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

<table>
<thead>
<tr>
<th>Algorithms</th>
</tr>
</thead>
<tbody>
<tr>
<td>for_each</td>
</tr>
<tr>
<td>find</td>
</tr>
<tr>
<td>find_if</td>
</tr>
<tr>
<td>find_end</td>
</tr>
<tr>
<td>find_first_of</td>
</tr>
<tr>
<td>adjacent_find</td>
</tr>
<tr>
<td>count</td>
</tr>
<tr>
<td>count_if</td>
</tr>
<tr>
<td>mismatch</td>
</tr>
<tr>
<td>equal</td>
</tr>
<tr>
<td>search</td>
</tr>
<tr>
<td>search_n</td>
</tr>
<tr>
<td>copy</td>
</tr>
<tr>
<td>copy_backward</td>
</tr>
<tr>
<td>swap</td>
</tr>
<tr>
<td>swap_ranges</td>
</tr>
<tr>
<td>iter_swap</td>
</tr>
<tr>
<td>transform</td>
</tr>
<tr>
<td>replace</td>
</tr>
<tr>
<td>replace_if</td>
</tr>
<tr>
<td>replace_copy</td>
</tr>
<tr>
<td>replace_copy_if</td>
</tr>
<tr>
<td>fill</td>
</tr>
<tr>
<td>fill_n</td>
</tr>
<tr>
<td>generate</td>
</tr>
<tr>
<td>generate_n</td>
</tr>
<tr>
<td>remove</td>
</tr>
<tr>
<td>remove_if</td>
</tr>
<tr>
<td>remove_copy</td>
</tr>
<tr>
<td>remove_copy_if</td>
</tr>
<tr>
<td>unique</td>
</tr>
<tr>
<td>unique_copy</td>
</tr>
<tr>
<td>reverse</td>
</tr>
<tr>
<td>reverse_copy</td>
</tr>
<tr>
<td>rotate</td>
</tr>
<tr>
<td>rotate_copy</td>
</tr>
<tr>
<td>random_shuffle</td>
</tr>
<tr>
<td>partition</td>
</tr>
<tr>
<td>stable_partition</td>
</tr>
<tr>
<td>sort</td>
</tr>
<tr>
<td>stable_sort</td>
</tr>
<tr>
<td>partial_sort</td>
</tr>
<tr>
<td>partial_sort_copy</td>
</tr>
<tr>
<td>nth_element</td>
</tr>
<tr>
<td>lower_bound</td>
</tr>
<tr>
<td>upper_bound</td>
</tr>
<tr>
<td>equal_range</td>
</tr>
<tr>
<td>binary_search</td>
</tr>
<tr>
<td>merge</td>
</tr>
<tr>
<td>inplace_merge</td>
</tr>
<tr>
<td>includes</td>
</tr>
<tr>
<td>set_union</td>
</tr>
<tr>
<td>set_intersection</td>
</tr>
<tr>
<td>set_difference</td>
</tr>
<tr>
<td>set_symmetric_difference</td>
</tr>
<tr>
<td>push_heap</td>
</tr>
<tr>
<td>pop_heap</td>
</tr>
<tr>
<td>make_heap</td>
</tr>
<tr>
<td>sort_heap</td>
</tr>
<tr>
<td>min</td>
</tr>
<tr>
<td>max</td>
</tr>
<tr>
<td>min_element</td>
</tr>
<tr>
<td>max_element</td>
</tr>
<tr>
<td>lexicographical_compare</td>
</tr>
<tr>
<td>next_permutation</td>
</tr>
<tr>
<td>prev_permutation</td>
</tr>
</tbody>
</table>
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.

```cpp
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)
```
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.
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;
    }
```
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;
}
```
#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};

    vector<int>myvector (10);       // myvector: 0 0 0 0 0 0 0 0 0 0
    copy ( myints, myints+sizeof(myints), myvector.begin() );

    cout << "myvector contains:" << endl;
    ostream_iterator<int> os (cout, "\n");
    copy ( myvector.begin(), myvector.end(), os );

    cout << '\n';

    return 0;
}
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

```cpp
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
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) {}  
    bool operator()(int n) { return n > value_; } 
};

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

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 };

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

```cpp
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);
}
```

- **points to first element in first range**
- **points past last element in first range**
- **points to first element in second range**
- **destination of results of transformation**
- **binary operation**
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, we can use function objects to override the default operator used by an algorithm. For example, `sort` by default uses `operator<`. To instead sort in descending order, we could provide the function object `greater<T>` to `sort`.

```
sort(myvector.begin(), myvector.end(), greater<int>());
```
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

```cpp
bind2nd( greater<int>(), 15 ); // x > 15
bind1st( minus<int>(), 100);  // 100 - x
```

Example: Convert a member function into a function object

```cpp
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:

```cpp
typedef T1 argument_type;
typedef T2 result_type;
```

}} unary function object

```cpp
typedef T3 first_argument_type;
typedef T4 second_argument_type;
```

}} binary function object

U Waterloo CS247 (Spring 2017) — p.23/36
Function Object Adaptors are Function Objects!

template <class Operation>
class binder2nd : public unary_function <typename Operation::first_argument_type,
    typename Operation::result_type>
{
    protected:
        Operation op;
        typename Operation::second_argument_type value;

    public:
        binder2nd ( const Operation& x,
                    const typename Operation::second_argument_type& y) : op (x), value(y) {}

        typename Operation::result_type
            operator() (const typename Operation::first_argument_type& x) const
            { return op(x,value); }
};

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`.

```
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
Adapted from Doug Schmidt

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.
STL Example: Course Schedule (cont.)

> 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
CS457 M 1330 1430
CS444 T 1600 1730
CS442 W 1030 1200
CS457 W 1330 1430
CS444 R 1600 1730
CS445 F 1130 1300
CS457 F 1330 1430
#include <iostream>
#include <string>

enum class DayOfWeek {M, T, W, R, F, S, U};

class Course {
public:
    friend std::ostream& operator<<(std::ostream&, const Course&);

    Course (std::string title, char day, int startTime, int endTime);

    bool overlaps (const Course&) const; // detects conflicts
    bool before (const Course&) const;   // used in sorting

private:
    std::string title_;
    DayOfWeek day_;
    int startTime_;  
    int endTime_;    
};

std::ostream& operator<<(std::ostream&, const Course&);
```cpp
#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_);
}
#include <iostream>
#include <fstream>
#include <string>
#include <vector>
#include <iterator>
#include <algorithm>
#include "Course.h"
using namespace std;

// Forward declare helper routines.
int inputCourses (int argc, char *argv[], vector<Course> &schedule);
void printConflicts (vector<Course> &schedule);
void printSchedule (vector<Course> &schedule);

int main (int argc, char *argv[]) {
    vector<Course> schedule;
    if ( inputCourses( argc, argv, schedule ) < 0 ) return -1;
    sort ( schedule.begin(), schedule.end(), mem_fun_ref( &Course::before ) );
    printConflicts ( schedule );
    printSchedule ( schedule );

    return 0;
}
```cpp
int inputCourses (int argc, char *argv[], vector<Course> &schedule) {
    if ( argc < 2 ) return -1;

    ifstream ifile (argv[1]);
    string title;
    char day;
    int startTime;
    int endTime;

    while (ifile >> title >> day >> startTime >> endTime) {
        Course c (title, day, startTime, endTime);
        schedule.push_back(c);
    }

    return ifile.eof() ? 0 : -1;
}
```
```cpp
void printConflicts (vector<Course> &schedule)
{
    vector<Course>::iterator iter = schedule.begin();
    vector<Course>::iterator end = schedule.end();

    while ( ( iter =
               adjacent_find( iter, end, mem_fun_ref(&Course::overlaps)) ) != end )
    {
        cout << "CONFLICT:" << endl << " " << *iter << endl
             << " " << *(iter+1) << endl
             << endl;
        iter++;
    }
}

void printSchedule (vector<Course> &schedule)
{
    ostream_iterator<Course> iter (cout, "\n");
    copy (schedule.begin(), schedule.end(), iter);
}
```
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**