The Standard Template Library

7 Standard Library II the Template Library

© 2006 Wouter Verkerke, NIKHEF

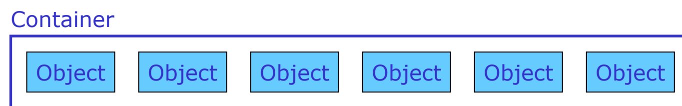
Introduction to STL

- STL = The **S**tandard **T**emplate **L**ibrary
  - A collection of template classes and functions for general use
  - Started out as experimental project by Hewlett-Packard
  - Now integral part of ANSI C++ definition of 'Standard Library'
  - Excellent design!
- Core functionality – Collection & Organization
  - Containers (such as lists)
  - Iterators (abstract methods to iterate of containers)
  - Algorithms (such as sorting container elements)
- Some other general-purpose classes
  - Classes string, complex, bits

© 2006 Wouter Verkerke, NIKHEF

Overview of STL components

- Containers
  - Storage facility of objects



- Iterators
  - Abstract access mechanism to collection contents
  - "Pointer to container element" with functionality to move pointer
- Algorithms
  - Operations (modifications) of container organization of contents
  - Example: Sort contents, apply operation to each of elements

© 2006 Wouter Verkerke, NIKHEF

293

STL Advantages

- STL containers are generic
  - Templates let you use the same container class with any class or built-in type
- STL is efficient
  - The various containers provide different data structures.
  - No inheritance nor virtual functions are used (we'll cover this shortly).
  - You can choose the container that is most efficient for the type of operations you expect
- STL has a consistent interface
  - Many containers have the same interface, making the learning curve easier
- Algorithms are generic
  - Template functions allow the same algorithm to be applied to different containers.
- Iterators let you access elements consistently
  - Algorithms work with iterators
  - Iterators work like C++ pointers
- Many aspects can be customized easily

© 2006 Wouter Verkerke, NIKHEF

294

Overview of STL containers classes

- Sequential containers (with a defined order)
  - `vector`
  - `list`
  - `deque` (**d**ouble-**e**nded **que**ue)

  Fundamental container implementations with different performance tradeoffs

  - `stack`
  - `queue`
  - `priority_queue`

  Adapters of fundamental containers that provide a modified functionality

- Associative containers (no defined order, access by key)
  - `set`
  - `multiset`
  - `map`
  - `Multimap`
  - `unordered_set, unordered_map (C++2011)`

© 2006 Wouter Verkerke, NIKHEF

Common container facilities

- Common operations on fundamental containers
  - **`insert`** – Insert element at defined location
  - **`erase`** – Remove element at defined location

  - **`push_back`** – Append element at end
  - **`pop_back`** – Remove & return element at end

  - **`push_front`** – Append element at front
  - **`pop_front`** – Remove & return element at front

  - **`at`** – Return element at defined location (with range checking)
  - **`operator[]`** – Return element at defined location (no range checking)

  - Not all operations exist at all containers (e.g. push_back is undefined on a set as there is no 'begin' or 'end' in an associative container)

© 2006 Wouter Verkerke, NIKHEF

Vector <vector>

- Vector is similar to an array

  - Manages its own memory allocation
  - Initial length at construction, but can be extended later
  - Elements initialized with default constructor
  - **Offers fast random access to elements**
  - Example

```
#include <vector>
vector<int> v(10) ;

v[0] = 80 ;
v.push_back(70) ; // creates v[10] and sets it to 70

vector<double> v2(5,3.14) ; // initialize 5 elements to 3.14
```

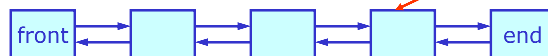
© 2006 Wouter Verkerke, NIKHEF

297

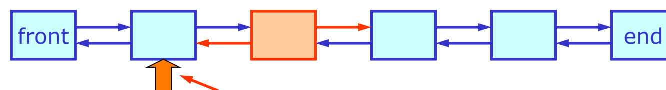
List <list>

```
Template<class T>
Struct ListElem {
  T elem ;
  ListElem* prev ;
  ListElem* next ;
}
```

- Implemented as doubly linked list



  - Fast insert/remove of in the middle of the collection



  - **No random access**
  - Example

iterator ‘pointer’ in collection

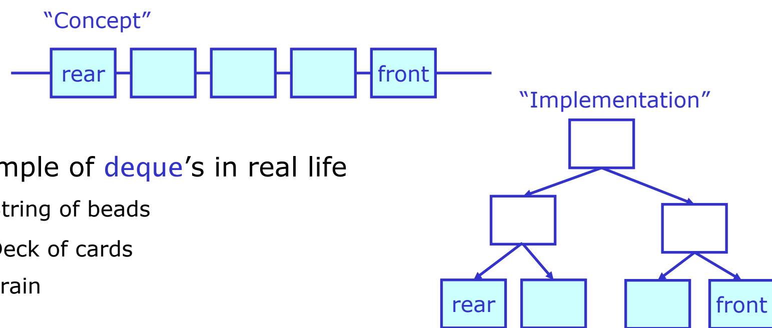
```
#include <list>
list<double> l ;
l.push_front(30.5) ; // append element in front
l.insert(somewhere,47.5) ; // insert in middle
```

© 2006 Wouter Verkerke, NIKHEF

298

Double ended queue <deque>

- Deque is sequence optimized for insertion at both ends
  - *Inserting at ends* is as efficient as `list`
  - *Random access* of elements efficient like `vector`



- Example of deque's in real life
  - String of beads
  - Deck of cards
  - Train

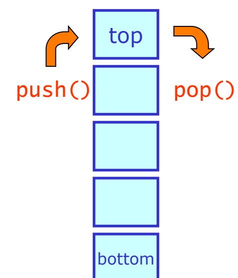
"Deque, it rhymes with 'check' (B. Stroustrup)"

© 2006 Wouter Verkerke, NIKHEF

299

Stack <stack>

- A stack is an adapter of deque
  - It provides a restricted view of a deque
  - Can only insert/remove elements at end ('top' in stack view')
  - No random access



- Example

```
void sender() {
   stack<string> s ;
   s.push("Aap") ;
   s.push("Noot") ;
   s.push("Mies") ;
   receiver(s) ;
}
void receiver(stack<string>& s) {
   while(!s.empty()) cout << s.pop() << " " ;
}

// outputs "Mies Noot Aap"
```

© 2006 Wouter Verkerke, NIKHEF

300

Queue <queue>

- A queue is another adapter of deque
  - It provides a different restricted view of a deque
  - Can only insert elements at back, remove elements from front
  - No random access

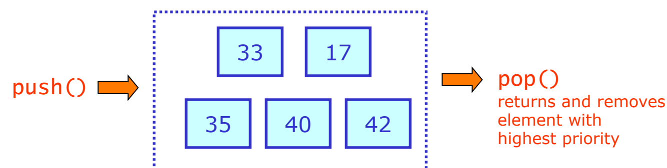
- Plenty of real-life applications
  - Ever need anything at city hall? Take a number!
  - Example implementation

```
void sender() {
   queue<string> s ;
   s.push("Aap") ; s.push("Noot") ; s.push("Mies") ;
   receiver(s) ;
}
void receiver(queue<string>& s) {
   while(!s.empty()) cout << s.pop() << " " ;
}
// outputs "Aap Noot Mies" (reversed compared to stack)
```

301

Priority_queue <queue>

- Like queue with priority control
  - In priority queue pop() returns element with highest rank, rank determined with operator<()
  - Also a container adapter (by default of vector)




```
void sender() {
   priority_queue<int> s ;
   s.push(10) ; s.push(30) ; s.push(20) ;
   receiver(s) ;
}
void receiver(queue<int>& s) {
   while(!s.empty()) cout << s.pop() << " " ;
}
// outputs "30 20 10"
```

302

Sequential container performance comparison

- Performance/capability comparison of sequential containers

	index []	Insert/remove element In Middle	At Front	At Back
vector	const	O(n) +		const +
list		const	const	const
deque	const	O(n)	const	const
stack				const +
queue			const	const +
prio_que			O(logN)	O(logN)

- const : *constant CPU cost*
- O(n) : *CPU cost proportional to Nelem*
- O(logN) : *CPU cost proportional to log(Nelem)*

© 2006 Wouter Verkerke, NIKHEF

303

Sequential versus associative containers

- So far looked at several forms of *sequential* containers
  - Defining property: storage organization revolves around *ordering*: all elements are stored in a user defined order
  - Access to elements is always done by relative or absolute position in container
  - Example:

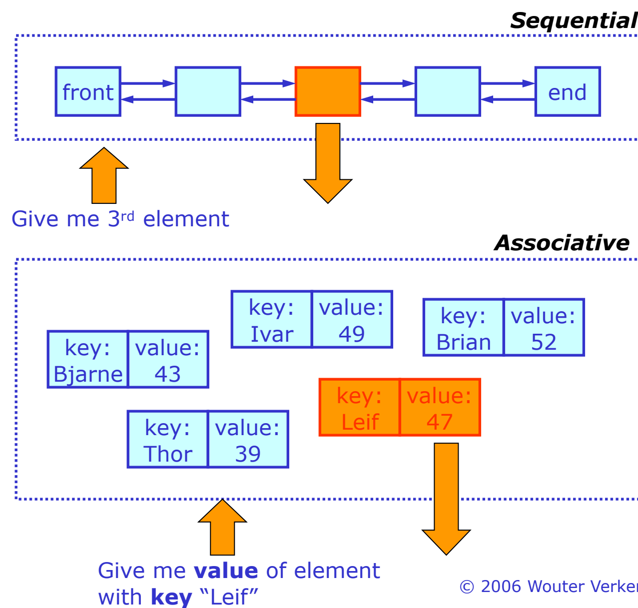
```
vector<int> v ;
v[3] = 4rd element of vector v

List<double> l ;
double tmp = *(l.begin()) ; // 1st element of list
```

- For many types of problems *access by key* is much more natural
  - Example: Phone book. You want to know the phone number (=value) for a name (e.g. 'B. Stroustrup' = key)
  - You don't care in which order collection is stored as you never retrieve the information by positional reference (i.e. you never ask: give me the 103102nd entry in the phone book)
  - Rather you want to access information with a 'key' associated with each value
- Solution: the **associative container**

© 2006 Wouter Verkerke, NIKHEF

304

Sequential versus associative containers



© 2006 Wouter Verkerke, NIKHEF

305

Pair <utility>

- Utility for associative containers – stores a key-value pair

```
template<type T1, type T2>
struct pair {
   T1 first ;
   T2 second ;
   pair(const T1&, const T2&) ;
} ;

template<type T1, type T2>
pair<T1,T2> make_pair(T1,T2) ; // exists for convenience
```

  - Main use of pair is as input or return value

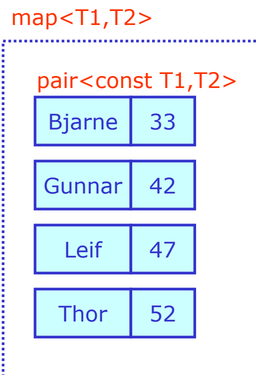
```
pair<int,float> calculation() {
   return make_pair(42,3.14159) ;
}
int main() {
   pair<int,float> result = calculation() ;
   cout << "result = " << pair.first
        << " " << pair.second << endl ;
}
```

© 2006 Wouter Verkerke, NIKHEF

306

Map <map>

- Map is an associative container
  - It stores pairs of *const* keys and values
  - Elements stored in ranking by keys (using `key::operator<()`)
  - **Provides direct access by key**
  - Multiple entries with same key prohibited



© 2006 Wouter Verkerke, NIKHEF

307

Map <map>

- Map example

```
map<string,int> shoeSize ;

shoeSize.insert(pair<string,int>("Leif",47)) ;
showSize.insert(make_pair("Leif",47)) ;

shoeSize["Bjarne"] = 43 ;
shoeSize["Thor"] = 52 ;

int theSize = shoeSize["Bjarne"] ;     // theSize = 43
int another = shoeSize["Stroustrup"] ; // another = 0
```

  - If element is not found, new entry is added using default constructors

© 2006 Wouter Verkerke, NIKHEF

308

Set <set>

- A set is a map without values
  - I.e. it is just a collection of keys
  - No random access through []
  - Storage in ranked by key (<)
  - No duplicates allowed

```
set<string> people ;

people.insert("Leif") ;
people.insert("Bjarne") ;
people.insert("Stroustrup") ;
```

set<T1>

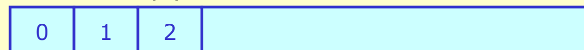
© 2006 Wouter Verkerke, NIKHEF

309

Hash table <unordered_set> C++2011

- An unordered set, is a a set with a sepcial internal ordering to facilitate a fast lookup mechanism, the "hash table"
  - Instead of unified collected, a hash table splits set into a vector of n small subsets.
  - A given object O is stored or retrieved from a subset i, that is defined as the hash value modulo the table size:

i = h(O) % n



  - The hash function h(O) should generate pseudo-random numbers in a deterministic way from the object O, so that the distribution of objects over the n subset is more or less uniform
  - Number of subsets must be a prime number, and be of the order of the expected total number of elements, to maintain a fast setup

© 2006 Wouter Verkerke, NIKHEF

310

Hash table <unordered_set> C++2011

- In case two stored elements end up in the same subset "collision", this does not cause errors, subset is std::set again, but if it happens a lot it will slow down lookup
- Syntax: unordered_set<yourClass> myHashTable
- Difficult part of hash tables is the hash function.
  - Class std::unordered_set provides a default hash function through template class hash<yourKeyType>
  - It is also possible to provide your own hash function, if you wish
  - Syntax: unordered_set<yourClass,yourHasher> myHashTable

© 2006 Wouter Verkerke, NIKHEF

311

Multiset <set>, Multimap <map>

- Multiset
  - Identical to `set`
  - Except that multiple keys of the same value are allowed
- Multimap
  - Identical to `map`
  - Except that multiple keys of the same value are allowed

© 2006 Wouter Verkerke, NIKHEF

312

Taking a more abstract view of containers

- So far have dealt directly with container object to insert and retrieve elements
  - Drawback: Client code must know exactly what kind of container it is accessing
  - Better solution: provide an *abstract interface* to the container.
  - Advantage: the containers will provide the same interface (as far as possible within the constraints of its functionality)
  - Enhanced encapsulation – You can change the type of container class you use later without invasive changes to your client code
- STL abstraction mechanism for container access: **the iterator**
  - An iterator is *a pointer to an element in a container*
  - *So how is an iterator different from a regular C++ pointer? – An iterator is aware of the collection it is bound to*.
  - *How do you get an iterator:* A member function of the collection will give it to you

© 2006 Wouter Verkerke, NIKHEF

313

Taking a more abstract view of containers

- Illustration of iterators vs C++ pointers

```
double array[10] ;

int i = 0 ;
double* ptr ;

ptr = &array[0] ;

while(i<10) {

  cout << *ptr << endl ;

  ++ptr ;
  ++i ;
}
```

Allocate C++ array of 10 elements

```
vector<double> v(10) ;

vector<double>::iterator iter ;

iter = v.begin() ;

while(iter!=v.end()) {

  cout << *iter << endl ;

  ++iter ;
}
```

Allocate STL vector of 10 elements

© 2006 Wouter Verkerke, NIKHEF

314

Taking a more abstract view of containers

- Illustration of iterators vs C++ pointers

```
double array[10] ;                vector<double> v(10) ;

int i = 0 ;
double* ptr ;                     vector<double>::iterator iter ;

ptr = &array[0] ;                 iter = v.begin() ;

while(i<10) {                     while(iter!=v.end()) {

  cout << *ptr << endl ;            cout << *iter << endl ;

  ++ptr ;                           ++iter ;
  ++i ;                           }
}
```

Allocate a pointer. Also allocate an integer to keep track of when you're at the end of the array

Allocate an STL iterator to a vector

© 2006 Wouter Verkerke, NIKHEF

315

Taking a more abstract view of containers

- Illustration of iterators vs C++ pointers

```
double array[10] ;                vector<double> v(10) ;

int i = 0 ;
double* ptr ;                     vector<double>::iterator iter ;

ptr = &array[0] ;                 iter = v.begin() ;

while(i<10) {                     while(iter!=v.end()) {

  cout << *ptr << endl ;            cout << *iter << endl ;

  ++ptr ;                           ++iter ;
  ++i ;                           }
}
```

Make the pointer point to the first element of the array

Make the iterator point to the first element of the vector

© 2006 Wouter Verkerke, NIKHEF

316

Taking a more abstract view of containers

- Illustration of iterators vs C++ pointers

```
double array[10] ;

int i = 0 ;
double* ptr ;

ptr = &array[0] ;

while(i<10) {

  cout << *ptr << endl ;

  ++ptr ;
  ++i ;
}
```

Check if you're at the end of your array

```
vector<double> v(10) ;

vector<double>::iterator iter ;

iter = v.begin() ;

while(iter!=v.end()) {

  cout << *iter << endl ;

  ++iter ;
}
```

Check if you're at the end of your vector

© 2006 Wouter Verkerke, NIKHEF

317

Taking a more abstract view of containers

- Illustration of iterators vs C++ pointers

```
double array[10] ;

int i = 0 ;
double* ptr ;

ptr = &array[0] ;

while(i<10) {

  cout << *ptr << endl ;

  ++ptr ;
  ++i ;
}
```

Access the element the pointer is currently pointing to

```
vector<double> v(10) ;

vector<double>::iterator iter ;

iter = v.begin() ;

while(iter!=v.end()) {

  cout << *iter << endl ;

  ++iter ;
}
```

Access the element the iterator is currently pointing to

© 2006 Wouter Verkerke, NIKHEF

318

Taking a more abstract view of containers

- Illustration of iterators vs C++ pointers

```
double array[10] ;                vector<double> v(10) ;

int i = 0 ;
double* ptr ;                     vector<double>::iterator iter ;

ptr = &array[0] ;                 iter = v.begin() ;

while(i<10) {                     while(iter!=v.end()) {

  cout << *ptr << endl ;            cout << *iter << endl ;

  ++ptr ;                           ++iter ;
  ++i ;
}                                 }
```

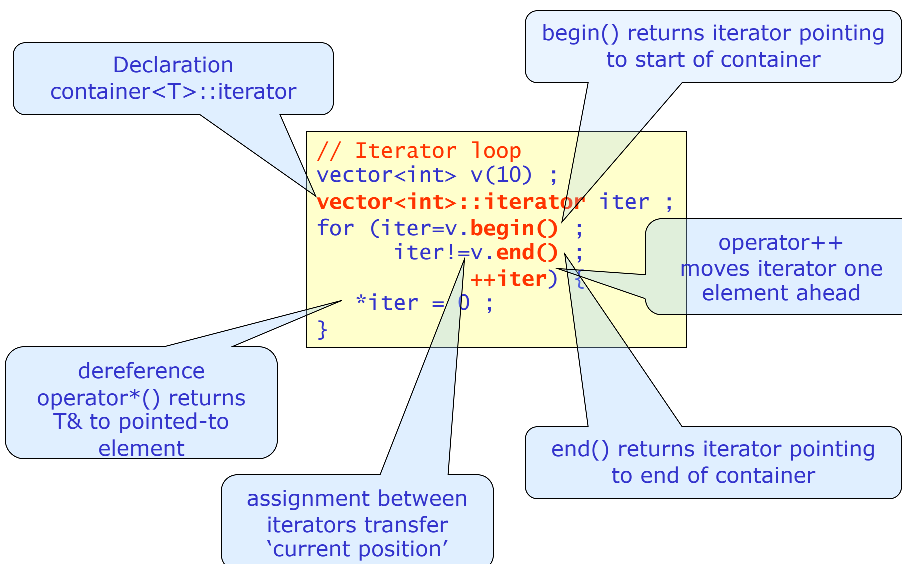
Modifiy the pointer to point to the next element in the array

Modify the iterator to point to the next element in the array

© 2006 Wouter Verkerke, NIKHEF

319

Containers Iterators – A closer look at the formalism



© 2006 Wouter Verkerke, NIKHEF

320

Why iterators are a better interface

- Iterators hide the structure of the container
  - Iterating over a vector, list or map works the same. Your client code needs to make no (unnecessary) assumptions on the type of collection you're dealing with

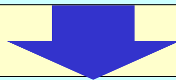
```
// Iterator loop over a vector
vector<int> v(10) ;
int sum = calcSum(v.begin()) ;

int calcSum(vector<int>::iterator iter) {
  int sum(0) ;
  while(iter) {
    sum += *iter ;
    ++iter ;
  }
  return sum ;
}
```

```
// Iterator loop over a list
list<int> l ;
int sum = calcSum(l.begin()) ;

int calcSum(list<int>::iterator iter) {
  int sum(0) ;
  while(iter) {
    sum += *iter ;
    ++iter ;
  }
  return sum ;
}
```


Auto types work great with STL contains C++2011

- Note that 'auto' types are particularly handy when using STL classes as iterator type names are usually long, and never explicitly needed

```
// Iterator loop
vector<int> v(10) ;
vector<int>::iterator iter ;
for (iter=v.begin() ;iter!=v.end() ; ++iter) {
   *iter = 0 ;
}
```



```
// Iterator loop
vector<int> v(10) ;
for (auto iter=v.begin() ; iter!=v.end() ; ++iter) {
   *iter = 0 ;
}
```

© 2006 Wouter Verkerke, NIKHEF

Even better: range-based for loops C++2011

- C++2011 also introduces concept of 'range-based' for loops over any entity that supports iterators

```
std::vector<int> v = {0, 1, 2, 3, 4, 5};

// Loop over all elements of v
for (auto i : v) { // access by value,
    cout << i << endl ;
}

// Loop over all elements of v
for (auto&& i : v) { // access by reference,
  cout << i << endl ;
}
```

- Works for any container that defines methods begin() and end() that return an iteratable type

© 2006 Wouter Verkerke, NIKHEF

323

Iterators and ranges

- A pair of iterators is also an efficient way to define subsets of containers
  - Example with multimap-based phone book

```
#include <map>

multimap<string,int> pbook ;
pbook.insert(pair<string,int>("Bjarne",00205726666)) ; // office phone
pbook.insert(pair<string,int>("Bjarne",00774557612)) ; // home phone
pbook.insert(pair<string,int>("Bjarne",0655765432)) ; // cell phone
pbook.insert(pair<string,int>("Fred",0215727576))  ; // office phone

auto iter begin=pbook.lower_bound("Bjarne"),
          end=pbook.upper_bound("Bjarne") ;

for(iter=begin ; iter!=end ; ++iter) {
    cout << iter->first << " " << iter->second << endl ;
}
```

  - Running iterator in standard way between bounds supplied by `lower_bound()` and `upper_bound()` accesses all elements with key `"Bjarne"`
  - No special mechanism needed to indicate subsets

324

Iterators and container capabilities

- Not all containers have the same capabilities
  - For example list does not allow random access (i.e. you cannot say: give me the 3rd element. You can say: give the next element with respect to the current iterator position, or the preceding element)
- Solution: STL provides multiple types of iterators with different capabilities
  - All iterators look and feel the same so you don't notice they're different, except that if a container can't do a certain thing the corresponding iterator won't let you do it either
  - Within a given set of functionality (e.g. only sequential access but no random access) all STL containers that provide that interface look and feel the same when accessed through iterators
- What classes of iterators exist:
  - Input, Output, Forward, Bidirectional, RandomAccess

© 2006 Wouter Verkerke, NIKHEF

325

Types of iterators

- Input iterator
  - Special iterator for input, for example from keyboard
  - Iterator allows to read input and must be incremented before next input is allowed

```
var = *iter++ ;
```

- Output iterator
  - Special iterator for output sequence, for example to standard output
  - Iterator allows to write and must be incremented before next output is allowed

```
*iter++ = var ;
```

- Forward iterator
  - Input and output are both allowed
  - Iteration must occur in positive increments of one

```
*iter = var ;
var = *iter ;
++iter ;
```

© 2006 Wouter Verkerke, NIKHEF

326

Types of iterators

- Bidirectional iterator
  - Can move forward and backward in steps of one

```
*iter = var ;
var = *iter ;
++iter ;
--iter ;
```

- Random access iterator
  - Can move with arbitrary step size, or move to absolute locations

```
*iter = var ;
var = *iter ;
++iter ;
--iter ;
iter[3] = var ;
iter += 5 ;
iter -= 3 ;
```

© 2006 Wouter Verkerke, NIKHEF

327

Containers and iterators

- Iterator functionality overview

Input Iterator →
Forward Iterator → Bidirectional Iterator → Random Access Iterator
Output Iterator →

- Iterators provided by STL containers

Container	Provided iterator
`vector`	random access
`deque`	random access
`list`	bidirectional
`(multi)set`	bidirectional
`(multi)map`	bidirectional
`stack`	none
`(priority_)queue`	none

Obtaining iterators

```
// Forward iterators
C.begin()
C.end()

// Reverse iterators
C.rbegin()
C.rend()
```

© 2006 Wouter Verkerke, NIKHEF

328

Iterators as arguments to generic algorithms

- Iterators allow generic algorithms to be applied to containers
  - Iterators hide structure of container → Access through iterator allows single algorithm implementation to operate on multiple container types
  - Examples

```
vector vec<int>(10) ;

// Shuffle vector elements in random order
random_shuffle(vec.begin(),vec.end()) ;

// Sort vector elements according to operator< ranking
sort(vec.begin(),vec.end()) ;

list l<string> ;
// Sort list elements according to operator< ranking
sort(l.begin(),l.end()) ;
```

© 2006 Wouter Verkerke, NIKHEF

329

STL algorithms – sort

- Sorts elements of a sequential container
  - Uses `operator<()` for ranking
  - Needs RandomAccess iterators
- Example

```
vector<int> grades(100) ;
sort(grades.begin(),grades.end()) ; // sort all elements

auto halfway = grades.begin()+50 ; // C++2011
sort(grades.begin(),halfway) ; // sort elements [0,49]
```

- Notes
  - Pair of iterators is used to indicate range to be sorted
  - Range does not include element pointed to by upper bound iterator
    - Function `end()` returns a 'past the end' iterator so that the last element of the container is included in the range when `end()` is used as endpoint

© 2006 Wouter Verkerke, NIKHEF

330

STL algorithms – find

- Finds element of a certain value in a sequential container
  - Uses `operator==()` to establish matches
  - Expects ForwardIterator
  - Return value: iterator pointing to element, `end()` otherwise
- Example

```
list<int> l(10) ;
int value = 3 ;

// Find first occurrence
auto iter = find(l.begin(),l.end(),value) ;
if (iter != l.end()) {
  // element found

  // Find remaining occurrences
  iter = find(++iter,l.end(),value) ;
  while (iter != l.end()) {
     // element found
     iter = find(++iter,l.end(),value)) ;
  }
}
```

© 2006 Wouter Verkerke, NIKHEF

331

STL algorithm – for_each

- Calls function for each element in sequential container
  - Pass *call back function* to algorithm
  - Call back function should take single argument of type matching container, returning void
  - Expects ForwardIterator
- Example

```
// the call back function
void printRoot(float number) {
  cout << sqrt(number) << endl ;
}

// Execute the algorithm
vector<float> v(10) ;
for_each(v.begin(),v.end(),printRoot) ;

// Calls printRoot for each element of v,
// passing its value
```

© 2006 Wouter Verkerke, NIKHEF

332

STL algorithm – copy

- Copies (part of) sequential container to another sequential container
  - Takes two input iterators and one output iterator
- Example

```
list<int> l(10) ;
vector<int> v(10) ;

// copy from list to vector
copy(l.begin(), l.end(), v.begin()) ;

// copy from list to standard output
copy(l.begin(), l.end(), ostream_iterator<int>(cout,"\n")) ;
```

- Note on `ostream_iterator`
  - Construct output iterator tied to given ostream.
  - Optional second argument is printed after each object is printed

© 2006 Wouter Verkerke, NIKHEF

333

STL algorithm overview

- There are many algorithms!

accumulate	adjacent_difference	binary_search	copy	copy_backward
count	count_if	equal	equal_range	fill
fill_n	find	find_if	for_each	generate
generate_n	includes	inner_product	inplace_merge	iota
iter_swap	lexographical_compare	compare	lower_bound	make_heap
max	max_element	merge	min_element	mismatch
next_permutation	partial_sort_copy	partial_sum	partition	pop_heap
prev_permutation	push_heap	random_shuffle	remove	remove_copy
remove_copy_if	remove_if	replace	replace_copy	replace_copy_if
replace_if	reverse	reverse_copy	rotate	rotate_copy
search	set_difference	set_intersection	set_symmetric_difference	set_union
sort	sort_heap	stable_partition	stable_sort	swap
swap_ranges	transform	unique	nth_element	partial_sort
unique_copy	upper_bound			

© 2006 Wouter Verkerke, NIKHEF

334

Modifying default behavior – Function objects

- STL container and algorithm behavior can easily be adapted using optional call back functions or function objects
  - Example from for_each

```
// the call back function
void printRoot(float number) {
  cout << sqrt(number) << endl ;
}

// Execute the algorithm
vector<float> v(10) ;
for_each(v.begin(),v.end(),printRoot) ;
```

- Mechanism generalized in several ways in STL
  - Customization argument not necessarily a function, but can also be a 'function object', i.e. anything that can be evaluated with operator().
  - STL provides a set of standard function objects to apply common behavior modifications to algorithms and containers

© 2006 Wouter Verkerke, NIKHEF

335

Function objects

- Illustration of function object concept
  - A class with operator()(int) has the same call signature as function(int)

```
// the call back function

void printRoot(float number) {
  cout << sqrt(number) << endl ;
}
```

```
// Execute the algorithm
vector<float> v(10) ;
for_each(v.begin(),v.end(),
         printRoot) ;
```

```
// the function object
class printRoot {
public:
  operator()(float number) {
     cout << sqrt(number) << endl ;
  }
} ;
```

```
// Execute the algorithm
vector<float> v(10) ;
for_each(v.begin(),v.end(),
         printRoot()) ;
```

© 2006 Wouter Verkerke, NIKHEF

336

Template function objects

- Function objects can also be templated
  - Introduces ***generic*** function objects
  - `printRoot` is example of *unary* function object (takes 1 argument)
  - Binary function objects are also common (2 arguments, returns `bool`)
    - To implement comparisons, ranking etc

```
// the template function object
template <class T>
class printRoot {
public:
  void operator()(T number) {
     cout << sqrt(T) << endl ;
  }
} ;
```

```
// Execute the algorithm on vector of float
vector<float> v(10) ;
for_each(v.begin(),v.end(),printRoot<float>()) ;
```

```
// Execute the algorithm on vector of int
vector<int> v(10) ;
for_each(v.begin(),v.end(),printRoot<int>()) ;
```

337

STL Function objects <functional>

- STL already defines several template function objects
  - Arithmetic: `plus, minus, divides,…`
  - Comparison: `less, greater, equal_to,…`
  - Logical: `logical_and, logical_or,…`
- Easy to use standard function objects to tailor STL algorithms
  - Example

```
vector<int> v(10) ;

// Default sort using int::operator<()
sort(v.begin(),v.end()) ;

// Customized sort in reversed order
sort(v.begin(),v.end(),greater<int>()) ;
```

  - Also used in some container constructors

```
bool NoCase(const string& s1, const string& s2) ;
map<string,int,NoCase> phoneBook ;
```

338

Numeric support in STL

- STL has some support for numeric algorithms as well
  - Won't mention most of them here except for one:
- Class complex
  - STL implements complex numbers as template

```
complex<double,double> a(5,3), b(1,7) ;
complex<double,double> c = a * b ;
```

  - Performance optimized template specializations exist for `<double,double>`, `<float,float>`, `<long double, long double>`
  - Default template argument is `float`

```
complex a(5,3), b(1,7) ;
complex c = a * b ;
```

  - Nearly all operators are implemented
    - complex * double, int * complex etc etc…

© 2006 Wouter Verkerke, NIKHEF

339

Tuples - Generic data types C++2011

- Previously introduced template class pair<A,B> as utility class for associated containers.
- Concept extended to completely generic unnamed type building utility in C++2011, template class tuple

```
#include <tuple>

// Making 2-tuple
std::tuple<int,char> foo (10,'x');

// Make 4-tuple using make_tuple
auto bar = std::make_tuple ("test", 3.1, 14, 'y');

// access single element
std::get<2>(bar) = 100;

// Unpack multiple elements
int myint; char mychar;
std::tie (myint, mychar) = foo; // unpack elements
```

© 2006 Wouter Verkerke, NIKHEF

340