Brown University ENGN2912B Fall 2015

Scientific Computing in C++

Lecture 11 | The Standard Template Library (STL)

The background to STL

In the early development of C++, before templates were mature, there was no well- defined way to generalize the idea of *containing* a set of data types. Of course, computer scientists had long ago devised a number of efficient data structures (containers) that allow efficient access to elements for various purposes. Examples of such containers are the *vector*, *list*, *set* and *map.*

Bjarne Stroustroup, the inventor of C++, did not provide a specification for containers as part of the language. As a result, C++ programmers were left on their own to implement such containers as they needed them. This deficiency made C++ code difficult to reuse between systems that made different choices on the design of containers.

Some approaches to generic containers were designed so that all objects must inherit from an abstract base class, "object," that all containers hold. This approach did not catch on because of the inherent inefficiency of a single rooted object hierarchy, e.g., if a class doesn't need to be stored in a container it still needs to inherit from the containable baseclass for consistency, i.e. everything is a "containable" object.


Bjarne Stroustrup


Alex Stepanov


David Musser

Around 1990, the situation changed with the design of STL by Alex Stepanov (a former colleague of mine at GE.). His templated approach to containers was accepted rapidly by the C++ community, because the design is logically (mathematically) consistent and well-suited to the processing efficiency inherent in C++. STL became formally part of the C++ language in 1993. Note, Alex was initially trained as a pure mathematician. David Musser (also a colleague) worked with Alex to develop the use of STL in generic programming, where algorithms, such as sorting, are written once to work on all data types.

The STL vector<T>

The most widely used STL container is the templated array, called vector<T>. The STL vector is a sequence of elements. The container can hold any data type, T, and provide a random indexed access to the elements. Here *random* means that any element can be accessed regardless of the previous access.

In order to create a vector it is necessary to specify the type, T. A simple example follows.

```
#include <vector> // STL vector include file
vector<int> v(10);
```

This statement declares v to be of type vector<int>. The number of elements is specified as 10. The initial value of the 10 elements is 0 by default. The elements of v can be accessed and assigned by the same [] syntax as with an ordinary array. That is,

```
v[0] = 3;  v[4]  += v[0];
```

The constructor of vector also allows for the default value of elements to be specified,

```
vector<int> v(l0, 5);
```

In this example, the elements of v are all set to 5 as initial values.

There is no way to know how many elements a standard C++ array can contain from just the pointer to the array. By contrast, the vector member function size(), returns the number of elements in the vector. For the simple example above,

```
size_t n = v.size(); // n == l0
```

Note that the return type of size() is size_t. size_t is a typedef for unsigned int, but may be changed in the future as CPU word lengths increase. It is good practice to use the size t designation, although most programmers simply use unsigned int.

If the vector is empty, as in the following declaration using the default constructor,

```
vector<double> vd;
```

the predicate empty() will return true. That is,

```
bool b = vd.empty();// b == true
```

Dynamic reallocation

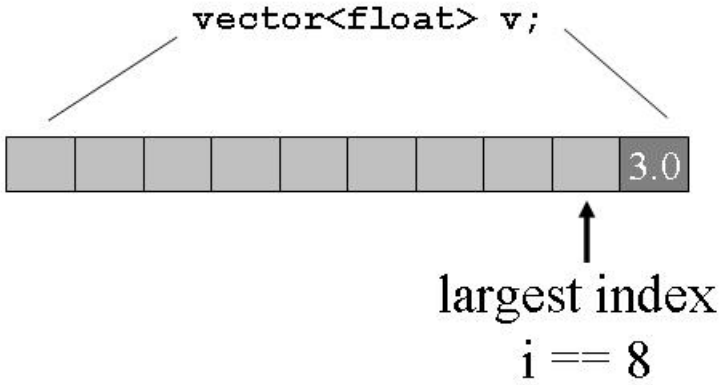
It is often the case that the required size of a vector is not known at compile time. Fortunately, vector provides dynamic reallocation of the number of elements as required. The method,

```
void push_back(const T& e)
```

is used to add new elements to the *end* of the vector. Here "end" means after the element with the largest occupied index, as shown below. The following fragment illustrates the use of push back.

```
vector<float> v(9, 1.41); //allocate a vector with 9 elements
v.push_back(3.0); // push a new element on to the end.
```

The location of the new element is illustrated below.



To understand the reallocation process in more detail consider the declaration,

```
vector<double> x;
```

The initial vector, x, has no elements, i.e. *empty*. As

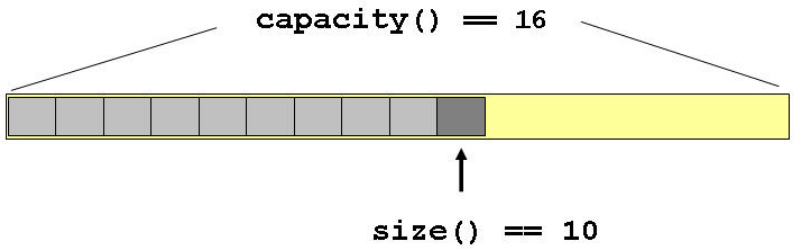
elements are pushed onto the back, e.g.,

```
x.push_back(1.456);
```

the internal allocation of space for the vector is increased in chunks. For example, after the statement, the internal allocation will be at least 1 element.

The reallocation policy of the capacity of the vector depends on the particular implementation of STL. A typical reallocation policy is to increase the capacity by a fixed factor. Using a factor proportional to size means that the processing overhead of reallocation is only proportional to the size of the vector, not proportional to the square, which would be the case if the vector is reallocated each time a new element is added.

The relationship between the vector and its allocation is shown below.



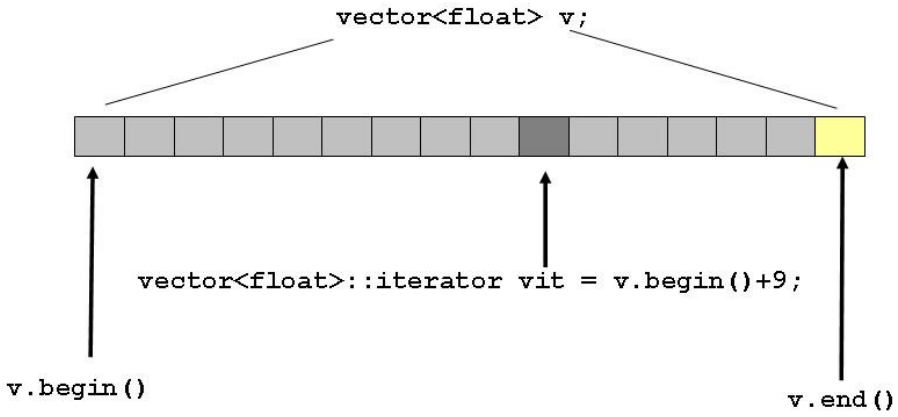
The size method always reports the actual number of elements in the vector. Here thesize is 10 but the capacity is 16. The allocation process goes on behind the scene, and is seldom of direct interest to the programmer[1]. An actual example of allocation in Visual C++ is as follows.

```
#include <vector> vector<char> v;
unsigned kold  = 0;
for(unsigned i= 0;  i<=5l2; ++i) {
  v.push_back('a');
  unsigned k = v.capacity(); //returns the allocated size
  if(k!=kold){
    kold  = k;
    cout << "alloc[" << i << "] = " << k << endl;
  }
 }
```

The output of the program follows.

```
alloc[0] 1
alloc[1] 2
alloc[2] 3
alloc[3] 4
alloc[4] 6
alloc[6] 9
alloc[9] 13
alloc[13] 19
alloc[19] 28
alloc[28] 42
alloc[42] 63
alloc[63]  94
```

The capacity increases as soon as the number of elements in the array is equal to the capacity, and by a factor of roughly 1.5 times the existing size at the time of expansion. The STL iterator



[1] STL does permit the programmer to design their own allocation scheme. The full template signature for the vector constructor is, vector<T, A=defaultAlloc>. So A can be specified

as a custom allocator, but this topic is beyond the scope of the course. STL introduces a new concept, called the *iterator*, for accessing elements of vector. The iterator concept is illustrated above, which behaves like a regular pointer variable into a memory array.

The vector class provides methods, which return an iterator as follows.

```
vector<T>::iterator begin();

vector<T>::iterator  end();
```

The return type of begin and end is vector<T>::iterator, which is a member type of the vector<T>, indicated by the namespace operator ::. The iterator returned by begin() points to the starting element of the vector. Thus, the following relationship holds.

```
vector<float> v(2);
v[0] = 1.43f, v[1]=2.14f;
vector<float>::iterator vit =v.begin();
float x = *vit; // x == 1.43
```

As claimed, vit acts just like a pointer and so can be dereferenced with the * operator. The

iterator can be incremented to point to the next element, i.e.,

```
float y   = *(++vit); // y = 2.14
```

The end() method returns an iterator that points just *past* the last element of the array. So for the example,

```
 v.end()  == v.begin()+2;
```

It is not possible to access a valid vector element using the iterator returned by end(). The purpose of the end() iterator value is to form a convenient designation of the limits in a for loop.

```
for(vector<float>::iterator vit=v.begin(); vit != v.end(); ++vit)
   cout << *vit << endl;
```

The result printed out will be,

```
1.43
2.14
```

The iterator is treated as a pointer without concern for the underlying memory mechanisms. There is no need for the programmer to worry about memory leaks, all the allocation and pointer access is handed internally to the vector class and its allocation.

The const iterator

It may be the case that the programmer wants to be sure that access to the elements of a vector<T> is guaranteed not to change the elements. The vector class has two other bounds methods with a const iterator return type as follows,

```
vector<T>::const iterator begin() const

vector<T>::const iterator end() const
```

These iterators insure that the access they provide to the vector can only read the value of elements, but not change them. For example accessing the vector v from above.

```
vector<T>::const iterator cvit = v.begin();

*cvit = 3;
```

The compiler will complain producing the following error.

```
error C3892:'cvit': you cannot assign to a variable that is const
```

It is necessary to use a const iterator if the vector is passed in to a function with a const reference. For example,

```
void f(vector<float> const& v) {
 for(vector<float>::const iterator vit = v.begin();
 vit!= v.end(); ++vit) {
 //do something with the vector elements
}
```

If the iterator, vit, is not declared const iterator then a compiler error will result. Iterator

Arithmetic

As already indicated in the examples above, the iterator can be treated as any integer type in the application of arithmetic operators. The iterator to vector<float> v is used as an illustration.

```
vector<float>::iterator vit = v.begin();float

x = *(vit += 1); //          x == 2.14

float x1 = *(vit-=1); // x == 1.43, i.e, vit == v.begin()
```

All the integer operators are applicable in a similar manner..

The last element of the vector is given by,

```
float x_back = *(--v.end()); // x_back == 2.14
```

This last example emphasizes again that the `end()` method produces an `iterator` that points to the index position just after the end of sequence of vector elements.

Other vector<T> accessors and mutators

The vector class supports several convenient accessors beyond the iterator arithmetic and dereferencing just discussed.

```
//v[0] = 1.43, v[1]=2.14    initial contents of v

float x = v.front(); // x == 1.43
float y = v.back(); // y = 2.14
```

The entire vector can be reset to empty by the `clear()` method. From the previous example,

```
//v[0] = 1.43, v[1]=2.14    initial contents

v.clear();
bool e = v.empty()  //   e == true
float x = v[0];       // A big crash will occur!
```

An element can be inserted just before a specified iterator position.

```
//v[0] = 1.43, v[1]=2.14    initial contents

unsigned s = v.size();                              //  s == 2;
vector<float>::iterator vit = v.begin()+1;          // points to v[1]

v.insert(vit, 7.3);

// v[0] = 1.43, v[1] = 7.3, v[2]=2.14
s = v.size()                                        //  s == 3
```

Similarly, an element can be erased at a specified iterator position.

```
// v[0] = 1.43, v[1] = 7.3, v[2]=2.14     initial contents

vector<float>::iterator vit = v.begin();  // points to v[0]

v.erase(vit);
```

```
// v[0] = 7.3, v[1]=2.14
s = v.size()      // s == 2
```

Note that the iterators used to designate insert and erase operations are not guaranteed to point to anything meaningful after an operation is complete. Thus using vector iterator in a loop to insert or erase elements of the vector will not work, since the iterator invalid after the first insert or erase. That is,

```
for(vector<float>::iterator vit = v.begin(); vit!=v.end(); ++vit)
      v.erase(vit); // A big crash will occur!
```

pop_back

The inverse of pushing an element onto the back of the vector is to remove (pop) an element from the back.

```
// v[0] = 1.43, v[1] = 7.3, v[2] = 2.14 initial contents
v.pop_back(); // v[0] = 1.43, v[1] = 7.3
unsigned n = v.size() // n == 2
```

resize

It is sometimes necessary to resize a vector under direct program control. For example the programmer may want to refer to elements using the [] operator, e.g.

```
v[3]=3.76f;
```

Unless the vector has size()>=4, this assignment will fail.

Consider the following function, which determines the size of a vector used as an output.

```
void f(vector<float>& v) {   // v is the output of   f
  unsigned n;
  // determine the number of elements v requires, say n == 4
  v.resize(n); // v will now have 4 elements
  // compute the element values
}

vector<float> w; // w.size()==0 initially
f(w);            // w.size()==4 after the function call
```

The resize operation does not necessarily affect all the elements already in a vector.
The operation either erases elements from the end to reduce the size of an array, or adds elements on the end to increase the length of the vector to the specified size. For example,

```
//v[0] = 1.43, v[1]=2.14 initial contents
v.resize(4, 1.0f); //note, a default filler can be specified
//v[0] = 1.43, v[1]=2.14, v[2]=1.0, v[3]=1.0
```

The first two elements of the vector were not affected by the increase in size. The value of the additional elements is 0 by default, but can be specified if desired as shown in the example.

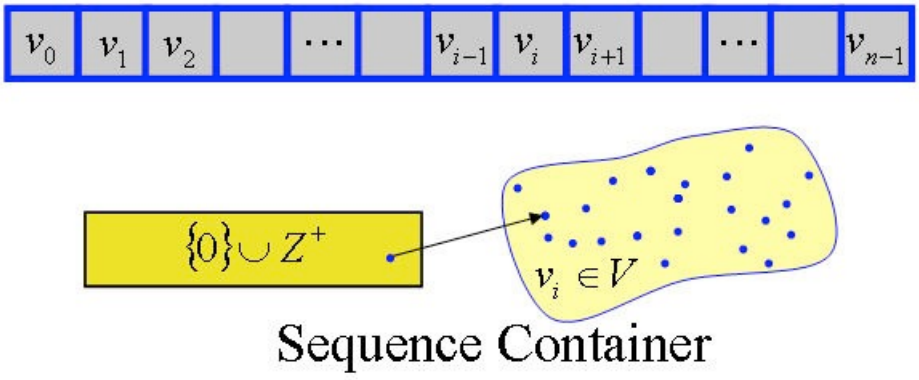
The equality comparison operator ==

A global equality comparison operator is defined that is true if the two vectors have exactly the same elements at all corresponding index positions.

```
vector<double> a(2), b(2), c(2);
a[0]=3.2l; a[l]=l.2l;
b[0]=l.7l; b[l]=5.4;
c[0]=3.2l; b[l]=l.2l;
bool a_eq_b = a==b; // a_eq_b == false
bool a_eq_c = a==c; // a_eq_c  == true
```

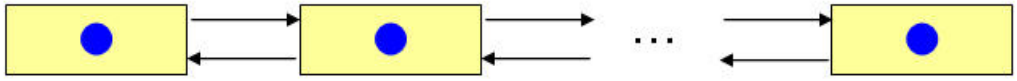
Of course, equality fails immediately for vectors of unequal size.

The sequence container



The figure above represents the *sequence* container type. vector<T> is a particular example of a sequence container, where the type T is called a *value*. The values in a sequence container are in strict linear order and can be accessed by a map from the unsigned integers to the set of values. Thus, a *value* can always be accessed by an integer *position* or *index*.

The `list<T>` container is another example of a sequence container and is illustrated below.



The elements of the list are accessed through pointers where the list is traversed by retrieving the address of adjacent element. Each element contains pointers to the previous and next element.

The advantage of the list<T> container over the vector<T> is that insertions and erasures do not involve reconstructing the entire container. The disadvantage of list<T> is that access is not random, the list must be traversed to reach a particular element.

The STL algorithm library

So far, it is clear that the templated vector<T> class is very convenient and can be applied to any data type without any re-implementation. However, the design is no big conceptual leap beyond the obvious need for a flexible container and the intrinsic power of templates.

The real innovation of STL is the concept of *generic* programming. Traditionally, it is very difficult to reuse code across different applications. There is always something special about function and class interfaces that must be adapted to the new use. However, with the advent of templates it became possible to implement algorithms on containers that will work for any situation, provided that a few operations and comparisons are defined for the element type of the container.

The payoff is huge, since considerable effort can go into making the generic algorithms efficient, and their reuse requires little or no change, thus serving millions of programmers.

Let's start with a simple generic algorithm, *find*. The ability to determine if a specific value is an element of a vector is frequently required. For the vector<float> container, the *find* algorithm can be implemented directly as,

```
vector<float> v;
// fill v with some values
bool found  = false;
float x =1.87f;
for(vector<float>::iterator vit=v.begin();vit!=v.end()&&!found; ++vit)
if(*vit==x) //== must be defined for the elements of the vector
  found = true;
//at this point found will be true if x is an element of v
```

Note that in general any type T that is applied to the *find* algorithm must have the equality comparison operator == defined. This requirement is minimal since most types will need the test for equality anyway.

The implementation above is deficient in several ways:

1. The implementation does not report the location in the vector where x is found;
2. The *find* algorithm should be implemented as a templated function to handle all types;
3. The *find* algorithm should be able to work on other containers such as a `list` or a `queue`, or even the plain memory array, which will be discussed later.

Fortunately, all of these deficiencies are eliminated by the design of the STL find algorithm, which is demonstrated in the following example.

```
#include <string>
#include <algorithm>  //<-       needed  for  find(.)

vector<string> vs;        //create  a  vector  of strings vs.push_back("The");
vs.push_back("cat");
vs.push_back("in");
vs.push_back("the");
vs.push_back("hat");

vector<string>::iterator vbegin = vs.begin(); // start of vs
vector<string>::iterator vend = vs.end(); // just past end of vs

// the call to  find
vector<string>::iterator  sit  =  find(vbegin,  vend,  "the");

// if the find failed then the value of sit will be vend
if(sit!=vend)
 cout << "Found \"" << *sit << "\" at location " << (sit-vbegin) <<  endl;
else
 cout  <<  "Not  found"  <<  endl;
```

In this case the program prints out[2],

```
Found  "the"  at  location 3
```

The result of find is to return an iterator pointing to the location of the found element. If the query value is not found then, find returns an iterator pointing to the end of the array. Since this position corresponds to the same value as that returned by end(), the failure of the find algorithm can be easily tested by,

```
bool found = sit != vs.end(); //failure means sit  ==    vs.end()
```

The iterator power hierarchy

To understand the previous code example a bit better, it will be helpful to see the general signature of the find algorithm.

---

[2] As a side note, to print " it is necessary to use the \ character that tells the compiler to interpret the following character literally as an ASCII character. That is '\"' is interpreted as the character ". The designation '\\' produces the backslash character itself. Without the \, the compiler is free to interpret the character as part of C++ syntax.

```
template <class inputiterator >
inputiterator find(inputiterator first, inputiterator last, const T& value);
```

As expected, find is a templated function with three arguments. However, there is a strange extra template argument type, inputiterator, which has not been seen before.

The design of STL classifies the flexibility that a container iterator has to access its elements and what effect the iterator can have on the container contents by dereferencing the iterator. The properties of the five iterator categories are shown in the table below, where x and y are of type #Iterator, where # is one of the categories, Input, Forward, Bidirectional, and Output Random Access.

Iterator Op.\ It. Type	Input	Output	Forward	Bidirectional	Random Access
x(y)(copy const)	y	y	y	y	y
x=y (assignment)	y	y	y	y	y
x==y	y	N	y	y	y
x!=y	y	N	y	y	y
x++, ++x	y	y	y	y	y
x--, --x	N	N	N	y	y
*x	r-value	l-value	y	y	y
(*x).f	y	N	y	y	y
x->f	y	N	y	y	y
x + n	N	N	N	N	y
x += n	N	N	N	N	y
x - n	N	N	N	N	y
x -= n	N	N	N	N	y
x[n]	N	N	N	N	y

Table from http://www.oreillynet.com/pub/a/network/2005/10/18/what-is-iterator-in-c-plus-plus.html?page=4. Also, take a look at https://en.wikipedia.org/wiki/Operators_in_C_and_C++

From the table, it can be seen that an inputiterator is fairly weak, it cannot go backward and can only move forward one container element at a time. If the inputiterator is dereferenced, the result can only exist on the right side of an assignment, i.e., *r-value*. The dereference operator, *, produces a r-value constant, i.e., T x = *iter;
A good example of how an inputiterator can arise is an input stream with no buffer. The stream can only advance not index backward. The elements of the stream can be looked at as r-values to be assigned to a receiving l-value in the program.

An Outputiterator only provides l-values to be used in an assignment, and can only advance one element at a time and not index backward. For example, *iter = 3;

The Forwarditerator can read and write container elements but can only advance.
Naturally, the Bidirectionaliterator can move forwards and backwards but onlyone element at a time.

The most powerful iterator is the RandomAccessiterator , which can do everythingin the first column of the table above.

Thus, the find function signature,

```
template <class inputiterator >
inputiterator find(inputiterator first, inputiterator last, const T& value);
```

indicates that find only requires the power of an inputiterator to operate and the query value can be guaranteed to be found. If the element is found, it is possible to dereference the returned inputiterator to get a r-value to be assigned to a variable. Inthe example above the r-value used as a string constant.

```
vector<string>::iterator sit = find(vbegin, vend, "the");
string x = *sit;       // *sit  is a  r-value
```

Normallythese distinctions among iterators are not too important to the typical programmer. The iterator returned by the STL vector class is a RandomAccessiterator and thus has no restrictions to worry about.

These distinctions are mostly important to programmers that want to develop *generic* algorithms. If it is known that a container can provide a RandomAccessiterator then it is guaranteed that any element can be accessed in constant time. If so, then algorithms like binary search might be considered as a strategy for finding the query element. If the container can only provide an inputiterator then exhaustive search is the only option.

The C++ array as a container

One of the great design choices of STL generic algorithms is that they all work on the plain C++ array. Consider the following example.

```
string  as[5] = {"The", "cat", "in", "the", "hat"};
string* aend  = as+5;  // l +  last element address
string* ait   = find(as, aend, "hat");
if(ait!=aend)
  cout << "Found \"" << *ait << "\" at location " << ait-abegin << endl;
else
  cout  <<  "Not  found"  <<  endl;
```

The result is as expected.

```
Found "hat" at location 4
```

Note that in this case, the pointer to the memory array, as, has the power of a RandomAccessiterator.

This design choice enables a programmer to be able to design his own containers using allocation of blocks of memory. As long as the programmer is able to implement iterators of appropriate power to the storage elements, the STL algorithms will all function efficiently.

The predicate

STL provides a similar algorithm to find, find if, as specified below.

```
template <class inputiterator, class  Predicate>
inputiterator find_if(inputiterator first, inputiterator last, Predicate  pred);
```

This signature introduces another new kind of template argument, the `Predicate` type.

The Predicate type is any function of the container element that can return a type that can be cast to bool. The following example shows an application of the find if function.

```
int is_odd(int n) {
  return n%2;
}

int main() {
  vector<int> vi;
  vi.push back(0);
  vi.push back(2);
  vi.push back(3);
  vector<int>::iterator iit = find_if(vi.begin(), vi.end(), is_odd);
  if(iit!=vi.end())
    cout  <<  "I found the odd number "  <<  *iit << endl;
}
```

The output of the program is as follows.

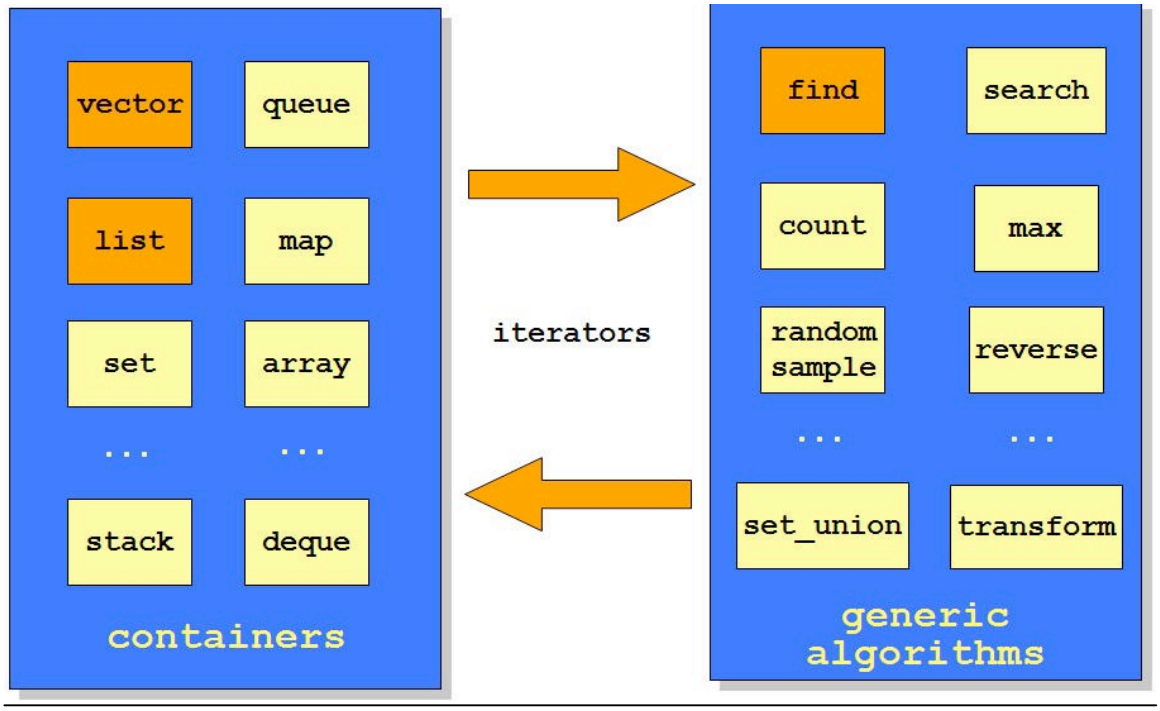
```
I found the odd number 3
```

This example shows that an algorithm can be templated over both data types and functions. This design choice also contributes to the flexibility of the algorithm library. To use find if all the programmer needs to do is implement a Predicate function and they are ready to use find if on any STL container.

A closely related algorithm is `for_each`:

```
template <class InputIterator, class Function>
Function for_each(InputIterator first, InputIterator last, Function fn);
```

A brief recap



The figure above shows an overview of STL. The part of the library that has been discussed so far is shown in orange. As discussed, the containers and iterators can be used without ever considering the algorithm part of STL. However, considerable effort has been applied to insure that the STL algorithms are as efficient in space and computation as possible.

The following are some reasons why the C++ programmer should be familiar with the algorithm side of STL and make use of the algorithms where possible:

1. Efficiency - It may be the case that a typical C++ programmer can implement *find* using a `for` loop with the same efficiency as the STL `find` algorithm. However for more difficult algorithms, such as *sort*, it is very unlikely an average programmer will beat the computer scientists who implemented the STL `sort` algorithm.

2. Correctness - In an earlier lecture the problem of implementing correct loop indexing and loop termination conditions is a common source of bugs in programs. The loops used internally to the STL algorithms are fully debugged and guaranteed to be correct.

3. Clarity - As soon as the algorithm becomes a bit complex, such as search, which finds a subsequence in a container, the program loop must be studied to determine what operation is being carried out, often leading to misinterpretation. The STL algorithm names and signatures are part of the C++ language and therefore

universally understood.