COMP2004
Programming Practice
2002 Summer School

Kevin Pulo
School of Information Technologies
University of Sydney

Mutating Algorithms

- All the algorithms so far have not modified elements
- Some algorithms do modify values
  - ie. use output iterators
- They don't modify iterators - only elements
  - ie. ranges are fixed
- This causes some algorthms to be a little strange

copy

- We've used this before
- Copies elements from one range to another
- Returns an iterator at the end of second range

copy example

```
int  main()  {
        char  a[ ]  =  "1234567890";
        vector<char>  v(a,  a + strlen(a));
        list<char>  l1(v.size());
        list<char>  l2;
        copy(v.begin(),  v.end(),  l1.begin());
        copy(v.begin(),  v.end(),
                                back_inserter(l2));
}
```

copy_backward

- Just like copy only does the assigments in reverse order
- Useful when ranges overlap
- Is passed the range to copy from
- And the end of the range to copy to
- The return value is an iterator at end of destination range

copy_backward example

```
int  main()  {
        int  a[ ]  =  {1,2,3,4,5,6,7,8,9,10};
        copy_backward(a,  a + 8,  a + 10);
        copy(a,  a + 10,
                        ostream_iterator<int>(cout));
}
```

swap_ranges

- Exchanges the contents of two ranges
  - Avoids need for a temporary container
- Is passed a range and the start of the second range
- The second range must be as big as the first
- Returns an iterator at end of second range

swap_ranges example

```
int  main()  {
        int  a[ ]  =  {1, 2, 3, 4, 5};
        vector<int>  v;
        for  (int  i = 1;  i <= 5;  ++i)
                v.push_back(i * 10);
        swap_ranges(v.begin(),  v.end(),  a);
}
```

transform

- Like for_each except copies return values into a range
- Returns iterator at end of output range
- Two variations
  - 2 input ranges and binary function
  - 1 input range and unary function

transform example

```
int  main()  {
        int  a1[ ]  =  {1, 2, 3, 4, 5};
        int  a2[ ]  =  {6, 7, 8, 9, 10};
        transform(a1,  a1 + 5,  a2,
                  ostream_iterator<int>(cout, "  "),
                  plus<int>());
        cout  <<  endl;
        transform(a1,  a1 + 5,
                  ostream_iterator<int>(cout, "  "),
                  bind1st(multiplies<int>(), 5));
}
```

replace

- Replaces all occurances of a value with another
  - ie. modifies original range
- There is also replace_copy
  - Doesn't modify original range
  - Copies result to an output iterator

replace example

```
int  main()  {
    int  a1[ ]  =  {1, 2, 3, 4, 5, 1, 2, 3, 4, 5};
    int  a2[ ]  =  {1, 2, 3, 4, 5, 1, 2, 3, 4, 5};
    int  a3[10];
    replace(a1,  a1 + 10,  3,  10);
    replace_copy(a2,  a2 + 10,  a3,  3,  10);
    for  (int  i = 0;  i < 10;  ++i)
        cout  <<  a1[i]  <<  '\t'  <<  a2[i]  <<  '\t'
              <<  a3[i]  <<  endl;
}
```

replace_if

- Like replace except a binary predicate is used instead of a value
- There is also replace_if_copy

```
int  main()  {
        int  a[ ]  =  {1, 2, 3, 4, 5, 1, 2, 3, 4, 5};
        replace_if(a,  a + 10,
                        bind2nd(less<int>(), 3),  10);
        copy(a,  a + 10,
                 ostream_iterator<int>(cout, "  "));
}
```

fill and fill_n

- fill assigns a value to each element in a range
- fill_n assigns a value to n elements starting at an iterator

fill and fill_n example

```
int  main()  {
        vector<int>  v(4);
        fill(v.begin(),  v.end(),  42);
        fill_n(back_inserter(v),  4,  24);
        copy(v.begin(),  v.end(),
                 ostream_iterator<int>(cout, "  "));
}
```

generate and generate_n

- Like fill except uses a generator function object

```
int  main()  {
        vector<int>  v(1000);
        generate(v.begin(),  v.end(),  rand);
        generate_n(
                 ostream_iterator<int>(cout, "\n"),
                 1000,  rand);
}
```

remove

- Removes all occurances of a value
- Since ranges are of fixed size it's a little strange
- Merely rearranges the elements
- Returns a new end iterator
- Elements after it have unspecified values
- Can actually remove the elements with

```
new_end = remove(c.begin(), c.end(), v);
c.erase(new_end,  c.end());
```

remove example

```
int  main()  {
    int  a[ ]  =  {1, 2, 1, 2, 3, 1, 2, 3, 4};
    const  int  N  =  9;
    int  *i  =  remove(a,  a + N,  1);
    copy(a,  i,
            ostream_iterator<int>(cout, "  "));
    cout  <<  endl;
    copy(i,  a + N,
            ostream_iterator<int>(cout, "  "));
    cout  <<  endl;
}
```

remove_if

- Removes elements matching a unary predicate function
- Has the same strangeness as remove
- There is also remove_copy and remove_copy_if
  - They copy the only the elements that don't match

remove_if example

```
int  main()  {
    int  a[ ]  =  {1, 2, 3, 4, 5, 1, 2, 3, 4, 5};
    vector<int>  v;
    copy(a,  a + 10,  back_inserter(v));

    v.erase(remove_if(v.begin(),  v.end(),
                          bind2nd(less<int>(), 3)));

    copy(v.begin(),  v.end(),
             ostream_iterator<int>(cout, "  "));
}
```

unique

- Removes adjcent duplicate elements
- Has the strangeness of remove
- Returns a new end iterator
- Can be passed a Binary Predicate to use for comparisons
- There is also unique_copy

unique example

```
int  main()  {
    int  a[ ]  =  {1,1,2,1,2,3,1,1,1,2,3,2,2};
    const  int  N  =  13;
    int  *i  =  unique(a,  a + N);
    copy(a,  i,
            ostream_iterator<int>(cout, "  "));
    cout  <<  endl;
    copy(i,  a + N,
            ostream_iterator<int>(cout, "  "));
    cout  <<  endl;
}
```

reverse

- Reverses the elements in a range
- Requires bidirectional iterators
- There is also reverse_copy
  - Different from copy_backward

reverse example

```
int  main()  {
    list<string>  l;
    l.push_back("one");
    l.push_back("two");
    l.push_back("three");
    reverse(l.begin(),  l.end());
    copy(l.begin(),  l.end(),
        ostream_iterator<string>(cout, "\n"));
}
```

random_shuffle

- Rearranges the range into a random order
- All permutations are equally favoured
- A hard algorithm to get right yourself

random_shuffle example

```
int  main()  {
        string  a[ ]  =  { "one",  "two",  "three" };
        const  int  N  =  3;
        random_shuffle(a,  a + N);
        copy(a,  a + N,
           ostream_iterator<string>(cout, "\n"));
}
```

sort

- Sorts a range
- By default sort into ascending order
- Can be passed a Strict Weak Ordering Function Object
- stable_sort should be used if stability is required

sort example

```
int  main()  {
    string  a[ ]  =  {"one","two","three","four"};
    const  int  N  =  4;
    sort(a,  a + N,  not2(less<string>()));
    copy(a,  a + N,
          ostream_iterator<string>(cout, "\n"));
}
```

partial_sort

- Puts the smallest n elements of a range at the start in sorted order
- More efficient than sort if you only need a few elements
- Can be passed a Strict Weak Ordering Function Object
- There is also partial_sort_copy which only copies the n elements

partial_sort example

```
int  main()  {
    vector<int>  v(50);
    generate(v.begin(),  v.end(),  rand);
    partial_sort(v.begin(),  v.begin() + 10,
                                                      v.end());
    copy(v.begin(),  v.end(),
               ostream_iterator<int>(cout, "\n"));
}
```

is_sorted

- Tests if a range is sorted
- Returns the appropriate bool
- Useful if something can be done faster with a sorted range

is_sorted example

```
template<typename  It>
void  slow_sort(It  begin,  It  end)  {
        while  (!is_sorted(begin,  end))
                random_shuffle(begin,  end);
}
int  main()  {
        int  a[ ]  =  {1, 2, 3, 5, 4};
        const  int  N  =  5;
        slow_sort(a,  a + N);
        copy(a,  a + N,
                ostream_iterator<int>(cout, "  "));
}
```

merge

- Combines two sorted ranges into an output range
- Is stable

```
int  main()  {
        int  even[ ]  =  {0, 2, 4, 6, 8, 10};
        int  odd[ ]  =  {1, 3, 5, 7, 9, 11};
        merge(odd,  odd + 6,
                 even,  even + 6,
                 ostream_iterator<int>(cout, "  "));
}
```

Useful mutating algorithms

- Can look these up in normal STL reference (+ today's tutorial)
- rotate, rotate_copy
- next_permutation, prev_permutation
- partition, stable_partition
- random_sample, random_sample_n
- nth_element
- binary_search
- lower_bound, upper_bound, equal_range