Adventures in Perfect Forwarding

Scott Meyers, Ph.D.
Software Development Consultant

smeyers@aristeia.com
http://www.aristeia.com/

Voice: 503/638-6028
Fax: 503/974-1887

Lvalues and Rvalues

Lvalues:

- Generally expressions you can take the address of.
  - E.g., named variables.
- Copy requests *copy* (i.e., don't change source expression).

Rvalues:

- Generally expressions that aren't lvalues.
  - E.g., temporary objects.
- C++98/03 copy requests often become *move* requests.

Examples:

```
std::shared_ptr<Widget> factory();     // return type is rvalue

auto p1 = factory();                   // move construction
                                       // from rvalue source

auto p2 = p1;                          // copy construction
                                       // from lvalue source
```

Rvalue References and Universal References

Rvalue references:

- Like "normal" ("lvalue") references, but bind only to rvalues.
- Syntax: *type*&& (e.g., int&&, std::string&&).

"&&" parameters in function templates:

- Mean "bind to anything" ⇒ *universal references*:

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

  - ➡ T's deduced type T& for lvalue args, T for rvalue args.
  - ➡ Function param type T& for lvalue args, T&& for rvalues.

```
int x;
f(x);                 // lvalue: T is int&
                      // instantiates f(int&)
f(factory());         // rvalue:
                      // T is std::shared_ptr<Widget>
                      // instantiates f(std::shared_ptr<Widget>&&)
```

Practically speaking, not an rvalue reference!

Perfect Forwarding

Given

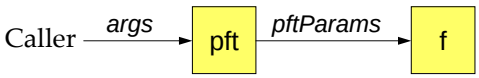
```
returnType f( fParams );
```

a perfect forwarding template for f

```
template< templateParms >
returnType pft( pftParams );
```

is (typically) designed so that

```
pft( args );
```

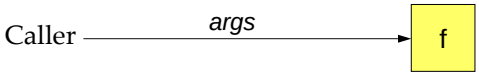


internally invokes

```
f( pftParams );
```

which behaves the same as

```
f( args );
```

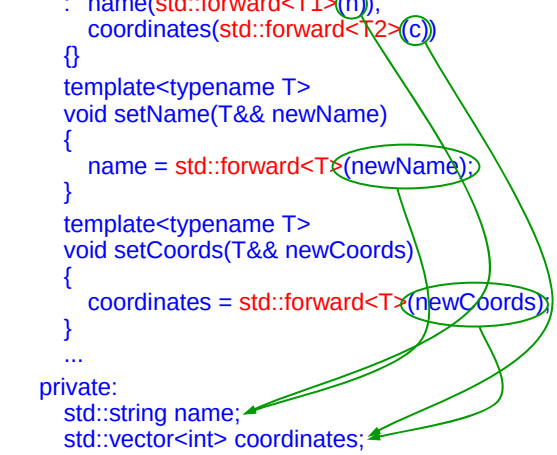


would have.

Use Cases for Perfect Forwarding

Constructors and setters:

```
class Widget {
public:
  template<typename T1, typename T2>                  // PF ctor
  Widget(T1&& n, T2&& c)
  : name(std::forward<T1>(n)),
    coordinates(std::forward<T2>(c))
  {}
  template<typename T>                                // PF setter
  void setName(T&& newName)
  {
    name = std::forward<T>(newName);
  }
  template<typename T>                                // PF setter
  void setCoords(T&& newCoords)
  {
    coordinates = std::forward<T>(newCoords);
  }
  ...
private:
  std::string name;
  std::vector<int> coordinates;
};
```



Use Cases for Perfect Forwarding

std::make_shared:

```
std::string s("PF Example");
auto pw = std::make_shared<Widget>(s,                          // lvalue
                                   std::vector<int>({1, 2, 3}));  // rvalue

template<typename T, typename... ParamTs>
shared_ptr<T> make_shared(ParamTs&&... params)
{                                                              // very
  void *memForSP =                                             // simplified!
    ::operator new(sizeof(T)+sizeof(RCBlock));
  auto sp =
    new (memForSP) shared_ptr<T>(make_shared_tag,
                                 std::forward<ParamTs>(params)...);
  return *sp;
}
```

g++ does it indirectly:

- make_shared→allocate_shared→shared_ptr ctor→...→T ctor

Perfect Forwarding is Imperfect

Several kinds of arguments can't be perfect-forwarded.

- 0 as null pointer constant
- Braced initializer lists
- Integral const static class members lacking a definition
- Template names (e.g., std::endl)
- Non-const lvalue bitfields

Further Information has details.

- Though imperfect, perfect forwarding is still quite good.

"Specializing" Forwarding Templates

Forwarding to f is easy,

```
template<typename T>
void fwd(T&& param)
{
  ...
  f(std::forward<T>(param));
  ...
}
```

but what if we want to treat some types differently?

- I.e., want to "specialize" fwd for pointers.

"Specializing" Forwarding Templates

Function templates can't be specialized, only overloaded.

- But the form of forwarding templates is constrained.
  - ➡ The only parameter type that works is T&&!
    - ◆ "Works" ≡ handles all combos of const/non-const + lvalue/rvalue

```
template<typename T>
void fwd(T&& param);          // fine, handles all combos

template<typename T>
void fwd(T*&& param);         // handles rvalues only
                              // (param type is rref-to-pointer)
```


"Specializing" Forwarding Templates

```
template<typename T>          // #1: "normal" forwarding
void fwd(T&& param)           // template
{
  f(std::forward<T>(param));
}

template<typename T>          // #2: attempted "specialized"
void fwd(T*&& param)          // forwarding template
{
  f(std::forward<T*>(param));
}

std::string* ps;
const std::string *pcs;

fwd(ps);                      // calls #1
fwd(pcs);                     // calls #1
fwd((std::string*&&)ps);      // calls #2
```

Using std::enable_if

One solution: overload via std::enable_if:

```
template<typename T>
typename std::enable_if<
   !std::is_pointer<                                  // template for
      typename std::remove_reference<T>::type         // non-pointer types
   >::value
>::type
fwd(T&& param)
{
   f(std::forward<T>(param));
}

template<typename T>
typename std::enable_if<
   std::is_pointer<                                   // template for
      typename std::remove_reference<T>::type         // pointer types
   >::value
>::type
fwd(T&& param)
{
   f(std::forward<T>(param));
}
```


Using std::enable_if

Drawbacks:

- No ordering among templates with satisfied enable_ifs.
  - ➧More/less "specialized" requires manual implementation.
- New "specialization" ⇒ existing ones may have to be modified.

Consider a new "specialization" for forwarding const char*s.

- Next slide.

Scott Meyers, Software Development Consultant
http://www.aristeia.com/
© 2012 Scott Meyers, all rights reserved.

Using std::enable_if

```
template<typename T>                                    // general version
typename std::enable_if<
   !std::is_pointer<typename std::remove_reference<T>::type>::value
>::type
fwd(T&& param)
{ f(std::forward<T>(param)); }

template<typename T>                                    // general T* version
typename std::enable_if<
   std::is_pointer<typename std::remove_reference<T>::type>::value
   && !std::is_same<typename std::remove_reference<T>::type,
                    const char*>::value
>::type
fwd(T&& param)
{ f(std::forward<T>(param)); }

template<typename T>                                    // const char* version
typename std::enable_if<
   std::is_same< typename std::remove_reference<T>::type,
                 const char*>::value
>::type
fwd(T&& param)
{ f(std::forward<T>(param)); }
```


Dispatching Through a Class Template

Enables use of class template partial specialization:

```
template<typename T>                                  // General template
class fwd_impl {
public:
   template<typename U>                               // Standard forwarding function
   static void call(U&& param)
   {
      f(std::forward<U>(param));
   }
};

template<typename T>                                  // Partial specialization
class fwd_impl<T*> {
public:
   template<typename U>
   static void call(U&& param)
   {
      f(std::forward<U>(param));
   }
};

template<typename T>                                  // Standard declaration
void fwd(T&& param)
{
  return fwd_impl<typename std::remove_reference<T>::type>::call(std::forward<T>(param));
}
```

Turn e.g., T*& into T*

Standard forwarding call

Dispatching Through a Class Template

Adding a new specialization now easy:

```
template<>
class fwd_impl<const char*> {
public:
  template<typename U>
  static void call(U&& param)
  {
    f(std::forward<U>(param));
  }
};
```

- Existing specializations not affected.
- Compiler automatically chooses most specialized version.

Meanwhile, Back on Earth...

Specializing for pointers unrealistic.

- Rarely useful to distinguish pointer lvalues and rvalues.

Specializing for UDTS very realistic.

- Often useful to distinguish lvalues and rvalues
- std::shared_ptr is such a UDT.
  - ➡ Moving cheaper than copying.

New goal: "specialize" fwd for std::shared_ptrs.

"Specializing" for std::shared_ptrs

Easy:

```
template<typename T>
class fwd_impl<std::shared_ptr<T>> {
public:
  template<typename U>
  static void call(U&& param)
  {
    f(std::forward<U>(param));
  }
};
```

But doesn't catch const std::shared_ptrs:

```
std::shared_ptr<Widget> spw;
...
fwd(spw);                          // calls above specialization
const std::shared_ptr<Widget> cspw;
...
fwd(cspw);                         // calls general fwd_impl
                                   // template
```


"Specializing" for std::shared_ptrs

To catch both, you can add another partial specialization:

```
template<typename T>
class fwd_impl<const std::shared_ptr<T>> {
public:
  template<typename U>
  static void call(U&& param)
  {
    f(std::forward<U>(param));
  }
};
```

But this won't handle volatile-qualified std::shared_ptrs, sigh.

"Specializing" for std::shared_ptrs

If std::shared_ptr is all you care about, strip cv qualifiers inside fwd:

```
template<typename T>
void fwd(T&& param)
{
  return fwd_impl<
    typename std::remove_cv<
      typename std::remove_reference<T>::type
      >::type
    >::call(std::forward<T>(param));
}

                std::shared_ptr<Widget> spw;
const           std::shared_ptr<Widget> cspw;
volatile        std::shared_ptr<Widget> vspw;
const volatile  std::shared_ptr<Widget> cvspw;

fwd(spw);                          // all invoke fwd_impl for
fwd(cspw);                         // std::shared_ptr
fwd(vspw);
fwd(cvspw);
```


Gratuitous Animal Photo: Atlas Moth

Source: http://lafinjack.net/images/random/

Forwarding and the pImpl Idiom

A class with a forwarding constructor and a setter:

```
class Widget {
public:
  template<typename T>
  Widget(T&& param)                          // forwarding ctor
  : name(std::forward<T>(param))
  {}

   template<typename T>
   void setName(T&& newName)                 // forwarding setter
   {
     name = std::forward<T>(newName);
  }
  ...
private:
  std::string name;
};
```


Forwarding and the pImpl Idiom

Pimplization is straightforward:

```
// File Widget.h
class WidgetImpl;                            // declaration only

class Widget {
public:
  template<typename T>
  Widget(T&& param);                         // declaration only

   template<typename T>
   void setName(T&& newName);                // declaration only
  ...
private:
  std::shared_ptr<WidgetImpl> p;             // pImpl.  (std::unique_ptr
};                                           // might be preferable.)
```

Widget clients compile with this.

Forwarding and the pImpl Idiom

Simple implementation equally straightforward:

```
// File WidgetImpl.cpp
struct WidgetImpl {                              // simplest thing that
  std::string name;                              // could possibly work
};

template<typename T>
Widget::Widget(T&& param)                        // forwarding ctor
: p(new WidgetImpl)
{
  p->name = std::forward<T>(param);
}

template<typename T>
void Widget::setName(T&& newName)                // forwarding setter
{
  p->name = std::forward<T>(newName);
}
```

Widget clients link against a compiled version of this.

Forwarding and the pImpl Idiom

Prospective client code:

```
#include "Widget.h"

int main()
{
  Widget w("This is a test");                          // lvalue
                                                       // const char[15]

  w.setName(std::string("This is another test"));      // rvalue std::string

  std::string s("Still testing...");
  w.setName(s);                                        // lvalue std::string
}
```

Compiles without fuss.

Forwarding and the pImpl Idiom

Linking a different story:

- g++ 4.7:

```
...:pimpl.cpp:(.text.startup+0x26): undefined reference to `Widget::Widget<char const (&) [15]>(char const (&) [15])'
...:pimpl.cpp:(.text.startup+0x4c): undefined reference to `void Widget::setName<string>(string &&)'
...:pimpl.cpp:(.text.startup+0x79): undefined reference to `void Widget::setName<string &>(string &)'
```

- VC11:

```
pimpl.obj : error LNK2019: unresolved external symbol "public: __thiscall Widget::Widget<char const (&)[15]>(char const
(&)[15])" (??$?0AAY0P@$$CBD@Widget@@QAE@AAY0P@$$CBD@Z) referenced in function _main

pimpl.obj : error LNK2019: unresolved external symbol "public: void __thiscall Widget::setName<class
std::basic_string<char,struct std::char_traits<char>,class std::allocator<char> > >(class std::basic_string<char,struct
std::char_traits<char>,class std::allocator<char> > &&)"
(??$setName@V?$basic_string@DU?$char_traits@D@std@@V?$allocator@D@2@@std@@@Widget@@QAEX$$QAV?$
basic_string@DU?$char_traits@D@std@@V?$allocator@D@2@@std@@@Z) referenced in function _main

pimpl.obj : error LNK2019: unresolved external symbol "public: void __thiscall Widget::setName<class
std::basic_string<char,struct std::char_traits<char>,class std::allocator<char> > &>(class std::basic_string<char,struct
std::char_traits<char>,class std::allocator<char> > &)"
(??$setName@AAV?$basic_string@DU?$char_traits@D@std@@V?$allocator@D@2@@std@@@Widget@@QAEXAAV?$
basic_string@DU?$char_traits@D@std@@V?$allocator@D@2@@std@@@Z) referenced in function _main
```


Forwarding and the pImpl Idiom

The usual template instantiation problem:

- Compilers need template source for instantation.
- VC11 and g++ 4.7 (on Windows) want it during compilation.

Pimpl precludes that.

Possible solutions:

- Abandon pImpl, i.e., admit defeat (!).
- Use different build chain.
  - ➧Some use link-time instantation.
- Perform explicit instantation.

Explicit Instantiation

The templates again:

```
template<typename T>
Widget::Widget(T&& param) ...                    // forwarding ctor

template<typename T>
void Widget::setName(T&& newName) ...            // forwarding setter
```

The uses again:

```
Widget w("This is a test");                      // lvalue
                                                 // const char[15]

w.setName(std::string("This is another test")); // rvalue std::string

w.setName(s);                                    // lvalue std::string
```

Instantiations we need:

- Widget ctor with T = const char (&)[15]
- setName with T = std::string and also T = std::string&

Explicit Instantiation

This is *exactly* what g++ reported:

```
...:pimpl.cpp:(.text.startup+0x26): undefined reference to `Widget::Widget<char const (&) [15]>(char const (&) [15])'
...:pimpl.cpp:(.text.startup+0x4c): undefined reference to `void Widget::setName<string>(string &&)'
...:pimpl.cpp:(.text.startup+0x79): undefined reference to `void Widget::setName<string &>(string &)'
```

Explicit Instantiation

Piece of cake:

```
// In WidgetImpl.cpp (or a file that can see it)
template Widget::Widget<char const(&)[15]>(char const(&)[15]);
template void Widget::setName<std::string>(std::string&&);
template void Widget::setName<std::string&>(std::string&);
```

- For rvalues, deduced and parameter types not the same.
  - ➡ E.g., T = std::string, but parameter type is std::string&&.
- g++'s linker tells you what you need.

```
...:pimpl.cpp:(.text.startup+0x26): undefined reference to `Widget::Widget<char const (&) [15]>(char const (&) [15])'
...:pimpl.cpp:(.text.startup+0x4c): undefined reference to `void Widget::setName<string>(string &&)'
...:pimpl.cpp:(.text.startup+0x79): undefined reference to `void Widget::setName<string &>(string &)'
```

With code above, g++ 4.7 compiles/links/runs.

Explicit Instantiation

VC11 compiler rejects these explicit instantiation requests:

```
widgetimpl.cpp(24) : error C2143: syntax error : missing ';' before '<'
```

MS has confirmed two (!) compiler bugs. Plus a workaround :-)

- Explicitly force implicit instantiation:

```
// In WidgetImpl.cpp (or a file that can see it)
namespace {
  void forceInstantiations()        // never called
  {
    Widget w("This is a test");     // const char(&)[15]
    w.setName(std::string());       // rvalue std::string

    std::string s;
    w.setName(s);                   // lvalue std::string
  }
}
```

This works with g++, too (⇒ probably portable).

Explicit Instantiation and Arrays

Unpleasant and impractical:

```
...
template Widget::Widget<char const(&)[15]>(char const(&)[15]);
template Widget::Widget<char const(&)[16]>(char const(&)[16]);
template Widget::Widget<char const(&)[17]>(char const(&)[17]);
...
```

Would be nice to "partially specialize" the forwarding templates.

- T[n] arguments could be cast to T* before forwarding.
- Exercise for the attendee :-)

Summary

- Perfect forwarding fails for some kinds of arguments.
- To "specialize" forwarding templates, use helper class templates or std::enable_if.
- Templates + pImpl ⇒ explicitly force instantiations.

Further Information

Topics discussed here:

- "Perfect Forwarding Failure Cases," comp.std.c++ discussion initiated 16 January 2010.
- "Specializing Perfect Forwarding Templates?," comp.lang.c++ discussion initiated 15 March 2011.
- "Onward, Forward!," Dave Abrahams, *C++Next*, 7 Dec. 2009.
- "Universal References in C++11," *C++ and Beyond 2012*, August 2012.

Further Information

Topics related to topics discussed here:

- "Initial Thoughts on *Effective C++11*," *C++ and Beyond 2012*, August 2012.
- "unordered_map::emplace in C++0x," comp.lang.c++ discussion initiated 28 April 2011.
  - ➡ Forwarding multiple parameters as one object (forward_as_tuple):

Further Information

Nuances of std::forward specification/implementation.

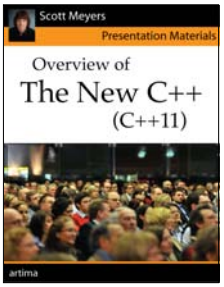
- Focus on subtle "safely forward an X as a Y" cases.
  - ➧ E.g., in converting move constructors/operator=s.
- "forward," Howard E. Hinnant, WG21 Document N2951, 27 September 2009.
  - ➧ Abrahams' "Onward, Forward!" contains relevant comments.
- "Proposed wording for US 90," Howard Hinnant and Daniel Krügler, WG21 Document N3143, 15 October 2010.

Licensing Information

Scott Meyers licenses materials for this and other training courses for commercial or personal use. Details:

- Commercial use: http://aristeia.com/Licensing/licensing.html
- Personal use: http://aristeia.com/Licensing/personalUse.html

Courses currently available for personal use include:



About Scott Meyers

Scott is a trainer and consultant on the design and implementation of C++ software systems. His web site,

http://aristeia.com/

provides information on:

- Training and consulting services
- Books, articles, other publications
- Upcoming presentations
- Professional activities blog

Scott Meyers, Software Development Consultant
http://www.aristeia.com/
© 2012 Scott Meyers, all rights reserved.