Standard Template Library

http://www.ics.uci.edu/~thornton/ics45c/Notes/StandardLibrary/

STL

- A software library for the C++ programming language that influenced many parts of the C++ Standard Library. It provides four components called *algorithms*, *containers*, *functional*, and *iterators*
- They are not part of the C++ language, but were created in addition to the built-in data types.
- The creation of Alexander Stepanov (1979)

vector

- The first step using vector is to include the appropriate header:

```
#include <vector>
```

- Vectors are declared with the following syntax:

```
vector<type> variable_name (number_of_elements);
```

- The number of elements is optional. You could declare it like this:

```
vector<type> variable_name;
```

```
std::vector<int> v;
// declares a vector of integers
```

vector

```
std::vector<int> v;

for (int i = 0; i < 10; i++)
{
    v.push_back(i);
}
```

```
std::vector<int> v{1, 2, 3, 4, 5};
```

The objects in the initializer list become the vector's elements, and the size of the vector is the number of objects in the initializer list

vector

Statement below creates a vector containing 10 elements that all have the value 100

```
std::vector<int> v(10, 100);
```

Individual elements are accessible using the same **[ ]** (*indexing*) operator that when accessing elements of an array. This operator has been overloaded to do the appropriate thing on a vector and is not *bounds-checked*

```
for (int i = 0; i < 10; i++)
{
    v[i] += 10;
    std::cout << v[i] << std::endl;
}
```

vector member function `at()`

- Throws an exception when an attempt is made to access an element outside of the boundaries of the vector's size
- Like the [ ] operator, include the ability to use it to write to a cell

```
for (int i = 0; i < 10; i++)
{
    v.at(i) += 10;
    std::cout << v.at(i) << std::endl;
}
```

The `at()` member function returns a reference to the cell in question

vector member function `at()`

- If the vector is `const`, you won't be able to assign into its cells

```
void foo(const std::vector<int>& v)
{
    v.push_back(15);    // illegal
    v[3] = 50;          // illegal
    v.at(3) = 50;       // illegal

    for (size_t i = 0; i < v.size(); i++)  // legal
    {
        std::cout << v.at(i) << std::endl; // legal
    }
}
```

vector

http://www.cplusplus.com/reference/vector/vector/

Generic algorithms

- In addition to containers, the C++ Standard Library provides a set of *generic algorithms*, which generalize commonly-occurring operations that you might like to perform:
  - Apply the same function to each of a range of values
  - Find a value in a range of values that has a particular property (e.g., is a positive number, etc.)
  - Remove from a range of values the ones that have a particular property
  - Shuffle the values in a range of values randomly
  - Sort the values in a range of values

Iterators

- In the C++ Standard Library, iterators are the glue between the *containers* and *the generic algorithms*
- An *iterator* is an *abstraction* for a position in a container, whose main role is the provide access to the value stored at that position (e.g., in a particular cell of a `std::vector`), while hiding all of the details about the container's underlying implementation
- Given an iterator, a generic algorithm can access (and potentially modify) the values stored in a container without having to know what kind of container it is

Iterators

- In the C++ Standard Library, iterators are the glue between the *containers* and *the generic algorithms*
- An *iterator* is an *abstraction* for a position in a container, whose main role is the provide access to the value stored at that position (e.g., in a particular cell of a `std::vector`), while hiding all of the details about the container's underlying implementation
- Given an iterator, a generic algorithm can access (and potentially modify) the values stored in a container without having to know what kind of container it is

Iterators

- Iterators Make the Algorithms Container Independent
- Algorithms Do Depend on Element-Type Operations
  - == operator to compare each element to the given value
  - Other algorithms require that the element type have the < operator
  - most algorithms provide a way to supply our own operation to use in place of the default operator

Iterators

- You can use the `*` operator to dereference an iterator, just like you can use the `*` operator to dereference a pointer.
- The `++` operator moves an iterator forward (i.e., to the next position in the container).
- The `--` operator moves an iterator backward (i.e., to the previous position in the container)
- `!=` can be compared against another iterator
- You can sometimes use other kinds of arithmetic (e.g., given an iterator `i`, writing `i + 3` or `i += 3` can be legal)

http://www.cs.northwestern.edu/~riesbeck/programming/c++/stl-iterators.html

Iterators

- Example: call to an algorithm `copy()`

```
copy(v.begin(), v.end(), l.begin());
```

`copy()` takes three arguments, all iterators:

  - an iterator pointing to the first location to copy from
  - an iterator pointing one element past to the last location to copy from
  - an iterator pointing to the first location to copy into

- `v` and `l` are some STL containers and `begin()` and `end()` are member functions that return iterators pointing to locations within those containers

Iterators

Note 1:

`begin()` returns a location you can dereference

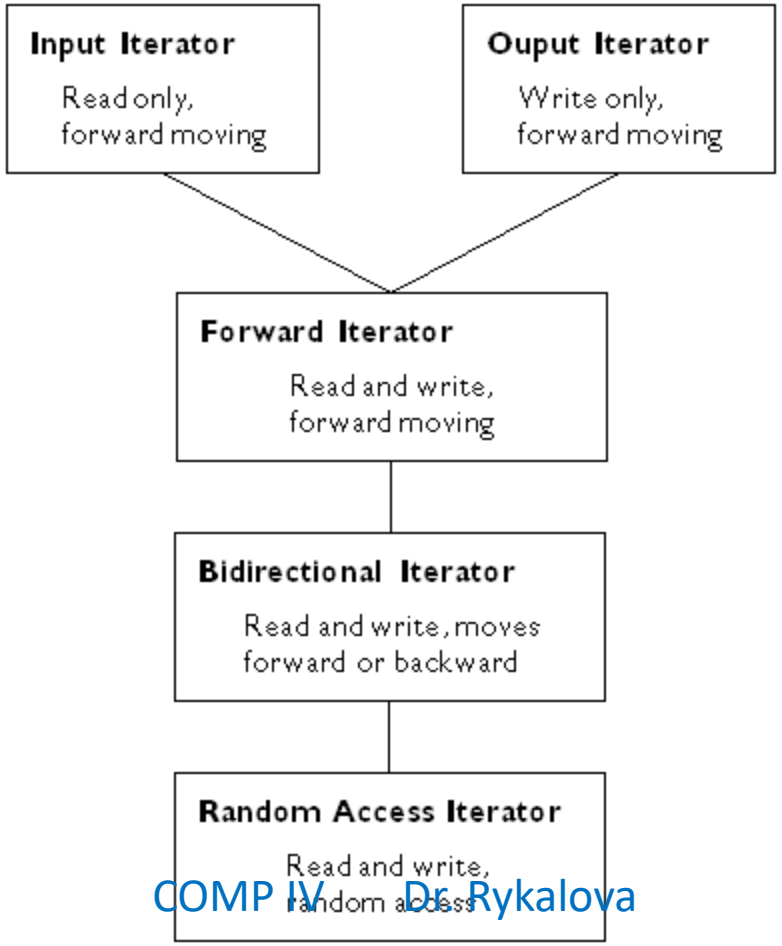
`end()` does not. Dereferencing the end pointer is an error. The end pointer is only to be used to see when you've reached it.

Note 2:

`copy()` assumes that the destination already has room for the elements being copied. It would be an error to copy into an empty list or vector. However, this limitation is easily overcome with insert operators

Iterator Classes

- Categories of kind of iterators
- Each category specifies the operations the iterator supports



Iterator Classes: Input

Input Iterator

- is a useful but limited class of iterators
- If `iter` is an InputIterator, you can use:
  - `++iter` and `iter++` to increment it, i.e., advance the pointer to the next element
  - `*iter` to dereference it, i.e., get the element pointed to
  - `==` and `!=` to compare it another iterator (typically the "end" iterator)

http://www.cs.northwestern.edu/~riesbeck/programming/c++/stl-iterators.html

Iterator Classes : Input

- All STL containers can return at least this level of iterator.
- *Example*: code that prints out everything in a vector:

```
vector<int> v;
vector<int>::iterator iter;

v.push_back(1);
v.push_back(2);
v.push_back(3);

for (iter = v.begin(); iter != v.end(); iter++)
     cout << (*iter) << endl;
```

Iterator Classes : Output

- If `iter` is an InputIterator, you can use:
  - `++iter` and `iter++` to increment it, i.e., advance the pointer to the next element
  - `*iter =` … to store data in the location pointed to
- Output iterators are only for storing. If something is no more an output iterator, you can't read from it with `*iter`, nor can you test it with `==` and `!=`
- Two very useful subclasses of *OutputIterator*:
  - insert operators
  - ostream iterators

Iterator Classes : Insert

- Insert iterators let you "point" to some location in a container and insert elements
  - You do this with just dereferencing and assignment:

```
*iter = value;
```

- This inserts the value in the place pointed to by the iterator
  - If you assign again, a new value will be inserted

Iterator Classes : Insert

- `back_inserter<container>` returns an *OutputIterator* pointing to the end of the container:
  - using the container's `push_back()` operation.
- `front_inserter<container>` returns an *OutputIterator* pointing to the front of the container:
  - using the container's `push_front()` operation.
- `inserter<container, iterator>` returns an *OutputIterator* pointing to the location pointed to by iterator of the container:
  - using the container's `insert()` operation

Iterator Classes

Iterator form	Produced by
input iterator	`istream_iterator`

output iterator	`ostream_iterator`
	`inserter()`
	`front_inserter()`
	`back_inserter()`

bidirectional iterator	*list* *set* and *multiset* *map* and *multimap*
random access iterator	ordinary pointers *vector* *deque*

http://stdcxx.apache.org/doc/stdlibug/2-2.html

Passing a Function to an Algorithm

- Predicate - a third argument to an algorithm
- An expression that can be called and that returns a value that can be used as a condition
  - Unary
  - Binary
- possible to convert the element type to the parameter type of the predicate

Sorting Algorithms

- sort words by size
  - maintain alphabetic order among the elements that have the same length
  - use the `stable_sort` algorithm
    - ✓ maintains the original order among equal elements

Sorting Algorithms

```
elimDups(words); // put words in alphabetical
                 // order & remove duplicates
// resort by length, maintaining alphabetical
// order among words of the same length
stable_sort(words.begin(),words.end(),isShorter);
for (const auto &s : words) // no need to copy
                            // the strings
    cout << s << " ";  // print each element
                       // separated by a space
cout << endl;
```

Lambda expression

Lambda *expression*

- We can pass any kind of ***callable*** object to an algorithm
- Callables:
  - Functions
  - function pointers
  - Functor
    - class that overload the function-call operator
  - lambda expressions

lambda expression

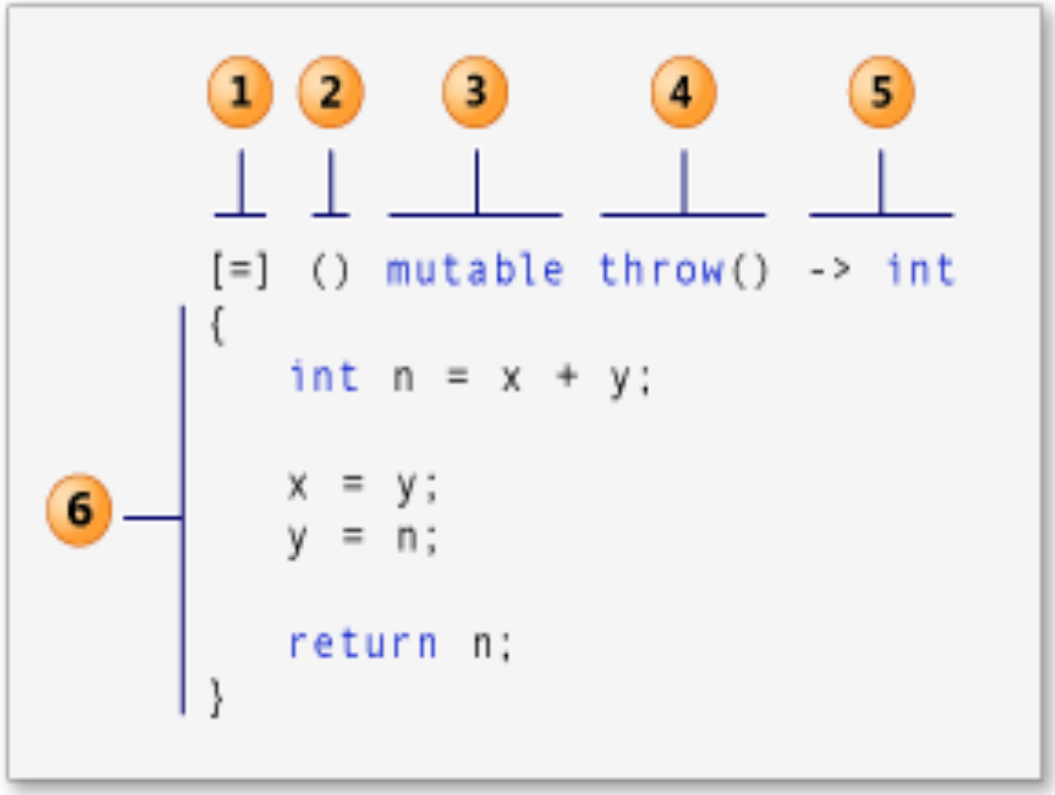
- Way of defining an anonymous function object right at the location where it is invoked or passed as an argument to a function
- Typically lambdas are used to encapsulate a few lines of code that are passed to algorithms or asynchronous methods

lambda expression

```
void abssort(float* x, unsigned n) {
   std::sort(x, x + n,
         // Lambda expression begins
         [](float a, float b) {
         return (std::abs(a) < std::abs(b));
   } // end of lambda expression
   );
}
```

https://msdn.microsoft.com/en-us/library/dd293608.aspx

lambda expression



1. capture clause/list (the lambda-introducer)
2. parameter list *Optional* (the lambda declarator)
3. mutable specification *Optional*
4. exception-specification *Optional*
5. trailing-return-type *Optional*

lambda expression

```
[captures] (parameters) ->
        returnTypesDeclaration { lambdaStatements; }
```

`[captures]` capture clause/lambda introducer

- specifies which outside variables are available for the lambda function and how they should be captured (by value or reference)

lambda expression

- A lambda expression can have more power than an ordinary function by having access to variables from the enclosing scope
- Capture external variables from enclosing scope by three ways :
  - Capture by reference
  - Capture by value
  - Capture by both (mixed capture)

lambda expression

- Syntax used for capturing variables :

  [&] capture all external variable by reference

  [=] capture all external variable by value

  [a, &b] capture a by value and b by reference

  [this] captures the current object (*this) by reference

- A lambda with empty capture clause [ ] can access only those variable which are local to it

http://en.cppreference.com/w/cpp/language/lambda

https://stackoverflow.com/questions/7627098/what-is-a-lambda-expression-in-c11

lambda expression

```
[&] { x += 1; }          // capture by reference
```

```
class foo {
    int &x;
public:
    foo(int &x) : x(x) {}
    void operator()() const { x += 1; }
};
```

lambda expression

```
[=] { return x + 1; } // capture by value
```

```
class bar {
    int x;
public:
    bar(int x) : x(x) {}
    int operator()() const { return x + 1; }
};
```

lambda expression

```
// define a special-purpose custom printing function
void print_it(int i)
{
    cout << ":" << i << ":";
}
//...
// apply print_it to each integer in the list
for_each(int_list.begin(), int_list.end(), print_it);
cout << endl;
```

```
for_each(int_list.begin(), int_list.end(),
                [](int i) {cout << ":" << i << ":"; });
cout << endl;
```

lambda expression

```
auto lambda = [] { cout << Code within a lambda
                expression" << endl; };

auto lambda = [](void) { cout << Code within a
           lambda expression" << endl; };

auto lambda = [](void) -> void { cout << "Code
within a lambda expression" << endl; };
```

lambda expression

```
#include <iostream>
using namespace std;

int main()
{
  auto sum = [](int x, int y) { return x + y; };
  cout << sum(5, 2) << endl;
  cout << sum(10, 5) << endl;
}
```

https://www.youtube.com/watch?v=p7b-6EFIN28