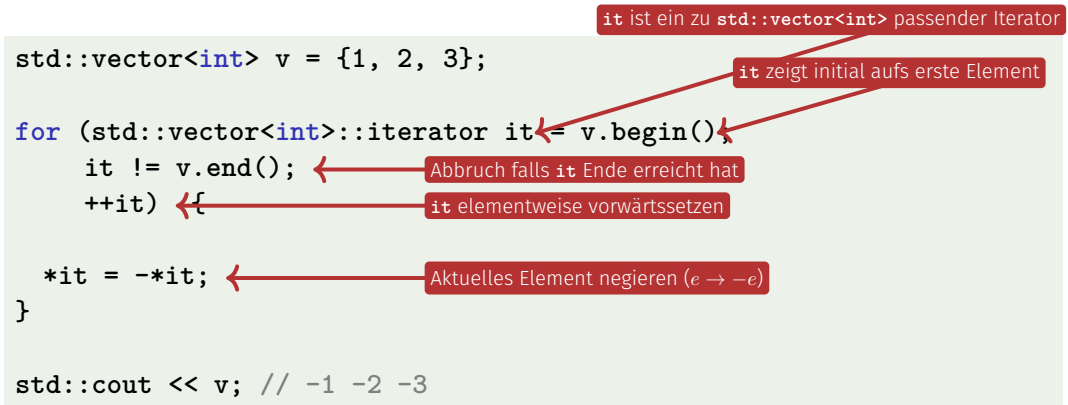
- Iteration erfordert nur die vier eben gesehenen Operationen
- Aber deren Implementierung hängt vom Container ab
- $\Rightarrow$ Jeder C++-Container implementiert seinen eigenen *Iterator*
- Gegeben ein Container `c`:
  - `it = c.begin()`: Iterator aufs erste Element
  - `it = c.end()`: Iterator *hinters* letzte Element
  - `*it`: Zugriff aufs aktuelle Element
  - `++it`: Iterator um ein Element verschieben
- Iteratoren sind quasi gepimpte Zeiger

Iteratoren

- Iteratoren ermöglichen Zugriff auf verschiedene Container auf *uniforme* Weise: **`*it`**, **`++it`**, etc.
- Nutzer bleiben unabhängig von der Container-Implementierung
- Iterator weiss, wie man die Elemente „seines“ Containers abläuft
- Nutzer müssen und sollen interne Details nicht kennen
- ⇒ Containerimplementierung kann geändert werden, ohne das Nutzer Code ändern müssen



Beispiel: Iteration über `std::vector`




```
std::vector<int> v = {1, 2, 3};

for (std::vector<int>::iterator it = v.begin();
     it != v.end();
     ++it) {

  *it = -*it;
}

std::cout << v; // -1 -2 -3
```

Beispiel: Iteration über `std::vector`

Zur Erinnerung: Type-Aliasse können genutzt werden um oft genutzte Typnamen abzukürzen

```
using ivit = std::vector<int>::iterator; // int-vector iterator

for (ivit it = v.begin();
     ...
```

Negieren als Funktion

Wie zuvor: Übergabe eines *Arbeitsbereichs (-intervalls)*

```
void neg(std::vector<int>::iterator begin;
         std::vector<int>::iterator end) {

  for (std::vector<int>::iterator it = begin;
       it != end;
       ++it) {

    *it = -*it;
  }
}
```

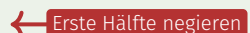
Elemente im Intervall (`begin`, `end`) negieren

Negieren als Funktion

Wie zuvor: Übergabe eines *Arbeitsbereichs (-intervalls)*

```
void neg(std::vector<int>::iterator begin;
         std::vector<int>::iterator end);
```

```
// in main():
std::vector<int> v = {1, 2, 3};
neg(v.begin(), v.begin() + (v.size() / 2));
```



Algorithmen-Bibliothek in C++

- Die C++-Standardbibliothek enthält viele nützliche Algorithmen (Funktionen), die auf durch Iteratoren bestimmten Intervallen [*Anfang*, *Ende*) arbeiten
- Zum Beispiel **`find`**, **`fill`** and **`sort`**; siehe auch `https://en.cppreference.com/w/cpp/algorithm`
- Dank Iteratoren können diese ≥ 100 (!) Algorithmen auf beliebigen* Containern ausgeführt werden: Den 17 (!) Standardcontainern von C++, auf unserem **`avec`** und **`llvec`** (kommt gleich), etc.
- Gäbe es diesen uniformen Zugriff auf Containerelemente nicht, müsste *sehr* viel Code dupliziert werden

Nicht jeder Algorithmus kann auf jedem Container aufgerufen werden. Z.B. kann ein **`std::set`** nicht sortiert werden.

Ein Iterator für `llvec`

Wir brauchen:

1. Einen **`llvec`**-spezifischen Iterator mit mindestens folgender Funktionalität:
   - Zugriff aktuelles Element: **`operator*`**
   - Iterator vorwärtssetzen: **`operator++`**
   - Ende-Erreicht-Prüfung: **`operator!=`** (oder **`operator==`**)
2. Memberfunktionen **`begin()`** und **`end()`** für **`llvec`** um einen Iterator auf den Anfang bzw. hinter das Ende zu erhalten

Iterator `llvec::iterator` (Schritt 1/2)

```
class llvec {
  ...
public:
  class iterator {
    ...
  };

  ...
}
```

- Der Iterator gehört zu unserem Vektor, daher ist **`iterator`** eine öffentliche *innere Klasse* von **`llvec`**
- Instanzen unseres Iterators sind vom Typ **`llvec::iterator`**

Iterator `llvec::iterator` (Schritt 1/2)

```
class iterator {
  llnode* node;

public:
  iterator(llnode* n);
  iterator& operator++();
  int& operator*() const;
  bool operator!=(const iterator& other) const;
};
```

Zeiger auf aktuelles Vektor-Element

Erzeuge Iterator auf bestimmtes Element

Iterator ein Element vorwärtssetzen

Zugriff auf aktuelles Element

Vergleich mit anderem Iterator

Iterator `llvec::iterator` (Schritt 1/2)

```
// Constructor
llvec::iterator::iterator(llnode* n): node(n) {}

// Pre-increment
llvec::iterator& llvec::iterator::operator++() {
  assert(this->node != nullptr);

  this->node = this->node->next;

  return *this;
}
```

Iterator initial auf `n` zeigen lassen

Iterator ein Element vorwärtssetzen

Referenz auf verschobenen Iterator zurückgeben

Iterator `llvec::iterator` (Schritt 1/2)

```
// Element access
int& llvec::iterator::operator*() const {
  return this->node->value;
}


// Comparison: when are two iterators not equal?
bool llvec::iterator::operator!=(
    const llvec::iterator& other) const
{
  return this->node != other.node;
}
```

Zugriff auf aktuelles Element

this Iterator verschieden von **other** falls sie auf unterschiedliche Elemente zeigen

Ein Iterator für `llvec` (Wiederholung)

Wir brauchen:

1. Einen `llvec`-spezifischen Iterator mit mindestens folgender Funktionalität:
   - Zugriff aktuelles Element: `operator*`
   - Iterator vorwärtssetzen: `operator++`
   - Ende-Erreicht-Prüfung: `operator!=` (oder `operator==`)

2. Memberfunktionen `begin()` und `end()` für `llvec` um einen Iterator auf den Anfang bzw. hinter das Ende zu erhalten

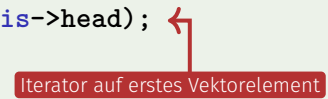
Iterator `llvec::iterator` (Schritt 2/2)

```
class llvec {
  ...
public:
  class iterator {...};

  iterator begin();
  iterator end();

  ...
}
```

`llvec` braucht Memberfunktionen um Iteratoren *auf den Anfang* bzw. *hinter das Ende* des Vektors herausgeben zu können

Iterator `llvec::iterator` (Schritt 2/2)

```
llvec::iterator llvec::begin() {
  return llvec::iterator(this->head);
}

llvec::iterator llvec::end() {
  return llvec::iterator(nullptr);
}
```



Const-Iteratoren

- Neben **`iterator`** sollte jeder Container auch einen *Const-Iterator* **`const_iterator`** bereitstellen
- Const-Iteratoren gestatten nur Lesezugriff auf den darunterliegenden Container
- Zum Beispiel für **`llvec`**:

```
llvec::const_iterator llvec::cbegin() const;
llvec::const_iterator llvec::cend() const;

const int& llvec::const_iterator::operator*() const;
...
```

- Daher nicht möglich (Compilerfehler): **`*(v.cbegin()) = 0`**

Const-Iteratoren

Const-Iterator *kann* verwendet werden um nur Lesen zu erlauben:

```
llvec v = ...;
for (llvec::const_iterator it = v.cbegin(); ...)
  std::cout << *it;
```

Hier könnte auch der nicht-const **`iterator`** verwendet werden

Const-Iteratoren

Const-Iterator *muss* verwendet werden falls Vektor selbst const ist:

```
const llvec v = ...;
for (llvec::const_iterator it = v.cbegin(); ...)
  std::cout << *it;
```

Hier kann nicht der **`iterator`** verwendet werden (Compilerfehler)

Bereichsbasierte `for`-Schleife

- Sequenzielle Iteration mittels eines Iterators über einen **`llvec`** (const-Iterator möglich; andere Container möglich):

```
llvec v(3); // v == {0, 0, 0}
for (llvec::iterator it = v.begin(); it != v.end(); ++it)
  std::cout << *it; // 000
```

- Kann alternativ auch wie folgt geschrieben werden:

```
for (int i : v) std::cout << i; // 000
```

  Wird dann zu Iterator-basierter Schleife übersetzt.

- Modifizierender Zugriff ist auch möglich:

```
for (int& i : v) i += 3;
for (int i : v) std::cout << i; // 369
```

Typ-generischer Container

Typ-spezifischer Container

Typ-generischer Container

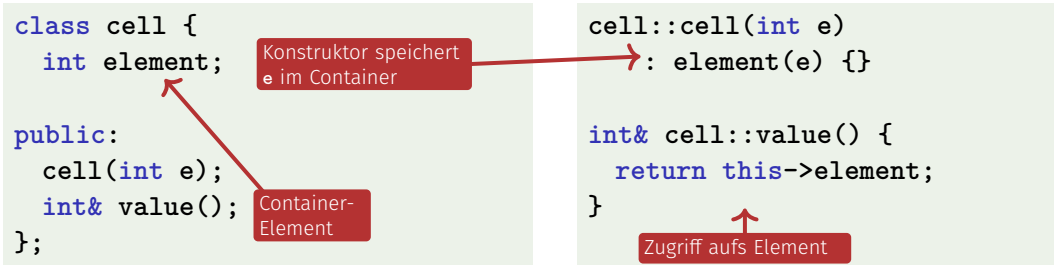
`https://upload.wikimedia.org/wikipedia/commons/d/df/Container_01_KMJ.jpg` (CC BY-SA 3.0)

P.S.: Templates sind nicht klausurrelevant

(Typ-generische Container (allgemein, Templates) sind nicht klausurrelevant)

Typ-generischer Container

Klasse **`cell`**: Ein einfacher, 1-elementiger Container für **`int`**



Besser: Generische **`cell<E>`** für jeden Elementtyp **`E`** (analog zu **`std::vector<E>`**)

Typ-generischer Container mit Templates

Templates ermöglichen *Typ-generische* Funktionen und Klassen:

```
template<typename E>
class cell {
  E element;

public:
  cell(E e);
  E& value();
};
```

Sei `E` ein beliebiger Typ …

…dann verwaltet `cell` ein Element vom Typ `E`

- Typen können als *Parameter* genutzt werden
- Typ-Parameter sind im „templatierten“ Gültigkeitsbereich nutzbar

Typ-generischer Container mit Templates

- Signaturen und Implementierungen müssen „templatiert“ werden
- Bei separaten Implementierungen muss das Klassen-Präfix in generischer Form angegeben werden

```
template<typename E>
class cell {
  E element;

public:
  cell(E e);
  E& value();
};
```

```
template<typename E>
cell<E>::cell(E e)
    : element(e) {}

template<typename E>
E& cell<E>::value() {
  return this->element;
}
```

Typ-generischer Container mit Templates

```
cell<int> c1(313);
cell<std::string> c2("terrific!")
```

- Typ-Parameter müssen bei *Deklarationen*, z.B. **`cell<int>`**, explizit angegeben werden …
- …aber überall sonst werden sie vom Compiler *inferriert*, z.B. bei **`c1(313)`**, d.h. beim Aufruf des generischen Konstruktors **`cell(E e)`** (wobei Typparameter **`E`** vom Compiler mit **`int`** instanziiert wird)

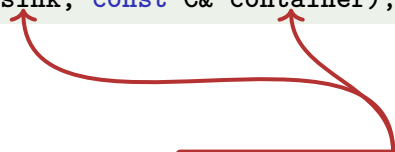
Mehr Templates: generischer Ausgabe-Operator

- Ziel: Ein *generischer* Ausgabe-Operator **<<** für *iterierbare Container*: **llvec**, **avec**, **std::vector**, **std::set**, ...
- D.h. **std::cout << c << '\n'** soll für jeden solchen Container **c** funktionieren

Mehr Templates: generischer Ausgabe-Operator

- Generischer Ausgabe-Operator mit zwei Typ-Parametern

```
template <typename S, typename C>
S& operator<<(S& sink, const C& container);
```



Intuition: Operator funktioniert für jeden Ausgabestrom **`sink`** vom Typ **`S`** und jeden Container **`container`** vom Typ **`C`**

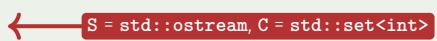
Mehr Templates: generischer Ausgabe-Operator

- Generischer Ausgabe-Operator mit zwei Typ-Parametern

```
template <typename S, typename C>
S& operator<<(S& sink, const C& container);
```

- Der Compiler inferiert passende Typen aus den Aufrufargumenten

```
std::set<int> s = ...;
std::cout << s << '\n';
```



Mehr Templates: generischer Ausgabe-Operator

Implementierung von `<<` *schränkt* `S` und `C` *ein* (Compilerfehler falls nicht erfüllt):

```
template <typename S, typename C>
S& operator<<(S& sink, const C& container) {
  for (typename C::const_iterator it = container.begin();
       it != container.end();
       ++it) {

    sink << *it << ' ';
  }

  return sink;
}
```

`C` muss Iteratoren bereitstellen – mit passenden Funktionen

Mehr Templates: generischer Ausgabe-Operator

Implementierung von `<<` *schränkt* `S` und `C` *ein* (Compilerfehler falls nicht erfüllt):

```
template <typename S, typename C>
S& operator<<(S& sink, const C& container) {
  for (typename C::const_iterator it = container.begin();
       it != container.end();
       ++it) {

    sink << *it << ' ';
  }

  return sink;
}
```

`S` muss Ausgabe von Elementen (`*it`) und Zeichen (`' '`) unterstützen

Templates: Abschluss

- Templates realisieren in C++ *statische Codegenerierung* bzw. *statische Metaprogrammierung*
- Template-Code wird pro Typ-Instanziierung *kopiert*. Bei Benutzung von **`cell<int>`** und **`cell<std::string>`** legt der Compiler zwei *instanziierte Kopien* des **`cell`**-Codes an: sozusagen die beiden (nicht mehr generischen) Klassen **`cell_int`** und **`cell_stdstring`**.
- Templates reduzieren Codeduplikation und fördern Code-Wiederverwendbarkeit
- Leider sind Compiler-Fehlermeldungen, die sich auf Templates beziehen, oft noch komplexer, als es C++-Fehlermeldungen ohnehin schon oft sind