The C++ Standard Template Library (STL)

CS247 guest lecture, Mike Godfrey

July 8 and 10, 2014

Lecture #1

The C++ STL

- The C++ Standard Template Library (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`, `map`, `set`, `stack`, `queue`, …

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

3. *Algorithms* that take an iterator, and perform an interesting operation on the elements in that range
   e.g., `sort`, `random_shuffle`, `next_permutation`

A very brief social history of the STL

- The C++ Standard Library == STL + other stuff (e.g., `auto_ptr`)

- The STL is primarily the design of one person, Alexander Stepanov, tho many have worked on it subsequently
  - Stepanov was a colleague of Bjarne Stroustrup at AT&T Research in the 1980s; later he moved to HP and then SGI, taking the STL "ownership" with him.
  - Stroustrup liked the STL and lobbied for its inclusion in the 1998 ANSI standardization of C++.

- The requirements on an implementation of the STL (as imposed by the ANSI standard) are tough; any correct and (compliantly) efficient implementation is very complex.
  - This is why you get reams of incomprehensible error messages if you make a mistake using a vector, and why you don't want to implement it yourself

Design philosophy of the STL

- A collection of useful, efficient, type-safe, generic containers
  - A container has (almost) no understanding of the element type
    - Exception: ordered containers *can-be-sorted* via some `operator<`
  - Each container should define its own iterators
  - Container methods use static dispatch (i.e., are not declared `virtual`)
- A collection of useful, efficient, generic algorithms that operate on iterators
  - "Generic" => algorithm knows nothing about the structure it's operating on, apart from the fact that it can be traversed by an iterator, and knows (almost) nothing about the elements in the structure
  - Define container methods only when the generic algorithms are unsuitable or much less efficient

Some facts about the STL

- The STL assumes little about the contained elements.
  - The element type must allow copying/assignment
  - This is legal: `vector<vector<string>> v;`
- For *ordered* collections, 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; objects are *copied* to/from containers more often than you might realize.

  e.g., when a vector doubles in size; if you have "big" objects, you might want to use pointers instead
- The methods/algorithms are highly efficient at making copies of (sub)structures that contain the elts; it's unlikely you'll do better yourself.

Why is there no inheritance in the STL?

- Stepanov thinks OOP (i.e., inheritance) is wrong; his "religion" is *generic programming*. For the sake of "true" genericity, templates are used to achieve a more flexible ("ad hoc") kind of polymorphism.
  - "If it looks like a Duck, it's a Duck." vs. "If it inherits from Duck, it's a Duck."
    - This is called "Duck Typing"; see Wikipedia
  - "I think that object oriented-ness is almost as much of a hoax as Artificial Intelligence." – Stepanov
- The algorithms are designed so that (almost) any algorithm can be used with any STL container, or any other data structure that supports the idea of iterators
- The containers are just different enough that code reuse isn't really practical. No container methods are virtual, in the interests of efficiency.
  - So no container destructors are declared as `virtual` either! Leak danger!

Review: Polymorphic containers

- Suppose we want to model a graphical `Scene` that has an ordered list of `Figures` (`Rectangles`, `Circles`, and maybe other concrete classes we haven't implemented yet)
  - Recall `Figure` is an abstract base class (ABC) so can't be instantiated
  - The `Scene` will have a textual `Caption`, plus the list of `Figures`, which we can implement using a `vector`
- To draw the scene, we print the `Caption` (somehow) then we draw the list of `Figures` in order
  - The question is: What should the list look like?
    1. `vector<Figure>`
    2. `vector<Figure&>`
    3. `vector<Figure*>`

Review: Polymorphic containers

- `vector<Figure>` won't work
  - This would be a `vector` of `Figure` objects, and *no other kind* … but `Figure` is an ABC and can't be instantiated (compiler will complain)
  - If `Figure` were not an ABC & we wanted to store only `Figure`s then this would work … but then `v` wouldn't be a polymorphic container

```
Circle c1 ("cyan", 0, 0, 5);
v.push_back(c1);  // "static slicing": this calls
                  // the Figure copy cxr, not Circle!
```

- `vector<Figure&>` won't work
  - `vector`s store "objects" (meaning instance of a data type, such as classes and basic types like `int`); a reference is not an object, it's an another name for an existing object

Review: Polymorphic containers

- `vector<Figure*>` will work!
  - This stores a list of ptrs to `Figure`, meaning they can point to any instance of a subclass of `Figure` (e.g., `Circle`, `Rectangle`, etc.)
  - The actual objects will be somewhere else, likely on the heap, and as always we need to have an understanding of who is responsible for deleting them eventually.

```
class Scene {
    public:
        Scene();
        ~Scene();
        string getCaption () const;
        void setCaption (string caption) ;
        void addFigure(Figure* f);
        void draw() const;
    private:
        vector<Figure*> v;
        string caption;
};
Scene::Scene() {}
Scene::~Scene(){           // Assume we own these Figures
    while (!v.empty()) {
        Figure* f = v.back();
        v.pop_back();
        delete f;     // Works for Circles, Rectangles, …
    }
}
```

```
string Scene::getCaption () const {
    return caption;
}
void Scene::setCaption(string caption) {
    this->caption = caption;
}
void Scene::addFigure (Figure * f) {
    // Don't bother to add NULL ptrs
    if (NULL != f) {
        v.push_back(f);  // Works for Circles, Rectangles, …
    }
}
void Scene::draw() const {
    cout << "Caption: \"" << caption << "\"\n";
    for (int i=(int) v.size()-1; i>= 0; i--) {
        Figure *f = v.at(i);
        f->draw();       // Works for Circles, Rectangles, …
    }
}
```

Container of objects or ptrs?

Elements are copied to/from containers -- more often than you would guess

```
Circle c ("red");
vector<Figure>  figList;
figList.push_back(c);
```

Objects:

- copy operations could be expensive
- two red circles
- changes to one do not affect the other
- when figList dies, it will destroy its copy of red circle
- risk of static slicing

```
Circle c ("red");
vector<Figure*>  figList;
figList.push_back(&c);
```

Pointers:

- allows for polymorphic containers
- when figList dies, only pointers are destroyed
- client code must cleanup referents of pointer elements

```
#include <iostream>
#include <string>
#include <vector>
using namespace std;
class Balloon {
    public :
        Balloon (string colour);
        Balloon (const Balloon& b);  // 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 << " balloon is born" << endl;
}
Balloon::~Balloon() {
    cout << colour << " balloon dies" << endl;
}
```

```
void Balloon::speak() const {
    cout << "I'm a " << colour << " balloon" << 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);

}
```

```
red balloon is born
red balloon is born
green balloon is born
green balloon is born
red balloon is born
red balloon dies
blue balloon is born
blue balloon is born
green balloon is born
red balloon is born
green balloon dies
red balloon dies
blue balloon dies
green balloon dies
red balloon dies
blue balloon dies
green balloon dies
red balloon dies
```

The C++ Standard Library

C++ Standard Library == STL + some other stuff

- What's in the official C++ Standard Library is defined by the various language standards
  - C++98, 03, 11, 14 (in discussion)
- We are going to concentrate on the C++03 standard
  - The STL parts were actually defined in C++98 and were not changed for C++03
  - Compilers were slow to provide good implementations of the STL
- We will mention a few things of particular interest that are new in the C++11 standard

The C++ Standard Template Library

I. Generic *containers* that take the element type as a parameter
   e.g., `vector`, `list`, `deque`, `map`, `set` plus `stack`, `queue`, etc.

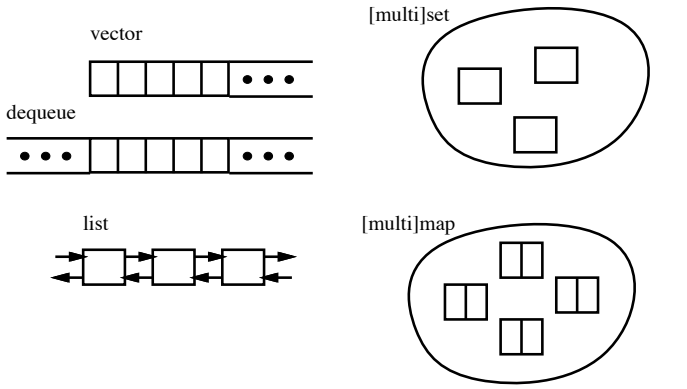
II. Kinds of *iterators* that can navigate through the containers

III. *Algorithms* that take an iterator, and perform an interesting operation on the elements in that range
   e.g., `sort`, `random_shuffle`, `next_permutation`

STL containers

- C++98/03 defines three main data container categories:
  1. Sequence containers: `vector`, `deque`, `list`
  2. Container adapters: `stack`, `queue`, `priority_queue`
  3. Ordered associative containers: `[multi]set`, `[multi]map`
- C++11 adds:
  1. Sequence containers: `array`, `forward_list`
  2. [nothing new]
  3. [nothing new]
  4. *Unordered* associative containers: `unordered_[multi]set`, `unordered_[multi]map`

STL containers: Conceptual view



Sequence containers *Ordered associative containers*

The main STL containers (apart from `vector`)

- `deque` is a double-ended queue, similar to `vector`
  - Allows fast direct access, plus easy growth at either end
  - As with vector, slow to insert/delete in middle
- `list` is a plain-old doubly-linked list
  - Support forward or backward iterators but no direct access
  - Fast to insert/delete in middle (once you've found the right spot)
- `set` is a mathematical set
  - `multiset`s allows an element to occur more than once
- `map` is like a clever array where index type can be almost anything
  - Also called "associative array"
  - `multimap`s allow the same key to map to multiple values

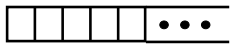
STL (C++98/03) container	*Some* useful operations
all containers	`size, empty, insert, erase`
`vector<T>`	`[], at, back, push_back, pop_back`
`deque<T>`	`[], at, back, push_back, pop_back, front, push_front, pop_front`
`list<T>`	`back, push_back, pop_back front, push_front, pop_front, sort, merge, reverse, splice`
`set<T>, multiset<T>`	`find`
`map<T1, T2>, multimap<T1, T2>`	`[], at, find, count`
Other ADTs in the STL	`stack, queue, priority_queue, bitset`

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

1. Sequence containers

- There is a total ordering of contiguous values (i.e., no gaps) on elements based on the *order* in which you added them into the container
  i.e., ordering is *not* based on an some intrinsic/key value of the element
  e.g., `vector`, `deque`, `list`

- 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
     - `list` is must be a doubly-linked list, so deleting an arbitrary elt is O(n)



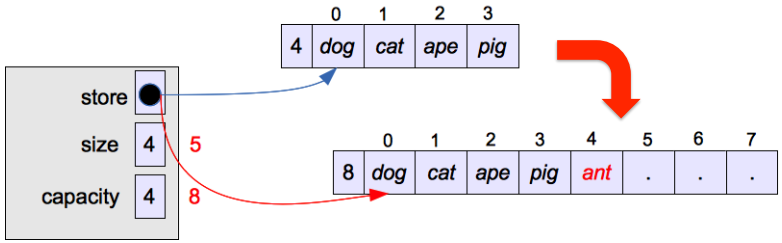
vector

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

- *Vector elements must stored contiguously* according to the C++ standard, so ptr arithmetic will work and O(1) random access is guaranteed
  - So we pretty much have to use a C-style array as our workhorse underneath

- Append is *amortized constant time*, sometimes have to copy N elements

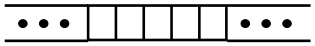
Access kind	Complexity	API support
random access	O(1)	`operator[]` or `at()`
append/delete last	O(1)*/O(1)	`push_back/pop_back`
prepend/delete first	O(N)	*not supported as API call*
random insert/delete	O(N)	`insert/erase`



Implementing a `vector`

- We know this is done using a dynamically-allocated heap-based array
  - A vector object probably has at least three fields: a ptr to the heap-based array plus `int`s that track the `size` and `capacity`
  - Use obvious definitions of `operator[]` and `at()` based on address calculation (start of array + offset)
  - Insertion onto the end is *amortized constant time*, as we discussed
- Calling `push_back` when `size==capacity` forces a *re-allocation*
  - Grab a new heap-based array of twice the old size (usually), and copy the elements over
  - Obviously, this invalidates any external refs to individual vector elements, including iterators [show example]
  - Note that this is an O(N) *object* copy, not ptr copy; this can be expensive
  - Can we do better?

`deque`



***Conceptual** model*

- == "double-ended queue"; similar to `vector`s, but allow fast insertion/deletion at beginning and end
- Random access "fast", but *no guarantee elements are stored contiguously*
  - … so pointer arithmetic won't work
  - `operator[]` and `at()` must be overloaded to work correctly

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(1)*/O(1)	`push_front/pop_front`
random insert/delete	O(N)	`insert/erase`

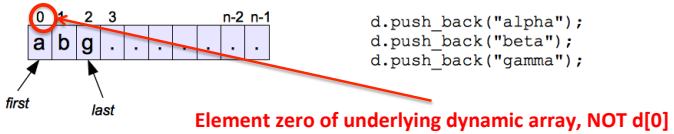
```
#include <iostream>
#include <string>
#include <deque>
using namespace std;

void print (deque<string> d) {
  cout << "Printing d:\n";
  for (int i=0;i<d.size();i++){
     cout << "d["<< i
          << "] = \""
          << d[i] << "\""
          << endl;
  }
  cout << endl;
}
```

```
int main (int argc, char*
   argv[]) {
     deque<string> d;
     d.push_back("alpha");
     d.push_back("beta");
     d.push_back("gamma");
     d.push_front("upsilon");
     d.push_front("zeta");
     print(d);
} // Output below
// Printing v:
// d[0] = "zeta"
// d[1] = "upsilon"
// d[2] = "alpha"
// d[3] = "beta"
// d[4] = "gamma"
```

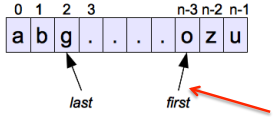
Implementing a `deque`: A circular buffer of objects

- Idea #1
  - Maintain a dynamic array of objects, like we did with `vector`
  - When we need more storage, re-allocate: get a bigger dynamic array from the heap, copy the objects over to the new space, and `delete` the old dynamic array
  - Clever part: Treat that storage like a *circular buffer* so you can add to back or front in constant time
- (This idea works, but we will ultimately reject it)



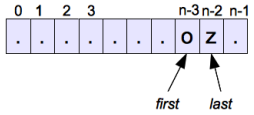

```
d.push_back("alpha");
d.push_back("beta");
d.push_back("gamma");
```

Element zero of underlying dynamic array, NOT d[0]




```
d.push_front("upsilon");
d.push_front("zeta");
d.push_front("omega");
```

"first" always points to d[0]




```
d.pop_back();
d.pop_back();
d.pop_back();
d.pop_back();
```

```
for (unsigned int i=0; i<d.size(); i++) {
    cout << "d["<< i << "] = \"" << d[i] << "\"" << endl;
}
// Output:
// d[0] = "omega"
// d[1] = "zeta"
```

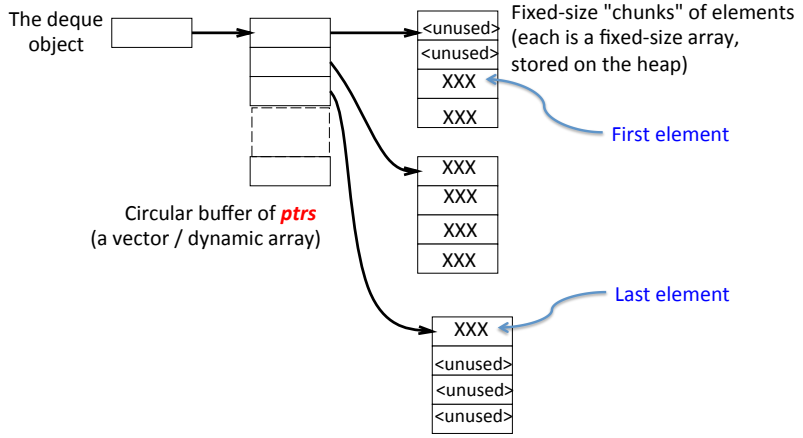
Implementing a `deque`: A circular buffer of *objects*

- This means we need to provide specialized definitions of `operator[]` and `at()` using offsets and modular arithmetic when you start to add elements at the front
  - However, you just adding or subtracting a constant `int` or two, which is fast to perform
  - This means that insertion *at either end* is also amortized constant time in terms of # of object copies performed
    - It's O(1) unless you need to reallocate, in which case it is O(N)
    - [And that's only if we use this implementation, which we won't]

Implementing a `deque`: A circular buffer of *objects*

- This implementation has the disadvantage that (like with `vector`), any insertion potentially invalidates external references to elements and any active iterators
  - So it "works", but this isn't what is normally done
  - Also, it requires O(N) *object copies* on reallocation
- The C++ standard requires that insertion at the beginning/end of a `deque` be *constant time* in terms of the number of times a copy cxr is called (i.e., how many *object* copies are created)
  - *Amortized* constant time is not acceptable, according to the standard
  - So we need another way

Implementing a `deque`: A circular buffer of *pointers*



Implementing a `deque`: A circular buffer of *pointers*

"All hard problems in [software engineering] can be solved by adding a level of indirection."

— Prof. David Wheeler

- The approach that *is* normally used for a `deque` involves a set of fixed-length arrays ("chunks"), plus a master dynamic array of ptrs (a circular buffer) that point to the chunks
  - After the first insertion, there is one chunk with one element filled
  - Subsequent adds at either end may require another chunk to be allocated and linked to by the master circular buffer

Implementing a `deque`: A circular buffer of *pointers*

- Let's assume N elements, and chunk size of K
  - Then circular buffer has about N/K elements
- We need to allocate new chunks periodically (every K insertions at front/back)
  - This has a mild overhead cost, but involves no *object* copying (apart from the single new object being added at the front/back)
- If the circular buffer is full, then we need to reallocate
  - But we don't reallocated the chunks, just the circular buffer of ptrs by copying (only) the pointers over to a new, bigger buffer
  - This means that any external references to elements are still valid!

Implementing a `deque`: A circular buffer of *pointers*

- How often does reallocation (of the circular ptr buffer) happen?
  - As with `vector` , only when we exceed the capacity of (about) N elements
- How much copying is involved in a reallocation?
  - We copy N/K *pointers* (size of circular buffer)
  - With a `deque` , we are copying only N/K ptrs which is fast; with a `vector`, we may be copying N large objects (not ptrs)
  - The C++ standard requires that insertion at the beginning/end of a `deque` be constant time in terms of the number of times a copy cxr is called (i.e., how many object copies are created, *not ptr copies*)
    - So the occasional N/K ptr copies don't "count" against the complexity according to the C++ standard, as only one object is copied into a "chunk"

Implementing a `deque`: A circular buffer of *pointers*

- How well does this work?
  - `operator[]`, `at()`, and iterator impls need to be a bit clever
    - However, it's just modular arithmetic + a ptr dereference or two
    - It's still O(1), tho in reality often a bit slower than a `vector`, which is: *"address + i * size_of(elementType)"*
  - Inserting at either end is faster than a `vector` as reallocations cost N/K *pointer* copies rather than N object copies
    - If K is large, the N/K may be much smaller than N; however, you may end up with more wasted space on average

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

- If you need to insert at the front, use a `deque`
  - And if you need to insert in the middle, use a `list`
- If you don't need to insert at the front, you can use either a `deque` or a `vector`
  - Random access to elements is constant time for both, but a `vector` may be faster in reality
  - Reallocations:
    - Take longer with a `vector` (N vs. N/K)
    - `vector` invalidates external refs to elts, but not so with a `deque`
    - `vector` copies elts (which may be objects), `deque` copies only ptrs
  - The official C++ standard advises "When in doubt, use a `vector` ", but some other experts say the opposite

```
    cout << "\nWith a deque:" << endl;
    deque<int> d;
    d.push_back(4);   d.push_back(3);
    d.push_back(37);  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 0x7ff87bc039ec
With a deque:
15 15  0x7ff87c00220c 0x7ff87c00220c
15 15  0x7ff87c00220c 0x7ff87c00220c
```

```
// Preserving external reference integrity
#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);              // Causes a reallocation
    cout << *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

- With a `vector`, any external pointer directly to an element (such as `p` in the example) will be valid as long as no new elements are added (and there is no call to `resize` or `reserve` and if the element is not popped off)
  - Here, adding a fifth element to `v` causes a reallocation, which causes the `vector` objects to be copied to a new space
  - Note that the run-time system (probably) did not overwrite the value stored at `p`; however, as we can now see, the address stored at `p` is no longer where `v.at(3)` is located
  - So if you kept using `p`, your program may continue to work for a while, until the heap storage that `p` points into gets re-purposed for another object, and it's really hard to trace back to this original mistake
  - This is a nice example of how an observed error (a "failure") and its underlying "root cause" may be quite a conceptual distance apart

Integrity of external references

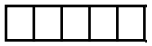
- With a `deque`, any external pointer directly to an element (such as `p` in the example) will be valid as long as you don't remove that element (e.g., by `pop_front/back` or via `resize` to a smaller size)
  - It's hard to predict when exactly a `deque` will perform a reallocation (it's not part of the `public` API; you would have to read/hack the library source code)
  - However, it's pretty likely that resizing to a BIG size will cause a realloc
  - But with `deque`s, it's the buffer of pointers that gets re-allocated; the chunks of element storage stay where they are; only the new element is copied (into its place)
  - So `p` will continue to be a valid pointer to that element of `d`
    - Note, tho, if more elements are added to the front, then the index of that element within `d` may no longer be 3

`list`

- Is implemented as a (plain, old) doubly-linked list (PODLL)
- Supports only sequential access, via iterators; no random access via `operator[]` or `at()`

Access kind	Complexity	API support
random access	O(N)	*not supported as an API call*
append/delete last	O(1)	`push_back/pop_back`
prepend/delete first	O(1)	`push_front/pop_front`
random insert/delete	O(1) (once you've arrived at the elt; O(N) to get there)	`insert/erase`

C++11 sequence containers: `std::array`



- Functionally, it's a (compile-time) fixed-size `vector`:

```
array<string,12> monthName = {"Jan", … "Dec"}
```

Q: Why not just use a C-style array?

- It's just as efficient as a C-style array, with none of the drawbacks
  e.g., it requires explicit effort to do insecure pointer trickery
- Like `vector`, `array` defines:
  - An `at()` method, providing safe, bounds-checked accessing
  - A `size()` method that returns the extent (which is set when you instantiate the `array`)
  - … and most of the other methods that `vector` supports
- There is no good reason to use a C-style array over `std::array`

C++11 sequence containers: `std::array`

Q: Why not just use a `vector`?

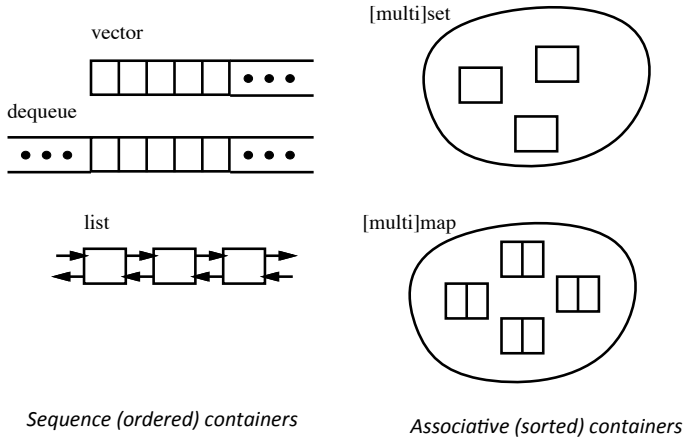
- Strong typing: If you know the size should be fixed, enforce it!
- Because `array`s are *sometimes* faster and more space efficient (i.e., less overhead)
  - Especially true is you have many, many small arrays; you may not notice a real difference otherwise
- The actual storage for `vector` elements is always on the heap (as it's a dynamic C-style array); the storage for `array` elements can be put on the run-time stack, since its size is known
  - Accessing a `vector` element costs a ptr deference to the heap (pretty fast)
  - Accessing an `array` element is an address calculation (even faster)

- So, all in all, `std::array` is nice, but not a big deal

C++11 sequence containers: `std::forward_list`

- Basically, it's just a singly-linked list
- Compared to `std::list` (which is implemented as a doubly-linked list):
  - You lose immediate access to the end of the list
    i.e., no `push_back()`, `back()`
  - You lose the ability to iterate backwards
  - You gain a little space efficiency, as you save the cost of one ptr per list element
  - There is no `size()` method, as keeping a size counter would make the linked list operations ever so slightly slower
- That's about it; it's no big deal

STL containers: Conceptual view



2. Container adapters

- Usually, a trivial wrapping of another sequence container data structure by a narrower interface with a specialized idea of how to add/remove elements
  - `stack`, `queue`, `priority_queue`
- The implementations use the ideas of *composition* and *delegation*
- You *can* specify in the constructor call which container you want to be used in the underlying impl.; the possibilities are:

`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;
};
```

Note that the STL provides its own definition of Stack: please use that one! This is just an ad hoc example.

```
template <typename T>
Stack<T>::Stack(): s() {}

template <typename T>
Stack<T>::~Stack () {}

template <typename T>
void Stack<T>::push(T val){
  s.push_back(val);
}

template <typename T>
void Stack<T>::pop () {
  assert (!isEmpty());
  s.pop_back();
}
// … etc
```

Lecture #2

Recap: Generics

- Want to treat element type as a kind of parameter
  - Get stronger typed containers, etc.
- Generic functions and classes!

```
template <typename T>
T flurble (const T& t) {…}
```

- C++ Standard Template Library!
  - Generic containers (`vector`, `deque`, `map`, `stack`, …)
  - Generic iterators (for traversing "collections")
  - Generic algorithms (`random_shuffle`, `sort`, `find`)

Recap — STL containers

1. Sequence containers

   e.g., `vector`, `deque`, `list`
   - General purpose container, ordered by "insertion order"
   - Roughly the same semantics, but different performance guarantees

2. Container adapters

   e.g., `stack`, `queue`, `priority_queue`
   - Sequence container with specialized add/remove
   - Implemented (usually) in terms of standard sequence container, but w. smaller intf

3. Associative containers [to come]

   e.g., `set`, `map`, `multiset`, `multimap`
   - Ordered by a "primary key" from the data
   - C++11 also adds unordered versions of each of these

Recap: Duck typing vs. inheritance

- STL doesn't use inheritance or define any methods as `virtual`
  - Encourages reuse via *adaptation* (instantiate & wrap), rather than inheritance
  - C++ templates allow for ad hoc polymorphism (+ thus "duck typing")
- STL design philosophy:
  - A container class should define the API that is natural for it!
  - So no inheritance, no implied subtyping
  - Your de facto shape is what defines you (do you walk like a duck?), not who you inherit from

The *adapter* design pattern

- Often, you find you have some library data structure that kinda implements what you want, but its API does not resemble what you would naturally desire.
  - For example, `vector` can be used to model a `stack`, but `vector` has quite a few more operations that you need for just `stack`s
  - Often, the provided operations are close to but not quite what you want e.g., the name `pop_back` instead of just `pop`
  - You don't want to force the rest of your system to remember the magical incantations for making a `vector` behave like a `stack`
- An adapter is sometimes also called a *wrapper* class
  - It's built on top of an adaptee / workhorse class, in this case: `vector`

```
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;
};
```

Note that the STL provides its own definition of Stack: please use that one! This is just an ad hoc example.

```
template <typename T>
Stack<T>::Stack(): s() {}

template <typename T>
Stack<T>::~Stack () {}

template <typename T>
void Stack<T>::push(T val){
  s.push_back(val);
}

template <typename T>
void Stack<T>::pop () {
  assert (!isEmpty());
  s.pop_back();
}
// … etc
```

The adapter solution

1. Define the API you really want
   - For `Stack`, we want: `push`, `pop`, `top`, etc, *not* `push_back`, `pop_back`, `back`
2. Instantiate (don't inherit) an object from the "workhorse" class that will do the actual heavy lifting, probably as a private data member
3. Define the appropriate operations as fairly trivial operations on the workhorse class
   - This is sometimes called *delegation* or *forwarding*

Adapters and the STL

- STL defines its own adapter classes for `stack`, `queue`, and `priority_queue`
  - You can also specify if you want, e.g., your `stack` to be implemented by a `vector`, `deque`, or `list` from the STL
- "The STL way" encourages to define your own adapter classes based on STL container classes, if you have special purpose needs that are almost satisfied by an existing STL class
  - And *not* using inheritance!
  - STL containers declare their methods as non-`virtual` for efficiency, so adapting is usually the right thing to do

Duck typing redux

- Suppose we would like to write a card game implementation, and we want to model a pile of playing cards
  - [actually, a pile of `Card*`, since the cards will be a shared resource and get passed around]
- We want it to support the natural `CardPile` ideas of `addCard`, `discard`, `shuffle`, `merge`, `print`, etc.
- We also want clients to be able to treat a `CardPile` like a sequential polymorphic container: iterate, find, add, delete, …
  - Should it be random access? If so, should we use `vector` or `deque`?
  - Or can we use `list`, which could be faster for merging two piles, …

Duck typing redux

- So the tempting thing is to say, right, let's just inherit from `vector` (or `deque` or `list`)
  - Then we get all of the `vector` operations for free, and unlike when we created a `Stack`, we actually want most of the `vector` operations to be a feature of `CardPiles`
  - We expect clients to want to iterate thru `CardPiles`, ask about their size, (maybe) access random elements, etc.

```
// Legal, but is it a good idea?
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);
        void shuffle ();
};
```

Duck typing redux

- The STL designers would say:
  - That's way ugly and a little dangerous!
  - All `vector` (etc.) methods are non-`virtual` and the dxr is also non-`virtual`
  - Better idea: create the interface you actually want with the members you need and ignore the rest
- Duck typing is heavily used in some languages (e.g., Python)
  - It's more flexible, "you get exactly what you want", "pay as you go"
  - It's less "regular", meaning you have to remember more details and exceptions to (apparent) conventions
  - My opinion: it indulges the programmer at the expense of clients and (later) maintainers

```
// So let us try instantiate-and-wrap (i.e., adaptation)
class CardPile {
    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 ();
        // Wrapped container methods and types
        typedef vector<Card*>::iterator iterator;
        typedef vector<Card*>::const_iterator const_iterator;
        iterator begin();
        const_iterator begin() const;
        iterator end();
        const_iterator end() const;
        int size() const;
        Card* at(int i) const;
        void pop_back();
        Card* back() const;
        bool empty() const;
    protected:                  // or maybe private
        vector<Card*> v;
};
// An example of a function wrapper
Card* CardPile::at(int i) const {
    return v.at(i);
}
```

But there is another way …

- Remember I said to remember to say `public` whenever you inherit something?

  `class Circle : public Figure { …`

  – That's because if you don't you get `private` inheritance by default

- We don't really want `private` inheritance with `Circle`, but it we *did* then this is what would happen:
  – Inside the class def. of `Circle`, we would have have *direct* access to the non-`private` methods of `Figure` (i.e., without "wrapping")
  – Outside of `Circle`, clients would *not* be able to treat a `Circle` as if it were polymorphically a `Figure`
    - External clients who have a Circle can't use `public` members of `Figure`
    - Cannot instantiate a `Circle` to a `Figure*`

`private` inheritance

- That is, all of the inherited (`public` and `protected`) members of the parent are `private` in the Child, and you break external polymorphism with the parent.

- We can selectively make some of the methods of the parent visible to clients using `using`, as in "`using Figure::getColour;`"
  – Making something `private` into `public` is called "promotion"

- Just list in the `public` part the features from the parent that you want to promote to be members of your own
  – Note that selection granularity is at the level of member name; if you inherit a set of overloaded methods with the same name, you must promote all or none of them

```
// Using private inheritance
class CardPile : private 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);
        void shuffle ();
        // …
        // "Promoted" container methods and types
        using vector<Card*>::iterator;
        using vector<Card*>::const_iterator;
        using vector<Card*>::begin;
        using vector<Card*>::end;
        using vector<Card*>::size;
        using vector<Card*>::at;
        using vector<Card*>::pop_back;
        using vector<Card*>::back;
        using vector<Card*>::empty;
};
```

- We get all versions of any overloaded methods we "use", so we get both of these versions of `begin()`:

```
vector<Card*>::iterator vector<Card*>::begin();
vector<Card*>::const_iterator vector<Card*>::begin() const;
```

`private` inheritance

- This approach is "safe" because it breaks polymorphism!
  - Can't instantiate a `CardPile` to a `vector<Card*>`, so there's no risk of a call to the wrong dxr causing a memory leak or worse
  - Also, clients won't accidentally call "wrong" version of inherited non-`virtual` function
    - None of the inherited functions are visible to clients unless explicitly made so by using `using` (in which case, you use the parent definition anyway)
    - If you redefine an inherited function, clients will get the that version, since they can't see the parent version
- All that said, `private` inheritance is a bit of a hack
  - It is not conceptually different from instantiating a `vector` as a `private` data member, and creating wrapper functions for the desired API elements of the parent, it's just less typing

3. 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 records sorted by SIN or name or …)
    - and *not* by the order of insertion
  - Implemented using a kind of BST => lookup is O(log N)
  - Can iterate through container elements "in order"
- *Unordered* associative containers [new in C++11]

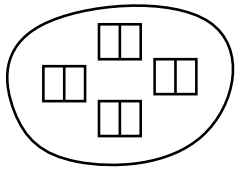
  `unordered_[multi]map`, `unordered[multi]set`
  - No ordering assumed among elements
  - Implemented using hash tables => lookup is O(1)
  - Can iterate through container elements, but no particular ordering assumed

Ordered associative containers

`set, multiset, map, multimap`

- `map` is probably the most useful
  - It's also called an associative array in other languages
- We don't usually think of sets, etc. as being sorted (or sortable), but they must be to use these library classes
  - We must ensure we have an ordering defined on the element type via `operator<()`, even if there isn't a natural one … or …
  - We can provide our own functor definition of via `operator<()` to the container constructor

`map`



- A map maps a *key* to a (unique) *value*
- Typical declaration:

```
map<T1, T2> m;
```

  - `T1` is the *key field* type; it must support `operator<`, which must in turn be a *strict weak ordering*
    i.e., anti-reflexive, anti-symmetric, transitive
    - It's common to use strings or numbers as key; can use ptrs
    - Can use a user-defined class but you must ensure there is a "reasonable" `operator<` defined or provide an ordering *functor* to the `map` constructor
  - `T2` is the *value field*; can be anything

"The same key"

- The compiler will use the following test for equality for elements in ordered associate containers, *even if you have your own definition of* `operator==`

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

- It is possible to have two containers of the same underlying key type that are sorted on different definitions of `operator<`
  - This involves creating a *functor* object, and passing it as an argument to the set/map constructor
  - For example, we could have one `set` of `Student`s sorted by name, and another `set` sorted by student number

```
#include <iostream>
#include <string>
#include <map>
using namespace std;
int main (int argc, char* argv[]) {
    // record the number of times each word appears.
    map<string, int> m;
    string token;
    while (cin >> token) { // This is where it all happens
        m[token]++;
    }
    // After below line is executed, m["the"] is part of the
    // map even if it didn't appear in the input stream!
    cout << "\"the\" occurred " << m["the"] << " times\n";

    // So let's erase it if so ...
    if (0 == m["the"]) {
        m.erase("the");
    }
```

```
    // map supports bi-directional iterators; so this will
    // print pairs in "alphabetic" order of key
    for (map<string,int>::const_iterator i=m.begin();
            i!=m.end(); i++) {
        // For maps, "first" gets you the key ...
        // ... and "second" gets you the value
        cout << (*i).first << "  " << (*i).second << endl;
    }
}
```

Querying for an element

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

```
if ( words [ "bach" ] == 0 )
        // bach not present
```

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

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

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

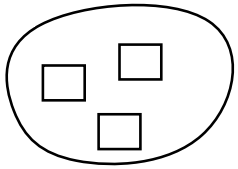
end() is iterator value that points beyond the last element in a collection

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

// Examples adapted from Josuttis
int main (int argc, char* argv[]) {
    map<string,string> dict;
    dict["car"] = "voiture";
    dict["hello"] = "bonjour";
    dict["apple"] = "pomme";
    cout << "Printing simple dictionary" << endl;
    for (map<string,string>::iterator i=dict.begin();
            i!=dict.end();i++){
        cout << (*i).first << ":    " << (*i).second << endl;
    }
```

```
    // Examples 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
            i=mdict.lower_bound("car");
            i!=mdict.upper_bound("car"); i++) {
        cout << (*i).first << ":    " << (*i).second << endl;
    }
}
```

set



- Typical declaration:

```
set<T> s;
```

- Normal (i.e., non-multi) `set`s do not allow duplicate elements
  - If you try to `insert` an element who key value is already present in an element in the set, then you get back an iterator pointing to that element as the return value of `insert` (and the set is left unchanged) plus a boolean flag to indicate the success

- `multiset::count(const key_type& x)` returns the number of elements that match the key value `x`

```
// Example with user defined class and operator<
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;}
```

```
// Need not to be friend of Student as getSNum() is public
bool operator< (const Student& s1, const Student& s2) {
    return s1.getSNum() < s2.getSNum();
}

// Also need not be a friend of Student
ostream& operator<< (ostream &os, const Student & s) {
    os << s.getName() << " " << s.getSNum()
        << " " << s.getGPA();
    return os;
}
```

```
int main (int argc, char* argv[]) {
    Student * pJohn = new Student ("John Smit", 666, 3.7);
    Student * pMary = new Student ("Mary Jones", 345, 3.4);
    map<string,Student*> m;
    m["johnS"]= pJohn;
    m["maryJ"]=pMary;
    // Will print alphabetical order by key (name)
    for (map<string,Student*>::const_iterator i=m.begin();
            i!=m.end(); i++) {
        cout << (*i).first << " " << (*i).second->getName()
            << " " << (*i).second->getGPA() << endl;
    }
    set<Student> s;
    s.insert(*pJohn);
    s.insert(*pMary);
    // Will print in numeric order of sNum
    for (set<Student>::iterator i=s.begin(); i!=s.end();i++){
        cout << (*i) << 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

- Mathematically, maps are sets
  - So, if you have an implementation of `set` , then you can implement `map` as a set of pairs `<key,value>`, and use the `operator<` defined on the key type as `operator<` for the pair type
  - And implementing the `multi`-versions requires only a little more work

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

I. Generic *containers* that take the element type as a parameter
e.g., `vector, list, deque, map, set` plus `stack, queue,` etc.

II. Kinds of *iterators* that can navigate through the containers

III. *Algorithms* that take an iterator, and perform an interesting operation on the elements in that range
e.g., `sort, random_shuffle, next_permutation`

II. Iterators

- The iterator is a fundamental *design pattern* of OOP
  - It represents an abstract way of walking through some interesting data structure, call it `v`, e.g., using a `for` loop
  - You start at the beginning, advance one element at a time, until you reach the end

- In its simplest form, you are given:
  - A *ptr* to the first element of the collection
  - A *ptr* to *just beyond* the last element; reaching this value is the stopping criterion the iteration
  - A way of advancing to the "next" element (usually, `operator++`)
    - Bi-directional iterators also define `operator--` to move backwards
  - Note that dereferencing the iterator/ptr gets you to the element

So if `f` expects an iterator param …

- Usual usage pattern for iterators in the STL:

  `f (iter1, iter2, …)`

- The implementer of `f` will assume that:
  - $iter_1$ and $iter_2$ are pointers
  - If I set $p=iter_1$, then `p++` should "advance to the next element"
  - `(*p)` should get me to the current element
  - I should stop when $p==iter_2$ (and without doing any work using `p` at that value)

- That's all we need to know to write `f`!

STL containers provide iterators

- If `c` is a `vector`, `deque`, `list`, `map`, `set`, etc., then
  - `c.begin()` will always return a ptr to the "first" element
  - `c.end()` will always return a ptr to "one beyond" the last element
  - `operator++` will be defined so that it "works"

  - So you say:

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

This is the type!

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

Kinds of iterators

- *Forward iterators,* where `begin()` starts you at the first element, `++` takes you to the next element, and `end()` is one past the last element

- *Bidirectional iterators* allow you to go backwards too by decrementing the iterator
  e.g., `vi++, vi--`

- *Reverse iterators*, where `rbegin()` starts you at the last element, `++` takes you to the *previous* element, and `rend()` is one before the first element
  - The base type must support bidirectional iterators

Kinds of iterators

- ***`const` iterators***, where you promise not to change the collection or the elements
  - In C++11, use `cbegin()` / `cend()` instead of `begin()` / `end()`

- *Random access iterators*, e.g., for navigating vectors, allow you to access random elements in no worse than ACT
  e.g., if `vi` is an iterator into (the middle of) a `vector`, then `vi[3]` is the third element *after* the element pointed to by `vi`
  - Random access iterators are also bi-directional

Random access iterators

- If `vi` is a random access iterator, then `vi[3]` gets you access to the third element after the one currently pointed to by iterator `vi` *in no worse that amortized constant time (ACT)*
  - Obviously, since `vi` is an iterator, you can get there eventually by just incrementing it three times, but perhaps not in ACT
  - For example, iterators for `vector`s *are* random access, as you can get to the third element from here by a simple address calculation
    - The STL definition requires that `vector` elements be stored *contiguously* i.e., like as array is
  - On the other hand, iterators for STL `list`s are *not* random access as to get to k elements from the current one, you have to follow k pointers; similarly, `map`s are implemented as red-black trees, so you can't just jump to the third element from the current one
    - Thus, iterators for `list`, `map`, and `set` do not support `operator[]`

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

int main (int argc, char* argv[]) {
    vector<string> v;
    v.push_back("alpha");
    v.push_back("beta");
    v.push_back("gamma");
    v.push_back("delta");

    cout << "Forwards" << endl;
    for (vector<string>::const_iterator vi=v.begin();
            vi!=v.end(); vi++) {
        cout << (*vi) << endl;
    }
```

```
    cout << "\nBackwards using a bidirectional iterator\n";
    for (vector<string>::iterator vi=v.end()-1;
            vi!=v.begin(); vi--) { // Stops one too early
        cout << (*vi) << endl;
    }
    cout << "\nBackwards using a reverse iterator\n";
    for (vector<string>::reverse_iterator rit=v.rbegin();
            rit!=v.rend(); rit++) {
        cout << (*rit) << endl;
    }
    // Print pairs of elements; thus, need to double
    // increment the iterator each time thru;
    cout << "\nTwo per line w. a random access iterator\n";
    for (vector<string>::iterator vi = v.begin();
            vi!=v.end(); vi++, vi++) {
        // vi[1] is legal as vi is a random access iterator
        cout << *vi << " " << vi[1] << endl;
    }
}
```

Note: if `v` were a `list` instead of a `vector`, then `vi[1]` would be a compile-time error.

Why iterators are awesome

- They provide a simple, natural interface for accessing container elements; they are implemented for you by each STL container, e.g.,

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

- You can create iterators for your own (STL-derived or entirely homebrew) containers to provide access to your elements in an order you decide is appropriate while at the same time hiding the grungy implementation details from clients
  - Then you can use any of the STL algorithms on your container; also, external clients can define their own algorithms
  - Defining your own iterator isn't hard, but is subtle; we won't look at it

Why iterators are awesome

- Each STL container class defines at least one iterator type, plus can point you to the first and "last" element for each
  - `vector` and `deque` provide forward and backward random access bi-directional iterators (`const` or not)
  - `list`, `[multi]set`, `[multi]map` provides forward and backward, bi-directional iterators (`const` or not)

"I agree, iterators are awesome! I want to use an iterator, right now!"

- OK, but the collection of stuff you are iterating thru needs to provide you with one (or you need to define one for your collection)
  - As we saw, STL containers do this!
  - Also, ptrs to C-arrays can be used as iterators (C++ expressly supports this for backward compatibility … alas)

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

template <typename T>
void printV (const vector<T> & v) {
    for (typename vector<T>::const_iterator vi=v.begin();
            vi!=v.end(); vi++) {
        cout << (*vi) << endl;
    }
}

void printA (string A[], int extent) {
    for (int i=0; i<extent; i++) {
        cout << A[i] << endl;
    }
}
```

```
int main (int argc, char* argv[]) {
    vector<string> v;
    v.push_back("cat");        v.push_back("zebra");
    v.push_back("alpaca");     v.push_back("alligator");
    v.push_back("dog");        v.push_back("sloth");
    v.push_back("monitor lizard");
    const int N = 7;
    assert (N == v.size());
    string A[N];
    for (int i=0; i<v.size(); i++) {
        A[i] = v.at(i);
    }
    printV(v);
    printA(A, N);
    sort(v.begin(), v.end()); // Use vector<T> iterator
    printV(v);
    sort (&A[0], A+N);        // Or C ptrs; note "one beyond"
    printA(A, N);
}
```

C++11 and `auto`

```
for (typename vector<T>::const_iterator vi=v.begin();
```

- Boy, that's some awkward declaration there
  - We need the `typename` because … well, it's complicated …

- C++11 has introduced a form of *type inferencing*: the compiler *infers* the type of a newly declared variable from the underlying type of its initializing expression; in the above case:

```
vector<string> v; // …
for (auto vi=v.cbegin(); vi!=v.cend(); vi++) {
    // "Aha", says the compiler to itself, "vi must
    // be of type vector<string>::const_iterator"
```

Why iterators are awesome

- They are a nice example of both information-hiding and polymorphism (same basic interface for all containers), tho the implementation is necessarily a bit of a hack job
- As defined/used within the STL, they are compatible with C pointers, meaning you can use STL algorithms with some legacy C data structures
- You can use iterators to represent ranges of values inside containers, e.g., for inserting multiple elements
  - So you don't need to always start at the beginning and end at the end

Why iterators are awesome

- All STL containers support `insert` + `erase` methods, e.g.,

`v.insert (iter1, iter2, iter3)`

  - Insert into `v` at position $\texttt{iter}_1$ a range of (external) elements whose begin/end points are $\texttt{iter}_2/\texttt{iter}_3$

`v.erase(iter1, iter2)`

  - Erase from `v` all elements in the range $\texttt{iter}_1$ to $\texttt{iter}_2$ (i.e., *not* including any element $\texttt{iter}_2$ might point to)

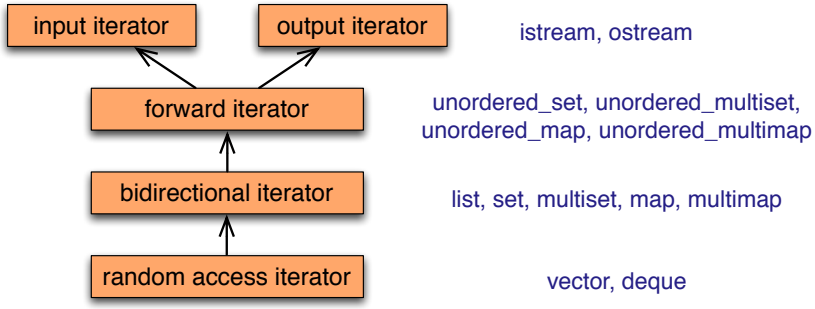
```
int main (int argc, char* argv[]) {
    list<string> a;
    a.push_back("Keon");
    a.push_back("Sittler");
    a.push_back("Gilmour");
    list<string> b;
    b.push_back("Ramage");
    b.push_back("Vaive");
    b.push_back("Clark");

    // Insert just before last element
    a.insert (--a.end(), b.begin(), b.end());
    cout << "After insertion" << endl;
    // Output: Keon Sittler Ramage Vaive Clark Gilmour
    for (list<string>::const_iterator it=a.begin();
            it!=a.end(); it++) {
        cout << (*it) << endl;
    }

    // Erase up to but not including 3rd element
    a.erase(a.begin(), ++++a.begin());
    cout << "After erase" << endl;
    // Output: Ramage Vaive Clark Gilmour
    for (list<string>::const_iterator it=a.begin();
            it!=a.end(); it++) {
        cout << (*it) << endl;
    }
}
```

STL Iterator Categories

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



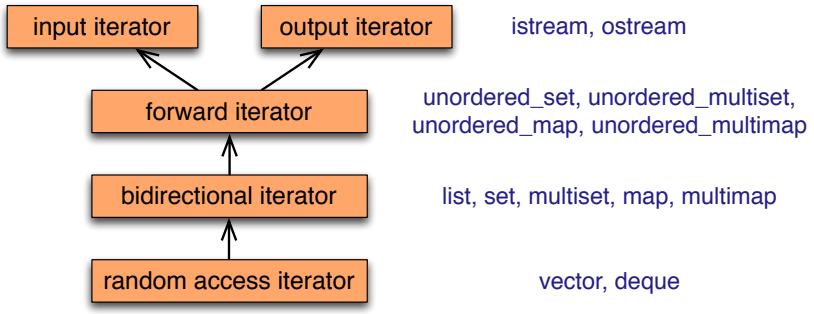
Why should you care??

Other Iterator Categories

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()`

```
#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 ));
```

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.

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

The C++ Standard Template Library

I. Generic *containers* that take the element type as a parameter
   e.g., `vector`, `list`, `deque`, `map`, `set` plus `stack`, `queue`, etc.

II. Kinds of *iterators* that can navigate through the containers

III. *Algorithms* that take an iterator, and perform an interesting operation on the elements in that range
   e.g., `sort`, `random_shuffle`, `next_permutation`

III. Algorithms (and iterators)

- STL algorithms perform an abstract operation on a set of data,
  e.g., `sort`, `random_shuffle`, `find`
  - They take iterators (typically, at begin/end spots of the container), plus possibly additional arguments.
  - They can be used on any data structure that can be walked with an iterator, not just STL containers.

- Usual form of a call to an STL algorithm:
  `f(iter`$_1$`, iter`$_2$`, arg`$_1$`, arg`$_2$`, …)`

  e.g., If `m` is a list and you're looking for an element that matches `val`, say
  `"find (m.begin(), m.end(), val)"`

Algorithms vs. container methods

- By using iterators we (usually) need define only one implementation of a given algorithm, and it will work on (almost) all STL containers, as well as any other structure that can be walked with an iterator!

- In practice, we sometimes provide tuned container-specific versions of some routines as methods of the container for efficiency; for example:
  - `find()` using naive iterators is O(N) whereas `set::find()` is O(log N) as it uses a BST and `unordered_set::find()` is O(1) as it uses a hash table
  - `sort()` uses a variation of quicksort, which is only efficient when there's random access to the data, as in `vector` and `deque` but not `list` or other STL containers; these containers provide their own `sort()` methods

STL (C++98/03) container	*Some* useful operations
all containers	`size, empty,` **`insert, erase`**
`vector<T>`	`[], at, back, push_back, pop_back`
`deque<T>`	`[], at, back, push_back, pop_back, front, push_front, pop_front`
`list<T>`	`back, push_back, pop_back front, push_front, pop_front,` **`sort, merge,`** `reverse, splice`
`set<T>, multiset<T>`	**`find`**
`map<T`$_1$`, T`$_2$`>, multimap<T`$_1$`, T`$_2$`>`	`[], at,` **`find, count`**
Other ADTs in the STL	`stack, queue, priority_queue, bitset`

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

Some useful `algorithms`

- `find` locates the first element that "matches" a given object
- `count` counts the number of elements that "matches" a given object
- `for_each` applies a function to each element (similar to `map` in Scheme)
- `remove` removes all matching elements
- `replace` replaces all matching elements with a specified new object
- `sort` sorts the elements (not very useful for associative containers)
- `unique` removes adjacent "identical" elements (useful if container is sorted)
- `min_element, max_element` …
- `nth_element,`…
- `random_shuffle,`…
- `next_permutation` can cycle through all N! permutations of an ordered container!

```
// #include other stuff too
#include<algorithm>
int main(…) {
    vector<Figure*> v;
    // … add stuff to v
    random_shuffle (v.begin(), v.end());

    vector<string> wordList();
    // … add stuff to wordList
    sort (wordList.begin(), wordList.end());

    string s = "edcba"; // strings are like char vectors
    // changes made in place
    random_shuffle (s.begin(), s.end());
    cout << s << endl;
    sort (s.begin(), s.end());
    cout << s << endl;
}
```

```
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>
using namespace std;
bool isSorted (vector<string> v){
    for (int i=0; i<v.size()-1; i++) {
        if (v[i] > v[i+1]) {
            return false;
        }
    }
    return true;
}
void print (vector<string> v) {
   cout << "\nPrinting a vector:" << endl;
   for (vector<string>::iterator i=v.begin();i!=v.end();i++){
        cout << " " << (*i) << endl;
    }
}
```

```
int main (int argc, char* argv[]) {
    vector<string> v;
    v.push_back("hello"); v.push_back("there");
    v.push_back("world"); v.push_back("how");
    v.push_back("are");   v.push_back("we");
    v.push_back("doing"); v.push_back("today");
    print(v);
    int nfactorial = 1;
    for (int i=1; i<=v.size(); i++) {
        nfactorial *= i;
    }
    for (int i=2; i<=nfactorial; i++) {
        next_permutation (v.begin(), v.end());
        if (isSorted(v)) {
            cout << "Found a sorted version at "
                    << "permutation number " << i <<endl;
            print (v);
        }
    }
}
```

STL `algorithms`

- We've barely brushed the surface; there's lots of good, useful and well tuned stuff in here
  - http://www.cplusplus.com/reference is a good reference for the API
- An engineer understands his/her tools well, and knows how to use them appropriately
  - It's probably worth getting to know the C++ standard library if you're going to work in C++
  - A lot of high-quality work has been done for you, but you still need to understand the trade-offs
- The new C++11 standard adds a lot to the library

End of lecture #2

The C++ Standard Template Library (STL)

CS247 guest lecture, Mike Godfrey

July 8 and 10, 2014