The C++ STANDARD LIBRARY EXTENSIONS

A Tutorial and Reference

PETE BECKER

The C++ Standard Library Extensions

```
  refwrap rw1(j);
  rw1 = rw0;      // okay: refwrap can be copied
  }
```

Example 8.1 Copying References
(funobjref/refcopy.cpp)

An object of type `reference_wrapper<Ty>` can be implicitly converted to a reference to `Ty` and has a member function, named `get`, that returns a reference to `Ty`. Finally, if the type `Ty` is a callable type, you can use the object's function call operator to call the object it refers to.

```
template<class Ty>
  class reference_wrapper
    : public unary_function<T1, Ret>       // see Section 8.2
    : public binary_function<T1, T2, Ret> // see Section 8.2
  {
public:
  typedef Ty type;
  typedef T0 result_type;                  // see Section 8.2
  explicit reference_wrapper(Ty&);
  Ty& get() const;
  operator Ty&() const;
  template <class T1, class T2, ..., class TN>
  typename result_of<T(T1, T2, ..., TN)>::type
    operator()(T1&, T2&, ..., TN&) const;  // see Section 8.3
  };
```

8.1 Creation

You can create a `reference_wrapper` object directly with the template's constructor, and you can create a `reference_wrapper` object indirectly by calling the functions `ref` and `cref`.

```
explicit reference_wrapper::reference_wrapper(Ty& obj);
```

The constructor constructs an object that refers to `obj`.

```
#include <functional>
#include <iostream>
using std::tr1::reference_wrapper;
```

```
using std::cout;

int main()
  { // demonstrate basic use of reference_wrapper
  int Stuhldreher = 3;
  reference_wrapper<int> rw(Stuhldreher);
  cout << rw << '\n';            // displays value of Stuhldreher
  Stuhldreher = 4;
  cout << rw << '\n';            // displays new value of Stuhldreher
  rw.get() = 5;                  // changes value of Stuhldreher
  cout << Stuhldreher << '\n';   // displays new value
  return 0;
  }
```

Example 8.2 Construction (`funobjref/construct.cpp`)

```
template <class Ty>
  reference_wrapper<const Ty> cref(
    const Ty& obj);
template <class Ty>
  reference_wrapper<const Ty> cref(
    reference_wrapper<Ty> rw);
template <class Ty>
  reference_wrapper<Ty> ref(
    Ty& obj);
template <class Ty>
  reference_wrapper<Ty> ref(
    reference_wrapper<Ty> rw);
```

The first function template returns an object of type `reference_wrapper<const Ty>` that refers to `obj`. The second function template returns an object of type `reference_wrapper<const Ty>` that refers to `rw.get()`. The third function template returns an object of type `reference_wrapper<Ty>` that refers to `obj`. The fourth function template returns an object of type `reference_wrapper<Ty>` that refers to `rw.get()`.

```
#include <functional>
#include <iostream>
using std::tr1::reference_wrapper;
using std::tr1::ref; using std::tr1::cref;
using std::cout;
```

```
void show(int& i)
  { // show value referred to by reference to int
  cout << "int&: " << i << '\n';
  }

void show(const int& i)
  { // show value referred to by reference to const int
  cout << "const int&: " << i << '\n';
  }

int main()
  { // demonstrate use of ref and cref
  int Miller = 3;
  show(ref(Miller));      // calls show(int&);
  reference_wrapper<int> rw0(Miller);
  show(ref(rw0));         // calls show(int&);
  show(cref(Miller));     // calls show(const int&);
  reference_wrapper<const int> rw1(Miller);
  show(cref(rw1));        // calls show(const int&);
  return 0;
  }
```

Example 8.3 Function Templates `ref` and `cref` (`funobjref/refcref.cpp`)

One important difference between using the constructor for `reference_wrapper` directly and using either of the functions `ref` and `cref` is that these functions return a `reference_wrapper<Ty>` object, where `Ty` is the type of their argument. The constructor, on the other hand, can be passed an object of a type that is convertible to the type `Ty` that the `reference_wrapper` object holds. Thus, although `reference_wrapper` itself directly supports polymorphic types, the functions `ref` and `cref` do not; it takes a bit of trickery to get these functions to return a `reference_wrapper` that refers to a base of the type of their argument. We'll look at how to do that in the exercises.

```
#include <functional>
#include <iostream>
using std::tr1::reference_wrapper;
using std::cout;

struct base
  { // base class
  virtual void show() const
```

```
    { // show name of base class
    cout << "base\n";
    }
  };

struct derived0 : base
  { // one derived class
  void show() const
    { // show name of derived class
    cout << "derived0\n";
    }
  };

struct derived1 : base
  { // another derived class
  void show() const
    { // show name of derived class
    cout << "derived1\n";
    }
  };

int main()
  { // demonstrate reference_wrapper's support for polymorphism
  derived0 Crowley;
  derived1 Layden;
  reference_wrapper<base> rw0(Crowley);
  rw0.get().show();          // calls derived0::show
  reference_wrapper<base> rw1(Layden);
  rw1.get().show();          // calls derived1::show
  return 0;
  }
```

Example 8.4 Polymorphic Types
(`funobjref/polymorph.cpp`)

8.2 Nested Types

```
typedef Ty reference_wrapper::type;
```

The nested type `reference_wrapper<Ty>::type` is a synonym for the template argument `Ty`.

```
typedef T0 reference_wrapper::result_type;
```

The template specialization `reference_wrapper<Ty>` is derived from `std::unary_function<T1, Ret>`—hence defining the nested type `result_type` as a synonym for `Ret` and the nested type `argument_type` as a synonym for `T1`—only if the type `Ty` is one of the following:

- A function type or pointer to function type taking one argument of type `T1` and returning `Ret`
- A pointer to member function type with cv-qualifier *cv* that takes no arguments; the type `T1` is *cv* `Ty*`, and `Ret` is the return type of the pointer to member function
- A type that is derived from `std::unary_function<T1, Ret>`

The template specialization `reference_wrapper<Ty>` is derived from `std::binary_function<T1, T2, Ret>`—hence defining the nested type `result_type` as a synonym for `Ret`, the nested type `first_argument_type` as a synonym for `T1`, and the nested type `second_argument_type` as a synonym for `T2`—only if the type `Ty` is one of the following:

- A function type or pointer to function type taking two arguments of types `T1` and `T2` and returning `Ret`
- A pointer to member function type with cv-qualifier *cv* that takes one argument of type `T2`; the type `T1` is *cv* `Ty*`, and `Ret` is the return type of the pointer to member function
- A type that is derived from `std::binary_function<T1, T2, Ret>`

If it is not derived from `unary_function` or `binary_function`, the template specialization defines the nested type `result_type` according to the rules for a weak result type (see Section 6.2).

8.3 Invocation

```
typename result_of<Ty(T1, T2, ..., TN)>::type
  operator()(T1&, T2&, ..., TN&) const;
```

If the template argument `Ty` is not a callable type, this operator is not present. For a callable type (see Section 6.1) `Ty` and an object `rw` of type `reference_wrapper<Ty>`, the expression `rw(a1, a2, ..., aN)` is equivalent to *INVOKE* `(rw.get(), a1, a2, ..., aN)` (see Section 6.2).

When a `reference_wrapper` object holds a callable object, you can call the `reference_wrapper` object, which will, in turn, call its target object.

```
#include <functional>
#include <iostream>
using std::tr1::reference_wrapper; using std::tr1::cref;
using std::cout;

void hello()
  { // simple function
  cout << "Hello, world\n";
  }

void goodbye()
  { // another simple function
  cout << "Goodbye, cruel world\n";
  }

int main()
  { // demonstrate invocation of reference_wrapper object
  typedef void (*const fun)();
  reference_wrapper<fun> rw(&hello);
  rw();          // calls hello
  rw = cref(&goodbye);
  rw();          // calls goodbye
  return 0;
  }
```

Example 8.5 Invoking
(`funobjref/invoke.cpp`)

If you look at the declaration of `reference_wrapper`'s function call operator, you'll see that all the arguments are passed by reference. This lets you pass modifiable objects to the target object.

```
#include <functional>
#include <iostream>
using std::tr1::reference_wrapper;
```