- **Library Organization and Standard Containers**
- **Stroustrup ch. 16**
- **Container highlights**
  - **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 COPIES of objects, and move them by assignment - so elements must have public*
      - copy ctor - usually needed if the container has an internal array of objects
      - assignment operator
      - destructor - if a class type
    - *But if a container has a class type for its contents, when that item is removed, the dtor is automatically run.*
    - *for some containers, algorithms, the objects in the container might also need:*
      - a default ctor
      - an equality operator
      - a less than operator
    - *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 int or delete thing 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.
  - **iterators as an abstracted pointer**
    - *act like pointers to elements in the container*
      - same concept as in Project 2
    - *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
    - *use like a pointer, but contains whatever, 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, because avoids have to special case the last member in an iteration, avoids having to define < or > for iterators*
    - *Notice UNCHECKED policy - why?*
        - operator[] vs. .at() member function
- **Two gotchas with iterators**
    - *Gotcha #1. If the container is const, then you must use a const_interator to access it*
        - through template magic, 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();
              }
              ```
    - *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.
- **The typedefs inside a container class template**
    - *- a way to access useful information about an instantiated container*
        - 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.*

- **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*
    - *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.

- **Stroustrup ch. 17**
- **comparison operation**
  - **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 469*
- **for pointers, have to define comparison functions or function objects**
  - **S p. 468**
  - **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 or function object that says how to order the pointers based on the pointed-to data**
- **Two basic container types**
  - **sequences**
    - *vector, list, deque*
    - *adapted to stack, queue, priority-queue*
  - **associative**
    - *map, set, multimap, multiset*
- **Two different underlying implementations:**
  - **array-based**
    - *vector, deque*
    - *contain an internal dynamically allocated array*
  - **node-based**
    - *list, map, set, multimap, multiset*
    - *allocate space an item at a time*
  - **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 vallue 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*

      -
        ```
        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.

    -
      ```
      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**
  - **no subscript, but add and remove at both ends**
    - *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.*
    - *associative containers are always sorted, so don't need to sort 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*
- **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 expensive copy/assignment of multiple objects**
  - **push_back is provided, but not push_front.**
  - **vector used the sort algorithms (can be quite efficient)**
  - **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 quickly**
- **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 vectors.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 or deque - also available to you*

- **Associative containers**
  - **set is actually the simplest, most basic**
    - *a binary tree of elements ordered using < or a ordering you supply.*
      - 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
    - *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.
      - typically don't care - put them all in, at the end you have exactly one of each
      - allows you to easily create the unique "set" of a bunch of items:
    - *.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
    - *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.
      - example:

        -
          ```
          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);
          ```
        - yuch, but that's the way it is
  - **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?
        - original Standard wasn't quite clear how it was supposed to be done; correction scheduled for the next Standard; some implementations (gcc) does it the "right" way.
        - The right way: 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
        - The wrong way - add const to the types of the object in the container.
            - In other words, a set<Thing> is NOT supposed to get turned into a set<const Thing>.
        - Recommendation: don't put const in a container 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 correctedness 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?*
    - One one acceptable 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
    - two 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 overwhelmingly desirable.
        - 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.
            - preferred unless the object is grossly expensive to copy.
            - Note that if container has pointers, then this approach should be fairly cheap.

- **map container - a set whose elements are pair<> objects**
  - *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 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
  - *see S p. 482*
  - *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
  - *standard container typedefs 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:
      - `typedef map<string, Thing> Thing_map_t;`
      - `e.g.  Thing_map_t::iterator it = my_thing_map.begin();`

      -
        ```
        e.g. void foo (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*
    - 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);
      - the bool tells you whether the insertion was successful, the iterator tells you where the new pair or existing pair is in the tree.

            - insertion fails if key was already there - returned pair.second will be false
                - if need to put it in, call erase(ret.first), then insert again

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

            - if succeeded, the iterator points to the new pair in the map - not particularly useful, but there it is.
            - e.g.
                - pair<const string, Thing> p(s, t);
                - pair<map<string, Thing>::iterator_type, bool> result = things.insert(p);
                - 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.
        - find(key) returns an iterator that points to the pair (cf. set)
            - == .end() if not there.
            - returned pair.first is the key of the pair
                - it->first
            - returned pair.second is the value of the pair.
                - it->second
        - erase(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 value (the second) in the pair
        - subscript works by calling insert with a pair whose first (the key) is the subscript value and whose second is the default ctor'd value, and then giving you a reference to the second of whatever the returned iterator is pointing to
            - if it 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 it wasn't already there, the insert happens, and you get a reference to the value (second) that is now there
            - either way, the referencee to the second gives you a way to either read it or write it
            - see Stroustrup p. 488 bottom
        - 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, depending on the nature of the second of the pair.
        - so can find out if key present by testing for default value - if the default value could not be an actual value.
        - Not a good choice for looking things up, because any keys used that aren't there will be added
            - can't remove any, or keep them out with just subscripts
            - so can pollute the map with bogus keys if e.g. they are user supplied

- *Best of both worlds - using simple subscripts without polluting the map*
  - put it in with subscripts if you want it to be there unconditionally
    - `m[key] = value;`
  - could check for present first using find - poor choice; slow, doing two searches, wasted default construction of second
    ```
    if m.find(key) == m.end()
         not there, don't put it in ...
    else
         my_value_type v = m[key];
    ```
  - or if default value means wasn't there, clean it up

    ```
    my_value_type v = m[key];
    if (v == my_value_type())
         erase(key);
    else
         // use v
    ```
- *Fastest and easiest insertion is calling insert with make_pair*
  - `m.insert(make_pair<key, v>);`

- **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;
             }
        ```
      - inside the command handling loop:

        -
          ```
               // get the command from the user
               cin >> command;
               // get the function pointer
               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!");
                    }
          ```

- **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; // note the space!**
  - *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*

- **TR1 provides hashed containers**
  - **TR1: Standard Library Technical Report No. 1: a set of additions to the C++ Standard Library that are slated to become part of the next Standard (hopefully this year, full implementations to follow).**
    - *gcc 4.x includes big parts of TR1 - can also download a version for MSVS.*
  - **hashed containers have interfaces like set and map, with some additional member functions.**
    - *See Handout on course website for an introduction to them.*
    - *very easy to use if you know how to use set or map.*
    - *default hash function supplied for a built-in types and std::string*
    - *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.*
  - **At this time, can't use in Project 3 - need to be thoroughly familiar with the current Standard Containers**

- **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 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 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!
    - 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.
  - *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.
    - 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 TR1::ordered_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 in realistic conditions.
- **Build your own custom container 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*