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

- Allows iteration over ranges defined by iterators or pointers.
- Syntax: `for(type var : range) {...}`
- Works on pairs of iterators, STL containers, raw arrays.
- Can define your own ranges using *concept maps*, a new template feature.

Major Language Changes, pt. 2

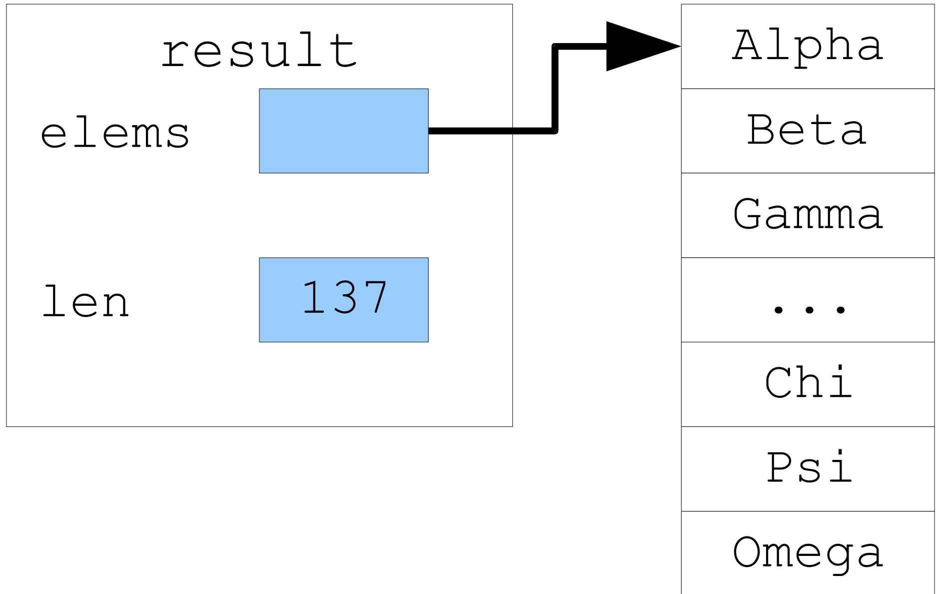
Without C++0x:

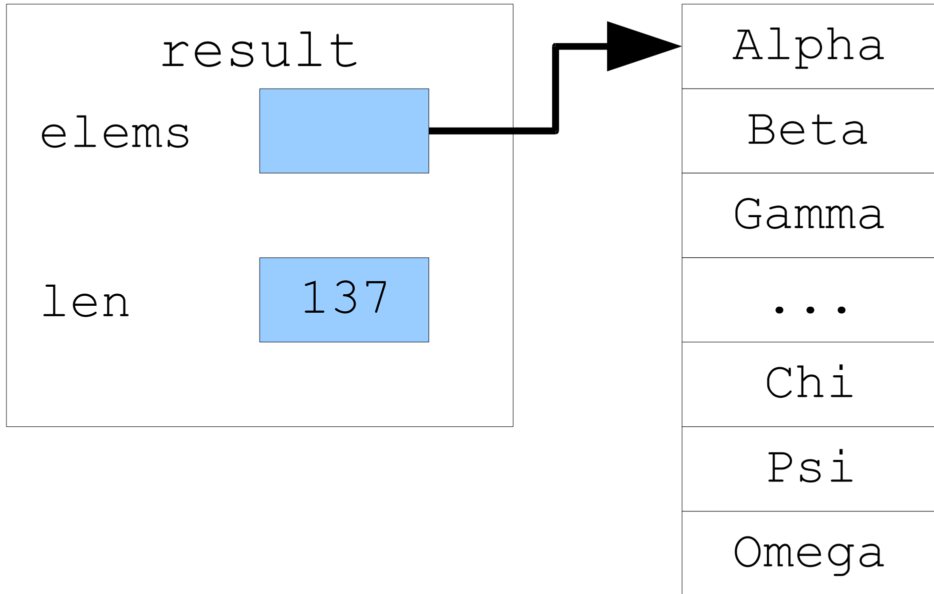
```
vector<string> ReadAllWords(const string& filename)
{
    ifstream input(filename.c_str());
    if(!input.is_open()) throw runtime_error();

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

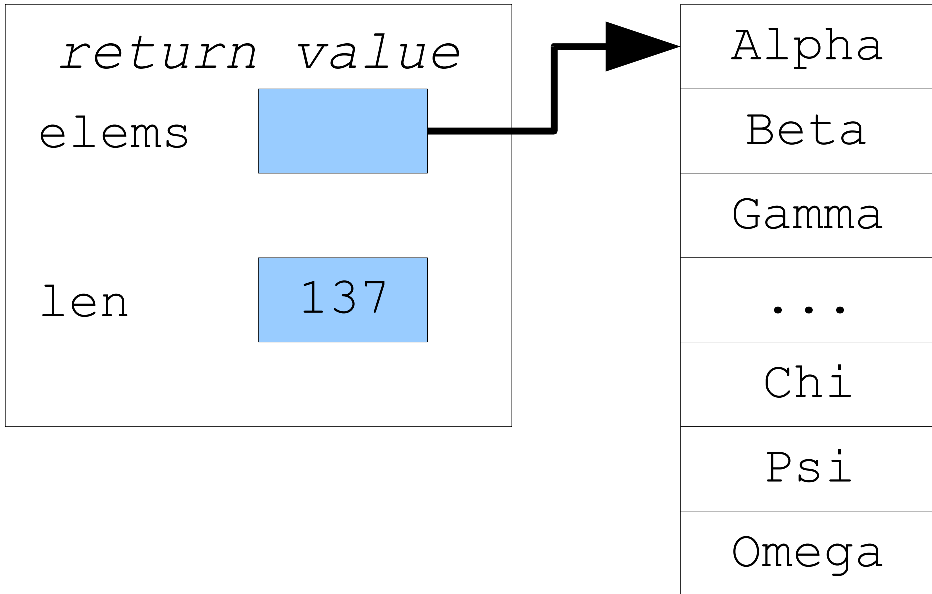
Without C++0x:

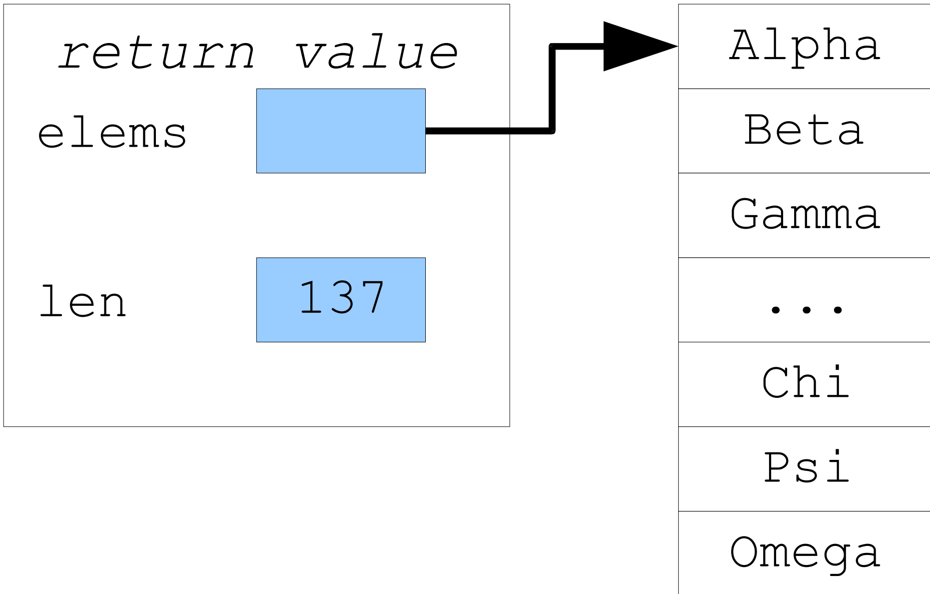
```
vector<string> ReadAllWords(const string& filename)
{
    ifstream input(filename.c_str());
    if(!input.is_open()) throw runtime_error();

    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

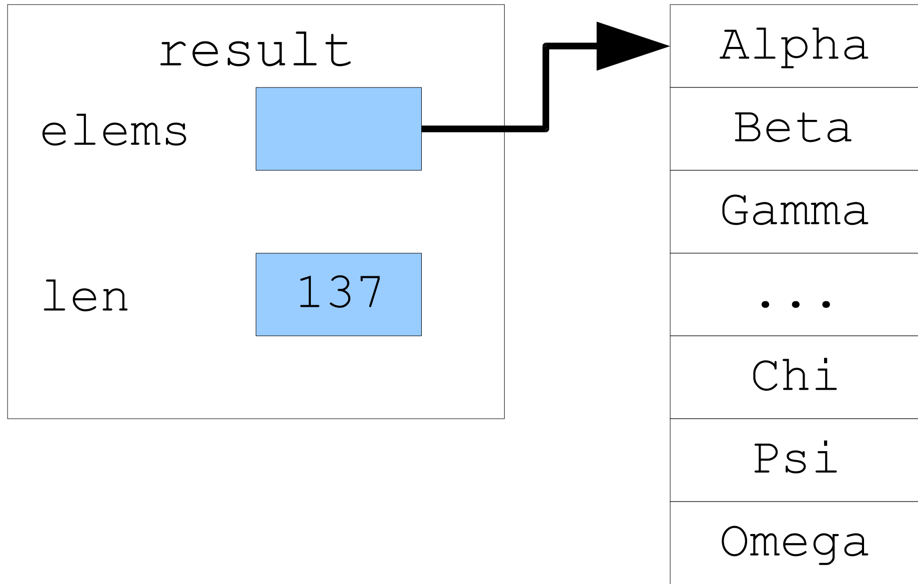
With C++0x:

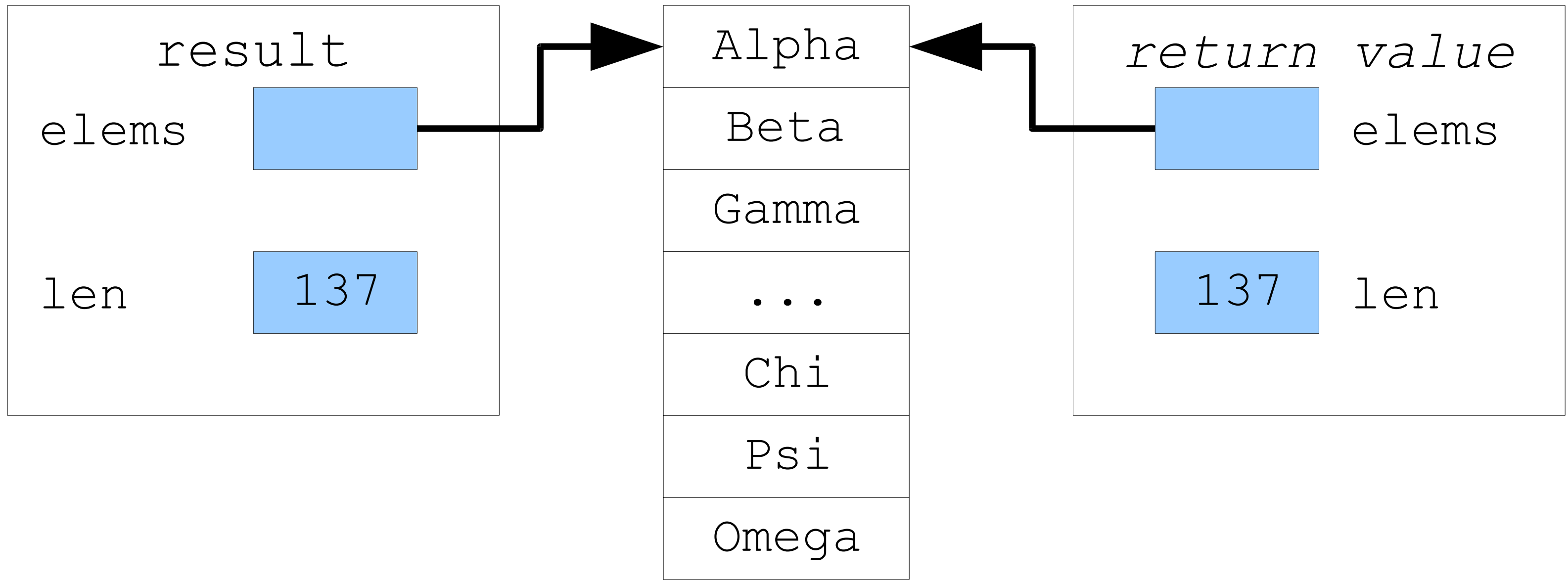
```
vector<string> ReadAllWords(const string& filename)
{
    ifstream input(filename.c_str());
    if(!input.is_open()) throw runtime_error();

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

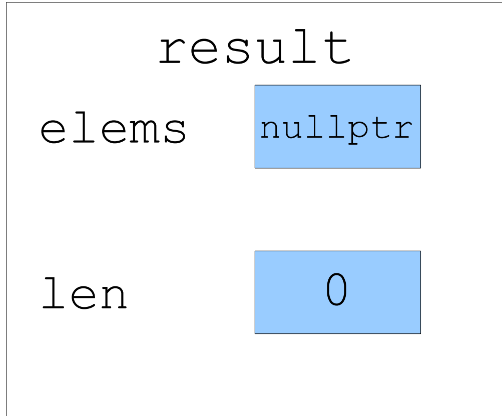
With C++0x:

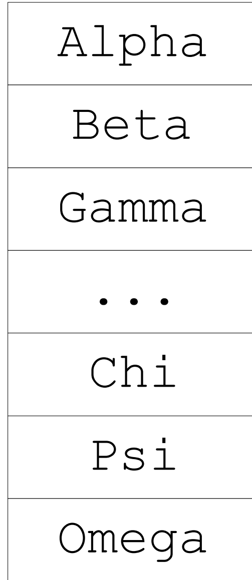
```
vector<string> ReadAllWords(const string& filename)
{
    ifstream input(filename.c_str());
    if(!input.is_open()) throw runtime_error();

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

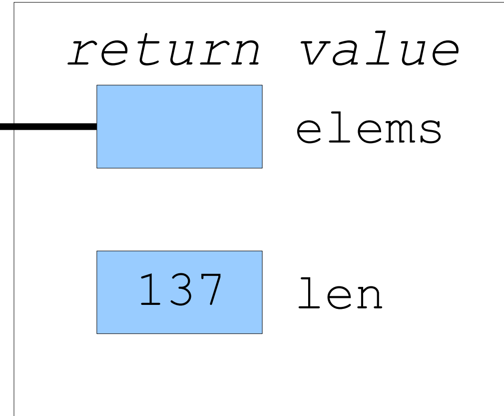
No change to the code...

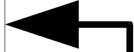

result
elems
len
137
Alpha
Beta
Gamma
...
Chi
Psi
Omega

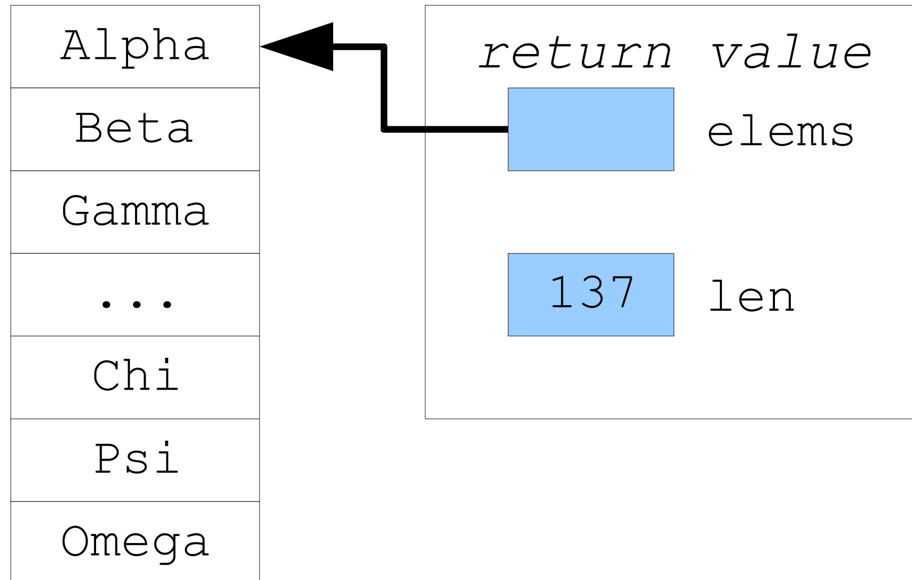

result
elems
len
137
Alpha
Beta
Gamma
...
Chi
Psi
Omega
return value
elems
137
len


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 will try to use move semantics first, but will fall back on copying if it needs to.
- Objects can be movable without being copyable.

Without C++0x:

```
void OpenFile(ostream& 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().c_str());

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


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

Major Language Changes, pt. 3

Without C++0x:

```
template <typename UnaryFunction>
void TabulateValues(double start, double stop, UnaryFunction fn)
{
    const double step = (stop - start) / (NUM_STEPS - 1);
    for(int i = 0; i < NUM_STEPS; ++i)
    {
        const double value = start + step * i;
        cout << "f(" << value << ") = " << fn(value) << endl;
    }
}
```

Without C++0x:

```
template <typename UnaryFunction>
void TabulateValues(double start, double stop, UnaryFunction fn)
{
    const double step = (stop - start) / (NUM_STEPS - 1);
    for(int i = 0; i < NUM_STEPS; ++i)
    {
        const double value = start + step * i;
        cout << "f(" << value << ") = " << fn(value) << endl;
    }
}

TabulateFunctionValues(0, 1, sqrt);
```

Without C++0x:

```
template <typename UnaryFunction>
void TabulateValues(double start, double stop, UnaryFunction fn)
{
    const double step = (stop - start) / (NUM_STEPS - 1);
    for(int i = 0; i < NUM_STEPS; ++i)
    {
        const double value = start + step * i;
        cout << "f(" << value << ") = " << fn(value) << endl;
    }
}

TabulateFunctionValues(0, 1, sqrt);

TabulateFunctionValues(0, 1, bind2nd(multiplies<double>(), 2.7));
```

Without C++0x:

```
template <typename UnaryFunction>
void TabulateValues(double start, double stop, UnaryFunction fn)
{
    const double step = (stop - start) / (NUM_STEPS - 1);
    for(int i = 0; i < NUM_STEPS; ++i)
    {
        const double value = start + step * i;
        cout << "f(" << value << ") = " << fn(value) << endl;
    }
}

TabulateFunctionValues(0, 1, sqrt);

TabulateFunctionValues(0, 1, bind2nd(multiplies<double>(), 2.7));

TabulateFunctionValues(0, 1, "I want to be tabulated too!");
```

Implicit Interfaces

- Some templates only work on specific types.
  - Recall discussion of functors
- Constraints form an implicit interface.
- Can we formalize this?

With C++0x:

```
auto concept RealNumberFunction<typename T>
{
    double operator () (double);
}

template <RealNumberFunction UnaryFunction>
void TabulateValues(double start, double stop, UnaryFunction fn)
{
   const double step = (stop - start) / (NUM_STEPS - 1);
   for(int i = 0; i < NUM_STEPS; ++i)
   {
      const double value = start + step * i;
      cout << "f(" << value << ") = " << fn(value) << endl;
   }
}
```

With C++0x:

```
auto concept RealNumberFunction<typename T>
{
    double operator () (double);
}

template <RealNumberFunction UnaryFunction>
void TabulateValues(double start, double stop, UnaryFunction fn)
{
    const double step = (stop - start) / (NUM_STEPS - 1);
    for(int i = 0; i < NUM_STEPS; ++i)
    {
        const double value = start + step * i;
        cout << "f(" << value << ") = " << fn(value) << endl;
    }
}
```

With C++0x:

```
auto concept RealNumberFunction<typename T>
{
    double operator () (double);
}

template <RealNumberFunction UnaryFunction>
void TabulateValues(double start, double stop, UnaryFunction fn)
{
   const double step = (stop - start) / (NUM_STEPS - 1);
   for(int i = 0; i < NUM_STEPS; ++i)
   {
      const double value = start + step * i;
      cout << "f(" << value << ") = " << fn(value) << endl;
   }
}
```

C++0x Concepts

- Perhaps the single largest addition to C++.
- Formally specify what behaviors are expected out of a template type.
- Zero runtime cost.
- Allow static checking of template body.

Other C++0x Features

- Lambda Expressions
- Concurrency
- Variadic Templates
- Smart Pointers
- Random Number Generators
- 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.
  - Plus lots of other minor additions.
- C++0x will make good libraries easy to write.
  - Concepts are a library designer's dream.
  - Move semantics allow for very elaborate library design.
- Synergy!