Advanced C++

February 12, 2015

Paul Bossi lecturing on behalf of
Mike Spertus

Review of when an object is moved vs. when it is copied

- In general, the compiler will only try to move from an object when it can prove the object will never be used again
- In practice, this means that the object is an unnamed temporary value, such as the return value of a function
- For example, in `btree.cpp` on chalk, the function `f()` returns a `btree`.
- In the following declaration of `t1`, the return value of `f()` can never be used after the initialization because it doesn't even have a name
  - `btree t1(f());`
- In that case, it is safe for the compiler to cannibalize the object by moving the return value of `f()` into `t1` because it has a death sentence and no one will ever see it again.
- On the other hand, in the following code, we need to use the copy constructor because `t1` will not be happy if its value is changed
  - `btree t4(t1);`
- Occasionally, the programmer knows an object will never be used again, and would like to tell the compiler that it is free to cannibalize the object
  - `btree t7(move(t1)); // move constructor can cannibalize t1`

When is a constructor used vs. an assignment operator

- A constructor is used when an object is created, while an assignment operator (i.e. `operator=()`)is used when modifying the value of an existing variable
- ```
  // New object is being created
  // Use a constructor
  btree t1(f()); // Move constructor
  btree t2{f()}; // Move constructor
  btree t4(t1); // Copy constructor
  // In the following line, don't be confused
  // by the "=", a new object is being
  // created, so we need a constructor
  btree t3 = f(); // Move constructor
  ```
- ```
  // An existing object is being modified
  // Use an assignment operator
  t2 = f(); // Move assignment: operator=(btree &&)
  t2 = t1; // Copy assignment: operator=(btree const &)
  ```

There were some problems with the matrix example code

- Let’s look at what went wrong and see if there are any lessons to be learned
- First, all of the sample programs compiled correctly under Visual Studio 2013
  - Remember, a compiler will not necessarily reject incorrect code
  - Also, C++ is not an exact standard. It has undefined and implementation-defined behavior
  - As we will see, Visual Studio also has some bugs
  - As we will also see, g++ has some relevant problems
- All of these problems were eventually fixed on chalk

Some boring mistakes

- Mike was missing some header files
- Sometimes you get away with this because you include one header that coincidentally includes another
  - We’re hoping to change that in C++17
- But it makes your code fragile and non-portable
- Mike forgot some namespace qualifiers and using statements
  - E.g., `std::max` instead of just `max`

`minor` called with wrong number of arguments

- Some of you ran into a bizarre-looking error where g++ complained that `minor()` was being called with the wrong number of arguments but the number of arguments appeared to be correct
- The problem was that the `g++` standard libraries `#define` a macro named `minor()`
- Since macros are just textual substitution and don't respect namespaces, they can come out of nowhere and bite you
  - This is why macros are evil and namespaces are goodness
- The solution is to get rid of the macro by putting the following line after including the standard library header
  - `#undef minor`

Problems with `max`

- In `accumulateMax`, the function `std::max` is called with two arguments of different types, so it can't deduce what type it is taking the max of
  - Mike got this wrong even though slide 18 of lecture 4 warned of exactly this!
    - Well, that slide used `min` instead of `max`, so who could have known ☺
- The solution is to put the right type of accumulator in accumulate
  - Slide 20 of Lecture 4 warned of exactly this!
- Mike learned the hard way that these warnings were worth knowing!

Dependent base classes

- What does this print?

```
void f() { cout << "::f" << endl; }
struct A {
  void f() { cout << "A::f" << endl; }
};

template<typename T>
struct B : public T {
  void g() { f(); }
};

int main()
{
  B<A> ba;
  ba.g(); // Does this print ::f or A::f?
}
```

Let's see

- Microsoft says A::f()
- g++ says ::f()
- Which is right?

g++ is right

- A base class that depends on the template parameter is called a dependent base class
- The compiler would like to know where it found `f()`, so we don't look in the dependent base class
- If you want the inherited one do any of the following inside B
  - `using T::f; // First line inside B`
  - `T::f(); // When calling f() inside g()`
  - `this->f(); // When calling f() inside g()`

What does this have to do with matrix

- In PSMatrix.h, `MatrixCommon<T, rows, cols>` is a dependent base class of `Matrix<T, rows, cols>`
  - Because it is a base class that uses (all three) template parameters of Matrix, we need to explicitly access its data member as above
    - E.g., `this->data[i][j];` instead of just `data[i][j]`

Making matrix more reliable

- If you try taking the determinant of a non-square matrix, you get a long, confusing, and unhelpful error message deep in the guts of determinant calculation
-

```
Matrix.cpp
1>d:\program files (x86)\microsoft visual studio 12.0\vc\include\array(210): error C2148: total size of array must not exceed 0x7fffffff bytes
1>          d:\dropbox\cspp51044\2015\lecture 5\matrix.h(102) : see reference to class template instantiation 'std::array<std::array<double,0>,-1>' being compiled
1>          d:\dropbox\cspp51044\2015\lecture 5\matrix.h(85) : see reference to class template instantiation 'mpcs51044::Matrix<-1,0>' being compiled
1>          d:\dropbox\cspp51044\2015\lecture 5\matrix.h(80) : while compiling class template member function 'double mpcs51044::Matrix<0,1>::determinant(void) const'
1>          d:\dropbox\cspp51044\2015\lecture 5\matrix.h(85) : see reference to function template instantiation 'double mpcs51044::Matrix<0,1>::determinant(void)
const' being compiled
1>          d:\dropbox\cspp51044\2015\lecture 5\matrix.h(85) : see reference to class template instantiation 'mpcs51044::Matrix<0,1>' being compiled
1>          d:\dropbox\cspp51044\2015\lecture 5\matrix.h(80) : while compiling class template member function 'double mpcs51044::Matrix<1,2>::determinant(void) const'
1>          d:\dropbox\cspp51044\2015\lecture 5\matrix.h(85) : see reference to function template instantiation 'double mpcs51044::Matrix<1,2>::determinant(void)
const' being compiled
1>          d:\dropbox\cspp51044\2015\lecture 5\matrix.h(85) : see reference to class template instantiation 'mpcs51044::Matrix<1,2>' being compiled
1>          d:\dropbox\cspp51044\2015\lecture 5\matrix.h(80) : while compiling class template member function 'double mpcs51044::Matrix<2,3>::determinant(void) const'
1>          d:\dropbox\cspp51044\2015\lecture 5\matrix.h(85) : see reference to function template instantiation 'double mpcs51044::Matrix<2,3>::determinant(void)
const' being compiled
1>          d:\dropbox\cspp51044\2015\lecture 5\matrix.h(85) : see reference to class template instantiation 'mpcs51044::Matrix<2,3>' being compiled
1>          d:\dropbox\cspp51044\2015\lecture 5\matrix.h(80) : while compiling class template member function 'double mpcs51044::Matrix<3,4>::determinant(void) const'
1>          d:\dropbox\cspp51044\2015\lecture 5\matrix.cpp(18) : see reference to function template instantiation 'double mpcs51044::Matrix<3,4>::determinant(void)
const' being compiled
1>          d:\dropbox\cspp51044\2015\lecture 5\matrix.cpp(10) : see reference to class template instantiation 'mpcs51044::Matrix<3,4>' being compiled
```

Static assertions

- C++ has a `static_assert` command that lets you check a condition and have the compiler output an error message
- For example, suppose our code depends on a subtle bug fix in version 1.52 of Boost
- It would be very easy for someone to compile the code with an old version of Boost, leading to subtly buggy behavior that would be very difficult to diagnose
- We can tell the compiler to protect us with a descriptive error message during compilation
- ```
  static_assert(BOOST_VERSION >= 105200,
    "This code requires Boost 1.52 or later");
  ```
  ```

The STL

- The main topic of today’s lecture is the Standard Template Library, which is C++’s approach to
  - Containers
  - Iterators
  - Algorithms
- The STL was initially developed by Alex Stepanov late in the C++98 process but was so revolutionary that a late change was made to bring it in
- For the theory behind the STL, Stepanov has two excellent books on the computer science behind their design
  - *Elements of Programming*
  - *From Mathematics to Generic Programming*
- We will focus on the practical aspects

A tour of standard library containers

- Sequence containers
  - `vector`, `array`, `deque`, `list`, `forward_list`, `bitset` (don’t use `vector<bool>`, which has been deprecated)
- Associative containers
  - `set`, `unordered_set`, `map`, `unordered_map`
- Container adaptors
  - Adapt a sequence container to support a specific interface
  - `stack`, `queue`, `priority_queue`
- heap
  - Maybe next quarter

Lists

- `std::list` is a doubly-linked list
- `std::forward_list` is a singly-linked list
- Interestingly, lists have no `size()` method because calculating the size of a linked list is expensive.
  - See http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2543.htm for a discussion of design decisions

`std::map`

- `std::map` is a key-value store
- Internally, it is implemented as a binary tree
- This means that the keys have to be "less-than comparable"
  - More on this in next slide
- To create and use a map from strings to ints
  - ```
    map<string, int> msi;
    msi.insert(make_pair("foo", 7)); // Key is "foo" value is 7
    msi["bar"] = 5; // Key is "bar" value is 5
    cout << msi["baz"]; // Prints 0 because
    // creates a key with default value if necessary
    auto it = msi.find("quux");
    if(it != msi.end()) // Only print a value if the
      cout << *it;       // key exists
    ```
    ```

Key comparison in maps

- When you don't specify how to compare keys, the map uses the functor `std::less` which defaults to calling `operator<()`
- If you want to use your own type as a key, implement `operator<()`. You can also specialize `std::less` for your class
- **Advanced:** You can also tell the map how you would like it to compare keys
  - For example, create a case-insensitive map as follows
  - ```
    #include<boost/algorithm/string/predicate.hpp>
    #include<map>
    ...
    std::map<string, int, bool(*)(string, string)>
      ciMap([](string l, string r) { return ilexicographical_compare(l, r); });
    ciMap["foo"] = 7;
    ciMap["FoO"] = 5;
    cout << ciMap["foo"] << endl; // Prints 5
    ```
  - Note that this example uses (and hopefully motivates) some features we will study later this quarter
    - **Function pointer types:** The function pointer type `bool(*)(string, string)` as the type of the key comparator. We will learn more about these later, but this is the type "pointer to function taking two strings and returning a boolean"
    - **Lambda functions:** These are expressions that evaluate to functions or functors, allowing you to handily create a function inside a statement. We will learn more about these later, but this is a quick way to turn Boost's `ilexicographical_compare` into a function with the above signature.
    ```

Other associative containers

- `std::set` just stores a set of keys without values but is internally implemented by a `std::map`
- `std::multimap` is a key-value store where multiple elements can have the same name

Hash tables

- There was wide desire to add hash tables to C++11
  - `std::map` requires that its elements a "less than comparable," but there is not always a natural ordering
  - `std::map` may be much slower than a true hashtable on large collections
- Google code search (now defunct ☹) showed that we couldn't call them hash_table.

Unordered associative containers

- Instead of hash table, we used the name `std::unordered_map`, which acts more or less just like a `std::map`
  - Instead of `std::less`, it uses `std::hash` by default
  - Hash functions are already provided for standard library types like `std::string`, but you will need to specify your own specialization of `std::hash` for your own types
  - If you iterate the elements of `std::map`, you get them in order, but for a `std::unordered_map`, you don't get them in any particular order
- There are also `unordered_set`, `unordered_multimap`, and `unordered_multiset`

Making stack exception-safe

- You would expect to be able to pop an object of a stack
  - ```
    stack<A> stk;
    ...
    A a(stk.pop()); // Illegal!
    ```
- The problem with this would be if A's copy constructor threw an exception.
  - The top element could be lost forever
- Instead, stack::pop has void return type.
- Do the following instead
  - ```
    stack<A> stk;
    ...
    A a(stk.top());
    stk.pop();
    ```

Checking if a container is empty

- This is item 4 from Scott Meyer's Effective STL
- You often see code that checks if a container is empty by comparing its size to zero
- Don't do this
- Call the empty method instead
- Figuring out whether a container is empty can be a lot more efficient internally than figuring out exactly how many elements are in it
-
```
std::map<string, int> si;
/* ... */
if(si.size() != 0) // Bad!
  /* ... */
if(!si.empty()) // Right
  /* ... */
```

Iterators

- Iterators are the C++ generalization of C pointer arithmetic
  - In fact, C pointers are iterators, but avoid them because it is easy to overrun a buffer
    - This is what caused HeartBleed! http://nakedsecurity.sophos.com/2014/04/08/anatomy-of-a-data-leak-bug-openssl-heartbleed/
- All STL containers can produce iterators that let you safely run through their elements without running off the end

Before we understand iterators, let's look at pointers

- A pointer stores the address of an object
- `A *` is type "pointer to `A`"
  - ```
    A *ap; // Note ap is not initialized. Don't use yet!
    ```
- `New` expressions return a pointer to the newly created object
  - ```
    A *ap2 = new A();
    ```
- `&` takes the address of an object
  - ```
    A a;
    ap = &a; // & takes address of obj. Now we can use ap
    ```
- `->` is short for `(* ).`
  - ```
    ap2->x = 3; // Sets obj's x member to 3. (Assume A has 'int x' member)
    (*ap2).x = 3; // Same as above line
    ```
- `*` "dereferences" a pointer, returning a reference to the object it is pointing to
  - ```
    // a gets copy of obj pointed to by ap2
    a = *ap2;
    a.x = 5; // Doesn't modify ap2->x
    // ar is reference to obj pointed to by ap2
    A &ar = *ap2;
    ar.x = 7; // modifies obj pointed to by ap2
    ```
    ```

Pointers into arrays

- If a pointer points to an element of an array, then you can also use it to access other elements of the array via “pointer arithmetic”
  - ```
    A *arp = new A[10]; // arp is addr of 0th elt
    A *arp2 = arp+2; // arp2 is addr of 2nd elt
    A *arp3 = &(*arp)[3]; // arp3 is addr of 3rd elt
    A &arr3 = (*arp)[3] // arr3 is ref to 3rd elt
    A as[10]; // Array of 10 A objects
    A *asp = as; // Name of array is addr of 0th elt
    A *asp2 = as+2; // asp2 points 2 objects past as
    asp2 = &as[2]; // Does the same thing
    as++;  // as now points to the 1st elt
    ```
    ```

Understanding iterators

- Iterators are an abstraction of "pointer to C-style array element" for any container or sequence, not just C-style arrays
- Based on Section 6.3 of Josuttis' *The C++ Standard Library, 2nd edition*
- A type behaves as an iterator by supporting the following operators
  - `operator *` gives the element currently being iterated
  - `operator++` causes the iterator to advance to the next element
  - `operator==` and `operator!=` to compare iterators
  - `operator=` to assign iterators
  - Some iterators define additional operators like `operator--`

Iterator categories

- Iterators come in various flavors
- Forward Iterators
  - Can advance these but cannot decrement them. `std::forward_list<T>::iterator` is an example of forward iterators
- Bidirectional iterators
  - Can increment or decrement. E.g., `std::list<T>::iterator`.
- Random access iterators
  - Can add or subtract and integer to advance or retreat by a specific amount. E.g., pointer arithmetic
- Input iterators
  - Can get values from them but not assign to them. E.g., istream iterators
- Output iterators
  - Can assign to them but not get values from them. E.g., `ostream_iterator`.

Knowing what kind of iterator you have matters

- If vec is a `std::vector<int>`, then you can sort it with
  - `std::sort(vec.begin(), vec.end());`
- If lst is a std::list<int>, you get a long horribly confusing error message if you try to sort it with
  - `std::sort(lst.begin(), lst.end());`
- The problem is that `std::sort()` expects random-access iterators and linked lists only have forward iterators
  - The reason for this requirement is that sorting algorithms are built on swapping iterators, which is only efficient with random-access iterators

Advanced: Querying an iterator for its properties

- The standard library expects that if a `T` is an iterator type, then `iterator_traits<T>` will have member types `iterator_category`, `value_type`, `difference_type`, `pointer`, `reference`.

Example: traits for pointers as iterators

```
namespace std {
  template <class T>
  struct iterator_traits<T*> {
    typedef T                          value_type;
    typedef ptrdiff_t                  difference_type;
    typedef random_access_iterator_tag iterator_category;
    typedef T*                         pointer;
    typedef T&                         reference;
 };
}
```

If you create your own iterator, give it local typedefs for traits

- The standard library provides a `iterator_traits` primary template that looks in the iterator class

```
template<typename T>
struct iterator_traits {
  typedef typename T::iterator_category iterator_category;
  typedef typeneame T::value_type value_type;
  typedef typename T::difference_type difference_type;
  typedef typename T::pointer pointer;
  typedef typename T::reference reference;
};
```

Creating an iterator sounds hard. Is there an easier way?

- Creating a proper iterator can require a lot of boilerplate
- Fortunately, templates are good for automating boilerplace
- Boost::iterator includes a lot of helper classes that make it easy to create your own iterators
- See the HW

Advanced: querying iterator traits

- Suppose I want to write a function that uses a slow but simple algorithm for forward iterators and a more efficient algorithm to leverage random access iterators
- For example, `std::sort` only works with random-access iterators. Suppose you wanted to create a mySort function that calls std::sort on random-access iterators and does a bubble-sort on forward iterators so (smallish) `std::forward_list`s could be sorted.
  - If this looks interesting to you, it is an extra credit HW problem

Iterator tags

- The standard library provides classes that signify what kind of iterator you have
- Like we've seen before, these classes have empty bodies because we are really just looking at their type name
- Note that use of inheritance to reflect "isA" relationship
- ```
  struct input_iterator_tag {};
  struct output_iterator_tag {};
  struct forward_iterator_tag
    : public input_iterator_tag {};
  struct bidirectional_iterator_tag
    : public forward_iterator_tag {};
  struct random_access_iterator_tag
    : public bidirectional_iterator_tag {};
  ```
  ```

Calling the right function

- ```
  template<typename T>
  void
  mySort(T beg, T end, forward_iterator_tag &&)
  {
    /* Code to bubble-sort */
  }

  template<typename T>
  void
  mySort(T beg, T end, random_access_iterator_tag &&)
  {
    std::sort(beg, end);
  }

  template<typename T>
  void myFunc(T beg, T end)
  {
    myFunc(beg, end, iterator_traits<T>::iterator_category());
  }
  ```
  ```

Algorithms

- C++ includes a wide range of algorithms to that work on iterators
- We have already seen `copy`, `for_each`, and `accumulate`

Know your algorithms

- for_each
- find
- find_if, find_if_not
- find_first_of
- adjacent_find
- count, count_if
- mismatch, equal
- is_permutation
- search, search_n, binary_search

More algorithms

- `copy`, `copy_n`, `copy_if`, `copy_backward`
- `move`, `move_backward`
- `iter_swap`
- `transform`
- `replace`, `replace_if`
- `generate`
- `rotate`, `rotate_copy`, `random_shuffle`, `shuffle`
- `all_of`, `any_of`, `none_of`
  - Check if all/any/none of the items in a container (or range) have a certain property
  - Creating an example will be part of your job in the HW

Follow remove with erase

- This is item 32 of Effective STL
- Otherwise you won't get rid of anything!
- To take all of the 99s out of a vector:

```
v.erase(remove(v.begin(), v.end(), 99),
        v.end());
```

More algorithms

- `copy_n`

```
vector<int> v = getData();
// Print 5 elements
copy_n
  (v.begin(), 5,
   ostream_iterator<int>
    (cout, "\n"));
```

- Bet you’ve wished this was in C++ for years

More algorithms

- find_if_not

```
vector<int> v = { 1, 3, 5, 6, 7};

// Print first elt that is not odd
cout << *find_if_not
             (v.begin(),
              v.end(),
              [](int i) {
                return i%2 == 1;
              });
```

More algorithms

- partition_copy

```
vector<int> primes;
vector<int> composites;
vector<int> data = getData();
extern bool is_prime(int i);

partition_copy
   (data.begin(),
    data.end(),
    back_inserter(primes),
    back_inserter(composites),
    is_prime);
```

More algorithms

- `minmax,minmax_element`
  - Gets both the biggest and smallest items in the range
- `Sort variants`
  - `sort, stable_sort, partial_sort, nth_element, merge`
- `is_heap, is_heap_until, is_sorted, is_sorted_until, partial_copy`
- Set operations
  - `include,set_union, set_intersection, set_difference, set_symmetric_difference`

Homework 6-1

- Try to take the determinant of a non-square matrix in our matrix code (E.g., a `Matrix<4, 3>`).
- Submit the error message you get and look at how ugly and confusing it is
- Add a `static_assert` statement to `Matrix.h` with a descriptive message to protect against taking the determinant of a non-square matrix
- Try compiling again. What error message do you get now?
- We also want to make sure a matrix is properly initialized. Unfortunately, there is no easy way to tell if a matrix initializer list has the wrong shape at compile time (this may be fixed in C++14), so we will need to throw an exception at runtime
- Modify `Matrix`'s `initializer_list` constructor to throw a `std::invalid_argument` exception if the initializer list is a different shape than the matrix

HW 6-2: Extra credit

- Use `Boost::Iterator`'s `function_output_iterator` to create an `ostream_joiner` that acts just like `ostream_iterator` except that the delimiter only goes between elements (and not after the final element).
- Use this to easily create an "`operator<<`" to print vectors in `ostream`s.
- This is actually proposed to be added to C++. You can find further background in [Delimited Iterators (rev. 4)]()

HW 6-3

- Write a program that counts how many distinct words are in a corpus of text
  - To get a large corpus of text, download a number of books from Project Gutenberg into a directory
  - Boost.Filesystem has a convenient iterator class called `directory_iterator` that lets your program know all the files in the directory
  - You can use an `ifstream`, which gives you an input stream from a file, to read them in
  - Extra credit: Compare storing the words in a `std::set` (or `std::map`) and then in a `std::unordered_set` (or `std::unordered_map`). Is there any difference in performance? Does it depend on the size of the input?

HW 6-4

- Extend HW 6-3 to tell you the 20 most common words in the input texts