The STL is a major component of the C++ Standard Library; it is a large collection of general-purpose generic classes, functions, and iterators:

1. **Generic containers** that take the element type as a parameter.
   - e.g., vector, list, deque, set, stack, queue, ...

2. Different kinds of **iterators** that can navigate through the containers.

3. **Algorithms** that (via iterators) perform an interesting operation on a range of elements.
   - e.g., sort, random_shuffle, transform, find

---

**Design Philosophy of the STL**

Generic containers that take the element type as a parameter.
- know (almost) nothing about the element type
  - exception: ordered containers expect elements to have operator<
- operations are (mostly) limited to add, remove, and retrieve
- define their own iterators

Useful, efficient, generic algorithms that:
- know nothing about the data structures they operate on
- know (almost) nothing about the elements in the structures
- operate on range of elements accessed via iterators

---

The C++ Standard Template Library (STL)

STL algorithms are designed so that (almost) any algorithm can be used with any STL container, or any other data structure that supports iterators.
- Element type must support copy constructor/assignment.

- For ordered containers, the element type must support operator< or you can provide a special functor (function-object) of your own.

- The STL assumes value semantics for its contained elements: elements are copied to/from containers more than you might realize.

- The container methods and algorithms are highly efficient; it is unlikely that you could do better.
No Inheritance in the STL!

Basically, the primary designer (Alexander Stepanov) thinks that OOP (i.e., inheritance) is wrong, and generic programming is better at supporting polymorphism, flexibility and reuse.

- Templates provide a more flexible (“ad hoc”) kind of polymorphism.
- The containers are different enough that code reuse isn’t really practical.
- Container methods are not virtual, to improve efficiency.

Review: Polymorphic Containers

Suppose we want to model a graphical Scene that has an ordered list of Figures (i.e., Rectangles, Circles, and maybe other concrete classes we haven’t implemented yet).

- Figure is an abstract base class (ABC)
- Rectangle, Circle, etc. are derived classes

What should the list look like?
1. vector <Figure>
2. vector <Figure&>
3. vector <Figure*>
```cpp
#include <iostream>
#include <string>
#include <vector>
using namespace std;

class Balloon {
  public:
    Balloon (string colour); // Copy constructor
    virtual ~Balloon();
    virtual void speak() const;
  
  private:
    string colour;
};

Balloon::Balloon(string colour) : colour{colour} {
  cout << colour << " balloon is born" << endl;
}

Balloon::Balloon(const Balloon &b) : colour{b.colour} {
  cout << colour << " copy balloon is born" << endl;
}

void Balloon::speak() const {
  cout << "I am a " << colour << " balloon" << endl;
}

Balloon::~Balloon() {
  cout << colour << " balloon dies" << endl;
}

// How many Balloons are created?
int main (int argc, char* argv[]) {
  vector<Balloon> v;
  Balloon rb ("red");
  v.push_back(rb);
  Balloon gb ("green");
  v.push_back(gb);
  Balloon bb ("blue");
  v.push_back(bb);
}
```

---

STL Containers

C++98/03 defines three main data-container categories:

1. Sequence containers: vector, deque, list
2. Container adapters: stack, queue, priority_queue

C++11 adds:

0. Addition of emplace[_front, _back].
1. Sequence containers: array, forward_list
2. [nothing new]
3. [nothing new]

C++14 adds:

1. Non-member cbegin/cend/rbegin/rend/crebegin/crend.

---

STL Containers: Conceptual View

Sequence containers

Ordered associative containers

C++98/03 defines three main data-container categories:

1. Sequence containers: vector, deque, list
2. Container adapters: stack, queue, priority_queue

C++11 adds:

0. Addition of emplace[_front, _back].
1. Sequence containers: array, forward_list
2. [nothing new]
3. [nothing new]

C++14 adds:

1. Non-member cbegin/cend/rbegin/rend/crebegin/crend.
1. Sequence Containers

There is a total ordering of contiguous values (i.e., no gaps) on elements based on the order in which they are added to the container.

They provide very similar basic functionality, but differ on:

1. Some access methods.
   - `vector` and `deque` allow random access to elements (via `[]` / `at()`), but `list` allows only sequential access (via iterators).
   - `deque` allows `push_back` and `push_front` (+ `pop_front`, + `front`).

2. Performance.
   - `vector` and `deque` are optimized for (random access) retrieval, whereas `list` is optimized for (positional) insertion/deletion.

---

**vector<T>**

Can think of as an expandable array that supports access with bounds checking, via `vector<T>::at()`.

Vector elements must be stored contiguously according to the C++ standard, so pointer arithmetic will work and O(1) random access is guaranteed.

- So it pretty much has to be implemented using a C-style array.

Calling `push_back` when vector is at capacity forces a reallocation.

---

### Access kind | Complexity | API support
--- | --- | ---
random access | O(1) | `operator[]` or `at()`
append/delete last | O(1)*/O(1) | `push_back/pop_back`
prepend/delete first | O(N) | *not supported as API call*
random insert/delete | O(N) | `insert/erase`

---

(likely) vector Implementation
**deque<T>**

== “double-ended queue”; similar to vectors, but allow fast insertion/deletion at beginning and end.

Random access is “fast”, but no guarantee that elements are stored contiguously.

− So pointer arithmetic won’t work.
− `operator[]` and `at()` are overloaded to work correctly.

<table>
<thead>
<tr>
<th>Access kind</th>
<th>Complexity</th>
<th>API support</th>
</tr>
</thead>
<tbody>
<tr>
<td>random access</td>
<td>O(1)</td>
<td><code>operator[]</code> or <code>at()</code></td>
</tr>
<tr>
<td>append/delete last</td>
<td>O(1)/O(1)</td>
<td><code>push_back/pop_back</code></td>
</tr>
<tr>
<td>prepend/delete first</td>
<td>O(1)/O(1)</td>
<td><code>push_front/pop_front</code></td>
</tr>
<tr>
<td>random insert/delete</td>
<td>O(N)</td>
<td><code>insert/erase</code></td>
</tr>
</tbody>
</table>

---

**vector vs. deque**

So, in real life, should you use `vector` or `deque`?

• If you need to insert at the front, use `deque`.
• If you need to insert in the middle, use `list`.

Random access to elements is constant time in both, but a `vector` may be faster in reality.

Reallocations:

• take longer with a `vector`.
• `vector` invalidates external refs to elements, but not so with a `deque`.
• `vector` copies elements (which may be objects), whereas `deque` copies only `ptrs`.

---

**Integrity of External References**

```cpp
#include <vector>
#include <deque>
using namespace std;

int main(int argc, char* argv[]) {
    cout << "\nWorking with a vector:" << endl;
    vector<int> v;
    v.push_back(4); v.push_back(3);
    v.push_back(7); v.push_back(15);
    int* p = &v.back();
    cout << "p " << *p << " " << v.at(3) << " " // Must be same
    << *p << " " << &v.at(3) << endl; // Must be same
    v.push_back(99); // Causes a reallocation
    cout << "p " << *p << " " << v.at(3) << " " // May be different**
    << *p << " " << &v.at(3) << endl; // Probably different
```

** p is no longer pointing to v[3], but the old value of 15 may still "be there"
Integrity of External References

```cpp
cout << "\nWith a deque:" << endl;
d< int> d;
d.push_back(4); d.push_back(3);
d.push_back(7); d.push_back(15);
p = &d.back();
cout << *p << " " << d.at(3) << " " // Must be same
   << p << " " << &d.at(3) << endl; // Must be same
d.resize(32767); // Probably causes realloc
cout << *p << " " << d.at(3) << " " // Must be same
   << p << " " << &d.at(3) << endl; // Must be same
}
// My output below, YMMV but comments above will hold
With a vector:
15 15 0x7ff87bc039cc 0x7ff87bc039cc
15 15 0x7ff87bc039cc 0x7ff87bc039cc
With a deque:
15 15 0x7ff87c00220c 0x7ff87c00220c
15 15 0x7ff87c00220c 0x7ff87c00220c
```

(C++11) std::array

Effectively, a very thin wrapper around a C++ array, to make it a little more like a fixed-size vector.

**C++ array vs. std::array**
- not implicitly converted by compiler into a pointer
- supports many useful functions like
  - an at() method, for safe, bounds-checked accessing
  - a size() method that returns the extent of the array (which you set when you instantiated the array)

**std::array vs. std::vector**
- strong typing: if you know the array size should be fixed, enforce it!
- array contents may be stored on the stack rather than the heap
- std::array is faster and more space efficient

(C++11) std::forward_list

Basically, a plain-old singly-linked list.

**std::forward_list vs. std::list**
- more space efficient, and insertion/deletion operators are slightly faster
- no immediate access to the end of the list
  - no push_back(), back()
- no ability to iterate backwards
- no size() method

```
list<T>

Implemented as a (plain old) doubly-linked list (PODLL)
Designed for fast insertion and deletion.

Supports only sequential access to elements via iterators.
- No random access via indexing operator[] or at().

<table>
<thead>
<tr>
<th>Access kind</th>
<th>Complexity</th>
<th>API support</th>
</tr>
</thead>
<tbody>
<tr>
<td>random access</td>
<td>O(N)</td>
<td>not supported as an API call</td>
</tr>
<tr>
<td>append/delete last</td>
<td>O(1)</td>
<td>push_back/pop_back</td>
</tr>
<tr>
<td>prepend/delete first</td>
<td>O(1)</td>
<td>push_front/pop_front</td>
</tr>
<tr>
<td>random insert/delete</td>
<td>O(1)</td>
<td>insert/erase</td>
</tr>
</tbody>
</table>
```

If you've arrived at the elt; O(N) to get there)
2. Container Adapters

Usually a trivial wrapping of a sequence container, to provide a specialized interface with ADT-specific operations to add/remove elements.

- stack, queue, priority_queue

You can specify in the constructor call which container you want to be used in the underlying implementation:

stack: vector, deque*, list
queue: deque*, list
priority_queue: vector*, deque

[* means default choice]

```
template <typename T>
class Stack {
  public:
    Stack () ;
    virtual ~Stack () ;
    void push ( T val );
    void pop () ;
    T top () ;
    void print () ;
    bool isEmpty () ;
  private:
    vector<T> s ;
};
```

```
template <typename T>
class Stack {
  public:
    Stack () ;
    virtual ~Stack () ;
    void push ( T val ) ;
    void pop () ;
  private:
    vector<T> s ;
    // _ etc
}
```

STL Container Adapters

Implemented using the adapter design pattern.

1. Define the interface you really want. e.g., for stack, we want push(), pop()
2. Instantiate (don’t inherit) a private data-member object from the “workhorse” container class that will do the actual heavy lifting (e.g., vector).
3. Define operations by delegating to operations from the workhorse class.

"The STL Way"

“The STL Way” encourages you to define your own adapter classes, based on the STL container classes, if you have special-purpose needs that are almost satisfied by an existing STL class.

- STL doesn’t use inheritance or define any methods as virtual.
- Encourages reuse via adaptation, rather than inheritance.
- Interface can be exactly what you want, not constrained by inheritance.
Inheritance vs. Adaptation

Suppose we would like to implement a card game and we want to model a pile of playing cards.

- Actually, a pile of `Card`, since the cards will be a shared resource and will get passed around.

We want it to support natural `CardPile` capabilities, like `addCard`, `discard`, `merge`, `print`, etc.

We also want the client programmer to be able to treat a `CardPile` like a sequential polymorphic container: `iterate`, `find`, `insert`, `erase`.

Inheriting from an STL Container

```cpp
class CardPile : public vector<Card*> {  
    public:
        // Constructors and destructor
        CardPile();
        virtual ~CardPile();
        // Accessors
        void print() const;
        int getHeartsValue() const;
        // Mutators
        void add(.Card* card);
        void add(CardPile & otherPile);
        void remove(Card* card);
        shuffle();
    }
```

Traditional STL Adaptation

```cpp
class CardPile {
    public:
        // Constructors and destructor
        CardPile();
        virtual ~CardPile();
        // Accessors
        void print() const;
        int getHeartsValue() const;
        // Mutators
        void add( Card * card );
        void add( CardPile & otherPile );
        void remove( Card * card );
        void shuffle();

        // If want shuffling to be repeatable, pass in a random number generator.
        void shuffle( std::mt19937 & gen );
    private:
        std::vector<Card*> pile;
    };
```
But there is another way...

Public inheritance:

```cpp
class Circle : public Figure { ...
```

- Inside the class definition of Circle, we have direct access to all non-private members of Figure.
- Circle is a subtype of Figure, and it provides a superset of the Figure's public interface.

```
private
```

Private inheritance:

```cpp
class Circle : private Figure { ...
```

- Inside the class definition of Circle, we have direct access to all non-private members of Figure.
- Circle is not a subtype of Figure; it does not support Figure's public interface.
- Client code that instantiates a Circle cannot treat it polymorphically as if it were a Figure.
  - Cannot invoke any Figure public methods.
  - Cannot instantiate a Circle to a Figure*. 

Private Inheritance of STL Container

```cpp
class CardPile : private std::vector<Card*> { 
public:
  // Constructors and destructor
  CardPile();
  virtual ~CardPile();

  // Accessors
  void print() const;
  int getHeartsValue() const;

  // Mutator ops ("natural" for CardPile)
  void add( Card * card );
  void add( CardPile & otherPile );
  void remove( Card * card );
  void shuffle();

  // If want shuffling to be repeatable, pass in a random number generator.
  void shuffle( std::mt19937 & gen );
```

Private Inheritance of STL Container (cont)

```cpp
using std::vector<Card*>::iterator;
using std::vector<Card*>::const_iterator;
using std::vector<Card*>::begin;
using std::vector<Card*>::end;
using std::vector<Card*>::size;
using std::vector<Card*>::at;
using std::vector<Card*>::pop_back;
using std::vector<Card*>::back;
using std::vector<Card*>::empty;
```
This approach is safe because it breaks polymorphism!

- Cannot instantiate a CardPile to a vector<Card*>, so there is no risk of a call to the wrong destructor causing a memory leak.
- The client code cannot accidentally call the wrong version of an inherited non-virtual method.
  - None of the inherited functions are visible to clients unless explicitly made so by using using (in which case, the parent definition is used).
  - If you redefine an inherited function, the client code will get that version, since they can’t see the parent version.

Private inheritance is not conceptually very different from adaptation.

- It requires a little less typing.
- It encourages reuse of the parent class’s interface where applicable.

A set is a collection of (unique) values.

- Typical declaration:
  ```
  set<T> s;
  ```

- T must support a comparison function with strict weak ordering
  - i.e., anti-reflexive, anti-symmetric, transitive.
  - Default is operator<
  - Can use a user-defined class, but you must ensure that there is a “reasonable” operator< defined or provide an ordering functor to the set constructor.

Sets do not allow duplicate elements.

- If you try to insert an element that is already present in the set, the set is unchanged. Return value is a pair <iterator, bool>.
  - The second of the pair indicates whether the insertion was successful.
  - The first of the pair is the position of the new/existing element.
Equality vs. Equivalence

Equivalence

The container search methods (e.g., find, count, lower_bound,...) will use the following test for equality for elements in ordered associate containers even if you have your own definition of operator==.

\[
\text{if } ( \neg (a<b) \land \neg (b<a) )
\]

Equality

Whereas the STL algorithms find, count, remove_if compare elements using operator==.

map<key,T>

A map maps a key to a (unique) value.

- Typical declaration:
  
  \[
  \text{map<T1, T2> m;}
  \]

- \(T1\) is the key field type; it must support a comparison function with strict weak ordering
  
  - i.e., anti-reflexive, anti-symmetric, transitive.
  
  - Default is operator<.

  - Can use a user-defined class, but you must ensure that there is a “reasonable” operator< defined or provide an ordering functor to the map constructor.

- \(T2\) is the value field; can be anything that is copyable and assignable.

Querying Map for Element

Intuitive method (lookup via indexing) will insert the key if it is not already present:

\[
\text{if } ( \text{words["bach"] == 0} ) \quad // \text{bach not present}
\]

Alternatively, can use map's find() operation to return an iterator pointing the queried key/value pair.

\[
\text{map<string, int>::iterator it; it = words.find("bach"); if ( it == words.end(...))} \quad // \text{bach not present}
\]

end() is iterator value that points beyond the last element in a collection
#include <iostream>
#include <string>
#include <cassert>
#include <map>
using namespace std;

// Examples adapted from Josuttis
int main () {
    map<string,string> dict;
    dict["car"] = "voiture";
    dict["hello"] = "bonjour";
    dict["apple"] = "pomme";
    cout << "Printing simple dictionary" << endl;
    for ( auto i: dict ){
        cout << i.first << "\t" << i.second << endl;
    }

    multimap<string,string> mdict;
    mdict.insert(make_pair( "car", "voiture" ));
    mdict.insert(make_pair( "car", "auto" ));
    mdict.insert(make_pair( "car", "wagon" ));
    mdict.insert(make_pair( "hello", "bonjour" ));
    mdict.insert(make_pair( "apple", "pomme" ));
    cout << "Printing all def of \"car\"" << endl;
    for (multimap<string,string>::const_iterator i=mdict.lower_bound("car"); i!=mdict.upper_bound("car"); i++) {
        cout << (*i).first << ":    " << (*i).second << endl;
    }
}

How are they Implemented?

[multi]set and [multi]map are usually implemented as a red-black tree (see CS240).

- This is a binary search tree that keeps itself reasonably balanced by doing a little bit of work on insert/delete.
- Red-black trees guarantee that lookup/insert/delete are all \( O(\log N) \) worst case, which is what the C++ standard requires.
- Optimized search methods (e.g., find, count, lower_bound, upper_bound).

Because the containers are automatically sorted, you cannot change the value of an element directly (because doing so might compromise the order of elements).

- There are no operations for direct element access.
- To modify an element, you must remove the old element and insert the new value.

(C++11) Unsorted Associative Containers

They are:

unordered_[multi]set
unordered_[multi]map

They are pretty much the same as the sorted versions except:

- They’re not sorted.
- They’re implemented using hash tables, so they are \( O(1) \) for insert/lookup/remove.
- They do provide iterators that will traverse all of the elements in the container, just not in any “interesting” order.
The C++ Standard Template Library (STL)

The STL is a major component of the C++ Standard Library; it is a large collection of general-purpose generic classes, functions, and iterators:

1. **Generic containers** that take the element type as a parameter.
   - e.g., vector, list, deque, set, stack, queue, ...

2. Different kinds of **iterators** that can navigate through the containers.

3. **Algorithms** that (via iterators) perform an interesting operation on a range of elements.
   - e.g., sort, random_shuffle, transform, find

---

STL Containers Provide Iterators

If `c` is a vector, deque, list, set, map, etc. then

- `c.begin()`/`c.begin()` returns a pointer to the first element
- `c.end()`/`c.end()` returns a pointer to just beyond the last element
- `operator++` is defined to advance to the next element

**Example**

```cpp
vector<string>::const_iterator vi = v.begin();
map<int,string>::iterator mi = mymap.begin();
list<Figure*>::reverse_iterator li = scene.rbegin();
```

- The iterator types are **nested** types, defined inside the respective container classes, who understand what "++" should mean!

---

STL Iterators

The iterator is a fundamental **design pattern**.

- It represents an abstract way of walking through all elements of some interesting data structure.
- You start at the beginning, advance one element at a time, until you reach the end.

In its simplest form, we are given:

- A pointer to the first element in the collection.
- A pointer to just beyond the last element; reaching this element is the stopping criterion for the iteration.
- A way of advancing to the next element (e.g., `operator++`, `operator--`).

---

Kinds of Iterators

Iterator **categories** are hierarchical, with lower level categories adding constraints to more general categories.

```
input iterator
  +--- forward iterator
  +--- bidirectional iterator
  +--- random access iterator

output iterator
  +--- istream, ostream

unordered_set, unordered_multiset, unordered_map, unordered_multimap
  +--- list, set, multiset, map, multimap
    +--- vector, deque
```

Why should you care??
**Input and Output Iterators**

Input iterators are read-only iterators where each iterated location may be read only once.

Output iterators are write-only iterators where each iterated location may be written only once.

Operators: `++`, `*` (can be `const`), `==`, `!=` (for comparing iterators)

Mostly used to iterate over streams.

```cpp
#include <iostream>
#include <iterator>
...
copy ( istream_iterator<char> (cin), // input stream
      istream_iterator<char> (), // end-of-stream
      ostream_iterator<char> (cout) ) // output stream
```

**Other Kinds of Iterators**

Forward iterators can read and write to the same location repeatedly.

Bidirectional iterators can iterate backwards (---) and forwards (++)

Random access iterators: can iterate backwards (---) and forwards (++) access any element ([[]]), iterator arithmetic (+, −, +=, −=).

**Insert Iterators (Inserters)**

Iterator that inserts elements into its container:

- `back_inserter`: uses container's `push_back()`
- `front_inserter`: uses container's `push_front()`
- `inserter`: uses container's `insert()`

```cpp
#include <algorithm>
#include <iterator>
#include <iostream>
#include <vector>
#include <string>

istream_iterator<string> is (cin); // input stream
istream_iterator<string> eof; // end sentinel
vector<string> text;
copy ( is, eof, back_inserter( text ) );
```

**Concluding Remarks**

Iterators are a great example of both information hiding and polymorphism.

- Simple, natural, uniform interface for accessing all containers or data structures.
- Can create iterators (STL-derived or homespun) for our own data structures.
- STL iterators are compatible with C pointers, so we can use STL algorithms with legacy C data structures.