问题

- 如何在计算器题目中加入索引？
- 输入:

[chen] 3+2; [chen] 5+7; [shen] 3+2; [shen] 2+6; q

chen

- 输出:

3+2=5

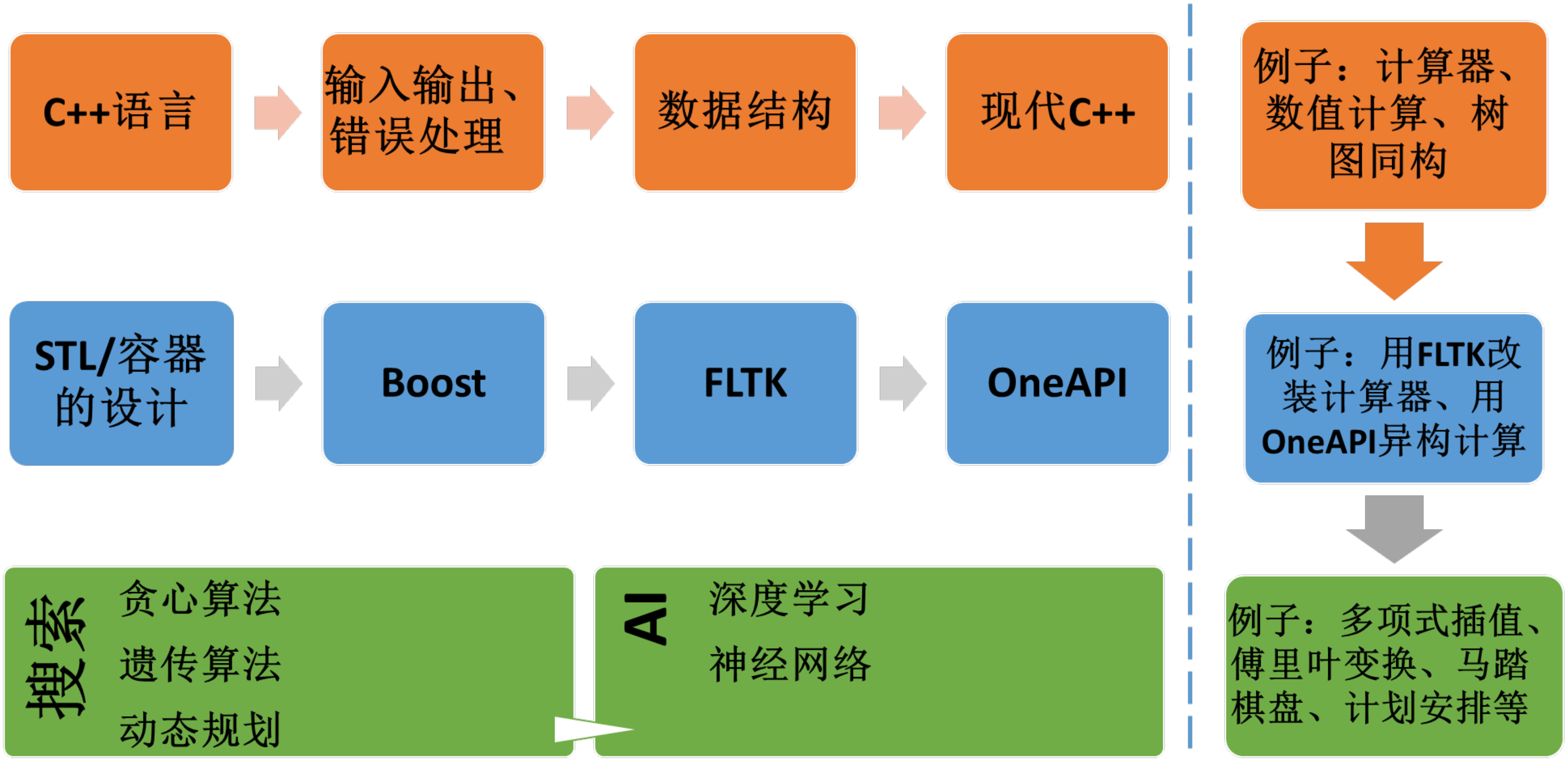
5+7=12

问题

- 用户进行输入，如何快速将输入转换成内部的vector<string>?
- 用户进行输入，如何快速将输入转换为输出？

问题求解与实践
——标准模板库(STL) (2)

主讲教师： 陈雨亭、沈艳艳


C++语言
输入输出、错误处理
数据结构
现代C++
例子：计算器、数值计算、树图同构
STL/容器的设计
Boost
FLTK
OneAPI
例子：用FLTK改装计算器、用OneAPI异构计算
搜索
贪心算法
遗传算法
动态规划
AI
深度学习
神经网络
例子：多项式插值、傅里叶变换、马踏棋盘、计划安排等

STL Containers and Iterators

Section 1

Introduction to STL and STL Containers

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 takes the element type as a parameter.
     - e.g., `vector, list, deque, set, stack, queue, etc.`
  2. Different kinds of **iterators** that can navigate through the containers.
  3. **Algorithms** that (via iterators) perform an interesting operating on a range of elements
     - e.g., `sort, random-shuffle, transform, find`

The C++ Standard Template Library (STL)

➢Design Philosophy

- Generic **containers** that take the element type as a parameter
  - Know (almost) nothing about the element type
    - Exception: ordered containers accepting elements to have `operator<`
  - Operations are (mostly) limited to add, remove, and retrieve
  - Define their own **iterators**

- Useful, efficient, and 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)

➢ Design Philosophy (con't)

- 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.

The C++ Standard Template Library (STL)

- 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 reusability.
    - 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.

The C++ Standard Template Library (STL)

➢ 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*>`

The C++ Standard Template Library (STL)

- Containers of Objects or Pointers?

```
Circle c (“red”);
vector<Figure> figList;
figList.emplace_back(c);
```

```
Circle c (“red”);
vector<Figure*> figList;
figList.emplace_back(c);
```

Objects:

- Copy operations could be expensive
- Two red circles
- Changes to one do not affect the other
- When `figList` dies, its copy of red circle is destroyed
- Risk of static slicing

Pointers:

- Allows for polymorphic containers
- When `figList` dies, only pointers are destroyed
- Client code must clean up referents of pointer elements

The C++ Standard Template Library (STL)

- Balloon Example

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

class Balloon {
   public:
      Balloon (string colour);
      Balloon (const Balloon& b); // copy ctor
      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
  - Sequence Containers: `vector, deque, list`
  - Container Adapters: `stack, queue, priority_queue`
  - Ordered Associative `Containers: [multi]set, [multi]map`

- C++11 adds:
  - Addition of `emplace{_front, _back}`
  - Sequence Containers: `array, forward_list`
  - Unordered Associative Containers: `unordered_[multi]set, unordered_[multi]map`

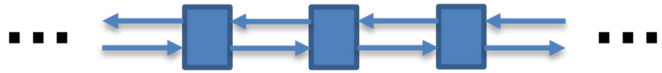
- C++14 adds:
  - Non-member `cbegin, cend, rbegin, rend, crbegin, crend.`

STL Containers – Conceptual View

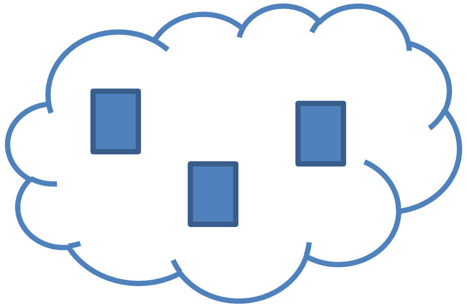
- Sequence Containers
  - Vector

  - Deque

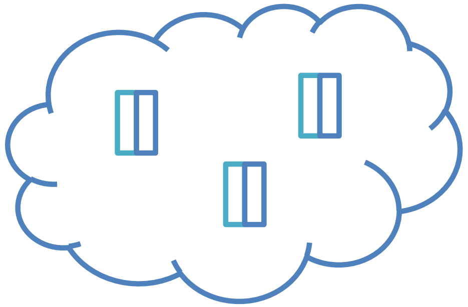
  - List



- Sequence Containers
  - [multi]set



  - [multi]map



STL Containers

	STL containers	*Some* useful operations
	all containers	size, empty, emplace, erase
Sequence	vector<T>	[], at, clear, insert, back, {emplace,push,pop}_back
Sequence	deque<T>	[], at, emplace{,_front,_back}, insert, {,push_,pop_}back, {,push_,pop_}front
Sequence	list<T>	insert, emplace, merge, reverse,splice, {,emplace_,push_,pop_}{back,front}, sort
Sequence	array<T>	[], at, front, back, max_size
Sequence	forward_list<T>	assign, front, max_size, resize, clear, {insert,erase,emplace}_after, {push,pop,emplace}_front
Associative	set<T>, multiset<T>	find, count, insert, clear, emplace, erase, {lower,upper}_bound
Associative	map<T1,T2>, multimap<T1,T2>	[]*, at*, find, count, clear, insert, emplace, erase, {lower,upper}_bound
Unordered Associative	unordered_set<T>, unordered_multiset<T>	find, count, insert, clear, emplace, erase, {lower,upper}_bound
Unordered Associative	unordered_map<T1,T2>, unordered_multimap<T1,T2>	[]*, at*, find, count, clear, insert, emplace, erase, hash_function
Container Adaptors	stack	top, push, pop, swap
Container Adaptors	queue	front, back, push, pop
Container Adaptors	priority_queue	top, push, pop, swap
Other	bitset (N bits)	[], count, any, all, none, set, reset, flip

Red means "there's also a stand-alone algorithm of this name"

Can't iterate over stack, queue, priority_queue.

* Not on multimap

Section 2

STL Sequence Containers

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 :
  - Some access methods
    - `vector` and `deque` allow random access to elements (via `[]` / `at()`), whereas `list` allows only sequential access (via iterators)
    - deque allows `push_back` and `push_front` (as well as `pop_front` and `pop_back`)
  - Performance
    - `vector` and `deque` are optimized for (random access) retrieval, whereas `list` is optimized for (positional) insertion / deletion

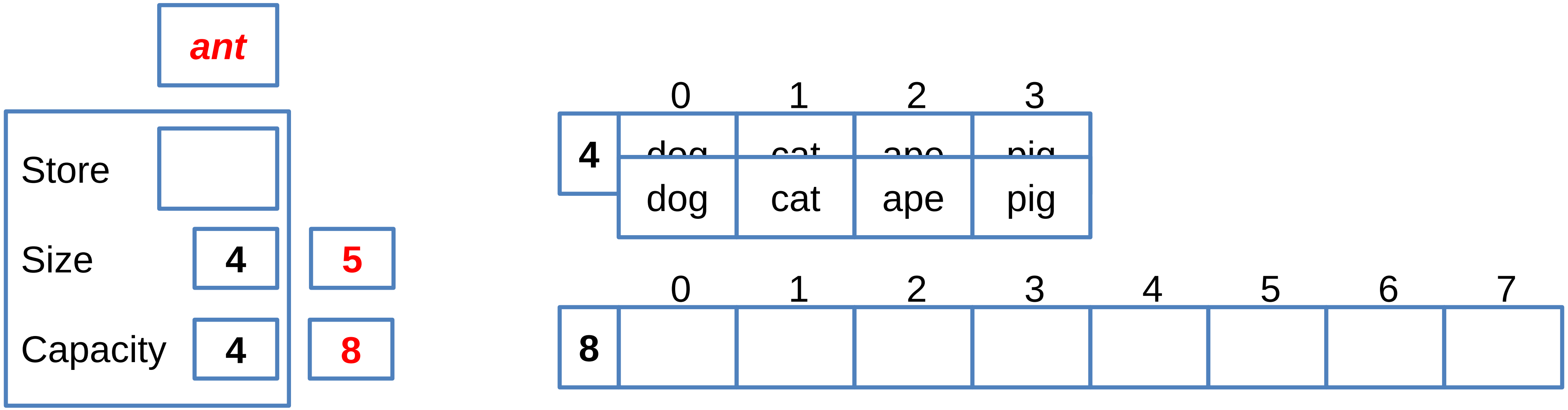
Sequence Containers: vector<T>

- Can think of it 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 Method	Complexity	API Support
Random Access	O(1)	`operator[]` or `at()`
Append / Delete Last Element	O(1) / O(1)	`push_back` / `pop_back`
Append / Delete First Element	O(N)	Not supported as API call
Random Insert / Delete	O(N)	`insert` / `erase`

Sequence Containers: vector<T>

- (Likely) Vector Implementation
  - And the **<u>Doubling Policy</u>** Memory Allocation



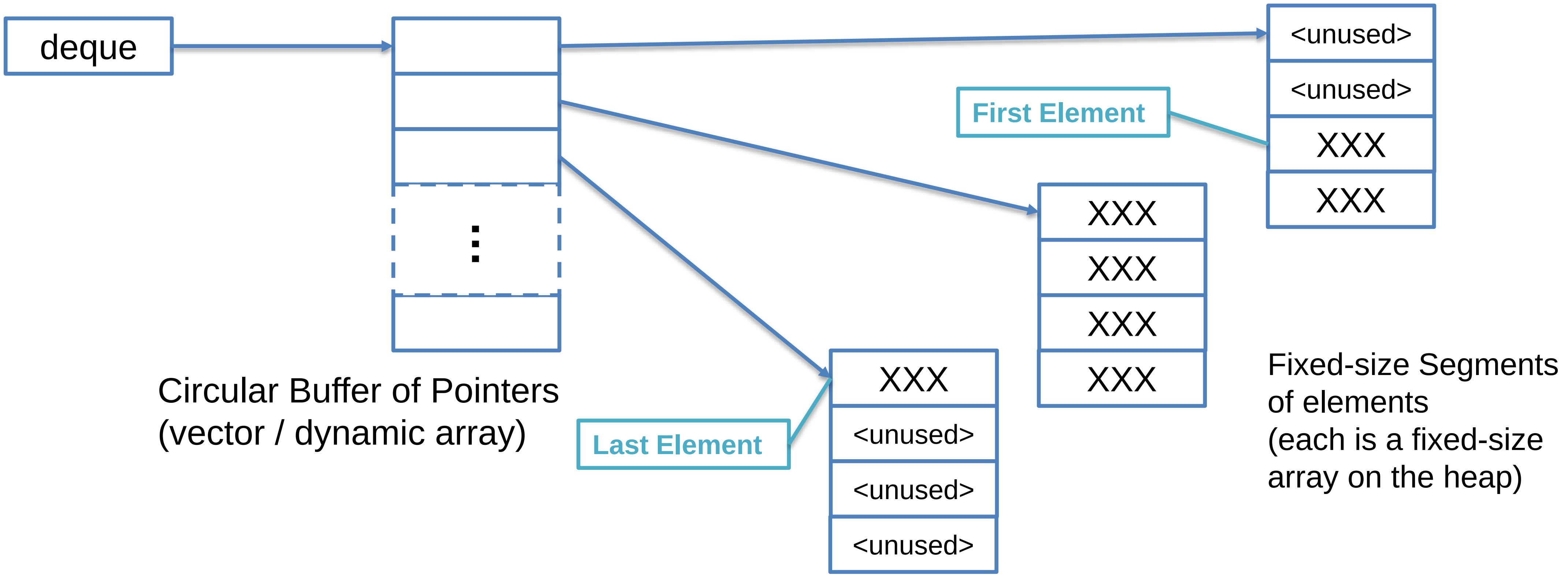
Sequence Containers: deque<T>

- "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
  - As a result, pointer arithmetic won't work
  - Operator[] and at() are overloaded to work correctly

Access Method	Complexity	API Support
Random Access	O(1)	`operator[]` or `at()`
Append / Delete Last Element	O(1) / O(1)	`push_back` / `pop_back`
Append / Delete First Element	O(1) / O(1)	`push_front` / `pop_front`
Random Insert / Delete	O(N)	`insert` / `erase`

Sequence Containers: deque<T>

- deque implementation – Indexed and Segmented Circular Buffer



Sequence Containers: vector vs. deque

- So, in real life, should you use `vector` or `deque`?
  - If you need to insert from just one end in FILO fashion, use `vector`.
  - If you need to insert from both ends, 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 (due to multi-level dereferencing through circular pointer buffer for `deque`)
- Reallocations
  - Take longer with a `vector`.
  - `vector` invalidates external references to elements, but not so with a `deque`.
  - `vector` copies **elements** (which may be objects), whereas `deque` copies **only pointers**.

Sequence Containers

➢ Integrity of External References

```
// Vector Implementation
#include <vector>
#include <deque>
using namespace std;

int main(int argc, char* argv[]) {
   cout << “\nWith a vector:” << endl;
   vector<int> v;
   v.push_back(4); v.push_back(3);
   v.push_back(37); v.push_back(15);
   int* p = &v.back();
   cout << *p << “ ” << v.at(3) << “ ”   //must be same
        << p << “ ” << &v.at(3) << endl; //must be same
   v.push_back(99);
   cout << *p << “ ” << v.at(3) << “ ”   //may be different*
        << p << “ ” << &v.at(3) << endl; //probably different
```

Because p is no longer pointer to v[3], but the old value of 15 may still be there.

```
// Deque Implementation

   cout << “\nWith a deque:” << endl;
   deque<int> v;
   d.push_back(4); d.push_back(3);
   d.push_back(37); d.push_back(15);
   int* 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
}
```

```
// Output below, YMMV but comments above will hold
With a vector:
15 15  0x7ff87bc039cc 0x7ff87bc039cc
15 15  0x7ff87bc039cc 0x7ff87bc039ec
With a deque:
15 15  0x7ff87c00220c 0x7ff87c00220c
15 15  0x7ff87c00220c 0x7ff87c00220c
```

Sequence Containers: list<T>

- Implemented as a (plain old) doubly-linked list (PODLL), and designed for fast insertion and deletion.
- Supports only sequential access to elements via iterators.
  - No random access via indexing `operator[]` or `at()`.

Access Method	Complexity	API Support
Random Access	O(N)	Not supported as API call
Append / Delete Last Element	O(1)	`push_back` / `pop_back`
Append / Delete First Element	O(1)	`push_front` / `pop_front`
Random Insert / Delete	O(1) (once arrived at the element) O(N) (to get there)	`insert` / `erase`

Sequence Containers: std::array (C++11)

- A very thin wrapper around a C++ array, to make it a little more like a fixed-size `vector`
- `std::array` vs. C++ 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 specified by the programmer upon instantiation.
- `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

Sequence Containers: std::forward_list (C++11)

- Basically, a plain-old single-linked list.
- `std::forward_list` vs. `std::list`
  - More space-efficient, and insertion/deletion operations are slightly faster
  - No immediate access to the end of the list
    - i.e. No `push_back()`, `back()`
  - No ability to iterate backwards
  - No `size()` method

Section 3

STL Container Adapters

STL Container Adapters

- Usually a trivial wrapping of a sequence container in order 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(default), list`
  - **`queue`**`: deque(default), list`
  - **`priority_queue`**`:   vector(default), deque`
- Implemented using the DP: Adapter Pattern
  - Define the interface you really want (e.g., for `stack`, we want `push()` and `pop()`)
  - Instantiate (don’t inherit) a private data-member object from the “workhorse” container class that will do the actual heavy-lifting (e.g., `vector`).
  - Define operations by delegating to operations from the workhorse class.

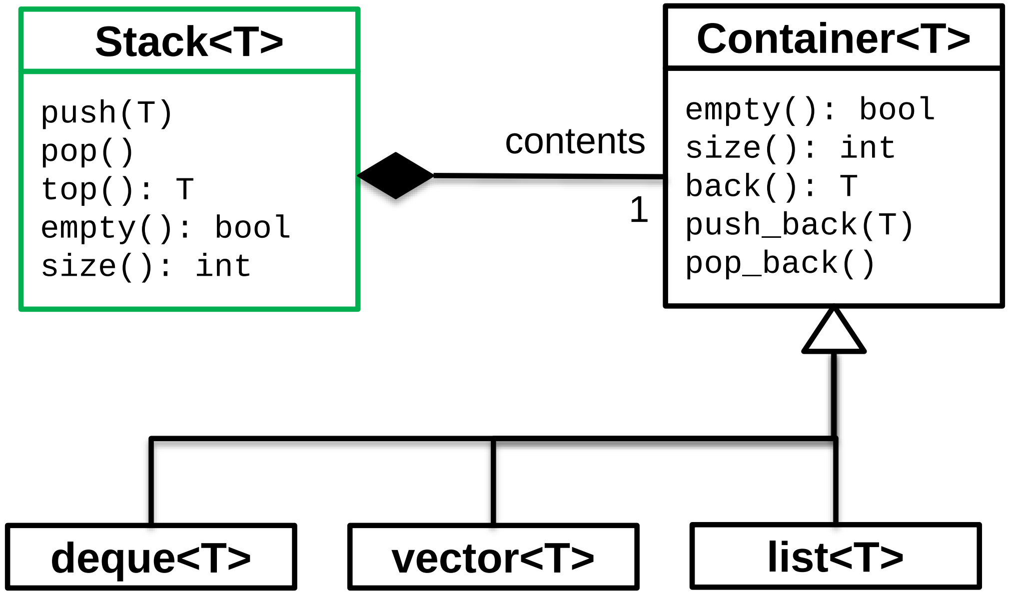
STL Container Adapters – DP: Adaptor

- STL Stack Implementation (Simplified Version)

```
template <class T, Container = deque<T> >
class stack{
    public:
         bool empty() const;
         int size() const;
         T& top const;
         void push (const T& val);
         void pop();
    private:
         // this container is the adapter
         Container contents_;
};

void stack::push(const T& val) {
    contents_.push_back(val);
}

void stack::pop() {
    contents_.pop_back();
}
```



STL Container Adapters

- "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 ***adaption***, rather than inheritance.
    - Interface can be exactly what you want, not constrained by inheritance.

Source: BlackShellMedia@Twitter

STL Container Adapters

- 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`.

STL Container Adapters

- Inheriting from an STL Container – Legal, but probably not a good idea…

```
// legal, but is it a good idea?
class CardPile : public vector<Card*> {
    public:
        // Constructor and Destructor
        CardPile();
        virtual ~CardPile();

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

        // Mutator
        void add(Card* card);
        void add(CardPile& otherPile);
        void remove(Card* card);
        void shuffle();
};
```

STL Container Adapters

- Traditional STL Adaptation (How STL Containers are intended to be used)
  - As if `CardPile` has a built-in adapter to the STL container

```
class CardPile {
   public:
      // Constructor and Destructor
      CardPile();
      virtual ~CardPile();

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

      // Mutator
      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 );
```

```
      // Wrapped container methods and types
      using iterator = std::vector<Card*>::iterator;
      using const_iterator = std::vector<Card*>::const_iterator;
      CardPile::iterator begin();
      CardPile::const_iterator begin() const;
      CardPile::iterator end();
      CardPile::const_iterator end() const;
      int size() const;
      Card* at(int i) const;
      void pop_back();
      Card* back() const;
      bool empty() const;
   private:
      std::vector<Card*> pile;
};
// Example of function wrapper
void CardPile::add( CardPile& otherPile ){
   for( auto card : otherPile ) pile.emplace_back( card );
   otherPile.pile.clear();
} //CardPile::add
```

STL Container Adapters – Private Inheritance

➢ Public Inheritance

```
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 Inheritance

```
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*`

STL Container Adapters – Private Inheritance

- Private inheritance is used to allow reuse of a base class' ***implementation*** without having to support the base class' interface.
- All of the inherited `public` (and `protected`) members of the base class are `private` in the child class, and can be used to implement child class methods. However, they are NOT exported to the public.
- We can selectively make some of the methods of the base class visible to the client code using the keyword `using`, as in

  - `using Figure::getColour;`
  - Known as ***Promotion***

STL Container Adapters – Private Inheritance

- Private Inheritance of STL Container

```
class CardPile : private std::vector<Card*>
{

  public:
    // Constructor and Destructor
    CardPile();
    virtual ~CardPile();

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

    // Mutator
    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 );
```

```
    // "Promoted" container methods and types
    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;

};
```

STL Container Adapters – Private Inheritance

- 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 and inherited function, the client code will get that version, since they cannot 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' interface where applicable.

Section 4

STL Associative Containers

STL Associative Containers

➢ Ordered Associative Containers

`[multi]map, [multi]set`

- The ordering of the elements is based on a ***key*** value – a piece of the element (e.g., employee record sorted by SIN number)
  - Not by the order of insertion
- Implemented using a kind of **binary search tree** => lookup is O(log N)
- Can iterate through container elements “in order”

➢ Unordered Associative Containers

`unordered_[multi]map, unordered_[multi]set`

- No ordering assumed among the elements
- Implemented using **hash tables** => lookup is O(1)
- Can iterate through container elements, but no particular ordering is assumed.

STL Associative Containers – set<T>

- 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 “reasonable” `operator<` defined or provided an ordering ***functor*** to the set constructor.
- Sets do not allow duplicate elements
  - If you are trying to `insert` an element that is already present in the `set`, the `set` is unchanged.
  - Result 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

STL Associative Containers – set<T>

```
// Example with user-defined class and operator<
#include <algorithm>
#include <set>
#include <iostream>
#include <string>
using namespace std;

class Student {
   public:
      Student(string name, int sNum, double gpa);
      string getName() const;
      int getSNum() const;
      double getGPA() const;
   private:
      string name_;
      int sNum_;
      double gpa_;
};

Student::Student(string name, int sNum, double gpa) :
        name_(name), sNum_(sNum), gpa_(gpa) {}
string Student::getName() const { return name_; }
int Student::getSNum() const { return sNum_; }
double Student::getGPA() const { return gpa_; }

bool operator== (const Student& s1, const Student& s2) {
   return (s1.getSNum() == s2.getSNum()) &&
          (s1.getName() == s2.getName()) &&
          (s1.getGPA() == s2.getGPA())
}
```

```
bool operator< (const Student& s1, const Student& s2) {
   return s1.getSNum() < s2.getSNum();
}

ostream& operator<< (ostream& os, const Student& s) {
   os << s.getName() << " " << s.getSNum()
      << " " << s.getGPA();
   return os;
}

int main{
   // Peter and Mary have the same SNum
   Student* pJohn = new Student("John Smith", 666, 3.7);
   Student* pMary = new Student("Mary Jones", 345, 3.4);
   Student* pPeter = new Student("Peter Piper", 345, 3.1);

   set<Student> s;
   s.insert(*pJohn);
   s.insert(*pMary);
   s.insert(*pPeter);

   // Will print in numeric order of sNum
   for (auto iter = s.begin(); iter != s.end(); iter++){
      cout << *iter << endl;
   }
   if ( s.find(*pPeter) != s.end() )
      cout << "Found it with set's find()!" << endl;
   if ( find( s.begin(), s.end(), *pPeter ) != s.end() )
      cout << "Found it with STL algorithm find()" << endl;
}
```

STL Associative Containers – set<T>

- Equivalence vs. Equality
  - Equivalence
    - The container search methods (e.g., `find`, `count`, `lower_bound`, …) will use the following test for equality for elements in ordered associative containers **<u>even if you have your own definition of `operator==`</u>**.

```
if( !( a < b ) && !( b < a ) )
```

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

STL Associative Containers – map<T>

- A set is a collection of (unique) values
  - Typical declaration:

    ```
    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 "reasonable" `operator<` defined or provided an ordering ***functor*** to the map constructor.

  - `T2` is the ***value field type***; it can be anything that is copyable and assignable.

STL Associative Containers – map<T>

➢ Querying Map for Element

- Intuitive method (look up via indexing) will *insert the key if it is not already present*:

```
if( works[ “bach” ] == 0 )
        // bach not present
```

- Alternatively, can use map’s **find()** operation to return an iterator pointer the queried key/value pair

```
map<string, int>::iterator it;

it = words.find( “bach” );
if( it == words.end( ) )
        // bach not present
```

end( ) is an iterator value that points beyond the last element in a collection

STL Associative Containers – map<T>

- Example - Dictionary

```
#include <iostream>
#include <map>
#include <cassert>
#include <string>
using namespace std;

// Example adapted from Josuttis
int main() {
   map<string, string> dict;

   dict["car"] = "vioture";
   dict["hello"] = "bonjour";
   dict["apple"] = "pomme";

   cout << "Printing simple dictionary" << endl;

   for( auto it : dict ){
      cout << it.first << ":\t" << it.second << endl;
   }
}
```

```
// Example adapted from Josuttis
   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 << "\nPrinting all defs of "\car"\" << endl;

   for(multimap<string, string>::const_iterator
        it = mdict.lower_bound("car");
        it != mdict.upper_bound("car"); it++){

      cout << (*it).first << ": " <<
           << (*it).second << endl;
   }
```

STL Associative Containers

- `[multi]set` and `[multi]map` are usually implemented as a **red-black tree**
  - 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

STL Unsorted Associative Containers (C++11)

- `unordered_[multi]set` and `unordered_[multi]map`

- They are pretty much the same as the sorted version, except:
  - They are not sorted. (Duh)
  - They are 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

Section 5

STL Iterators

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--`)

STL Iterators

- STL Containers provider their own Iterators
  - If `c` is a `vector`, `deque`, `list`, `set`, `map`, etc., then
    - `c.begin()` / `c.cbegin()` returns a pointer to the first element
    - `c.end()` / `c.cend()` returns a pointer to just beyond the last element
    - `operator++` is defined to advance to the next element
  - Example

Type

```
vector<string>::const_iterator
map<int,string>::iterator
list<Figure*>::reverse_iterator
```

Pointer to the First Element

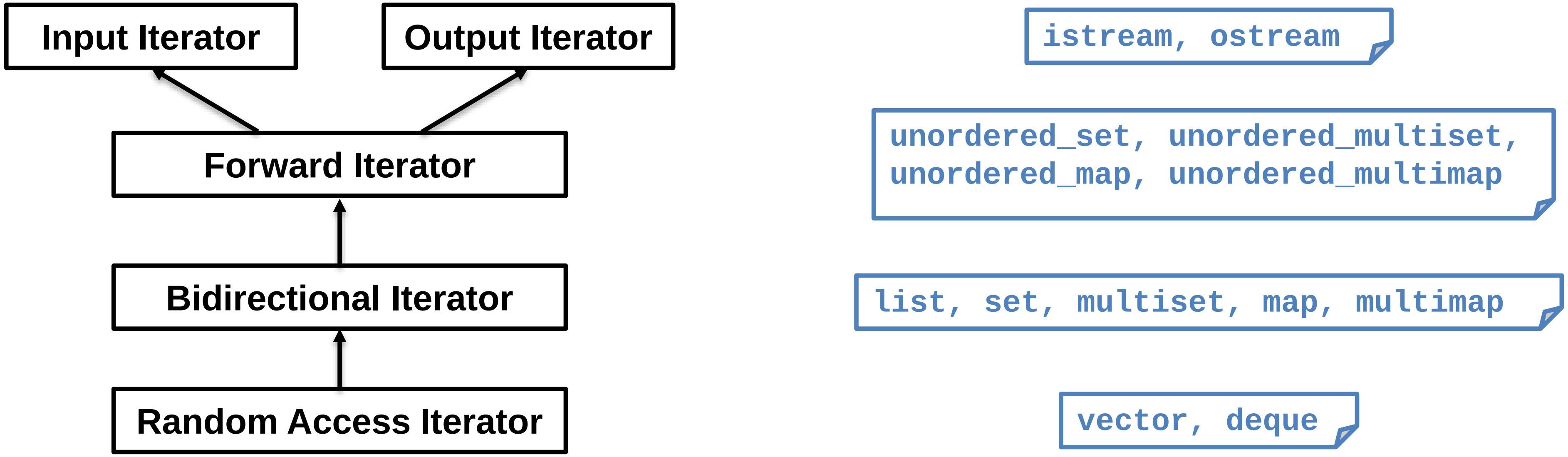
```
vi = v.begin();
mi = mymap.begin();
li = scene.rbegin();
```

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

STL Iterators

➢ Kinds of Iterators

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



- Why should you care?

STL Iterators – Input and Output Iterators

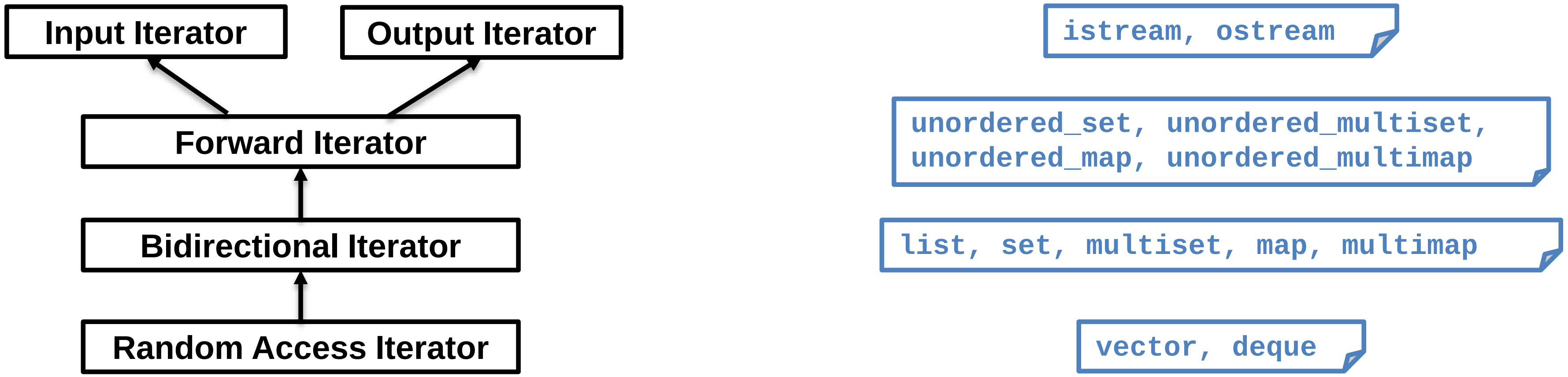
- Input Iterators
  - **Read-only** iterators where each iterated location may be **read only once**.
- Output Iterators
  - **Write-only** iterators where each iterated location may be **written only once**.
- Operator
  - ++, * (can be const), ==, != (for comparison iterators)

- Mostly used to iterate over streams

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

STL 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 (+, -, +=, -=)



STL Iterators – Inserters

- Inserters (Insert Interator) is used to insert elements into its container:
  - **back_inserter** uses container's push_back()
  - **front_inserter** uses container's push_front()
  - **inserter** uses container's insert()

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

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

Summary

Summary

- 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 algorithm with legacy C data structures