Practical Programming Methodology
(CMPUT-201)

Michael Buro

Lecture 24

- STL Algorithms
- Exceptions

Non-Mutating Algorithms

Work on range but do not change elements

```
for_each : apply a function to each element
find     : find an element
equal    : checks whether two ranges are the same
count    : count elements equal to value
search   : search for a sub-sequence
...
```


for_each

```
template <class InpIterator, class UnaryFunc>
UnaryFunc for_each(InpIterator begin,
                   InpIterator end,
                   UnaryFunc f)
```

- Applies function or functor f to each element in `[begin, end)`
- Returns the function object after it has been applied to all elements in `[begin, end)`

for_each Example

```
#include <set>
#include <algorithm>

struct Add
{
  int sum;

  Add() { sum = 0; }
  void operator()(int x) { sum += x; }
};

set<int> s;
s.insert(1); s.insert(2); s.insert(3);

Add f = for_each(s.begin(), s.end(), Add());

cout << f.sum << endl;          // 1+2+3 = 6
```

for_each Implementation

```
template <typename InputIterator, typename Functor>
Functor for_each(InputIterator first,
                 InputIterator end,
                 Functor f)
{
  for (; first != end; ++first) f(*first);
  return f;
}
```


Mutating Algorithms

Work on range and possibly change elements

```
remove_if : moves elements for which a predicate is false
            to front, returns new_end, size unchanged
partition : reorders elements; x with pred(x)=true come
            first
generate  : assigns results of function calls to each element
copy      : copies input range to output iterator
fill      : assigns a value to each element
reverse   : reverses range
rotate    : general rotation of range w.r.t. to mid-point
random_shuffle : randomly shuffles all elements
sort      : sorts a range

... many more
```



```
#include <algorithm>
struct Even { // functor
  bool operator()(int x) { return (x & 1) == 0; }
};
const int N = 20;
vector<int> v, w;  int a[N];

partition(v.begin(), v.end(), Even());//even | odd
generate(v.begin(), v.end(), rand);

copy(v.begin(), v.end(), w.begin()); // dangerous!
                         // w must be large enough
copy(v.begin(), v.end(), back_inserter(w));//better

fill(v.begin(), v.end(), 314159);

reverse(a, a+N); // array viewed as STL container
rotate(v.begin(), v.begin()+1, V.end()); // "<<< 1"
random_shuffle(a, a+N);
```


sort

```
template <typename RandomAccessIterator>
sort(RandomAccessIterator first,
     RandomAccessIterator end);
// uses operator <

template <typename RandomAccessIterator,
          typename StrictWeakOrdering>
sort(RandomAccessIterator first,
     RandomAccessIterator end,
     StrictWeakOrdering less);
// uses comparison functor less
```

- Sorts random access range in ascending order
- Implements "introspection sort" which combines quicksort and heapsort
- Worst and average case complexity: $\Theta(n \log n)$
- Fast!

sort Examples

```
#include <algorithm>
#include <functional> // for less<T>, greater<T> ...
using namespace std;

vector<int> v(10);
const int N = 20;
int a[N];

generate(v.begin(), v.end(), rand);
generate(a, a+N, rand);

sort(v.begin(), v.end());  // asc., uses <(int,int)
sort(a, a+N);              // ascending

sort(v.begin(), v.end(), greater<int>()); // desc.
sort(a, a+N, greater<int>()); // descending
```


What else is there in STL?

Hashed associative containers

- e.g. `hash_set<T,HashFunc,EqualKey>`
- organized as hash tables
- faster than the standard tree-based containers
- but need more space
- see `www.sgi.com/tech/stl`

More sorting related functions
(`stable_sort, merge, lower_bound` ...)

More C++ libraries at `www.boost.org`

Error Handling

Historical C-Way

- Use function return codes to indicate error conditions
- E.g. `int fgetc(FILE *stream);`
  - Returns read character (value in 0..255)
  - Or -1 if read error occurred
- Drawbacks
  - What if function returns full range of values?
  - Errors can be easily ignored

Modern Solution: Exceptions

Exceptions

- Dealing with rare error conditions
- Write code as if nothing can go wrong
- Enclose it in try-block which will be exited if some operation fails and throws an exception
- Add a catch-block to handle exceptions

Example

```
struct MyException {
  string msg;
  MyException(const std::string &m): msg(m) { }
};

void foo()
{ ...
  if (error) throw MyException("division by 0");
  ...
}

int main()
{
  try { foo(); }
  catch (MyException &e) {
    cout << "caught exception! cause: " << e.what;
  }
}
```


Syntax: function definition

```
<type> <func>(<ParamList>) [throw (<ExceptionList>)]
{
  ... throw <exception>(<Params>);
}
```

- If <ExceptionList> is empty, <func> not allowed to throw exception
- If throw clause is missing, <func> can throw anything
- Important: if <func> throws an exception not on the list, function `std::unexpected()` is called (program terminates)

Syntax: try-catch block

```
try { ... }

catch (<type> [<var>]) { ... }

[ <var> is optional, there can be more than on catch block ]

or

catch (...) { ... } : catches all exceptions

Re-throw in catch blocks (throw;): catch search continues
```


Example

```
void foo3() {
  throw MyException();
  i = 0; // not reached
}

void foo2() {
  string s2;
  foo3();
  i = 0; // not reached
} // s2 destroyed


void foo1() {
  vector<int> v1;
  foo2();
  i = 0; // not reached
} // v1 destroyed
```

```
void main() {
  try { foo1(); }
  catch (MyExcpetion &e) {
    // execution continues here
  }
}
```

Catching Exceptions

Once an exception is thrown (can be any type!), program execution is stopped

The runtime system then looks for the next catch statement whose type is compatible with the thrown value:

- If the exception was thrown in a try block, the following catch statements are checked
- If no match, the search for an exception handler resumes in the caller ("stack unwinding") after all local objects have been destroyed.
- If no matching catch statement is found, the program is aborted by calling terminate()

If match found, execution resumes there