Part 5: STL

Prof. Dr. Ulrik Schroeder

C++ - Einführung ins Programmieren

WS 03/04

http://learntech.rwth-aachen.de/lehre/ws04/c++/folien/index.html

Nachtrag: Typen, Klassen und Casts

- Bisher kennengelernt:

  Klassischer C-Cast operator:

  (TYPE)expression

  oder

  TYPE(expression)

- Aus vielerlei Gründen gefährlich in C++!
  - Frage: Was passiert, wenn der Cast fehlschlägt?
  - Antwort: Nichts! Naja, zumindest:
    - Keine Exception!
    - Fehler ist bei Verwendung des ge-casteten Objekts.
- C++ bringt neue cast-Operatoren:
  - dynamic_cast <new_type> (expression)
  - reinterpret_cast <new_type> (expression)
  - static_cast <new_type> (expression)
  - const_cast <new_type> (expression)

Class casts

- Up casts are safe and implicit
  - all heirs fullfill at least the contract of their base classes and only might add additional features
- Down Casts are sometimes necessary, but unsafe
  - e.g. treating specific objects from a container

```
int main( ) {
    Staff *s[ 3 ]; // dynamic: Staff, Employee, Manager
    // ... automatic Up Cast for heteregeneous filling
    for ( int i = 0; i < sizeof( s ) / sizeof( staff* ); i++ ) {
        Manager * m = (Manager *)( s[ i ] );
        m->do_something....
    } // for
} // main
```

unsafe static cast (C type) => no runtime check

dynamic_cast<TYPE>(expression)

- Up casts are safe and implicit
  - all heirs fullfill at least the contract of their base classes and only might add additional features
- Down Casts are sometimes necessary, but unsafe
  - e.g. treating specific objects from a container

```
int main( ) {
    Staff *s[ 3 ]; // dynamic: Staff, Employee, Manager
    // ... automatic Up Cast for heteregeneous filling
    for ( int i = 0; i < sizeof( s ) / sizeof( staff* ); i++ ) {
        Manager * m = dynamic_cast< Manager *>( s[ i ] );
        if ( m != NULL ) m->grantGratification( 2000 );
    } // for
} // main
```

results in NULL pointer, if not convertible to Manager*

Other cast-operators

- static_cast<TYPE>(expression)
  - casts pointers to related classes (up- and down)
  - Performs no checks! Thus, no guarantee if conversion was successful!
  - But: Faster than dynamic_cast!
  - Use only if you know, it can be casted!
  - Can also convert anything from and to void*
- reinterpret_cast<TYPE>(expression)
  - Converts any pointer type to any other pointer type!
  - Even of unrelated classes!
  - Converts pointers to ints, too.
- const_cast<TYPE>(expression)
  - manipulates the constness of the object pointed by a pointer!
  - Constness may be set or to be removed.
  - Use with care → const-ness sould be obeyed (in general)!

http://www.cplusplus.com/doc/tutorial/typecasting/

boost::lexical_cast<T>

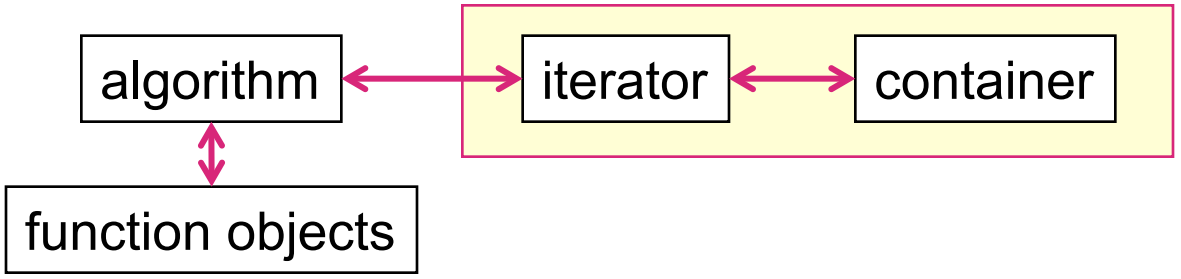
- For conversion between numbers and strings (two-ways)
- Do ALWAYS prefer over atoi etc.
- Raises exception in case of a „non-convertible“ variable.
- Example:

```
#include <boost/lexical_cast.hpp>

std::string text = „Dies ist ein Text“, nr_text=„2.0“;

try{

    int i = boost::lexical_cast<int>(nr_text),
        j = boost::lexical_cast<int>(text);

} catch (const boost::bad_lexical_cast & e) {

    std::cerr << „Error while casting to int“;

}
```

http://www.boost.org/doc/libs/1_58_0/doc/html/boost_lexical_cast.html

STL – standard template library

- abstraction of container types (generic), an iteration over these containers, and algorithms for searching (traversing), sorting, reversing, …(based on function objects for variation)



- implementation tuned for runtime efficiency (everything defined inline)
- STL is not really object oriented
  - algorithms implemented as global functions
  - container methods not defined as virtual => can not serve as base classes (at least not be used polymorphic

https://www.sgi.com/tech/stl/stl_introduction.html

Criteria for data structures (container)

- Access
  - to only one specific element (STACK, QUEUE)
  - sequential, neighborship (LIST)
  - indexed (ARRAY, VECTOR, MATRIX)
  - associative via access key (TABLE)
  - position in recursive structure (TREE, GRAPH)
- Traversal
  - Iteration in constant order: forward, backward
  - miscellaneous: Preorder, Inorder, Postorder, Breadth-first
- Adding elements
  - position, order
- Deleting
  - position: any, front, back,
- Storage
  - static, constant area, limited space
  - dynamic, growing and shrinking, virtually unlimited

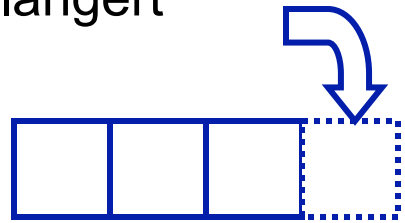
optimized

multiplication w/ elementsize

Many containers are hybrid / offer features from more than one container type (Linked list with index, Array with iterator, ...)

Einige Containerklassen

- Dynamisches Array: java.util.Vector => vector<T>
  - wird dynamisch durch Einfügen neuer Elemente verlängert
  - Zugriff auf jedes Element über Index



- Tabelle: java.util.Dictionary => map<T>
  - Abbildung eines Schlüsselwertes auf ein Objekt
  - verschiedene Implementierungen

Hashtable

key1	value1
key2	value2
...	...
keyy	valuey
keyz	valuez

Sequential container

- vector, list, deque etc.
  - common: iterators (to be used like pointer),
    - begin() : points to first, end(): behind last element [begin, end)
  - Random-access for vector: [ ] operator
  - Bidirectional: list, only ++, --, but insertion, deletion at any position

basic iteration looks the same for all

```
vector<int>::interator pos;
for ( pos = x.begin(); pos != x.end(); pos++)
    cout<<(*pos)<<" ";
```

for lists:

```
sort( l.begin(), l.end() ); // uses quicksort
l1.merge( l2 ); // splice, reverse, unique, ...
```

associative container

- set, map (unique objects/keys), multiset, multimap (non-unique)

```
typedef set<int> IntSet;
typedef IntSet::iterator SetIter;
int main( ) {
    IntSet  lotto;
    SetIter pos;
    srand();
    while( lotto.size() < 6) lotto.insert( 1 + rand()%49 );
    cout << "This week's winning numbers: ";
    for( pos = lotto.begin(); pos != lotto.end(); pos++)
        cout << *pos << "  ";
    return 0;
} // main( )
```

even if random number occurs twice …

Multi-Maps

```
typedef multimap<int, string> MULTI_MAP;
typedef MULTI_MAP::iterator ITERATOR;

int main( ) {
    MULTI_MAP  m;
    ITERATOR pos;
    m.insert(pair<int, string>(21931, "Nicole") );
    …
    pos = m.find( 21931 );
    if( pos != m.end( ) )
        cout << pos->first << "  " << pos->second << endl;
    …
    cout << "key " << 21931 << " exists " << m.count(21931) << "
times " << endl;
    return 0;
}
```

Algorithms and Functions

- Eventually, functional programming meets C++!
- Recall the for-loop over an int-vector:

```
vector<int> x(100);
... fill vector with data...
for (vector<int>::iterator pos = x.begin(); pos != x.end(); pos++)
    cout<<(*pos)<<" ";
```

even easier in C++11 → next week

- Mainly functional: Apply/map a function to all elements of x.

```
#include <algorithm>
void func(int i) {  // function:
  std::cout << ' ' << i;
}
 for_each (x.begin(), x.end(), func);
```

Think more generic...

- What if you want to write a generic vector-min function?

```
template <class T>
T vector-min(const & vector<T> values) {
    T min_val = 9999999;
    for (vector<T>::iterator pos = x.begin(); pos != x.end(); pos++)
      min_val = (min_val > *pos) ? *pos : min_val;
  return min_val;
}
```

- Problem: You need the maximum w.r.t. type T!
  - Differs for each type!
  - Since no class constraint in template definition
    → vector of vectors possible...

The solution: Type Traits!

- Type traits help here!
- Are defined as templates – no loss of execution speed:

```
template <class T>
T vector-min(const & vector<T> x) {
    T min_val = vigra::numericTraits<T>::max();
    for (vector<T>::iterator pos = x.begin(); pos != x.end(); pos++)
      min_val = (min_val > *pos) ? *pos : min_val;
  return min_val;
}
```

Also defined: min(), zero(), and type promotion, eg. int+float → float

- For this example:

  #include <vigra/numerictraits.hxx>

- For more general Traits, like std::is_arithmetic<T>:
  - #include <type_traits>

Algorithms and Functors

- What if the function to be applied needs some state for precessing the result? Recall the min-function!
- Bad solution: global variables – For many reasons!
- Better: Abstract functional behavior w.r.t. a class
- We call this class a functor!

```
#include <algorithm>
class Functor {          // function object type:
 public:
   void operator() (int i) {
     std::cout ++m_linenr << ': ' << i;
   }
private:
     int m_linenr;
};
```

```
...
Functor func;
 for_each (x.begin(), x.end(), func);
```

- Many functors/functions are already pre-defined:
  - plus, minus, multiplies, divides etc.
  - greater, greater_equal, less, less_equal
  - bit_*X*, logical_*X*, with *X* in:
    - and,
    - or,
    - not
- Functor/function binding (currying) is also supported:
  - before C++11: boost::bind, **now:** std::bind

```
#include <functional>
double my_divide (double x, double y) {return x/y;}
vector<double> xd;
...fill vector... and divide each element by two:
 for_each (xd.begin(), xd.end(), std::bind (my_divide,  _1,  2));
```

Further references:

- For casting:

  http://www.cplusplus.com/doc/tutorial/typecasting/

- For the STL-containers and iterators:

  http://www.cplusplus.com/reference/stl/

- For the functional programming like shown here:

  http://www.cplusplus.com/reference/algorithm/

  http://www.cplusplus.com/reference/functional/

- Functional Programming is getting easier in C++11 next week:
  - lambda-functions
  - automatic assignments
  - index ranging
  - etc...