C++0x

What is C++0x?

- Updated version of C++ language.
- Addresses unresolved problems in C++03.
- Almost completely backwards compatible.
- Greatly increases expressiveness (and complexity!) of language.
- Greatly reduces difficulty of use.

Major Language Changes, pt. 1

Without C++0x:

```
void PrintMultiMap(const multimap<string, string> & m)
{
   const pair<multimap<string, string>::const_iterator,
              multimap<string, string>::const_iterator>
      myPair = m.equal_range("C++0x");

   for(multimap<string, string>::const_iterator itr =
       myPair.first; itr != myPair.second; ++itr)
       cout << itr->second << " has key C++0x" << endl;
}
```

Without C++0x:

```
void PrintMultiMap(const multimap<string, string> & m)
{
   const pair<multimap<string, string>::const_iterator,
              multimap<string, string>::const_iterator>
      myPair = m.equal_range("C++0x");

   for(multimap<string, string>::const_iterator itr =
       myPair.first; itr != myPair.second; ++itr)
       cout << itr->second << " has key C++0x" << endl;
}
```

With C++0x:

```
void PrintMultiMap(const multimap<string, string> & m)
{
   const auto myPair = m.equal_range("C++0x");

   for(auto itr = myPair.first;
       itr != myPair.second; ++itr)
       cout << itr->second << " has key C++0x" << endl;
}
```

With C++0x:

```
void PrintMultiMap(const multimap<string, string> & m)
{
   const auto myPair = m.equal_range("C++0x");

   for(auto itr = myPair.first;
       itr != myPair.second; ++itr)
       cout << itr->second << " has key C++0x" << endl;
}
```

With C++0x:

```
void PrintMultiMap(const multimap<string, string> & m)
{
    const auto myPair = m.equal_range("C++0x");

    for(const auto& elem: myPair)
        cout << elem.second << " has key C++0x" << endl;
}
```

With C++0x:

```
void PrintMultiMap(const multimap<string, string> & m)
{
    const auto myPair = m.equal_range("C++0x");

    for(const auto& elem: myPair)
        cout << elem.second << " has key C++0x" << endl;
}
```

With C++0x:

```
void PrintMultiMap(const multimap<string, string> & m)
{
    for(const auto& elem: m.equal_range("C++0x"))
        cout << elem.second << " has key C++0x" << endl;
}
```

With C++0x:

```
void PrintMultiMap(const multimap<string, string> & m)
{
   for(const auto& elem: m.equal_range("C++0x"))
       cout << elem.second << " has key C++0x" << endl;
}
```

For Comparison:

```
void PrintMultiMap(const multimap<string, string> & m)
{
   for(const auto& elem: m.equal_range("C++0x"))
       cout << elem.second << " has key C++0x" << endl;
}

void PrintMultiMap(const multimap<string, string> & m)
{
   const pair<multimap<string, string>::const_iterator,
              multimap<string, string>::const_iterator>
      myPair = m.equal_range("C++0x");

   for(multimap<string, string>::const_iterator itr =
       myPair.first; itr != myPair.second; ++itr)
       cout << itr->second << " has key C++0x" << endl;
}
```

Type Inference

- The `auto` keyword tells the C++0x compiler to give the variable the same type as its initializer.
- Can be used for local variables only.
- Compiler must be able to figure out type; error otherwise.

Range-Based For Loop

- Syntax: `for(`*`type var : range`*`) {...}`
- Automatically creates an iterator and walks from `begin()` to `end()`
- Can traverse STL containers, pairs of iterators, arrays, strings, etc.
- Like the CS106B `foreach` loop, but much smarter.

Major Language Changes, pt. 2

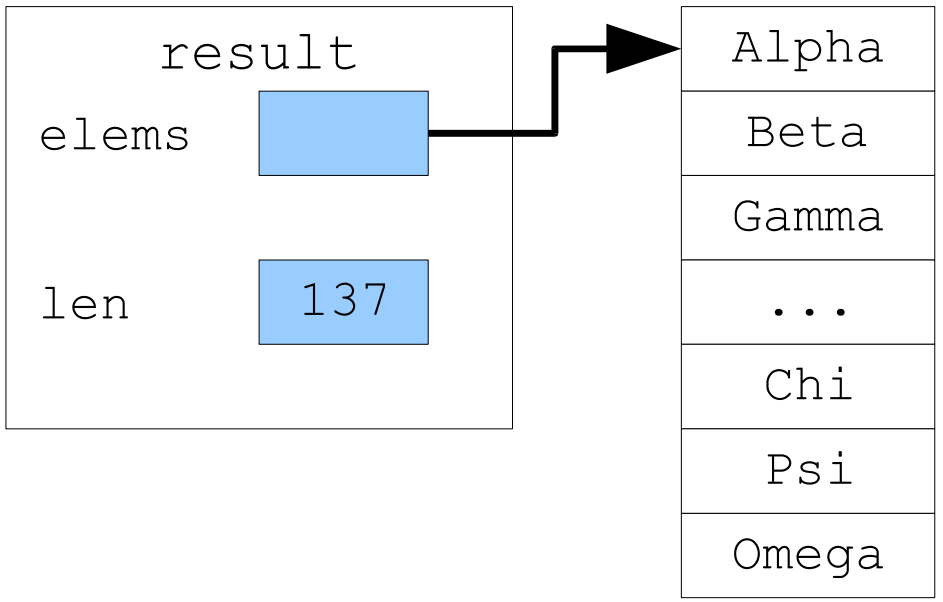
Without C++0x:

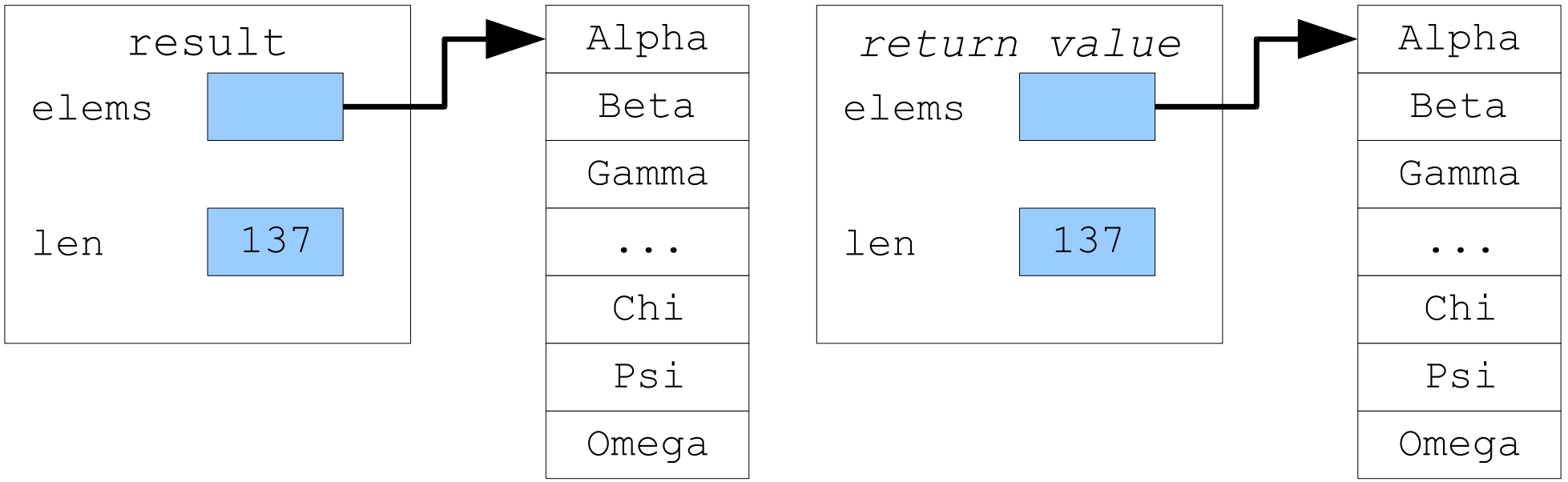
```
vector<string> ReadAllWords(const string& filename)
{
    ifstream input(filename.c_str());

    vector<string> result;
    result.insert(result.begin(),
                  istream_iterator<string>(input),
                  istream_iterator<string>());
    return result;
}
```

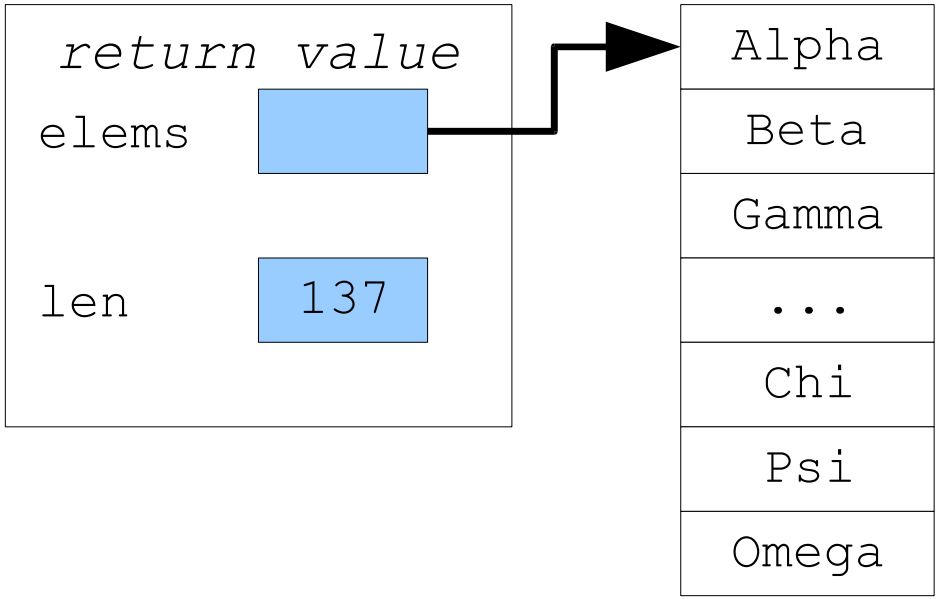
Without C++0x:

```
vector<string> ReadAllWords(const string& filename)
{
    ifstream input(filename.c_str());

    vector<string> result;
    result.insert(result.begin(),
                  istream_iterator<string>(input),
                  istream_iterator<string>());
    return result;
}
```

How efficient is this code?


result
elems
len
137
Alpha
Beta
Gamma
...
Chi
Psi
Omega


result
elems
len
137
Alpha
Beta
Gamma
...
Chi
Psi
Omega
return value
elems
len
137
Alpha
Beta
Gamma
...
Chi
Psi
Omega


return value
elems
len
137
Alpha
Beta
Gamma
...
Chi
Psi
Omega

With C++0x:

```
vector<string> ReadAllWords(const string& filename)
{
    ifstream input(filename);

    vector<string> result;
    result.insert(result.begin(),
                  istream_iterator<string>(input),
                  istream_iterator<string>());
    return result;
}
```

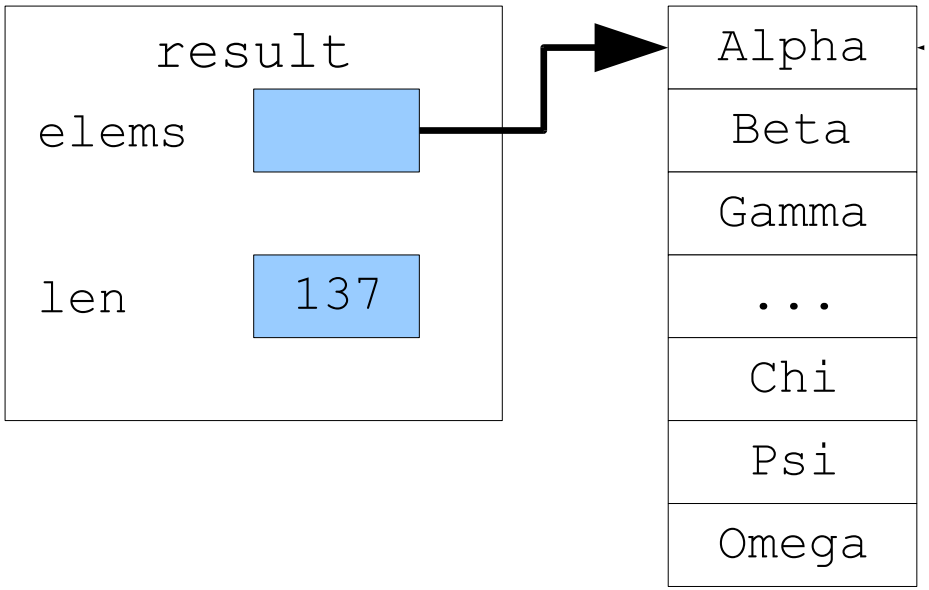
With C++0x:

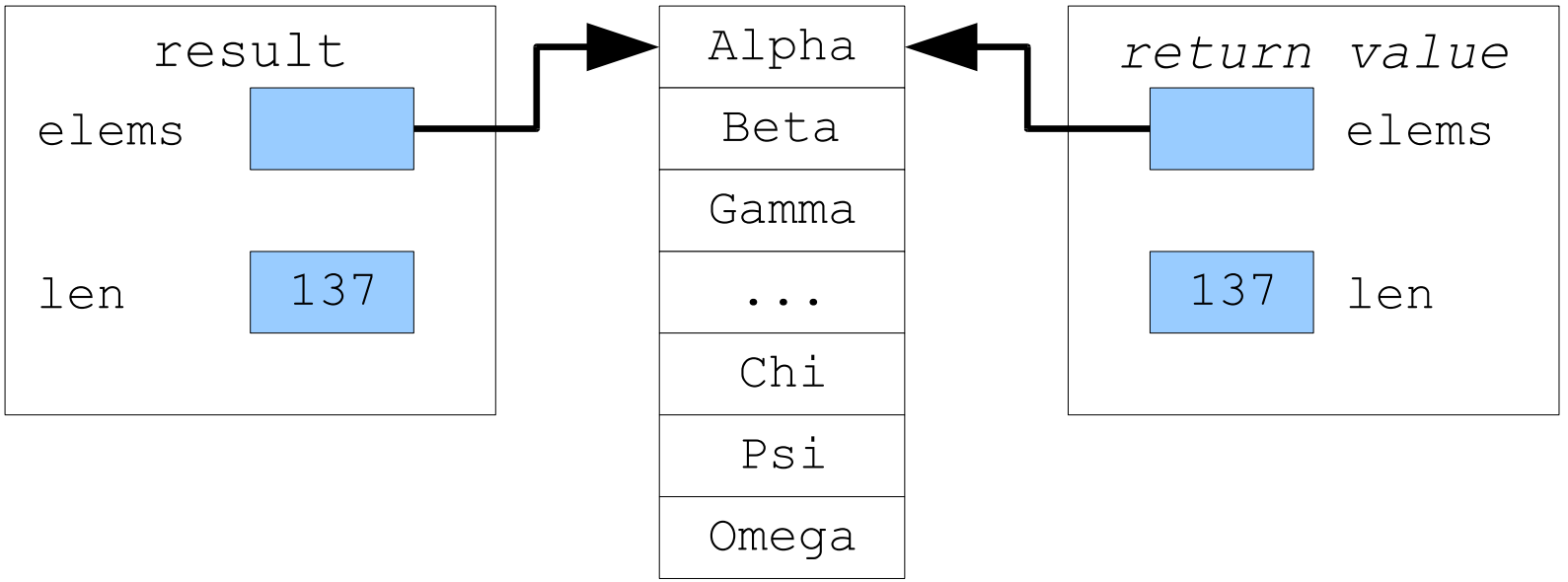
```
vector<string> ReadAllWords(const string& filename)
{
    ifstream input(filename);

    vector<string> result;
    result.insert(result.begin(),
                  istream_iterator<string>(input),
                  istream_iterator<string>());
    return result;
}
```

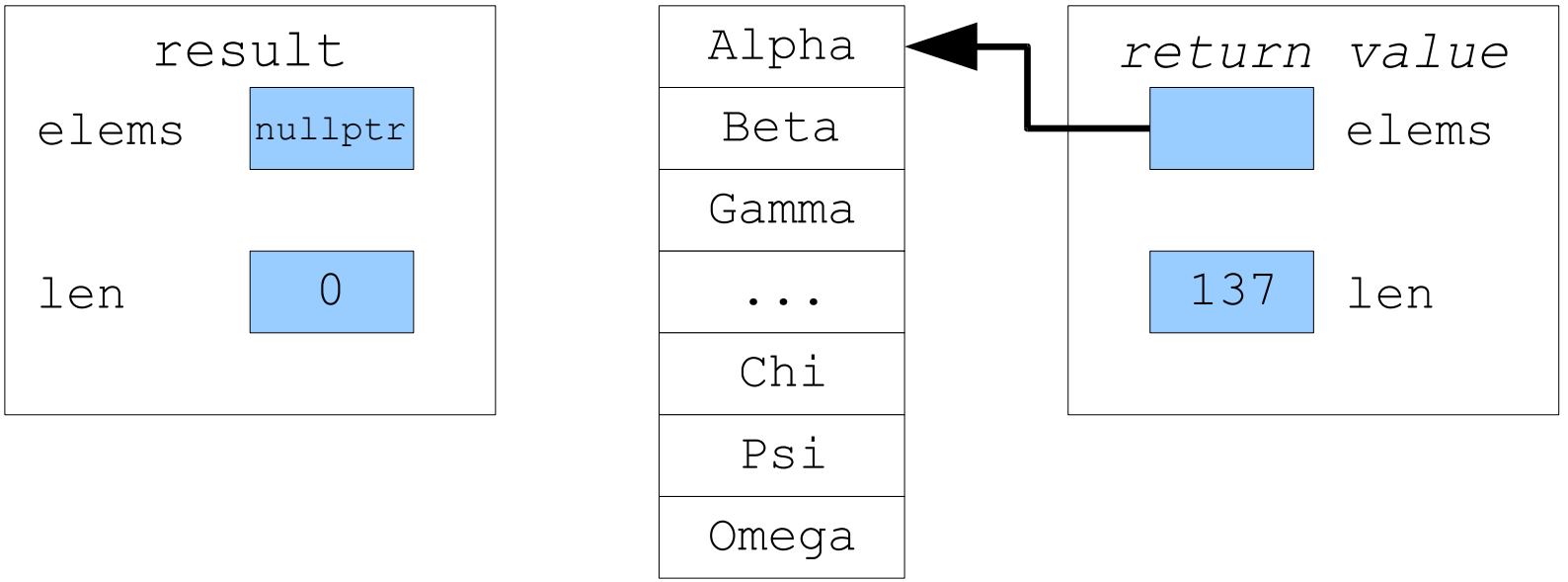
With C++0x:

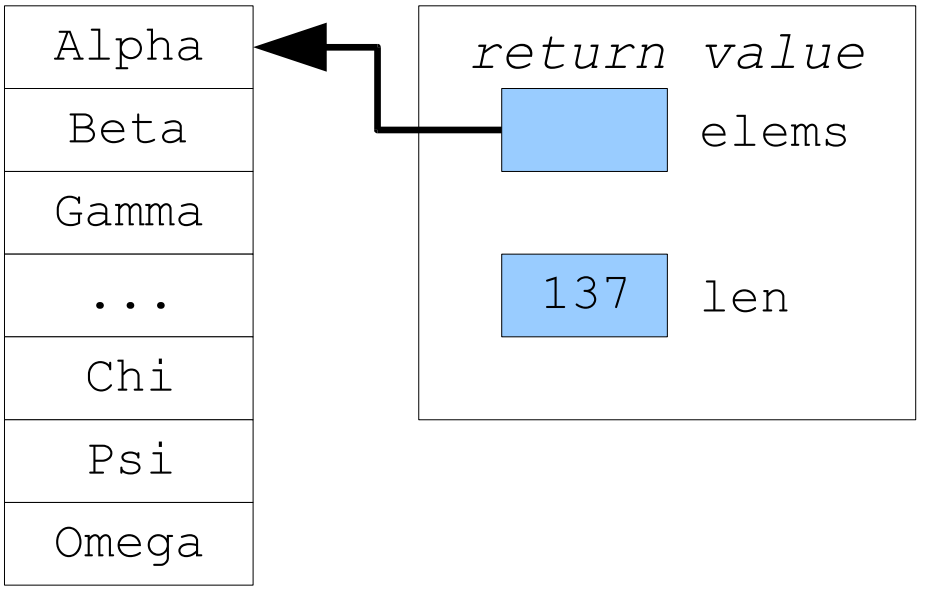
```
vector<string> ReadAllWords(const string& filename)
{
    ifstream input(filename);

    vector<string> result;
    result.insert(result.begin(),
                  istream_iterator<string>(input),
                  istream_iterator<string>());
    return result;
}
```

No major change to the code...


result
elems
len
137
Alpha
Beta
Gamma
...
Chi
Psi
Omega


result
elems
len
137
Alpha
Beta
Gamma
...
Chi
Psi
Omega
return value
elems
137
len


result
elems
nullptr
len
0
Alpha
Beta
Gamma
...
Chi
Psi
Omega
return value
elems
137
len


Alpha
Beta
Gamma
...
Chi
Psi
Omega
return value
elems
137
len

Move Semantics

- *Copy Semantics* (C++03): Can duplicate an object.
  - Copy constructor and *copy* assignment operator
- *Move Semantics* (C++0x): Can move an object from one location to another
  - *Move constructor* and *move assignment operator*
- Move semantics gives asymptotically better performance in many cases.

Rvalue Reference

- Syntax: Type &&
  - e.g. vector&&, string&&, int&&
- Reference to a temporary variable.
- Represents a variable whose contents can be moved from one location to another.

With C++0x:

```
/* Move constructor */
template <typename T>
vector<T>::vector(vector&& other)
{
   elems = other.elems;
   len   = other.len;

   other.elems = nullptr;
   other.len   = 0;
}
```

With C++0x:

```
/* Move constructor */
template <typename T>
vector<T>::vector(vector&& other)
{
    elems = other.elems;
    len   = other.len;

    other.elems = nullptr;
    other.len   = 0;
}
```

With C++0x:

```
/* Move constructor */
template <typename T>
vector<T>::vector(vector&& other)
{
    elems = other.elems;
    len   = other.len;

    other.elems = nullptr;
    other.len   = 0;
}
```

With C++0x:

```
/* Move constructor */
template <typename T>
vector<T>::vector(vector&& other)
{
    elems = other.elems;
    len   = other.len;

    other.elems = nullptr;
    other.len   = 0;
}
```

With C++0x:

```
/* Move assignment operator. */
template <typename T> vector<T>&
vector<T>::operator =(vector&& other)
{
    if(this != &other)
    {
        delete [] elems;
        elems = other.elems;
        len   = other.len;

        other.elems = nullptr;
        other.len   = 0;
    }
    return *this;
}
```

With C++0x:

```
/* Move assignment operator. */
template <typename T> vector<T>&
vector<T>::operator =(vector&& other)
{
    if(this != &other)
    {
        delete [] elems;
        elems = other.elems;
        len   = other.len;

        other.elems = nullptr;
        other.len   = 0;
    }
    return *this;
}
```

With C++0x:

```
void MyClass::moveOther(MyClass&& other)
{
   /* Move all elements, then empty other. */
}
void MyClass::clear()
{
   /* Deallocate memory. */
}
MyClass::MyClass(MyClass&& other)
{
   moveOther(move(other));
}
MyClass& MyClass::operator= (MyClass&& other)
{
   if(this != &other)
   {
       clear();
       moveOther(move(other));
   }
   return *this;
}
```

With C++0x:

```
void MyClass::moveOther(MyClass&& other)
{
   /* Move all elements, then empty other. */
}
void MyClass::clear()
{
   /* Deallocate memory. */
}
MyClass::MyClass(MyClass&& other)
{
   moveOther(move(other));
}
MyClass& MyClass::operator= (MyClass&& other)
{
   if(this != &other)
   {
       clear();
       moveOther(move(other));
   }
   return *this;
}
```

Move Semantics ≠ Copy Semantics

- Old-style copying behavior is still legal.
  - Can still return objects by copying.
  - C++0x tries move semantics first, falls back to copying.
- Objects can be movable without being copyable.

Without C++0x:

```
void OpenFile(ofstream& toOpen)
{
    while(true)
    {

        cout << "Enter filename: ";
        toOpen.open(GetLine().c_str());

        if(!toOpen.fail()) return;

        toOpen.clear();
        cout << "ohnoez no fielz!" << endl;
    }
}
```

With C++0x:

```
ofstream OpenFile()
{
    while(true)
    {
        ofstream toOpen;
        cout << "Enter filename: ";
        toOpen.open(GetLine());

        if(!toOpen.fail()) return toOpen;


        cout << "ohnoez no fielz!" << endl;
    }
}
```

Summary of Move Semantics

- Makes existing code **more efficient**.
- Makes future code **easier to understand**.
- Primarily for library designers, but extremely useful nonetheless.

Major Language Changes, pt. 3

Without C++0x:

```
pair
make_pair
```

With C++0x:

```
pair
make_pair

tuple
make_tuple
tuple_cat
tie
```

Without C++0x:

```
bind1st
bind2nd

ptr_fun
mem_fun
mem_fun_ref

not1
not2
```

With C++0x:

`bind1st`	**`bind`**
`bind2nd`	
`ptr_fun`	**`function`**
`mem_fun`	**`mem_fn`**
`mem_fun_ref`	
`not1`	
`not2`	

Without C++0x:

```
vector                  set
deque                   multiset
list

stack                   map
queue                   multimap
priority_queue

bitset
```

With C++0x:

```
vector                  set
deque                   multiset
list                    unordered_set
forward_list            unordered_multiset
array

stack                   map
queue                   multimap
priority_queue          unordered_map
                        unordered_multimap
bitset
```

Without C++0x:

```
accumulate
adjacent_difference
adjacent_find
binary_search
copy
copy_backward
count
count_if
equal
equal_range
fill
fill_n
find
find_end
find_first_of
find_if
for_each
generate
generate_n
includes
inner_product
inplace_merge
iter_swap
lexicographical_compare
lower_bound
make_heap
max
max_element
merge
min
min_element
mismatch
next_permutation
nth_element
partial_sort
partial_sort_copy
partial_sum
partition
pop_heap
prev_permutation
push_heap
random_shuffle
remove
remove_copy
remove_copy_if
remove_if
replace
replace_copy
replace_copy_if
replace_if
reverse
reverse_copy
rotate
rotate_copy
search
search_n
set_difference
set_intersection
set_symmetric_difference
set_union
sort
sort_heap
stable_partition
upper_bound
stable_sort
swap
swap_ranges
transform
unique
unique_copy
```

With C++0x:

accumulate
adjacent_difference
adjacent_find
binary_search
copy
copy_backward
count
count_if
equal
equal_range
fill
fill_n
find
find_end
find_first_of
find_if
for_each
generate
generate_n
includes
inner_product
inplace_merge
iter_swap
lexicographical_compare
lower_bound
make_heap
max
max_element
merge
min
min_element
mismatch
next_permutation
nth_element
partial_sort
partial_sort_copy
partial_sum
partition
pop_heap
prev_permutation
push_heap
random_shuffle
remove
remove_copy
remove_copy_if
remove_if
replace
replace_copy
replace_copy_if
replace_if
reverse
reverse_copy
rotate
rotate_copy
search
search_n
set_difference
set_intersection
set_symmetric_difference
set_union
sort
sort_heap
stable_partition
upper_bound
stable_sort
swap
swap_ranges
transform
unique
unique_copy

all_of
any_of
none_of
find_if_not
copy_n
copy_if
move
move_backward
is_partitioned
partition_copy
partition_point
is_sorted
is_heap
is_heap_until
minmax
minmax_element
iota

Supercharging `namespace std`

- Major extensions to existing libraries
  - STL, streams, numerics, etc.
- Increases overall complexity of language.
- Makes language easier to use.

And a few more things...

Multithreading

Multithreading

```
vector<string> ReadFile(string filename);

void LoadAllData(vector<string>& longLatData,
                 vector<string>& geoCodes);
{
   auto llReader  = async(ReadFile, "long-lat.txt");
   auto geoReader = async(ReadFile, "geo-codes.txt");

   longLatData = llReader.get();
   geoReader   = geoReader.get();
}
```

Lambda Expressions

Lambda Expressions

```
/* Prints out all long words. */
copy_if(v.begin(), v.end(),
        ostream_iterator<string>(cout, "\n"),
        [](string str) -> bool
        {
             return str.size() > 6
        });
```

Other C++0x Features

- Variadic Templates
- Smart Pointers
- Random Number Generators
- Regular Expression Matchers
- Compile-Time Arithmetic
- Generalized Constant Expressions
- *Plus a whole lot more.*

Summary of C++0x

- C++0x will make programmers more efficient.
  - Type inference and ranged-based for loops will greatly enhance programmer productivity.
  - Enhanced STL makes it easier to write good code.
  - Move semantics make functions clearer.
- C++0x will make good libraries easy to write.
  - Move semantics allow for very elaborate library design.
- Synergy!