C++ Standard Template Library

Pataki Norbert

2014. április 2.

Generic programming, generikus programozás

- Expression problem
- OOP vs. generic programming
- Párhuzamos bővíthetőség
- Lifting abstraction

C++ Standard Template Library

- Konténerek
- Algoritmusok
- Iterátorok
- Átalakítók
- Funktorok
- Allokátorok

Iterátor invalidáció

```
std::vector<double> v;
std::multiset<int> s;
std::list<std::string> ls;
// ...
std::vector<double>::iterator i = v.begin();
std::multiset<int>::const_iterator ci = s.begin();
std::list<std::string>::iterator li = ls.begin();
```

- Meddig hivatkozik az iterátor az ottlévő elemre?
- Meddig hivatkozik az iterátor érvényes tárterületre?

Konténerek

	vector	deque	list	asszoc.
felép.	din.tömb	fix méretű tömbök	láncolt lista	P-F
Iterátor	Random	Random	BiDir	BiDir
Invalid	Reallokáció	(*)	Soha	Soha

- (*):
  - beszúráskor fel kell tenni, hogy az iterátorok érvénytelenné válnak
  - a deque közepéből történő törléskor fel kell tenni, hogy az iterátorok érvénytelenné válnak
  - a deque valamely széléről történő törlés csak a törölt iterátort érvényteleníti
  - a deque iterátorai érvénytelenné válhatnak anélkül, hogy az elemeire hivatkozó referenciák és pointerek is invalidálódnának

Iterátor invalidáció

```
std::list<int> n;
// ...
for( std::list<int>::iterator i = n.begin();
     i != n.end();
     ++i )
{
  if ( 0 > *i )
  {
    n.erase( i );
  }
}
```

Iterátor invalidáció

```
std::list<int> n;
// ...
for( std::list<int>::iterator i = n.begin();
     i != n.end(); )
{
  if ( 0 > *i )
  {
    i = n.erase( i );
  }
  else
  {
    ++i;
  }
}
```

Iterátor invalidáció

```
std::set<int> s;
// ...
for( std::set<int>::iterator i = s.begin();
     i != s.end(); )
{
  if ( 0 > *i )
  {
    s.erase( i++ ); // C++11: iterator-t ad vissza
  }
  else
  {
    ++i;
  }
}
```

vector reallokáció

- Eljárás:
  1. új memóriaterület allokálása (ált. kétszeres kapacitással)
  2. elemek átmásolása
  3. régi elemek megszüntetése
  4. régi memóriaterület deallokálása
- költség, invalid iterátorok
- a kapacitás nem csökken automatikusan

vector::reserve

```
const int N = 1000;
// reserve nélkül
vector<int> v;
for(int i = 0; i < N; ++i)
  v.push_back( i );

// reserve-vel
vector<int> v;
v.reserve( N );
for(int i = 0; i < N; ++i)
  v.push_back( i );
```

Kapacitás csökkentése

- A vector és a string kapacitása nem csökken automatikusan
- Felesleges memóriafoglalás
- „Swap-trükk”:

```
vector<int> v;
...
vector<int>( v ).swap( v );

string s;
...
string( s ).swap( s );
```

- „Minimális” kapacitás
- C++11: `v.shrink_to_fit();`

vector<bool>

- teljes specializáció (Szabvány)
- nem bool-okat tárol, tömörebb ábrázolásmód
- Szabvány szerint az operator[]-nak vissza kell adnia egy referenciát az adott elemre
- reference egy belső proxy osztály, szimulálja a referenciákat
- reference-t ad vissza az operator[]

```
vector<bool> v;
...
bool* p = &v[0]; // ???
```

vector vs. string

- az interface-ük eltér
- `string`: specifikus tagfüggvények
- `string` másolás: használhat referencia-számlálást
- `vector<char>`, ha szükséges

Include-ok

Bizonyos STL implementációk esetén lefordul:

```
#include <vector>
//#include <algorithm>
// ...
vector<int> v;
int x;
// ...

vector<int>::iterator i =
  find( v.begin(), v.end(), x );
```

- Nincs definiálva melyik fejállomány melyik másikat használja.
- Hordozhatósági problémákat vet fel.
- Hogyan kerülhető el a hiba?

Iterátorok

- `container<T>::iterator`
- `container<T>::const_iterator`
- `container<T>::reverse_iterator`
- `container<T>::const_reverse_iterator`
- `istream_iterator<T>`, `ostream_iterator<T>`
- `istreambuf_iterator<T>`, `ostreambuf_iterator<T>`
- inserter iterátorok

Melyik iterátort használjuk?

- container<T>::iterator:
  - iterator vector<T>::insert(iterator position,
    const T& x);
  - iterator vector<T>::erase(iterator position);
  - iterator vector<T>::erase(iterator first,
    iterator last);

Iterátorok konverziója

- van `iterator` → `const_iterator` konverzió
- nincs `const_iterator` → `iterator` konverzió...
- megoldási lehetőség:

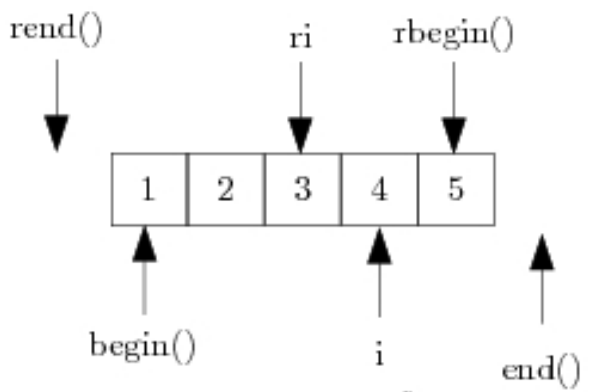
```
typedef deque<int>::iterator Iter;
typedef deque<int>::const_iterator ConstIter;
deque<int> d;
ConstIter ci;
...
Iter i = d.begin();
advance( i, distance<ConstIter>( i, ci ) );
```

- hatékonyság?

Iterátorok konverziója

- `reverse_iterator` $\longrightarrow$ `iterator` konverzió: `base()` tagfüggvénnyel
- Probléma: hova mutat az az iterator, amit a base visszaad?

```
i = ri.base();
```



Iterátorok konverziója

- Jó-e az az iterator, amit a base tagfüggvény visszaad?
  - Beszúráskor: OK
  - Törléskor: NEM!

```
Container<T> c;
...
Container<T>::reverse_iterator ri = ...

// Ez nem minden esetben fordul le:
c.erase( --ri.base() );

// Ez mindig jó:
c.erase( (++ri).base() );
```

Spec. iterátorok

```
// másoljuk a std.input-ot std.output-ra...

copy( istreambuf_iterator<char>( cin ),
      istreambuf_iterator<char>(),
      ostreambuf_iterator<char>( cout ) );
```

Inserterek motivációja

```
const int N = 10;
double v[N];
...
double cv[N];

// v másolása cv-be:
copy( v, v + N, cv );

// ez általában nem működik a valódi STL
// konténerek esetében...
```

copy implementációja

```
template <class InIt, class OutIt>
OutIt copy( InIt first, InIt last, OutIt dest )
{
  while ( first != last )
  {
    *dest++ = *first++;
  }
  return dest;
}
```

back_inserter

```
double f( double x )
{
...
}

list<double> values;
...
vector<double> results;
results.reserve( values.size() );

// f-et alkalmazzuk values összes elemére, és az
// adatokat results végére szúrjuk:
transform( values.begin(), values.end(),
           back_inserter( results ), f );
```

front_inserter

```
double f( double x )
{
...
}

list<double> values;
...
list<double> results;

// f-et alkalmazzuk values összes elemére, és
// az adatokat results elejére szúrjuk:
transform( values.begin(), values.end(),
           front_inserter( results ), f );
```

inserter

```
double f( double x )
{
...
}

list<double> values;
...
vector<double> results;
results.reserve( values.size() );

// f-et alkalmazzuk values összes elemére, és az
// adatokat results közepére szúrjuk:
transform(
  values.begin(), values.end(),
  inserter( results,
            results.begin() + results.size() / 2 ),
  f );
```

inserter

```
double f( double x )
{
...
}

list<double> values;
...
multiset<double> results;

// f-et alkalmazzuk values összes elemére, és
// az adatokat results-ba szúrjuk (rendezett lesz)
transform( values.begin(), values.end(),
           inserter( results, results.begin() ), f );
```

back_inserter implementációja (vázlat)

```
template <class Cont>
class back_insert_iterator
{
  Cont* c;
public:
  explicit back_insert_iterator( Cont& cont ): c( &cont )
  { }

  back_insert_iterator& operator++()
  {
    return *this;
  }

  back_insert_iterator& operator++( int )
  {
    return *this;
  }
```

back_inserter implementációja (vázlat)

```
  back_insert_iterator&
  operator=( typename Cont::const_reference d )
         // const typename Cont::value_type& d
  {
    c->push_back( d );
    return *this;
  }

  back_insert_iterator& operator*()
  {
    return *this;
  }
};
```

back_inserter implementációja (vázlat)

```
template <class Cont>
back_insert_iterator back_inserter( Cont& c )
{
  return back_insert_iterator<Cont>( c );
}
```

Funktorok

- Olyan objektumok, amelyeknek van operator()-a – (globális) függvényhívások szimulálása
- Objektumok - állapotok, adattagok, egyéb tagfüggvények
- Hatékonyság (inline)
- Speciális hibák elkerülése
- Ősosztályok : unary_function, binary_function sablonok.
- Különös elvárások
- C++11: Lambda

unary_function, binary_function

- speciális typedef-eket biztosítanak (alkalmazkodóképesség)
- ezekre a typedef-ekre szükségük van a függvényobjektum adaptereknek (`not1`, `not2`, `bind1st`, `bind2nd`)
- `unary_function`:
  - `argument_type`
  - `result_type`
- `binary_function`:
  - `first_argument_type`
  - `second_argument_type`
  - `result_type`

unary_function, binary_function

- A `unary_function` és a `binary_function` sablon osztályok
- Származtatni példányokból kell
- `unary_function`: 2 sablon paraméter
- `binary_function`: 3 sablon paraméter
- az utolsó paraméter mindig az `operator()` visszatérési értéknek ,,megfelelő" típus
- az első paraméter(ek)nek az `operator()` paramétere(i)nek ,,megfelelő" típus(ok)

Példák

```
struct IntGreater: binary_function<int, int, bool>
{
  bool operator()( const int& a, const int& b ) const
  {
    return b < a;
  }
};

struct CPtrLess:
  binary_function<const char*, const char*, bool>
{
  bool operator()( const char* a, const char* b ) const
  {
    return strcmp( a, b ) < 0;
  }
};
```

Példák

```
class Sum: public unary_function<int, void>
{
  int partial_sum;
public:
  Sum(): partial_sum( 0 ) { }
  void operator()( int i )
  {
    partial_sum += i;
  }

  int get_sum() const
  {
    return partial_sum;
  }
};
```

Függvények alkalmazkodóképessége

- függvények + adapterek? (pl. predikátumfüggvény negálása)
- direktben nem megy, mert hiányoznak a typedef-ek...
- ptr_fun visszaad egy alkalmazkodóképes funktort. Ennek az operator()-a meghívja az eredeti függvényt.
- példa:

```
bool is_even( int x );

// Első páratlan szám megkeresése:
list<int> c;
...
list<int>::iterator i =
  find_if( c.begin(), c.end(),
           not1( ptr_fun( is_even ) ) );
```

mem_fun, mem_fun_ref

```
template <class InputIter, class Fun>
Fun for_each( InputIter first, InputIter last, Fun f )
{
  while( first != last )
    f( *first++ );
  return f;
}
...
struct Foo
{
  void bar();
};

list<Foo> c;
// bar meghívása c összes elemén?
```

mem_fun, mem_fun_ref

```
for_each( c.begin(), c.end(), mem_fun_ref( &Foo::bar ) );
...
struct Figure
{
  virtual void draw() const = 0;
  virtual ~Figure() { }
};
...
list<Figure*> c;
// draw meghívása (a dinamikus típusnak megfelelő):
for_each( c.begin(), c.end(), mem_fun( &Figure::draw ) );
```

mem_fun_ref – Impl.

```
template <class Ret, class T>
class mem_fun_ref_t: public unary_function<T, Ret>
{
  Ret ( T::*fptr )();
public:
  explicit mem_fun_ref_t( Ret ( T::*f )() ) : fptr( f )
  { }
  Ret operator()( T& t ) const { return ( t.*fptr )(); }
};

template <class Ret, class T>
inline mem_fun_ref_t<Ret, T> mem_fun_ref( Ret ( T::*f )() )
{
  return mem_fun_ref_t<Ret, T>( f );
}
```

Különös elvárások

- Érték-szerinti paraméterátadás:
  - másolás (copy ctor) – költséges lehet
  - nem viselkedhet polimorfikusan (virtuális függvények)
  - pimpl-idiom
  - explicit specializáció?
- Predikátum funktor ill. függvény ne érjen ill. ne tartson fenn olyan változót, amely befolyásolhatja az operator() eredményét (azaz, az operator() értéke mindig csak a paramétereitől függjön). Ez amiatt fontos, mert az algoritmusok másolatokat készíthetnek a funktorból.

Predikátumok másolása az STL-en belül

```
template <typename FwdIterator, typename Predicate>
FwdIterator remove_if( FwdIterator begin,
                       FwdIterator end,
                       Predicate p )
{
  begin = find_if( begin, end, p );
  if ( begin == end ) return begin;
  else
  {
    FwdIterator next = begin;
    return remove_copy_if( ++next, end, begin, p );
  }
}
```

Hibás predikátum

```
class BadPredicate: public unary_function<Widget, bool>
{
  size_t timesCalled;
public:
  BadPredicate(): timesCalled( 0 ) { }

  bool operator()( const Widget& )
  {
    return ++timesCalled==3;
  }
};
// ...
std::list<Widget> c;
// ...
c.erase( remove_if( c.begin(), c.end(), BadPredicate() ),
         c.end() ); // Mit töröl?
```

Asszociatív konténerek

- Szabványosak:
  - set
  - multiset
  - map
  - multimap
- C++11:
  - unordered_set
  - unordered_multiset
  - unordered_map
  - unordered_multimap

Jellemzők

- Rendezettség, összehasonlító típus
- Keresőfa, piros-fekete fa
- Ekvivalencia
- Logaritmikus bonyolultságú műveletek

Ekvivalencia vs. egyenlőség

- K: Mikor egyezik meg két elem?
  - V1: Egyenlőség – `operator==` (pl. `find` algoritmus)
  - V2: Ekvivalencia – pl. asszociatív konténerek, rendezett intervallumok kereső algoritmusai (pl. `equal_range`)
- K: Igaz-e, hogy a kettő megegyezik?
  - V: Nem!
  - V: Például: case-insensitive string összehasonlítás

Ekvivalencia

```
// Spec. eset:
// a és b ekvivalensek, ha
( !( a < b ) && !( b < a ) )

// Általános eset:
( !(s.key_comp()( a, b ) ) &&
  !(s.key_comp()( b, a ) ) )
```

- A multiset / multimap esetében az ekvivalens elemek sorrendje nem definiált

Példa

```
struct StringSizeLess :
  binary_function<string, string, bool>
{
  bool operator()( const string& a,
                   const string& b ) const
  {
    return a.size() < b.size();
  }
};

multiset<string, StringSizeLess> a;
a.insert( "C++" );
cout << a.count( "ADA" ); // ?
```

Amire figyelni kell

```
template <class T,
          class Compare = less<T>,
          class Allocator = allocator<T> >
class set;

...

set<string*> ssp;
// ...
// Rendezettség?
```

Megoldás

```
// Összehasonlító típus:
struct StringPtrLess:
  binary_function<const string*, const string*, bool>
{
  bool operator()( const string* a,
                   const string* b ) const
  {
    return *a < *b;
  }
};

set<string*, StringPtrLess> ssp;
// ...
```

Felhasználói predikátumok vs. Ekvivalencia

```
// Próbáljuk meg "<=" alapján
// rendezni az elemeket:
set<int, less_equal<int> > s;

s.insert( 10 );
s.insert( 10 );

// Ellenőrzés: a 10 benne van-e már a konténerben?
!(10 <= 10) && !(10 <= 10): hamis
// s-ben 10 kétszer szerepel. Ennek ellenére
// s.count( 10 ) == 0
```

Felhasználói predikátumok vs. Ekvivalencia

- általánosan igaz, hogy megegyező értékekre a predikátumoknak hamisat kell visszaadniuk
- a multikonténereknél is
- különben a konténer inkonzisztenssé válik

Felhasználói típusok / felhasználói rendezések

```
class Employee
{
public:
  Employee( const std::string& name );
  const string& name() const;
  const string& get_title() const;
  void set_title( const std::string& title );
  ...
};
```

Felhasználói típusok / felhasználói rendezések

```
struct EmployeeCompare :
  std::binary_function<Employee,
                       Employee,
                       bool>
{
  bool operator()( const Employee& a,
                   const Employee& b ) const
  {
    return a.name() < b.name();
  }
};

std::multiset<Employee, EmployeeCompare> employees;
// ...
std::string name = "John Doe";
employees.find( name )->set_title( "vice president" );
```

Felhasználói típusok / felhasználói rendezések

- A `map` és a `multimap` a kulcsot konstansként kezeli, nem lehet megváltoztatni. A kulcshoz rendelt érték megváltoztatható.
- A `set` és a `multiset` esetében a tárolt értékek konstanssága implementáció-függő.
- A fenti kód portolhatósági hibákhoz vezet.
- C++11 előtt/után
- A konstansság és a nem-konstansság nem elég kifinomult felosztás. `mutable`?
- Szükség lenne ,,rendezés-megőrző" módosítóra?
- Elméletileg meghívható lehetne minden olyan metódus, ami nem változtatja meg a rendezéshez használt tagok értékét.
- Az `Employee` típus írója nem tudhatja előre, hogy milyen rendezéseket használnak később a típuson. (Példa: `Student` típus: magasság vagy név alapján, stb.)

Kereső algoritmusok

- ```
  template <class InputIterator, class T>
  typename
    iterator_traits<InputIterator>::difference_type
  count( InputIterator first, InputIterator last,
         const T& value );
  ```
- ```
  template <class InputIterator, class T>
  InputIterator find( InputIterator first,
                      InputIterator last,
                      const T& value );
  ```

Kereső algoritmusok

```
template <class ForwardIterator,
          class LessThanComparable>
bool binary_search( ForwardIterator first,
                    ForwardIterator last,
                    const LessThanComparable& value );

template <class ForwardIterator,
          class T,
          class StrictWeakOrdering>
bool binary_search( ForwardIterator first,
                    ForwardIterator last,
                    const T& value,
                    StrictWeakOrdering comp );
```

Kereső algoritmusok

- ```
  template <class ForwardIterator,
            class T,
            class SWeakOrdering>
  ForwardIterator lower_bound( ForwardIterator first,
                               ForwardIterator last,
                               const T& value,
                               SWeakOrdering comp );
  ```
- ```
  template <class ForwardIterator,
            class T,
            class SWeakOrdering>
  ForwardIterator upper_bound( ForwardIterator first,
                               ForwardIterator last,
                               const T& value,
                               SWeakOrdering comp );
  ```

Kereső algoritmusok

```
template <class ForwardIterator,
          class T,
          class StrictWeakOrdering>
pair<ForwardIterator, ForwardIterator>
equal_range( ForwardIterator first,
             ForwardIterator last,
             const T& value,
             StrictWeakOrdering comp );

// Fontos: ugyanazt a rendezési predikátumot kell
// használni a kereséshez és a rendezéshez...
```

A remove(_if) algoritmus

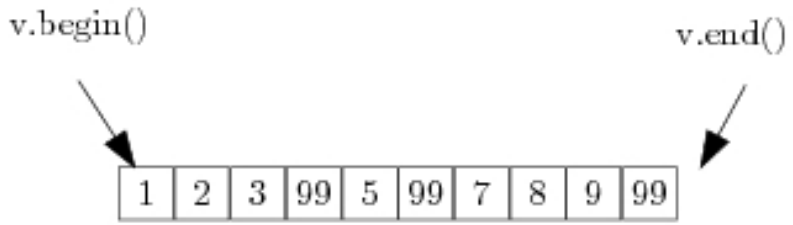
```
template <class FwdIter, class T>
FwdIter remove( FwdIter first, FwdIter last,
                const T& value);
```

- Mit csinál? Mit nem csinál?
- Miért?
- Hogyan csináljuk helyesen?
- Mire kell figyelni?

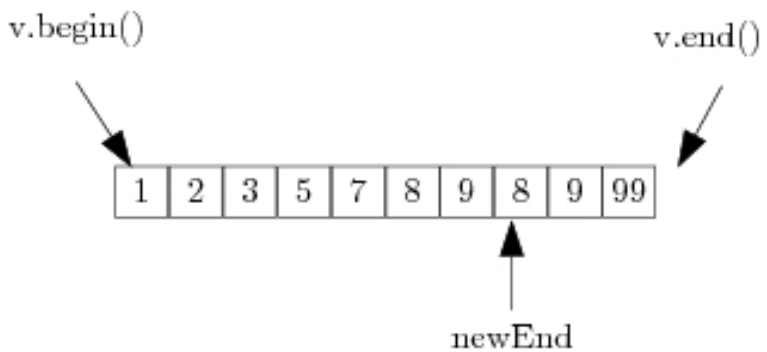
A remove(_if) algoritmus

- Nem töröl.
- Úgy rendezi át a konténer elemeit, hogy megmaradóak a konténer elején álljanak.
- Konténer végén (általában) ugyanazok az elemek állnak, mint az algoritmus meghívása előtt (partition algoritmus).
- Visszaad egy iterátort, ami az ,,új logikai végét" jelenti a konténernek.
- Az algoritmusok konténer-függetlenek. Nem tudják, mit jelent az, hogy ,,törölni". Az iterátoroktól nem lehet eljutni a konténerhez.
- `remove`, `remove_if`, `unique`
- `partition`

Példa



`remove( v.begin(), v.end(), 99 );` után:



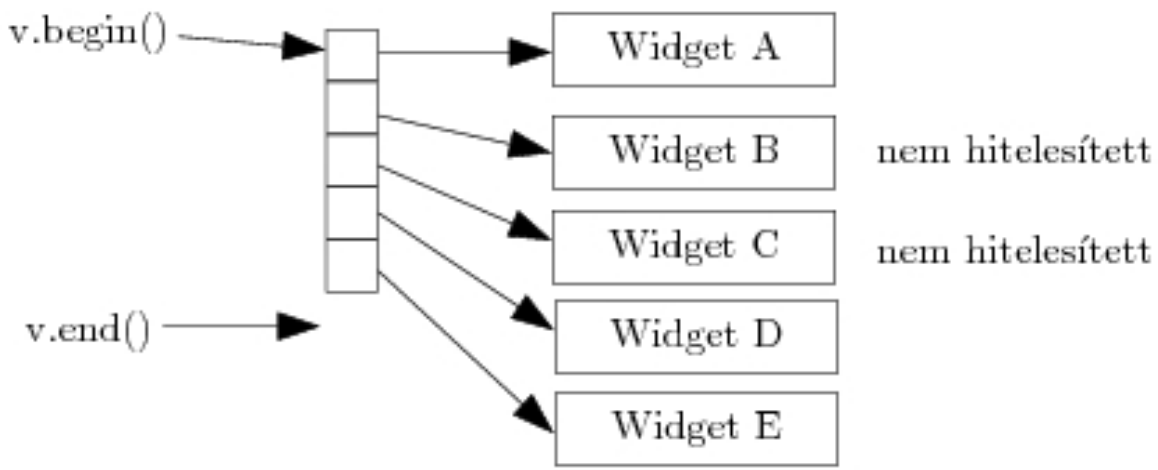
A remove helyes használata

```
c.erase( remove( c.begin(),
                 c.end(),
                 t ),
         c.end() );

list<int> li;
li.remove( t );
```

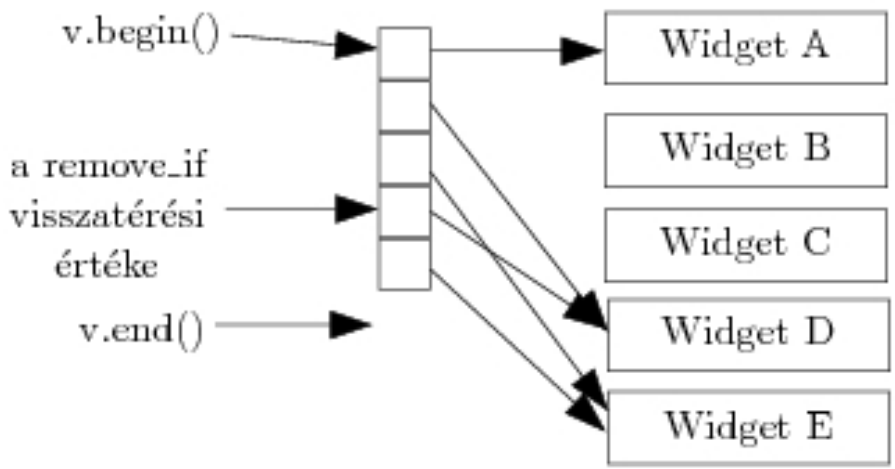
Erase-remove

```
// írhatunk egy ilyen segéd függvény sablont:

template <class Cont, class T>
void erase_remove( Cont& c, const T& t )
{
  c.erase( remove( c.begin(), c.end(), t ), c.end() );
}
```

memory leak – remove_if



memory leak – remove_if



memory leak elkerülése

- Hogyan kerüljük el a memory leaket?
- Végigmegyünk a konténeren, és ha eltávolítandó elemet találunk, akkor megszüntetjük a dinamikusan allokációt, és nullpointerre állítjuk a pointert.
- A remove-val eltávolítjuk a nullpointereket.
- Kikötés, hogy nem volt nullpointer a konténerben...

Algoritmusok vs. tagfüggvények

- Bizonyos konténerek esetében léteznek olyan tagfüggvények, amelyeknek a neve megegyezik egy algoritmus nevével
- AssocCont::find, AssocCont::count, AssocCont::equal_range, AssocCont::lower_bound, AssocCont::upper_bound
- list::remove, list::sort, list::unique, list::reverse
- Melyiket válasszuk? Miért?

Algoritmusok vs. tagfüggvények

- A tagfüggvényeket érdemes választani
- Hatékonyság, helyes működés, leforduló kódok
- Hatékonyság: AssocCont::find, AssocCont::count, list::remove
- Helyes működés: list::remove, AssocCont::count, AssocCont::find
- Leforduló kódok: list::sort

Algoritmusok vs. kézzel írt ciklusok

- Sokszor egyszerűbbnek tűnik kézzel írt ciklusokkal dolgozni algoritmusok helyett (nem kell funktorozni, mem_fun, binderek, stb.)
- Miért használjunk algoritmusokat?
- Hatékonyság, a nevek szemantikája miatt: nagyobb kifejezőerővel rendelkeznek...
- Speciális könyvtári optimalizációk, kevesebb kiértékelés...
- Érvénytelen iterátorok elkerülése...
- Bizonyos esetekben javít a kód átláthatóságán a kézzel írt ciklus, de ez ritka...

Példa

```
size_t fillArray( double *pArray, size_t arraySize );

double data[maxNumDoubles];

deque<double> d;
//...
size_t numDoubles =
  fillArray( data, maxNumDoubles );

for( size_t i = 0; i < numDoubles; ++i )
{
  d.insert( d.begin(), data[i] + 41 );
}
```

Példa

```
size_t fillArray( double *pArray, size_t arraySize );

double data[maxNumDoubles];

deque<double> d;
//...
size_t numDoubles =
  fillArray( data, maxNumDoubles );

deque<double>::iterator insertLocation = d.begin();

for( size_t i = 0; i < numDoubles; ++i )
  d.insert( insertLocation++, data[i] + 41 );
```

Példa

```
size_t fillArray( double *pArray, size_t arraySize );

double data[maxNumDoubles];
deque<double> d;
//...
size_t numDoubles =
  fillArray( data, maxNumDoubles );

deque<double>::iterator insertLocation = d.begin();

for( size_t i = 0; i < numDoubles; ++i )
{
  insertLocation =
    d.insert( insertLocation, data[i] + 41 );
  ++insertLocation;
}
```

Példa

```
transform( data, data + numDoubles,
           inserter( d, d.begin() ),
           bind2nd( plus<double>(), 41 ) );
```

iterator_traits

- Különböző információk kinyerése az iterátorokból

```
struct output_iterator_tag { };
struct input_iterator_tag  { };
struct forward_iterator_tag :
  input_iterator_tag { };
struct bidirectional_iterator_tag :
  forward_iterator_tag { };
struct random_access_iterator_tag :
  bidirectional_iterator_tag { };
```

iterator_traits

```
// Példa: pointerek. Hmmm... void*???
template <class T>
struct iterator_traits<T*>
{
    typedef T           value_type;
    typedef ptrdiff_t   difference_type;
    typedef random_access_iterator_tag
                        iterator_category;
    typedef T*          pointer;
    typedef T&          reference;
};
```

iterator_traits

```
// Alkalmazások:
template <class Iterator>
void algoritmus( Iterator first, Iterator last )
{
  typename
   iterator_traits<Iterator>::value_type o = *first;
  ...
}
```

iterator_traits

```
// Alkalmazások:
template <class Iter>
inline void foo( Iter beg, Iter end )
{
    foo( beg, end,
         typename
           iterator_traits<Iter>::iterator_category() );
}

template <class BiIterator>
void foo( BiIterator beg, BiIterator end,
          bidirectional_iterator_tag )
{
    // foo algoritmus implementációja
    // bidirectional iteratorokhoz
}
```

iterator_traits

```
template <class RaIterator>
void foo( RaIterator beg, RaIterator end,
          random_access_iterator_tag )
{
    // foo algoritmus implementációja
    // random_access iteratorokhoz
}
```

- distance
- advance

- `unordered_set`, `unordered_multiset`
- `unordered_map`, `unordered_multimap`
- `array`
- `forward_list`