

The C++ Standard Template Library

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



- What is the STL?
- Generic programming: why use the STL?
- STL overview: helper class & function templates, containers, iterators, generic algorithms, function objects, adaptors
- STL examples
- Conclusions: writing less, doing more
- References for more information on the STL



What is the 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 the 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 the STL?

- Reuse: “write less, do more”
 - The 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 Overview: helper operators

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

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



STL Overview: helper operators (cont'd)

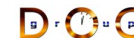
```
template <class T, class U>
inline bool
operator <= (const T& t, const U& u)
{
    return !(u < t);
}
```

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



STL Overview: helper 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 Overview: operators example

```

class String
{
public:
    String (const char *s)
        : s_(s) {}
    String (const String &s)
        : s_(s.s_) {}
    bool operator < (const String &s) const
    {return
        (strcmp (this->s_, s.s_) < 0)
        ? true : false;}
    bool operator == (const String &s) const
    {return
        (strcmp (this->s_, s.s_) == 0)
        ? true : false;}
    const char *c_str () { return s_; }
private:
    const char * s_;
};

```

```

#include <iostream>

int
main (int, char *[])
{
    const char * wp = "world";
    const char * hp = "hello";
    String w_str (wp);
    String h_str (hp);

    std::cout << false << std::endl; // 0
    std::cout << true << std::endl; // 1
    std::cout << (h_str < w_str) << std::endl;
    std::cout << (h_str == w_str) << std::endl;
    std::cout << (hp < wp) << std::endl;
    std::cout << (hp == wp) << std::endl;

    return 0;
}

```



STL Overview: pair helper class

```

template <class T, class 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 Overview: pair helper class (cont'd)

```

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

```

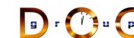


STL Overview: pair helper class (cont'd)

```

// Pair less than comparison operator
template <class T, class 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 Overview: pair example

```

class String
{
public:
    String (const char *s): s_ (s) {}
    String (const String &s): s_ (s.s_) {}
    bool
    operator < (const String &s) const
    {return
    (strcmp (this->s_, s.s_) < 0)
    ? true : false;}
    bool
    operator == (const String &s) const
    {return
    (strcmp (this->s_, s.s_) == 0)
    ? true : false;}
    const char *c_str () { return s_; }
private:
    const char *s_;
};
#include <iostream>
#include <pair>

int
main (int, char *[])
{
    std::pair<int, String>
    pair1 (3, String ("hello"));

    std::pair<int, String>
    pair2 (2, String ("world"));

    std::cout << (pair1 == pair2) << std::endl;

    std::cout << (pair1 < pair2) << std::endl;

    return 0;
}

```

STL Overview: containers, iterators, algorithms

- Containers:
 - Sequence: vector, deque, list
 - Associative: set, multiset, map, multimap
 - Iterator factories
- Iterators:
 - Input, output, forward, bidirectional, random access
 - Each container declares a trait for the type of iterator it provides
- Generic Algorithms:
 - Sequence (mutating & non-mutating), sorting, numeric

STL Overview: containers

- STL containers are Abstract Data Types (ADTs)
- All containers are parameterized by the type(s) they contain
- Sequence containers are ordered
- Associative containers are unordered
- Each container declares an *iterator* typedef (trait)
- Each container provides special factory methods for iterators: `begin()/end()` & `rbegin()/rend()`

STL Overview: sequence containers

- An **`std::vector`** can be used as an array & a stack
 - provides reallocation, indexed storage, `push_back`, `pop_back`
- An **`std::deque`** (pronounced “deck”) is a double ended queue
 - adds efficient insertion & removal at the *beginning* as well as at the end of the sequence
- An **`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

STL Overview: associative containers

- An `std::set` is an unordered collection of unique keys
 - e.g., a set of student id numbers
- An `std::map` associates a value with each unique key
 - e.g., a student's first name
- An `std::multiset` or an `std::multimap` can support multiple equivalent (non-unique) keys
 - e.g., student 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 Overview: container example

```
#include <iostream>
#include <vector>
#include "String.h"

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

    for (i = 1; i < argc; ++i) // Start with 1st arg
    {
        projects.push_back (String (argv [i]));
        std::cout << projects [i - 1].c_str () << std::endl;
    }

    return 0;
}
```

STL Overview: iterators

- STL iterators are a C++ implementation of the *Iterator pattern*
 - This pattern provides access 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 Overview: iterators

- 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

STL Overview: simple iterator example

```
#include <iostream>
#include <vector>
#include "String.h"

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

    for (int i = 1; i < argc; ++i) {
        projects.push_back (String (argv [i]));
    }

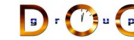
    for (std::vector<String>::iterator j = projects.begin ();
         j != projects.end (); ++j) {
        std::cout << (*j).c_ptr () << std::endl;
    }

    return 0;
}
```



STL Overview: 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 Overview: 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 Overview: input iterator example

```
// Fill a vector with values read from standard input.
std::vector<int> v;
std::copy (std::istream_iterator<int>(std::cin),
           std::istream_iterator<int>(),
           std::back_inserter (v));
```



STL Overview: 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 Overview: 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 and output iterators
std::ifstream ifile ("example_file");
copy (istream_iterator<int, ptrdiff_t> (ifile),
      istream_iterator<int, ptrdiff_t> (),
      ostream_iterator<int> (cout) );
```



STL Overview: forward iterators

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



STL Overview: forward iterator example

```
template<class ForwardIterator, class T>
ForwardIterator find_linear (ForwardIterator first,
                           ForwardIterator last, T& value) {
    while (first != last) if (*first++ == value) return first;
    return last;
}
```

```
vector<int> v (3, 1);
int value = 7;
```

```
v.push_back (7);           // vector v: 1 1 1 7
```

```
vector<int>::iterator i = find_linear (v.begin(), v.end(), value);
```



STL Overview: bidirectional iterators

- Bidirectional iterators allow algorithms to pass through the elements forward and backward
- Bidirectional iterators must implement the requirements for forward iterators, plus decrement operators (prefix & postfix)



STL Overview: bidirectional iterator example

```
template <class BidirectionalIterator, class 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)) iter_swap (left_el, right_el);
            right_el++;
            left_el++;
        }
        last--;
        left_el = first, right_el = first;
        right_el++;
    }
}
```



STL Overview: random access iterators

- Random access iterators allow algorithms to have random access to elements stored in a container which has to provide random access iterators, like the vector
- 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 Overview: random access iterator examples

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

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

```
(j < i) ? cout << "j < i" : cout << "not (j < i)"; cout << endl;
(j > i) ? cout << "j > i" : cout << "not (j > i)"; cout << endl;
```

```
i = j;
i <= j && j <= i ? cout << "i and j equal" : cout << "i and j not equal"; cout << endl;
```



STL Overview: generic algorithms

- Algorithms operate over *iterators* rather than containers
- Each container declares an 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



STL Overview: generic algorithms (cont'd)

- Some examples of STL generic algorithms:
 - std::find(): returns a forward iterator positioned at the first element in the given sequence range that matches a passed value
 - std::mismatch(): returns a pair of iterators positioned respectively at the first elements that do not match in two given sequence ranges
 - std::copy(): copies elements from a sequence range into an output iterator
 - std::fill(): assign a value to the elements in a sequence
 - std::replace(): replaces all instances of a given existing value with a given new value, within a given sequence range
 - std::random_shuffle(): shuffles the elements in the given sequence range



STL Overview: generic algorithm example

```
#include <vector>
#include <algo>
#include <assert>
#include "String.h"

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

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

    if (j == projects.end ())
        return 1;

    assert ((*j) == String ("Lab8"));
    return 0;
}
```



STL Overview: function objects

- Function objects (aka *functors*) declare & define operator ()
- STL provides helper base class templates unary_function and binary_function to facilitate writing 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 Overview: function objects example

```
#include <vector>
#include <algo>
#include <function>
#include "String.h"

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

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

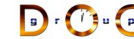
    // Sort in descending order: note explicit ctor for greater
    std::sort (projects.begin (), projects.end (), std::greater<String> ());

    return 0;
}
```



STL Overview: 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 & insert iterators
- Function adaptors include negators & binders
- STL adaptors can be used to *narrow* interfaces (*e.g.*, a Stack adaptor for vector)



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)

```
STL> 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
```

```
STL> 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)

```
// Meeting.h
#include <iostream>
struct Meeting {
    enum Day_Of_Week
    {MO, TU, WE, TH, FR, SA, SU};
    static Day_Of_Week
    day_of_week (char c);

Meeting (const char * title,
        Day_Of_Week day,
        unsigned int start_time,
        unsigned int finish_time);
Meeting (const Meeting & m);

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

// Meeting.h, continued ...
const char * title_;
// Title of the meeting

Day_Of_Week day_;
// Week day of meeting

unsigned int start_time_;
// Meeting start time in HHMM format

unsigned int finish_time_;
// Meeting finish time in HHMM format
};

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



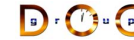
STL Example: course schedule (cont'd)

```
// Meeting.cc
#include <assert>
#include "Meeting.h"

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.cc, continued ...
Meeting::Meeting (const char * title,
                 Day_Of_Week day,
                 unsigned int start_time,
                 unsigned int finish_time)
: title_ (title), day_ (day),
  start_time_ (start_time),
  finish_time_ (finish_time)
{
}

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



STL Example: course schedule (cont'd)

```
// Meeting.cc, continued ...
Meeting & Meeting::operator =
    (const Meeting & m) {
    this->title_ = m.title_;
    this->day_ = m.day_;
    this->start_time_ = m.start_time_;
    this->finish_time_ = m.finish_time_;
    return *this;
}

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



```
// Meeting.cc, continued ...
```

```
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)

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

#include <stdlib> // main.cpp
#include <vector>
#include <assert>
#include <algo>
#include <iterator>
#include "Meeting.h"
int parse_args (int argc, char * argv[],
                std::vector<Meeting>& schedule)
{
    for (int i = 1; i < argc; i+=4) {
        schedule.push_back (Meeting
            (argv [i],
             Meeting::day_of_week (*argv [i+1]),
             static_cast<unsigned int>
             (atoi (argv [i+2])),
             static_cast<unsigned int>
             (atoi (argv [i+3]))));
    }
    return 0;
}
```

**STL Example: course schedule (cont'd)**

```
// main.cpp, continued ...

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

    if (parse_args (argc, argv,
                   schedule) < 0)
        return -1;

    std::sort (schedule.begin (),
               schedule.end ());

    if (print_schedule (schedule) < 0)
        return -1;

    return 0;
}
```

**STL Example: course schedule (cont'd)**

```
// main.cpp, continued ...
int print_schedule
(vector<Meeting> &schedule)
{
    // Find \& print out any conflicts
    for (vector<Meeting>::iterator j
         = schedule.begin ();
         j != schedule.end (); ++j)
    {
        j = adjacent_find (j,
                           schedule.end ());
        if (j == schedule.end ())
            break;

        std::cout << "CONFLICT:" << std::endl
        << " " << *j << std::endl
        << " " << *(j+1) << std::endl << std::endl;
    }
}
```



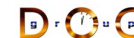
```
// main.cpp, continued ...
```

```
// Print out schedule, using
// STL output stream iterator

std::ostream_iterator<Meeting>
out_iter (std::cout, "\n");

std::copy (schedule.begin (),
           schedule.end (),
           out_iter);

return 0;
}
```



Concluding Remarks

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

References: for more information on the 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