CS 247: Software Engineering Principles

Generic Algorithms

Reading: Eckel, Vol. 2
Ch. 6 Generic Algorithms

C++ Standard Template Library

- A collection of useful, typesafe, generic (i.e., type-parameterized) containers that
  - know (almost) nothing about their elements
  - focus mostly on membership (insert, erase)
  - know nothing about algorithms
  - can define their own iterators

- A collection of useful, efficient, generic algorithms that
  - know nothing about the data structures they operate on
  - know (almost) nothing about the elements in the structures
  - operate on structures sequentially via iterators

STL Algorithms

for_each
find
find_if
find_end
find_first_of
adjacent_find
count
count_if
mismatch
equal
search
search_n
copy
copy_backward
swap
swap_ranges
iter_swap
transform
replace
replace_if
replace_copy
replace_copy_if

fill
fill_n
generate
generate_n
remove
remove_if
remove_copy
remove_copy_if
unique
unique_copy
reverse
reverse_copy
rotate
rotate_copy
random_shuffle
partition
stable_partition
sort
stable_sort
partial_sort
partial_sort_copy
nth_element

lower_bound
upper_bound
equal_range
binary_search
merge
inplace_merge
includes
set_union
set_intersection
set_difference
set_symmetric_difference
push_heap
pop_heap
make_heap
sort_heap
min
max
min_element
max_element
lexicographical_compare
next_permutation
prev_permutation

Overview

Most STL algorithms "process" a sequence of data elements

- traverse a sequence of elements bounded by two iterators
- access elements through the iterators
- operate on each element during traversal

```
template<class InputIterator, class T>
InputIterator find (InputIterator first, InputIterator last, const T& val)
```

points to first element in input range

points past last element in input range

Non-Modifying Algorithms

A number of the algorithms read, but never write to, the elements in their input range.

```
template<class InputIterator, class T>
  InputIterator find (InputIterator first, InputIterator last, const T& val)
{
  while (first!=last) {
    if (*first==val) return first;
    ++first;
  }
  return last;
}
```

Algorithms over Two Sequences

Algorithms that operate over two sequences of data specify the full range over the first sequence and only the start of the second sequence.

```
template <class InputIterator1, class InputIterator2>
  bool equal ( InputIterator1 first1, InputIterator1 last1, InputIterator2 first2 )
{
  while (first1!=last1) {
    if (!(*first1 == *first2))
      return false;
    ++first1; ++first2;
  }
  return true;
}
```

```cpp
#include <iostream>     // std::cout
#include <algorithm>    // std::equal
#include <vector>       // std::vector

using namespace std;

int main () {
  int myints[] = {11, 22, 33, 44, 55, 66};   // myints: 11 22 33 44 55 66

  vector<int>myvector (myints,myints+5);     // myvector: 11 22 33 44 55

  // using default comparison: operator==
  if ( equal (myvector.begin(), myvector.end(), myints) )
    cout << "The contents of both sequences are equal.\n";
  else
    cout << "The contents of both sequences differ.\n";


  return 0;
}
```

Modifying Algorithms

Some algorithms overwrite element values in existing container.

- we must take care to ensure that the destination sequence is large enough for the number of elements being written.

```
template<class InputIterator, class OutputIterator>
  OutputIterator copy (InputIterator first, InputIterator last, OutputIterator result)
{
  while (first!=last) {
    *result = *first;
    ++result; ++first;
  }
  return result;
}
```

Overwriting vs. Inserting

The default behaviour is to write to a destination sequence, overwriting existing elements.

Can impose insertion behaviour instead by providing an inserter iterator as the destination.

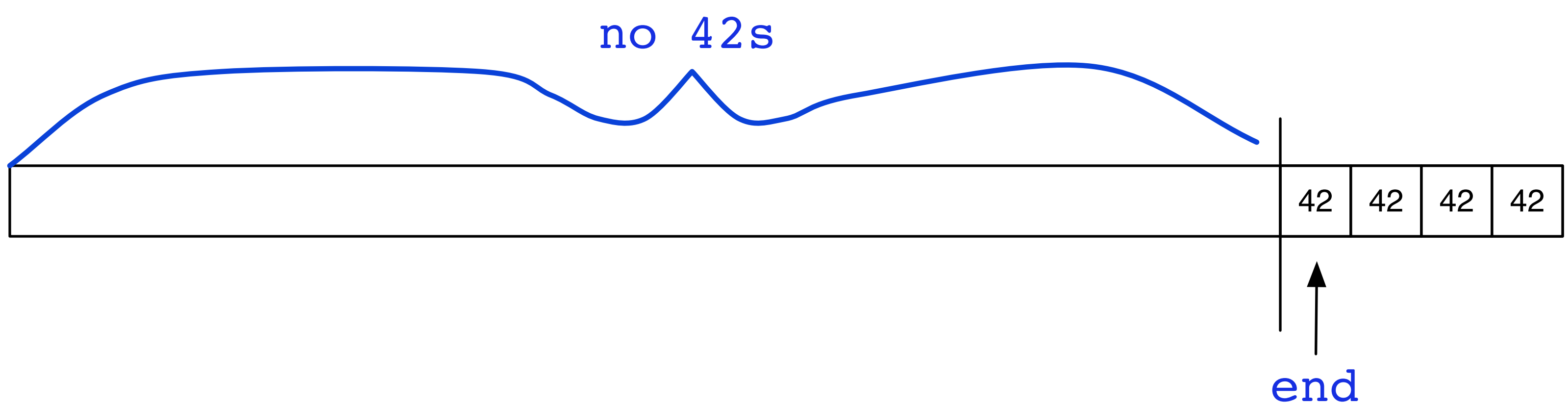
```cpp
#include <iostream>     // std::cout
#include <algorithm>    // std::copy
#include <vector>       // std::vector

using namespace std;

int main () {
    vector<int> myvector;
    myvector.push_back(11);
    myvector.push_back(77);

    int myints[]={11,22,33,44,55,66,77};

    copy ( myints+1, myints+6, inserter(myvector, myvector.begin()+1) );

    cout << "myvector contains:";
    for (vector<int>::iterator it = myvector.begin(); it!=myvector.end(); ++it)
        cout << ' ' << *it;

   cout << '\n';

    return 0;
}
```

```
#include <algorithm>
#include <fstream>
#include <iterator>
#include <vector>
#include <string>

using namespace std;

int main () {
    ifstream inFile( "input.txt" );
    istream_iterator< string > is (inFile);
    istream_iterator< string > eof;
    vector< string > text;
    copy ( is, eof, back_inserter( text ));

    sort( text.begin(), text.end() );

    ofstream outFile( "output.txt" );
    ostream_iterator< string > os (outFile, "\n");
    copy ( text.begin(), text.end(), os );
}
```

"Removing" Elements

```
template <class ForwardIterator, class T>
ForwardIterator remove (ForwardIterator first, ForwardIterator last, const T& val)
```

Algorithms never directly change the size of containers -- need to use container operators to add/remove elements. Instead, algorithms rearrange elements -- sometimes placing undesirable elements at the end of the container and returning an iterator past the last valid element.




```
vector<int>::iterator end = remove ( vec.begin(), vec.end(), 42);
vec.erase ( end, vec.end() ); // to remove the 42s
```

Algorithms that Apply Operations

A number of algorithms apply operations to the elements in the input range:

- e.g., transform(), count_if(), sort()

Some STL algorithms accept a predicate

- applied to all elements in iteration
- used to restrict set of data elements that are operated on

```
bool gt20(int x) { return 20 < x; }
bool gt10(int x) { return 10 < x; }

int a[] = { 20, 25, 10 };
int b[10];

remove_copy_if( a, a+3, b, gt20 ); // b[] == {25};
cout << count_if( a, a+3, gt10 );  // Prints 2
```

Function Objects

If we need a function that that refers to data other than the iterated elements, we need to define a function object (aka functor)

- class that overloads `operator()`, the function call operator
- `operator()` allows an object to be used with function call syntax

```
class gt_n {
  int value_;
public:
  gt_n(int val) : value_(val) {}
  bool operator()(int n) { return n > value_; }
};
```

constructor

function calls

```
int main() {
  gt_n gt4(4);
  cout << gt4(3) << endl;   // Prints 0 (for false)
  cout << gt4(5) << endl;   // Prints 1 (for true)
}
```

Eckel, Allison, *Thinking in C++*, Vol. 2, Prentice Hall, 2004

Function Objects

```
// We supply this function object
class gt_n {
  int value;
public:
  gt_n(int val) : value(val) {}
  bool operator()(int n) { return n > value; }
};


int main() {
  int a[] = { 5, 25, 10 };
  const size_t SIZE = sizeof a / sizeof a[0];

  gt_n gt15(15);
  cout << count_if( a, a+ SIZE, gt15 );    // Prints 1
  cout << count_if( a, a+ SIZE, gt_n(0) ); // Prints 3
}
```

Eckel, Allison, *Thinking in C++*, Vol. 2, Prentice Hall, 2004

Another Example

```
class inc {
public:
    inc(int amt) : increment_(amt) {}
    int operator()(int x) { return x + increment_; }
private:
    int increment_;
};
```

input range

```
transform( V.begin(), V.end(), V.begin(), inc( 100 ) );
```

points to first element in input range

points past last element in input range

destination of results of transformation

Classification of Function Objects

Generator: A type of function object that
- takes no arguments
- returns a value of an arbitrary type

Unary Function: A type of function object that
- takes a single argument of any type
- returns a value that may be of a different type (which may be void)

Binary Function: A type of function object that
- takes two arguments of any two (possibly distinct) types
- returns a value of any type (including void)

Unary Predicate: A Unary Function that returns a bool.

Binary Predicate: A Binary Function that returns a bool.

Predefined Function Objects

Header `<functional>` defines a number of useful generic function objects

- plus<T>, minus<T>, times<T>, divides<T>, modulus<T>, negate<T>
- greater<T>, less<T>, greater_equal<T>, less_equal<T>, equal_to<T>, not_equal_to<T>
- logical_and<T>, logical_or<T>, logical_not<T>

Can be used to customize many STL algorithms. For example, function objects are often used to override the default operator used by an algorithm. For example, sort by default uses `operator<`. To instead sort in descending order, could use the function object `greater`.

```
sort(svec.begin(), svec.end(), greater<string>());
```

```cpp
#include <iostream>     // std::cout
#include <algorithm>    // std::transform
#include <vector>       // std::vector
#include <functional>   // std::plus

int op_increase (int i) { return ++i; }

int main () {
  std::vector<int> foo;
  std::vector<int> bar;

  // set some values:
  for (int i=1; i<6; i++)
    foo.push_back (i*11);                        // foo: 11 22 33 44 55

  bar.resize(foo.size());                        // allocate space

  std::transform (foo.begin(), foo.end(), bar.begin(), op_increase);
                                                 // bar: 12 23 34 45 56

  // std::plus adds together its two arguments:
  std::transform (foo.begin(), foo.end(), bar.begin(), foo.begin(), std::plus<int>());
                                                 // foo: 23 45 66 87 111

  std::cout << "foo contains:";
  for (std::vector<int>::iterator it=foo.begin(); it!=foo.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';

  return 0;
}
```

Predefined Function Object Adaptors

`<functional>` also defines a number of useful generic adaptors to modify the interface of a function object.

bind1st - convert a binary function object to a unary function object (by fixing the value of the first operand)

bind2nd - convert a binary function object to a unary function object (by fixing the value of the second operand)

mem_fun - convert a member function into a function object (when member function is called on pointers to objects)

mem_fun_ref - convert a member function into a function object (when member function is called on objects)

not1 - a function adaptor that reverses the truth value of a unary function object

not2 - a function adaptor that reverses the truth value of a binary function object

ptr_fun - convert a function pointer to a function object <u>so that a generic adaptor can be applied</u> -- otherwise, can simply use function pointer

Function Object Adaptors

Example: Convert a binary function object to a unary function object

```
bind2nd( greater<int>(), 15 ); // x > 15

bind1st( minus<int>(), 100);   // 100 – x
```

Example: Convert a member function into a function object

```
vector<string> strings;
vector<Shape*> shapes;

transform( strings.begin(), strings.end(), dest,
                         mem_fun_ref( &string::length ) );

transform( shapes.begin(), shapes.end(), dest2,
                         mem_fun( &Shape::size ) );
```

Adaptable Function Objects

To be <u>adaptable</u>, a function object class must provide nested type definitions for its arguments and return type:

```
typedef T1 argument_type;

typedef T2 result_type;

typedef T3 first_argument_type;
typedef T4 second_argument_type;
```

unary function object

binary function object

Adaptable Function Objects

Since these names are expected of all standard function objects as well as of any function objects you create to use with function object adaptors, the **<functional>** header provides two templates that define these types for you: **unary_function** and **binary_function**. You simply derive from these classes while filling in the argument types as template parameters. Suppose, for example, that we want to make the function object **gt_n**, defined earlier in this chapter, adaptable. All we need to do is the following:

Can create our own adaptable function objects by deriving from unary_function or binary function

```
class gt_n : public unary_function<int, bool> {
  int value;
public:
    gt_n(int val) : value(val) {}
    bool operator()(int n) {
      return n > value;
 }
};
```

Summary of STL Algorithms

Generic algorithms <u>process</u> <u>sequences of data</u> <u>of any type</u> in a <u>type-safe</u> manner.

- process: generate, search, transform, filter, stream, compare, ...
- sequences of data: algorithms iterate over elements in a sequence
- of any type: algorithms are function templates that parameterize the type of *iterator* (the type of data is *mostly* irrelevant)
- type-safe: compiler detects and reports type errors

Concluding Remarks

The goal of the STL is to provide a set of data structures and algorithms that are:

- generic – parameterized by type
- strongly typed – e.g., vector<Figure *>
- flexible – large APIs, many possible modes of use
- extensible – inherit/extend for your own needs OR create a specialized, restricted API via *adapters*
- efficient – static method dispatch, specialized algorithms
- (relatively) easy to use