Standard Library Containers, Iterators, and Algorithms

CS 355

Standard Template Library (STL)

- Provides a set of *generic containers* and *algorithms* for storing and managing collections of data.
- Uses C++'s template mechanism for *parameterized types*.
- Designed for efficiency (usually speed since templates can cause bloat).
- Heavy use of iterators, which provide a generic means for traversing the elements within a container.

Container Flavors

	Container	Description	Efficient at
sequence containers	vector<T>	Dynamic array.	Random access. Add / delete at end.
	deque<T>	Double ended queue.	Random access. Add / delete at either end.
	array<T,n>	Static (fixed size) array.	Random access.
	list<T>	Doubly-linked lists.	Sequential access forward or backwards.
	forward_list<T>	Singly-linked list.	Sequential access forward.
associative containers	set<T>, multiset<T>	Ordered set (Red Black Tree)	O(log N) insert / delete. Ordered traversal.
	map<K,V>, multimap<K,V>	Ordered dictionary (RBT), maps keys to values	O(log N) insert / delete. Ordered traversal.
unordered containers	unordered_set<T>	Hashed set	Average O(1) insert /delete.* *YMMV
	unordered_map<K,V>	Hashed Dictionary maps keys to values	Average O(1) insert /delete.

vector<T> Example

```
#include <vector>

// ...

std::vector<Card> deck;
for (int r = ACE; r <= KING; r++)
  for (int s = SPADES; s <= HEARTS; s++)
    deck.push_back(Card(r,s));

for (int i = 0; i < 52; i++) {
  const int j = arc4random() % 52;
  const Card tmp = deck[i];
  deck[i] = deck[j];
  deck[j] = tmp;
}

std::vector<Card> hand(5);
for (int i = 0; i < 5; i++) {
  hand[i] = deck.back();
  deck.pop_back();
}
```

Iterators

- An **iterator** is an object is used to *traverse* through the objects in a container.
- These are the core mechanism for accessing elements in a container.
  - Most STL operations use iterators for input/output (e.g., `find(key)` returns an iterator).
  - The *standard algorithms* interact with containers via iterators.

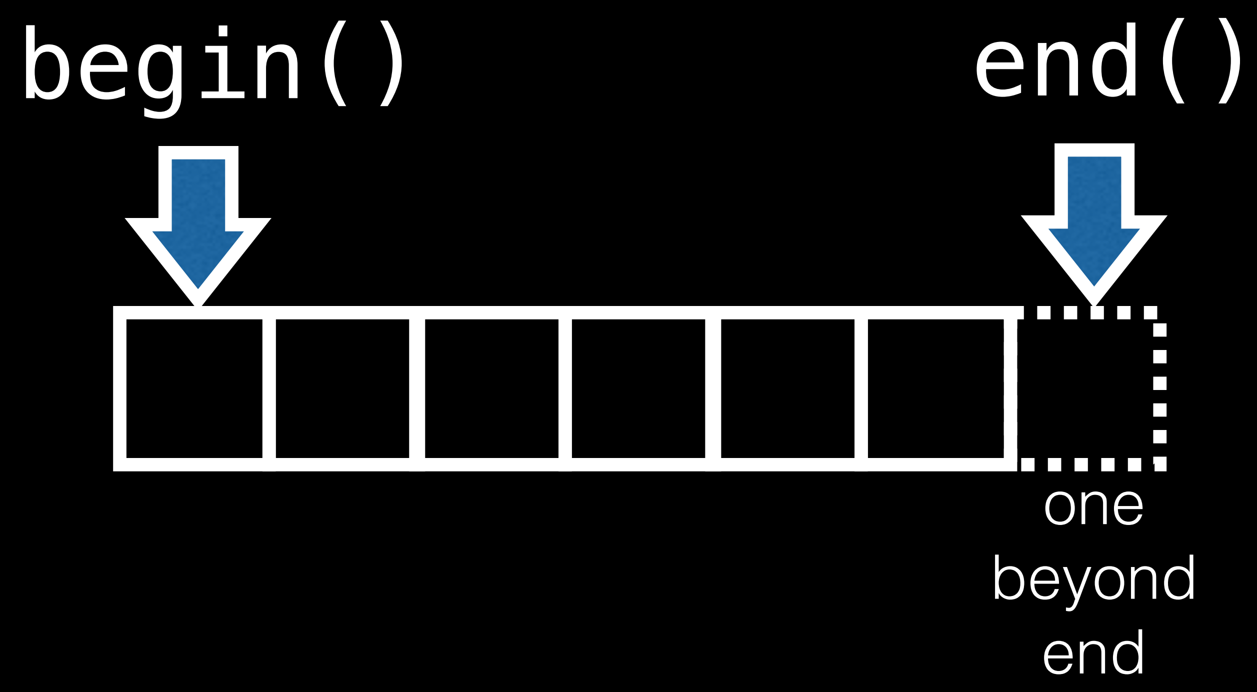
Iterators as Pointers

- An *iterator* is a generalization of a *pointer* that addresses a contiguous sequence of elements.
- Each container provides its own type of iterator(s).
- The behavior of an iterator is defined by the usual “pointer” operations:

operator	Description
`*, ->`	*Dereference* : return element at current position.
`++`	*Increment* : step forward to next element.
`!=, ==`	Do two iterators represent the same position?
`=`	Assign to an iterator (set its position).

`begin()` and `end()`

- Container classes provide basic member functions that allow iterators to access their elements.
- **`begin()`** : returns an iterator for the first item.
- **`end()`**: returns an iterator one beyond the last item.



Iterating through `vector` elements

`vector` provides `iterator` and `const_iterator` types.

```
int numAces = 0;
for (std::vector<Card>::const_iterator iter = deck.begin();
     iter != deck.end(); ++iter)
  if (iter->rank == ACE)
     numAces++;
```

Always use <u>pre</u>-increment to increment iterator. Why?

Less verbose using `auto`:

```
int numAces = 0;
for (auto iter = deck.cbegin(); iter != deck.cend(); ++iter)
  if (iter->rank == ACE)
     numAces++;
```

Use `cbegin()/cend()` for `const_iterator`.

Even less verbose using range operator:

```
int numAces = 0;
for (auto elem : deck)
  if (elem.rank == ACE)
     numAces++;
```

Common `vector` operations

Operation	Description
`c.size()`	Number of elements.
`c.empty()`	Contains any elements?
`c.capacity()`	Max number of elements without reallocation.
`c.reserve(n)`	Enlarge capacity (if not enough yet).
`c.shrink_to_fit()`	Reduce capacity to actual size.
`c[i]`	read (l-value), write (r-value) element `i`.
`c.push_back(e)`	Add element `e` onto end,
`c.back()`	Last element.
`c.pop_back()`	Remove last element.
`c.resize(n)`	Resize vector.
`c.clear()`	Remove all elements.

Plus all the usual constructors (default, copy, move),
a destructor, assignment, comparisons, etc...

An ordered set of primes

```
#include <set>
// ...

std::set<int> somePrimes = {
    2, 3, 5, 7, 11, 13, 17, 19, 23, 29
};
int lastPrime = *somePrimes.rbegin();
for (int n = lastPrime+1; n < 100; n++) {
  bool isPrime = true;
  for (int num : somePrimes)
    if (n % num == 0) {
      isPrime = false;
      break;
    }
  if (isPrime)
    somePrimes.insert(n);
}
```

{ ... } initialization

reverse iterator

range loop

Printing Cards

We can overload the *output stream* (`ostream`) operator << to "pretty print" our cards.

```
#include <iostream>
// ...

std::ostream& operator<<(std::ostream& os, const Card& card) {
  const static std::string rankStr[] = {
    "", "Ace", "2", "3", "4", "5", "6", "7", "8", "9", "10",
    "Jack", "Queen", "King"
  };
  const static std::string suitStr[] = {
    "Spades", "Clubs", "Diamonds", "Hearts"
  };
  os << rankStr[card.rank] << " of " << suitStr[card.suit];
  return os;
}
```

An ordered set of cards

In order to store `Card`'s in an ordered `set` we need to define a comparison function. Here we *overload* the `<` operator for comparing two `Card`'s:

```
bool operator<(const Card& A, const Card& B) {
  return A.rank < B.rank || (A.rank == B.rank &&
                             A.suit < B.suit);
}
```

We create a set of 5 `cards` from the first 5 cards in the `deck` and print them out:

```
std::set<Card> cards(deck.begin(), deck.begin()+5);
for (Card c : cards)
    std::cout << c << std::endl;
```

Function Objects

- We consider anything the *behaves* like a function to be a function.
- We can create a *"function object"* by overloading the parentheses `()` operator:

```
class Transmorgifier { // class for function objects
  const double scale, shift;
public:
  Transmorgifier(double a, double b) : scale{a}, shift{b} {}
  double operator() (double x) const { // f(x)
    return scale*x + shift;
  }
};

// ...

Transmorgifier f(10,3);
std::cout << f(5) << std::endl; // outputs "53"
```

Advantages of Function Objects

1. Each function can be bundled with *state information* (*e.g.*, `scale` and `shift` in our `Transmorgifier` objects).

2. Each function can have its *own type* (normally function types are only distinguished by their signature).

3. Can be faster when templates are used (more on that later....).

The Standard Library provides a large set of predefined functions.

An ordered set using a comparator function object.

We define a class for function objects that compare `Card`'s:

```
struct CardComparator {
  bool operator()(const Card& A, const Card& B) {
    return A.rank < B.rank || (A.rank == B.rank &&
                               A.suit < B.suit);
  }
};
```

We use the comparator class as part of the type:

```
std::set<Card,CardComparator> cards(deck.begin(),
                                    deck.begin() + 5);
```

This allows us to create sets with different comparators.

Common `set` operations

Operation	Description
`c.size()`	Number of elements.
`c.empty()`	Contains any elements?
`c.count(val)`	Number of elements equal to `val`.
`c.find(val)`	Return position of `val` (or `end()` if not found).
`c.insert(val)`	Insert `val` into set.
`c.insert(beg,end)`	Insert values from iterators.
`c.erase(val)`	Erase `val` from set,
`c.erase(beg,end)`	Erase range of values from iterators.
`c.clear()`	Remove all elements.
`set_union`	(provided by algorithms)
`set_intersection`	(provided by algorithms)
`set_difference`	(provided by algorithms)

Plus all the usual constructors (default, copy, move),
a destructor, assignment, comparisons, etc...

Equality Test and Hash Function Object for unordered set of `Card`'s

Hashing requires an "*equality*" test:

```
bool operator==(const Card& A, const Card& B) {
  return A.rank == B.rank && A.suit == B.suit;
}
```

(we could also create a function object for this)

and a *hash function*:

```
struct CardHash { // perfect hash function
  std::size_t operator()(const Card& card) const {
    return (card.rank - 1)*4 + card.suit;
  }
};
```

Unordered Set of Card's

```
#include <unordered_set>
// ...

std::unordered_set<Card,CardHash> handy;

int i = 0; // get last 5 cards in deck
for (auto iter = deck.rbegin(); i < 5; ++iter, i++)
   handy.insert(*iter);

for (Card c : handy)
  std::cout << c << std::endl;
```

Ordered Map
Key = `Card`, Value = `bool`

```
#include <map>
// ...

std::map<Card,bool> faceCardMap;
for (Card card : deck)
  faceCardMap[card] =
    JACK <= card.rank && card.rank <= KING;

for (auto iter = faceCardMap.cbegin();
     iter != faceCardMap.cend(); ++iter) {
  std::pair<Card,bool> keyVal = *iter;
  std::cout << keyVal.first;
  std::cout << (keyVal.second ? " is " : " is not ");
  std::cout << "a face card." << std::endl;
}
```

key : < operator defines order

“associative array”

(key,val) stored in `std::pair`

Unordered Map

```
#include <unordered_map>
#include <string>
#include <iostream>
#include <sstream>
//... define operator<<, operator==, and CardHash for Cards

std::unordered_map<Card,std::string,CardHash> cardToStringMap;
for (Card card : deck) {
  std::stringstream ss;
  ss << card;
  cardToStringMap[card] = ss.str();
}

auto iter = cardToStringMap.find(Card(JACK,HEARTS));
if (iter == cardToStringMap.end())
  std::cout << "not found!" << std::endl;
else
  std::cout << "found [" << iter->second << "]" << std::endl;
```

<algorithm>

- Provides about 80 algorithms.
- They operate on sequences.
    - Input usually pairs of iterators: [begin,end).
    - Output often a single iterator where end = “not found.”
- Designed for correctness, maintenance, and <u>performance</u>.
- If you find yourself writing code with lots of loops, often these loops can be replaced with an “algorithm” (from the standard library or one of your own).
- Work well with lambda function (which we will cover later).

Generating a deck of cards

```
#include <algorithm>
using namespace std; // assume std::
// ...

Card nextCard() {
  static Card card {ACE, SPADES};
  Card next = card;
  card.suit = (card.suit + 1) % 4;
  if (card.suit == 0) card.rank++;
  if (card.rank > KING) card.rank = ACE;
  return next;
}

// ...

vector<Card> deck(52);
generate(deck.begin(),deck.end(),nextCard);
```

Shuffling a deck

```
#include <cstdlib>
// ...

int randy(int n) {return arc4random() % n;}

// ...

// Five shuffles
for (int i = 0; i < 5; i++)
  random_shuffle(deck.begin(),deck.end(),randy);
```

Deal five cards and sort hand

```
vector<Card> hand;
for (int i = 0; i < 5; ++i) {
  Card card = deck.back();
  deck.pop_back();
  hand.push_back(card);
}

// sort by rank, then suit
sort(hand.begin(),hand.end());
```

Check for flush (all five cards of same suit)

```
class SameSuit { // function obj comparing suits
  Card referenceCard;
public:
  SameSuit(const Card& c) : referenceCard(c) {}
  bool operator()(const Card& card) const {
    return referenceCard.suit == card.suit;
  }
};

// ...

const bool flush = all_of(hand.begin()+1,
                          hand.end(),
                          SameSuit(hand[0]));
```

For each card, count how many cards have the same rank.

```
class SameRank { // function obj comparing ranks
  Card referenceCard;
public:
  SameRank(const Card& c) : referenceCard(c) {}
  bool operator()(const Card& card) const {
    return referenceCard.rank == card.rank;
  }
};

// ....

int cardCounts[5];
for (int i = 0; i < 5; i++)
  cardCounts[i] = count_if(hand.begin(),
                           hand.end(),
                           SameRank(hand[i]));
sort(cardCounts,cardCounts+5);
```

Check for straight
(all five cards in sequence)

```
// one of each rank
const static int highCard[] = {1,1,1,1,1};
const bool allDifferent = equal(cardCounts,
                                cardCounts+5,
                                highCard);
const bool straight = allDifferent &&
     (hand[0].rank + 4 == hand[4].rank ||
      (hand[0].rank == ACE &&
       hand[1].rank == 10));
```

Find best Poker hand

```
const static int fourOfAKind[]  = {1,4,4,4,4}; // card counts
const static int fullHouse[]    = {2,2,3,3,3};
const static int threeOfAKind[] = {1,1,3,3,3};
const static int twoPair[]      = {1,2,2,2,2};
const static int onePair[]      = {1,1,1,2,2};

if (straight && flush) {
  // straight flush
} else if (equal(cardCounts,cardCounts+5, fourOfAKind)) {
  // four of a kind
} else if (equal(cardCounts,cardCounts+5, fullHouse)) {
  // full house
} else if (flush) {
  // regular flush
} else if (straight) {
  // regular straight
} else if (equal(cardCounts,cardCounts+5, threeOfAKind)) {
  // 3 of a kind
} else if (equal(cardCounts,cardCounts+5, twoPair)) {
  // two pair
} else if (equal(cardCounts,cardCounts+5, onePair)) {
  // one pair
} else {
  // hight card (ACE if hand[0] is ACE, else hand[4])
}
```

Algorithms are more awesome
when
we use *lambda functions*
(stay tuned)