A modest STL tutorial

By Jak Kirman

I am using a software tool called hyperlatex to create this document. The tutorial is also available in compressed postscript form. For those wanting to install a copy of these pages on their system, they may be freely copied providing they are not modified in any significant way (other than, say, locale changes). The file tutorial.tar.Z contains a tarred distribution of the pages. Please note that I will be making modifications to this document in the coming months, so you may want to occasionally check for changes. I will start putting in version numbers, and if I can manage to, changebars.

Disclaimer

I started looking at STL a little over a year ago; for a long time I could compile only minimal subsets of the HP version of the library. More recently, the compilers used by the students I teach have been able to support more of STL, and I have been adding it to the introductory and advanced courses I teach.

I have been using C++ for six or seven years, and teaching C++ and object-oriented design courses in industry for five. I really like the design philosophies in STL; I think that you can learn a great deal about how generalization can simplify programming by understanding why STL is constructed the way it is.

At the moment, books are starting to appear on STL; I haven't yet bought any of them, but I have looked at some. I would be very interested to hear peoples' opinions of any books written on STL.

I haven't seen very much online documentation on STL, apart from the good but rather dense paper by Stepanov and Lee, I thought I would try to write something to give people a taste of what a good library will be do for them.

Another reason for getting people to start trying out STL soon is to put pressure on the compiler-writers to get their compilers patched up enough to take the strain it puts on them...

This is a fairly recent project (started April 1995) to which I can't devote too much time, so it is still very short, and pretty poorly organized.

I would greatly appreciate comments or suggestions from anyone.

Outline

STL contains five kinds of components: containers, iterators, algorithms, function objects and allocators.

In the section Example I present a simple example, introducing each of the five categories of STL components one at a time.

In the section Philosophy I explain the rationale behind the organization of STL, and give some hints on the best ways to use it. (Not yet written)

The Components section goes into each category of component in more detail.

The section Extending STL shows how to define your own types to satisfy the STL requirements. (Not yet written)

A first example

Most of you probably use some kind of auto-allocating array-like type. STL has one called `vector`. To illustrate how vector works, we will start with a simple C++ program that reads integers, sorts them, and prints them out. I will gradually replace bits of this program with STL calls.

Version 1: Standard C++

Here is a standard C++ program to read a list of integers, sort them and print them:

```
#include <stdlib.h>
#include <iostream.h>

// a and b point to integers.  cmp returns -1 if a is less than b,
// 0 if they are equal, and 1 if a is greater than b.
inline int cmp (const void *a, const void *b)
{
  int aa = *(int *)a;
  int bb = *(int *)b;
  return (aa < bb) ? -1 : (aa > bb) ? 1 : 0;
}

// Read a list of integers from stdin
// Sort (c library qsort)
// Print the list

main (int argc, char *argv[])
{
  const int size = 1000;  // array of 1000 integers
  int array [size];
  int n = 0;
  // read an integer into the n+1 th element of array
  while (cin >> array[n++]);
  n--; // it got incremented once too many times

  qsort (array, n, sizeof(int), cmp);

  for (int i = 0; i < n; i++)
    cout << array[i] << "\n";
}
```

Version 2: containers, iterators, algorithms

STL provides a number of container types, representing objects that contain other objects. One of these containers is a class called `vector` that behaves like an array, but can grow itself as necessary. One of the operations on `vector` is `push_back`, which pushes an element onto the end of the vector (growing it by one).

A vector contains a block of *contiguous* initialized elements -- if element index k has been initialized, then so have all the ones with indices less than k.

Notice that this means you cannot just bang things into a vector in a random order; it has to be grown like a worm. If efficiency is not a primary concern, you could use a map from integers to values in order to fill a vector in random order.

You can ask a vector how many elements it has with `size`. This is the *logical* number of elements -- the number of elements up to the highest-indexed one you have used. There is a also a notion of *capacity* -- the number of elements the vector can hold before reallocating.
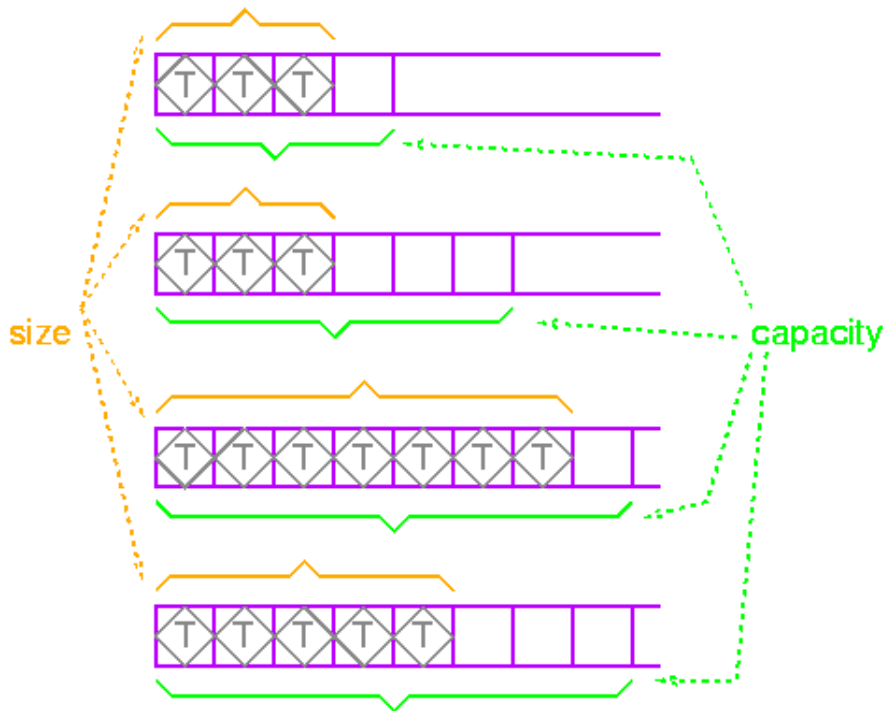
Let's read the elements and push them onto the end of a vector. This removes the arbitrary limit on the number of elements that can be read.

Also, instead of using qsort, we will use the STL sort routine, one of the many algorithms provided by STL. The sort routine is generic, in that it will work on many different types of containers. The way this is done is by having algorithms deal not with containers directly, but with *iterators*.

Preview of iterators

I'll go into iterators in detail later, but for now here is enough to get started.

Iterators provide a way of specifying a position in a container. An iterator can be incremented or dereferenced, and two iterators can be compared. There is a special iterator value called "past-the-end".

You can ask a vector for an iterator that points to the first element with the message `begin`. You can get a past-the-end iterator with the message `end`. The code

```
vector<int> v;
```

```
// add some integers to v
vector::iterator i1 = v.begin();
vector::iterator i2 = v.end();
```

will create two iterators like this:
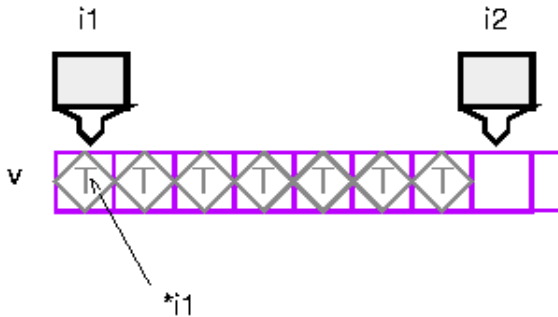
Operations like sort take two iterators to specify the source range. To get the source elements, they increment and dereference the first iterator until it is equal to the second iterator. Note that this is a semi-open range: it includes the start but not the end.

Two vector iterators compare equal if they refer to the same element of the same vector.

Putting this together, here is the new program:

```
#include <string.h>
#include <algo.h>
#include <vector.h>
#include <stdlib.h>
#include <iostream.h>

main ()
{
  vector<int> v;  // create an empty vector of integers
  int input;
  while (cin >> input)     // while not end of file
    v.push_back (input);  // append to vector

  // sort takes two random iterators, and sorts the elements between
  // them.  As is always the case in STL, this includes the value
  // referred to by first but not the one referred to by last; indeed,
  // this is often the past-the-end value, and is therefore not
  // dereferenceable.
  sort(v.begin(), v.end());

  int n = v.size();
  for (int i = 0; i < n; i++)
    cout << v[i] << "\n";
}
```

Incidentally, this is much faster than qsort; at least a factor of 20 on the examples I tried. This is presumably due to the fact that comparisons are done inline.

Version 3: iterator adaptors

In addition to iterating through containers, iterators can iterate over streams, either to read elements or to write them.

An input stream like `cin` has the right *functionality* for an input iterator: it provides access to a sequence of elements. The trouble is, it has the wrong *interface* for an iterator: operations that use iterators expect to be able to increment them and dereference them.

STL provides *adaptors*, types that transform the interface of other types. This is very much how electrical adaptors work. One very useful adaptor is `istream_iterator`. This is a template type (of course!); you parameterize it by the type of object you want to read from the stream. In this case we want integers, so we would use an `istream_iterator<int>`. Istream iterators are initialized by giving them a stream, and thereafter, dereferencing the iterator reads an element from the stream, and incrementing the iterator has no effect. An istream iterator that is created with the default constructor has the past-the-end value.
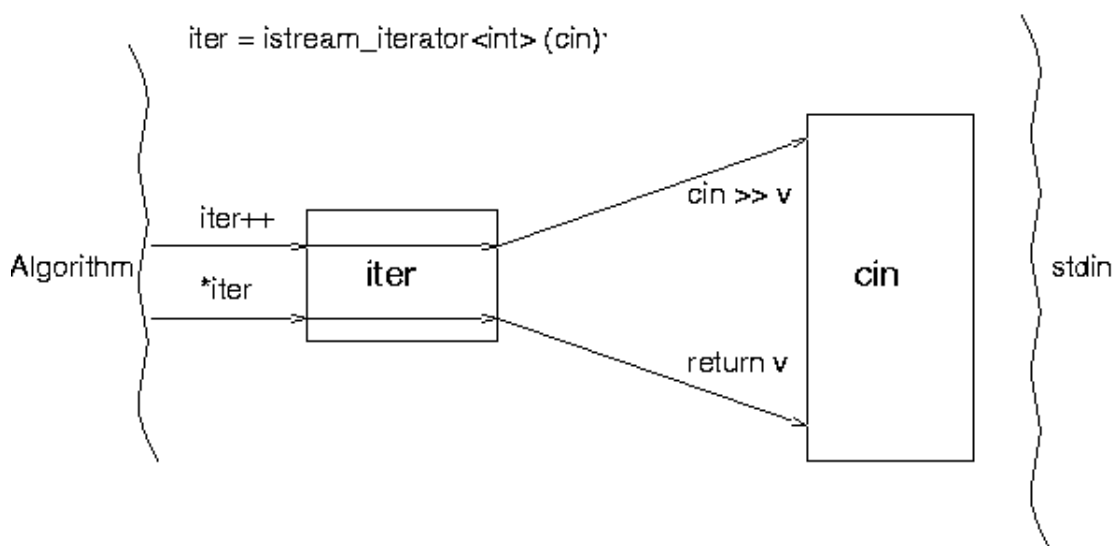
In order to read the elements into the vector from standard input, we will use the STL `copy` algorithm; this takes three iterators. The first two specify the source range, and the third specifies the destination.

The names can get pretty messy, so make good use of typedefs. Iterators are actually parameterized on two types; the second is a distance type, which I believe is really of use only on operating systems with multiple memory models. Here is a typedef for an iterator that will read from a stream of integers:

```
typedef istream_iterator<int,ptrdiff_t> istream_iterator_int;
```

Some implementations of STL, particularly for compilers that do not support default template arguments, define `istream_iterators` with only one parameter, and supply a hard-coded distance type. In this case, you would write:

```
typedef istream_iterator<int> istream_iterator_int;
```

So to copy from standard input into a vector, we can do this:

```
copy (istream_iterator_int (cin), istream_iterator_int (), v.begin());
```

The first iterator will be incremented and read from until it is equal to the second iterator. The second iterator is just created with the default constructor; this gives it the past-the-end value. The first iterator will also have this value when the end of the stream is reached. Therefore the range specified by these two iterators is from the current position in the input stream to the end of the stream.

There is a bit of a problem with the third iterator, though: if v does not have enough space to hold all the elements, the iterator will run off the end, and we will dereference a past-the-end iterator (which will cause a segfault).

What we really want is an iterator that will do insertion rather than overwriting. There is an adaptor to do this, called `back_insert_iterator`. This type is parameterized by the container type you want to insert into.
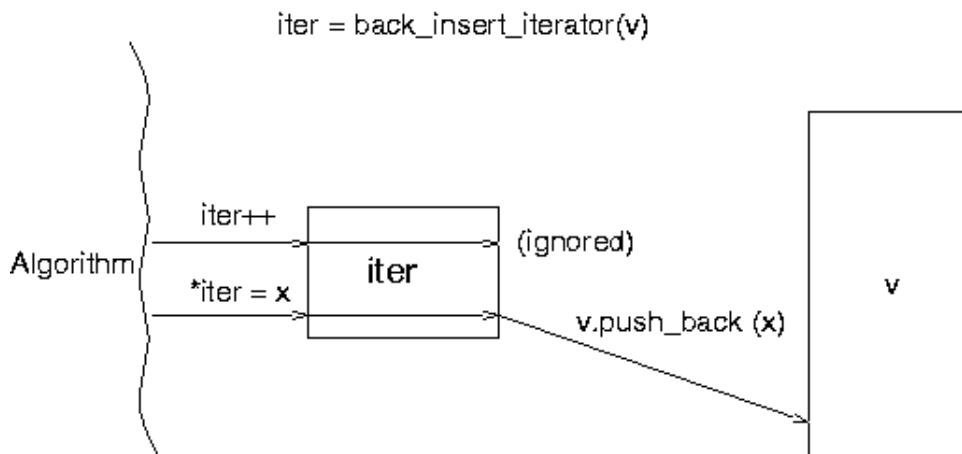
So input is done like this:

```
typedef istream_iterator<int,ptrdiff_t> istream_iterator_int;

vector<int> v;
istream_iterator_int start (cin);
istream_iterator_int end;
back_insert_iterator<vector<int> > dest (v);

copy (start, end, dest);
```

Similarly, to print out the values after sorting, we use copy:

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

`ostream_iterator` is another adaptor; it provides output iterator functionality: assigning to the dereferenced iterator will write the data out. The `ostream_iterator` constructor takes two arguments: the stream to use and the separator. It prints the separator between elements.

iter = ostream_iterator<int>(cout, "\n")

iter++

Algorithm

iter

*iter = x

cout << x << "\n"

cout

stdout
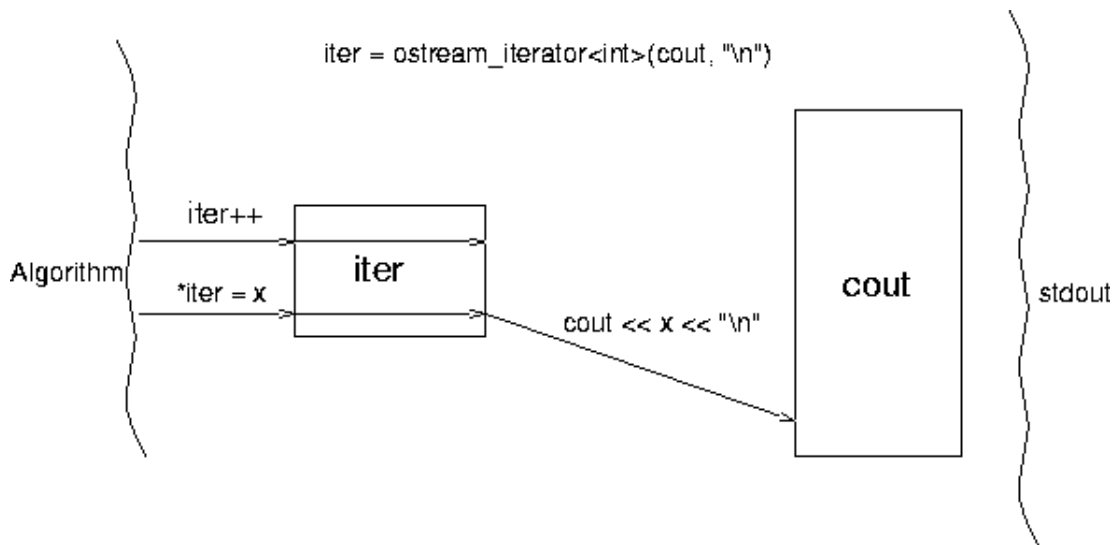
Putting this all together,

```cpp
#include <string.h>
#include <algo.h>
#include <vector.h>
#include <stdlib.h>
#include <iostream.h>

main ()
{
  vector<int> v;
  istream_iterator<int,ptrdiff_t> start (cin);
  istream_iterator<int,ptrdiff_t> end;
  back_insert_iterator<vector<int> > dest (v);

  copy (start, end, dest);
  sort(v.begin(), v.end());
  copy (v.begin(), v.end(), ostream_iterator<int>(cout, "\n"));
}
```

Discussion

I find the final version of the program the cleanest, because it reflects the way I think of the computation happening: the vector is copied into memory, sorted, and copied out again.

In general, in STL, operations are done on containers as a whole, rather than iterating through the elements of the container explicitly in a loop. One obvious advantage of this is that it lends itself easily to parallelization or hairy optimizations (e.g., one could be clever about the order the elements were accessed in to help avoid thrashing).

Most of the STL algorithms apply to *ranges* of elements in a container, rather than to the container as a whole. While this is obviously more general than having operations always apply to the entire container, it makes for slightly clumsy syntax. Some implementations of STL (e.g., ObjectSpace), provide supplementary versions of the algorithms for common cases. For example, STL has an algorithm count that counts the number of times a particular element appears in a container:

```cpp
template <class InputIterator, class T, class Size>
void count (InputIterator start, InputIterator end, const T& value, Size& n);
```

To find how many elements have the value 42 in a vector `v`, you would write:

```
int n = 0;
count (v.begin(), v.end(), 42, n);
```

ObjectSpace defines an algorithm `os_count` that provides a simpler interface:

```
int n = os_count (v, 42);
```

Philosophy

STL components

Containers

Containers are objects that conceptually contain other objects. They use certain basic properties of the objects (ability to copy, etc.) but otherwise do not depend on the type of object they contain.

STL containers may contain pointers to objects, though in this case you will need to do a little extra work.

vectors, lists, deques, sets, multisets, maps, multimaps, queues, stacks, and priority queues, *did I miss any?* are all provided.

Perhaps more importantly, built-in containers (C arrays) and user-defined containers can also be used as STL containers; this is generally useful when applying operations to the containers, e.g., sorting a container. Using user-defined types as STL containers can be accomplished by satisfying the requirements listed in the STL container requirements definition.

If this is not feasible, you can define an adaptor class that changes the interface to satisfy the requirements.

All the types are "templated", of course, so you can have a vector of ints or Windows or a vector of vector of sets of multimaps of strings to Students. Sweat, compiler-writers, sweat!

To give you a brief idea of the containers that are available, here is the heirarchy:
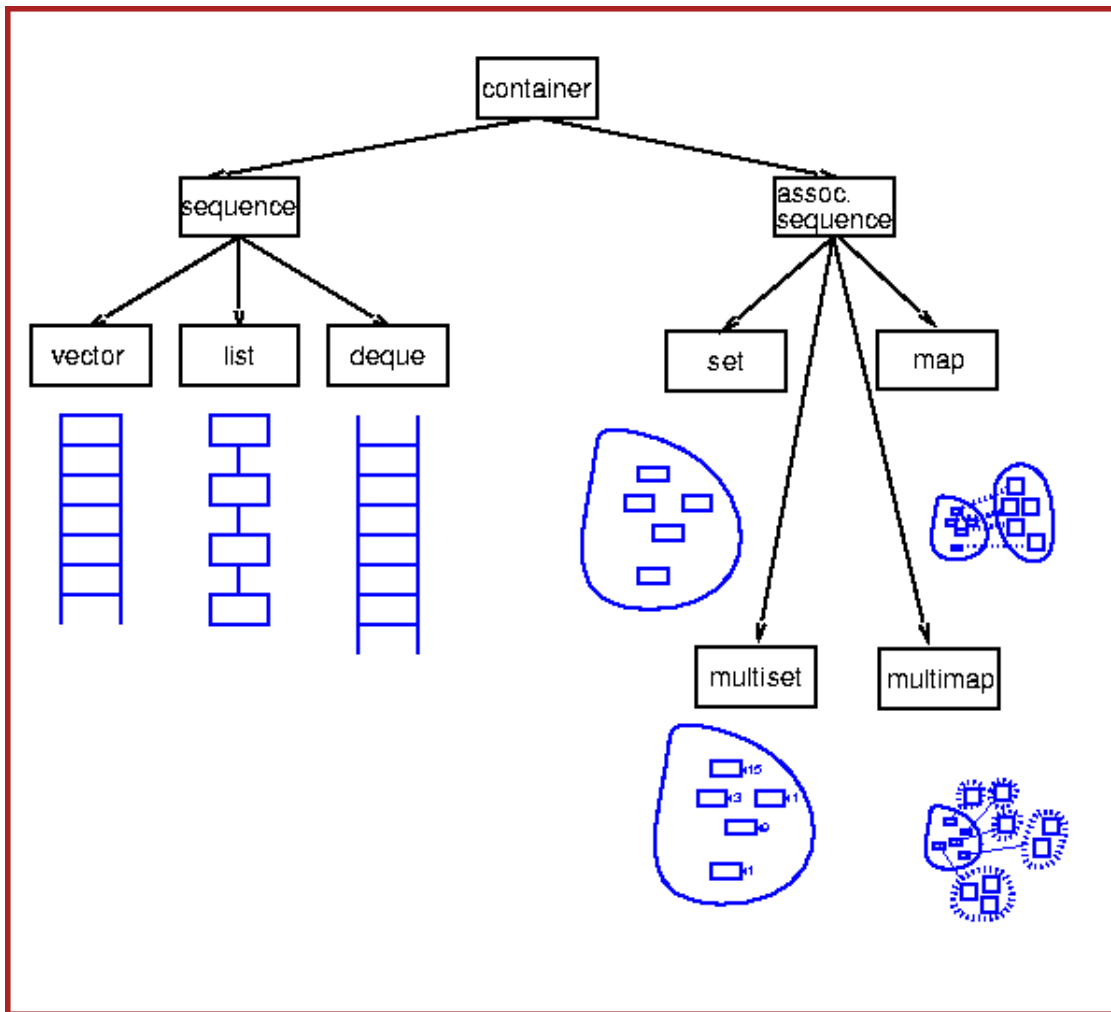
Sequences

Contiguous blocks of objects; you can insert elements at any point in the sequence, but the performance will depend on the type of sequence and where you are inserting.

Vectors

Fast insertion at end, and allow random access.

Lists

Fast insertion anywhere, but provide only sequential access.

Deques

Fast insertion at either end, and allow random access. Restricted types, such as stack and queue, are built from these using adaptors.

Stacks and queues

Provide restricted versions of these types, in which some operations are not allowed.

Associative containers

Associative containers are a generalization of sequences. Sequences are indexed by integers; associative containers can be indexed by any type.

The most common type to use as a key is a string; you can have a set of strings, or a map from strings to employees, and so forth.

It is often useful to have other types as keys; for example, if I want to keep track of the names of all the Widgets in an application, I could use a map from Widgets to Strings.

Sets

Sets allow you to add and delete elements, query for membership, and iterate through the set.

Multisets

Multisets are just like sets, except that you can have several copies of the same element (these are often called bags).

Maps

Maps represent a mapping from one type (the *key* type) to another type (the *value* type). You can associate a value with a key, or find the value associated with a key, very efficiently; you can also iterate through all the keys.

Multimaps

Multimaps are just like maps except that a key can be associated with several values.

Should add other containers: priority queue, bit vector, queue.

Examples using containers

Here is a program that generates a random permutation of the first `n` integers, where `n` is specified on the command line.

```
#include <iostream.h>
#include <vector.h>
#include <algo.h>
#include <iterator.h>

main (int argc, char *argv[])
{
  int n = atoi (argv[1]); // argument checking removed for clarity

  vector<int> v;
  for (int i = 0; i < n; i++)                 // append integers 0 to n-1 to v
    v.push_back (i);

  random_shuffle (v.begin(), v.end());                        // shuffle
  copy (v.begin(), v.end(), ostream_iterator<int> (cout, "\n")); // print
}
```

This program creates an empty vector and fills it with the integers from 0 to n. It then shuffles the

vector and prints it out.

It is quite common to want a sequence of elements with arithmetically increasing values; common enough that there is an algorithm that does something like this for us. It is called *iota*:

```
template <class ForwardIterator, class T>
void iota (ForwardIterator first, ForwardIterator last, T value);
```

This function allows us to fill a range of a container with increasing values, starting with some initial value:

```
vector<int> a(100); // initial size 100

iota (a.begin(), a.end(), 0);
```

This call will fill the array `a` with the values 0, 1, 2...
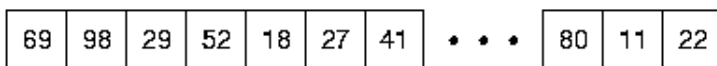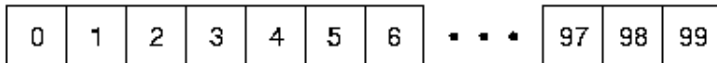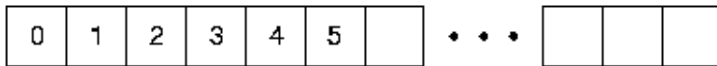
Unfortunately, this is not quite what we wanted -- this overwrites an existing vector, whereas in our case, we had an empty vector, and we wanted the elements appended to it. There are two problems here. The first is that the termination condition for the `iota` function is specified by an iterator; the loop terminates when the moving iterator becomes equal to the terminal iterator.

Many algorithms in STL come in several flavors, corresponding to different terminating conditions. For example, `generate` uses two iterators to specify a range; `generate_n` uses one iterator and an integer to specify the range.

The `iota` function, unfortunately, does not have an `iota_n` counterpart, but it is very easy to write:

```
template <class ForwardIterator, class T>
void iota_n (ForwardIterator first, int n, T value)
{
  for (int i = 0; i < n; i++)
    *first++ = value++;
```

```
}
```

In order to append to the vector instead of overwriting its contents, we will use an adaptor `back_inserter`:

```
#include <iostream.h>
#include <vector.h>
#include <algo.h>
#include <iterator.h>

main (int argc, char *argv[])
{
  int n = atoi (argv[1]); // argument checking removed for clarity

  vector<int> v;
  iota_n (v.begin(), 100, back_inserter(v));

  random_shuffle (v.begin(), v.end());                          // shuffle
  copy (v.begin(), v.end(), ostream_iterator<int> (cout, "\n")); // print
}
```

`back_inserter` is a function that takes a container as an argument, and returns an iterator. The iterator is defined in such a way that writing a value through it and incrementing it will cause the value to be appended to the container.