Practical C++11

What I Learned Adding C++11 Support to ODB

Boris Kolpackov

Code Synthesis

v1.1, May 2013

Practical C++11

”The imagination of nature is greater than the imagination of man”

- Everyday application development
- Hard and fast rules of thumb
- Don’t have all the answers
- Assume basic knowledge of C++11

auto (type deduction from initializer)

```
int& f ();

auto x = f (); // x is int, not int&
```

auto (type deduction from initializer)

```
int& f ();

auto x = f (); // x is int, not int&
```

- *Core type*
- *Const-ness/reference-ness*

auto rule of thumb

```
const auto& x  // x is not modified
auto& x        // x is modified, shared
auto x         // x is modified, private
```

auto examples

```
std::vector<std::shared_ptr<object>> v;

const auto& o (*v.back ());
cout << o << endl;

auto& p (v.back ());
if (p == 0)
  p.reset (new object);

for (auto i (v.begin ()); i != v.end (); ++i)
  ...
```

auto examples

```
std::vector<std::shared_ptr<object>> v;

const auto& o (*v.back ());
cout << o << endl;

auto& p (v.back ());
if (p == 0)
  p.reset (new object);

for (auto i (v.begin ()); i != v.end (); ++i)
  ...
```

auto examples

```
std::vector<std::shared_ptr<object>> v;

const auto& o (*v.back ());
cout << o << endl;

auto& p (v.back ());
if (p == 0)
  p.reset (new object);

for (auto i (v.begin ()); i != v.end (); ++i)
  ...
```

auto intuition

```
const int& f ();

auto& x = f ();
```

auto intuition

```
const int& f ();

auto& x = f ();


template <typename T>
void g (T& x);

g (f ()); // T = const int, x is const int&
```

Perfect forwarding

```
template <typename T>
void f (T&& x);
```

Perfect forwarding

```
template <typename T>
void f (T&& x);

int i;
f (i); // T = int&; x is int&
```

Perfect forwarding

```
template <typename T>
void f (T&& x);

int i;
f (i); // T = int&; x is int&

f (2); // T = int;  x is int&&
```

Perfect forwarding

```
template <typename T>
void f (T&& x);

int i;
f (i); // T = int&; x is int&

f (2); // T = int;  x is int&&
```

Perfect forwarding & auto

```
auto&& x = f ();
```

Perfect forwarding & auto

```
auto&& x = f ();
```

```
template <typename F1, typename F2, typename F3>
void compose (F1 f1, F2 f2, F3 f3)
{
  auto&& r = f1 ();
  f2 ();
  f3 (std::forward<decltype (f1 ())> (r));
}
```

Perfect forwarding and overloading

```
void f (int);
void f (const std::string&);
```

Perfect forwarding and overloading

```
void f (int);
void f (const std::string&);

int i = 1;
std::string s = ”aaa”;

f (i);     // f(int)
f (2);     // f(int)
f (s);     // f(string)
f (”bbb”); // f(string)
```

Perfect forwarding and overloading

```
void f (int);
void f (const std::string&);
```

Perfect forwarding and overloading

```
void f (int);
void f (const std::string&);

template <typename T>
void f (T&&);
```

Perfect forwarding and overloading

```
void f (int);
void f (const std::string&);

template <typename T>
void f (T&&);

f (i);     // f(int)
f (2);     // f(int)
f (s);     // f(T&&)  T = std::string&
f (”bbb”); // f(T&&)  T = const char (&)[4]
```

Perfect forwarding and overloading

```
template <typename T>
class lazy_ptr
{
  lazy_ptr (T*);

  template <typename ID>
  lazy_ptr (const ID&);



};
```

Perfect forwarding and overloading

```
template <typename T>
class lazy_ptr
{
  lazy_ptr (T*);

  template <typename ID>
  lazy_ptr (const ID&);

  template <typename ID>
  lazy_ptr (ID&&);
};
```

Rule of thumb

- T&& is not an rvalue reference
- Perfect forwarding and overloading don't mix

Perfect forwarding and overloading

Any way to make this work?

Perfect forwarding and overloading

Any way to make this work?

```
template <typename T,
          typename disable_forward<
            T,
            int,
            std::string>::type = 0>
void f (T&&);
```

Range-based for loop

```
for ( declaration : expression ) statement
```

Range-based for loop

```
for ( declaration : expression ) statement
```

```
{
  auto&& __range = expression;

  for (auto __i = begin-expression,
            __e = end-expression;
       __i != __e;
       ++__i)
  {
    declaration = *__i;
    statement
  }
}
```

Range-based for loop

```
for ( declaration : expression ) statement
```

```
{
  auto&& __range = expression;

  for (auto __i = begin-expression,
            __e = end-expression;
       __i != __e;
       ++__i)
  {
    declaration = *__i;
    statement
  }
}
```

Range-based for loop

```
for ( declaration : expression ) statement
```

```
{
  auto&& __range = expression;

  for (auto __i = begin-expression,
            __e = end-expression;
       __i != __e;
       ++__i)
  {
    declaration = *__i;
    statement
  }
}
```

Range-based for loop

```
for ( declaration : expression ) statement


{
  auto&& __range = expression;

  for (auto __i = begin-expression,
            __e = end-expression;
       __i != __e;
       ++__i)
  {
    declaration = *__i;
    statement
  }
}
```

Range-based for & rvalue

```
std::vector<int> f ();

for (int& i: f ())
  i = 0;
```

Range-based for & auto

```
std::vector<std::string> v;

for (const auto& s: v)
  cout << s;
```

Range-based for loop

```
for (auto __i = __range.begin (),
          __e = __range.end ();
       __i != __e;
       ++__i)
...
```

Range-based for loop

```
for (auto __i = __range.begin (),
          __e = __range.end ();
       __i != __e;
       ++__i)
...

for (auto __i = begin (__range),
          __e = end (__range);
       __i != __e;
       ++__i)
...
```

Reverse range-based for

```
template <typename T>
struct reverse_range
{
  T&& x_;
  reverse_range (T&& x): x_ (std::forward<T> (x)) {}

  auto begin () const -> decltype (this->x_.rbegin ())
  {
    return x_.rbegin ();
  }

  auto end () const -> decltype (this->x_.rend ())
  {
    return x_.rend ();
  }
};
```

Reverse range-based for

```
template <typename T>
reverse_range<T> reverse_iterate (T&& x)
{
  return reverse_range<T> (std::forward<T> (x));
}
```

Reverse range-based for

```
template <typename T>
reverse_range<T> reverse_iterate (T&& x)
{
  return reverse_range<T> (std::forward<T> (x));
}

for (int& x: reverse_iterate (v))
  ...
```

Efficient argument passing in C++11

```
void f (const std::vector<int>&);
```

Efficient argument passing in C++11

```
void f (const std::vector<int>&);

f ({1, 2, 3, 4});
```

Efficient argument passing in C++11

```
void f (const std::vector<int>& v)
{
  std::vector<int> c (v); // copy
  ...
}

void f (std::vector<int>&& v)
{
  std::vector<int> c (std::move (v)); // move
  ...
}
```

Efficient argument passing in C++11

```
struct email
{
  email (const std::string& f,
         const std::string& l,
         const std::string& a)
    : first_name_ (f),
      last_name_ (l),
      address_ (a)
  {
  }

  ...

  std::string first_name_;
  std::string last_name_;
  std::string address_;
};
```

Combinatorial explosion in C++11

```
email (const string&, const string&, const string&);
email (string&&, const string&, const string&);
email (const string&, string&&, const string&);
email (string&&, string&&, const string&);
email (const string&, const string&, string&&);
email (string&&, const string&, string&&);
email (const string&, string&&, string&&);
email (string&&, string&&, string&&);
```

Pass by value

```
email (std::string f, std::string l, std::string a)
  : first_name_ (std::move (f)),
    last_name_ (std::move (l)),
    address_ (std::move (a))
{
}
```

Pass by value

```
email (std::string f, std::string l, std::string a)
  : first_name_ (std::move (f)),
    last_name_ (std::move (l)),
    address_ (std::move (a))
{
}
```

- Only works if definitely making a copy
- Hardcoding assumptions about implementation into interface
- Only works if type is movable
- Other, more obscure, problems

Perfect forwarding

```
template <typename T1, typename T2, typename T3>
email (T1&& f, T2&& l, T3&& a)
  : first_name_ (std::forward<T1> (f)),
    last_name_ (std::forward<T2> (l)),
    address_ (std::forward<T3> (a))
{
}
```

Perfect forwarding

```
template <typename T1, typename T2, typename T3>
email (T1&& f, T2&& l, T3&& a)
  : first_name_ (std::forward<T1> (f)),
    last_name_ (std::forward<T2> (l)),
    address_ (std::forward<T3> (a))
{
}
```

- Has to be template
- Cannot be used for virtual functions
- Pushes diagnostics into implementation
- Incompatible with overloading
- Loose, ”type-less”, and undocumented interface

Truly universal reference

- Binds to lvalues and rvalues
- Allows us to determine which one at runtime
- No such beast exists in C++11
- Can we create our own?

Pass by universal reference

```
email (uref<std::string> f,
       uref<std::string> l,
       uref<std::string> a)
  : first_name_ (f.move ()),
    last_name_ (l.move ()),
    address_ (a.move ())
{
}
```

Truly universal reference

```
template <typename T>
class uref
{
  ...

  bool lvalue () const;
  bool rvalue () const;

  operator const T& () const;
  const T& get () const;
  T&& rget () const;

  T move () const; // Make copy if lvalue.
};
```

Pass by universal reference

```
email (uref<std::string> f,
       uref<std::string> l,
       uref<std::string> a)
  : first_name_ (f.move ()),
    last_name_ (l.move ()),
    address_ (a.move ())
{


}
```

Pass by universal reference

```
email (uref<std::string> f,
       uref<std::string> l,
       uref<std::string> a)
   : first_name_ (f.move ()),
     last_name_ (l.move ()),
     address_ (a.move ())
{


}
```

- Non-idiomatic
- Inelegant

Pass by universal reference

```
email (uref<std::string> f,
       uref<std::string> l,
       uref<std::string> a)
  : first_name_ (f.move ()),
    last_name_ (l.move ()),
    address_ (a.move ())
{
  if (f.get ().empty ())
    ...
}
```

- Non-idiomatic
- Inelegant

Rule of thumb

Choose only one method

- Pass by `const` reference

Rule of thumb

Choose between two methods

- Pass by value if *conceptually* making a copy
- Pass by `const` reference otherwise

Rule of thumb

1. Does the function *conceptually* make a copy of its argument?
2. If NO, then pass by `const` reference
3. If YES, then pass by value
4. Based on evidence, optimize a select few cases with rvalue overloads

Polymorphic move constructor

```
class fruit
{
  virtual fruit* clone () const = 0;
};

class apple: public fruit
{
  virtual apple* clone ()
  {
    return new apple (new apple (*this));
  }
};
```

Polymorphic move constructor

```
class fruit
{
  virtual fruit* clone () const = 0;
};

class apple: public fruit
{
  virtual apple* clone ()
  {
    return new apple (new apple (*this));
  }
};

fruit_catalog c;
c.add (apple (apple::granny_smith));
c.add (pear (pear::bartlett));
```

Polymorphic move constructor

```
class fruit
{
  virtual fruit* clone () const = 0;
  virtual fruit* move () = 0;
};

class apple: public fruit
{
  virtual apple* move ()
  {
    return new apple (std::move (*this));
  }
};
```

Polymorphic move constructor

```
class fruit
{
  virtual fruit* clone () const & = 0;
  virtual fruit* clone () && = 0;
};

class apple: public fruit
{
  virtual apple* clone () &&
  {
    return new apple (std::move (*this));
  }
};
```

Polymorphic move constructor

```
class fruit
{
  virtual fruit* clone () const & = 0;
  virtual fruit* clone () && = 0;
};

class apple: public fruit
{
  virtual apple* clone () &&
  {
    return new apple (std::move (*this));
  }
};

template <typename T>
void f (T&& f)
{
  unique_ptr<fruit> p (std::forward<T> (f).clone ());
}
```

Polymorphic move and argument passing

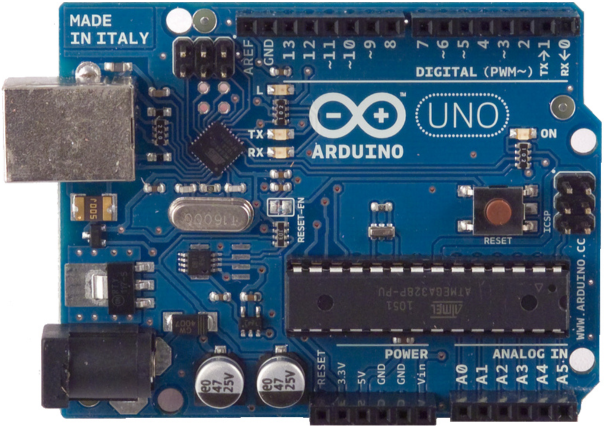
```
class fruit_catalog
{
  void add (const fruit& f)
  {
    unique_ptr<fruit> p (f.clone ());
    ...
  }
};
```

Polymorphic move and argument passing

```
class fruit_catalog
{
  void add (fruit f)
  {
    unique_ptr<fruit> p (std::move (f).clone ());
    ...
  }
};
```

State of C++11 support

- C++98 vs C++11 days
- Is this time different?
- Yes, for application development
- Not for library development


MADE
IN ITALY
AREF
GND
13
12
~11
~10
~9
8
7
~6
~5
4
~3
2
TX→1
RX←0
DIGITAL (PWM~)
L
TX
RX
ARDUINO
UNO
ON
RESET-FN
RESET
ICSP
WWW.ARDUINO.CC
POWER
ANALOG IN
RESET
3.3V
5V
GND
GND
Vin
A0
A1
A2
A3
A4
A5

Arduino Uno

- Microcontroller
- 16Mhz
- 32Kb flash

Arduino and C++11

- Rvalue references
- Lambdas
- Initializer lists
- `auto`
- Range-based for

C++11 and library development

```
#if _MSC_VER >= 1600
#  define ODB_CXX11
#  define ODB_CXX11_NULLPTR
/*
#  define ODB_CXX11_DELETED_FUNCTION
#  define ODB_CXX11_EXPLICIT_CONVERSION_OPERATOR
#  define ODB_CXX11_FUNCTION_TEMPLATE_DEFAULT_ARGUMENT
#  define ODB_CXX11_VARIADIC_TEMPLATES
#  define ODB_CXX11_INITIALIZER_LIST
*/
#endif
```

Simultaneous C++98 and C++11 support

- `pkg-config --std c++11` ?
- C++11 support has to be header-only
- C++11 code has to be inline or template

Resources

- Using C++11 auto and decltype
- Rvalue reference pitfalls
- Perfect forwarding and overload resolution
- C++11 range-based for loop
- Efficient argument passing in C++11, Part 1, 2, and 3
- A Sense of Design (my blog)