The STL Made Simple

Bruce Eckel

November 22, 1999

Contents

1 The Basic Concepts **2**

2 Containers of strings **5**

3 Inheriting from STL containers **8**

3.1 Directory listing 9

4 The amazing set **11**

5 The magic of maps **13**

6 STL algorithms **15**

7 Where to go from here **19**

Preface

Of course you've been hearing all about how powerful the STL is and how easy it's going to make your life. But when you go to learn about it, what do you get? Lots about iterators: forward iterators, bidirectional iterators, other types of iterators. A number of different types of containers, some of which seem to do the same thing. And a seemingly endless number of algorithms. Do you really have to know all the details and theory for this stuff in order to get some good from the STL?

Emphatically no. The premise of this paper is that there's a relatively small subset of elements and ideas (let's say 10%) you need to understand in order to get 90% of the usefulness from the STL. The "simple 10%" is stuff you can use on a day-to-day basis without looking anything up. Understanding this portion will also get you comfortable with the basic ideas in the STL so that you can go look up the more difficult stuff if you ever need it.

After learning about the basic concepts, you'll see a number of increasingly powerful examples. By the end you should have a good grasp of both the simplicity and power of the STL portion of the Standard C++ Library.

1 The Basic Concepts

The primary idea in the STL is the *container* (also known as a *collection*), a fundamental concept in object-oriented programming. A container is just that: a place to hold things. You need containers because objects are constantly marching in and out of your program and there must be someplace to put them while they're around. You can't make named local objects because in a typical program you don't know how many, or what type, orthe lifetime of the objects you're working with. So you actually need a container that will expand whenever necessary to fill your needs.

All the containers in the STL hold objects and expand themselves. In addition, they hold your objects in a particular sequence. The difference between one container and another is the way the objects are held and how the sequence is created. Let's start by looking at the simplest containers.

A **vector** is a linear sequence that allows you to rapidly move around in a random fashion, but it's expensive to insertan element in the middle of the sequence. A **list** is also a linear sequence, but it's expensive to move around randomlyand cheap to insert an element in the middle. Thus **list** and **vector** are virtually identical in their basic functionality,but different in the cost of their activities. So for your firstshot at a program, you could use either one, and only experiment with the other if you're tuning for efficiency.

Most of the problems you set out to solve will only require a simple linear sequence like a **vector** or **list**. Both of these (indeed, most of the STL containers) have member functions **push_front( )** and **push_back( )** which youuse to insert a new element into the front or the back of thesequence.

But now how do you retrieve those elements? With a **vector**, it is possible to use the indexing **operator**[], but that doesn't work with **list**. Since it would be nicest to learna single interface, we'll use the one defined for all STL containers: the *iterator*.

An iterator is a class that abstracts the process of moving through a sequence. It allows you to select each element of a sequence *without knowing the underlying structure of that sequence.* This is a powerful feature, partly because it allows us to learn a single interface that works with all containers, and partly because it allows containers to be used interchangeably.

One more observation and you're ready for an example. Even though the STL containers hold objects by value (that is, they hold thewhole object inside themselves) that's probably not the way you'llgenerally use them time if you're doing object-oriented programming. That's because in OOP, most of the time you're creating objectson the heap with **new** and *upcasting* the address to the base-class type, manipulating it as a pointer to the baseclass. The beauty of this is that you don't worry about the specific type of object you're dealing with, which greatly reduces the complexity of your code and increases the maintainability of your program. This process of upcasting is what you try to do in OOP, so you'll usually be using containers of pointers.

Consider the classic "shape" example where shapes have a set of common operations, and you have different types of shapes. Here's what it looks like using the STL **vector** to hold pointers to various types of **shape** created on the heap:

```
//: STLSHAPE.CPP -- simple shapes w/ STL
#include <vector.h>
#include <iostream.h>
using namespace std;
class shape {
  public:
    virtual void draw() = 0;
    virtual ~shape() {};
```

```
};

class circle : public shape {
  public:
    void draw() { cout << "circle::draw\n"; }
    ~circle() { cout << "~circle\n"; }
};

class triangle : public shape
{
  public:
    void draw() { cout << "triangle::draw\n"; }
    ~triangle() { cout << "~triangle\n"; }
};

class square : public shape
{
  public:
    void draw() { cout << "square::draw\n"; }
    ~square() { cout << "~square\n"; }
};

typedef vector<shape*> container;
typedef container::iterator iter;

int main()
{
  container shapes;
  shapes.push_back(new circle);
  shapes.push_back(new square);
  shapes.push_back(new triangle);
  for ( iter i=shapes.begin(); i != shapes.end(); i++)
      (*i)->draw();  // ... sometime later:
  for ( iter j=shapes.begin(); j != shapes.end(); j++)
      delete *j;
  return 0;
}
```

You can see here the very important line:

```
using namespace std;
```

This is essential because the Standard C++ libraries are placedin their own *namespace*, a relatively new feature in C++ that allows you to keep all the names in your program from colliding. The Standard C++ libraries are in a namespace called **std**, and without the **using** directive shown above, the library names wouldn't be visible in the program and you'll get all sorts of unseemly error messages at compile time.

The creation of **shape**, **circle**, **square** and **triangle** should be fairly familiar. **shape** is a pure abstract base class (because of the *pure specifier* **=0**) that defines the interface for all types of **shapes**; thederived classes redefine the **virtual** function **draw( )** to perform the appropriate operation. Now

we'd like to create a bunch of different types of **shape** object, but where to put them? In an STL container, of course. For convenience, the **typedef**

```
typedef vector<shape*> container;
```

creates an alias for a **vector** of **shape***, and the**typedef**

```
typedef container::iterator iter;
```

uses that alias to create another one, for **vector<shape*>::iterator**. Notice that the container name must be used to produce the appropriate iterator, which is defined as a nested class. Although there aredifferent types of iterators (forward, bidirectional, reverse, etc.) all the ones we're interested in here have the same basic interface: you can increment them with **++**, you can dereference them to produce the object they're currently selecting, and you can test them to see if they're at the end of the sequence. That's what you'll want to do 90% of the time. And that's what is done in the above example: after creating a container, it's filled with different types of **shape***. Notice that the upcast happens as the **circle**, **square** or **rectangle** pointer is added to the **shapes** container, which doesn't know about those specific types but instead holds only **shape***. So as soon as the pointer is added to the container it loses its specific identity and becomes an anonymous **shape***. This is exactly what we want: toss them all in and let polymorphism sort it out.

The first **for** loop creates an iterator and sets it to the beginning of the sequence by calling the **begin( )** member function for the container. All containers have **begin( )** and **end( )** member functions that produce an iteratorselecting, respectively, the beginning of the sequence and onepast the end of the sequence. To test to see if you're done, you make sure you're **!=** to the iterator produced by **end( )**. Not < or <=. The only test that works is **!=**. So it's very common to write a loop like:

```
for(iter i = shapes.begin(); i != shapes.end(); i++)
```

This says: "take me through every element in the sequence."

What do you do with the iterator to produce the element it's selecting? You dereference it using (what else) the '*****' (which is actually an overloaded operator). What you get back is whateverthe container is holding. This container holds **shape***,so that's what ***i** produces. If you want to send a message to the **shape**, you must select that message with **->**, so you write the line:

```
(*i)->draw();
```

This calls the **draw( )** function for the **shape*** the iterator is currently selecting. The parentheses are ugly but necessary to produce the proper order of evaluation.

Finally, the STL containers don't call **delete** for the pointersthey contain. There's no elegant way for you to do it, either,unless you want to inherit from the container class in question and add a member function to perform the **delete** for all the contained pointers. One of the problems is that if you createan object on the heap with **new** and place it's pointer ina container, you can't tell if that pointer is also placed inside another container. So the STL just doesn't do anything about it,and puts the responsibility squarely in your lap. The last linesin the program move through and delete every object in the container so proper cleanup occurs.

It's very interesting to note that you can change the type of container that this program uses with two lines. Instead of including **vector.h**, you include **list.h**, and in the first **typedef** you say:

```
typedef list<shape*> container;
```

instead of using a **vector**. Everything else goes untouched. This is possible not because of an interface enforced by inheritance (there isn't any inheritance in the STL, which comes as a surprise when you first see it), but because the interface is enforcedby a convention adopted by the designers of STL, precisely so you could perform this kind of interchange. Now I can easily switch between **vector** and **list** and see which one works fastest for my needs.

So this is what you're going to be doing most of the time: creating a container of pointers to the base type, upcasting heap-based objects into that container, and using an iterator to traverse the sequence and perform operations on it. Sure, there are lots of other things you may want to do and you might use an STL algorithm to perform a particular operation (**sort( )** is a common one) but most of the time your program will probably end up with parts of it looking something like the STL SHAPE program above. That's the basic idea, and now we can take that concept and start applying it to the solution of different problems.

2 Containers of strings

One of the biggest time-wasters in C is character arrays: keeping track of the difference between static quoted strings and arrays created on the stack and the heap, and the fact that sometimes you're passing around a **char*** and sometimes you must copy the whole array (in C++ we sometimes refer to this as the general problem of *shallow copy* vs. *deep copy*). Especially because string manipulation is so common, character arrays area great source of misunderstandings and bugs.

Despite this, creating string classes remained a common exercise for beginning C++ programmers for many years. At last, the Standard C++ library has adopted a **string** class that solves thisproblem: **string** objects do all the work for you, in just the way you'd expect, including keeping track of memory even during assignments and copy-constructions. You simply don't need to think about it.

One of the places where this is particularly useful is pointed out in the prior example. At the end of **main( )**, itwas necessary to move through the whole list and **delete** all the **shape** pointers.

```
for (iter j = shapes.begin(); j != shapes.end(); j++)
    delete *j;
```

This highlights what could be seen as a flaw in the STL: there's no facility in any of the STL containers to automatically **delete** pointers they contain, so you must do it by hand. It's as if the assumption of the STL designers was that containers of pointersweren't an interesting problem, although I assert that it is oneof the more common things you'll want to do.

Automatically deleting a pointer turns out to be a rather aggressive thing to do because of the *multiple membership* problem. If a container holds a pointer to an object, it's not unlikelythat pointer could also be in another container. A pointer toan **aluminum** object in a list of **trash** pointers couldalso reside in a list of **aluminum** pointers. Then which list is responsible for cleaning up that object - which list "owns" the object?

This question is virtually eliminated if the object rather than a pointer resides in the list. Then it seems clear that when the list is destroyed, the objects it contains must also be destroyed. Here, the STL shines, as you can see when creating a containerof **string** objects:

```
//: STRLIST.CPP -- A list of strings

#include <string>
#include <vector>
#include <fstream>
#include <strstream>
#include <assert.h>

using namespace std;
int main(int argc, char* argv[])
{
  assert(argc == 2);
  ifstream in(argv[1]);
  assert(in);

  vector<string> strings;
  vector<string>::iterator w;
  const int sz = 255;
  char buf[sz];
  while ( in.getline(buf, sz) )
    strings.push_back(buf);

  // Do something to the strings...

  int i = 1;
  for (w = strings.begin(); w != strings.end(); w++)
  {
    ostrstream os(buf, sz);
    os << i++ << ends;
    *w = string(buf) + string(": ") + *w;
  }

  // Now send them out:
  for ( w = strings.begin(); w != strings.end(); w++)
    cout << *w << endl;
  // string objects clean themselves up,
  // since they aren't pointers!
  return 0;
}
```

Note the use of the new C++ include syntax:

```
#include <string>
```

When there's no extension, it indicates a C++ file; the **.h** extensions indicate C headers. Your programming system is supposed to translate from the extensionless header name to the appropriate name on the local system. Since most programming systems don'thave any kind of direct support for this (Borland C++ 5 *does* support it automatically, so you can use the extension ornot, as you choose) it's easy to use this syntax anyway: simplymake a copy of the file with the extension to the extensionlessversion. Of course, you can continue to use the old style of include if you wish. Notice that

```
#include <strstream>
```

gives a name that is nine characters long, but if you move to the INCLUDE directory you'll see that the file name is still STRSTREAM.H, so the system is performing the translation for you.

Once **strings** is created, each line in the file is read into **buf** and then simultaneously turned into a **string** and put in the **vector**:

```
while ( in.getline(buf, sz) )
  strings.push_back(buf);
```

Since **strings** is a **vector**<**string**>, **push_back( )**is expecting a **string** argument. It is handed a **char***, so the compiler looks for a way to automatically convert it intoa **string** object. The **string(char*)** constructor producesthe necessary automatic type conversion.

The operation that's being performed on this file is to add linenumbers. The easiest way to turn numbers into strings is to putthem into a stream, so an **ostrstream** (which formats toa block of memory) is used, with the constructor that accepts the memory you want to use and the size of that memory. Then you can write anything you want into that memory and it ends up formatted there (but remember to add a null terminator to the end of thestring with **ends**!). You can even get fancier formatting, such as justification within a field.

```
ostrstream os(buf, sz); os << i++ << ends;
```

Assembling **string** objects is quite easy, since **operator+** is overloaded. Amazingly enough, the iterator **w** can be dereferenced to produce a string that can be used as both an rvalue *and* an lvalue:

```
*w = string(buf) + string(": ") + *w;
```

The fact that you can assign back into the container via the iterator may seem a bit surprising at first, but it's a tribute to thecareful design of the STL.

The **operator**<< is overloaded for **ostream** and **string**, so you can just dereference the iterator to produce the **string** and send the result to **cout**.

Because the **vector**<**string**> contains the objects themselves, a number of interesting things take place. First, no cleanup is necessary. Even if you were to put addresses of the **string** objects as pointers into *other* containers, it's clear that **strings** is the "master list" and maintains ownership of the objects.

Second, you are effectively using dynamic object creation, and yet you never use **new** or **delete**! That's because,somehow, it's all taken care of for you by the **vector** (this is non-trivial. You can try to figure it out by looking at the header files for the STL - all the code is there - but it's quite an exercise). Thus your coding is significantly cleaned up.

The limitation of holding objects instead of pointers inside containers is quite severe: you can't upcast from derived types, thus you can't use polymorphism. The problem with upcasting objects byvalue is that they get sliced and converted until their type is completely changed into the base type, and there's no remnant of the derived type left. It's pretty safe to say that you *never* want to do this.

3 Inheriting from STL containers

The power of instantly creating a sequence of elements is amazing,and it makes you realize how much time you've spent (or rather, wasted) in the past solving this particular problem. For example,many utility programs involve reading a file into memory, modifying the file and writing it back out to disk. One might as well take the functionality in STRLIST.CPP and package it into a class for later reuse.

Now the question is: do you create a member object of type **vector**, or do you inherit? A general guideline is to always prefer composition (member objects) over inheritance, but with the STL this is often not true, because there are so many existing algorithms that work with the STL types that you may want your new type to *be* an STL type. So the list of **string**s should also *be* a **vector**, thus inheritance is desired.

```
//: STREDIT.H -- File editor tool
#include <string>
#include <vector>
#include<iostream>

using namespace std;
class streditor : public vector<string>
{
  public:
    streditor(char* filename);
    void write(ostream& out = cout);
};
```

The constructor opens the file and reads it into the **streditor**,and **write( )** puts the **vector** of **string** onto any **ostream**. Notice in **write( )** that you can have a default argument for a reference.

The implementation is quite simple:

```
//: STREDIT.CPP -- File editor tool
#include "stredit.h"
#include <fstream>
#include <assert.h>

streditor::streditor(char* filename)
{
  ifstream in(filename);
  assert(in);
  const int sz = 255;
  char buf[sz];
  while ( in.getline(buf, sz) )
    push_back(buf);
}

void streditor::write(ostream& out)
{
  for ( iterator w=begin(); w!=end(); w++)
    out << *w << endl;
}
```

The functions from STRLIST.CPP are simply repackaged. Often this is the way classes evolve - you start by creating a program tosolve a particular application, then discover some commonly-used functionality within the program that can be turned into a class.

The line numbering program can now be rewritten using **streditor**:

```
//: STREDTST.CPP -- Test the file edit tool
#include "stredit.h"
#include <strstream>

using namespace std;

int main(int, char* argv[])
{
  streditor
  File(argv[1]);  // Do something to the strings...
  const int sz = 255;
  char buf[sz];
  int i = 1;
  streditor::iterator w = File.begin();
  while ( w != File.end() )
  {
    ostrstream os(buf, sz);
    os << i++ << ends;
    *w = string(buf) + string(": ") + *w;
    w++;
  }  // Now send them to cout:
  File.write(cout);
  return 0;
}
```

Now the operation of reading the file is in the constructor:

```
streditor File(argv[1]);
```

and writing happens in the single line:

```
File.write(cout);
```

The bulk of the program is involved with actually modifying the file in memory.

3.1 Directory listing

As another example, consider the creation of a directory listing: you need a place to keep all the file names in a directory. Both a **list** and a **vector** seem to solve the problem, but you probably won't need to insert elements in the middle so we'll start with a **vector** (later, sorting may work better with a **list**, but the STL is designed to accommodate easy changes).

This tool creates a **vector**<**string**>, thus it's holdingt he **string** objects themselves and not pointers, so you don't have to worry about who's responsible for cleaning up. The **vector** "owns" the **string** objects.

```
//: FILELIST.H -- general file lister

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

using namespacestd;

class filelist : public vector<string>
{
  public:
    filelist (const char* afn ="*.*");
    void listall (ostream& os = cout,
                  const char* sep = "\n");
};
```

Only two member functions are necessary to implement the class. The constructor performs the filename expansion, filling up the **vector**, and the **listall( )** function prints the information to an **ostream** reference (defaulting to **cout**), delimited with your choice of separator (defaulting to newline). But is this meager **listall( )** function the limit of what you can do with a **filelist**? No! Because it's inherited from **vector**, you can do anything to it that you can to a **vector**, including creating an iterator and traversing the list for whatever your purpose. You can also usethe **vector**'s indexing **operator**[]. You can see all three operations in this example:

```
//: FLTEST.CPP -- test file lister
#include "filelist.h"

main()
{
  filelist files;
  files.listall();
  for (int i = 0; i < files.size(); i++)
    cout << "(" << files[i] << ")" << endl;
  copy (files.begin(), files.end(),
        ostream_iterator<string>(cout, "\n"));
}
```

First, **listall( )** is called, allowing it to use its default arguments. Second, a **for** loop counts from 0 to **size( )** (a **vector** member function) and the **vector** indexing **operator**[ ] is used to produce each **string** object. Finally, in the call to **copy( )** (an STL algorithm that's part of the Standard C++ Library) you can see the iterators produced by **begin( )** and **end( )** passed in as arguments; the **begin( )** iterator is used to move through the sequence until **end( )**, as seen in the earlier example.

The third iterator (that's what it is) in the call to **copy( )** requires some explanation. **copy( )** and many of theother STL algorithms just want to talk to iterators. In the case of **copy( )**, the three arguments are all iterators, specifying "where do I start", "where do I finish", and "where am I copying to"? An **ostream** is a bit like a container (you put stuff into it, right?) so someone decided it would be convenient to create an object that acts like an iterator in order to conveniently talk to that container: the **ostream_iterator**. This is a template; the template argument is the type you're writing to the **ostream** (and this typemust have an overloaded **operator**<< for **ostream**s), and the

constructor arguments are the specific **ostream** you're writing to and the delimiter to insert between each write. Thus,

```
copy (files.begin(), files.end(),
      ostream_iterator<string>(cout, "\n"));
```

means: "start at the beginning of **files** and go tothe end, and copy each **string** element to **cout**, separating each with a newline." Of course, you could write the code yourself as was done with the **for** loop, but the **copy( )** form is succinct and often used. It's also quite easy to changeyour mind and decide to move the information to a completely different container, since the only access is through an iterator (which all containers can produce).

The constructor uses functions found in the Borland-specific **direct.h** file, but this is not important, since the information is hidden in the implementation, and can thus be changed to suit the operating system and library:

```
//: FILELIST.CPP -- Member function definitions
#include "filelist.h"
#include <direct.h>  // DOS directory

functionsfilelist::filelist(const char* afn )
{
  ffblk fileinfo;
  int done = findfirst(afn, &fileinfo, 0);
  while (!done)     // automatic type conversion:
  {
    push_back(fileinfo.ff_name);
    done = findnext(&fileinfo);
  }
}

void filelist::listall(ostream& os, const char* sep)
{
  for (iterator i = begin(); i != end(); i++)
    os << (*i).c_str() <<  sep;
}
```

Something that may strike you as curious at first is the **for** loop in **listall**. Why can you simply say "**iterator**", "**begin( )**" and "**end( )**" without referring to the class or object? Because **filelist** is *inherited* from **vector**<**string**>, all those member functions and the nested class are *part of* **filelist**, so you can just refer to them.

Now that you have the tool, creating a list of files within a program becomes effortless. But even more amazing is how easy it was to create the tool itself - because most of the work has been done in the STL.

4 The amazing set

The **set** produces a container that will accept only oneof each thing you place in it; it also sorts the elements. To enable this you must tell the set how to sort the elements using a second template argument. For a set of **int**, you say:

```
set <int, less<int>> intset;
```

The **less**<**int**> is what establishes the sort order, which is ascending. Only in rare cases will you use anything except **less**<**type**> as the second argument. This argumentallows you to create special sort orders by defining a different class, which is beyond the scope of this paper.

To see how a **set** works, consider a set of **int**:

```
//: INTSET.CPP -- Simple use of STL set
#include <set.h>

using namespace std;
void main()
{
  set <int, less<int>> intset;
  for(int i = 0; i < 25; i++)
    for(int j = 0; j < 10; j++)  // Try to insert multiple copies:
      intset.insert(j);          // Print to output:
  copy(intset.begin(),
  intset.end(), ostream_iterator<int>(cout, "\n"));
}
```

The **insert( )** member does all the work: it tries putting the new element in, rejects it if it's already there, and keeps the list sorted. Very often the activities involved in using a set are simply insertion and a test to see whether it containsthe element. You can also form a union, intersection, or difference of sets, and test to see if one set is a subset of another.

Consider the problem of creating an index for a book. You might like to start with all the words in the book, but you only want one each and you want them sorted. Of course, a **set** isperfect for this, and solves the problem effortlessly:

```
//: WORDLIST.CPP -- Find unique words
#include <string>
#include <set>
#include <fstream>
#include <assert.h>

using namespace std;

const char* delimiters =
            " \t;()\"<>:{}[]+-=&*#.,/\\" "0123456789";

main (int argc, char* argv[])
{
  assert(argc == 2);
  ifstream in(argv[1]);
  assert(in);

  set<string, less<string>> concordance;
  const sz = 1024; char buf[sz];

  while (in.getline(buf,sz))  // Capture individual words:
  {
```

```
    char* s = strtok(buf, delimiters);
    while (s)
    {
      concordance.insert(s); // Auto
      type conv.s = strtok(0, delimiters);
    }
  } // output results:
  copy(concordance.begin(), concordance.end(),
       ostream_iterator<string>(cout, "\n"));
}
```

This is just an extension of the previous example, but it opens a file, reads each line and then breaks it up into words using the Standard C library function **strtok( )**. Other than that the **set** does all the work. Consider how much effort it would be to accomplish the same task in C, or even in C++ without the STL.

You don't have to use a **set** just to get a sorted sequence. You can use the **sort( )** function (along with a multitude of other functions in the STL) on a **vector** or **list**.

5 The magic of maps

A **map** is an *associative array*, which means you associate one object with another in an array-like fashion, but instead of selecting an array element with a number as you do with anordinary array, you look it up with an object! The example which follows counts the words in a text file, so the index is the **string** object representing the word, and the value being looked up is the object that keeps count of the strings.

In a single-item container like a **vector** or **list**,there's only one thing being held. But in a **map**, you've got two things: the *key* (what you look up by, as in **mapname**[**key**]) and the *value* that results from the lookup with the key.This is fine as long as you're using an array-style lookup, butwhat if you simply want to move through the entire map and list each key-value pair? Of course you use an iterator, like everything else in the STL, but since there are two items - the key and the value - which one should the iterator produce? Dereferencing a **map** iterator produces *both* items, packaged together into a single object (since a function can only return a singlevalue) called a **pair**, a template whose sole reason for existence is to package two objects. The definition for **pair** is remarkably simple:

```
template <class T1, class T2> struct pair
{
  T1 first;
  T2 second;
  pair (const T1& a, const T2& b) : first(a), second(b) {}
};
```

You can see that it does nothing else but the packaging operation. You access the members of a **pair** by selecting **first** or **second**.

This same philosophy of packaging two items together is also used to insert elements into the map, but the **pair** is created as part of the instantiated **map** and is called **value_type**, containing the key and the value. So one option for inserting a new element is to create a **value_type** object, loading it with the appropriate objects and then calling the **insert( )** member function for the **map**.

However, the following example will make use of a special feature of **map**: if you're trying to find an object by passing in a key to **operator**[] and that object doesn't exist, **operator**[] will automatically insert a new key-value pair for you, using the default constructor for the value object. With that in mind, consider an implementation of a word counting program:

```
//: WCOUNT.CPP -- Word Count with map class
#include <string>
#include <map>
#include <fstream>
#include <assert.h>

using namespace std;

class Count
{
    int i;
  public:
    Count() : i(0) {}
    void operator++(int) { i++; } // post-increment
    int val() { return i; }

};

typedef map<string, Count, less<string> > wordmap;

const char* delimiters =  "\t.,:;\"{}-+&^%$#@!~‘?/’\\|()[]<>*=";

int main(int argc, char* argv[])
{
  assert(argc == 2);
  ifstream in(argv[1]);
  assert(in);

  wordmap words;
  const int sz = 255;
  char buf[sz];
  while(in.getline(buf, sz))
  {
    char* word = strtok(buf, delimiters);
    while(word)
    {
      words[string(word)]++;
      word = strtok(0, delimiters);
    }
  }
  for (wordmap::iterator w = words.begin();
       w != words.end(); w++)
    cout << (*w).first << ": " << (*w).second.val() << endl;
  return 0;
}
```

The need for the **Count** class is to contain an **int** that's automatically initialized to zero. This is

necessary because of the crucial line:

```
words[string(word)]++;
```

This finds the word that has been tokenized by **strtok( )** (as mentioned earlier in this paper) and increments the **Count** object associated with that word, which is fine as long as there *is* a key-value pair for that string. If there isn't, the **map** automatically inserts a key for the word you're looking up, and a default **Count** object, which is initialized to zero by the constructor. Thus, when it's incremented the **Count** becomes 1.

Printing the entire list requires traversing it with an iterator (there's no **copy( )** shortcut for a **map**). Aspreviously mentioned, dereferencing this iterator produces a **pair** object, with the **first** member the key and the **second** member the value. In this case **second** is a **Count** object, so it's **val( )** member must be called to produce the actual word count.

If you want to find the count for a particular word, you can use the array index operator, like this:

```
cout << "the: " << words["the"]  << endl;
```

A more complicated version of **map** is the **multimap** which allows you to associate more than one object with a key. A perfect example is a thesaurus, where you have a word and you want to know all the words that are similar. When you look up a word, then, you'll get back an iterator to a list of words. There's an example of a **multimap** in the Rogue Wave HelpFiles, mentioned later.

6 STL algorithms

The other half of the STL is the algorithms, which are templatized functions designed to work with the containers. You've seen one example already in the **copy( )** algorithm used in WORDLIST.CPP:

```
copy ( concordance.begin(),
       concordance.end(),
       ostream_iterator<string>(cout, "\n") );
```

The original design intent of the STL was around the algorithms. The goal was that you use algorithms for almost every piece ofcode that you write. In this sense it was a bit of an experiment, and we won't know for a while how well it works. The reason for this is that all the syntax required to support the STL algorithmsisn't in place yet.

As an example, consider the **for_each( )** algorithm. You hand this two iterators for the starting and ending points, and a pointer to a function that takes an argument of the same type that your iterators produce. **for_each( )** will sweep from the beginning to the end, pull out each element and pass it as an argument while it dereferences your function pointer. So **for_each( )** actually performs the operations that have been explicitly written out in most of the examples in this paper. In STLSHAPE.CPP, for example:

```
for ( iter j=shapes.begin(); j!=shapes.end(); j++)
  delete *j;
```

You can see this clearly if you look at the template describing **for_each( )**:

```
template <class InputIterator, class  Function> Function
  for_each ( InputIterator first,
             InputIterator last,
             Function f)
  {
     while (first != last)
       f(*first++);
     return f;
  }
```

The first impression of this seems fairly simple: **Function** must be a pointer to a function which takes, as an argument, anobject of whatever **InputIterator** selects. However, the following example shows that there are actually a number of different ways this template can be expanded:

```
//: Algo1.cpp -- Use of STL for_each algorithm
#include <iostream>
#include <vector>
#include <algo>
#include <defalloc.h>

#ifdef __BORLANDC__
using namespace std;
#endif

class foo
{
  static int count;
  char * id;
public:
  foo (char * ID) : id(ID) { count++; };
  ~foo() {
     cout << id << " count = " << --count << endl;
   }
};

int foo::count = 0;

class fooVector : public vector<foo*>
{
  public:
  fooVector(char* ID) {
    for(int i = 0; i < 5; i++)
      push_back(new foo(ID));
  }
};

// (1) Simple function
void Destroy(foo* fp) { delete fp; }

// (2) Template class w/operator()()
template <class T> class Destroy
{
```

```
public:
  void operator()(T x) {
    delete x;
  }
};

// (3) Template function
template <class T>void wipe(T* x)
{
    delete x;
};

// (4) Defalloc.h deallocate() template function
// Borland memory.h has a deallocate template
// that supercedes the one in defalloc.h. This
// is a problem for their STL implementation.
// The following is the same as in defalloc.h,
// with a distinct name. It doesn't get the
// desired results because
// operator delete(buffer) doesn't call
// the destructor.
//
template <class T> inline void my_dealloc(T* buffer)
{
  ::operator delete(buffer);
  // delete buffer;
  // this works correctly
}

// (5) Why not have a generic tool to handle
// this problem?

template <class InputIterator>
void delete_all (InputIterator first,
                 InputIterator last)
{
  while (first != last)
  {
    delete *first;
    first++;
  }
};

void main()
{
  fooVector A("one");
  for_each(A.begin(), A.end(), Destroy);

#ifdef  _MSC_VER // Won't compile with BC5.01
  fooVector B("two");
  for_each(B.begin(), B.end(), Destroy<foo*>());
#endif
```

```
#ifdef __BORLANDC__
  // Won't compile with VC++ 4.2
  fooVector C("three");
  for_each(C.begin(), C.end(), wipe);
  fooVector D("four");
  // Correct  behavior, but doesn't produce
  // desired results
  for_each(D.begin(), D.end(), my_dealloc<foo*>);
  // Also compiles correctly:
  fooVector D2("four and 1/2");
  for_each(D2.begin(), D2.end(), my_dealloc);
#endif

  fooVector E("five");
  delete_all(E.begin(), E.end());
  // Maybe this isn't so bad after all:
  fooVector F("six");
  for ( fooVector::iterator i = F.begin();
        i != F.end(); i++)
  {
    delete *i;
  }
}
```

First of all, you'll notice that we've moved into no-compiler's land, as neither of the two prominent compilers will accept this entire program. But it seems reasonably safe to assume that someday all compilers will accept it.

The **class foo** keeps a static count of how many **foo** objects have been created, and tells you as they are destroyed. In addition, each foo keeps a **char*** identifier to make tracking the output easier.

The **fooVector** is inherited from instantiated **vector**<**foo***>, and in the constructor it creates some **foo** objects, handing each one your desired **char***. The **fooVector** makes testing quite simple, as you'll see.

The commented numbers next to the approaches for destruction correspond to the strings used to create the corresponding **fooVector** in **main( )**. Approach one is the simple pointer-to-function, which works but has the drawback that you must write a new **Destroy** function for each different type. The obvious solution is to make a template, but approach two shows that a template with an overloaded **operator( )( )** will also work, although Borland 5.01 won't compile it.

On the other hand, approach three also makes sense: why not use a template function? Borland handles this fine, but Microsoft 4.2 won't compile it.

The real question is this: since this is obviously something you might want to do a lot, there's probably already a way to do itin the STL, right? Well, the most likely candidate is the **deallocate( )** template that's in **defalloc.h**. However, Borland has a **deallocate( )** template in **memory.h** that clashes with this, and that's a problem for the STL. But to try it out anyway, I've copied the proper **deallocate( )** from **defalloc.h** and given it a unique name for approach four: **my_dealloc( )**. This compiles fine, but it doesn't produce the desired results because the code **::operator delete(buffer)**; explicitly says "just release the memory, don't call the destructor" which isn't what we want. So the approach fails and you cannot just write a simple line of code using the pre-existing STL code.

Since that failed, it seems like there *should* be an algorithm to **delete** all the pointers in a container, so why not make one. Approach five does this using **for_each( )** as a starting point. This is nice and since you have the source code for the STL in the header files, you could make your own extension by inserting this into **algo.h**. But it's too bad its not part of the standard.

After looking at all these issues, approach six, which is the one used throughout the rest of the paper, doesn't look so bad anymore. It's a few more characters to type but it's easy, straightforward, and you know what it will do. On the other hand, **for_each( )** is a relatively simple algorithm and if you are trying to do something more complicated it may be well worth your while to look through the STL algorithms to see if part or all of your problem is already solved.

7 Where to go from here

Much of the time you will find yourself making relatively simple use of the STL. Either you'll just create containers of objects (as shown earlier, **string** is probably the most popular candidate), or you'll create containers of pointers to base classes to support polymorphic calls on groups of objects. The convenience, efficiency and reliability of the STL for these activities will certainly improve your programming productivity.

However, the STL has powerful implications as a tool with whichto create other tools, as was briefly shown in the STREDIT and FILELIST tools. It's as if the STL has turned C++ into a "Very High Level Language" by moving you away from the low level details. As a result, people are beginning to create some verypotent tools.

When you step into this realm you must begin to understand much more of the underlying structure of the STL; the learning curve from relatively simple usage to creating sophisticated tools is rather steep and shouldn't be taken lightly.

The easiest place to start when looking for more complexity is in the help file and examples that come with the Standard C++ Library from Rogue Wave (packaged as part of Borland C++ 5). Here you'll find descriptions and examples of the rest of the STL containers as well as all the algorithms. Many of the examples are useful and nontrivial, such as an inventory system, radix sort, spellingchecker, telephone database, graphs, a concordance, an RPN calculator, bank teller simulation, polynomial root finder, and more.

Other good resources are the *C++ Programmer's Guide to the Standard Template Library* by Mark Nelson (IDG Books 1995,ISBN 1-56884-314-3) and *The STL Primer* by Graham Glassand Brett Schuchert (Prentice-Hall 1996, ISBN 0-13-454976-7).

In the past 10 years Bruce Eckel has published over 100 articles and given talks and seminars to thousands throughout the world. For more information on consulting, training and public seminars, visit MindView[1], or email `Bruce@EckelObjects.com`.

[1]See URL http://www.MindView.net