SFU CMPT-212 2008-1 Topic: Containers And Iterators

Ján Maňuch

E-mail: jmanuch@sfu.ca

Monday 17th March, 2008

Containers

- class templates with parameter specifying the type of objects stored in the container

Concept: a container `X<T>`

- stores objects of a *single* type `T` (the type `T` has to be either a *basic type*, or if it's a *class*, the class must have the copy constructor and the assignment operator)
- `X` should have the *default constructor* initializing container object to the empty state, the *copy constructor*, the *assignment operator* and the *destructor*
- has a member function `size()` returning the number of elements stored and a member function `empty()` returning `true` if container is empty

- defines the following types: iterators — `X::iterator` and `X::const_iterator`, and the type of stored objects `X::value_type`
- has member functions `begin()` and `end()` returning iterators (of type `iterator`) referring to the *first* and a *past-the-end* elements of the container

- defines comparison operators `==` and `!=` for comparing two container objects (containers are equal if they contain same elements) and `<`, `<=`, `>` and `>=` for lexicographical ordering (can be used only if `X::value_type` provides these operations)

Refinements:

1. *sequences* — elements are stored in a linear sequence: `vector`, `deque` and `list`
2. *associative containers* — elements have *keys* through which they are accessed: `set`, `multiset`, `map` and `multimap`

Sequences

- elements are stored in a *linear order*: we have the first and the last element, and it makes sense to insert/remove elements in particular location

Concept: a sequence `X` containing elements of type `T` has the following member functions (in addition to member functions of *container concept*):

- `X(int n, const T &t)` — constructs a sequences of `n` copies of `t`
- `X(iterator i, iterator j)` — initializes a sequence to values in range `[i,j)` (from some other container)
- `insert(iterator p,const T &t)` — inserts `t` at position `p`
- `insert(iterator p, int n, const T &t)` — inserts `n` copies of `t` at position `p`

- `insert(iterator p, iterator b, iterator e)` — inserts elements from range `[b,e)` at position `p`
- `erase(iterator p)` — removes the element at position `p`
- `erase(iterator b, e)` — removes elements in range `[b,e)`
- `clear()` — removes all elements
- iterators of `X` are at least forward iterators (specified later)

Models:

- `vector` — a representation of an array ("smart" array: grows and shrinks as needed), fast to insert and delete elements in the end
- `deque` — double ended queue, fast to insert and delete elements from both ends
- `list` — doubly linked list, easy to insert anywhere, but no random access to elements

Additional methods:

- `vector`, `deque`, `list`:
  - `T& front()` — the first element
  - `T& back()` — the last element
  - `push_back(T &t)` — insert `t` at the end
  - `pop_back()` — remove element at the end
- `deque`, `list`:
  - `push_front(T &t)` — insert `t` at the beginning
  - `pop_front()` — remove element at the beginning
- `vector`, `deque`:
  - `T& operator[](int n)` — random access to elements
  - `T& at(int n)` — random access to elements with range checking (`out_of_range` exception)

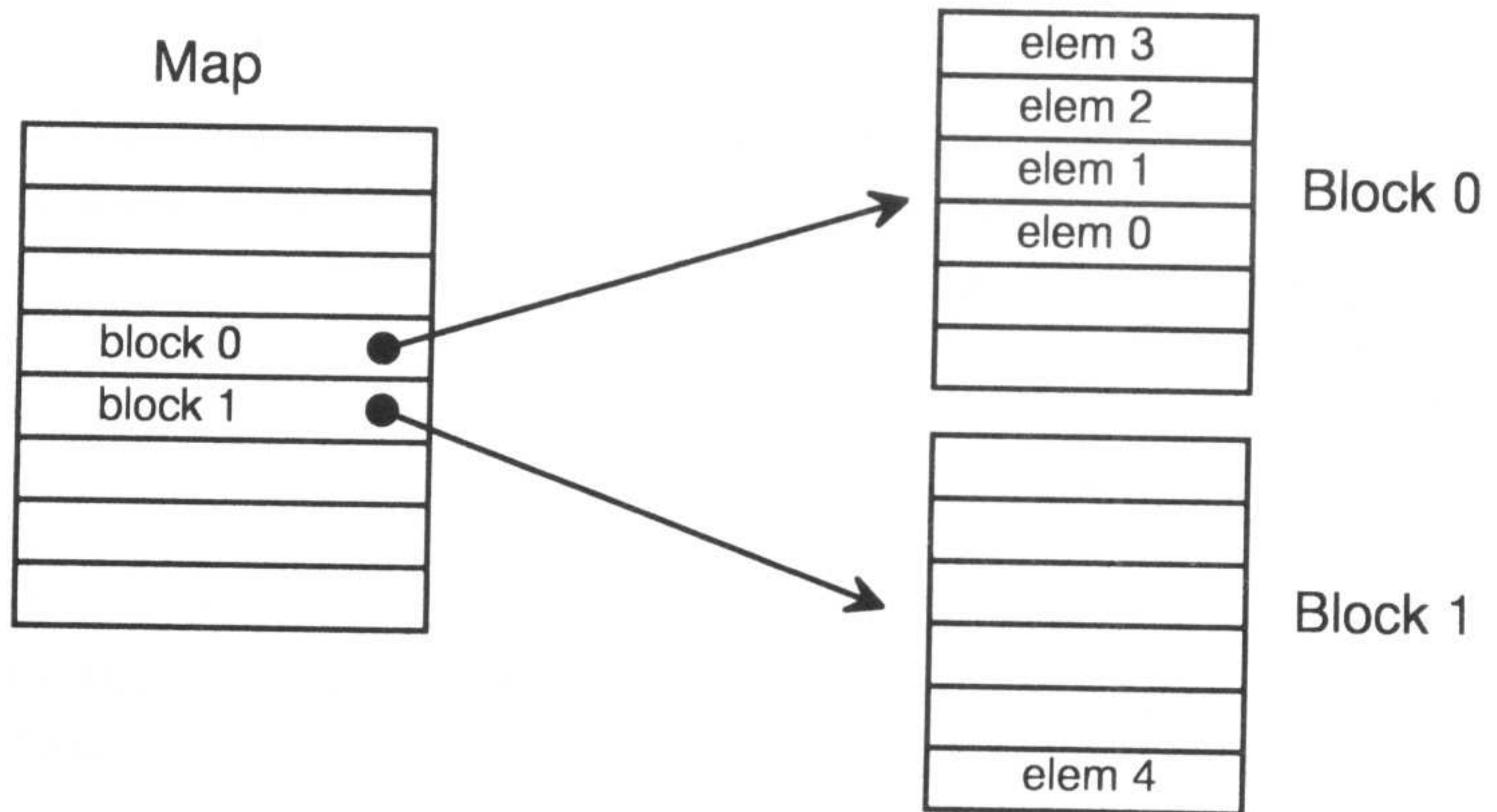
vector

- requires `<vector>` header file, defined in `namespace std`
- implemented as a dynamic array (one block, enlarged when needed and values copied): constant time insert/erase operations at the end of a `vector` (on average), but linear time anywhere else
- iterators of vector are *random-access* iterators (have similar properties as pointers)

deque

- requires `<deque>` header file, defined in `namespace std`
- constant time insert/erase operations at the front and the end of a `deque`
- *implementation*:
  - initially one memory block is allocated and the first element is placed in the middle; now adding element at the front or at the end is fast until the end of block is reached
  - `deque` doesn't enlarge old block, but allocate a new one and continue adding elements to new one
  - there several blocks (of the same fixed size) and pointers to each of them are kept in a map (array of pointers) again starting from the middle
  - when the size of map is not enough, resizing similar to `vector` will happen

Illustration of implementation of deque



list

- requires `<list>` header file, defined in `namespace std`
- *implementation:* linked list with both forward and backward links (double linked)
- *advantages:* constant time insert/erase operations at any position
- *disadvantages:*
  - iterators of `list` are just *bidirectional iterators*, not random-access iterators: you cannot use pointer arithmetic on iterators, neither *the index operator* is available;
  - finding `n`-th element of the list takes linear time
  - you cannot apply some algorithm, for instance `sort` (but `list` have they own `sort` member function)

Additional methods of `list<T>`:

- `void splice(iterator pos, list<T> &x)` — inserts content of list `x` in front of the position `pos`; `x` is emptied (constant time)

assuming that `T` *can be compared using* `operator<`

- `void merge(list<T> &x)` — merge current list with `x`; works correctly only if both current container and `x` were already sorted; the list `x` is emptied (linear time)
- `void sort()` — sort elements ($n \log n$ time)

assuming that `T` *can be compared using* `operator==`

- `void remove(const T&val)` — remove all instances equal to `val` (linear time)
- `void unique()` — remove repeated elements in the list (if they are consecutive); hence, if applied on sorted list, all elements will be different

Choosing a sequence container

memory issues:

- `vector` is the most memory efficient (at most twice the needed memory is used plus it provides methods like `resize()` and `reserve()` to manage memory more efficiently)
- `deque` also needs to store the *map*, but that's usually insignificant
- `list` needs to keep two pointer for every element stored in it

efficiency issues:

- finding `n`-the element: very fast in `vector`, fast in `deque`, very slow in `list`; hence, if the container is always processed sequentially, `list` is fine, but for random access is not suitable
- adding elements fast: only at the end for `vector`, at front and at end for `deque` and anywhere for `list`
- sorting and swapping: `vector` and `deque` can be sorted using generic algorithm `sort()` which swaps elements (physically copy elements around); while `list` has member function `sort()` which rearranges pointers instead, i.e.,
  for larger elements `list` is better, for smaller ones `vector` or `deque` is better

Sample questions

- Specify at least 5 requirements of the *container* concept.
- List two main groups of containers and for each at least 2 models implemented in STL.
- What is the main characterization of sequences?
- Explain how would you choose which sequence container to use?
- What's wrong with the following code:

```
vector<int> intVector(4,20);
intVector.insert(1,10);
```

- Implement the following template functions that reverses elements of any sequence container:

```
template <typename ContainerType>
void reverse(ContainerType &container)
```

Implement the same template if you could assume that the `ContainerType` is `vector`.

Iterators

- generalization of pointers — provides *uniform* interface to all different types of containers

  Example:

  assume you have

  - iterator `i` pointing to a location in `vector<int>` object
  - iterator `j` pointing to a location in `list<int>`

  we want to *move both iterators to the next element of containers*

  - for `i`: it's enough to increment `i` by 1
  - for `j`: we should follow the link to the next element in the node

  but since both iterators are *models* of *iterator concept*, we can use the `++` operator to accomplish the task

- *hence, if we write an algorithm using iterators of one container, it will work with iterators of another container* (assuming iterators of another container are in the same or higher category)

Concept:

- can be dereferenced (using the `*` operator) to return the object they point to
- can be compared for equality (`==` and `!=` is defined for them)

Types of iterators:

- *input iterator*:
  - allows reading values from the container (using dereferencing);
  - defines the `++` operator allowing to move through all elements of the container
  - the order can be *different* each time and after incrementing, the prior value of iterator might not be pointing at any meaningful location in the container
- *output iterator* — similar to input iterator, but instead of reading values, allows writing values

- *forward iterator*
  - is an *input* and *output iterator*
  - elements are traversed in the *same* order (unless content of container has changed)
- *bidirectional iterator*
  - is a *forward iterator*
  - can move through elements in reverse order: defines the `--` operator
- *random access iterator*
  - is a *bidirectional iterator*
  - allows pointer arithmetic (with integer values) and defines the index `[ ]` operator

Examples:

- iterators of `vector` and `deque` are random access iterators
- iterators of `list` are bidirectional operators

Note:

- iterators form a *hierarchy*

  Example: a *bidirectional iterator* is a *forward iterator*
  hence, in any place where a *forward iterator* is expected, a *bidirectional* (or *random access*) iterator can also be used

- the prototypes of *generic algorithms* of STL specify what kind of iterator is expected as an argument by the names of template parameters:

```
template <typename InputIterator, typename T>
InputIterator find(InputIterator begin,
  InputIterator end, const T& value);
```

  – looks for value `value` in range `[begin,end)`
  – returns iterator pointing to the first occurrence of `value` if found
  – otherwise returns `end`

Sample questions

- List 5 types of iterators and explain what are the differences between them.
- Why is important to understand hierarchy of those iterators?
- What's wrong with the following implementation of algorithm `find`:

```
1  template <typename InputIterator, typename T>
2  InputIterator find(InputIterator begin,InputIterator end,
3                     const T& value)
4  {
5    for (int i=0; begin+i!=end; i++)
6      if (*(begin+i)==value)
7        return begin+i;
8    return end;
9  }
```

Associative containers

- in *sequential containers*, elements are stored in a linear order and can be traversed in this way (and often elements can be accessed directly via index operator `[ ]`)
- in *associative containers*, elements are accessed via *keys*
- when inserting elements to an associative container, you have no control where this element will be inserted (you cannot specify the position)

  Example:

```
1  v.insert(v.begin()+2,10.3); // for vector<double>
2  s.insert(10.3);             // for set<double>
```

- the elements can be access via keys rather fast (in logarithmic time)

Concept of associative container X

- is a *container*
- defines a type used for keys `X::key_type`
- `X(InputIterator i, InputIterator j)` — initializes a sequence to values in range `[i,j)` (from some other container)
- `erase(const key_type &k)` — removes all elements with key `k`
- `clear()` — removes all elements
- `iterator find(const key_type &k)` — returns iterator to element with key `k` and `end()` if no such element exists
- `long count(const key_type &k)` — returns the number of elements with key `k`

Models for associative container

- `set`, `multiset`, `map` and `multimap` (there are also `hash` versions of these containers)
- `set` and `multiset` are defined in `<set>` header file and store only keys (`value_type` = `key_type`)
- `map` and `multimap` are defined in `<map>` header file, and each stored element is a `value_type` = `pair<key_type,data_type>`
- `set` and `map` do not allow multiple keys, i.e., each element has a unique key
- `multiset` and `multimap` allow several elements to have same keys

set and multiset

- inserting elements to `set`:

  `pair<iterator, bool> insert(const key_type &t);`

  if `t` was not in the `set` already, returns the iterator pointing to inserted `t` and `true`, otherwise the iterator pointing at element with key value `t` and `false`

  Note: you can access elements of `pair` template class directly with `first` and `second` data members

  Example: set_insert.cpp

```
1  set<double> s;
2  pair<set<double>::iterator,bool> answer=s.insert(2.3);
3  cout <<*(answer.first); // dereference iterator, prints 2.3
4  cout <<answer.second;   // prints true
5  answer=s.insert(2.3);   // fails to insert 2.3 again
6  cout <<*(answer.first); // dereference iterator, prints 2.3
7  cout <<answer.second;   // prints false
```

- inserting elements to `multiset`:

  `iterator insert(const key_type &t);`

  cannot fail, as `multiset` can contain several elements with the same key

  Example:

```
1 multiset<double> ms;
2 ms.insert(3.4);
3 ms.insert(2.3);
4 ms.insert(3.4);
5 multiset<double>::iterator i;
6 for (i=ms.begin(); i!=ms.end(); i++)
7   cout <<*i<<endl;
8 // prints: 2.3 3.4 3.4 (note: in sorted order!)
```

- insert range:

  `insert(InputIterator i,InputIterator j)` — insert elements from some other iterator in range `[i,j)`

map and multimap

- need two types: `key_type` and `data_type`

  Example:

```
1 map<char,float> x; // key=char, data=float
```

- can be used as *associative arrays*: elements of ordinary arrays are accessed through integer values, element of associative arrays are accessed through any (fixed) type (`key_type`)

- dereferencing iterator gives you `value_type=pair<key_type,data_type>`

```
1 map<double,char>::iterator i;
2 cout <<(*i).first;  // prints the key (double)
3 cout <<(*i).second; // prints the data (char)
```

- inserting elements to map:

```
pair<iterator, bool>
insert(const pair<key_type,data_type>& x);
```

— returned value is the same as for set (multimap version returns only iterator)

- to simplify working with maps, *index operator* is overloaded as well:

```
data_type& operator[](const key_type& k);
```

which can be used for both inserting and accessing values of the map:

```
map<double,char*> m;
m[3.14]="Pi";    // insert pair (3.14,"Pi") to map
cout <<m[3.14]; // access the data stored in the map
                // prints "Pi"
// beware:
char* name=m[1.41]; // insert pair (1.41,NULL) to map
                    // and return NULL
```

Example: days.cpp

Strict Weak Ordering

When using associative containers `set<Key>`, `multiset<Key>`, `map<Key,Data>` and `multimap<Key,Data>` either:

- `Key` has to provide `operator<`;
- or we have to pass class whose instance is a *binary predicate comparing two elements of type* `Key` as an additional template parameter.

Example:

```
1  class Point {
2  public:
3    int x,y;
4    Point (int ix=0,int iy=0) : x(ix), y(iy) {}
5  };
6  ...
7  set<Point> P;
8  // compiler error, operator<(Point,Point) not defined
```

Fix 1: either define `operator<` for `Point`, or

```
class Point_less {
public:
  bool operator()(const Point& p,const Point& q)
        { return (p.x<q.x && p.y<q.y); }
};
...
set<Point,Point_less> P;
// compiles, uses Point_less to sort keys = Points
```

Still a problem, see point1.cpp

Why?

- Because `Point_less` is not a “strict weak ordering” ordering!
- *Strict weak ordering:*
  - `x<x` should be `false`;
  - if `x<y` is `true`, then `y<x` should be `false`;
  - if `x<y` and `y<z` are `true`, then also `x<z` should be `true`;
  - if `x<y` and `y<x` are false, then `x=y` is true.

Example: point2.cpp

Sample questions

- What is the main difference between sequential and associative containers?
- List four STL models of associative container and explain differences between them.
- Consider a `set`. What are the possible values which the `count` method can return?
- Which containers are suitable for implementing "associative arrays"?
- Show two different ways how to insert a `value_type` pair to a `map<double,long>`. For instance, insert a pair with key `4.6` and data `10L`.

- Write a program that counts the occurrences of words in the input taken from the user (terminated by the word `"end"`) and displays the words and their occurrences in ascending order of word frequencies. The program should use an STL `map` to store a pair consisting of a word and its count. For each word, check whether it is already a key in the map. If not, add the key and value `1` to the map. Otherwise, increase the value for the word (key) by `1` in the map.

Other types of iterators

1. Reverse iterators

- Assume you want to traverse elements of the container `vector<int> container` in reverse order.
- *First (incorrect) approach:*

```
1 vector<int>::iterator i;
2 for (i=container.end(); i!=container.begin(); i--)
3   cout << *i << " ";
```

What's wrong?

- `end()` is a *sentinel* iterator marking we have reached the end of the container, it does not point to a meaningful element of the container (should not be dereferenced)
- it's not guaranteed that decreasing `end()` will make iterator to point at the last element of the container (depends on the container and implementation)

- the correct way is to use *reverse iterators* provided by containers with *bidirectional iterators*:

```
1  vector<int>::reverse_iterator i;
2  for (i=container.rbegin(); i!=container.rend(), i++)
3    cout << *i << " ";
```

Comments:

  - `rbegin()` returns an iterator pointing at the last element
  - `rend()` returns a sentinel marking the end of the container in the reverse order
  - `i++` moves the reverse iterator backward!

2. Stream iterators

- defined in header file `<iterator>`
- *Example:*

```
1 ostream_iterator<double> double_to_cout(cout, "; ");
2 // declares an output stream iterator double_to_count
3 // with delimiter string "; " attached to cout
```

  – to send a `double` value to adapter iterator, we can use dereferencing and the assignment:

```
1 *double_to_cout=3.14; // equivalent to: cout <<3.14<<"; ";
2 double_to_cout++;     // get ready for the next print-out
3 *double_to_cout=2;    // cout <<double(2)<<"; ";
```

- *Question: how is it useful?*

  Answer: output stream iterator can be used in any generic algorithm where an output iterator is expected

Example:

- `<algorithm>` defines a generic algorithm `copy()` copying data from one container to another:

```
copy(InputIterator first, InputIterator last,
     OutputIterator target);
```

copies range `[first,last)` into other container starting at position `target`

Example:

```
vector<double> a;
// ... fill a
// copy content of vector a to list b:
list<double> b(a.size());
copy(a.begin(), a.end(), b.begin());
```

Notes:

- overwrite the data in the target container
- assumes that there is enough space

- copy function expects an *output iterator* as target

```
copy(InputIterator first, InputIterator last,
    OutputIterator target);
```

hence, we can use the output stream iterator to copy the data from the container to the screen!

Example:

```
deque<double> d;
// ... fill d
ostream_iterator<double> double2cout(cout,"; ");
copy(d.begin(),d.end(),double2cout);
// or using anonymous syntax:
copy(d.begin(),d.end(),
     ostream_iterator<double>(cout,"; "));
```

Example: copy.cpp

- the `istream_iterator` template can adapt an input stream to the iterator interface:

```
istream_iterator<double> cin2double(cin);
double x = *cin2double++;
// read double value from cin
```

  Example with `copy`:

```
copy(cin2double,istream_iterator<double>(),
     d.begin());
```

  Notes:

  – `istream_iterator<double>()` indicates the input failure

  – hence, the above statement will read the input from `cin` and copy values to container `d` until some error occurs (for example the input is not of type `double`)

3. Insert iterators

- *Example:* in the previous example

```
1  copy(istream_iterator<double>(cin),
2       istream_iterator<double>(),
3       d.begin());
```

  - reads `double` values from `cin` until an input failure occurs
  - it overwrites the data in the container `d` and it assumes the `d` has enough space

- we would like to insert data instead of overwriting
  - appending using `back_insert_iterator`

```
1  vector<double> d;
2  copy(istream_iterator<double>(cin),
3       istream_iterator<double>(),
4       back_insert_iterator<vector<double> >(d));
```

Example: backinsert.cpp

- *Q.* The syntax `back_insert_iterator<vector<double> >(d))` is quite troublesome. Can we make compiler to check the container type of `d` and create the corresponding `back_insert_iterator`?

- *A.* Sure, function templates can guess type based on the parameters.

```
1 template <typename Container>
2 back_insert_iterator<Container> backInserter(Container &c)
3 { return back_insert_iterator<Container>(c); }
```

and now we can write the following code:

```
1 vector<double> a,b;
2 // ... fill a,b
3 copy(a.begin(),a.end(),backInserter(b));
4 // append content of a to the end of b
```

Notes:

  - `<iterator>` defines such function template called `back_inserter`
  - `back_insert_iterator` works only with containers which have `push_back()` method (sequences)

Other versions of insert iterators:

- `front_insert_iterator` — inserts in the front, works only with containers which provide `push_front()` member function (such as `deque` and `list`)
- `insert_iterator` — works with any container; takes 2 parameters: container and a position (iterator) where to start inserting

```
1 vector<int> v;
2 v.push_back(1);
3 v.push_back(9);
4 int a[]={7,2,4,3,6};
5 copy(a,a+5,inserter(v,v.begin()+1));
6 // a: [1,7,2,4,3,6,9]
```

Note: with associative containers like `set`, you can only use `insert_iterator` and the position where to insert (the second parameter) is basically ignored

Supplemental material

How are they implemented?

Example: For `set`:

- Inserter example: setinserter.cpp
- Erasor example: seterasor.cpp

Sample questions

- Write a code listing elements of a container `v` of type `list<double>` in reverse order.
- What is stream iterator? How can it be used?
- Which insert iterator would you use to insert elements to a vector at the front?