

Programmierkurs

Vorlesung 8

Dr. Ole Klein

Interdisziplinäres Zentrum für Wissenschaftliches Rechnen
Universität Heidelberg

15. Januar 2021

Iteratoren

Motivation

Pointer

Pointer als Handles

Iteratoren

Algorithmen

Iteratoren — Motivation

Im Internet findet man oft Codeschnipsel wie dieses:

```
std::vector<int> v(20);  
...  
for (std::vector<int>::iterator it = v.begin();  
     it != v.end();  
     ++it)  
    std::cout << *it << std::endl;
```

- ▶ Was passiert hier eigentlich?
- ▶ Was ist dieser iterator?
- ▶ Was macht *it?
- ▶ Was soll das Ganze?

Exkursion: Pointer

- ▶ Erinnerung: Jede Variable liegt an einer **Adresse** im Speicher
- ▶ Zugriff auf die Adresse mit Adressoperator `&`:

```
int i = 0;  
std::cout << &i << std::endl;
```

- ▶ Variablen, die die Adresse einer anderen Variable speichern, heißen **Pointer**
- ▶ Pointer zeigen immer auf Variablen eines bestimmten Typs. Der Typ einer Pointervariablen ist der Typ der Zielvariablen mit angehängtem `*`:

```
int i = 0;  
int* p = &i; // p is a pointer to int
```

- ▶ Pointer, die auf keine gültige Variable verweisen, sollte immer der spezielle Wert **`nullptr`** zugewiesen werden:

```
int* p = nullptr;
```

Arbeiten mit Pointern

- ▶ Pointern können neue Zielvariablen zugewiesen werden:

```
int i = 0, j = 2;
int* p = nullptr; // p does not point anywhere valid
p = &i;           // p now points to i
p = &j;           // p now points to j
```

- ▶ Um auf den Wert der Zielvariablen zuzugreifen, dereferenziert man den Pointer, indem man * voranstellt:

```
std::cout << *p << std::endl; // prints 2
```

- ▶ Wenn die Zielvariable eine Klasse ist, kann man mit -> direkt auf Member der Zielvariablen zugreifen:

```
Point p{1.0,2.0};
Point* pp = &p;
std::cout << pp->x() << std::endl; // prints 2.0
```

Pointer als Handle für Speicher

- ▶ array und vector legen ihre Daten in einen zusammenhängenden Speicherbereich
- ▶ Pointer auf Speicherbereich mit Memberfunktion data()
- ▶ Pointer unterstützen mathematische Operationen:

```
std::vector<int> v(20);  
int* data = v.data();  
std::cout << *data << std::endl; // prints first entry  
++data; // increase pointer by sizeof(int), now points to v[1]  
data += 10; // now points to v[11]  
std::cout << (data - v.data()) << std::endl; // prints 11  
data -= 11; // points to v[0] again
```

- ▶ Vektor mit Pointern ausgeben:

```
int* end = v.data() + v.size(); // first invalid address  
for(int* p = v.data() ; p != end ; ++p)  
    std::cout << *p << std::endl;
```

Iteratoren: Verallgemeinerte Pointer

- ▶ Nicht alle C++-Container speichern Einträge in zusammenhängendem Speicher (`list`, `map`, `deque`, ...)
- ▶ Nicht alle C++-Container erlauben Elementzugriff mit eckigen Klammern
- ▶ Wie allgemeine Algorithmen schreiben, die über Container-Elemente iterieren?

Iteratoren: Verallgemeinerte Pointer

- ▶ Nicht alle C++-Container speichern Einträge in zusammenhängendem Speicher (`list`, `map`, `deque`, ...)
- ▶ Nicht alle C++-Container erlauben Elementzugriff mit eckigen Klammern
- ▶ Wie allgemeine Algorithmen schreiben, die über Container-Elemente iterieren?

Container stellen Iteratoren zur Verfügung

- ▶ Iteratoren verhalten sich wie Pointer
- ▶ Typ des Iterators über geschachtelten Typ
`Container::iterator` bzw. `Container::const_iterator`
(erlaubt nur lesenden Zugriff auf Elemente)
- ▶ Iterator für erstes Element mit `begin()`
- ▶ Iterator `hint`er letztes Element mit `end()`
- ▶ Manche Container erlauben Rückwärtsdurchlauf mit `rbegin()`, `rend()`

Iteratoren: Beispiel

Ausgeben von Containern:

```
template<typename T>
void print(const T& t) {
    typename T::const_iterator end = t.end();
    for (auto it = t.begin() ; it != end ; ++it)
        std::cout << *it << std::endl;
}

std::vector<int> v(20);
print(v);

std::list<int> l;
...
print(l);
```

Iteratoren: Kategorien

- ▶ Je nach unterliegendem Container unterstützen Iteratoren nicht alle Pointer-Operationen
- ▶ Iterator-Kategorien:
 - `InputIterator` Lesen von `*it` und `++it`
 - `OutputIterator` Schreiben von `*it` und `++it`
 - `ForwardIterator` Vollzugriff auf `*it` sowie `++it`
 - `BidirectionalIterator` zusätzlich `--it`
 - `RandomAccessIterator` zusätzlich `it += n`, `it -= n`
- ▶ Jeder Container gibt an, was für eine Iteratorkategorie er hat
- ▶ Es gibt praktische Hilfsklassen wie z.B. `std::back_inserter(v)`, der neue Einträge an einen bestehenden Container anfügt

Algorithmen

- ▶ Alle Algorithmen in der Standardbibliothek arbeiten mit Iteratoren
- ▶ Manche Algorithmen haben Anforderungen an die Kategorie (z.B. `std::sort()`)
- ▶ Erlauben oft sehr klares Aufschreiben der Intention:

```
std::array<int,20> a;  
...  
// Replace all occurrences of 3 with 7  
std::replace(a.begin(),a.end(),3,7);  
  
// Count number of entries with value 7  
std::cout << std::count(a.begin(),a.end(),7);  
  
std::vector<int> v;  
  
// Copy array to vector, no need to resize  
std::copy(a.begin(),a.end(),std::back_inserter(v));
```

- ▶ Erfordern Umgewöhnung und Kenntnis der Möglichkeiten