- **Library Organization and Standard Containers**
- **Stroustrup ch 30 overview of std lib**
- **Stroustrup ch. 31 STL Containers**
- **Stroustrup ch. 34 sections on “almost containers”**
- **“Standard Template Library”**
  - **containers, “algorithms”, and iterators**
  - **iterators point to items in containers**
  - **“algorithms” are function templates written in terms of iterators**
  - **so same algorithm function can work on all appropriate containers**
  - **Focus on containers first**
- **Container highlights**
  - **A lot of additions made in C++11; some refinements in C++14.**
  - **A basic concept of the containers and the Standard Library**
    - *Provide good implementations of things likely to be generally useful.*
      - Not specialized things
    - *Container philosophy*
      - Containers have very good performance in big O sense
      - Complexity is actually part of the specifications in the Standard!
      - Avoid providing facilities that result in or encourage poor performing code
        - E.g. why no ordered array or ordered list - tree-based containers are better.
        - Why no hash table containers in original STL - no guaranteed performance.
        - Added later by popular demand, but a good big O can't be specified ---
  - **terminology**
    - *often write std::container_name<> to refer to one of the standard container templates*
      - e.g. std::list<> or std::map<>
      - these were in the original "Standard Template Library" that was incorporated into the C++ Standard Library, still sometimes called STL, though that is not the official name
  - **elements in a container**

- *must be all the same type*
- *all containers contain objects, usually copies of supplied objects, and move them by assignment - so elements must have properly defined public members:*
  - copy ctor - usually needed if the container has an internal array of objects
  - copy assignment operator
  - destructor - if a class type
  - move constructor and move assignment will be used if available and appropriate
  - must be swappable in normal sense - e.g. using std::swap (which copies and also moves if possible).
- *But client code can provide movable objects to be put into the container - faster than full copying:*
  - Thing my_thing;
    vector<Thing> v;
    v.push_back(std::move(my_thing)); // we don't need my_thing any more, so move it into the container;
  - Std containers have move versions of functions for adding items (take object by rvalue reference)
- *Client code can also ask for an object to be constructed "in place" inside the container, instead of copied or moved in - can be faster, more convenient:*
  - vector<Thing> v;
    Thing t1 {"Bill", 42, "green"}; // construct with three parameters.
    v.push_back(t1); // copy it into a new cell at the end
    v.emplace_back("Pete", 666, "red"); // construct a Thing with these parameters in a new cell.
  - Done with C+11 template magic: variadic templates and "perfect forwarding" - another use of rvalue references.
- *for some containers, algorithms, the objects in the container might also need:*
  - a default ctor
  - an equality operator
  - a less than operator
- *If a container has a class type object for its contents, when that item is removed (e.g. with erase() member function), the dtor is automatically run to destroy that object.*
- *use pointers to objects in a container if*
  - want polymorphic types, different subtypes
  - each object must be represented in more than one container
  - NOTE: containers do not *ever* do a delete on an item, even it if is a pointer! Why?
    - the container is no longer general - purpose, but has a different behavior in case of pointers
      - note that delete of int or delete of a Thing object wont compile!

        - Experience shows that if you allocate objects and put pointers to them into a container, you rarely want the container to take responsibility for getting rid of the objects - you created them, you should destroy them when you know it time to do so. The container is responsible for its guts - e.g. list nodes - not other objects that you happen to give it pointers to.
    - *If it has a pointer to an object for its contents, when that item is removed, nothing is done to the pointed-to object,*

- **iterators as an abstracted pointer**
    - *act like pointers to elements in the container*
        - same concept as in Project 2
    - *.begin(), .end() return iterators to first object, and "one past the last object"*
    - *note: begin() and end() return const_iterators if the container is const*
        - behave like pointer-to-const - you can't modify what they point to
        - compiler chooses a version that returns the const iterator instead of plain iteratore
    - *.cbegin() and .cend() will return const iterators even if container is non-const*
    - *more later, but each container has its own types of iterators*
        - type is available as container_name::iterator
            - e.g. list<Person *>::iterator it;
        - can be as simple as an actual pointer (the above just typedefs it)
            - or as subtle as threading through the leaf nodes in a binary tree
            - or as wild as following through the bins in a hashed container
    - *use like a pointer, but contains whatever needed, and advances however needed*
        - vector<Thing> iter++ goes to next cell in the internal array
        - list<Thing>, iter++ goes to next node
        - e.g. set<Thing>, iter++ goes to next item in order (red-black tree traversal)
    - *concept of "past the end" - actually simpler that "at the end", because avoids have to special case the last member in an iteration, and avoids having to define < or > for iterators*
        - for(it = container.begin(); it != container.end(); ++it) // the idiom
    - *Why should we avoid defining < or > for iterators? No efficient implementation in many cases!*

- **Three gotchas with iterators**
    - *Gotcha #1. If the container is const, then you must use a const_interator to access its items*
        - through template magic and overloaded functions, automatically supplied by begin() and end() and other functions.

- If a member function that accesses a member variable container is const, it means that the container is const in the function, so a const_iterator has to be used.
- A common error:
  - ```
    class Thing {
    public:
    Thing() {
         for(int i = 1; i < 11; ++i)
              ints.push_back(i);
              }
    void printem() const;
    private:
    list<int> ints;
    };

    void Thing::printem() const
    {
         // error - need a const_iterator here!
         for(list<int>::iterator it = ints.begin(); it != ints.end(); ++it)
              cout << *it << endl;
    }

    int main()
    {
         Thing t;
         t.printem();
    }
    ```
- Easy fix! auto keyword "automatic declare" a variable to have same type as initializer:
  - ```
         for(auto it = ints.begin(); it != ints.end(); ++it)
              cout << *it << endl;
    ```
- Easier to type, and automatically becomes a const_iterator if that's what's needed!

- *Gotcha #2. Depending on the container, altering the container contents may INVALIDATE an iterator already pointing into the container*
  - A serious problem with sequence containers that are array based, less so for node-based.
  - If iterator invalidated, the results of using the iterator are UNDEFINED
  - a RUN TIME ERROR, not a compile-time error
  - no warning, just have to program carefully - efficiency again!
  - best solutions depend on container type - not completely general.
- *Gotcha #3. A const_iterator means you can't use the iterator to modify the pointed-to item inside the container. But you can add or remove items using a const_iterator to designate the location!*
  - A const_iterator is analogous to a pointer to const object.
    - Consider that could always create/destroy with pointer-to-const object
    - const Thing* p = new Thing; // initialize pointer-to-const
    - delete p; // it is legal to destroy object using pointer-to-const

- container.erase(it) is not modifying the pointed to-object, but telling the container which object to remove.
  - that causes the object to be destroyed, but see above - that's legal.
- container.insert(it, x) is not modifying the pointed-to object, but telling the container where to insert x.
- Therefore, erase() and insert() member functions are defined as accepting a const_iterator argument.
  - Note: a non-const iterator will implicitly convert to a const_iterator
  - Change made in C++11, but obscure.

- **The types defined inside a container class template**
  - *- a way to access useful information about an instantiated container - type aliases or typedefs inside the class template*
    - list<Thing> t;
    - list<Thing>::iterator is the type of the iterator for a list of Things
      - for(list<Thing>::iterator iter = etc
    - list<Thing>::value_type is Thing
    - list<Thing>::pointer_type is Thing*
  - *others are mostly useful for writing other templates, but occasionally useful in just using a template.*
- **Comparison operation - used in algorithms, associative containers**
  - *usually just operator<*
  - *can get effect of operator> just by reversing the arguments*
  - *can get effect of operator== by just trying both < and >*
  - *arcane tidbit!*
    - can get other relations automatically generated by rel_ops template classes
      - S 35.5.3
  - *for pointers, have to define function objects*
    - a function, supplied by function pointer, will work, but function object usually simpler and preferred
    - see note about operator< on char * doesn't do what you think it will
      - puts in address order, not order of pointed-to C-strings
    - ditto for ANY pointers in containers
    - have to define function object that says how to order the pointers based on the pointed-to data

Two basic container types

sequences - items in the order they were explicitly placed into the container

- *vector, list, deque*
- *adapted to stack, queue, priority-queue*

associative - items are always sorted by the comparison function

- *map, set, multimap, multiset - ordered*
- *unordered_map, etc - hash-table based*

Two different underlying implementations:

array-based

- *vector, deque*
- *contain an internal dynamically allocated array*

node-based

- *list, map, set, multimap, multiset*
- *allocate space for one item at a time*

- **Unordered containers probably a combination of both**

This implementation determines whether an iterator pointing to an item stays valid if other items are added or removed

- *node-based - iterator points to a node, stays valid if other nodes added or removed*

array-based - iterator points to a cell, may be invalidated if number of items changes - e.g. memory gets reallocated, or an item is removed and other items "moved up" to fill the empty cell.

- if items added or removed, incrementing an existing iterator value to point to the next item is invalid

Issue often appears when scanning through a container and removing particular items:

- *iterate through the container; if we want to erase the item at the iterator, then we want to call cont.erase with the iterator, but continue the scan somehow with ++ on the iterator*

For associative containers, to scan and erase, just post-increment the iterator in the call to erase

- ```
  auto it = assoc_cont.begin();
  while (it != assoc_cont.end()) {
       if(dontwant(*it))
            assoc_cont.erase(it++); //point to correct next node before erasing
       else
            ++it;
       }
  ```

For sequence containers, member functions give you a correct next iterator if you are scanning the container
  ```

- vec.erase(it) returns an iterator to the true next item; can use this to scan a vector and remove things.
  - ```
    auto it = vec.begin();
    while (it != vec.end()) {
         if(dontwant(*it))
              it = vec.erase(it); // get the next value for the iterator back
         else
              ++it;
         }
    ```
- vec.insert(item) returns an iterator pointing to the next true element
- Scott Meyer's suggestion (see later on remove algorithm)
  - `vec.erase(remove_if(vec.begin(), vec.end(), dontwant), vec.end())`
    ```

list

- **add and remove at both ends, and insert using an iterator, but no subscript operator.**
  - *no efficient way to find an element "by number" - have to count from the end - O(n)*
- **no built-in "insert in order" function - have to do it yourself**
  - *why? inherently not very efficient - use some other container is recommended*
- **List has a sort member function**
  - *optimized for linked-list representation, while vector and deque have array-like properties, so algorithms work well for them.*
- **List has a remove member function that erases all elements that match a specified value**
  - *can take advantage of speed of pointer changes - just cut and splice to eliminate them*
  - *notice that the remove algorithm (later) does something very different*

forward_list<type>

- **forward_list<type>**
- **more limited than list<>**
- **concept - provide a lightweight list container that can be traversed only in the forward direction**
  - *principle - don't support stupid operations*
    - only a single member variable - pointer to first node
    - no operation supported that requires finding the end of the list in order to do the operation, or scanning the list from the begining.
      - no size() function! - use empty() instead! (good in general)
      - no insert() function because that normally mean insert before the interator locaton
      - only insert_after - insert the new item after the iterator location.
    - #include <forward_list>
      using namespace std;

      forward_list<int> fl;
      fl.push_front(42);
      // fl.push_back(42); // error - not provided
      auto it = fl.begin(); // ok
      // auto rit = fl.rbegin() // error - no reverse iterators

      // cout << fl.size() << endl; // error -size function not provided

    - *faster, less memory demand than list<> - good for use in high-performance, limited memory situations if it is all you need.*

vector

vector - insert

- *you can insert before a place pointed to by an iterator, but it could be quite slow.*
- *push_front not supplied because it would be ridiculous*

- **inserting/removing objects can involve moving multiple objects or very expensive copy/assignment of multiple objects if move not available.**
- **push_back is provided, but not push_front.**
- **vector used in the sort algorithms (can be quite efficient)**

because contains a contiguous array, can be traversed very rapidly on modern CPUs

- *means linear search might be surprisingly fast …*

if vector is in sorted order (either sort algorithm used or inserted in order), then can use binary_search and lower/upper bound algorithms to find things very quickly

- *most commonly, no duplicate items, but these are well defined if duplicates are allowed.*
- *by default, uses the operator< for comparisons*

oddness in these two very similar algorithms as commonly used

- binary_search returns true/false and so tells you whether or not the searched-for thing is present, but not where it is, nor where to put it if it isn't present.
- lower_bound returns an iterator that tells you where it is if it is present, or where to put it if it is not, but not directly whether or not it is there - you have to do another comparison.

using binary_search algorithm

- ```
  vector<int> vi;
  /* populate vi with values in sorted order, smallest to largest */
  int probe;
  cin >> probe;        // get probe value from user
  bool found = binary_search(vi.begin(), vi.end(), probe);
  if(found) {
          cout << "Found!" << endl;
          }
  else {
          cout << "Not found!" << endl;
          }
  ```

using lower_bound algorithm to erase if present, insert if not

- ```
  vector<int> vi;
  /* populate vi with values in sorted order, smallest to largest */
  int probe;
  cin >> probe;        // get probe value from user
  auto it = lower_bound(vi.begin(), vi.end(), probe); // returns vector<int>::iterator
  if(it != vi.end() && *it == probe) { // does non-end() iterator point to matching value?
          cout << "Found - erasing " << *it << endl;
          vi.erase(it);
  ```

```
        }
    else { // not there, and iterator points to where to put it (possibly at end)
        cout << “Not Found - inserting ” << probe << endl;
        vi.insert(it, probe);
        }
```

- **Basic concept for searching a container: specify what to search for with a probe object of the same type as what is in the container. Called *homogenous lookup* - the type of the probe is the same as the type of the things in the container.**
- **Trivial in case of vecter<int> or vector<string>, but what if the specification isn’t of the same type? We have to construct a probe object from the specification so that the probe type matches the container type.**
  - *Example: Using a Thing type that has string name*
    - ```
      class Thing {
      public:
        Thing(const string& name_) : name(name_) {}
        const string& get_name() const
          {return name;}
        bool operator< (const Thing& rhs) const
          {return name < rhs.name;}
        friend ostream& operator<< (ostream& os, const Thing& t);
      private:
       string name;
      };

      ostream& operator<< (ostream& os, const Thing& t)
      {
        os << "Thing " << t.id << ' ' << t.name;
        return os;
      }
      ```
    - ```
      vector<Thing> things = {Thing("Dick"), Thing("Harry"), Thing("Tom")}; // in order

      while(true) {
          cout << "Enter name: ";
          string str;
          cin >> str;
          Thing probe(str);
          auto it = lower_bound(things.begin(), things.end(), probe);
          if(it != things.end() && it->get_name() == str)
              cout << “Found: " << *it << endl;
          else
              cout << “Not found" << endl;
          }
      ```
- **Instead of constructing a homogenous probe Thing from the string, can’t we look things up just with the string? Called *heterogenous lookup* - the type of the probe is different from the type of the things in the container.**
  - *Can’t do this with binary_search algorithm, unfortunately.*
  - *But lower_bound algorithm has a form where you supply a comparison function that allows you to do a custom comparison:*
    - template<typename IT, typename T, typename Comparison>
      IT lower_bound(IT first, IT last, const T& value, Comparison comp);
    - For homogenous lookup, IT is the iterator type, T is normally the type of objects in the vector, and comp takes arguments of type const T&.

- *But fine print in the Standard says how the Comparison function can be used for heterogenous lookup:*
  - Has a first argument that matches the type of objects in the vector (what you get when you dereference an iterator)
  - Has a second argument that matches type T in the above template declaration - it can be either the same type as the objects in the vector, or a different type.
  - The comparison function returns true if the first argument comes before the second in a way consistent with the container ordering.
- *Example: Using a Thing type that has string name*
  - bool compare_Thing_to_string(const Thing& t, const string& s)
    {return t.get_name() < s;}

    while(true) {
    cout << "Enter name: ";
    string str;
    cin >> str;
    auto it = lower_bound(things.begin(), things.end(), str, compare_Thing_to_string);
    if(it != things.end() && it->get_name() == str)
    cout << "found: " << *it << endl;
    else
    cout << "not found" << endl;
    }

Uniform initialization syntax with { }

A great usability improvement

With structs and build-in arrays, we've always been able to initialize contents with { }

- int a[5] = {1, 2, 3, 4, 5};

```
struct S {int i; char c; double d;};

struct S s = {42, 'x', 3.14};  // in C
S s = {42, 'x', 3.14};  // in C++

// combine them in array of structure types
S sary[3] = {{1, 'a', 1.1}, {2, 'b', 2.2}, {3, 'c', 3.3} };
```

But with vector<> and other containers had to call member functions to do initializations in most cases

- vector<int> v;
  v.push_back(1);
  v.push_back(2);
  v.push_back(3);
  v.push_back(4);
  v.push_back(5);
  // ugh!

in C++11 Std. Lib. containers support initialization with { } syntax!

- vector<int> v = {1, 2, 3, 4, 5}; // hooray!

Even for map container:

- map<int, string> m = { {1, "hello"}, {2, "there"}, {3, "world"}, {4, "!!!"} };

- *Other uses of { } uniform initialization elsewhere, but not as important*
- *But take care here - some complex container constructors can be tricky …*

array<type, size>

Provides an interface a lot like vector<>, but wrapped around a fixed-size array

- *Concept: very fast - no memory allocation/deallocation needed*
- *no size overhead - only member variable is the array, size is available as a constant in the code*

trivial copy/assign - note move is identical to copy since shallow copy works correctly

- #include <array>
  using namespace std;

```
array<int, 5> a_5i;  // has sizeof 5 * sizeof(int)
array<double, 10> d_10;  // has sizeof 10 * sizeof(double)
```

- **operator[ ] is unchecked, at() member function is checked (like vector:)**
- **no push functions, no pop functions, use subscript, at, or iterators**
- **Concept - template parameter can be a built-in type with a value used in the code**
  - *sketch of class template*
    - template <typename T, int N>
      class array {
      public:
      int size() const {return N;}
      private:
      T elements[N];

      };

      array<int, 5> ary; gets instantiated as:
      public:
      int size() const {return 5;}
      private:
      int elements[5];
  - *All the advantages of both a built-in array and a vector. Good for memory limited, high-speed applications like embedded systems.*

deque

- **a complicated container that combines some features of both lists and vectors**
  - *basically two layers - a map array of pointers into blocks of memory --*
  - *last block filled from front to back*
  - *first block filled from back to front*
  - *so works like vector's push_back at both ends*
- **iterators more complicated than vector or list**
  - *e.g. ++ operation must determine if we are at the end of a block; if so, check the map to find the next block, and point to the first item in it*
- **relatively fast operations on each end - both push_back and push_front**
  - *not as fast as a list, but a lot better at front than array/vector would be*
- **relatively fast subscript access of individual elements**
  - *not as fast as an array or vector, but a lot better than a list would be*

stacks, queue, priority_queue

- **adapters - wrapper around a vector or other container**
- **push to put an item into container**

- **pop'ing removes the top element, but doesn't return a value. First, look at top using top(), front(), or back(), and then pop**
- **check using empty before looking at the top, front, or back value.**
- **give basic interface, but do not allow access to stuff "in the middle" - no iterators.**
  - *so not always suitable*
  - *can use push/pop front/back with vector or list to get your own stack or queue*
- **priority_queue - elements with the same priority do not have a defined order**
  - *typically uses a heap algorithm on a vector - also available to you*

Associative containers -

- **“associative” isn’t the best word to use here … “ordered” or “tree” is better - these containers always keep their items in order, and use a tree representation -**
- **set is actually the simplest, most basic**
  - *interface similar to our Ordered_list*
  - *a binary tree of elements ordered using < or a ordering you supply.*
    - set<int> si; // in order of size
    - set<char*> scl; // in order of address!
      - see note about operator< on char * doesn't do what you think it will
        - puts in address order, not order of pointed-to C-strings
    - can specify custom ordering with second template parameter:
      - For example, to set up a set container of Thing* with a custom ordering specified with a function object:

```
// order Thing pointers by the ID numbers of the Things
struct Less_Thing_ptrs {
        bool operator() (const Thing* tp1, const Thing* tp2) const
                {return tp1->get_ID_number() < tp2->get_ID_number();}
};

set<Thing* Less_Thing_ptrs> things;
```

  - *insert(x) puts it into the tree, self balances as needed, returns what happened.*
    - if already there, item is *not* inserted
    - returns std::pair<iterator-type, bool> where
      second is true if insertion succeeded, and iterator points to it;
      false if already there, and iterator points to where it is.
    - often don't care - just try to insert them all, at the end you have exactly one of each

- allows you to easily create the unique "set" of a bunch of items.
- Example: read a bunch of “words” from a file and get the unique set of them:

```
set<string> words;
ifstream input(“input.txt”);

// read and insert each white-space delimited string
string word;
while(input >> word) {
        words.insert(word);
        }

// how many different strings were found?
cout << words.size() << “ different words found” << endl;

// output them in alphabetical order
for (string& word : words) {
        cout << word << endl;
        }
```

- *.begin() to .end() gives them in order*

▼ *find(x) returns an iterator pointing to x if it is present, .end() if not.*

- does binary search through the tree

▼ *Homogenous lookup: in order to find an object in the tree, you have to construct a probe object that compares the same as the object you want.*

- note that x doesn't have to be a complete object, only the part used in the comparison.
- can be clumsy, or neat, depending on the nature of the object.
-

```
class Thing {
      string name;
      int cost;
      public:
      Thing(const string& name_) : name(name_) : cost(0) {}
      void set_cost(int cost_) {cost = cost_;}
      bool operator< (const Thing& rhs) const
           {return name < rhs.name;}
      };

set<Thing> things;
Thing t1("gizmo");
things.insert(t1);  // put it in
...
string v;
cin >> v;
Thing probe(v);
set<Thing>::iterator it = things.find(probe);
```

- *Heterogenous lookup - can be done, but a bit cryptic, and more than one way to do it. A C++14 extension, not very well known. See the handout.*

▼ **A gotcha for set containers**

▼ *objects in the container are supposed to be unmodifiable*

- required to ensure that you can't disorder the container by changing one of the objects in it so that the ordering by key field is no longer valid
  - e.g. for set<Thing>, consider iter->set_name("dohickey");
    - container might now be corrupted
  - If pointers in the container, the pointers are unmodifiable, but not the pointed-to object
    - e.g. for set<Thing *>
    - you can't change which object is pointed to by a item in the container, but you can change that object!
    - compiler won't warn you if you disorder the container by changing the Thing's key fields!
- How is unmodifiability implemented?
  - set::iterator and set::const_iterator behave the same way - you are supposed to get a compiler error if you try to modify an item in a set with a set::iterator, just like for a set::const_iterator
    - some early incorrect implementations turned set<Thing> into set<const Thing>
  - Recommendation: don't put const in a set declaration just to maintain the set ordering - it is supposed to take care of that for you. Sticking in an unnecessary const can make your code clumsy trying to preserve the const correctness that this requires.
    - don't declare a set<const T>
    - if pointers to unmodifiable objects, declare set<const T *> and always use const T * consistently throughout the program,
    - if pointers to modifiable objects, declare set<T *> and make sure your code doesn't modify the key field in an object being pointed-to from the container.
      - Preferably, don't provide a way to modify the key field once it is set.
- *What if you need to modify the key value of an object in the container?*
  - Only one workable approach: Make a copy of it, remove it from the container, change the copy, and put it back in - now it will be in the correct order.
- *what if you need to change some non-key part of the object?*
  - like the cost in Thing doesn't change ordering
  - three approaches:
    - Get a reference or pointer and do a const_cast to temporarily remove the constness, change the object through the reference or pointer
      - In general, avoid using a cast if at all possible. Do this only if absolutely unavoidable.
    - Do the same as if you were changing the key value: Copy the object from the container, remove it from the container, change the copy, and put the changed one into the container.
      - not so good if the object is really expensive to copy.

- Use a container of non-const pointers to the objects
  - get the pointer, change the non-key fields
  - can be safe if the object's class makes is key field immutable (like Thing's name)

- **map container - a set whose elements are pair<> objects**
  - *declare with 2 or 3 template type arguments:*
    - map<key-type, mapped-value-type, key-ordering-relation>
    - third parameter default to operator< of the key-type
    - example:
    - map<string, int> name_map; // strings in order
    - struct Reverse_string_order {
      bool operator() (const string& s1, const string& s2) const
      {return s1 > s2; //reverse order!}
      };

      map<string, int, Reverse_string_order> reverse_name_map;
  - *The map container compares the keys in the pairs to do the ordering,otherwise just uses the same code as set does (or map and set are two different interfaces to an underlying red-black tree container.)*
  - *pair is a template struct with members .first, .second*
    - e.g.
      ```
      template <typename T1, typename T2>
      struct pair {
           pair(T1 first_, T2 second_) : first(first_), second(second_) {}
           T1 first;
           T2 second;
      };
      ```
    - often handy to use for your own purposes
    - pair<string, int> p ("hello", 23);
    - p.first is string containing hello
    - p.second is int 23
    - make_pair is a function template that infers the types
    - string s; int i;
    - make_pair(s, i) returns a pair<string, int> containing a copy of s and i
      - an example of a function template being used to construct an object from a class template
      - e.g.
        - ```
          template <typename T1, typename T2>
          pair<T1, T2> make_pair(T1 first_, T2 second_)
          {
               return pair<T1, T2>(first_, second_)
          }
          ```
          ```

- *the pair used in a map is pair<const key_type, mapped_type>.*
  - can't change the key!
  - common error: forgetting the const when you declare the iterator or a pair type to use with a map container
  - can declare the iterator with auto or:
  - standard container typedefs/type aliases are your friends to help avoid this error, and save lots of typing
    - `map<string, Thing>::key_type ... string`
    - `map<string, Thing>::mapped_type ... Thing`
    - `map<string, Thing>::value_type ... pair<const string, Thing>`
    - your own typedefs (or type aliases)
      - `typedef map<string, Thing> Thing_map_t;`
      - `e.g. Thing_map_t::iterator it = my_thing_map.begin();`
      - `auto it = my_thing_map.begin();`
      - e.g.
```
void print_second (Thing_map_t::value_type& the_pair)
{
     cout << the_pair.second << endl;
}
```

- **Two ways of accessing map elements**
  - *insert/find/erase - general purpose, but often awkward because you have to work with the pair<> that is there*
    - concept: the iterator returned by the find function points to the pair in the container - you can access either the first or the second of the pair
    - insert(const value_type& v) returns a pair<iterator_type, bool>
      - pair<const key_type, value_type> x(key, value);
      - pair<iterator_type, bool> ret = m.insert(x);
      - pair<iterator_type, bool> ret = m.insert(make_pair(key, value));
      - the bool tells you whether the insertion was successful, the iterator tells you where the new pair or existing pair is in the tree.
      - if insert succeeded, the iterator points to the new pair in the map - not particularly useful, but there it is.
        - e.g.
          - pair<const string, Thing> my_pair(s, t);
          - pair<map<string, Thing>::iterator_type, bool> result = things.insert(my_pair);

- result.second is true if insert worked, false if not because a pair with the same key was already there.
- if succeeded, the iterator points to where it is in the map.

▼ insertion fails if key was already there - returned pair.second will be false

- if need to put it in, call erase with the iterator (ret.first), then insert again

```
if (!ret.second) {
     m.erase(ret.first);
     m.insert(my_pair);
```

- or change the second using the iterator - key is const, but not the mapped value

```
if (!ret.second) {
     ret.second->second = my_mapped_value;
```

▼ also have emplace - provide the two arguments for the pair:

- m.emplace(keyvalue, mappedvalue);

▼ find(key) returns an iterator that points to the pair (cf. set)

- string name;
  cin >> name;
  auto it = things.find(name);
- == .end() if not there.

▼ pointed-to pair.first is the key of the pair

- it->first

▼ pointed-to pair.second is the value of the pair.

- it->second

- erase(const_iterator) will remove the pair pointed to by the supplied iterator.
- erase(key_type) will remove the pair with the specified key, if present

▼ *subscript - operator[]*

- m[key] is a **reference** to the mapped-value (the second) in the pair
- subscript operator is convenient in many cases, but it is subtle - need to understand it!

▼ subscript works by first calling **insert** with a pair whose first (the key) is the subscript value and whose second is the default ctor'd value, and then returning a reference to the second of whatever the returned iterator is pointing to:

- if a pair with that key was already there, the insert didn't happen, but the iterator points to the pair that was already there, so you get a reference to the value (second) that is already there
- if a pair with that key wasn't already there, the **insert** happens, and you get a reference to the value (second) that is now there
- **either way, the returned reference to the second gives you a way to either read it or write it**

- tricky fact: you can't use subscript operator on a const map - because it might add a pair, the map has to be modifiable!

▼ more: if pair<key, v> not there, it is put in, with value being default value

- for built-in types, the appropriate type of zero
- for user-defined types, default ctor'd value - must have default ctor.

- so if subscript on left-hand side, we end up creating the pair, searching, inserting, then changing what was inserted - less efficient than simply inserting; how much so depends on the nature of the second of the pair.
- so you can find out if key was already present by testing for default value - but only if the default value could not be an actual value.

▼ Subscript operator is not the most efficient choice for looking things up, because any keys used in the lookup that aren't there will be added!

- can't remove any keys, or keep them out, using just subscript operator!
- so can pollute the map with bogus keys if e.g. they are user supplied - can take more time to clean it up!

▼ *Recommendation on using subscripts:*

▼ put it in with subscript on lhs if you want that key, value pair to be there unconditionally

- `m[key] = value;`

▼ Fastest insertion is calling insert with make_pair - but it works only if key is not already there!

- `m.insert(make_pair<key, v>);`

▼ Don't use on right-hand side to "read out" the value unless you are sure it is already there, or you don't mind the map being updated with the default value for the key!

- `my_value = m[key]; // adds pair with default value if not there`

▼ Don't double search just so you can use a subscript … pointless! - use find, check iterator, get the second to get the value

- Bad:

```
if(m.find(key) == m.end())
     // not there
else
     my_value_type v = m[key]; // searches a second time!
```

- Also Bad:

```
if(m.count(key) == 0) // count() does a search!
     not there
else
     my_value_type v = m[key]; // searches a second time!
```

- Good:

```
auto it = m.find(key); // search once
if(it == m.end())
     // not there
else
     my_value_type v = it->second;
```

cute example using map<string, command_function_ptr>

instead of bunch of if-elses or a switch, use a map to translate input command strings or codes to function pointers

pretty neat, but also a good exercise

- To get identifcal function pointer types, all command-handling functions must have same signature and return value - e.g. return void and have one argument: (Data& data)
- ```
  typedef void (*Command_fp_t)(Data&);
  typedef map<string, Command_fp_t> Command_map_t;

  void load_command_map(Command_map_t& cm)
       {
            cm["defrangulate"] = do_defrangulate_command;
            cm["transmogrify"] = do_transmogrify_command;
       }
  ```
- Or use brace initializiation and move semantics to be less verbose:

  ```
  Command_map_t load_command_map()
  {
       Command_map_t cm = {
            {“defrangulate”, do_defrangulate_command},
            {“transmogrify”, do_transmogrify_command}
            };
       return cm;
  }
  ```

inside the command handling loop, use the find function:

- ```
  // get the command from the user
  cin >> command;
  auto it = command_map.find(command);
  if(it == command_map.end())
       throw Error("Unrecognized command!");
  // get the function pointer
  command_fp_t cfp = it->second;
  cfp(data);  // call the command function
  ```

or use the subscript operator, test for default value, erase if wasn’t there - less efficient, but instructive on how the subscript operator works for a map container.

- ```
  // get the command from the user
  cin >> command;
  // get the function pointer with the subscript operator
  command_fp_t cfp = command_map[command];
  // it will be zero if the command is unrecognized
  // because zero is the default ctor'd value for a function pointer
  if(cfp)
       cfp(data);  // call the command function
  else {
       // remove the bad command
       command_map.erase(command);
       throw Error("Unrecognized command!");
       }
  ```

unordered containers

hashed containers have interfaces like set and map, with some additional member functions.
  ```

- *basic use is very easy if you know how to use set or map.*
- *default hash function supplied for built-in types, std::string, a few other library types.*
- *interface includes instrumentation and control functions to allow testing and tuning of the hashing.*
  - complete interface is much more elaborate than set and map.
- *Called unordered_set, unordered_map, etc. because if you iterate through the container, you will get all the elements just like with the other containers, but they come out in a strange order - depends on the hashing function - which is conceptually no meaningful order at all - so they are "unordered."*
  - Also, "hash" names were already in use for incompatible implementations.

- ***** Do not use unordered containers in this course! *****
  - *Hash containers useful only if great speed required, and lots of memory available, and memory demands do not slow down the application.*
  - *Not a good default choice - use only when justified and after testing to confirm performance.*
  - *To ensure really are fast, can require careful tuning - why they have an elaborate instrumentation interface - can see how well the hash is working, control the rehashing, etc.*
  - *Even if using strings, what if the strings are not random? How well will they hash?*
    - I have observed bad cases with my own hash functions …

Why .at() is not that useful

- **subscript operators are not range-checked!**
  - *analogous to subscripting a built-in array*
  - *not checked, but very fast!*
- **.at() member functions provide range checked access, throw a std::out_of_range() exception**
  - *the what() message is something like “range error”*
- **Problem: if you have a top-level catch for this exception, there is no useful information available about where or why the index was out of range!**
  - *it could have come from anywhere in the program!*
  - *A local try-catch is often pointless and clumsy*
- **Recommendation: Prefer your own range checks**
  - *if using subscripts on vector or array, check range yourself ahead of time, or use looping constructs that guarantee valid subscripts.*
  - *if using a map or set, use the find function and check the returned iterator, then dereference it only if it is != .end()*

- **operator[] vs. .at() member function**

- **as usual, safety means slow!**

Containers of containers

similar to Perl containers

- *but Perl syntax actually seems more obscure, at least to me ...*

a container can have other containers as a member

remember, the data in a container is dynamically allocated; the container object itself is not very big

- e.g.. your List and String class - only a couple of pointers, etc.

iterators point to things inside the container, making it possible to refer to things deep inside with no problem and no unnecessary data copying. Example:

```
typedef list<string> line_t;
typedef vector<line_t> paragraph_t;
typedef map<int, paragraph_t> document_t;

int main()
{
     document_t doc;



     // fill the document

// three different ways to output paragraph 23 line 4 of the document
// there shouldn't be any copying of the containers or data ...

     document_t::iterator it = doc.find(23);
     paragraph_t& para = it->second;
     line_t& line = para[4];
     for(line_t::iterator it = line.begin(); it != line.end(); it++)
          cout << *it << endl;


     line_t& line2 = doc[23][4];
     for(line_t::iterator it = line2.begin(); it != line2.end(); it++)
          cout << *it << endl;

     for(line_t::iterator it = doc[23][4].begin(); it != doc[23][4].end(); it++)
          cout << *it << endl;
}
```

vector<list<string>> paragraph;

- *e.g. each string is a word, each list is a line of text, the vector is the lines.*
- *typedefs and reference types can really clarify the code*

map<int <vector < list <string> >>>

- *the paragraphs are numbered, can be looked up by number*

How to pick a container

- **First, get clear on:**
  - *What items will be in the container?*
  - *What kind of order do they need to be in?*
  - *How are you going to put the items in the container?*
  - *How do you need to access the items in the container?*
  - *What algorithms will be used on the items?*
- **Use vector by default, with appropriate algorithms to access it.**
  - *e.g. use binary search if container items are in order*
- **Use sequence containers if**
  - *you want the objects to be in order of when they were put in, or some other arbitrary order*
  - *you want to use std lib algorithms that require a sequence container (many of them).*
- **Use associative containers if**
  - *you need fast logarithmic lookup automatically*
    - Never use a linear algorithm if you can apply a logarithmic one easily!
  - *you want the objects to be always in some kind of sorted order*
    - either operator< or an ordering that you specify
    - e.g. for convenience in output
  - *Since they are node-based, can be handy if iterators need to stay valid while information added or removed.*
- **How to choose a sequence container**
  - *Use vector if you want something like an built-in array*
    - Has subscripting just as fast as a built-in array.
      - use at() function to throw an exception if index is out of bounds
    - Adding to the end will be efficient (with push_back).
    - safely control a for loop with subscripting by calling size() to get the number of elements really there
    - Don't use if need to add/remove at the front, or insert/remove in the middle very often.
      - Can do it, but it will be slow
    - Can use algorithms like binary search that depend on fast subscripting for efficiency.

- especially the standard algorithms
  - binary_search, lower_bound
  - Never use a linear algorithm if you can apply a logarithmic one easily!
- Remember that iterators point to a cell in an internal array, not to a specific item, so can change their meaning or become invalid if other items are added or removed.
- a good default choice - can be expected to work very well in many situations

- *Use array instead of vector if the size is fixed and you still want to take advantage of STL interface*
- *Use a deque if you need vector-like capabilities but with operations at the front as well as the back*
  - Has subscripting, but slower than a built-in array
  - Slower than vector overall, but reasonably fast at both front and back modifications
  - Still stinks for modifications in the middle.
- *Use a list if you want to modify in the middle quickly, and don't mind linear operations elsewhere*
  - Can only do linear searches of the list
    - No way to quickly compute a subscript to go directly to the middle, as in binary search
    - Crazy, but true: the STL binary search algorithms will work on a linked list, but you don't want it - ridiculously inefficient - a glitch in the STL philosophy
      - does it by counting nodes in the list, then finding the middle node!
      - See handout!
  - Iterators to items remain valid if other items are moved around, inserted, or removed
    - iterator points to the node containing the item itself
  - Remember to use list's own member functions where provided, for speed
    - e.g. sorting.
  - note that linked-list is basically slower traversal than vector on most machine architectures most of the time!
- *Use adapters to be more expressive about the purpose of the container*
  - e.g. can get a stack just with push/pop_back, etc of a vector
  - use stack, queue, priority_queue - can tell immediately what it is used for, and how it will behave

- **How to choose associative containers**
  - *Use a map if you need to look values up from a key.*
    - Especially if the values are different type from the key.

- E.g. given a string, find the corresponding int

▼ disadvantages:

▼ if key is part of the value, some storage inefficiency

- e.g. a container of student records, which includes the student id number - the value
- key is student id number as a separate object
- but might still be worth it

- having to work with pairs is clumsy.

- Use a multimap if multiple values with the same key.

▼ *Use a set if you need to look up items where the key is either the stored object itself or part of the stored object.*

- Avoids map's storage inefficiency of duplicating the key, if key is part of the stored object
- Interface is MUCH simpler than map, so easier, more efficient to use.

▼ Especially if you want to "automatically" ensure that each item is represented in the container only once:

▼ set<int> - insert numbers in it, will contain each different number only once

- it is the "set" of integers that you processed - why it is called a "set"

▼ Disadvantage: to lookup an object with find(), you have to construct an object that matches the one you are looking for.

- E.g. to find a student record by ID, construct a dummy student record that contains the ID number.
- Possibly less efficient than using find on a map.
- No problem with build-in types - no construction time.

▼ Or use heterogenous lookup, which needs a more complicated comparisons

- See the handout

- Disadvantage that if you want to modify the objects in the set container, you get into const-correctness problems - avoid if possible.
- Use a multiset if you want multiple objects that compare alike.

▼ *Use a unordered_set or unordered_map if lookup speed is critical and you are willing to trade memory space to get it.*

- Note: sometimes using more memory slows things down - due to page faults, etc, in VM systems.
- Best approach: Write code so that you can easily switch containers, then test performance in realistic conditions.
- BUT DON”T USE IN THIS COURSE.

- **Build your own custom container only if necessary for efficiency.**
  - *Std. Lib. containers are very general - a custom container could suit a particular situation better.*
  - *BUT a custom container is likely only to be better if it takes advantage of something in the situation that the general container cannot.*
    - If you don't know what that is, might as well use the general container!
  - *Often, you can easily assemble a custom container by using a combination of the library containers.*
  - *Example - doing a hash container using previous STL containers*
  - *Example - double-ended map - lookup either with "key" or "value" and get the other one*