

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());

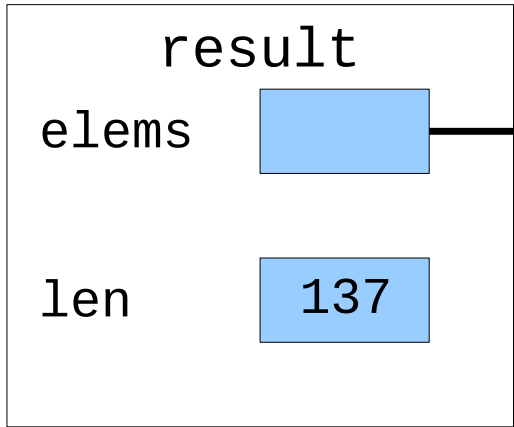
    vector<string> result;
    result.insert(result.begin(),
                  istream_iterator<string>(input),
                  istream_iterator<string>());
    return result;
}
```

Without C++0x:

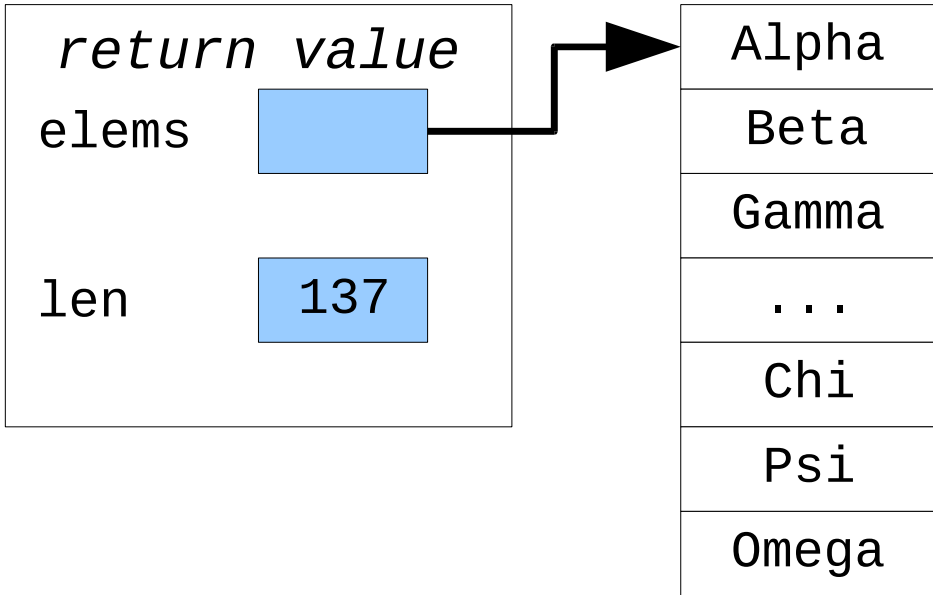
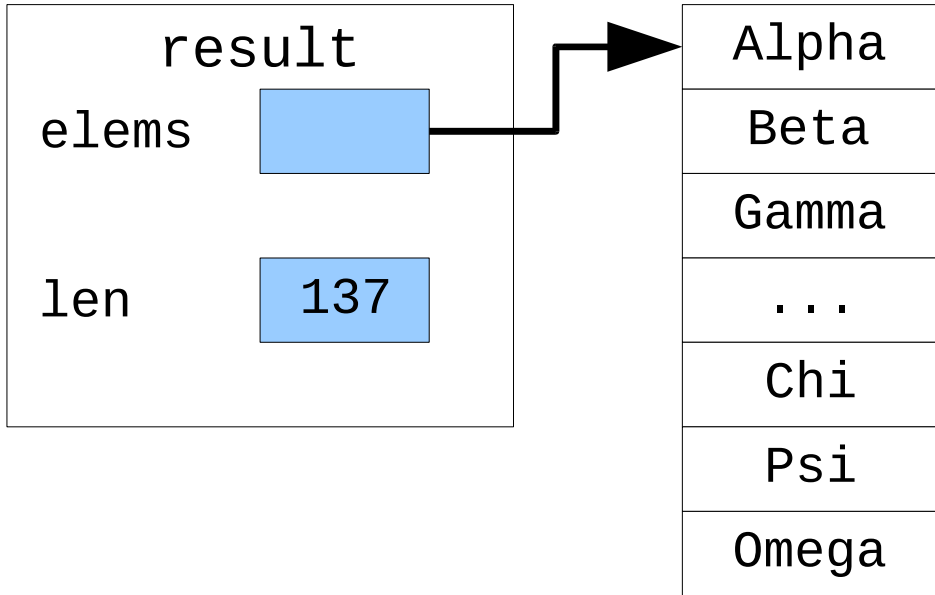
```
vector<string> ReadAllWords(const string& filename)
{
    ifstream input(filename.c_str());

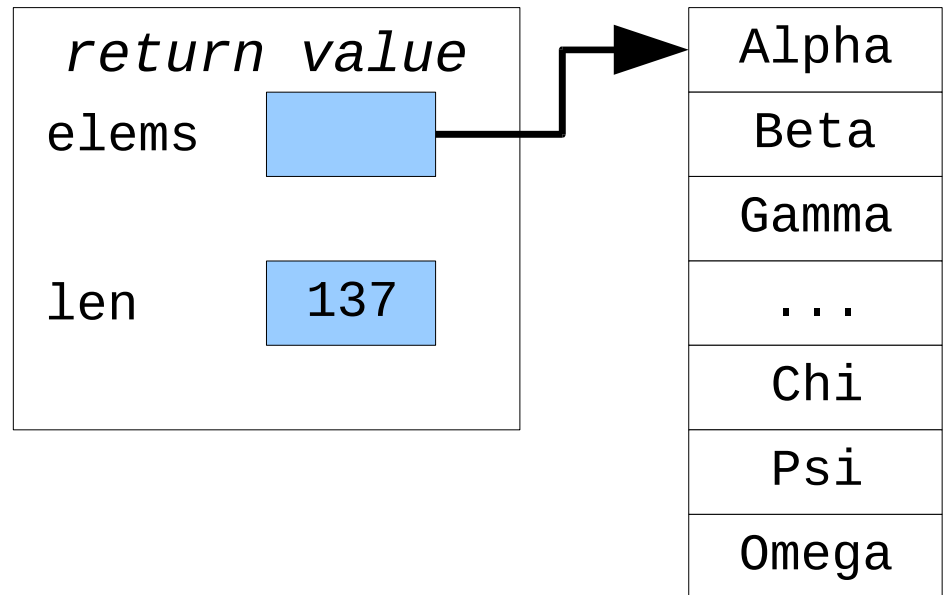
    vector<string> result;
    result.insert(result.begin(),
                  istream_iterator<string>(input),
                  istream_iterator<string>());
    return result;
}
```

How efficient is this code?



- 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);

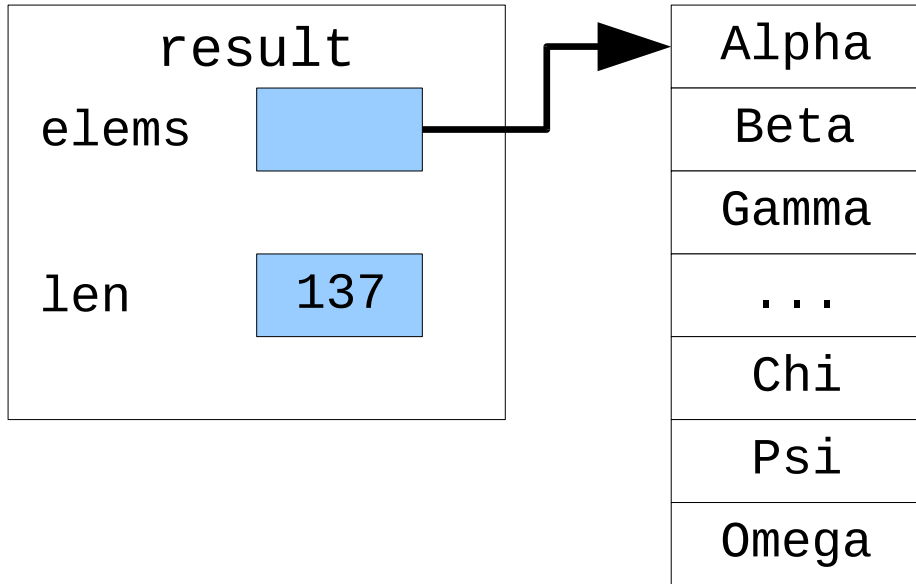
    vector<string> result;
    result.insert(result.begin(),
                  istream_iterator<string>(input),
                  istream_iterator<string>());
    return result;
}
```

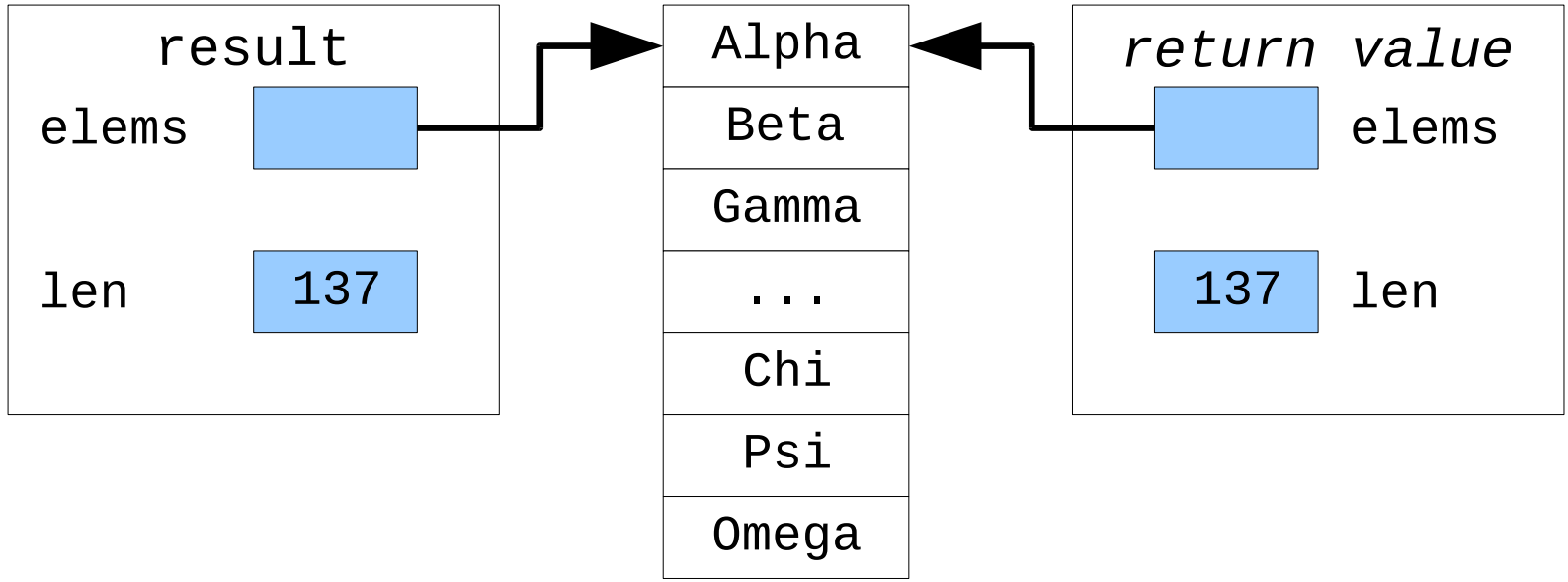
With C++0x:

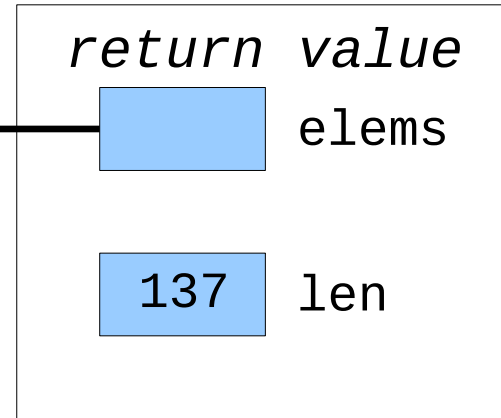
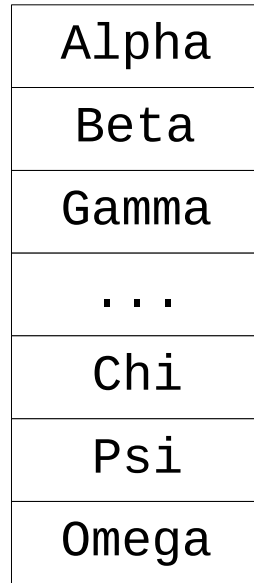
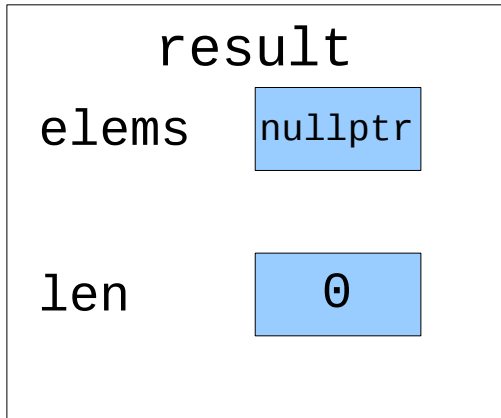
```
vector<string> ReadAllWords(const string& filename)
{
    ifstream input(filename);

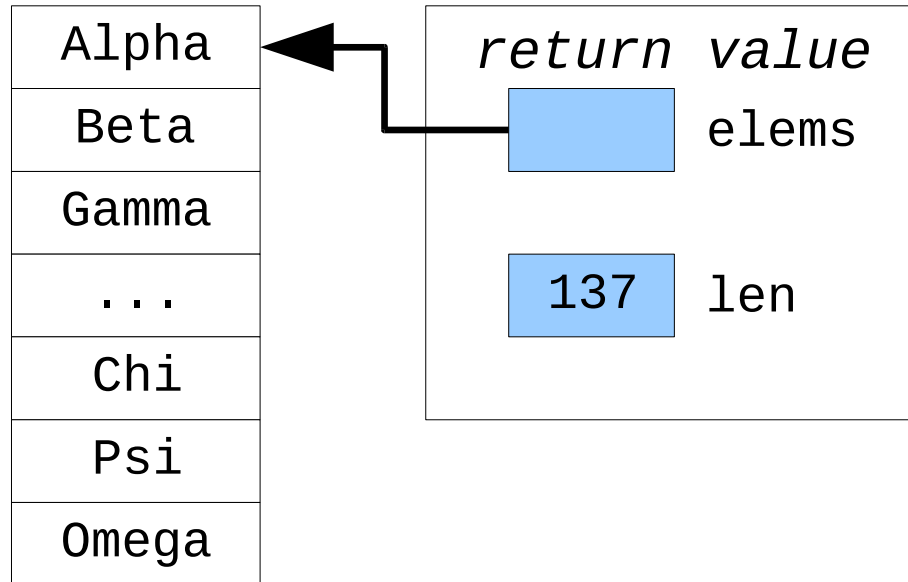
    vector<string> result;
    result.insert(result.begin(),
                  istream_iterator<string>(input),
                  istream_iterator<string>());
    return result;
}
```

No major change to the code...









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 \neq 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
bind2nd

bind

ptr_fun
mem_fun
mem_fun_ref

function
mem_fn

not1
not2

Without C++0x:

vector
deque
list

set
multiset

stack
queue
priority_queue

map
multimap

bitset

With C++0x:

vector

deque

list

forward_list

array

stack

queue

priority_queue

bitset

set

multiset

unordered_set

unordered_multiset

map

multimap

unordered_map

unordered_multimap

Without C++0x:

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

With C++0x:

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

Without C++0x

rand

With C++0x

rand

linear_congruential_engine
mersenne_twister
subtract_with_carry_engine
discard_block_engine
independent_bits_engine
shuffle_order_engine
random_device

uniform_int_distribution
uniform_real_distribution
bernoulli_distribution
geometric_distribution
negative_binomial_distribution
poisson_distribution
exponential_distribution
gamma_distribution
weibull_distribution
extreme_value_distribution
normal_distribution
lognormal_distribution
chi_squared_distribution
cauchy_distribution
fisher_f_distribution
student_t_distribution
discrete_distribution
piecewise_constant_distribution
general_pdf_distribution

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
- Metaprogramming Traits Classes
- 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!