- **Lecture Outline - algorithmns and function objects - highlights**
  - **Stroustrup ch. 18 topics**
    - *basic features of std. lib. algorithms*
    - *Standard Library function objects the make it easier to use the algorithms in a variety of situations*
    - *Your own custom function objects for specialized situations*
  - **See code examples on course web site**
    - *Often, the easiest way to understand the specific algorithms and function objects is to see an example of how they are used.*
- **Std Lib algorithms are all defined as function templates that take iterators as arguments**
  - **Use only the iterator interface to work**
    - *`e.g. *, =, ++, --, ->, etc`*
  - **algorithms all work on a range specified by two iterators**
    - *a begin and an end;*
    - *often these are a container begin() and end(), but not always.*
      - verbose, but general
  - **this way the algorithms will work for ANY suitable container**
    - *restrictions based on what kind of iterator is suitable - e.g. some require random-access iterators (like subscripts or pointers), others do not.*
      - template system used to define iterator "traits" so that compiler can choose different instantiations depending on the types of the iterators.
        - a specialized topic in template programming ... not covered here.
    - *but algorithms will work on built-in arrays - you supply a pointer.*
      - pointers are recognized as a random access iterator!

- **Good way to understand is to look at an implementation**
  - **look at how the for_each algorithm is actually implemented (can differ in details)**

    -
      ```
      // for_each
      template<class InputIterator, class Function>
      inline
      Function
      for_each(InputIterator first, InputIterator last, Function f)
      {
            for (; first != last; ++first)
                  f(*first);
            return f;
      }
      ```
    - *Why is "f" returned - come back to later, when discuss function objects*
  - **Now what happens when the template is instantiated:**

    -
      ```
      // for_each
      template<class InputIterator, class Function>
      inline
      Function
      for_each(InputIterator first, InputIterator last, Function f)
      {
            for (; first != last; ++first)
                  f(*first);
            return f;
      }

      void print_int(int i)
      {
            cout << i << end;
      }


      list<int> int_list;
      // put some ints in the list
      for(int i = 0; i < 10; i++)
            int_list.push_back(i);

      for_each(int_list.begin(), int_list.end(), print_int);
      /* compiler instantiates the template with:
      typename InputIterator is list<int>::iterator
      typename Function is void(*)(int)  (because it knows the type of print_int)
      and creates the function: */
      for_each(list<int>::iterator first, list<int>::iterator last, void(*f)(int))
      {
            for (; first != last; ++first)
                  f(*first);
            return f;
      }

      // Calling this function has the same effect as if WE had written:
            list<int>::iterator first = int_list.begin();
            list<int>::iterator last = int_list.end();
            for(; first != last; ++first)
                  print_int(*first);
      ```

- **Some additional examples - most algorithms are pretty simple**
  -
    ```
    // find - note: operator== for the type must be defined
    template <class InputIterator, class T>
    inline
    InputIterator
    find(InputIterator first, InputIterator last, const T& value)
    {
    	while (first != last && !(*first == value))
    		++first;
    	return first;
    }

    // find_if - note: the Predicate type returns a bool (or a type convertible to bool)
    template <class InputIterator, class Predicate>
    inline
    InputIterator
    find_if(InputIterator first, InputIterator last, Predicate pred)
    {
    	while (first != last && !pred(*first))
    		++first;
    	return first;
    }

    // copy - note: dereferencing result iterator must always be well-defined
    template <class InputIterator, class OutputIterator>
    inline
    OutputIterator
    copy(InputIterator first, InputIterator last, OutputIterator result)
    {
    	for (; first != last; ++first, ++result)
    		*result = *first;
    	return result;
    }
    ```
- **But some algorithms are a lot more more subtle, or relieve a lot of tedium**
  - *sort , merge - several variations: partial_sort, partition*
  - *binary_search, lower_bound - do a binary search of a sorted sequence or tell you where a new item should be inserted*
  - *unique - removes duplicates*
  - *random_shuffle - randomly permute a sequence*
  - *next_permutation - start with a sorted sequence, generates each permutation, tells you when it is done*
- **vector vs. built-in array - iterator interface works with anything that behaves like an iterator - e.g. a pointer**
  -
    ```
    // illustrate how algorithms apply to both containers and built-in arrays

    #include <iostream>
    #include <vector>
    #include <algorithm>

    using namespace std;

    void print(int i)
    {
    ```

```
	cout << i << endl;
}

int main()
{
	vector<int> vi;
	int ai[10];

	for(int i = 0; i < 10; i++) {
		vi.push_back(i+1);
		ai[i] = i+1;
		}

	for_each(vi.begin(), vi.end(), print);

	for_each(ai, ai+10, print);
}
```

- **Consequence of iterator interface: an algorithm cannot directly insert or erase elements from the container!**
  - **In other words, an algorithm can not change the number of elements in the container!**
  - **These operations have to be done by a container member function!**
  - **Why there is no "insert" algorithm in the std lib algorithms**
  - **remove algorithm is easy to misunderstand - "removed" items moved to end of the sequence, and returns an iterator to the new "end"**
  - **copy algorithm is often a problem about this - has to be room at the destination**
    -

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

using namespace std;

void print(int i)
{
      cout << i << endl;
}

int main()
{
      vector<int> vi;
      int ai[10];

      for(int i = 0; i < 10; i++) {
            vi.push_back(10 * (i+1));
            ai[i] = i+1;
            }
      for_each(ai, ai+10, print);
      for_each(vi.begin(), vi.end(), print);

      // copy vi into ai - there is space for it
      copy(vi.begin(), vi.begin() + 4, ai);

      for_each(ai, ai+10, print);
      for_each(vi.begin(), vi.end(), print);

}
```

  - **Filling a container can't be done just with the regular iterators unless space has already been created at every possible iterator position**
    - *can create a vector of a certain size, then iterator can be used to fill those already-created places.*
    - *But some cute tricks available using function objects.*

- **Function objects**
  - **Definition: an object whose class overloads the function call operator: operator(), and so can be used like a function.**

    -
      ```
      class My_function_object_class {
      public:
            return-type operator() (parameter-list)
                  { do whatever any normal function would do}
      };

      My_function_object_class fo;      // an instance of the class

      x = fo(args);    // use like a function!
      ```
  - **Otherwise, just a class - can have other member variables and functions.**
    - *Especially: a constructor with parameters to provide values for the function call operator to use*
  - **A simple one often declared with struct to make all members public**
  - **Can use a function object like a function pointer, but easier: Just name the class! The compiler can then get all the information necessary to compile the call.**
    - *if object appears as a function call, compiler looks at the class's declaration for operator() to get the prototype information*
    - *so unlike function pointers, you don't have to worry about declaring them, or casting, to get function information into another function!*

      -
        ```
        void foo(char * s1, char * s2, int (*fp) (const char *, const char *) {
              int result = fp(s1, s2);
              ....

        void foo(char *s1, char * s2, Function_object_type fo) {
              int result = fo(s1, s2);
        ```
      - it will work if the result compiles!
  - **importance for STL algorithms is that they work equally well with function pointers and function objects**
    - `for_each(x.begin(), x.end(), func_ptr);`
    - `for_each(x.begin(), x.end(), func_object);` or
    - `for_each(x.begin(), x.end(), func_object_class_name() );` // create an unnamed object of the class
  - **Associative containers normally take a function object class name as an optional parameter to specify the ordering relation**
    - *won't accept a function pointer in this slot.*
      - If you want to use a function pointer, you have to specify the function pointer type in this slot, then provide the function pointer itself as a constructor parameter.
    - *defaults to less<T>, which is a Std. Lib. class template for a function object class that defines an operator() that applies T's operator< between two T objects*
    - *Examples:*
      - set<int> si; // set of ints in default operator < order

        struct RevInt { bool operator() (int i1, int i2) const {return i2 < i1;} // reverse order of integers

        set<int, Revint> sri; // set of ints in reverse order

        map<int, string, Revint> mri; // map of ints to strings, but with the ints in reverse order

- **REALLY BIG advantage over function pointers is that the object can have member variables and other member functions!**
  - *a lot more sophisticated than function pointers*
- **Another advantage: The function code is often inlined, meaning that code using a function object will often be faster than code doing an ordinary function call, or a call using a function pointer.**

- **Example 1 a function object with state**

  -
    ```
    #include <iostream>
    #include <vector>
    #include <algorithm>

    using namespace std;
    // a function object class that calculates a mean, accumulating every supplied value

    class Calc_Mean {
    public:
          Calc_Mean() : sum(0.), n(0) {}
          void operator() (double x)  // accumulate the supplied value
                {
                      n++;
                      sum += x;
                }
          double get_mean() const
                {return sum / double(n);}
          int get_n() const
                {return n;}
    private:
          double sum;
          int n;
    };

    // prototypes
    void test1();
    void test2();


    int main()
    {
          // test1();
          test2();
    }

    void test1()
    {
          cout << "Enter a bunch of values, or EOF when done:" << endl;

          double x;
          Calc_Mean cm;

          while (cin >> x)
                cm(x);     // use like an ordinary function

          cout << endl;
          cout << "mean of " << cm.get_n() << " values is " << cm.get_mean() << endl;
    }

    void test2()
    {
          cout << "Enter a bunch of values, or EOF when done:" << endl;
          vector<double> data;
          double x;
          while (cin >> x)
                data.push_back(x);
    ```

```
		// use with an algorithm
		Calc_Mean cm = for_each(data.begin(), data.end(), Calc_Mean());

		cout << endl;
		cout << "mean of " << cm.get_n() << " values is " << cm.get_mean() << endl;
	}
```

- **Example 2 a function object with state including an initial value**

  -
    ```
    #include <iostream>
    #include <vector>
    #include <algorithm>

    using namespace std;

    // a function object class that calculates a mean, accumulating every supplied value,
    // but it takes an optional baseline value when initialized that is subtracted
    // from every value
    class Calc_Mean {
    public:
         Calc_Mean(double in_baseline = 0.) : sum(0.), n(0), baseline(in_baseline) {}
         void operator() (double x)  // accumulate the supplied value
              {
                   n++;
                   x = x - baseline;
                   sum += x;
              }
         double get_mean() const
              {return sum / double(n);}
         int get_n() const
              {return n;}
    private:
         double sum;
         int n;
         double baseline;
    };

    // prototypes
    void test1();
    void test2();


    int main()
    {
         // test1();
         test2();
    }


    void test1()
    {
         double b;
         cout << "Enter baseline value:";
         cin >> b;  // no error check
         cout << "Enter a bunch of values, ^D (EOF) when done:" << endl;

         Calc_Mean cm(b);

         double x;
         while (cin >> x)
              cm(x);     // use like an ordinary function
         cout << endl;

         cout << "mean of " << cm.get_n() << " values is " << cm.get_mean() << endl;
    }
    ```

```
void test2()
{
	double b;
	cout << "Enter baseline value:";
	cin >> b;  // no error check

	cout << "Enter a bunch of values, ^D (EOF) when done:" << endl;
	vector<double> data;
	double x;
	while (cin >> x)
		data.push_back(x);

	// use with an algorithm
	Calc_Mean cm = for_each(data.begin(), data.end(), Calc_Mean(b));
	cout << endl;
	cout << "mean of " << cm.get_n() << " values is " << cm.get_mean() << endl;
}

/* Output
Enter baseline value: 10
Enter a bunch of values, ^D (EOF) when done:
10 20 30

mean of 3 values is 10
*/
```

- **Function objects can be templated**
  - **Magic Trick #1: Use a function template to instantiate the function object template using the function parameters:**

    -
      ```
      // demonstration of a basic template magic trick:
      // using a function template to create a function object from a template,
      // with the function arguments specifying which template to instantiate


      #include <iostream>
      #include <string>
      #include <vector>

      using namespace std;

      // a function object class that accumulates and prints the "sum" of its argments
      // the template parameter is the type of the input.
      template<typename T>
      class Summer {
      public:
           Summer(const T& initial_value) : sum(initial_value) {}
           void operator() (const T& x)
                {
                     sum += x;
                     cout << sum << endl;
                }
      private:
           T sum;
      };

      // a function template that has the compiler deduce how to instantiate the
      // function object class
      template <typename X>
      Summer<X> make_Summer(const X& x)
      {
           return Summer<X>(x);
      }

      int main ()
      {
           vector<int> vi;
           for(int i = 0; i < 4; i++)
                vi.push_back(i+1);
           vector<string> vs;
           vs.push_back("Now ");
           vs.push_back("is ");
           vs.push_back("the ");
           vs.push_back("time.");
           string start("OK! ");

           // instantiate the function object class directly
           for_each(vi.begin(), vi.end(), Summer<int>(10) );
           for_each(vs.begin(), vs.end(), Summer<string>(start) );

           // instantiate the function object class using the supplied parameter type
           for_each(vi.begin(), vi.end(), make_Summer(10) );
           for_each(vs.begin(), vs.end(), make_Summer(start) );
      }
      ```

```
/* output:
11
13
16
20
OK! Now
OK! Now is
OK! Now is the
OK! Now is the time.
11
13
16
20
OK! Now
OK! Now is
OK! Now is the
OK! Now is the time.
*/
```

- **Magic Trick #2: Often handy to use a template member function!**
- **A simple example of the concept - a handy function object class for deleting pointers in a container - note that you can't call "delete" as a function - it is an operator in the language, not a function!**
  - `// A straightforward way to do it with a class template`

```
template<typename T>
class Delete {
void operator() (const T* ptr) const
      {
            delete ptr;
      }
};

for_each(ptrs.begin(), ptrs.end(), Delete<Thing>());

instantiates into
class Delete {
void operator() (const Thing * ptr) const
      {
            delete ptr;
      }
};

// expands into: (pseudocode)

void Delete<Thing>::operator() (const Thing * ptr) {delete ptr;}  // a_Delete_object

for_each(list<Thing *>::iterator first, list<Thing *>::iterator last, a_Delete_object)
{
      for (; first != last; ++first)
            a_Delete_object(*first);
      return a_Delete_object;
}

// the loop expands/inlines then into:

      for (; first != last; ++first)
            delete *first;

// but Delete<Thing>(); sure is clumsy!
// Use a member function template

/* See Scott Meyers, Effective STL, Item 7 for a discussion
of this general purpose Function Object class - will work for deleting any pointer */

struct Delete {
template<typename T>
void operator() (const T* ptr) const
      {
            delete ptr;
      }
};
// The class has a member function template - the compiler can deduce the type T from
// the call.

// note that you can't call delete as a function! It is an operator!
// list<Thing *> ptrs is a container of pointers to objects
```

```
for_each(ptrs.begin(), ptrs.end(), Delete());     // is that easy, or what?

// expands first into: (pseudocode)

template <typename T> void Delete::operator() (const T * ptr) {delete ptr;} // a_Delete_object

// possibly other overloaded function-call operators!

for_each(list<Thing *>::iterator first, list<Thing *>::iterator last, a_Delete_object)
{
     for (; first != last; ++first)
          a_Delete_object(*first);
     return a_Delete_object;
}

// the loop expands/inlines then into:

     for (; first != last; ++first)
          delete *first;
```

- **Insert iterators**
  - **Cute template tricks: Adapters that wrap an iterator interface around some other operation.**
    - **e.g. insert iterators - wrap an iterator interface around a function like push_back**
    -
```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

void print(int i)
{
      cout << i << endl;
}

int main()
{
      vector<int> vi;
      int ai[10];

      for(int i = 0; i < 10; i++) {
            ai[i] = i+1;
            }
      for_each(ai, ai+10, print);

      // copy ai into vi, making space as needed
      copy(ai, ai + 10, back_inserter(vi));

      for_each(ai, ai+10, print);
      for_each(vi.begin(), vi.end(), print);
}
```
    - *back_inserter defines operator= so that the expression in copy: *result = *first gets turned into the call: vi.push_back(*first);*
      - in tne function object class back_insert_iterator (instantiated by back_inserter template function:

```
template <class Container>
inline
back_insert_iterator<Container>&
back_insert_iterator<Container>::operator=(typename Container::const_reference value)
{
      container->push_back(value);
      return *this;
}
```
  - **other inserters for other situations - like filling a set - see Stroustrup or Josuittis**

- **stream iterators**
  - **idea is to allow a stream to be used as a source or destination in an algorithm, by giving the stream an iterator interface. Can be real handy - turns an i/O loop into a one-liner**
    - *turns iterator dereference into an input operator call, assignment into an output operator call*
  - **example of output stream iterator**

    -
      ```
      vector<int> stuff;
      /* put a bunch of ints into stuff with e.g. push_back */

      // create an output stream iterator - the stream followed by a delimiter character
      ostream_iterator<int> outiter(cout, ":")
      // write the integers with a ':' after each one
      copy(stuff.begin(), stuff.end(), outiter);
      cout << endl;

      // write the integers one per line
      copy(stuff.begin(), stuff.end(), ostream_iterator<int>(cout, "\n"));
      ```
  - **example of input stream iterator - note how the "end" is done.**

    -
      ```
      ifstream input_file;
      // open the input file

      vector<int> stuff;
      // read a bunch of ints until end of file, fill the vector
      istream_iterator<int> initer(input_file);
      // default ctor'd stream input iterator works  as end of file "end" value.
      istream_iterator<int> eofiter;

      copy(initer, eofiter, back_inserter(stuff));

      // as a one-liner
      copy(istream_iterator<int>(input_file), istream_iterator<int>(), back_inserter(stuff));
      ```

- **Using algorithms and adapters with ordinary functions**
  - **Algorithm doesn't care whether Function parameter is a function pointer or function object**
  - **The algorithm calls your function with the dereferenced iterator as the only argument - what if you need more arguments? What if the first argument is not the dereferenced iterator?**
    - *Calling an ordinary function that has two arguments - e.g. first is the dereferenced iterator, the second is something else.*
  - **No place in the for_each or other algorithms to supply the second parameter:**
    - *Could have done:*
      - `for_each_with_1_additional_arg(my_list.begin(), my_list.end(), my_function, 42)`
      - `f(*first, function_second_arg)`
    - *Instead, keep the single "slot" for the function, but use function objects to get that second argument in there.*
  - **This is what the adapters and binders are for:**
    - *Create a function object whose constructor parameter saves a pointer to the function and the supplied parameter values in member variables, and whose function call operator takes the dereferenced iterator as a parameter, and calls the function through the stored pointer supplying both the dereferenced iterator and the stored value as parameters.*
    - *semipseudocode:*

      -
        ```
        class MyBinder2nd {
        public:
                Binder(function_pointer_type  function_pointer, parameter_type parameter_value) :
                        saved_function_pointer(function_pointer), saved_parameter_value(parameter_value)
                        {}
                void operator() (derferenced_iterator_type x)
                        {saved_function_pointer(x, saved_argument_value);} // used saved 2nd parameter
        function_pointer_type  saved_function_pointer;
        parameter_type saved_parameter_value;
        ```
    - *The adapters typically consist of a function template and a function object class template*
      - The function template creates and returns a function object.
      - The compiler deduces the argument types for the function template and then specifies the types for the class to be used in instantiating the function object class.
    - *To allow them to be combined with nesting, each function object class defines some Standard-specified typedefs - makes them "adaptable"*
      - The outer function object template can "look inside" the inner object to see what the types are.
      - Just creates typedefs for the types of first, second, and result (the returned value)
    - *See handouts on web site for gruesome details*
  - **First, need an adapter to create a function pointer function object for use by the binder - ptr_fun**
    - *can use it by itself, but not needed by itself - the function name is a function pointer*
    - *identifies return type and parameter types of the function with typedefs that the binder template can use*
    - *creates a function object that stores the function pointer internally and defines operator() to call using that pointer*
  - **More than one ordinary argument**
    - *A binder creates a function object that contains the extra argument which is then passed along with the dereferenced iterator into the function.*

- *bind1st, bind2d "bind" the first or second argument to a value you supply, leaving the other one free to accept the dereferenced iterator.*
- *Confusing: for both of them, the function pointer adapter is the first parameter, and the binding value is the second.*

- **Examples of usage of some standard adapters**

  -
    ```
    #include <iostream>
    #include <list>
    #include <string>
    #include <algorithm>
    #include <functional>

    using namespace std;

    class Thing {
    public:
         Thing(int in_i) : i(in_i) {}
         void print() const
              {cout << "Thing " << i << endl;}
         void update()
              {i++;}
         void set_value(int in_i)
              {i = in_i;}
         int get_value() const
              {return i;}
    private:
         int i;
    };

    // write it a line by itself
    ostream& operator<< (ostream& os, const Thing& t)
    {
         os << "Thing " << t.get_value() << endl;
         return os;
    }


    /* The functions below do not take reference type parameters because the adapters
    and binders end up creating "references to references" (& &) which are currently illegal in
    the
    language, which turns out to be an extreme inconvenience that hopefully will be fixed in the
    next Standard (which is being worked on right now). The fix will be that a reference to a
    reference
    is simply a reference. Some compilers (like my current favorite) are ahead of the others in
    already
    doing this, but it is not in fact standard. */

    void print_Thing(Thing t)
    {
         t.print();
    }

    void print_int_Thing(int i, Thing t)
    {
         cout << i << ": " << t;
    }

    void print_Thing_int(Thing t, int i)
    {
         cout << i << ": " << t;
    }

    void print_x_Thing(int i, Thing t)
    ```

```
{
	cout << i << ": " << t;
}

void print_x_Thing(string s, Thing t)
{
	cout << s << ": " << t;
}

int main()
{
	Thing t1(1), t2(2), t3(3);
	list<Thing> obj_list;
	obj_list.push_back(t1);     // a copy
	obj_list.push_back(t2);
	obj_list.push_back(t3);

	// try setting the value using an adapter to pass an argument

	cout << "Output from applying an ordinary function using obj_list" << endl;
	for_each(obj_list.begin(), obj_list.end(), print_Thing);

	cout << "Output from applying an ordinary function with ptr_fun using obj_list" << endl;
	for_each(obj_list.begin(), obj_list.end(), ptr_fun(print_Thing));

	// using binders with ordinary functions
	int new_value = 42;
	cout << "Output from applying a bind2nd adapter on an ordinary function" << endl;
	for_each(obj_list.begin(), obj_list.end(), bind2nd(ptr_fun(print_Thing_int), new_value));
	cout << "Output from applying a bind1st adapter on an ordinary function" << endl;
	for_each(obj_list.begin(), obj_list.end(), bind1st(ptr_fun(print_int_Thing), new_value));

	// using overloaded functions requires a cast to select the right function
	cout << "Output from applying a bind1st adapter on an overloaded function" << endl;
	for_each(obj_list.begin(), obj_list.end(), bind1st(ptr_fun((void (*)(int, 
Thing))print_x_Thing), new_value));
	cout << "Output from applying a bind1st adapter on another overloaded function" << endl;
	for_each(obj_list.begin(), obj_list.end(), bind1st(ptr_fun((void (*)(string, 
Thing))print_x_Thing), string("Hello")));

	/* following is illegal under current Standard because the ostream has to be passed in by 
reference */
	// for_each(obj_list.begin(), obj_list.end(), bind1st(ptr_fun((ostream& (*)(ostream&, 
const Thing&))operator<< ), cout));


	cout << "done!" << endl;
}
```

- **Algorithms and member functions**
  - **The situation:**
    - *You have a container of objects or pointers to objects.*
    - *You use an algorithm to iterate over the container.*
    - *The dereferenced iterator is an object or pointer to the object*
    - *You want to call a member function of the object.*
  - **The situation is different beause the first function parameter is the (hidden) "this" pointer.**
    - *The call can't be f(the object) or f(the pointer), but has to be theobject.f() or thepointer->f();*
    - *A member function adapter does this - different flavors depending on whether the dereferenced iterator is an object reference or an object pointer.*
    - *But like ordinary functions, the first step is a function pointer, but it is a pointer to member function, which is different from an ordinary function pointer.*
    - *See the handout for a summary of what is presented here.*
  - **Pointer-to-member-functions**
    - *Pointers to member functions are not like regular pointers to functions, because member functions have a hidden "this" parameter as the first parameter, and so can only be called if you supply an object to play the role of "this", and use some special syntax to tell the compiler to set up the call using the hidden “this” parameter.*
    - *Declaring pointers-to-member-functions*
      - You declare a pointer-to-member-function just like a pointer-to-function, except that the syntax is a tad different: it looks like the verbose form of ordinary function pointers, and you qualify the pointer name with the class name, using some syntax that looks like a combination of scope qualifier and pointer.
      - Declaring a pointer to an ordinary function:
        - `return_type (*pointer_name) (parameter types)`
      - Declaring a pointer to a member function:
        - `return_type (class_name::*pointer_name) (parameter types)`
          - The odd-looking "::*" is correct.
    - *Setting a pointer-to-member-function*
      - You set a pointer-to-member-function variable by assigning it to the address of the class-qualified function name, similar to an ordinary function pointer.
      - Setting an ordinary function pointer to point to a function:
        - `pointer_name = function_name;     // simple form`
        - `pointer_name = &function_name;    // verbose form`
      - Setting a member function pointer to point to a member function:
        - `pointer_name = &class_name::member_function_name;`
    - *Using a pointer-to-member-function to call a function*
      - You call a function with a pointer-to-member-function with special syntax in which you supply the object or a pointer to the object that you want the member function to work on. The syntax looks like you are preceding the dereferenced pointer with an object member selection (the “dot “operator) or object pointer selection (the “arrow” operator).
      - Calling an ordinary function using a pointer to ordinary function:
        - `pointer_name(arguments); // short form, allowed`

- or
- `(*pointer_name)(arguments);  // the more verbose form`
- Calling the member function on an object using a pointer-to-member-function
  - `(object.*pointer_name)(arguments);`
  - or calling with a pointer to the object
  - `(object_ptr->*pointer_name)(arguments);`
  - Again, the odd looking things are correct: ".*" and "->*". The parentheses around the whole pointer-to-member construction are required because of the operator precedences.
- *Calling a member function from another member function using pointer to member*
  - This seems confusing but actually is just an application of the pointer to member syntax with “this” object playing the role of the hidden this parameter. If you want a member function f of Class A to call another member function g of class A through a pointer to member function, it would look like this:

    -
      ```
      class A {
            void f();
            void g();
      };


      void A::f()
      {
            // declare pmf as pointer to A member function,
            // taking no args and returning void
            void (A::*pmf)();
            // set pmf to point to A's member function g
            pmf = &A::g;

            // call the member function pointed to by pmf points on this object
            (this->*pmf)();  // calls A::g on this object
      }

      // using a typedef to preserve sanity - same as above with typedef

      // A_pmf_t is a pointer-to-member-function of class A
      typedef void (A::*A_pmf_t)();

      void A::f()
      {
            A_pmf_t p = &A::g;

            (this->*p)();    // calls A::g on this object
      }
      ```

- **Using algorithms to call member functions**
  - **Need to use an adapter to do the call with a pointer to member and the "this" parameter**
    - *The adapter consists of a function template that creates a function object in a class template that has the appropriate operator() specified.*
    - *two cases: the container has an actual object, vs the container has a pointer*
      - `mem_fun(pointer-to-member-function)` adapts a pointer from a container
        - (should have been named "mem_fun_ptr")
      - `mem_fun_ref(pointer-to-member-function)` adapts an object(reference) from a container
  - **if the member function has a parameter, use another adapter(`bind2nd`) to pass in a second value (remember the first parameter is the hidden "this" parameter)**
    - `bind2nd(pointer_to_function_adapter), second_value)`
    - *std lib does not contain adapters to allow you to supply two parameters (plus the first "this" pointer)*
    - *TR1 extensions have a super-general bind adapter and a more convenient member function adapter. See the handout on course website - you can use it with gcc 4.x*
  - **Example**

    -
      ```
      #include <iostream>
      #include <list>
      #include <string>
      #include <algorithm>
      #include <functional>

      using namespace std;

      class Thing {
      public:
            Thing(int in_i) : i(in_i) {}
            void print() const
                  {cout << "Thing " << i << endl;}
            void update()
                  {i++;}
            void set_value(int in_i)
                  {i = in_i;}
            int get_value() const
                  {return i;}
      private:
            int i;
      };

      // write it a line by itself
      ostream& operator<< (ostream& os, const Thing &t)
      {
            os << "Thing " << t.get_value() << endl;
            return os;
      }


      int main()
      {
            Thing t1(1), t2(2), t3(3);
            list<Thing *> ptr_list;
            ptr_list.push_back(&t1);
            ptr_list.push_back(&t2);
            ptr_list.push_back(&t3);
      ```

```
	cout << "Output using ptr_list" << endl;
	for_each(ptr_list.begin(), ptr_list.end(), mem_fun(&Thing::print) );

	list<Thing> obj_list;
	obj_list.push_back(t1);     // a copy
	obj_list.push_back(t2);
	obj_list.push_back(t3);

	// in below,
	// for obj_list, the derefenced iterator from for_each is a Thing object
	// for ptr_list, the derefenced iterator from for_each is a Thing pointer

	cout << "Output using obj_list" << endl;
	for_each(obj_list.begin(), obj_list.end(), mem_fun_ref(&Thing::print) );

	// try updating the objects
	for_each(ptr_list.begin(), ptr_list.end(), mem_fun(&Thing::update) );

	cout << "Output after update using ptr_list" << endl;
	for_each(ptr_list.begin(), ptr_list.end(), mem_fun(&Thing::print) );

	// try updating the objects
	for_each(obj_list.begin(), obj_list.end(), mem_fun_ref(&Thing::update) );

	cout << "Output after update using obj_list" << endl;
	for_each(obj_list.begin(), obj_list.end(), mem_fun_ref(&Thing::print) );

	// try setting the value using an adapterto pass an argument for the pointer list
	int new_value = 21;
	cout << "Set the value with an adapter on the pointer list" << endl;
	for_each(ptr_list.begin(), ptr_list.end(), bind2nd(mem_fun(&Thing::set_value),
new_value) );
	cout << "Output after setting value using ptr_list" << endl;
	for_each(ptr_list.begin(), ptr_list.end(), mem_fun(&Thing::print) );


	// try setting the value using an adapter to pass an argument for the object list
	new_value = 42;
	cout << "Set the value with an adapter on the object list" << endl;
	// Comeau's compiler objects to the line below
	for_each(obj_list.begin(), obj_list.end(), bind2nd(mem_fun_ref(&Thing::set_value),
new_value) );
	cout << "Output from using the print member function on obj_list" << endl;
	for_each(obj_list.begin(), obj_list.end(), mem_fun_ref(&Thing::print) );



	cout << "done!" << endl;
}
```

- **Two important adapters are missing from the Standard Library!**
  - **Ones that allow you to say whether you want the first or second of a std::pair<>**
  - **e.g.**
    - *std::map<string, Thing> where Thing::print() which you want to call.*
    - *write a function object class*

    -
      ```
      struct Thing_printer {
            void operator() (std::map<string, Thing>::value_type thePair)
                  {thePair.second.print();}
            }
      for_each (my_map.begin(), my_map.end(), Thing_printer());
      ```

- **Maybe something will get added in the next round of standardization!**
  - *Probable best solution: additional iterator types whose derefence is the first or second instead of the pair!*
- **Many (but not all) situations can be handled by the TR1 bind function template.**
  - *TR1 - Standard Library Technical Report No. 1 - a set of addtional Std. Lib. facilities scheduled to be added to the next version of the C++ Standard.*
  - *Developed by Boost (www.boost.org), an open-source community devoted to developing high-quality C++ libraries good enough to be considered for the Standard.*
  - *bind does everything that you can do with ptr_fun, mem_fun, mem_fun_ref, bind1st, bind2nd, and does it better, with a single function and syntax.*
  - *See handout and examples on course website - you can use it in Project 3.*

    -
      ```
      // call a member function that takes the second of the pair as an argument:
      obj_map<int, Thing>;
      ...

      for_each(obj_map.begin(), obj_map.end(),
            bind(&Thing::print,
                  bind<Thing>(&map<int, Thing>::value_type::second, _1)) );

      // call a non-member function that prints the second of each pair
      for_each(obj_map.begin(), obj_map.end(),
            bind(&print_Thing,
                  bind<Thing>(&map<int, Thing>::value_type::second, _1)) );

      // call a member function that takes a second argument
      ptr_map<int, Thing *>
      for_each(ptr_map.begin(), ptr_map.end(),
            bind(&Thing::set_value,
                  bind<Thing *>(&map<int, Thing *>::value_type::second, _1),
                  new_value) );

      // Note how basic syntax is the same for both Thing and Thing * containers and member and
      non-member functions!
      ```
- **Also, see my use_second experimental template in the examples folder.**

  -
    ```
    // call a function that takes the second of the pair as an argument:
    for_each(obj_map.begin(), obj_map.end(), use_second(print_Thing));
    ```

  -
    ```
    // call a member function for the object in the second of the pair that takes another
    argument:
    for_each(obj_map.begin(), obj_map.end(), use_second(bind2nd(mem_fun_ref(&Thing::set_value),
    new_value)) );
    ```
  - doesn't work for everything yet - there are serious technical problems - experiment with this to discover why concocting these things is non-trivial

- **Adaptable function objects: To be able to combine adapters, the function return type and argument types have to be visible to the template classes**
  - **std::binary_function, std::unary_function class templates**
    - see handout for examples of how these are used in the std. lib adapters.
    - If you want to use the adapters with your own function objects, you can inherit from these to supply the types: e.g. first arg, second arg, return type.
    - NOTE: NO POINT IN DOING THIS if your function object class is so specialized that it will never be adapted with one of the standard adapters! - Don't waste time - not really needed in this course!
  - **Example 1**
    -
```
#include <iostream>
#include <vector>
#include <algorithm>
#include <functional>

using namespace std;

/*
This example shows how one might go about defining function objects that integrate with
the adapters used with the algorithms.
*/

// a function object class that simply compares two integers for equality
// inherit from std::binary_function which contains typedefs
// that are required by adapters like bind2nd
struct Match_value : public binary_function<int, int, bool> {
	bool operator() (int x, int y) const // const required here for template instantiation
		{return x == y;}
};


int main()
{
	cout << "Enter a bunch of values, 0 when done:" << endl;
	vector<int> data;
	int i;
	while(cin >> i && i != 0)
		data.push_back(i);

	int v;
	cout << "Enter a match value:";
	cin >> v;

	// use the match_value function object with the bind2nd adapter to pass in the match
value
	// as the second parameter
	vector<int>::iterator it = find_if(data.begin(), data.end(), bind2nd(Match_value(), v));

	// use one of the predefined function object class templates that does the same thing
	// vector<int>::iterator it = find_if(data.begin(), data.end(), bind2nd(equal_to<int>(),
v));

	if(it == data.end())
		cout << "not found" << endl;
	else
		cout << "found" << endl;
}
```

- **Example 2**

  -
    ```
    #include <iostream>
    #include <list>
    #include <string>
    #include <algorithm>
    #include <functional>

    using namespace std;

    /*
    This example shows how one might go about defining function objects that integrate with
    the adapters used with the algorithms.
    */

    class Thing {
    public:
    	Thing(int in_i) : i(in_i) {}
    	void print() const
    		{cout << "Thing " << i << endl;}
    	void update()
    		{i++;}
    	void set_value(int in_i)
    		{i = in_i;}
    	int get_value() const
    		{return i;}
    private:
    	int i;
    };

    // write it a line by itself
    ostream& operator<< (ostream& os, const Thing &t)
    {
    	os << "Thing " << t.get_value() << endl;
    	return os;
    }


    // inherit from std::binary_function which contains typedefs
    // that are required by adapters like bind2nd
    struct Thing_int_printer : public std::binary_function<Thing, int, void> {
    	void operator() (Thing t, int i) const
    		{ cout << "Thing value: " << t.get_value() << ": int: " << i << endl;}
    };

    struct odd_Thing_pred : public std::unary_function<Thing, bool> {
    	bool operator() (Thing t) const
    		{ return t.get_value() % 2;}
    };



    int main()
    {
    	Thing t1(1), t2(2), t3(3);
    	list<Thing> obj_list;
    	obj_list.push_back(t1);
    	obj_list.push_back(t2);
    	obj_list.push_back(t3);
    ```

```
	cout << "Output using print member function" << endl;
	for_each(obj_list.begin(), obj_list.end(), mem_fun_ref(&Thing::print) );

	int int_value = 5;
	cout << "\nOutput using an adapter on my adaptable binary function object" << endl;

	for_each(obj_list.begin(), obj_list.end(), bind2nd(Thing_int_printer(), int_value));


	cout << "\nOutput using my adaptable unary predicate function object" << endl;
	list<Thing>::iterator it1 = find_if(obj_list.begin(), obj_list.end(), odd_Thing_pred());
	cout << *it1 << endl;

	cout << "Output using an adapter on my unary predicate function object" << endl;
	list<Thing>::iterator it2 = find_if(obj_list.begin(), obj_list.end(), 
not1(odd_Thing_pred()));
	cout << *it2 << endl;


	cout << "Done!" << endl;
}
```