Lecture Outline - algorithms and function objects - highlights

- **Stroustrup chs. 32, 33 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:**

▼

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

  list<int> int_list = {1, 3, 5, 7, 9};

  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**
  -

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

      int main()
      {
       vector<int> vi = {1,2,3,4,5,6,7,8,9,10};
       int ai[10] = {1,2,3,4,5,6,7,8,9,10};

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

       for_each(ai, ai+10, print);
      }
      ```
- **Range for - new in C++11- similar to algorithms - uses iterator interface**
  - *you can write a for loop that iterates through a container or built-in array in a super-compact form:*
  - *`for(type variable : container) {code using variable}`*
  - *The compiler will generate the code corresponding to a for loop that iterates through the whole container and assigns the dereferenced iterator value to the variable on each iteration.*
  - *The type of the variable needs to match the type of the dereferenced iterator.*
    - auto can be used here to automatically declare the type
    - auto& to declare a reference type variable
  - *The container can also be a built-in array if the declaration of the array is in scope.*
  - *This is implemented in terms of two templates: std::begin() and std::end().*
    - if the container is a built-in array, they return a pointer to the first cell of the array, and a pointer to one past the last cell of the array.
    - if the container is a container class (like vector<>), they return the iterator provided by the .begin() and .end() member functions.
  - *This works for any container (including your own, such as Ordered_iist<> in project 2), as long as:*
    - The container has a .begin() and .end() member function that returns something that behaves like an iterator - it has increment operators, dereference operators, assignment operator, == and != operators,

- **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**
- **Some algorithms appear to remove items from sequence containers, but not really:**
  - *Logic is to copy/move items to the front to overwrite unwanted items, and return a new "end" to mark the end of the desired contents; the rest of the container has the original items that were there (or their moved replacements)*
  - *remove algorithm is thus easy to misunderstand - it just copies or moves contents around, doesn"t take anything out of the container.*
  - *unique algorithm is similar - duplicates overwritten with following items.*
    - Example:

```
int main()
{
 vector<int> vi = {1,2,2,3,4,2,5,2,6};
 cout << "\nOriginal contents of container from begin to end:\n";
 copy(vi.begin(), vi.end(), ostream_iterator<int>(cout, " "));
 cout << endl;

 cout << "\nCall remove algorithm to \"remove\" all 2s\n";
 auto new_end = remove(vi.begin(), vi.end(), 2);

 cout << "\nNew contents of container from begin to new end:\n";
 copy(vi.begin(), new_end, ostream_iterator<int>(cout, " "));
 cout << endl;

 cout << "\nComplete contents of container from begin to end:\n";
 copy(vi.begin(), vi.end(), ostream_iterator<int>(cout, " "));
 cout << endl;
}

/* Output:

Original contents of container from begin to end:
1 2 2 3 4 2 5 2 6

Call remove algorithm to "remove" all 2s

New contents of container from begin to new end:
1 3 4 5 6

Complete contents of container from begin to end:
1 3 4 5 6 2 5 2 6
*/
```

- **copy algorithm is often a problem about this - has to be room at the destination**

```
int main()
{
 vector<int> v1 = {1,2,3,4,5,6,7,8,9,10};
 vector<int> v2 = {1,2,3};

 // copy part of v1 into v2 - there is space for it
 copy(v1.begin(), v1.begin()+3, v2);

 // copy all of v1 into v2 - oops!
 copy(v1.begin(), v1.end(), v2);
}
```

- **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 template tricks available using the STL iterator adapters*
- *wrap an iterator interface around the desired operations*
- *So can use them in the iterator slots of the STL algorithms*

Insert iterators

- **Call the container member functions to put objects in the container**
  - *e.g. insert iterators - wrap an iterator interface around a function like push_back*
  -
    ```
    int main()
    {
     vector<int> vi;
     int ai[10] = {1,2,3,4,5,6,7,8,9,10};

     // copy ai into vi, making space as needed
     copy(ai, ai + 10, back_inserter(vi));
    }
    ```
  - *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;
      }
      ```
- **to fill a sequence container with copy, use a front_inserter or back_inserter**
- **to fill a set container, use inserter; supply an iterator that "hints" where the item might go to speed things up, but it always goes into the right place.**
- **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, declare iterator in place
      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));
  ```
- *DO NOT USE THESE STANDALONE! ONLY USE IN ALGORITHMS!*
  - The point is that they look like an iterator in a Std. Lib. algorithm. No value otherwise - just obfuscates the code.

Function objects with state: for_each gives it back to you!

-
  ```
  // 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;
  }
  ```
- *See how f is returned? Not very useful if f is a function, but if it is a function object with state, then you get a copy back from for_each and can access its state!*
- *Can pass a function object over the contents of a container and get the results back in one line of code!*
- *This is the idiom: Use it!*
  - `My_FO_type results = for_each(container.begin(), container.end(), My_FO_type());`
  - Note that move semantics means can be very little overhead in copying out even complex 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!
- **STL algorithms is 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 template parameter in the declaration to specify the ordering relation**
  - *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
      ```
  - *won't accept a function pointer in this slot in the declaration.*
    - 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.
    - Example:

      -
        ```
        bool rev_comp (int i1, int i2) const {return i2 < i1;} // reverse order of integers

        set<int, bool (*) (int, int)> sri (rev_comp);  // set of ints in reverse order
        ```
  - *Preferred: use the function object class in the declaration. Simplest syntax.*
  - *Only reason to use a function pointer: can specify the actual ordering function at runtime (unusual).*
- **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 a non-number 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 a non-number 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 ()
{
 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 in a function object class!**
  - *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;
```

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.*
  - *Create a function object that saves the second argument value in a member variable, and a pointer to the function, and whose operator() accepts the dereferenced iterator and calls the function with that value for tirst argument and the stored value for the second argument.*
  - *semipseudocode of a simple two-argument binder that saves a second argument and accepts the first*

    -
      ```
      template class<F, P, DI>
      class Binder {
      public:
       Binder(F  f_ptr, P p_value) :
        saved_f_ptr(f_ptr), saved_p_value(p_value)
        {}
       void operator() (DI x)
        {
         saved_f_ptr(x, saved_p_value);
        }
      private:
       F  saved_f_ptr
       P saved_p_value;
      };
      ```

std::bind is a super-general function object generator

- **Generates function objects containing some parameters in the operator() definition, and storing some “bound” values in member variables, along with a pointer to the function. Works for a lot of parameters - amazing**
- **See handouts on web site for gruesome details about C++98 adapters and binders.**
- **These are deprecated in C++11 because std::bind is much more general and powerful - see the handout.**

std::function<> is a template that creates function objects that wrap a callable object of any type, as long as it can be called with the specified return type and parameters.

- **declaration syntax: function<return_type (parameter types)>**
- **Notice that functions, function pointers, function objects, lambdas, bind function objects, etc all have their own types depending not just on return and parameter types, but also what kind of thing they are:**
  - *`E.g. a function pointer void (*fp) (int, double) is not the same type as  [] (int, double) { /* code */}  or some funcion object.`*
    - `double func(int i, double d) {return i+d;} // func has type function return double, taking an int and double.`
    - `struct FOC {double operator() (int i, double d) {return i+d;}}; FOC fo; // fo has type FOC.`

    - `[](int i, double d){return i+d;} // has unspecified (unknown to programmer) type`
    - `bind(f, _1, _2) // has unspecified (unknown to programmer) type`
  - *but all of these can be called the same way, even though they are different types!*
    - `double result =  something (intv, doublev);`
  - *but because they have diferent types, can not store these different callable things in an STL container - containers are homogenous - all the items must have the same type.*

- **std::function<> allows you to store any kind of callable type and then call it through an object of a single type:**
  - *`std::function<double (int double)) f;`*
  - *`f = func; // store as a function pointer`*
  - *`f = fo;  // store the function object of type FOC`*
  - *`f = [](int i, double d){return i+d;}; // store the lambda expression function object`*
  - *`f= bind(func, _1, _2); // store the bind function object`*
  - *`f always has type function<double (int double)> regardless of what you store in it`*

- **This means you can have an STL container of function<double (int, double)> objects - all the same type, but each of which calls something that has a different type.**
  - e.g.

```
e.g. list<function<double (int, double)>> callables;

callables.push_back(func);
callables.push_back(fo);
callables.push_back([](int i, double d){return i+d;})
callables.push_back(bind(func, _1, _2));

for(auto callable : callables)
 double result = callable(intvar, doublevar);
```

- **This type-hiding ability is the key feature of std::function<> . Use it only when that is what you need.**
  - *function<> does some pretty heavy-weight stuff using inheritance and virtual functions:*
    - creates a base class to provide the common interface, and a derived class to wrap each type of callable object - virtual functions used to map from the base interface class to the specific callable object class
    - may dynamically allocate a derived class object to hold the callable object you store
  - *Do not use function<> if something simpler, more efficient will work instead.*
    - For example, can often use a function template and give it bind function objects to allow calling functions that differ in number of parameters.

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

- **The function call in the for_each function won't work - you can't call a member function that way.**
- **Need a function object that stores the pointer-to-member-function, and then the function call operator parameter is used as "this" object in a call using the pointer to member function.**

- **semipseudocode of a simple member function binder that accepts the dereferenced iterator as a object pointer used for the member function call, with the saved value as the single argement.**

  -
    ```
    template class<F, P, DI>
    class Binder {
    public:
     Binder(F  mf_ptr, P p_value) :
      saved_mf_ptr(mf_ptr), saved_p_value(p_value)
      {}
     void operator() (DI x) // use x as the object pointer
      {
       x->*saved_mf_ptr(saved_p_value);
      }
    private:
     F  saved_mf_ptr
     P saved_p_value;
    };
    ```
- **C++11 has std::mem_fn adapter for this (see the std::bind handout) if there are no ordinary parameters to the member function.**
- **std::bind will handle all cases - the first parameter is the "this" object, the remainder are the other parameters. See the bind handout.**

Using algorithms with map container is a pain

- **The dereferenced iterator is a pair; often you want to work with the .second of the pair, but the function gets the pair anyway.**
- **Ways to solve it**
  - *`std::map<std::string, Thing>  and Thing::print() is what you want to call.`*

    -
      ```
      note: smap<string, Thing>::value_type is a typedef/type alias for type of the items in the
      container, namely std::pair<const string, Thing>.  Use this Standard alias to save typing
      misery!

      Use a type alias for the map type to save more typing:
      using Things_t = std::map<std::string, Thing>;
      ```
  - *write a helper function*

    -
      ```
      void Thing_print_helper(const Things_t::value_type& thePair)
      {the_pair.second.print();}
      for_each (my_map.begin(), my_map.end(), Thing_print_helper);
      ```
  - *write a function object class helper*

    -
      ```
      struct Thing_printer {
       void operator() (const Things_t::value_type& thePair)
        {thePair.second.print();}
       }
      for_each (my_map.begin(), my_map.end(), Thing_printer());
      ```
  - *use a lambda expression*

    -
      ```
      for_each (my_map.begin(), my_map.end(),
       [](const Things_t::value_type& thePair)
        {thePair.second.print();}
       );
      ```
- **std::bind will do it, but it's ugly!**
  - *See handout and examples on course website.*

-
  ```
  // 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(&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(&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(&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!
  ```

-