# C++0x Today: Features for Building Better Libraries


Douglas Gregor, Assistant Director

Open Systems Lab @ Indiana University

# What is C++0x?

- Upcoming revision of the C++ standard
- Goals:
  - Make C++ easier to teach and learn
  - Make C++ a better language for systems programming and library-building
  - Maintain backward compatibility

# C++0x Status

- ISO standardization process:
  - Feature complete in June
  - Complete draft out for voting in October
  - ISO standard by the end of 2009 (we hope)
- Partial implementations coming online

EDG C++ 3.10

GCC 4.3

CodeWarrior 10

# Ground Rules

- Every feature I describe is in the current C++0x Working Paper (N2588)
  - Very likely to be in C++0x standard, but...
- Some features still might be in C++0x:
  - Concepts
  - Generalized initializer lists
  - Attributes

# A Cautionary Tale

- The smallest addition to C++0x...

```
vector<vector<double>> matrix;
```

No space!

- ... breaks backward compatibility!

```
template<int I> struct X {  static int const c = 2;};
template<> struct X<0> {  typedef int c;};
template<typename T> struct Y {  static int const c = 3;};
static int const c = 4;
int main() {
  std::cout << (Y<X< 1>>::c >::c>::c) << '\n';
} // C++98: prints 3, C++0x: prints 0
```

Changes from right-shift to closing angle-brackets

# C++0x Library Features: Outline

- Types and Type Deduction
- Rvalue references
- Variadic templates
- Miscellaneous features
- Moving forward...

# Types and Type Deduction

References:

- J. Järvi, B. Stroustrup, and G. Dos Reis. *Deducing the type of a variable from its initializer expression.* N1984=06-0054.
- J. Järvi, B. Stroustrup, and G. Dos Reis. *Decltype (revision 7): proposed wording*. N2343=07-0203.
- J. Merrill. *New Function Declarator Syntax Wording.* N2541=08-0051.

# Deducing a Variable's Type

- Writing types can be a real drag:

```
for(std::vector<Employee>::iterator e =
   employees.begin(); e != employees.end(); ++e) {
  // Do some large operation on each employee
}
```

- `auto` allows the deduction of a variable's type from its initializer:

```
for(auto e = employees.begin();
    e != employees.end(); ++e) {
  // Do some large operation on each employee
}
```

# How Does Deduction Work?

- Can use `auto` wherever a normal type specifier would work:

```
auto &e = employees.front();
auto *p = &employees[17];
```

- `auto` deduction is template argument deduction:

```
template<typename AUTO> void deduce(AUTO *p);
deduce(&employees[17]);
```

# Where Can You Use **auto**?

- `auto` can be used for
  - local variables,
  - global variables,
  - variables defined in `for` and `if` statements,
  - and in-class definitions of static data members.

# What You Can't Do With **auto**

- These aren't allowed:

```
void foo(auto x, auto y); // ill-formed

// ill-formed
template<typename T, typename U>
  auto add(T x, U y) { return x + y; }
```

# Deducing an Expression Type

- `decltype(e)` provides the type of an expression:
  - If `e` is an id-expression, `decltype(e)` is the type of the entity named,
  - If `e` is a function call, `decltype(e)` is the return type of the function called,
  - Otherwise, `T` is the type of `e`, and
    - if `e` is an lvalue, `decltype(e)` is `T&`,
    - if `e` is an rvalue, `decltype(e)` is `T`

# `decltype` Examples

- Given:

```
int& get_ith(std::vector<int>& v, int i);
int x; char c;
std::vector<int> v;
```

- We have the following deductions:
  - `decltype(x)` → `int`
  - `decltype(x + c)` → `int`
  - `decltype((x))` → `int&`
  - `decltype(get_ith(v, x))` → `int&`

# Where’s My `typeof`?

- `decltype` is the C++0x answer to the need for `typeof`
- Important distinction: the type of an expression does not involve references
  - Rather, we talk about lvalues/rvalues
- `typeof(e) == remove_reference<decltype(e)>::type`

# New Function Declarator Syntax

- Remember the **add** function, where we wanted to deduce the return type?
- Can use the new syntax for declaring functions:

```
template<typename T, typename U>
  auto add(T x, U y) -> decltype (x + y)
  {
    return x + y;
  }
```

# **auto** Backward Compatibility

- **auto** has lost its old meaning:
  ```
  auto int i = 17;
  ```

- And its really, really old meaning:
  ```
  auto f = 3.14159;
  ```

- One of the few places the committee has knowingly broken compatibility

# Extended SFINAE Cases

- **S**ubstitution **F**ailure **I**s **N**ot **A**n **E**rror is getting more powerful
  - Extended to arbitrary expressions

```
template<typename T, typename U>
  auto add(T x, U y) -> decltype (x + y); // #1
void int add(...); // #2


int C::* pm;
add(pm, pm); // okay: calls #2
```

# Limits on SFINAE

- SFINAE still doesn’t cope with failed instantiations:

```
template<typename T, typename U = T>
struct add_result {
  typedef decltype(T() + U()) type;
}
template<typename T, typename U>
 typename add_result<T, U>::type add(T x, U y); // #1
void int add(...); // #2

int C::* pm;
add(pm, pm); // error: instantiation failure in #1
```

# Recap – Types and Deduction

- Use `auto` to avoid writing long types
- Use `decltype` to determine the type of an expression
- Use the new function declarator syntax with `decltype` for forwarding functions
- Extended SFINAE makes more type inspection possible [*].

[*] Concepts provide more advanced type inspection.

# Effects on Library Design

- Type deduction becomes trivial; user intervention no longer required.
- Extended SFINAE allows more type inspection for better, more adaptive libraries.

# Rvalue References

References:


- H. Hinnant, B. Stroustrup, B. Kozicki. *A Brief Introduction to Rvalue References*. N2027=06-0097.
- H. Hinnant, D. Abrahams, P. Dimov. *A Proposal to Add an Rvalue Reference to the C++ Language.* N1690=04-0130.
- H. Hinnant. *A Proposal to Add an Rvalue Reference to the C++ Language: Proposed Wording (Revision 2).* N1952=06-0022.

# Rvalue References

- A new kind of reference (**&&**) that binds to rvalues. Two major uses:
  - *Move semantics*: allows one to “move” a value from one variable to another
  - *Argument forwarding*: allows one to accept any argument and forward it to another function

# Terminology

- *lvalue*: an expression that refers to an object in memory (that has a name)
- *rvalue*: an expression that refers to a temporary value
- A reference type is `T&` or `T&&`
  - `T&` is an lvalue reference type
  - `T&&` is an rvalue reference type

# Move Semantics

- Why is this slow?

```
struct Vector {
  float* data;
  int length;
  // ...
};
Vector operator+(Vector const& u, Vector const& v);

Vector u, v, w; // initialize u, v
w = u + v; // slow!
```

We copy resources that will be destroyed anyway!

# Copy vs. Move Assignment

- Copy assignment:

```
Vector& Vector::operator=(Vector const& v) {
  data = new float[v.length]; length = v.length;
  for (int i = 0; i < length; ++i)
    data[i] = v.data[i];
}
```

- Move assignment:

```
Vector& Vector::operator=(Vector&& v) {
 data = v.data; length = v.length;
 v.data = 0; v.length = 0;
}
```

# Cannibalizing Temporaries

```
class Vector {
public:
  Vector(Vector const&); // copy constructor
  Vector(Vector&&); // move constructor
  Vector& operator=(Vector const&); // copy assign
  Vector& operator=(Vector&&); // move assign
};
```

- Copy vs. move:

```
Vector u, v, w; // initialize u, v
w = u; // calls copy-assignment operator
w = u + v; // calls move-assignment operator
```

# Overloading Rvalue References

- Basic overloading rules:
  - lvalues can bind to any kind of reference (but prefer lvalue references)
  - rvalues can bind to an rvalue reference or a const-qualified lvalue reference

# Overloading Examples

```
struct X {};

void f(X&&); // #1
void f(const X&); // #2

void h(X x) {
  f(X()); // both #1 and #2 apply; prefers #1
  f(x); // both #1 and #2 apply; prefers #2
}
```

# Overloading Examples

```
struct X {};

void f(X&&); // #1
void f(const X&); // #2

void h(X x) {
  f(X()); // both #1 and #2 apply; prefers #1
  f(x); // both #1 and #2 apply; prefers #2
}
```

# Re-using Temporaries

- `Vector`'s `operator+` builds temporaries

```
Vector operator+(Vector const& u, Vector const& v);
```

- Could also "re-use" temporaries:

```
Vector operator+(Vector && u, Vector const& v) {
  for (int i = 0; i < u.length; ++i)
    u.data[i] += v.data[i];
  return std::move(u);
}
```

# Re-using Temporaries

- `Vector`'s `operator+` builds temporaries

```
Vector operator+(Vector const& u, Vector const& v);
```

- Could also "re-use" temporaries:

```
Vector operator+(Vector && u, Vector const& v) {
  for (int i = 0; i < u.length; ++i)
    u.data[i] += v.data[i];
  return std::move(u); // move, rather than copy, u
}
```

## Using `std::move`

- lvalues prefer to bind to lvalue references:
  - In "`return u;`", `u` is an lvalue
- `std::move` specifically says: "move this value"
  - In "`return std::move(u);`", `std::move(u)` is an rvalue
- Example forcing move-assign:

```
v = std::move(u);
```

# Improving `swap`

- The basic `swap` makes several copies:

```
template <class T>
void 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)
}
```

# Improving `swap`

- Improved `swap` moves the values:

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

# Move-Only Types: `unique_ptr`

```
template<typename T>
class unique_ptr {
  unique_ptr(unique_ptr const&);
  unique_ptr& operator=(unique_ptr const&);
  T* ptr;
public:
  unique_ptr() : ptr(0) { }
  explicit unique_ptr(T* ptr) : ptr(ptr) { }
  ~unique_ptr() { delete ptr; }
  unique_ptr(unique_ptr&& other);
  unique_ptr& operator=(unique_ptr&& other);
};
```

# Aside: deleted functions

```
template<typename T>
class unique_ptr {
  unique_ptr(unique_ptr const&) = delete;
  unique_ptr& operator=(unique_ptr const&) = delete;
  T* ptr;
public:
  unique_ptr() : ptr(0) { }
  explicit unique_ptr(T* ptr) : ptr(ptr) { }
  ~unique_ptr() { delete ptr; }
  unique_ptr(unique_ptr&& other);
  unique_ptr& operator=(unique_ptr&& other);
};
```

# `unique_ptr` implementation

- Move constructor:

```
unique_ptr::unique_ptr(unique_ptr&& x) : ptr(x.ptr)
  { x.ptr = 0; }
```

- Copy constructor:

```
unique_ptr& unique_ptr::operator=(unique_ptr&& x) {
  delete ptr;
  ptr = x.ptr;
  x.ptr = 0;
  return *this;
}
```

# “Perfect” Forwarding

- Task: try to write a function “forward” that forwards two of its arguments to a function object **f**:

```
template<class F, class T1, class T2>
void forward(F f, T1 const& a1, T2 const& a2) {
  f(a1, a2);
}
```

- Several problems here:
  - Non-const lvalues become const
  - Rvalues become const lvalues

# Perfect Forwarding

- Rvalue references allow perfect forwarding:

```
template<class F, class T1, class T2>
void forward(F f, T1&& a1, T2&& a2) {
  f(std::forward<T1>(a1), std::forward<T2>(a2));
}
```

- Uses special template argument deduction rules for `T1&&`, `T2&&`, reference collapsing

# Template Argument Deduction

- A function parameter of type `T&&` has special deduction rules:
  - Let `X` by the type of the argument
  - If the argument is an lvalue, `T` is `X&`
  - Otherwise, `T` is `X`.
- Example:

```
template<class T1, class T2> void f(T1&&, T2&&);
void g(X x) { f(x, X(); }
```

  - `T1` is `X&`, `T2` is `X`

# Reference Collapsing

- Reference collapsing occurs during instantiation:

```
template<typename T> struct X { typedef T& type; };
```

  - `X<int&>::type` will result in `int&`
- Rvalue reference collapsing rules:
  - `A& &` → `A&`
  - `A&& &` → `A&`
  - `A& &&` → `A&`
  - `A&& &&` → `A&&`

# Perfect Forwarding Revisted

- Rvalue references allow perfect forwarding:

```
template<class F, class T1, class T2>
void forward(F f, T1&& a1, T2&& a2) {
  f(std::forward<T1>(a1), std::forward<T2>(a2));
}
```

- How it works:
  - Template argument deduction determines types
  - Reference collapsing makes actual parameter type match the argument and its l/r-valueness
  - `std::forward` keeps the l/r-valueness

# Perfect Forwarding Is Imperfect

- Forwarding preserves:
  - object type, including `const`, `volatile`
  - l/r-valueness
  - address of the object (no copies)
- Forwarding does not preserve the value of integral constant expressions

```
void f(int*);
template<class T> void g(T&& t) {f(std::forward<T>(t));}
g(0); // error: can't initialize an int* with an int
```

# std::forward and std::move

- Forward forwards an argument:

```
template <class T> struct identity
  { typedef T type; };
template <class T>
  T&& forward(typename identity<T>::type&& a)
  { return a; }
```

- Move turns an argument into an rvalue:

```
template <class T>
  typename remove_reference<T>::type&& move(T&& a)
   { return a; }
```

# Rvalue references recap

- Reference binding rules:
  - An rvalue reference (`&&`) binds to rvalues
  - Lvalue references prefer lvalues,
    rvalue references prefer rvalues
- Use `std::move` to treat values as rvalues
- Use `std::forward` for perfect forwarding

# Effects on Library Design

- If a data structure owns resources and is copyable, it should be moveable.
- Should no longer fear returning containers:

```
vector<string> split(string str, char c);
```

- `auto_ptr`-like hacks can disappear.
- Function object adaptors should forward arguments perfectly.

# Variadic Templates

References:

- D. Gregor. *A Brief Introduction to Variadic Templates.* N2087=06-0157.
- D. Gregor, J. Järvi, and G. Powell. *Variadic Templates (Revision 3)*. N2080=06-0150.
- D. Gregor, J. Järvi, J. Maurer, and J. Merrill. *Proposed Wording for Variadic Templates (Revision 2).* N2242=07-0102.

# Parameterization in Templates

- Most templates have a fixed set of parameters:
  - `vector<T, Allocator>`
  - `copy<InputIterator, OutputIterator>`
  - `pair<T1, T2>`
- What about Boost's `tuple`?
  - `tuple<>`
  - `tuple<int>`
  - `tuple<char, float, string>`

# Tuple, In Brief

- tuple is a generalized pair:

```
tuple<string, int, double> t("Hello", 17, 3.14);
```

- Data access is through get:

```
get<0>(t) == "Hello"
get<1>(t) == 17
```

- `tie()` helps with multiple return values:

```
tuple<float, float> statistics(vector<float>);

tie(mean, stddev) = statistics(grades);
```

# Implementing `tuple`, Today

```
struct unused; // unspecified type

template<typename T1 = unused, typename T2 = unused,
         typename T3 = unused, typename T4 = unused,
         typename T5 = unused, typename T6 = unused,
         typename T7 = unused, typename T8 = unused,
         typename T9 = unused, typename T10 = unused>
  struct tuple;
```

# Problems with Today's `tuple`

- Usability problems:
  - Fixed upper limit on number of arguments
  - Poor error messages:

    "`tuple<int, float, unused, unused, unused, unused, unused, unused, unused, unused>` has no member 'foo'"
  - Code repetition -> longer compile times
- Other parts of Boost/TR1 have the same problems.

# Variadic Templates

- Naturally express templates that accept a variable number of template arguments.
- Benefits:
  - More general way to accept an arbitrary number of template arguments
  - Allows perfect argument forwarding, “inheriting” constructors, etc.
  - Eliminates most preprocessor metaprogramming

# A Variadic Tuple

```
template<typename ... Elements>
  struct tuple;
```

- A *parameter pack* is a new kind of entity
  - Template parameter packs bind to *zero or more* template arguments
  - Introduced with the ellipsis "..." *to the left* of the parameter name.
  - Think of it as "`typename T1, typename T2, …, typename T`*N*"

# Parameter Pack Binding

```
template<typename ... Elements>
  struct tuple { };
```

- `tuple` accepts any number of type arguments

```
tuple<> t0; // Elements is empty
tuple<int> t1; // Elements is int
tuple<int, float> t2; // Elements is
                      //   int, float
```

# Length of a Tuple

**Declaration**

```
template<typename Tuple>
  struct tuple_length;
```

**Basis Case**

```
template<>
struct tuple_length<tuple<> > {
public:
  static const int value = 0;
};
```

**Recursive Case**

```
template<typename Head, typename ... Tail>
struct tuple_length<tuple<Head, Tail ...> > {
public:
  static const int value =
    1 + tuple_length<tuple<Tail ...> >::value;
```

# Pack Expansions

```
template<typename Head, typename ... Tail>
struct tuple_length<tuple<Head, Tail ...> > {
public:
  static const int value =
    1 + tuple_length<tuple<Tail ...> >::value;
};
```

- The ellipsis "..." *to the right* of an argument indicates a *pack expansion*
  - A pack expansion "unpacks" a parameter pack into separate arguments.
  - Think of it as "`T1, T2, …, T`*N*"

# Unraveling `tuple_length`

- Given `tuple<short, int, long>`:
  - **Recursive case**: `Head=short, Tail=int, long`
  - result is 1 + length of `tuple<int, long>`
    - **Recursive case**: `Head=int`, `Tail = long`
    - result is 1 + length of `tuple<int>`
      - **Recursive case**: `Head=long`, `Tail` is empty
      - result is 1 + length of `tuple<>`
        - **Basis case**: result = 0

# Recursive Data Storage

- Basis case:

```
template<> struct tuple<> { };
```

- Recursive case:

```
template<typename Head, typename... Tail>
struct tuple<Head, Tail...> :
   tuple<Tail...>
{
  Head head;
};
```

# Fun with Pack Expansions

- Pack expansions apply to an entire template argument
  - `Tail...` expands to `T1, T2, …, TN`
  - `Tail&...` expands to `T1&, T2&, …, TN&`
  - `typename add_reference<Tail>::type...` expands to

```
typename add_reference<T1>::type,
typename add_reference<T2>::type,
                 …
typename add_reference<TN>::type
```

# Tuple of References

- From `tuple` of types, let's build a `tuple` of references to those types:

```
template<typename Tuple>
  struct add_references;

template<typename... Elements>
struct add_references<tuple<Elements...>> {
  typedef tuple<Elements& ...> type;
};
```

# Sequence Transform

```
template<template<class T> class Metafun,
         typename Sequence>
  struct transform;
```

- Used to transform sequences:

```
  transform<add_reference, tuple<short, int, long>>
    ::type
```

becomes

```
   tuple<short&, int&, long&>
```

# Tuple Transform

```
template<template<class T> class Metafun,
         typename Sequence>
  struct transform;

template<template<class T> class Metafun,
         typename... Elements>
  struct transform<Metafun,
                   tuple<Elements...>> {
    typedef
      tuple<typename Metafun<Elements>::type...>
        type;
  };
```

# Zipping Tuples

- Turn two tuples into a tuple of pairs:

```
zip<tuple<short, int>, tuple<float, double>>::type
```

becomes

```
tuple<pair<short, float>, pair<int, double>>
```

- Implementation:

```
template<typename, typename> struct zip;

template<typename... Elems1, typename... Elems2>
  struct zip<tuple<Elems1...>,tuple<Elem2...> > {
    typedef tuple<pair<Elems1, Elems2>...> type;
  };
```

# Quick Review

- An ellipsis to the left of a template parameter indicates a *parameter pack*
  - Template parameter packs bind to multiple template arguments
- An ellipsis to the right of a template argument indicates a *pack expansion*
  - Template argument pack expansions expand into multiple template arguments

# Variadic Function Templates?

- Say we want to write a simple function that prints all of its arguments:

```
print("Hello", 17, std::string("World"));
```

- Today's solutions are poor:
  1. Overloaded templates for each number of arguments
  2. C-style varargs
  3. Operator overloading tricks

# Function Parameter Packs

```
template<typename... Args>
  void print(Args const&... args);
```

- A *function parameter pack*:
  - Accepts zero or more function arguments
  - Is introduced by an ellipsis *to the left* of the function parameter name
  - Has a type that involves one or more template parameter packs

# Printing Arguments

Basis Case

```
void print() { }
```

Recursive Case

```
template<typename First, typename... Rest>
void print(First const& first, Rest const&... rest)
{
  std::cout << first;
  print(rest...);
```

# Inheriting Constructors

```
class EmployeeRoster : public std::list<Employee> {
  typedef std::list<Employee> inherited;

public:








};
```

# Inheriting Constructors

```
class EmployeeRoster : public std::list<Employee> {
  typedef std::list<Employee> inherited;

public:
  template<typename T1>
    explicit EmployeeRoster(const T1& a1)
      : inherited(a1) { }

  template<typename T1, typename T2>
    EmployeeRoster(const T1& a1, const T2& a2)
      : inherited(a1, a2) { }
};
```

# Inheriting Constructors

```
class EmployeeRoster : public std::list<Employee> {
  typedef std::list<Employee> inherited;

public:
  template<typename ... Args>
    explicit EmployeeRoster(const Args&... args)
      : inherited(args ...) { }




};
```

# Perfect Forwarding

- Use rvalue references:

```
template<typename T1, typename T2>
  EmployeeRoster(T1&& a1, T2&& a2)
    : inherited(std::forward<T1>(a1),
                std::forward<T2>(a2)) { }
```

- ... with variadic templates:

```
template<typename ... Args>
  explicit EmployeeRoster(Args&& ... args)
    : inherited(std::forward<Args>(args)...) { }
```

# Perfect Forwarding Function

- Rvalue references and variadic templates and decltype… *oh my!*

```
template<typename F, typename... Args>
auto forward(F& f, Args&&... args)
      -> decltype(f(std::forward<Args>(args)...))
  {
    return f(std::forward<Args>(args)...);
  }
```

# Where Can We Unpack? (1/2)

- In an argument to a template:
  - `tuple<Args...>`
- In an argument to a function:
  - `print(args...)`
- In a function type's parameter list:
  - `R (*)(Args...)`
- In a special `sizeof` expression:
  - `sizeof…(Args)`

# Where Can We Unpack? (2/2)

- In a base class list:
  - `class MyClass : public Mixins...`
- In a base-class initializer list:
  - ```
    my_class(Args... args)
      : Mixins(args)...
    ```
- In a throw specifier:
  - `throw(Exceptions...)`
- In an initializer list:
  - `any array[] = { args... };`
    ```

# The Most-Requested Feature

```
template<typename... Types>
struct my_tuple {
  Types... members; // ill-formed!
};
```

- C++0x will not have member expansions:
  - It's a syntactic Pandora's Box: given

    `my_tuple<TTypes...>::members`
  - how do we know if "members" is a pack or not?

# Tuples to the Rescue

```
template<typename... Types>
struct my_tuple {
  std::tuple<Types...> members; // okay!
};
```

- Tuple can handle storage of members
  - Can use `get<I>(members)` to extract a specific members
  - Expansion trick: `get<Is>(members)...` where `Is` is a non-type parameter pack

# Review: Variadic Templates

- Use variadic templates to eliminate repeated template parameters
  - Ellipsis *to the left* of a parameter name is a *parameter pack*
  - Ellipsis *to the right* of an argument is a *pack expansion*

# Effects on Library Design

- Should drastically reduce the need for preprocessor metaprogramming.
- Arbitrary-length type-lists become more common.
- Type-safe variadic functions should eliminate the need for some operator overloading (e.g., ‘%’ in Boost.Format).
- Forwarding any number of function arguments becomes simple.

# Miscellaneous Features

# Static Assertions

- Statically check that a certain condition is true
  - If not, provide a custom error message

```
template<typename RAIter>
void sort(RAIter first, RAIter last) {
  static_assert(is_convertible<
       iterator_traits<RAIter>::iterator_category,
       random_access_iteratr_tag
     >::value,
    “Iterators are not Random Access Iterators”);
}
```

# Explicit Conversion Operations

- User-defined conversion operators have always been implicit conversions:

```
template<typename T> struct shared_ptr {
  operator bool() const;
};
shared_ptr<int> p;
if (p) { ... } // we like this
p + 17; // we don't like this
```

- They can now be explicit:
  - `explicit operator bool() const;`

# Delegating Constructors

- Classes with several constructors often need to duplicate initialization logic:

```
struct Rectangle {
  float left, top, right, bottom;
  float area;
  Rectangle(float l, float t, float r, float b)
    : left(l), top(t), right(r), bottom(b),
      area((right-left) * (bottom-top)) { }
  Rectangle(Point lt, Point rb)
   : left(lt.x), top(lt.y), right(rb.x), bottom(rb.y),
      area((rb.x - lt.x) * (rb.y - lt.y)) { }
};
```

# Delegating Constructors

- Delegating constructors allow one constructor to call another:

```
struct Rectangle {
  float left, top, right, bottom;
  float area;
  Rectangle(float l, float t, float r, float b)
    : left(l), top(t), right(r), bottom(b),
      area((right-left) * (bottom-top)) { }
  Rectangle(Point lt, Point rb)
   : Rectangle(lt.x, lt.y, rb.x, rb.y) { }
};
```

# Inheriting Constructors

- Subclassing inherits all of the members of the base class... except constructors.
- *Inheriting constructors* allows one to explicitly inherit constructors. Example:

```
class EmployeeList : public std::vector<Employee> {
public:
 using std::vector<Employee>::vector;
 double giveRaise(double percent);
 // inherits push_back(), begin(), etc. from vector
};
```

# Defaulted Functions

- C++ provides default definitions for special class members
  - What does `struct X { };` contain?
- To be explicit, one can now explicitly request default implementations:

```
struct X {
  X() = default;
  X(X const &) = default;
  X& operator=(X const&) = default;
};
```

# Template Aliases

- Like a **typedef**, but for templates:

```
template<typename T>
  using ArenaVector
    = vector<T, ArenaAllocator<T>>;
```

- The same syntax works for types:

```
using string = basic_string<char>;
```

# Summary and Recap

- `auto` and `decltype` allow the deduction of types from expressions
- Improved SFINAE for compile-time inspection
- Rvalue references enable move semantics and perfect forwarding
- Variadic templates eliminate preprocessor metaprogramming

# Preparing for C++0x

- If you are using GCC 4.3 or newer
  - Compile with `-Wc++0x-compat`
  - Play with `-std=gnu++0x`
- Libraries should move to C++0x first:
  - C++0x feature detection macros in Boost
  - Expect side-by-side C++98/C++0x implementations
- “Boost.Fusion using C++0x” workshop

# Get Involved!

- Boost should lead the charge for C++0x
- New libraries and improvements to existing libraries using C++0x features
- All you need is a C++0x compiler and a PDF reader

# Questions?

# Lambda Expressions/Closures

- Lambdas allow the construction of function objects with little syntax
- Useful, e.g., for using generic algorithms:

```
transform(input.begin(), input.end(),
          std::back_inserter(results),
           [](int x) { return -x; });

for_each(employees.begin(), employees.end(),
         [](Employee& e) { e.RaiseSalary(0.03); });
```

# Lambda Expressions/Closures

- Lambdas can have state (making them closures)
  - Can either reference or copy data from the stack

```
double min_salary = ...;
double u_limit = 1.1 * min_salary;
std::find_if(employees.begin(), employees.end(),
  [=](const employee& e) {
   return e.salary() >= min_salary
          && e.salary() < u_limit; });
```

Copied into closure

# Lambda Expressions/Closures

- Lambdas can have state (making them closures)
  - Can either reference or copy data from the stack

```
double min_salary = ...;
double u_limit = 1.1 * min_salary;
std::find_if(employees.begin(), employees.end(),
  [&](const employee& e) {
   return e.salary() >= min_salary
         && e.salary() < u_limit; });
```

Referenced from closure

# More on Capture Lists

- The [brackets] in a lambda contains a capture list, containing:
  - Optional *default capture*:
    - **&** for reference capture
    - **=** for copy capture
  - Optional, comma-separated *capture list*:
    - **&var**: capture var by reference
    - **var**: capture var by value
    - **this**: capture the this pointer
  - Example: [&, min_salary, this, &employees]