PY410 / 505
Computational Physics 1

Salvatore Rappoccio
• C++ underwent major revision in mid-00’s
• C++0x (x was supposed to be 4, but..) turned into C++11
• There is now C++17, other updates
• Major changes in C++11

Advanced C++

Copy: member data is cloned

```cpp
template <class T> swap(T& a, T& b)
{
    T tmp(a);  // now we have two copies of a
    a = b;     // now we have two copies of b
    b = tmp;   // now we have two copies of tmp (aka a)
}
```

Expensive!

http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2027.html
http://avidinsight.uk/2013/05/understanding-cpp11-move-semantics/
Move: member data is reassigned

http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2027.html
http://avidinsight.uk/2013/05/understanding-cpp11-move-semantics/
Move: member data is reassigned

Setting this to “null” is not allowed in C++03!

Cheap!

But not supported in old C++

http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2027.html
http://avidinsight.uk/2013/05/understanding-cpp11-move-semantics/
lvalue reference

```cpp
A a;
A& a_ref1 = a; // an lvalue reference
```

rvalue reference

```cpp
A a;
A&& a_ref2 = a; // an rvalue reference
```

rvalue reference can bind to a TEMPORARY variable!

```cpp
A& a_ref3 = A(); // Error!
A&& a_ref4 = A(); // Ok
```

After function A()'s temporary return value goes out of scope, does not delete the memory used for it

http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2027.html
Advanced C++

• Move semantics

```cpp
template <class T> swap(T& a, T& b)
{
    T tmp(std::move(a));
    a = std::move(b);
    b = std::move(tmp);
}
```

Moves a’s member data to tmp, state of a is undefined
Moves b’s member data to a, state of b is undefined
Moves tmp’s member data to b, state of tmp is undefined

http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2027.html
Advanced C++

- How does this help?

<table>
<thead>
<tr>
<th>Old bad way</th>
<th>Old annoying way</th>
<th>New way, “explicitly”:</th>
</tr>
</thead>
</table>
| A modify( A & a){
  return A(a);
}
A a;
A retval = modify( a ); | void modify( A & a){
  ...
}
A a;
A retval;
modify( retval ); | A modify( A & a){
  return A(a);
}
A a;
A && retval = modify( retval ); |

| New way, “implicitly”: | |
|-------------------------| |
| “A” must have a move constructor! | |

Old bad way:
- Easy to write.
- Lots of copies.
- Really dumb.

Old annoying way:
- Performant.
- Annoying to write.

New way, “explicitly”:
- Performant.
- Confusing.

New way, “implicitly”:
- Easy to write.
- Performant.

wins!
• The “new way” with C++11 looks just like the “old way” how you wanted all along, but requires a “move constructor” to be guaranteed to be implemented correctly

• Move constructor example (std::vector):

```cpp
template<typename T>
class Vector {
    // ...
    Vector(Vector&& a) noexcept : elem{a.elem}, sz{a.sz} { a.sz = 0; a.elem = nullptr; }
    Vector& operator=(Vector&& a) noexcept { elem = a.elem; sz = a.sz; a.sz = 0; a.elem = nullptr; }
    // ...
public:
    T* elem;
    int sz;
};
```

(“noexcept” means it cannot throw exception… it’s complicated)
So now, to make your code performant, implement the “Rule of 5”:
– Copy constructor
– Move constructor
– Copy operator=
– Move operator=
– Destructor

See “AdvCpp”!
“Old school” C++ (03 and earlier): Initializing data was annoying

Old way

```cpp
int aa[] = {1,2,3,4};
std::vector<int> a(aa);
```

New way

```cpp
std::vector<int> a = {1,2,3,4};
```

Better way to initialize lists in new standard

Advanced C++

• Type inference
  – Previously: had to explicitly state type
  – Now: compiler can deduce the type

Old way
```cpp
std::vector< std::map<int,float>::const_iterator >::const_iterator i = v.begin();
```

New way
```cpp
auto i = v.begin();
```

Can also use “decltype” (declare type) to make other variables of that type!
```cpp
decltype(i) j = i+2;
```

• Range-based for loop
  – Looked this before, can be combined with “auto” to make things very compact

```cpp
vector<int> aa = {1,2,3,4};
for ( auto x : aa )
    cout << x << endl;
```
Anonymous (lambda) functions

Imagine you want to sort:

```
[](int x, int y) -> int { return x + y; }
```

 Previously:

```
// sort using a custom function object
struct {
    bool operator()(int a, int b) const
    {
        return a < b;
    }
} customLess;
std::sort(s.begin(), s.end(), customLess);
```

 C++11:

```
// sort using a lambda function
std::sort(s.begin(), s.end(), [] (int a, int b){return a < b;});
```

Lots less typing

Advanced C++

- Can allocate lists of whatever types you want (tuples)

```cpp
typedef std::tuple<int, double, long &, const char *> test_tuple;
long lengthy = 12;

test_tuple proof (18, 6.5, lengthy, "Ciao!");

lengthy = std::get<0>(proof); // Assign to 'lengthy' the value 18.
std::get<3>(proof) = "Beautiful!"; // Modify the tuple's fourth element.
```
• Better pointers
  – std::shared_ptr is like a regular pointer, but calls “delete” when it goes out of scope automatically:

  ```cpp
  shared_ptr<A> factory_for_A() {
    return shared_ptr<A>(new A());
  }
  
  shared_ptr<A> a = factory_for_A();
  ```

  – Can also now hold vector<shared_ptr> (in previous C++, had auto_ptr, but this was not supported)

  ```cpp
  std::vector<std::shared_ptr<A>> v_stuff;
  ```

  v_stuff can hold a list of A *, or ANYTHING derived from A!
Advanced C++

• We’ve seen some examples of objects from the Standard Template Library (STL).
  – std::vector, std::map, std::string, etc

• There are many algorithms that can operate on them!
  – std::sort, std::find, etc

• This brings the full power of C++ and templates to bear

• The STL documentation should become your absolute best friend when coding C++
Advanced C++

- Example: Sorting:
  - Example: AdvCpp/sorting.cpp

```cpp
std::sort
```

<table>
<thead>
<tr>
<th>Defined in header <code>&lt;algorithm&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>template</strong>&lt; class RandomIt &gt;</td>
</tr>
<tr>
<td>void sort( RandomIt first, RandomIt last );</td>
</tr>
<tr>
<td>constexpr void sort( RandomIt first, RandomIt last );</td>
</tr>
<tr>
<td><strong>template</strong>&lt; class ExecutionPolicy, class RandomIt &gt;</td>
</tr>
<tr>
<td>void sort( ExecutionPolicy&amp;&amp; policy, RandomIt first, RandomIt last );</td>
</tr>
<tr>
<td><strong>template</strong>&lt; class RandomIt, class Compare &gt;</td>
</tr>
<tr>
<td>void sort( RandomIt first, RandomIt last, Compare comp );</td>
</tr>
<tr>
<td>constexpr void sort( RandomIt first, RandomIt last, Compare comp );</td>
</tr>
<tr>
<td><strong>template</strong>&lt; class ExecutionPolicy, class RandomIt, class Compare &gt;</td>
</tr>
<tr>
<td>void sort( ExecutionPolicy&amp;&amp; policy, RandomIt first, RandomIt last, Compare comp );</td>
</tr>
</tbody>
</table>