The C++ Standard Template Library

Douglas C. Schmidt

Professor
d.schmidt@vanderbilt.edu
www.dre.vanderbilt.edu/∼schmidt/

Department of EECS
Vanderbilt University
(615) 343-8197

February 12, 2014

The C++ Standard Template Library

- What is STL?
- Generic Programming: Why Use STL?
- Overview of STL concepts & features
  - *e.g., helper class & function templates, containers, iterators, generic algorithms, function objects, adaptors*
- A Complete STL Example
- References for More Information on STL

What is STL?

The Standard Template Library provides a set of well structured **generic** *C++ components that work together in a* **seamless** *way.*

–Alexander Stepanov & Meng Lee, *The Standard Template Library*

What is STL (cont'd)?

- A collection of composable class & function templates
  - Helper class & function templates: operators, pair
  - Container & iterator class templates
  - Generic algorithms that operate over *iterators*
  - Function objects
  - Adaptors
- Enables generic programming in C++
  - Each generic algorithm can operate over *any iterator for which the necessary operations are provided*
  - Extensible: can support new algorithms, containers, iterators

Generic Programming: Why Use STL?

- **Reuse: "write less, do more"**
  - STL hides complex, tedious & error prone details
  - The programmer can then focus on the problem at hand
  - *Type-safe* plug compatibility between STL components
- **Flexibility**
  - Iterators decouple algorithms from containers
  - Unanticipated combinations easily supported
- **Efficiency**
  - Templates avoid virtual function overhead
  - Strict attention to time complexity of algorithms

STL Features: Containers, Iterators, & Algorithms

- **Containers**
  - *Sequential*: `vector`, `deque`, `list`
  - *Associative*: `set`, `multiset`, `map`, `multimap`
  - *Adapters*: `stack`, `queue`, `priority_queue`
- **Iterators**
  - Input, output, forward, bidirectional, & random access
  - Each container declares a trait for the type of iterator it provides
- **Generic Algorithms**
  - Mutating, non-mutating, sorting, & numeric

STL Container Overview

- STL containers are *Abstract Data Types* (ADTs)
- All containers are parameterized by the type(s) they contain
- Each container declares various traits
  - *e.g.*, `iterator`, `const_iterator`, `value_type`, *etc.*
- Each container provides factory methods for creating iterators:
  - `begin()/end()` for traversing from front to back
  - `rbegin()/rend()` for traversing from back to front

Types of STL Containers

- There are three types of containers
  - **Sequential containers** that arrange the data they contain in a linear manner
    - Element order has nothing to do with their value
    - Similar to builtin arrays, but needn't be stored contiguous
  - **Associative containers** that maintain data in structures suitable for fast associative operations
    - Supports efficient operations on elements using keys ordered by `operator<`
    - Implemented as balanced binary trees
  - **Adapters** that provide different ways to access sequential & associative containers
    - *e.g.*, `stack`, `queue`, & `priority_queue`

STL Vector Sequential Container

- A **std::vector** is a dynamic array that can grow & shrink at the end
  - *e.g., it provides (pre—re)allocation, indexed storage,* `push_back()`, `pop_back()`
- Supports *random access* iterators
- Similar to—but more powerful than—built-in C/C++ arrays

```
#include <iostream>
#include <vector>
#include <string>

int main (int argc, char *argv[])
{
  std::vector <std::string> projects;

  std::cout << "program name:"
            << argv[0] << std::endl;

  for (int i = 1; i < argc; ++i) {
    projects.push_back (argv [i]);
    std::cout << projects [i - 1]
              << std::endl;
  }

  return 0;
}
```


STL Deque Sequential Container

- A **std::deque** (pronounced "deck") is a double-ended queue
- It adds efficient insertion & removal at the *beginning* & *end* of the sequence via `push_front()` & `pop_front()`

```
#include <deque>
#include <iostream>
#include <iterator>
#include <algorithm>

int main() {
  std::deque<int> a_deck;
  a_deck.push_back (3);
  a_deck.push_front (1);
  a_deck.insert (a_deck.begin () + 1, 2);
  a_deck[2] = 0;
  std::copy (a_deck.begin (), a_deck.end (),
             std::ostream_iterator<int>
                          (std::cout, " "));
  return 0;
}
```

STL List Sequential Container

- A **std::list** has constant time insertion & deletion at *any* point in the sequence (not just at the beginning & end)
  - performance trade-off: does not offer a *random access* iterator
- Implemented as doubly-linked list

```
#include <list>
#include <iostream>
#include <iterator>
#include <string>
int main() {
  std::list<std::string> a_list;
  a_list.push_back ("banana");
  a_list.push_front ("apple");
  a_list.push_back ("carrot");

  std::ostream_iterator<std::string> out_it
    (std::cout, "\n");

  std::copy (a_list.begin (), a_list.end (), out_it);
  std::reverse_copy (a_list.begin (), a_list.end (),
                     out_it);
  std::copy (a_list.rbegin (), a_list.rend (), out_it);
  return 0;
}
```


STL Associative Container: Set

- An **std::set** is an ordered collection of unique keys
  - *e.g., a set of student id numbers*

```
#include <iostream>
#include <iterator>
#include <set>

int main () {
  std::set<int> myset;

  for (int i = 1; i <= 5; i++) myset.insert (i*10);
  std::pair<std::set<int>::iterator,bool> ret =
    myset.insert (20);
  assert (ret.second == false);

  int myints[] = {5, 10, 15};
  myset.insert (myints, myints + 3);

  std::copy (myset.begin (), myset.end (),
             std::ostream_iterator<int> (std::cout, "\n"));
  return 0;
}
```

STL Pair Helper Class

- This template group is the basis for the `map` & `set` associative containers because it stores (potentially) heterogeneous pairs of data together
- A pair binds a key (known as the first element) with an associated value (known as the second element)

```
template <typename T, typename U>
struct pair {

  // Data members
  T first;
  U second;

  // Default constructor
  pair () {}

  // Constructor from values
  pair (const T& t, const U& u)
    : first (t), second (u) {}
};
```


STL Pair Helper Class (cont'd)

```
// Pair equivalence comparison operator
template <typename T, typename U>
inline bool
operator == (const pair<T, U>& lhs, const pair<T, U>& rhs)
{
  return lhs.first == rhs.first && lhs.second == rhs.second;
}

// Pair less than comparison operator
template <typename T, typename U>
inline bool
operator < (const pair<T, U>& lhs, const pair<T, U>& rhs)
{
  return lhs.first < rhs.first ||
         (!(rhs.first < lhs.first) && lhs.second < rhs.second);
}
```

STL Associative Container: Map

- An **std::map** associates a value with each unique key
  – a student's id number
- Its `value_type` is implemented as `pair<const Key, Data>`

```
#include <iostream>
#include <map>
#include <string>
#include <algorithm>

typedef std::map<std::string, int> My_Map;

struct print {
  void operator () (const My_Map::value_type &p)
  { std::cout << p.second << " "
              << p.first << std::endl; }
};

int main() {
  My_Map my_map;
  for (std::string a_word;
       std::cin >> a_word; ) my_map[a_word]++;
  std::for_each (my_map.begin(),
                 my_map.end(), print ());
  return 0;
}
```


STL Associative Container: MultiSet & MultiMap

- An **std::multiset** or an **std::multimap** can support multiple equivalent (non-unique) keys
  – *e.g., student first names or last names*
- Uniqueness is determined by an *equivalence* relation
  – *e.g.,* `strncmp()` *might treat last names that are distinguishable by* `strcmp()` *as being the same*

STL Associative Container: MultiSet Example

```
#include <set>
#include <iostream>
#include <iterator>

int main()    {
  const int N = 10;
  int a[N] = {4, 1, 1, 1, 1, 1, 0, 5, 1, 0};
  int b[N] = {4, 4, 2, 4, 2, 4, 0, 1, 5, 5};

  std::multiset<int> A(a, a + N);
  std::multiset<int> B(b, b + N);
  std::multiset<int> C;
  std::cout << "Set A: ";
  std::copy(A.begin(), A.end(), std::ostream_iterator<int>(std::cout, " "));
  std::cout << std::endl;

  std::cout << "Set B: ";
  std::copy(B.begin(), B.end(), std::ostream_iterator<int>(std::cout, " "));
  std::cout << std::endl;
```


STL Associative container: MultiSet Example (cont'd)

```
  std::cout << "Union: ";
  std::set_union(A.begin(), A.end(), B.begin(), B.end(),
                 std::ostream_iterator<int>(std::cout, " "));
  std::cout << std::endl;

  std::cout << "Intersection: ";
  std::set_intersection(A.begin(), A.end(), B.begin(), B.end(),
                        std::ostream_iterator<int>(std::cout, " "));
  std::cout << std::endl;
  std::set_difference(A.begin(), A.end(), B.begin(), B.end(),
                      std::inserter(C, C.end()));
  std::cout << "Set C (difference of A and B): ";
  std::copy(C.begin(), C.end(), std::ostream_iterator<int>(std::cout, " "));
  std::cout << std::endl;
  return 0;
}
```

STL Iterator Overview

- STL iterators are a C++ implementation of the *Iterator pattern*
  - This pattern provides access to the elements of an aggregate object sequentially without exposing its underlying representation
  - An Iterator object encapsulates the internal structure of how the iteration occurs
- STL iterators are a generalization of pointers, i.e., they are objects that point to other objects
- Iterators are often used to iterate over a range of objects: if an iterator points to one element in a range, then it is possible to increment it so that it points to the next element

STL Iterator Overview (cont'd)

- Iterators are central to generic programming because they are an interface between containers & algorithms
  - Algorithms typically take iterators as arguments, so a container need only provide a way to access its elements using iterators
  - This makes it possible to write a generic algorithm that operates on many different kinds of containers, even containers as different as a vector & a doubly linked list

Simple STL Iterator Example

```
#include <iostream>
#include <vector>
#include <string>

int main (int argc, char *argv[]) {
  std::vector <std::string> projects;  // Names of the projects

  for (int i = 1; i < argc; ++i)
      projects.push_back (std::string (argv [i]));

  for (std::vector<std::string>::iterator j = projects.begin ();
       j != projects.end (); ++j)
      std::cout << *j << std::endl;
  return 0;
}
```


STL Iterator Categories

- Iterator *categories* depend on type parameterization rather than on inheritance: allows algorithms to operate seamlessly on both native (i.e., pointers) & user-defined iterator types
- Iterator categories are hierarchical, with more refined categories adding constraints to more general ones
  - *Forward* iterators are both *input* & *output* iterators, but not all *input* or *output* iterators are *forward* iterators
  - *Bidirectional* iterators are all *forward* iterators, but not all *forward* iterators are *bidirectional* iterators
  - All *random access* iterators are *bidirectional* iterators, but not all *bidirectional* iterators are *random access* iterators
- Native types (i.e., pointers) that meet the requirements can be used as iterators of various kinds

STL Input Iterators

- *Input* iterators are used to read values from a sequence
- They may be dereferenced to refer to some object & may be incremented to obtain the next iterator in a sequence
- An *input* iterator must allow the following operations
  - Copy ctor & assignment operator for that same iterator type
  - Operators == & != for comparison with iterators of that type
  - Operators * (can be const) & ++ (both prefix & postfix)

STL Input Iterator Example

```
// Fill a vector with values read from standard input.
std::vector<int> v;
for (istream_iterator<int> i = cin;
     i != istream_iterator<int> ();
     ++i)
  v.push_back (*i);

// Fill vector with values read from stdin using std::copy()
std::vector<int> v;
std::copy (std::istream_iterator<int>(std::cin),
           std::istream_iterator<int>(),
           std::back_inserter (v));
```

STL Output Iterators

- *Output* iterator is a type that provides a mechanism for storing (but not necessarily accessing) a sequence of values
- *Output* iterators are in some sense the converse of Input Iterators, but have a far more restrictive interface:
  - Operators = & == & != need not be defined (but could be)
  - Must support non-const operator * (*e.g., *iter = 3)*
- Intuitively, an *output* iterator is like a tape where you can write a value to the current location & you can advance to the next location, but you cannot read values & you cannot back up or rewind

STL Output Iterator Example

```
// Copy a file to cout via a loop.
std::ifstream ifile ("example_file");
int tmp;
while (ifile >> tmp) std::cout << tmp;

// Copy a file to cout via input & output iterators
std::ifstream ifile ("example_file");
std::copy (std::istream_iterator<int> (ifile),
           std::istream_iterator<int> (),
           std::ostream_iterator<int> (std::cout));
```

STL Forward Iterators

- *Forward* iterators must implement (roughly) the union of requirements for *input* & *output* iterators, plus a default ctor
- The difference from the *input* & *output* iterators is that for two *forward* iterators `r` & `s`, `r==s` implies `++r==++s`
- A difference to the *output* iterators is that `operator*` is also valid on the left side of `operator=` (`*it = v` is valid) & that the number of assignments to a *forward* iterator is not restricted

STL Forward Iterator Example

```
template <typename ForwardIterator, typename T>
void replace (ForwardIterator first, ForwardIterator last,
              const T& old_value, const T& new_value) {
   for (; first != last; ++first)
      if (*first == old_value) *first = new_value;
}

// Iniitalize 3 ints to default value 1
std::vector<int> v (3, 1);
v.push_back (7);        // vector v: 1 1 1 7
replace (v.begin(), v.end(), 7, 1);
assert (std::find (v.begin(), v.end(), 7) == v.end());
```

STL Bidirectional Iterators

- *Bidirectional* iterators allow algorithms to pass through the elements forward & backward
- *Bidirectional* iterators must implement the requirements for *forward* iterators, plus decrement operators (prefix & postfix)
- Many STL containers implement *bidirectional* iterators
  - *e.g.*, `list`, `set`, `multiset`, `map`, & `multimap`

STL Bidirectional Iterator Example

```
template <typename BidirectionalIterator, typename Compare>
void bubble_sort (BidirectionalIterator first, BidirectionalIterator last,
                  Compare comp) {
  BidirectionalIterator left_el = first, right_el = first;
  ++right_el;
  while (first != last)
  {
    while (right_el != last) {
      if (comp(*right_el, *left_el)) std::swap (left_el, right_el);
      ++right_el;
      ++left_el;
    }
    --last;
    left_el = first, right_el = first;
    ++right_el;
  }
}
```

STL Random Access Iterators

- *Random access* iterators allow algorithms to have random access to elements stored in a container that provides *random access* iterators
  - *e.g.*, `vector` & `deque`
- *Random access* iterators must implement the requirements for *bidirectional* iterators, plus:
  - Arithmetic assignment operators += & -=
  - Operators + & - (must handle symmetry of arguments)
  - Ordering operators < & > & <= & >=
  - Subscript operator [ ]

STL Random Access Iterator Example

```
std::vector<int> v (1, 1);
v.push_back (2); v.push_back (3); v.push_back (4); // vector v: 1 2 3 4

std::vector<int>::iterator i = v.begin();
std::vector<int>::iterator j = i + 2; cout << *j << " ";
i += 3; std::cout << *i << " ";
j = i - 1; std::cout << *j << " ";
j -= 2;
std::cout << *j << " ";
std::cout << v[1] << endl;

(j < i) ? std::cout << "j < i" : std::cout << "not (j < i)";
std::cout << endl;
(j > i) ? std::cout << "j > i" : std::cout << "not (j > i)";
std::cout << endl;
i = j;
i <= j && j <= i ? std::cout << "i & j equal" :
                   std::cout << "i & j not equal"; std::cout << endl;
```

Implementing Iterators Using STL Patterns

- Since a C++ iterator provides a familiar, standard interface, at some point you will want to add one to your own classes so you can "plug-&and-play with STL algorithms
- Writing your own iterators is a straightforward (albeit *tedious* process, with only a couple of subtleties you need to be aware of, *e.g.*, which category to support, etc.
- Some good articles on using & writing STL iterators appear at
  - `http://www.oreillynet.com/pub/a/network/2005/10/18/what-is-iterator-in-c-plus-plus.html`
  - `http://www.oreillynet.com/pub/a/network/2005/11/21/what-is-iterator-in-c-plus-plus-part2.html`

STL Generic Algorithms

- Algorithms operate over *iterators* rather than containers
- Each container declares an `iterator` & `const_iterator` as a trait
  - `vector` & `deque` declare *random access* iterators
  - `list`, `map`, `set`, `multimap`, & `multiset` declare *bidirectional* iterators
- Each container declares factory methods for its iterator type:
  - `begin()`, `end()`, `rbegin()`, `rend()`
- Composing an algorithm with a container is done simply by invoking the algorithm with iterators for that container
- Templates provide compile-time type safety for combinations of containers, iterators, & algorithms

Categorizing STL Generic Algorithms

- There are various ways to categorize STL algorithms, *e.g.*:
  - **Non-mutating**, which operate using a range of iterators, but don.t change the data elements found
  - **Mutating**, which operate using a range of iterators, but can change the order of the data elements
  - **Sorting & sets**, which sort or searches ranges of elements & act on sorted ranges by testing values
  - **Numeric**, which are mutating algorithms that produce numeric results
- In addition to these main types, there are specific algorithms within each type that accept a predicate condition
  - Predicate names end with the `_if` suffix to remind us that they require an "if" test.s result (true or false), as an argument; these can be the result of functor calls

Benefits of STL Generic Algorithms

- STL algorithms are decoupled from the particular containers they operate on & are instead parameterized by iterators
- All containers with the same iterator type can use the same algorithms
- Since algorithms are written to work on iterators rather than components, the software development effort is drastically reduced
  - *e.g.*, instead of writing a search routine for each kind of container, one only write one for each iterator type & apply it any container.
- Since different components can be accessed by the same iterators, just a few versions of the search routine must be implemented

Example of std::find() Algorithm

Returns a *forward* iterator positioned at the first element in the given sequence range that matches a passed value

```
#include <vector>
#include <algorithm>
#include <assert>
#include <string>

int main (int argc, char *argv[]) {
  std::vector <std::string> projects;
  for (int i = 1; i < argc; ++i)
    projects.push_back (std::string (argv [i]));

  std::vector<std::string>::iterator j =
    std::find (projects.begin (), projects.end (), std::string ("Lab8"));

  if (j == projects.end ()) return 1;
  assert ((*j) == std::string ("Lab8"));
  return 0;
}
```


Example of std::find() Algorithm (cont'd)

STL algorithms can work on both built-in & user-defined types

```
int a[] = {10, 30, 20, 15};
int *ibegin = a;
int *iend =
  a + (sizeof (a)/ sizeof (*a));
int *iter =
  std::find (ibegin, iend, 10);
if (iter == iend)
  std::cout << "10 not found\n";
else
  std::cout << *iter << " found\n";
```

```
int A[] = {10, 30, 20, 15};
std::set<int> int_set
  (A, A + (sizeof (A)/ sizeof (*A)));

std::set<int>::iterator iter =
  // int_set.find (10) will be faster!
  std::find (int_set.begin (),
             int_set.end (), 10);
if (iter == int_set.end ())
  std::cout << "10 not found\n";
else
  std::cout << *iter << " found\n";
```

Example std::adjacent_find() Algorithm

Returns the first iterator `i` such that `i` & `i + 1` are both valid iterators in `[first, last)`, & such that `*i == *(i+1)` or `binary_pred (*i, *(i+1))` is true (it returns `last` if no such iterator exists)

```
// Find the first element that is greater than its successor:
int A[] = {1, 2, 3, 4, 6, 5, 7, 8};
const int N = sizeof(A) / sizeof(int);

const int *p = std::adjacent_find(A, A + N, std::greater<int>());

std::cout << "Element " << p - A << " is out of order: "
          << *p << " > " << *(p + 1) << "." << std::endl;
```


Example of std::copy() Algorithm

Copies elements from a input iterator sequence range into an output iterator

```
std::vector<int> v;
std::copy (std::istream_iterator<int>(std::cin),
           std::istream_iterator<int>(),
           std::back_inserter (v));

std::copy (v.begin (),
           v.end (),
           std::ostream_iterator<int> (std::cout));
```

Example of std::fill() Algorithm

Assign a value to the elements in a sequence

```
int a[10];
std::fill (a, a + 10, 100);
std::fill_n (a, 10, 200);

std::vector<int> v (10, 100);
std::fill (v.begin (), v.end (), 200);
std::fill_n (v.begin (), v.size (), 200);
```


Example of std::replace() Algorithm

Replaces all instances of a given existing value with a given new value, within a given sequence range

```
std::vector<int> v;
v.push_back(1);
v.push_back(2);
v.push_back(3);
v.push_back(1);

std::replace (v.begin (), v.end (), 1, 99);
assert (V[0] == 99 && V[3] == 99);
```

Example of std::remove() Algorithm

Removes from the range `[first, last)` the elements with a value equal to value & returns an iterator to the new end of the range, which now includes only the values not equal to value

```
#include <iostream>
#include <algorithm>
#include <iterator>

int main () {
  int myints[] = {10, 20, 30, 30, 20, 10, 10, 20};
  int *pbegin = myints, *pend = myints + sizeof myints / sizeof *myints;
  std::cout << "original array contains:";
  std::copy (pbegin, pend, std::ostream_iterator<int> (std::cout, " "));
  int *nend = std::remove (pbegin, pend, 20);
  std::cout << "\nrange contains:";
  std::copy (pbegin, nend, std::ostream_iterator<int> (std::cout, " "));
  std::cout << "\ncomplete array contains:";
  std::copy (pbegin, pend, std::ostream_iterator<int> (std::cout, " "));
  std::cout << std::endl;
  return 0;
}
```


Example of std::remove_if() Algorithm

Removes from the range `[first, last)` the elements for which pred applied to its value is true, & returns an iterator to the new end of the range, which now includes only the values for which pred was false.

```
#include <iostream>
#include <algorithm>

struct is_odd { // Could also be a C-style function.
  bool operator () (int i) { return (i%2)==1; }
};

int main () {
  int myints[] = {1, 2, 3, 4, 5, 6, 7, 8, 9};
  int *pbegin = myints;
  int *pend = myints + sizeof myints / sizeof *myints;
  pend = std::remove_if (pbegin, pend, is_odd ());
  std::cout << "range contains:";
  std::copy (pbegin, pend, std::ostream_iterator<int> (std::cout, " "));
  std::cout << std::endl;
  return 0;
}
```

Example of std::transform() Algorithm

Scans a range & for each use a function to generate a new object put in a second container *or* takes two intervals & applies a binary operation to items to generate a new container

```
#include <iostream>
#include <algorithm>
#include <ctype.h>
#include <functional>

class to_lower {
public:
  char operator() (char c) const
  {
    return isupper (c)
      ? tolower(c) : c;
  }
};
```

```
std::string lower (const std::string &str) {
  std::string lc;
  std::transform (str.begin (), str.end (),
                  std::back_inserter (lc),
                  to_lower ());
  return lc;
}

int main () {
  std::string s = "HELLO";
  std::cout << s << std::endl;
  s = lower (s);
  std::cout << s << std::endl;
}
```


Another Example of std::transform() Algorithm

```
#include <iostream>
#include <algorithm>
#include <functional>
#include <numeric>
#include <vector>
#include <iterator>

int main() {
   std::vector<float> v (5, 1); // a vector of 5 floats all initialized to 1.0.
   std::partial_sum (v.begin(), v.end(), v.begin());

   std::transform(v.begin(), v.end(), v.begin(),
                  v.begin(), std::multiplies<float>());
   std::copy (v.begin (), v.end (), std::ostream_iterator<float> (std::cout, "\n"));

   std::transform(v.begin(),v.end(), v.begin (),
                  std::bind2nd(std::divides<float>(), 3));
   std::copy (v.begin (), v.end (), std::ostream_iterator<float> (std::cout, "\n"));
   return 0;
}
```

Example of std::for_each() Algorithm

Applies the function object `f` to each element in the range `[first, last)`; `f`'s return value, if any, is ignored

```
template<class T>
struct print {
  print (std::ostream &out): os_(out), count_(0) {}
  void operator() (const T &t) { os << t << ' '; ++count_; }
  std::ostream &os_;
  int count_;
};

int main() {
  int A[] = {1, 4, 2, 8, 5, 7};
  const int N = sizeof(A) / sizeof(int);

  // for_each() returns function object after being applied to each element
  print<int> f = std::for_each (A, A + N, print<int>(std::cout));
  std::cout << std::endl << f.count_ << " objects printed." << std::endl;
}
```


STL Function Objects

- Function objects (aka *functors*) declare & define `operator()`
- STL provides helper base class templates `unary_function` & `binary_function` to facilitate user-defined function objects
- STL provides a number of common-use function object class templates:
  - **Arithmetic**: `plus`, `minus`, `times`, `divides`, `modulus`, `negate`
  - **comparison**: `equal_to`, `not_equal_to`, `greater`, `less`, `greater_equal`, `less_equal`
  - **logical**: `logical_and`, `logical_or`, `logical_not`
- A number of STL generic algorithms can take STL-provided or user-defined function object arguments to extend algorithm behavior

STL Function Objects Example

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

int main (int argc, char *argv[])
{
  std::vector <std::string> projects;

  for (int i = 0; i < argc; ++i)
    projects.push_back (std::string (argv [i]));

  // Sort in descending order: note explicit ctor for greater
  std::sort (projects.begin (), projects.end (),
             std::greater<std::string> ());

  return 0;
}
```


STL Adaptors

- STL adaptors implement the *Adapter* design pattern
  - *i.e., they convert one interface into another interface clients expect*
- Container adaptors include `stack`, `queue`, `priority_queue`
- Iterator adaptors include `reverse_iterators` & `back_inserter()` iterators
- Function adaptors include negators & binders
- STL adaptors can be used to *narrow* interfaces (*e.g., a* `stack` *adaptor for* `vector`*)*

STL Container Adaptors

- The `stack` container adaptor is an ideal choice when one need to use a "Last In, First Out" (LIFO) data structure characterized by having elements inserted & removed from the same end
- The `queue` container adaptor is a "First In, First Out" (FIFO) data structure characterized by having elements inserted into one end & removed from the other end
- The `priority_queue` assigns a priority to every element that it stores
  - New elements are added to the queue using the `push()` function, just as with a `queue`
  - However, its `pop()` function gets element with the highest priority

STL stack & queue Container Adaptor Definitions

```
template <typename T,
          typename ST = deque<T> >
class stack
{
public:
  explicit stack(const ST& c = ST());
  bool empty() const;
  size_type size() const;
  value_type& top();
  const value_type& top() const;
  void push(const value_type& t);
  void pop();

private :
   ST container_ ;
   //.
};
```

```
template <typename T,
          typename Q = deque<T> >
class queue
{
public:
  explicit queue(const Q& c = Q());
  bool empty() const;
  size_type size() const;
  value_type& front();
  const value_type& front() const;
  value_type& back();
  const value_type& back() const;
  void push(const value_type& t);
  void pop();

private:
  Q container_;
  // .
};
```

STL stack & queue Container Adaptor Examples

```
// STL stack
#include <iostream>
#include <stack>

int main()   {
  std::stack<char> st;
  st.push ('A');
  st.push ('B');
  st.push ('C');
  st.push ('D');

  for (; !st.empty (); st.pop ()) {
    cout << "\nPopping: ";
    cout << st.top();
  }
  return 0;
}
```

```
// STL queue
#include <iostream>
#include <queue>
#include <string>
int main()  {
  std::queue<string> q;
  std::cout << "Pushing one two three \n";
  q.push ("one");
  q.push ("two");
  q.push ("three");

  for (; !q.empty (); q.pop ()) {
    std::cout << "\nPopping ";
    std::cout << q.front ();
  }
  return 0;
}
```


STL priority_queue Container Adaptor Example

```
#include <queue> // priority_queue
#include <string>
#include <iostream>

struct Place {
  unsigned int dist;  std::string dest;
  Place (const std::string dt, size_t ds) : dist(ds), dest(dt) {}
  bool operator< (const Place &right) const { return dist < right.dist; }
};

std::ostream &operator << (std::ostream &os, const Place &p)
{ return os << p.dest << " " << p.dist; }

int main () {
  std::priority_queue <Place> pque;
  pque.push (Place ("Poway",   10));
  pque.push (Place ("El Cajon",  20));
  pque.push (Place ("La Jolla", 3));
  for (; !pque.empty (); pque.pop ()) std::cout << pque.top() << std::endl;
  return 0;
}
```

STL Iterator Adaptors

- STL algorithms that copy elements are passed an iterator that marks the position within a container to begin copying
  - *e.g.*, `copy()`, `unique_copy()`, `copy_backwards()`, `remove_copy()`, & `replace_copy()`
- With each element copied, the value is assigned & the iterator is incremented
- Each copy requires the target container is of a sufficient size to hold the set of assigned elements
- We can use iterator adapters to expand the containers as we perform the algorithm
  - Start with an empty container, & use the inserter along with the algorithms to make the container grow only as needed

STL back_inserter() Iterator Adaptor Example

- `back_inserter()` causes the container's `push_back()` operator to be invoked in place of the assignment operator
- The argument passed to `back_inserter()` is the container itself

```
// Fill vector with values read
// from stdin using std::copy()
std::vector<int> v;
std::vector<int>::iterator in_begin =
  std::istream_iterator<int>(std::cin)
std::vector<int>::iterator in_end =
  std::istream_iterator<int>(),
std::copy (in_begin,
           in_end,
           std::back_inserter (v));
```

STL Function Adaptors

- STL has predefined functor adaptors that will change their functors so that they can:
  - Perform function composition & binding
  - Allow fewer created functors
- These functors allow one to combine, transform or manipulate functors with each other, certain values or with special functions
- STL function adapters include
  - Binders (`bind1st()` & `bind2nd()`) bind one of their arguments
  - Negators (`not1` & `not2`) adapt functors by negating arguments
  - Member functions (`ptr_fun` & `mem_fun`) allow functors to be class members

STL Binder Function Adaptor

- A binder can be used to transform a binary functor into an unary one by acting as a converter between the functor & an algorithm
- Binders always store both the binary functor & the argument internally (the argument is passed as one of the arguments of the functor every time it is called)
  - `bind1st(Op, Arg)` calls 'Op' with 'Arg' as its first parameter
  - `bind2nd(Op, Arg)` calls 'Op' with 'Arg' as its second parameter

STL Binder Function Adaptor Example 1

```
#include <vector>
#include <iostream>
#include <algorithm>
#include <numeric>
#include <functional>

int main (int argc, char *argv[]) {
  std::vector<int> v (10, 2);
  std::partial_sum (v.begin (), v.end (), v.begin ());
  std::random_shuffle (v.begin (), v.end ());
  std::copy (v.begin (), v.end (), std::ostream_iterator<int> (std::cout, "\n"));
  std::cout << "number greater than 10 = "
            << count_if (v.begin (), v.end (),
                         std::bind2nd (std::greater<int>(), 10)) << std::endl;
  return 0;
}
```


STL Binder Function Adaptor Example 2

```
#include <vector>
#include <iostream>
#include <algorithm>
#include <iterator>
#include <functional>
#include <cstdlib>
#include <ctime>

int main (int argc, char *argv[]) {
  srand (time(0));
  std::vector<int> v, v2 (10, 20);
  std::generate_n (std::back_inserter (v), 10, rand);
  std::transform (v.begin (), v.end (), v2.begin (), v.begin (), std::modulus<int>());
  std::copy (v.begin (), v.end (), std::ostream_iterator<int> (std::cout, "\n"));
  std::cout << std::endl;
  int factor = 2;
  std::transform (v.begin (), v.end (),
                  v.begin(), std::bind2nd (std::multiplies<int> (), factor));
  std::copy (v.begin (), v.end (), std::ostream_iterator<int> (std::cout, "\n"));
  return 0;
}
```

STL Binder Function Adaptor Example 3

This example removes spaces in a string that uses the `equal_to` and `bind2nd` functors to perform `remove_if` when `equal_to` finds a blank char in the string

```
#include <iostream>
#include <string>

int main() {
  std::string s = "spaces in text";
  std::cout << s << std::endl;
  std::string::iterator new_end =
    std::remove_if (s.begin (), s.end (), std::bind2nd (std::equal_to<char>(), ' '));

  // remove_if() just moves unwanted elements to the end and returns an iterator
  // to the first unwanted element since it'.s a generic algorithm & doesn't "know"
  // whether the container be changed.  s.erase() *does* know this, however..
  s.erase (new_end, s.end ());
  std::cout << s << std::endl;
  return 0;
}
```


STL Binder Function Adaptor Example 4

```
#include <vector>
#include <algorithm>
#include <functional>
#include <iostream>
#include <iterator>

int main()  { // Contrasts std::remove_if() & erase().
   std::vector<int> v;
   v.push_back (1); v.push_back (4); v.push_back (2);
   v.push_back (8); v.push_back (5); v.push_back (7);
   std::copy (v.begin (), v.end (), std::ostream_iterator<int> (std::cout, " "));
   int sum = std::count_if (v.begin (), v.end (),
                            std::bind2nd (std::greater<int>(), 5));
   std::cout << "\nThere are " << sum << " number(s) greater than 5" << std::endl;
   std::vector<int>::iterator new_end = // "remove" all the elements less than 4.
     std::remove_if (v.begin (), v.end (), std::bind2nd (std::less<int>(), 4));
  v.erase (new_end, v.end ());
  std::copy (v.begin (), v.end (), std::ostream_iterator<int> (std::cout, " "));
  std::cout << "\nElements less than 4 removed" << std::endl;
  return 0;
}
```

STL Negator Adapters & Function Adaptors

- A negator can be used to store the opposite result of a functor
  - `not1(Op)` negates the result of unary 'Op'
  - `not2(Op)` negates result of binary 'Op'
- A member function & pointer-to-function adapter can be used to allow class member functions or C-style functions as arguments to STL predefined algorithms
  - `mem_fun(PtrToMember mf)` converts a pointer to member to a functor whose first arg is a pointer to the object
  - `ptr_fun()` converts a pointer to a function & turns it into a functor

STL Negator Function Adaptor Example

```
#include <vector>
#include <iostream>
#include <iterator>
#include <algorithm>
#include <functional>

int main() {
  std::vector<int> v1;
  v1.push_back (1); v1.push_back (2); v1.push_back (3); v1.push_back (4);
  std::vector<int> v2;
  std::remove_copy_if (v1.begin(), v1.end(), std::back_inserter (v2),
                       std::bind2nd (std::greater<int> (), 3));
  std::copy (v2.begin(), v2.end (),
             std::ostream_iterator<int> (std::cout, "\n"));
  std::vector<int> v3;
  std::remove_copy_if (v1.begin(), v1.end(), std::back_inserter (v3),
                       std::not1 (std::bind2nd (std::greater<int> (), 3)));
  std::copy (v3.begin(), v3.end (),
             std::ostream_iterator<int> (std::cout, "\n"));
  return 0;
}
```

STL Pointer-to-MemFun Adaptor Example

```
class WrapInt {
public:
  WrapInt (): val_ (0) {}
  WrapInt(int x): val_ (x) {}

  void showval() {
    std::cout << val_ << " ";
  }

  bool is_prime() {
    for (int i = 2; i <= (val_ / 2); i++)
      if ((val_ % i) == 0)
        return false;
    return true;
  }
private:
  int val_;
};
```


STL Pointer-to-MemFun Adaptor Example (cont'd)

```
int main() {
  std::vector<WrapInt> v (10);

  for (int i = 0; i <10; i++)
    v[i] = WrapInt (i+1);

  std::cout << "Sequence contains: ";
  std::for_each (v.begin (), v.end (),
       std::mem_fun_ref (&WrapInt::showval));
  std::cout << std::endl;

  std::vector<WrapInt>::iterator end_p = // remove the primes
  std::remove_if (v.begin(), v.end(),
                  std::mem_fun_ref (&WrapInt::is_prime));

  std::cout << "Sequence after removing primes: ";
  for_each (v.begin (), end_p, std::mem_fun_ref (&WrapInt::showval));
  std::cout << std::endl;

  return 0;
}
```

STL Pointer-to-Function Adaptor Example

```
#include <vector>
#include <iostream>
#include <iterator>
#include <algorithm>
#include <functional>

int main () {
  std::vector<char *> v;
  v.push_back ("One"); v.push_back ("Two"); v.push_back ("Three"); v.push_back ("Four");

  std::cout << "Sequence contains:";
  std::copy (v.begin (), v.end (), std::ostream_iterator<char *> (std::cout, " "));
  std::cout << std::endl << "Searching for Three.\n";
  std::vector<char *>::iterator it = std::find_if (v.begin (), v.end (),
                       std::not1 (std::bind2nd (std::ptr_fun (strcmp), "Three")));
  if (it != v.end ()) {
    std::cout << "Found it! Here is the rest of the story:";
    std::copy (it, v.end (), std::ostream_iterator<char *> (std::cout, "\n"));
  }
  return 0;
}
```


STL Utility Operators

```
template <typename T, typename U>
inline bool
operator != (const T& t, const U& u)
{
  return !(t == u);
}

template <typename T, typename U>
inline bool
operator > (const T& t, const U& u)
{
  return u < t;
}
```

STL Utility Operators (cont'd)

```
template <typename T, typename U>
inline bool
operator <= (const T& t, const U& u)
{
  return !(u < t);
}

template <typename T, typename U>
inline bool
operator >= (const T& t, const U& u)
{
  return !(t < u);
}
```


STL Utility Operators (cont'd)

- Question: why require that parameterized types support operator == as well as operator $<$?
  - Operators $>$ & $>=$ & $<=$ are implemented only in terms of operator $<$ on u & t (and ! on boolean results)
  - Could implement operator == as `!(t < u) && !(u < t)` so classes T & U only had to provide operator $<$ & did not have to provide operator ==
- Answer: efficiency (*two* operator $<$ calls are needed to implement operator == implicitly)
- Answer: allows *equivalence classes* of *ordered* types

STL Example: Course Schedule

- Goals:
  - Read in a list of course names, along with the corresponding day(s) of the week & time(s) each course meets
    - Days of the week are read in as characters M,T,W,R,F,S,U
    - Times are read as unsigned decimal integers in 24 hour HHMM format, with no leading zeroes (*e.g., 11:59pm should be read in as 2359, & midnight should be read in as 0)*
  - Sort the list according to day of the week & then time of day
  - Detect any times of overlap between courses & print them out
  - Print out an ordered schedule for the week
- STL provides most of the code for the above

STL Example: Course Schedule (cont'd)

```
% cat infile
CS101 W 1730 2030
CS242 T 1000 1130
CS242 T 1230 1430
CS242 R 1000 1130
CS281 T 1300 1430
CS281 R 1300 1430
CS282 M 1300 1430
CS282 W 1300 1430
CS201 T 1600 1730
CS201 R 1600 1730
```

```
% cat infile | xargs main

CONFLICT:
 CS242 T 1230 1430
 CS281 T 1300 1430

CS282 M 1300 1430
CS242 T 1000 1130
CS242 T 1230 1430
CS281 T 1300 1430
CS201 T 1600 1730
CS282 W 1300 1430
CS101 W 1730 2030
CS242 R 1000 1130
CS281 R 1300 1430
CS201 R 1600 1730
```

STL Example: Course Schedule (cont'd)

```
struct Meeting {
  enum Day_Of_Week
    {MO, TU, WE, TH, FR, SA, SU};
  static Day_Of_Week
    day_of_week (char c);

  Meeting (const std::string &title,
           Day_Of_Week day,
           size_t start_time,
           size_t finish_time);
  Meeting (const Meeting & m);
  Meeting (char **argv);

  Meeting &operator =
    (const Meeting &m);
  bool operator <
    (const Meeting &m) const;
  bool operator ==
    (const Meeting &m) const;

  std::string title_;
  // Title of the meeting

  Day_Of_Week day_;
  // Week day of meeting

  size_t start_time_;
  // Meeting start time in HHMM format

  size_t finish_time_;
  // Meeting finish time in HHMM format
};

// Helper operator for output
std::ostream &
operator << (std::ostream &os,
             const Meeting & m);
```


STL Example: Course Schedule (cont'd)

```
Meeting::Day_Of_Week
Meeting::day_of_week (char c)
{
  switch (c) {
  case 'M': return Meeting::MO;
  case 'T': return Meeting::TU;
  case 'W': return Meeting::WE;
  case 'R': return Meeting::TH;
  case 'F': return Meeting::FR;
  case 'S': return Meeting::SA;
  case 'U': return Meeting::SU;
  default:
    assert (!"not a week day");
    return Meeting::MO;
  }
}

Meeting::Meeting
  (const std::string &title,
   Day_Of_Week day,
   size_t start, size_t finish)
 : title_ (title), day_ (day),
   start_time_ (start),
   finish_time_ (finish) {}

Meeting::Meeting (const Meeting &m)
 : title_ (m.title_), day_ (m.day_),
   start_time_ (m.start_time_),
   finish_time_ (m.finish_time_) {}

Meeting::Meeting (char **argv)
  : title_ (argv[0]),
    day_ (Meeting::day_of_week (*argv[1])),
    start_time_ (atoi (argv[2])),
    finish_time_ (atoi (argv[3])) {}
```

STL Example: Course Schedule (cont'd)

```
Meeting &Meeting::operator =
  (const Meeting &m) {
  title_ = m.title_;
  day_ = m.day_;
  start_time_ = m.start_time_;
  finish_time_ = m.finish_time_;
  return *this;
}
bool Meeting::operator ==
  (const Meeting &m) const {
  return
    (day_ == m.day_ &&
     ((start_time_ <= m.start_time_ &&
       m.start_time_ <= finish_time_) ||
      (m.start_time_ <= start_time_ &&
       start_time_ <= m.finish_time_)))
    ? true : false;
}
```

```
bool Meeting::operator <
  (const Meeting &m) const
{
  return
    day_ < m.day_
    ||
    (day_ == m.day_
     &&
     start_time_ < m.start_time_)
    ||
    (day_ == m.day_
     &&
     start_time_ == m.start_time_
     &&
     finish_time_ < m.finish_time_)
    ? true : false;
}
```


STL Example: Course Schedule (cont'd)

```
std::ostream &operator <<
  (std::ostream &os,
   const Meeting &m) {
  const char *d = "   ";
  switch (m.day_) {
  case Meeting::MO: d="M "; break;
  case Meeting::TU: d="T "; break;
  case Meeting::WE: d="W "; break;
  case Meeting::TH: d="R "; break;
  case Meeting::FR: d="F "; break;
  case Meeting::SA: d="S "; break;
  case Meeting::SU: d="U "; break;
  }
  return
    os << m.title_ << " " << d
       << m.start_time_ << " "
       << m.finish_time_;
}
```

```
struct print_conflicts {
  print_conflicts (std::ostream &os)
   : os_ (os) {}

  Meeting operator () (const Meeting &lhs,
    const Meeting &rhs) {
    if (lhs == rhs)
      os_ << "CONFLICT:" << std::endl
          << " " << lhs << std::endl
          << " " << rhs << std::endl
          << std::endl;
    return lhs;
  }
  std::ostream &os_;
};
```

STL Example: Course Schedule (cont'd)

```
template <typename T>
class argv_iterator : public std::iterator <std::forward_iterator_tag, T> {
public:
  argv_iterator (void) {}
  argv_iterator (int argc, char **argv, int increment)
    : argc_ (argc), argv_ (argv), base_argv_ (argv), increment_ (increment) {}

  argv_iterator begin () { return *this; }
  argv_iterator end () { return *this; }

  bool operator != (const argv_iterator &) { return argv_ != (base_argv_ + argc_); }

  T operator *() { return T (argv_); }
  void operator++ () { argv_ += increment_; }

private:
  int argc_;
  char **argv_, **base_argv_;
  int increment_;
};
```


STL Example: Course Schedule (cont'd)

```
int main (int argc, char *argv[]) {
  std::vector<Meeting> schedule;

  std::copy (argv_iterator<Meeting> (argc - 1, argv + 1, 4),
             argv_iterator<Meeting> (),
             std::back_inserter (schedule));

  std::sort (schedule.begin (), schedule.end (), std::less<Meeting> ());

  // Find & print out any conflicts.
  std::transform (sched.begin (), sched.end () - 1,
                  sched.begin () + 1,
                  sched.begin (),
                  print_conflicts (std::cout));

  // Print out schedule, using STL output stream iterator adapter.
  std::copy (sched.begin (), sched.end (),
             std::ostream_iterator<Meeting> (os, "\n"));
  return 0;
}
```

Summary of the Class Scheduling Example

- STL promotes *software reuse*: writing less, doing more
  - Effort focused on the `Meeting` class
  - STL provided algorithms (*e.g.*, sorting & copying), containers, iterators, & functors
- STL is *flexible*, according to open/closed principle
  - `std::copy()` algorithm with output iterator prints schedule
  - Sort in ascending (default `std::less`) or descending (via `std::greater`) order
- STL is *efficient*
  - STL inlines methods, uses templates extensively
  - Optimized both for performance & for programming model complexity (*e.g., requiring* $<$ *&* $==$ *& no others)*

References: For More Information on STL

- David Musser's STL page
  - http://www.cs.rpi.edu/ musser/stl.html
- Stepanov & Lee, "The Standard Template Library"
  - http://www.cs.rpi.edu/ musser/doc.ps
- SGI STL Programmer's Guide
  - http://www.sgi.com/Technology/STL/
- Musser & Saini, "STL Tutorial & Reference Guide"
  - ISBN 0-201-63398-1
- Austern, "Generic Programming & the STL"
  - ISBN 0-201-30956-4