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

The C++ Standard Template Library

- What is the STL?
- Generic programming: why use the STL?
- STL overview: helper class and 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 and function templates
  - Helper class and function templates: operators, pair
  - Container and 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 and 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 $>$ and $>=$ and $<=$ are implemented only in terms of operator $<$ on u and t (and ! on boolean results)
  - Could implement operator == as
    `!(t < u) && !(u < t)`
    so classes T and U only had to provide operator $<$ and 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 * 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::end
  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 * 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
- Iterators:
  - Input, output, forward, bidirectional, random access
  - Each container declares a trait for the type of iterator it provides
- Generic Algorithms:
  - Sequence (mutating and 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

STL Overview: sequence containers

- A **vector** can be used as an array and a stack
  - provides reallocation, indexed storage, push_back, pop_back
- A **deque** (pronounced “deck”) is a double ended queue
  - adds efficient insertion and removal at the *beginning* as well as at the end of the sequence
- A **list** has constant time insertion and deletion at *any* point in the sequence (not just at the beginning and end)
  - performance trade-off: does not offer a random access iterator

STL Overview: associative containers

- A **set** is an unordered collection of unique keys
  - *e.g.*, a set of student id numbers
- A **map** associates a value with each unique key
  - *e.g.*, a student's first name
- A **multiset** or a **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].s_ << std::endl;
    }

  return 0;
}
```

STL Overview: iterators

- Iterator *categories* depend on type parameterization rather than on inheritance: allows algorithms to operate seamlessly on both native (i.e., pointers) and user-defined iterator types
- Iterator categories are hierarchical, with more refined categories adding constraints to more general ones
  - Forward iterators are both input and 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

STL Overview: iterators (cont'd)

- Input iterators are used to read values from a sequence.
- An input iterator must allow the following operations
  - Copy ctor and assignment operator for that same iterator type
  - Operators == and != for comparison with iterators of that type
  - Operators * (can be const) and ++ (both prefix and postfix)
- Note that native types that meet the requirements (i.e., pointers) can be used as iterators of various kinds

STL Overview: iterators (cont'd)

- Output iterators differ from input operators as follows:
  - Operators = and == and != need not be defined (but could be)
  - Must support non-const operator * (*e.g.*, *iter = 3)
- Forward iterators must implement (roughly) the union of requirements for input and output iterators, plus a default ctor

STL Overview: iterators (cont'd)

- Bidirectional iterators must implement the requirements for forward iterators, plus decrement operators (prefix and postfix)
- Random access iterators must implement the requirements for bidirectional iterators, plus:
  - Arithmetic assignment operators += and -=
  - Operators + and - (must handle symmetry of arguments)
  - Ordering operators $<$ and $>$ and $<=$ and $>=$
  - Subscript operator [ ]

STL Overview: 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).s_ << std::endl;
  }

  return 0;
}
```


STL Overview: generic algorithms

- Algorithms operate over *iterators* rather than containers
- Each container declares an iterator as a trait
  - vector and deque declare random access iterators
  - list, map, set, multimap, and 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, and algorithms

STL Overview: generic algorithms (cont'd)

- Some examples of STL generic algorithms:
  - **`find()`**: returns a forward iterator positioned at the first element in the given sequence range that matches a passed value
  - **`mismatch()`**: returns a pair of iterators positioned respectively at the first elements that do not match in two given sequence ranges
  - **`copy()`**: copies elements from a sequence range into an output iterator
  - **`replace()`**: replaces all instances of a given existing value with a given new value, within a given sequence range
  - **`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 and 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 and insert iterators
- Function adaptors include negators and 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 and 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, and midnight should be read in as 0)
  - Sort the list according to day of the week and then time of day
  - Detect any times of overlap between courses and 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 and 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 and for programming model complexity (*e.g.*, requiring $<$ and $==$ and no others)

References: for more information on the STL

- David Musser's STL page
  - http://www.cs.rpi.edu/ musser/stl.html
- Stepanov and 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 and Saini, "STL Tutorial and Reference Guide"
  - ISBN 0-201-63398-1
- Austern, "Generic Programming and the STL"
  - ISBN 0-201-30956-4