

3. Generische Programmierung in C++

Nicht-Modifizierende Algorithmen (`#include <algorithm>`)

<code>for_each</code>	eine (lesende) Operation auf alle Elemente anwenden
<code>find</code>	erstes Auftreten eines Wertes ermitteln
<code>find_if</code>	erste Erfüllung eines Prädikates ermitteln
<code>search</code>	erstes Auftreten einer Teilfolge ermitteln, gleiche Werte / Prädikat ! overloaded
<code>find_end</code>	letztes Auftreten einer Teilfolge ermitteln, gleiche Werte
<code>find_end_if</code>	letztes Auftreten einer Teilfolge ermitteln, Prädikat
<code>find_first_of</code>	erstes Auftreten eines Elementes aus einem Bereich ermitteln, gleiche Werte
<code>find_first_of_if</code>	erstes Auftreten eines Elementes aus einem Bereich ermitteln, Prädikat
<code>adjacent_find</code>	erstes Auftreten benachbarter gleicher Elemente ermitteln, gleiche Werte / Prädikat ! overloaded
<code>min_element</code>	Position des kleinsten Elements ermitteln, < / binäres bool Prädikat
<code>max_element</code>	Position des größten Elements ermitteln, < / binäres bool Prädikat
<code>count</code>	Elemente zählen, gleich Vorgabewert
<code>count_if</code>	Elemente zählen, für die Prädikat erfüllt
<code>equal</code>	Vergleich Bereich gegen Bereichsanfang, gleiche Werte / Prädikat
<code>lexicographical_compare</code>	Vergleich zweier Bereiche, elementweise gleich / Prädikat
<code>mismatch</code>	Position der ersten Abweichung bei Vergleich Bereich gegen Bereichsanfang gleiche Werte / Prädikat (Ergebnis: Iteratorpaar!)

3. Generische Programmierung in C++

Modifizierende Algorithmen (---> nicht für [multi]set/map !) (`#include <algorithm>`)

<code>copy</code>	einen Bereich an einen Bereichsanfang kopieren (Platz muss ausreichen!)
<code>copy_backward</code>	einen Bereich vor ein Bereichsende kopieren (Platz muss ausreichen!)
<code>swap_ranges</code>	Austausch Bereich gegen Bereichsanfang
<code>transform</code>	eine lesende und schreibende Operation auf alle Elemente anwenden auch mit Bereich und Bereichsanfang, die binär verknüpft werden und Ergebnisse an einen weiteren Bereichsanfang geschrieben werden (Platz muss ausreichen!)
<code>fill</code>	einen Bereich mit einem Wert füllen
<code>fill_n</code>	ab Bereichsanfang n Elemente mit einem Wert füllen (Platz muss ausreichen!)
<code>generate</code>	einen Bereich mit dem Ergebnis einer Operation füllen
<code>generate_n</code>	ab Bereichsanfang n Elemente mit dem Erg. einer Operation füllen (Platz muss ausreichen!)
<code>replace</code>	in einem Bereich alle Elemente mit altem Wert durch neuen Wert ersetzen
<code>replace_if</code>	in einem Bereich alle Elemente für die eine Prädikat gilt durch neuen Wert ersetzen
<code>replace_copy</code>	in einem Bereich alle Elemente mit altem Wert durch neuen Wert ersetzt an einen Bereichsanfang kopieren
<code>replace_copy_if</code>	in einem Bereich alle Elemente für die eine Prädikat gilt durch neuen Wert ersetzt an einen Bereichsanfang kopieren

3. Generische Programmierung in C++

Löschende Algorithmen (`#include <algorithm>`)

**ACHTUNG: Größe und Ende des manipulierten Containers ändert sich NICHT!
Ergebnis ist immer das neue Ende des Containers!**

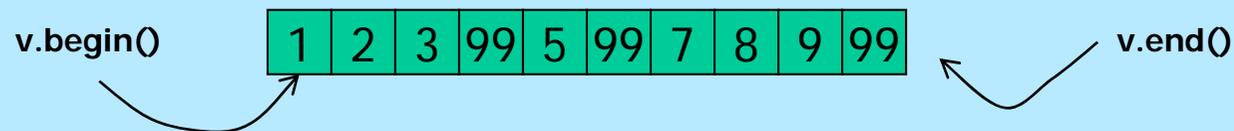
<code>remove</code>	in einem Bereich alle Elemente löschen, die gleich einem Wert sind
<code>remove_if</code>	in einem Bereich alle Elemente löschen, für die ein Prädikat gilt
<code>remove_copy</code>	aus einem Bereich alle Elemente die gleich einem Wert sind, gelöscht an einen Bereichsanfang kopieren
<code>remove_copy_if</code>	aus einem Bereich alle Elemente für die ein Prädikat gilt, gelöscht an einen Bereichsanfang kopieren
<code>unique</code>	in einem Bereich gleiche aufeinanderfolgende Elemente zu einem kollabieren UND in einem Bereich bzgl. einem Prädikat gleiche aufeinanderfolgende Elemente zu einem kollabieren
<code>unique_copy</code>	aus einem Bereich gleiche aufeinanderfolgende Elemente zu einem kollabiert an einen Bereichsanfang kopieren UND aus einem Bereich bzgl. einem Prädikat gleiche aufeinanderfolgende Elemente zu einem kollabiert an einen Bereichsanfang kopieren

3. Generische Programmierung in C++

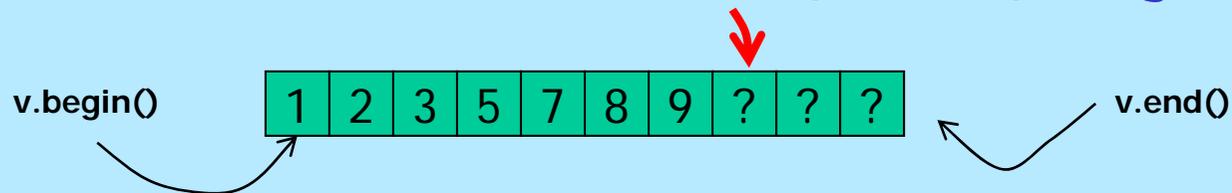
Löschende Algorithmen

ACHTUNG: Größe und Ende des manipulierten Containers ändert sich NICHT! Ergebnis ist immer das neue Ende des Containers!

Scott Meyers: *Effective STL* Item 32 (pp.139): "Follow **remove**-like Algorithms by **erase** if you really want to remove something."



```
vector<int>::iterator newEnd(remove(v.begin(), v.end(), 99));
```



```
v.erase((remove(v.begin(), v.end(), 99), v.end())); // !!!
```

3. Generische Programmierung in C++

Löschende Algorithmen

ACHTUNG: Auch wenn damit die logische Konsistenz wiederhergestellt ist, wird bei Vektoren kein Speicherplatz zurückgegeben!

Szenario: Sammle Kandidaten K in einem `vector<K> k` (~100.000) treffe Auswahl (die besten 10) entferne und lösche den Rest: der Vektor belegt immer noch 100.000 Elemente im Speicher :-)

Scott Meyers: *Effective STL* Item 17 (pp.77): "Use » the `swap` trick« to trim excess capacity."

`vector<K>(k).swap(k); // wie bitte ???`

```
// {  
//     vector<K> tmp(k); // copy-ctor erzeugt so viele elemente wie nötig  
//     tmp.swap(k); // k ist der neue (reduzierte) Vektor, tmp wird freigegeben  
// }
```

3. Generische Programmierung in C++

Mutierende Algorithmen (`#include <algorithm>`)

<code>reverse</code>	in einem Bereich die Reihenfolge aller Elemente umkehren
<code>reverse_copy</code>	aus einem Bereich alle Elemente in umgekehrter Reihenfolge an einen Bereichsanfang kopieren
<code>rotate</code>	einen Bereich so rotieren, dass ein neuer Bereichsanfang zum ersten Element wird
<code>rotate_copy</code>	einen Bereich rotiert, an neuem Bereichsanfang an einen Bereichsanfang kopieren
<code>next_permutation</code>	in einem Bereich die nächste Permutation erzeugen, liefert false, wenn letzte (lexikogr. aufsteigende) erreicht ist, sonst true
<code>prev_permutation</code>	in einem Bereich die vorhergehende Permutation erzeugen, liefert false, wenn letzte (lexikogr. absteigende) erreicht ist, sonst true
<code>random_shuffle</code>	einen Bereich zufällig nach einem impliziten oder expliziten Zufallsgenerator 'verwürfeln'
<code>partition</code>	in einem Bereich alle Elemente, für die ein Prädikat gilt, nach vorn schieben, Ergebnis ist die erste Position, an der das Prädikat nicht gilt
<code>stable_partition</code>	in einem Bereich alle Elemente, für die ein Prädikat gilt, nach vorn schieben, Ergebnis ist die erste Position, an der das Prädikat nicht gilt, Reihenfolgen in Teilmengen bleibt erhalten

3. Generische Programmierung in C++

Sortierende Algorithmen (`#include <algorithm>`)

<code>sort</code>	einen Bereich nach <code><</code> oder einer Relation sortieren
<code>stable_sort</code>	einen Bereich nach <code><</code> oder einer Relation sortieren, gleiche Elemente bleiben in relativer Position zueinander !
<code>partial_sort</code>	einen Bereich bis zu einer Position nach <code><</code> oder einer Relation sortieren
<code>partial_sort_copy</code>	aus einem Bereich bis zu einer Position nach <code><</code> oder einer Relation sortiert in einen anderen Bereich kopieren
<code>nth_element</code>	einen Bereich nach <code><</code> oder einer Relation um eine Position sortieren, so dass diese an der richtigen Stelle steht (links <code><=</code> , rechts <code>></code>)
<code>make_heap</code>	einen Bereich nach <code><</code> oder einer Relation in einen Heap umwandeln ($\forall i : h[i] > h[2*i+1] \text{ und } h[i] > h[2*i+2]$)
<code>push_heap</code>	das letzte Element eines Bereichs <code>[f, l)</code> in einen Heap <code>[f, l-1)</code> zu einem Heap <code>[f, l)</code> einbauen (<code>make_heap</code> , <code>push_back</code> , <code>push_heap</code>) (nach <code><</code> oder einer Relation)
<code>pop_heap</code>	das erste (nach <code><</code> oder einer Relation größte) Element mit dem letzten vertauschen und einen neuen Heap in <code>[f, l-1)</code> erzeugen
<code>sort_heap</code>	einen Heap nach <code><</code> oder einer Relation sortieren (Ergebnis ist bei <code><</code> kein Heap mehr) $O(N \cdot \log(N))$

3. Generische Programmierung in C++

Algorithmen für sortierte Bereiche (`#include <algorithm>`)

- `lower_bound`** in einem (nach `<` oder einer Relation) sortierten Bereich die erste Position ermitteln, an der ein Wert sortiert nach `<` oder einer Relation eingefügt werden kann
- `upper_bound`** in einem (nach `<` oder einer Relation) sortierten Bereich die letzte Position ermitteln an der ein Wert sortiert nach `<` oder einer Relation eingefügt werden kann
- `equal_range`** in einem nach `<` oder einer Relation sortierten Bereich die erste und letzte Position für sortiertes Einfügen als `pair<FwdIter, FwdIter>` ermitteln
- `binary_search`** ist ein Wert in einem nach `<` oder einer Relation sortierten Bereich enthalten ?
- `includes`** ist ein nach `<` oder einer Relation sortierter Bereich in einem anderen nach `<` oder einer Relation sortierten Bereich enthalten ?

3. Generische Programmierung in C++

Algorithmen für sortierte Bereiche (`#include <algorithm>`)

- `merge`** zwei nach `<` oder einer Relation sortierte Bereiche an einen Bereichsanfang sortiert mischen (Platz muss ausreichen!)
- `inplace_merge`** zwei nach `<` oder einer Relation sortierte Teilbereiche die direkt hintereinander liegen werden sortiert gemischt (`f1, l1f2, l2`)
- `set_union`** zwei nach `<` oder einer Relation sortierte Bereiche an einen Bereichsanfang sortiert mischen, Elemente, die in beiden vorkommen werden nur einmal erfasst
- `set_intersection`** aus zwei nach `<` oder einer Relation sortierten Bereichen wird die Schnittmenge an einen Bereichsanfang übertragen
- `set_difference`** aus zwei nach `<` oder einer Relation sortierten Bereichen wird die Differenzmenge (im ersten und nicht im zweiten) an einen Bereichsanfang übertragen
- `set_symmetric_difference`** aus zwei nach `<` oder einer Relation sortierten Bereichen wird die Komplementärmenge (im ersten und nicht im zweiten oder umgekehrt) an einen Bereichsanfang übertragen

3. Generische Programmierung in C++

Numerische Algorithmen (`#include <numeric>`)

`accumulate` bildet die Summe (bzw. Anwendung eines Operators) eines Initialwertes mit allen Elementen eines Bereichs

`partial_sum` bildet die Summe (bzw. Anwendung eines Operators) aller Anfangsstücke eines Bereichs und kopiert diese ab einem Bereichsanfang

`adjacent_difference` bildet die Differenzen (bzw. Anwendung eines Operators) jeweils benachbarter Elemente eines Bereichs und kopiert diese ab einem Bereichsanfang (erstes Element wird übernommen)

`inner_product` bildet die Summe (bzw. Anwendung eines Operators) aus einem Initialwert und dem Skalarprodukt (bzw. Anwendung eines weiteren Operators) eines Bereichs und eines Bereichsanfangs

3. Generische Programmierung in C++

Folgende Containertypen existieren:

Typ	Header	Charakteristik	Iteratoren
<code>vector</code>	<code><vector></code>	dynamische Felder mit wahlfreiem Zugriff, erweiterbar am Ende	<code>RandomAccess</code>
<code>deque</code>	<code><deque></code>	dynamische Felder mit wahlfreiem Zugriff, erweiterbar am Anfang und am Ende	<code>RandomAccess</code>
<code>list</code>	<code><list></code>	doppelt verkettete Listen ohne wahlfreiem Zugriff, nicht sortiert	<code>Bidirectional</code>
<code>set</code>	<code><set></code>	Mengen mit impliziter Sortierung, keine Duplikate	<code>Bidirectional</code>
<code>multiset</code>	<code><set></code>	Mengen mit impliziter Sortierung, Duplikate erlaubt	<code>Bidirectional</code>
<code>map</code>	<code><map></code>	Mengen von Schlüssel/Wert- Paaren mit impliziter Sortierung nach dem Schlüssel, keine Duplikate	<code>Bidirectional</code>
<code>multimap</code>	<code><map></code>	Mengen von Schlüssel/Wert- Paaren mit impliziter Sortierung nach dem Schlüssel, Duplikate erlaubt	<code>Bidirectional</code>

3. Generische Programmierung in C++

gemeinsame (generische) Operationen aller Container-Klassen:

1. Erzeugung und Zerstörung

Ausdruck	Bedeutung
<code>Container<Typ> m</code>	erzeugt einen leeren Container ohne Elemente
<code>Container<Typ> m1 (m2)</code>	erzeugt einen Container als Kopie eines anderen (gleichen Typs)
<code>Container<Typ> m (f, l)</code> <i>f wie first</i> <i>l wie last</i>	erzeugt einen Container und initialisiert ihn mit Kopien der Elemente aus dem Bereich [f, l) <i>member template</i> : beliebige Quell-Container (auch anderer Typen)!
<code>m.~Container<Typ> ()</code>	löscht alle Elemente und gibt deren Speicherplatz frei

3. Generische Programmierung in C++

gemeinsame (generische) Operationen aller Container-Klassen:

2. Nichtverändernde Operationen

Ausdruck	Bedeutung
<code>m.size ()</code>	aktuelle Anzahl von Elementen
<code>m.empty ()</code>	leer? (entspricht <code>m.size()==0</code> ABER SCHNELLER!)
<code>m.max_size ()</code>	maximal mögliche Anzahl von Elementen
<code>m1 == m2</code> <code>m1 != m2</code> <code><, >, <=, >=</code>	Gleichheit (<code>==</code> auf alle Elemente angewendet) Ungleichheit (dito) lexikografische Ordnung

3. Generische Programmierung in C++

gemeinsame (generische) Operationen aller Container-Klassen:

3. Zuweisende Operationen

Ausdruck	Bedeutung
<code>m1 = m2</code>	Zuweisung
<code>m1.swap (m2)</code>	Vertauschen aller Elemente
<code>swap (m1, m2)</code>	!! immer schneller als <code>m1 = m2</code>

3. Generische Programmierung in C++

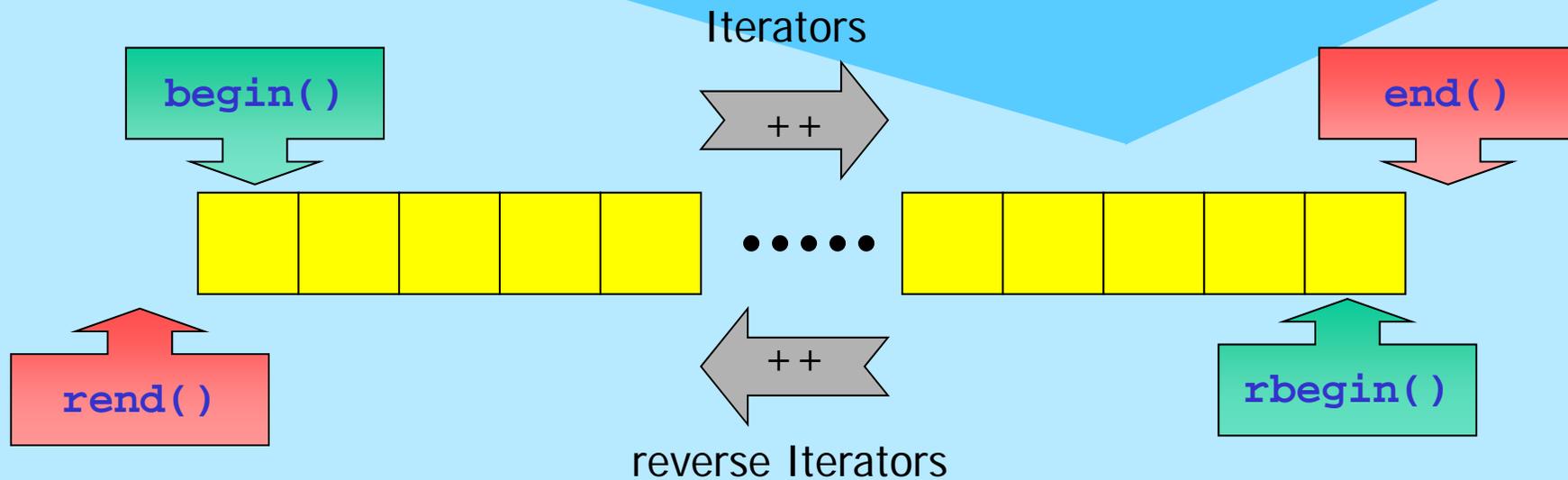
gemeinsame (generische) Operationen aller Container-Klassen:

4. Zugreifende Operationen

Ausdruck	Bedeutung
<code>m.begin ()</code>	Iterator auf das erste Element
<code>m.end ()</code>	Iterator hinter das letzte Element
<code>m.rbegin ()</code>	Iterator auf das letzte Element
<code>m.rend ()</code>	Iterator vor das erste Element

3. Generische Programmierung in C++

Abstrakte Iteratoren



alle Operationen liefern sog. *const iterators* für konstante Container
(über solche ist der Container nicht veränderbar)

3. Generische Programmierung in C++

gemeinsame (generische) Operationen aller Container-Klassen:

5. Einfügende/Löschende Operationen

Ausdruck	Bedeutung
<code>m.insert (pos, e)</code>	Kopie von e bei pos einfügen (Rückgabewert und Bedeutung von pos hängen von Containertyp ab)
<code>m.erase (pos)</code>	löscht Element an der Position pos
<code>m.erase (f, l)</code>	löscht alle Elemente aus dem Bereich [f, l) liefert die Position des Folgeelements
<code>m.clear ()</code>	leert den Container

3. Generische Programmierung in C++

Wichtige Eigenschaften der Container

- die Effizienz aller Operationen ist optimiert, der Standard macht Vorgaben zu oberen Grenzen
- die verfügbaren Implementationen orientieren sich am *state of the art* in ihrer algorithmischen Umsetzung
- die generischen (container-unabhängigen) Algorithmen sind in `<algorithm>` definiert
- häufig gibt es für spezielle Container bessere Algorithmen als Memberfunktionen (z.B. `set<T>::find`)
- nicht alle Kombinationen von Algorithmen und Containern sind möglich (sinnvoll) (z.B. `sort` auf `set`: Mengen sind bereits sortiert, `sort` auf `list` ist nicht möglich)
- Alle Container arbeiten mit einer Wertesemantik, d.h. Elemente werden immer kopiert !

3. Generische Programmierung in C++

Member-Algorithmen sind besser als globale:

```
#include <set>
#include <iostream>
#include <algorithm>
#include "Timer.h"
using namespace std;
typedef set<int> Set;

main() {
    Set s;  int probe = rand();
    {
        cout<<"Generating 1.000.000 nodes: "<<flush;
        Timer t;
        for(int i=0; i<500000; ++i) s.insert(rand());
        s.insert(probe);
        for(int i=1; i<500000; ++i) s.insert(rand());
        cout<<s.size()<<" Elemente in der Menge"<<endl;
    }
}
```

```
#include <ctime>
#include <iostream>
const double millis= 1000000.0;
class Timer { long t_;
    void report()
        { std::cout<<t_/millis<<"s"<<std::endl; }
public:
    Timer(): t_(clock()){ }
    ~Timer(){ t_=clock()-t_;report(); }
};
```

Timer.h

3. Generische Programmierung in C++

Member-Algorithmen sind besser als globale:

```
{
    cout<<"s.find(...): "<<flush;
    Timer t;
    for (int i=0; i<1000; ++i) s.find(probe); // member
}
{
    cout<<"find (...): "<<flush;
    Timer t;
    for (int i=0; i<1000; ++i)
        find(s.begin(), s.end(), probe); // generic
}
}
```

```
Generating 1.000.000 nodes:
999752 Elemente in der Menge
5.22s
s.find(...): 0s
find (...): 346.93s !!!
```

3. Generische Programmierung in C++

nicht alle Kombinationen von Algorithmen und Containern sind möglich:

```
★ // vector, algorithm, iostream
using namespace std;
typedef vector<int> V;

template <class CT>
void out(const CT& c) {
    for(typename CT::const_iterator i=c.begin(); i!=c.end(); ++i)
        std::cout<<*i<<' ';
    std::cout<<std::endl;
}

int main() {
    V v;
    for(int i=0; i<10; ++i) v.push_back(rand());
    out(v);
    sort(v.begin(), v.end());
    out(v);
}
```

3. Generische Programmierung in C++

nicht alle Kombinationen von Algorithmen und Containern sind möglich: **11.c**

★ // list, algorithm, iostream

```
using namespace std;
```

```
typedef list<int> L;
```

```
template <class CT>
```

```
void out(const CT& c) {
```

```
    for(typename CT::const_iterator i=c.begin(); i!=c.end(); ++i)
```

```
        std::cout<<*i<<' ';
```

```
    std::cout<<std::endl;
```

```
}
```

```
int main() {
```

```
    L l;
```

```
    for(int i=0; i<10; ++i) l.push_back(rand());
```

```
    out(l);
```

```
    sort(l.begin(), l.end()); // ??? list hat keine random access iteratoren!
```

```
    out(l);
```

```
}
```

l.sort(); // OK

3. Generische Programmierung in C++

nicht alle Kombinationen von Algorithmen und Containern sind möglich:

```
mio ahrens 72 ( c++/experimente ) > make ll
/usr/include/g++/stl_algo.h: In function `void sort<_List_iterator<int,int
&,int
*> >(_List_iterator<int,int &,int *>, _List_iterator<int,int &,int *>)':
ll.cc:17:   instantiated from here
/usr/include/g++/stl_algo.h:1320: no match for `_List_iterator<int,int &,int
*>
& - _List_iterator<int,int &,int *> &'
.....
mio ahrens 73 ( c++/experimente ) > ~/STLfilt/gfilt ll.cc
BD Software STL Message Decryptor v2.31 for gcc (03/03/2003)
stl_algo.h: In function
`void sort<list<int>::iter>(list<int>::iter, list<int>::iter)':
ll.cc:17:   instantiated from here
stl_algo.h:1320: no match for `list<int>::iter & - list<int>::iter &'
[STL Decryptor: Suppressed 14 more STL standard header messages]
```