Generic Algorithms

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

Generic Algorithms

- 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

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
C++ Templates (and STL) rely on "Duck Typing"

C++ templates realize polymorphism through type parameters and what is called “Duck Typing”:

- “If it walks like a duck, and talks like a duck, it must be a duck.”
- An object of class C is type-compatible with a template parameter type T if it supplies all of the methods/method signatures used in the template.

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

Non-Modifying Algorithms

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

```cpp
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;
}
```

Iterator Type Hierarchy

The Iterator type hierarchy is based on “duck typing”—the (sub)set of capabilities of the iterator.

Any STL algorithm that requires an InputIterator as a parameter will also accept a fancier iterator in the type hierarchy.

```
input iterator
  ↓
forward iterator
  ↓
bidirectional iterator
  ↓
random access iterator

output iterator
  ↓

iterator types:
- istream, ostream
- unordered_set, unordered_multiset, unordered_map, unordered_multimap
- list, set, multiset, map, multimap
- vector, deque
```

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.

```cpp
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;
}
```
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.

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.

"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.

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

```
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

```cpp
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 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.

```cpp
class gt_n {
    int value;
    public:
        gt_n(int val) : value(val) {} // constructor
        bool operator()(int n) { return n > value; }
    }
```

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

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

Another Example

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

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

Transform over Two Sequences

int add (int a, int b) {
    return a + b;
}

int main () {
    int A[]={11,22,33,44,55,66,77};

    vector<int> V ( 7, 10 ); // seven elements of value 10
    transform ( A, A+sizeof(A), V.begin(), V.begin(), add );

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 \texttt{<functional>} defines a number of useful generic function objects
- plus\textless T\textgreater, minus\textless T\textgreater, times\textless T\textgreater, divides\textless T\textgreater, modulus\textless T\textgreater, negate\textless T\textgreater
- greater\textless T\textgreater, less\textless T\textgreater, greater_equal\textless T\textgreater, less_equal\textless T\textgreater, equal_to\textless T\textgreater, not_equal_to\textless T\textgreater
- logical_and\textless T\textgreater, logical_or\textless T\textgreater, logical_not\textless T\textgreater

Can be used to customize many STL algorithms. For example, we can use function objects to override the default operator used by an algorithm. For example, \texttt{sort} by default uses \texttt{operator<}. To instead sort in descending order, we could provide the function object \texttt{greater\textless T\textgreater} to \texttt{sort}.

\begin{verbatim}
    sort(myvector.begin(), myvector.end(), greater<int>());
\end{verbatim}
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 predicate object
not2 - a function adaptor that reverses the truth value of a binary predicate object
ptr_fun - convert a function pointer to a function object so that a generic adaptor can be applied — 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 adaptable, a function object class must provide nested type definitions for its arguments and return type:

typedef T1 argument_type;

} unary function object

typedef T2 result_type;

} binary function object

typedef T3 first_argument_type;

typedef T4 second_argument_type;

Adaptable Function Objects are Function Objects!

Function Object Adaptors are Function Objects!

bind2nd() is a template function that outputs a binder2nd<Op> object
- protected data member binfunc (binary operation)
- protected data member value (value of second argument)
- public constructor (Op& binfunc, secondArgType value)
- operator() (firstArgType x) { returns op(x, value); }
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.

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

STL Example: Course Schedule

Goals:
- Read in list of course names and scheduled times.
- Day of week is read in as character: M,T,W,R,F,S,U.
- Time is read as integer in 24-hour HHMM format.
- Sort list according to day and then by time of day.
- Detect any overlaps between courses and print them out.
- Print out ordered schedule for the week.

Use STL to perform as much of the above as possible.

```cpp
#include <iostream>
#include <string>
enum class DayOfWeek {M, T, W, R, F, S, U};
class Course {
    public:
    friend std::ostream& operator<<(std::ostream& os, const Course& c);
    Course(std::string title, char day, int startTime, int endTime);
    bool overlaps(const Course& c) const; // detects conflicts
    bool before(const Course& c) const;  // used in sorting
    private:
    std::string title_;
    DayOfWeek day_;  
    int startTime_;  
    int endTime_;  
    std::ostream& operator<<(std::ostream& os, const Course& c);
};
```

STL Example: Course Schedule (cont.)

```bash
> cat infile
CS442 M 1030 1200
CS442 W 1030 1200
CS444 T 1600 1730
CS444 R 1600 1730
CS445 M 1130 1300
CS445 F 1130 1300
CS457 M 1330 1430
CS457 W 1330 1430
CS457 F 1330 1430
> ./sched infile
CONFLICT:
CS442 M 1030 1200
CS445 M 1130 1300
CS442 M 1030 1200
CS445 M 1130 1300
CS457 M 1330 1430
CS444 R 1600 1730
CS444 R 1600 1730
CS444 R 1600 1730
CS444 R 1600 1730
CS444 R 1600 1730
CS444 R 1600 1730
```
C++ Code Snippet

```
#include <cassert>
#include <iostream>
#include "Course.h"

using namespace std;

const std::string DayToString[] = {"M ", "T ", "W ", "R ", "F ", "S ", "U "};

DayOfWeek dayOfWeek (char c)
{
    switch (c) {
        case 'M': return DayOfWeek::M;
        case 'T': return DayOfWeek::T;
        case 'W': return DayOfWeek::W;
        case 'R': return DayOfWeek::R;
        case 'F': return DayOfWeek::F;
        case 'S': return DayOfWeek::S;
        case 'U': return DayOfWeek::U;
        default:
            assert ( 0 && "not a week day");
    }
}
```

```
Course::Course (string title, char day, int startTime, int endTime)
: title_{title}, day_{dayOfWeek(day)},
    startTime_{startTime}, endTime_{endTime}
{
}

bool Course::before (const Course& c) const {
    if ( day_ < c.day_ ) return true;
    if ( day_ == c.day_ ) {
        if ( startTime_ < c.startTime_ ) return true;
        if ( startTime_ == c.startTime_ ) {
            if ( endTime_ < c.endTime_ ) return true;
        }
    }
    return false;
}
```

```
bool Course::overlaps ( const Course& c ) const {
    if ( day_ == c.day_ &&
        (( startTime_ <= c.startTime_ && c.startTime_ <= endTime_ ) ||
         ( c.startTime_ <= startTime_ && startTime_ <= c.endTime_ )))
        return true;
    return false;
}
```

```
ostream& operator<< (ostream &os, const Course &c) {
    return
    (os << c.title_ << " " << DayToString[static_cast<int>(c.day_)]
     << " " << c.startTime_ << " " << c.endTime_);
}
```
### Summary of STL Algorithms

**Generic algorithms** process sequences of data of any type in a type-safe 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**