Praxisorientierte Einführung in C++
Lektion: "STL-Algorithmen"

Christof Elbrechter

Neuroinformatics Group, CITEC

July 3, 2014

Table of Contents

STL-Algorithmen (Überblick)

- STL bietet eine Vielzahl von Algorithmen
- Diese haben i.d.R. das bereits vorgestellte Iterator-basierte Interface
- Algorithmen sind definiert im Header `<algorithm>`
  - `std::copy`, `std::fill`, `std::find`, `std::binary_search`, `std::sort`, `std::max_element`, `std::replace_if`, `std::transform`, `std::for_each`, ...
- Ein paar mehr im Header `std::<numeric>`
  - `std::accumulate`, `std::inner_product`, `std::partial_sum`, ...

std::fill

- `std::fill` überschreibt alle Elemente einer Range mit einem bestimmten Wert
- Genau genommen kopiert sie eine gegebene Instanz in alle Instanzen einer Range

Beispiel `std::fill`

```
// Zeiger sind auch Iteratoren !
int ages = new int[10];

// Aber mit Containern geht's auch
std::vector<std::string> names(10);

// signatur: std::fill(begin,end,value)
std::fill(ages,ages+10, 42);
std::fill(names.begin(), names.end(),"Arthur Dent");
```

std::fill

- `value` muss irgendwie implizit in den Wert-Typ des Iterators umgewandelt werden können
- Im Beispiel:
  - ages-iterator-Typ ist `int` ⇒ direkte Zuweisung möglich
  - `names`-iterator-Typ ist `std::string` ⇒ indirekte Zuweisung eines `const char*`

std::copy

- std::copy kopiert alle Elemente einer Range in eine andere Range. Typen müssen nicht identisch, sondern nur kompatibel sein

Beispiel: std::copy

```
#include <vector>
#include <string>
#include <iostream>
#include <iterator>

int main(int n, char **ppc){
  std::vector<std::string> args(n-1);
  std::copy(ppc+1,ppc+n,args.begin());

  // oder mit inserter pattern
  std::vector<std::string> args2;
  std::copy(ppc+1,ppc+n,back_inserter(args2));

  // man kann auch in einen ostream 'kopieren'
  std::copy(args2.begin(),args2.end(),
        std::ostream_iterator<std::string>(std::cout,"\n"));
}
```

std::copy

- Stream-Iteratoren ermöglichen Iterator-basierte Funktionen mit Streams zu verwenden
- Auch möglich: istream-Iteratoren

Beispiel: Stream-Iteratoren (Teil 1/2)

```
#include <iostream>
#include <fstream>
typedef std::istream_iterator<int> iii;
typedef std::ostream_iterator<int> oii;
int main(){
  std::vector<int> v;
  std::cout << "Bitte ein paar Zahlen eingeben: " << std::flush;
  std::copy(iii(std::cin),iii(),std::back_inserter(v));
  std::cout << "Diese Elemente wurden eingegeben: " << std::endl;
  std::copy(v.begin(),v.end(),oii(std::cout," "));
  std::cout << std::endl;

  //...
```

std::copy

Beispiel: Stream-Iteratoren (Teil 2/2)

```
  //...
  // Oder in einen File:
  std::cout << "Schreibe Zahlen in File file.txt" << std::endl;
  std::ofstream ofs("file.txt");
  std::copy(v.begin(),v.end(),oii(ofs,"\n"));

  ofs.close();
  std::ifstream ifs("file.txt");
  std::cout << "Lese sie wieder: " << std::flush;
  std::copy(iii(ifs),iii(),oii(std::cout," "));
  std::cout << std::endl;
}
```

Ausgabe

```
Bitte ein paar Zahlen eingeben: 1 2 3 4
x
Diese Elemente wurden eingegeben:
1 2 3 4
Schreibe Zahlen in File file.txt
Lese sie wieder: 1 2 3 4
```

for_each

- `for_each` haben wir schon fünf mal besprochen und dreimal implementiert :)
- Erinnerung: `for_each(begin,end,Funktion-oder-Funktor)`
- Wendet `Funktion-oder-Funktor` auf jedes Element der gegebenen Range an
- Ist nicht schneller als `for`-Schleife
  - Loop-Unrolling lohnt hier nicht (aber $\Rightarrow$ siehe find)

std::max_element

- std::max_element sucht das maximale Element in einer Range
- Natürlich gibt es auch std::min_element
- Jeweils zwei Versionen:
  - Eine nimmt std::less<T> also den operator< zum Vergleich der Elemente
  - Die andere nimmt gegebene(n) Vergleichs-Funktion/Funktor

Beispiel: std::max_element

```
int main(){
   int p[] = {3,4,8,5,3,10,34};
   int max = *std::max_element(p,p+7);
}
```

std::transform

- `std::transform` ist ein sehr mächtiges Utility, um die Elemente einer Range zu transformieren
- Zwei unterschiedliche Versionen:

```
template<class input_iterator, class output_iterator, class UnaryFunction>
output_iterator transform(input_iterator start,
/*Version A:*/            input_iterator end,
                          output_iterator result,
                          UnaryFunction f);
template<class input_iterator, class output_iterator, class BinaryFunction>
output_iterator transform(input_iterator start1,
/*Version B:*/            input_iterator end1,
                          input_iterator2 start2,
                          output_iterator result,
                          BinaryFunction f);
```

- Version A: `result[i] = f(start[i])` $\forall i \in \{0,\text{end-start}\}$
- Version A: `result[i] = f(start1[i],start2[i])` $\forall i \in \{0,\text{end1-start1}\}$

std::transform

Beispiel Version A:

```
#include <string>
#include <iostream>
#include <algorithm>
#include <cstring>
#include <iterator>
#include <vector>

int main(int n, char **ppc){
  std::string s("This is a String"),s2;
  // out-of-place
  std::transform(s.begin(), s.end(), std::back_inserter(s2), toupper);
  // in-place
  std::transform(s.begin(), s.end(), s.begin(), toupper);
  std::cout << s2 << std::endl;
  std::vector<int> v;
  std::transform(ppc+1,ppc+n,std::back_inserter(v),atoi);
  std::copy(v.begin(),v.end(),std::ostream_iterator<int>(std::cout," "));
}
```

std::transform

Beispiel Version B:

```
#include <iostream>
#include <algorithm>
#include <iterator>
#include <functional>

int main(int n, char **ppc){
  int vec1[8] = {1,2,3,4,5,6,7,8};
  int vec2[8] = {3,5,6,7,8,3,5,6};

  int sum[8];
  std::transform(vec1,vec1+8,vec2,sum,std::plus<int>());
  // (std::plus finden wir in <functional> -> gleich!
  std::copy(sum,sum+8,std::ostream_iterator<int>(std::cout," "));
}
```

std::sort

- Sehr hilfreich
- Auch zwei Versionen (mit und ohne Vergleichs-Funktor)

Beispiel Version B:

```
#include <algorithm>
int main(){
  int a[]  = {3,4,5,3,7,4,3,1};
  std::sort(a,a+8);
}
```

- Sehr effizient (O(n * log(n)) (i.d.R. sog. *Intro-Sort*)
- Deutlich schneller als `void*`-basierte C-Version `qsort`

std::accumulate

- std::accumulate akkumuliert alle Elemente innerhalb einer Range

Beispiel: std::accumulate

```
#include <numeric>
int main(int n, char **ppc)
{
  int is[5] = { 1,2,3,4,5 };
  int sum = std::accumulate(is,is+5,0);

  // sum ist 15
}
```

std::accumulate

- Tatsächlich ist std::accumulate aber ein wenig allgemeiner

```
template <class InputIterator, class T>
T accumulate(InputIterator start,InputIterator finish,T init);

template <class InputIterator,class T,class BinaryOperation>
T accumulate(InputIterator start,InputIterator finish,
             T init,BinaryOperation binary_op);
```

- binary_op kann benutzt werden, um Elemente anders zu "akkumulieren"

std::accumulate

- Hier bilden wir ein Produkt:

std::accumulate mit 'BinaryFunction'

```
#include <numeric>
#include <iostream>
int mul(int a, int b) { return a*b; }

int main(int n, char **ppc){
  int is[5] = { 1,2,3,4,5 };
  int prod = std::accumulate(is, is+5, 1, mul);
  std::cout << prod << std::endl;
}
```

Wichtige Anmerkung zu `std::accumulate`

- Accumulator-Basis hat eigenen Template-Parameter

Negativ-Beispiel

```
#include <numeric>
#include <iostream>
int main(int n, char **ppc){
  float fs[5] = { 0.1, 0.2, 0.3, 0.4, 0.5 };
  float sum = std::accumulate(fs, fs+5, 0);
  std::cout << sum << std::endl;
}
```

- Ergebnis ist 0: Warum?

Wichtige Anmerkung zu `std::accumulate`

- Accumulator-Basis hat eigenen Template-Parameter

Negativ-Beispiel

```
#include <numeric>
#include <iostream>
int main(int n, char **ppc){
  float fs[5] = { 0.1, 0.2, 0.3, 0.4, 0.5 };
  float sum = std::accumulate(fs, fs+5, 0);
  std::cout << sum << std::endl;
}
```

- Ergebnis ist 0: Warum?
- **Lösung:** 0 hat den Typ `int`
- Ausweg: `float sum = std::accumulate(fs, fs+5, 0.0f);`

Funktional Programmieren

- Der STL-Header <functional> enthält Grundbausteine um mittels `transform` & Co. *funktional-angehaucht* zu programmieren
- Hier werden C++-Operatoren nochmal als Template-Funktionen angeboten
- Da Operator-Symbole ja keine Funktionen sind
- .. und somit auch nicht als Funktion/Funktor übergeben werden können
- Funktoren können ferner mit sog. *bind*-Patterns *ge-curried* werden[1]

[1] nach Haskell Curry, http://en.wikipedia.org/wiki/Currying

<functional> enthält:

- Arithmetische Operatoren:
  `plus`, `minus`, `multiplies`, `devides`, `modulus` und `negate`
- Vergleichsoperatoren:
  `less`, `equal_to`, `not_equal_to`, `greater_equal`, ...
- Logische Operatoren:
  `logical_and`, `logical_or` und `logical_not`
- Adapter:
  - Negatoren: `not1` und `not2`
  - Binder: `bind1st` und `bind2nd`
  - Convertoren: `ptr_fun`, `mem_fun` und `mem_fun_ref`
- Die Verwendung ist schon ein wenig kompliziert
- Ermöglicht aber teilweise sehr eleganten Code

Binder

- Arithmetische Operatoren sind binär (Ausnahme negate)
- Was, wenn man mit Konstanten rechnen möchte?

Beispiel

```
#include <iostream>
#include <functional>
#include <algorithm>
#include <iterator>

int add1(int n) { return n+1; }
int main(){
  int is[10]={0,1,2,3,4,5,6,7,8,9};

  std::transform(is,is+10,is,add1); // ok
  // aber: int std::plus<int>(int a, int b) { return a+b; } ??

  std::copy(is,is+10,std::ostream_iterator<int>(std::cout," "));
}
```

- $\Rightarrow$ Binder verwenden

Nun mit Binder

Beispiel

```
int main(){
  int is[10]={0,1,2,3,4,5,6,7,8,9};

  std::transform(is,is+10,is,std::bind1st(std::plus<int>(),1));

  std::copy(is,is+10,std::ostream_iterator<int>(std::cout," "));
}
```

- Binder bindet Objekt-Instanz an binäre Funktion
- Ergebnis: Neuer-Funktor mit noch einem offenen Parameter
- Noch 1000-mal cooler: `boost::bind`
- Soll hier aber nicht vorgestellt werden

mem_fun_ref

- mem_fun_ref erzeugt einen Funktor, welcher eine Member-Funktion aufruft

Beispiel

```
#include <iostream>
#include <functional>
#include <vector>
#include <algorithm>
#include <string>
int main () {
  std::string numbers[] = {"one","two","three","four","five"};
  int lengths[5];
  std::transform (numbers,numbers+5,lengths,
                  std::mem_fun_ref(&std::string::length));

  for (int i=0; i<5; i++) {
    cout << numbers[i] << " has " << lengths[i] << " letters.\n";
  }
  return 0;
}
```

Anmerkungen zu `mem_fun_ref`

- Es gibt wieder unterschiedliche Versionen!
- `mem_fun` arbeitet ähnlich – erzeugt aber für jedes Element eine Kopie

std::find

- Ähnlich zu `std::min_element` und `std::max_element`:
- Rückgabe ist Iterator welcher auf die Stelle *zeigt*, wo sich das gesuchte Element befindet (oder end falls nicht gefunden)
- **Aber:** oft schneller als naive `for`-`if`-Kombination

Naive Implementation

```
template<class ForwardIterator, class T>
ForwardIterator find_naive(ForwardIterator begin,
                           ForwardIterator end, const T&x){
   while(begin != end){
      if(*begin == x) return begin;
      else ++begin;
   }
}
```

- Problem: *Branch-Prediction* ist sehr schwer
- "Compiler muss jederzeit *damit rechnen*, dass die Schleife verlassen werden muss"

Benchmark-Vergleich naive_find und std::find

```
int main(){
  static const int N = 100000;
  int *p = new int[N];
  std::fill(p,p+N,0);

  for(int i=0;i<1000;++i){
    std::find(p,p+N,1);
    find_naive(p,p+N,1);
    find_opt(p,p+N,1);
  }
}
```

	find_naive	std::find	find_opt[2]
-O0	353ns	295ns	242ns
-O1	97ns	64ns	120ns
-O2	110ns	81ns	62ns
-O3	110ns	81ns	62ns
-O3 -funroll-loops	60ns	67ns	62ns
-O3 -march=native -funroll-loops	60ns	60ns	60ns

[2] gleich!

Implementation von `find_opt`

`find_opt`-Template

```
template<class RandomAccessIterator, class T>
RandomAccessIterator opt_find(RandomAccessIterator begin,
                              RandomAccessIterator end, const T &x){
  int n = (int)((end - begin) >> 2); // schneller als / 4
                                     // '/' ist immer sehr 'teuer'
  for (; n ; --n, begin+=4) {
    if(begin[0] == x) return begin;   // pro Durchlauf werden 4
    if(begin[1] == x) return begin+1; // Elemente auf einmal
    if(begin[2] == x) return begin+2; // verglichen
    if(begin[3] == x) return begin+3; // 'loop-unrolling'
  }
  switch(end - begin){                // restliche 0-3 Elemente
    case 3:  if(begin[2] == x) return begin+2;
    case 2:  if(begin[1] == x) return begin+1;
    case 1:  if(begin[0] == x) return begin;
  }
  return end;
}
```

- Gute Fingerübung und in Einzelfällen sinnvoll
- Aber *profile first – optimize later*!