Integrating the C++ Standard Template Library Into the Undergraduate Computer Science Curriculum

James P. Kelsh
James.Kelsh@cmich.edu

Roger Y. Lee
lee@cps.cmich.edu

Department of Computer Science
Central Michigan University
Mt. Pleasant, MI 48858

Introduction

When the computer science community compared Pascal and C, the C Standard Library was a reason sometimes offered for choosing C. After all (it was argued), how can you do real programming in a language whose support for character strings is as limited as Pascal's? The C++ Standard Template Library, now available with most compilers, gives us one more chance to ask how many of our habits should yield to new techniques. And many of those who seemed proud of the C Standard Library seem slow to adopt the C++ Standard Library, occasionally questioning the efficiency of a larger language and boilerplate code. But according to the goals of the C++ Standard Library, the facilities provided should be "efficient enough to provide genuine alternatives to hand-coded functions, classes, and templates." [Stroustrup, p. 430] The present authors contend that if we intend to teach problem-solving skills rather than reinventing the wheel, those of us teaching C++ should make the Standard Template Library an early part of our curriculum. This paper describes the authors' experience using the Standard Template Library in CS1 and CS2 in a department that exhibits a wide range of opinions on STL.

The Standard Template Library consists of algorithm implementations as well as "container" classes. This paper concentrates on the most generally useful container classes, specifically string, vector, list, and map.

String

Most instructors are happy with a built-in *string* type, but some feel unsure about introducing it without discussing all the details: constructors, substring search/extract functions, etc. Indeed, the STL *string* has a rich variety of functions for finding or replacing a substring, but we introduce *string* as just one more data type: Programs can read it, write it, assign it, and pass it as a parameter. Students quickly become unhappy if the language provides no way to store a name, but are happy with these four operations (read, write, assign, pass).

Some instructors feel compelled to teach all the constructors and variations, but we ask: Do you need to discuss excess-64 exponents in order to teach float or double? We do not always change their minds, but we continue offering examples of simple programs that illustrate the fundamentals of *string*:

```
// file: string.cpp
#include <string>
using namespace std;
#include <iostream>

int main()
{
  string family_name, given_name;

  cout << "Name (given name, then family name): ";
  cin >> given_name >> family_name;
  cout << family_name << ", " << given_name << endl;

  return 0;
}
```

The sample program above gives enough information for students to use *string* to add interest to their early programs. It shows the proper “include” (note that the STL tends to avoid “.h” in filenames) and the “using namespace std;” statement to make the names available. Running the program demonstrates that input to a *string* ends at whitespace. Students in CS1 need not discuss all the techniques for searching or reversing strings to write more satisfying programs.

We recommend treating it as the same sort of *string* type that we might be familiar with from some other language. Anyone who has programmed with string variables in Basic or dBase has a reasonable intuition for what can be done with a string datatype. If string manipulation is the subject being taught, or if you are tempted to assume something about the underlying representation, then consulting a good reference is advisable.

Some may advise against this simple approach: Building on the example presented above, one professor wanted to read the given name and family name into one *string.* Deciding that a *string* was an array of char, our friend wrote:

```
string str;
char ch;
int n = 0;
do
{
  cin.get(ch);
  if (ch != ‘\n’)
    str[n++] = ch;
} while (ch != ‘\n’)
```

Unfortunately, an assignment to str[0] is (generally) not allowed if the string is empty: *string* is a dynamic container, so str[0] doesn’t exist until a value has been assigned through one of the defined string operations. Extending a string is easy to with:

```
str = str + ch;
```

(Yes, there are other ways to *reserve* space before reading values, but our point is simply: A naive approach will probably work if you don't assume that you know the implementation.)

Before using STL *string*, the authors used their own string class. Certainly there is no shortage of string classes; one seems to come with every C++ text. But ours had one distinction: It was a *minimal* string class, rather than one designed to impress users with all the details it offered. By being minimal, it contained few things that could contradict the STL version. With only two modifications (adding an indexing operator and changing the read function to stop at white space) and a *typedef*, it remains useable as a substitute in cases where STL is not available. If a substitute class is used, we recommend keeping its syntax consistent with the STL string, to ease the transition to the standard library.

Vector

This is surprisingly difficult for some instructors to accept, even though the only place where programs are significantly affected by the difference between C-style arrays and vectors is the act of filling the array.

```
// C-style array:
// Keep counting

data_type ray[MAX_SIZE];
int count = 0;

while (!cin.eof() )
{

  cin >> ray[count];
  if (!cin.fail() )
    ++count;
}
```

```
// STL vector
// Just push_back

vector<data_type> ray


while (!cin.eof() )
{
  data_type temp;
  cin >> temp;
  if (!cin.fail() )
    ray.push_back (temp);
}
```

Simplifications in traversing and processing an array come from the vector's size() function:

```
// C-style array uses
// separate var: count

for (n=0; n < count; ++n)
{
  process (ray[n]);
}
```

```
// STL vector remembers
// its size

for (n=0; n < ray.size(); ++n)
{
  process (ray[n]);
}
```

But what of situations where STL vectors are not available? Or cases when pedagogical reasons suggest teaching classical programming? In these cases, we recommend borrowing from the vector approach to create a new datatype with syntax similar to that of the STL *vector*. A sample storing ints would be:

```
class int_ray
{
public:
  int_ray (int size);
  ~int_ray();
  void push_back(int x);
  int size() const;
  int& operator[](int n);

private:
  int max;
  int* ray;
  int current_size;
};

int_ray :: int_ray (int size)
{
  ray = new int [size];
  max = size;
  current_size = 0;
}

int_ray :: ~int_ray()
{
  delete[] ray;
}

int& int_ray::operator[](int n)
{
  if ( (n >= 0) && (n < max) )
    return ray[n];
  else
    // What here?
}

void int_ray::push_back(int x)
{
  if (current_size < max)
  {
    ray[current_size] = x;
    ++current_size;
  }
  else
    // What here?
}

int int_ray::size () const
{
  return current_size;
}
```

This small amount of additional typing will let students use conventional (C-style arrays) with any compiler, yet learn the syntax of STL vectors. One hybrid feature, standard in neither C-style arrays nor STL vectors, is the declaration:

```
int_ray ray(10);
```

This declares an (empty) int_ray with a maximum size of 10. After that declaration, students can program using the syntax of the STL *vector*. And we can hope it will acquaint them with some of the concerns they should consider in any use of arrays (indicated above by "`// What here?` ").

One warning is necessary: Some versions of STL, as processed by some compilers require that container classes must be comparable (with "<" and "=="). STL defines these comparisons on the containers by obtaining a "lexicographical" ordering based on the objects contained. This requires the programmer who may want to store a *student* class in a vector to define the "<" and "==" operators for *student*, even if the value returned is meaningless. The simplest way to do this is to define the operators as in-line member functions in the "header" (.h) file:

```
//  //  //  //  //  //  //  //  //  //  //  //  //  //
bool operator<  (const student& st) { return true; };
bool operator== (const student& st) { return true; };
//  //  //  //  //  //  //  //  //  //  //  //  //  //
```

List

This sparks considerable controversy. Some of our colleagues are convinced that CS2 should be a semester of "Pointers, pointers, and pointers." The thought of addressing dynamic structures without emphasizing the intricacies of *new* and *delete* seems revolutionary to them. Others see pointer manipulation as an "assembly language" concept and want students to use data structures without being intimidated by their earthy details.

The authors realize that we serve two masters (tool designers and tool users) and hope that our community will integrate these ideas in developing a consensus to present simple first or complex first. Or maybe concrete first or abstract first. Or maybe difficult first or ….

The STL *list* is declared:

```
#include <list>
typedef list<int> int_list;
```

This creates an (empty) list of ints named int_list. Our preference is to use a "typedef" to create a new name. Then this name, without extra reference to STL or "<" and ">", seems to make algorithms clearer without calling attention to the implementation of the data structures.

Acceptance of the STL *list* is mixed in our department. Accordingly, our present compromise is to introduce a relatively simple singly-linked list class whose interface is consistent with the STL list. After devoting some time to using as well as studying the implementation of the singly-linked

list, we show that it is actually a special case of a list class which can be singly-linked, doubly-linked (with separate head and tail nodes) or doubly-linked and circular, by conditional compilation pragmas. In general, the syntax is consistent with STL, to facilitate an easy transition to the STL list.

```
typedef int data_type;   // customize for your type

struct node
{
  data_type data;
  node* next;
};

class list
{
public:
  node* begin() const;
  node* end() const;
  int size() const;
  bool empty() const;
  void insert(node* p, const data_type& d);
  void erase(node*& p);

  list();
  list (const list& l);
  ~list();
list& operator= (const list& l);

private:
  void free_list();
  node* head;
#define tail NULL
};
```

Users of this list class will write code such as:

```
node* p;
for (p = lst.begin(); p != lst.end(); p = p->next)
{
  cout << p->data;
}
```

Notice that *begin()* returns a *node**. In the STL list, *my_list_type :: begin()* returns an object of type *my_list_type :: iterator*. For simplicity, we have not defined a nested class, using simply *node**. This is dangerous programming, since it allows read/write access to the list data, even if the list is passed as a *const* parameter. We live with it merely because we need a simple class for an introduction to lists. The STL list handles this danger by defining a *const_interator* type, which allows read access, but not write access. While our syntax is generally consistent with STL syntax, it is much less sophisticated.

We considered (and decided against) defining a pre- or post-increment operator for type node*. The operators on iterators defined by *list* simplify STL programming. Code to display the contents of an STL list named *lst* would look like:

```
my_list_type :: iterator p;
for (p = lst.begin(); p != lst.end(); ++p)
{
  cout << p->data;
}
```

An increment operator would have made our syntax more similar to that of the STL list, but we still favor something to remind CS2 students that there is something more than an array here. We also emphasize to the students that "p != lst.end()" is different from "p < lst.end()", a condition that might be used with a vector. In a direct comparison of pointers which may be widely scattered in memory, "<" is unreliable.

Map

Although Stroustrup, in proper object-oriented fashion, defines this solely in terms of how it functions (O(log n) access), we may be excused for thinking of it as a balanced binary search tree. Acceptance (in CS2) has been slow in our department, although some who reject it in CS2 are introducing it in an upper-division Algorithms course. *Map* may provide the clearest example of the differing goals: Teaching tools to solve problems vs. teaching complex programming (and perhaps screening out weaker students).

Map programming is deceptively simple. To declare a *map*:

```
#include <map>
using namespace std;
#include "student.h"
typedef map<string, student, less<string>  > map_type;

map_type student_map;
```

This declares an object *student_map*, of type *map_type*. The new object will store objects of type *student* in an *associative container*, with the capabilities of a binary search tree. The ordering key used is of type *string* in this example, and the order is dictated by the STL template predicate *less*. (We could provide our own function for ordering the key objects.)

Inserting an object, say *new_student*, into the map with the key value *new_key* (a *string*), is as simple as:

```
student_map[new_key] = new_student;
```

Note that no explicit action to allocate storage dynamically is apparent; All allocation and copying is done by the *map.* Although the similarity to array syntax makes some people think of O(1) access, this is actually an O(log n) operation. Similarly, finding an entry, given a key value (as in the statement "`cout << student_map["Doe, John"]);`

STL provides iterators (and const_iterators) to traverse the map. To display the entire contents of a *map*, the code would be:

```
void show_map (const map_type& st_map)
{
  map_type :: const_iterator i;

  for (i = st_map.begin(); i != st_map.end(); ++i)
    cout << "map[" << i->first << "]=" << i->second << endl;
  cout << endl;
}
```

When looking for an STL model, we were struck by the absence of an STL tree container. While the *map* provides binary search tree capability, nothing seems to provide a simple binary tree such as an expression tree, or the binary tree we might use to implement a general tree. Thus we feel compelled, even in CS2, to teach a simple binary tree which can be specialized into a binary search tree with or without balance condition. In our advanced algorithms course, we work with binary trees first and then introduce *map.* We believe that anyone who teaches balanced binary search trees should discuss *map.*

Conclusion

The Standard Template Library container classes *string*, *vector*, and *list* provide powerful tools for real world programming and can be valuable for teaching algorithms and problem-solving without becoming mired in details. We recommend that even those who prefer their own tools design them with syntax consistent with STL. Some version of the STL is currently available with most C++ compilers, although levels of compliance differ. In learning STL, we find an excellent explanation of the specification in [Stroustrup] and turn to [Nelson] for implementation details.

References

Nelson, Mark. *C++ Programmer's Guide to the Standard Template Library*. IDG, Chicago. 1995.

Stroustrup, Bjarne. *The C++ Programming Language, Third Edition.* Addison-Wesley, Menlo Park. 1997.