Boost.Move

Ion Gaztanaga

Table of Contents

Important

To be able to use containers of movable-only values you will need to use containers supporting move semantics, like **Boost.Container** containers

Note

Tested compilers: MSVC-7.1, 8.0, 9.0, GCC 4.3-MinGW in C++03 and C++0x modes, Intel 10.1

What is Boost.Move?

Rvalue references are a major C++0x feature, enabling move semantics for C++ values. However, we don't need C++0x compilers to take advantage of move semanatics. **Boost.Move** emulates C++0x move semantics in C++03 compilers and allows writing portable code that works optimally in C++03 and C++0x compilers.

Introduction

> **Note**
>
> The first 3 chapters are the adapted from the article *A Brief Introduction to Rvalue References* by Howard E. Hinnant, Bjarne Stroustrup, and Bronek Kozicki

Copying can be expensive. For example, for vectors `v2=v1` typically involves a function call, a memory allocation, and a loop. This is of course acceptable where we actually need two copies of a vector, but in many cases, we don't: We often copy a `vector` from one place to another, just to proceed to overwrite the old copy. Consider:

```
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)
}
```

But, we didn't want to have any copies of a or b, we just wanted to swap them. Let's try again:

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

This `move()` gives its target the value of its argument, but is not obliged to preserve the value of its source. So, for a `vector`, `move()` could reasonably be expected to leave its argument as a zero-capacity vector to avoid having to copy all the elements. In other words, **move is a potentially destructive copy**.

In this particular case, we could have optimized swap by a specialization. However, we can't specialize every function that copies a large object just before it deletes or overwrites it. That would be unmanageable.

In C++0x, move semantics are implemented with the introduction of rvalue references. They allow us to implement `move()` without verbosity or runtime overhead. **Boost.Move** is a library that offers tools to implement those move semantics not only in compilers with `rvalue references` but also in compilers conforming to C++03.

Implementing copyable and movable classes

Copyable and movable classes in C++0x

Consider a simple handle class that owns a resource and also provides copy semantics (copy constructor and assignment). For example a `clone_ptr` might own a pointer, and call `clone()` on it for copying purposes:

```
template <class T>
class clone_ptr
{
   private:
   T* ptr;

   public:
   // construction
   explicit clone_ptr(T* p = 0) : ptr(p) {}

   // destruction
   ~clone_ptr() { delete ptr; }

   // copy semantics
   clone_ptr(const clone_ptr& p)
      : ptr(p.ptr ? p.ptr->clone() : 0) {}

   clone_ptr& operator=(const clone_ptr& p)
   {
      if (this != &p)
      {
         T *p = p.ptr ? p.ptr->clone() : 0;
         delete ptr;
         ptr = p;
      }
      return *this;
   }

   // move semantics
   clone_ptr(clone_ptr&& p)
      : ptr(p.ptr) { p.ptr = 0; }

   clone_ptr& operator=(clone_ptr&& p)
   {
      std::swap(ptr, p.ptr);
      delete p.ptr;
      p.ptr = 0;
      return *this;
   }

   // Other operations...
};
```

`clone_ptr` has expected copy constructor and assignment semantics, duplicating resources when copying. Note that copy constructing or assigning a `clone_ptr` is a relatively expensive operation:

```
clone_ptr<Base> p1(new Derived());
// ...
clone_ptr<Base> p2 = p1;  // p2 and p1 each own their own pointer
```

clone_ptr is code that you might find in today's books on C++, except for the part marked as move semantics. That part is implemented in terms of C++0x rvalue references. You can find some good introduction and tutorials on rvalue references in these papers:

- *A Brief Introduction to Rvalue References*
- *Rvalue References: C++0x Features in VC10, Part 2*

When the source of the copy is known to be an rvalue (e.g.: a temporary object), one can avoid the potentially expensive clone() operation by pilfering source's pointer (no one will notice!). The move constructor above does exactly that, leaving the rvalue in a default constructed state. The move assignment operator simply does the same freeing old resources.

Now when code tries to copy an rvalue clone_ptr, or if that code explicitly gives permission to consider the source of the copy an rvalue (using boost::move), the operation will execute much faster.

```
clone_ptr<Base> p1(new Derived());
// ...
clone_ptr<Base> p2 = boost::move(p1);  // p2 now owns the pointer instead of p1
p2 = clone_ptr<Base>(new Derived());   // temporary is moved to p2
}
```

Copyable and movable classes in portable syntax for both C++03 and C++0x compilers

Many aspects of move semantics can be emulated for compilers not supporting rvalue references and **Boost.Move** offers tools for that purpose. With **Boost.Move** we can write clone_ptr so that it will work both in compilers with rvalue references and those who conform to C++03. You just need to follow these simple steps:

- Put the following macro in the **private** section: BOOST_COPYABLE_AND_MOVABLE(classname)
- Left copy constructor as is.
- Write a copy assignment taking the parameter as BOOST_COPY_ASSIGN_REF(classname)
- Write a move constructor and a move assignment taking the parameter as BOOST_RV_REF(classname)

Let's see how are applied to clone_ptr:

```
template <class T>
class clone_ptr
{
   private:
   // Mark this class copyable and movable
   BOOST_COPYABLE_AND_MOVABLE(clone_ptr)
   T* ptr;

   public:
   // Construction
   explicit clone_ptr(T* p = 0) : ptr(p) {}

   // Destruction
   ~clone_ptr() { delete ptr; }

   clone_ptr(const clone_ptr& p) // Copy constructor (as usual)
      : ptr(p.ptr ? p.ptr->clone() : 0) {}

   clone_ptr& operator=(BOOST_COPY_ASSIGN_REF(clone_ptr) p) // Copy assignment
   {
      if (this != &p){
         T *tmp_p = p.ptr ? p.ptr->clone() : 0;
         delete ptr;
         ptr = tmp_p;
      }
      return *this;
   }

   //Move semantics...
   clone_ptr(BOOST_RV_REF(clone_ptr) p)            //Move constructor
      : ptr(p.ptr) { p.ptr = 0; }

   clone_ptr& operator=(BOOST_RV_REF(clone_ptr) p) //Move assignment
   {
      if (this != &p){
         delete ptr;
         ptr = p.ptr;
         p.ptr = 0;
      }
      return *this;
   }
};
```

Question: What about types that don't own resources? (E.g. `std::complex`?)

No work needs to be done in that case. The copy constructor is already optimal.

Composition or inheritance

For classes made up of other classes (via either composition or inheritance), the move constructor and move assignment can be easily coded using the `boost::move` function:

```
class Base
{
   BOOST_COPYABLE_AND_MOVABLE(Base)

   public:
   Base(){}

   Base(const Base &x) {/**/}             // Copy ctor

   Base(BOOST_RV_REF(Base) x) {/**/}      // Move ctor

   Base& operator=(BOOST_RV_REF(Base) x)
   {/**/ return *this;}                   // Move assign

   Base& operator=(BOOST_COPY_ASSIGN_REF(Base) x)
   {/**/ return *this;}                   // Copy assign

   virtual Base *clone() const
   {  return new Base(*this);  }

   virtual ~Base(){}
};

class Member
{
   BOOST_COPYABLE_AND_MOVABLE(Member)

   public:
   Member(){}

   // Compiler-generated copy constructor...

   Member(BOOST_RV_REF(Member))  {/**/}      // Move ctor

   Member &operator=(BOOST_RV_REF(Member))   // Move assign
   {/**/ return *this;  }

   Member &operator=(BOOST_COPY_ASSIGN_REF(Member))   // Copy assign
   {/**/ return *this;  }
};

class Derived : public Base
{
   BOOST_COPYABLE_AND_MOVABLE(Derived)
   Member mem_;

   public:
   Derived(){}

   // Compiler-generated copy constructor...

   Derived(BOOST_RV_REF(Derived) x)             // Move ctor
      : Base(boost::move(static_cast<Base&>(x))),
        mem_(boost::move(x.mem_)) { }

   Derived& operator=(BOOST_RV_REF(Derived) x)  // Move assign
   {
```

```
      Base::operator=(boost::move(static_cast<Base&>(x)));
      mem_  = boost::move(x.mem_);
      return *this;
   }

   Derived& operator=(BOOST_COPY_ASSIGN_REF(Derived) x)  // Copy assign
   {
      Base::operator=(static_cast<const Base&>(x));
      mem_  = x.mem_;
      return *this;
   }
   // ...
};
```

> **Important**
>
> Due to limitations in the emulation code, a cast to `Base &` is needed before moving the base part in the move constructor and call Base's move constructor instead of the copy constructor.

Each subobject will now be treated individually, calling move to bind to the subobject's move constructors and move assignment operators. `Member` has move operations coded (just like our earlier `clone_ptr` example) which will completely avoid the tremendously more expensive copy operations:

```
Derived d;
Derived d2(boost::move(d));
d2 = boost::move(d);
```

Note above that the argument x is treated as a lvalue reference. That's why it is necessary to say `move(x)` instead of just x when passing down to the base class. This is a key safety feature of move semantics designed to prevent accidently moving twice from some named variable. All moves from lvalues occur explicitly.

Movable but Non-Copyable Types

Some types are not amenable to copy semantics but can still be made movable. For example:

- `unique_ptr` (non-shared, non-copyable ownership)
- A type representing a thread of execution
- A type representing a file descriptor

By making such types movable (though still non-copyable) their utility is tremendously increased. Movable but non-copyable types can be returned by value from factory functions:

```
file_descriptor create_file(/* ... */);
//...
file_descriptor data_file;
//...
data_file = create_file(/* ... */);  // No copies!
```

In the above example, the underlying file handle is passed from object to object, as long as the source `file_descriptor` is an rvalue. At all times, there is still only one underlying file handle, and only one `file_descriptor` owns it at a time.

To write a movable but not copyable type in portable syntax, you need to follow these simple steps:

- Put the following macro in the **private** section: `BOOST_MOVABLE_BUT_NOT_COPYABLE(classname)`
- Write a move constructor and a move assignment taking the parameter as `BOOST_RV_REF(classname)`

Here's the definition of `file descriptor` using portable syntax:

```
#include <boost/move/utility.hpp>
#include <stdexcept>

class file_descriptor
{
   int os_descr_;

   private:
   BOOST_MOVABLE_BUT_NOT_COPYABLE(file_descriptor)

   public:
   explicit file_descriptor(const char *filename = 0)          //Constructor
      : os_descr_(filename ? operating_system_open_file(filename) : 0)
   {  if(!os_descr_) throw std::runtime_error("file not found");  }

   ~file_descriptor()                                          //Destructor
   {  if(!os_descr_)  operating_system_close_file(os_descr_);  }

   file_descriptor(BOOST_RV_REF(file_descriptor) x)            // Move ctor
      :  os_descr_(x.os_descr_)
   {  x.os_descr_ = 0;  }

   file_descriptor& operator=(BOOST_RV_REF(file_descriptor) x) // Move assign
   {
      if(!os_descr_) operating_system_close_file(os_descr_);
      os_descr_   = x.os_descr_;
      x.os_descr_ = 0;
      return *this;
   }

   bool empty() const   {  return os_descr_ == 0;  }
};
```

Containers and move semantics

Movable but non-copyable types can be safely inserted into containers and movable and copyable types are more efficiently handled if those containers internally use move semantics instead of copy semantics. If the container needs to "change the location" of an element internally (e.g. vector reallocation) it will move the element instead of copying it. **Boost.Container** containers are move-aware so you can write the following:

```
#include <boost/container/vector.hpp>
#include <cassert>

//Remember: 'file_descriptor' is NOT copyable, but it
//can be returned from functions thanks to move semantics
file_descriptor create_file_descriptor(const char *filename)
{  return file_descriptor(filename);  }

int main()
{
   //Open a file obtaining its descriptor, the temporary
   //returned from 'create_file_descriptor' is moved to 'fd'.
   file_descriptor fd = create_file_descriptor("filename");
   assert(!fd.empty());

   //Now move fd into a vector
   boost::container::vector<file_descriptor> v;
   v.push_back(boost::move(fd));

   //Check ownership has been transferred
   assert(fd.empty());
   assert(!v[0].empty());

   //Compilation error if uncommented since file_descriptor is not copyable
   //and vector copy construction requires value_type's copy constructor:
   //boost::container::vector<file_descriptor> v2(v);
   return 0;
}
```

Constructor Forwarding

Consider writing a generic factory function that returns an object for a newly constructed generic type. Factory functions such as this are valuable for encapsulating and localizing the allocation of resources. Obviously, the factory function must accept exactly the same sets of arguments as the constructors of the type of objects constructed:

```
template<class T> T* factory_new()
{  return new T();  }

template<class T> T* factory_new(a1)
{  return new T(a1);  }

template<class T> T* factory_new(a1, a2)
{  return new T(a1, a2);  }
```

Unfortunately, in C++03 the much bigger issue with this approach is that the N-argument case would require 2^N overloads, immediately discounting this as a general solution. Fortunately, most constructors take arguments by value, by const-reference or by rvalue reference. If these limitations are accepted, the forwarding emulation of a N-argument case requires just N overloads. This library makes this emulation easy with the help of `BOOST_FWD_REF` and `boost::forward`:

```
#include <boost/move/utility.hpp>
#include <iostream>

class copyable_only_tester
{
   public:
   copyable_only_tester()
   {  std::cout << "copyable_only_tester()" << std::endl;   }

   copyable_only_tester(const copyable_only_tester&)
   {  std::cout << "copyable_only_tester(const copyable_only_tester&)" << std::endl;   }

   copyable_only_tester(int)
   {  std::cout << "copyable_only_tester(int)" << std::endl;   }

   copyable_only_tester(int, double)
   {  std::cout << "copyable_only_tester(int, double)" << std::endl;   }
};

class copyable_movable_tester
{
   // move semantics
   BOOST_COPYABLE_AND_MOVABLE(copyable_movable_tester)
   public:

   copyable_movable_tester()
   {  std::cout << "copyable_movable_tester()" << std::endl;   }

   copyable_movable_tester(int)
   {  std::cout << "copyable_movable_tester(int)" << std::endl;   }

   copyable_movable_tester(BOOST_RV_REF(copyable_movable_tester))
  {  std::cout << "copyable_movable_tester(BOOST_RV_REF(copyable_movable_tester))" << std::endl; ↵
  }

   copyable_movable_tester(const copyable_movable_tester &)
   {  std::cout << "copyable_movable_tester(const copyable_movable_tester &)" << std::endl;   }

   copyable_movable_tester(BOOST_RV_REF(copyable_movable_tester), BOOST_RV_REF(copyable_mov↵
able_tester))
```

```
   {  std::cout << "copyable_movable_tester(BOOST_RV_REF(copyable_movable_tester), ↵
BOOST_RV_REF(copyable_movable_tester))" << std::endl;   }

   copyable_movable_tester &operator=(BOOST_RV_REF(copyable_movable_tester))
   {  std::cout << "copyable_movable_tester & operator=(BOOST_RV_REF(copyable_movable_test↵
er))" << std::endl;
      return *this;  }

   copyable_movable_tester &operator=(BOOST_COPY_ASSIGN_REF(copyable_movable_tester))
   {  std::cout << "copyable_movable_tester & operator=(BOOST_COPY_ASSIGN_REF(copyable_movable_test↵
er))" << std::endl;
      return *this;  }
};

//1 argument
template<class MaybeMovable, class MaybeRv>
void function_construct(BOOST_FWD_REF(MaybeRv) x)
{  MaybeMovable m(boost::forward<MaybeRv>(x));   }

//2 argument
template<class MaybeMovable, class MaybeRv, class MaybeRv2>
void function_construct(BOOST_FWD_REF(MaybeRv) x, BOOST_FWD_REF(MaybeRv2) x2)
{  MaybeMovable m(boost::forward<MaybeRv>(x), boost::forward<MaybeRv2>(x2));  }

int main()
{
   copyable_movable_tester m;
   //move constructor
   function_construct<copyable_movable_tester>(boost::move(m));
   //copy constructor
   function_construct<copyable_movable_tester>(copyable_movable_tester());
   //two rvalue constructor
   function_construct<copyable_movable_tester>(boost::move(m), boost::move(m));

   copyable_only_tester nm;
   //copy constructor (copyable_only_tester has no move ctor.)
   function_construct<copyable_only_tester>(boost::move(nm));
   //copy constructor
   function_construct<copyable_only_tester>(nm);
   //int constructor
   function_construct<copyable_only_tester>(int(0));
   //int, double constructor
   function_construct<copyable_only_tester>(int(0), double(0.0));

   //Output is:
   //copyable_movable_tester()
   //copyable_movable_tester(BOOST_RV_REF(copyable_movable_tester))
   //copyable_movable_tester()
   //copyable_movable_tester(const copyable_movable_tester &)
   //copyable_movable_tester(BOOST_RV_REF(copyable_movable_tester), BOOST_RV_REF(copyable_mov↵
able_tester))
   //copyable_only_tester()
   //copyable_only_tester(const copyable_only_tester&)
   //copyable_only_tester(const copyable_only_tester&)
   //copyable_only_tester(int)
   //copyable_only_tester(int, double)
   return 0;
}
```

Constructor forwarding comes in handy to implement placement insertion in containers with just N overloads if the implementor accepts the limitations of this type of forwarding for C++03 compilers. In compilers with rvalue references perfect forwarding is achieved.

Move iterators

```
template<class Iterator>
class move_iterator;

template<class It>
move_iterator<It> make_move_iterator(const It &it);
```

`move_iterator` is an iterator adaptor with the same behavior as the underlying iterator except that its dereference operator implicitly converts the value returned by the underlying iterator's dereference operator to an rvalue reference: `boost::move(*underlying_iterator)` It is a read-once iterator, but can have up to random access traversal characteristics.

`move_iterator` is very useful because some generic algorithms and container insertion functions can be called with move iterators to replace copying with moving. For example:

```
//header file "movable.hpp"
#include <boost/move/core.hpp>
#include <boost/move/traits.hpp>

//A movable class
class movable
{
   BOOST_MOVABLE_BUT_NOT_COPYABLE(movable)
   int value_;

   public:
   movable() : value_(1){}

   //Move constructor and assignment
   movable(BOOST_RV_REF(movable) m)
   {  value_ = m.value_;   m.value_ = 0;  }

   movable & operator=(BOOST_RV_REF(movable) m)
   {  value_ = m.value_;   m.value_ = 0;  return *this;  }

   bool moved() const //Observer
   {  return value_ == 0; }
};

namespace boost{

template<>
struct has_nothrow_move<movable>
{
   static const bool value = true;
};

}  //namespace boost{
```

`movable` objects can be moved from one container to another using move iterators and insertion and assignment operations.w

```
#include <boost/container/vector.hpp>
#include "movable.hpp"
#include <cassert>

int main()
{
   using namespace ::boost::container;

   //Create a vector with 10 default constructed objects
   vector<movable> v(10);
   assert(!v[0].moved());

   //Move construct all elements in v into v2
   vector<movable> v2( boost::make_move_iterator(v.begin())
                     , boost::make_move_iterator(v.end()));
   assert(v[0].moved());
   assert(!v2[0].moved());

   //Now move assign all elements from in v2 back into v
   v.assign( boost::make_move_iterator(v2.begin())
           , boost::make_move_iterator(v2.end()));
   assert(v2[0].moved());
   assert(!v[0].moved());

   return 0;
}
```

Move inserters

Similar to standard insert iterators, it's possible to deal with move insertion in the same way as writing into an array. A special kind of iterator adaptors, called move insert iterators, are provided with this library. With regular iterator classes,

```
while (first != last) *result++ = *first++;
```

causes a range [first,last) to be copied into a range starting with result. The same code with result being an move insert iterator will move insert corresponding elements into the container. This device allows all of the copying algorithms in the library to work in the move insert mode instead of the regular overwrite mode. This library offers 3 move insert iterators and their helper functions:

```
// Note: C models Container
template <typename C>
class back_move_insert_iterator;

template <typename C>
back_move_insert_iterator<C> back_move_inserter(C& x);

template <typename C>
class front_move_insert_iterator;

template <typename C>
front_move_insert_iterator<C> front_move_inserter(C& x);

template <typename C>
class move_insert_iterator;

template <typename C>
move_insert_iterator<C> move_inserter(C& x, typename C::iterator it);
```

A move insert iterator is constructed from a container and possibly one of its iterators pointing to where insertion takes place if it is neither at the beginning nor at the end of the container. Insert iterators satisfy the requirements of output iterators. `operator*` returns the move insert iterator itself. The assignment `operator=(T& x)` is defined on insert iterators to allow writing into them, it inserts x right before where the insert iterator is pointing. In other words, an `insert iterator` is like a cursor pointing into the container where the insertion takes place. `back_move_iterator` move inserts elements at the end of a container, `front_insert_iterator` move inserts elements at the beginning of a container, and `move_insert_iterator` move inserts elements where the iterator points to in a container. `back_move_inserter`, `front_move_inserter`, and `move_inserter` are three functions making the insert iterators out of a container. Here's an example of how to use them:

```
#include <boost/container/list.hpp>
#include "movable.hpp"
#include <cassert>
#include <algorithm>

using namespace ::boost::container;

typedef list<movable> list_t;
typedef list_t::iterator l_iterator;

template<class MoveInsertIterator>
void test_move_inserter(list_t &l2, MoveInsertIterator mit)
{
   //Create a list with 10 default constructed objects
   list<movable> l(10);
   assert(!l.begin()->moved());
   l2.clear();

   //Move insert into l2 containers
   std::copy(l.begin(), l.end(), mit);

   //Check size and status
   assert(l2.size() == l.size());
   assert(l.begin()->moved());
   assert(!l2.begin()->moved());
}

int main()
{
   list_t l2;
   test_move_inserter(l2, boost::back_move_inserter(l2));
   test_move_inserter(l2, boost::front_move_inserter(l2));
   test_move_inserter(l2, boost::move_inserter(l2, l2.end()));
   return 0;
}
```

Move algorithms

The standard library offers several copy-based algorithms. Some of them, like `std::copy` or `std::uninitialized_copy` are basic building blocks for containers and other data structures. This library offers move-based functions for those purposes:

```
template<typename I, typename O> O move(I, I, O);
template<typename I, typename O> O move_backward(I, I, O);
template<typename I, typename F> F uninitialized_move(I, I, F);
template<typename I, typename F> F uninitialized_copy_or_move(I, I, F);
```

The first 3 are move variations of their equivalent copy algorithms, but copy assignment and copy construction are replaced with move assignment and construction. The last one has the same behaviour as `std::uninitialized_copy` but since several standand library implementations don't play very well with `move_iterator`s, this version is a portable version for those willing to use move iterators.

```
#include "movable.hpp"
#include <boost/move/algorithm.hpp>
#include <cassert>
#include <boost/aligned_storage.hpp>

int main()
{
   const std::size_t ArraySize = 10;
   movable movable_array[ArraySize];
   movable movable_array2[ArraySize];
   //move
   boost::move(&movable_array2[0], &movable_array2[ArraySize], &movable_array[0]);
   assert(movable_array2[0].moved());
   assert(!movable_array[0].moved());

   //move backward
  boost::move_backward(&movable_array[0], &movable_array[ArraySize], &movable_array2[ArraySize]);
   assert(movable_array[0].moved());
   assert(!movable_array2[0].moved());

   //uninitialized_move
   boost::aligned_storage< sizeof(movable)*ArraySize
                         , boost::alignment_of<movable>::value>::type storage;
   movable *raw_movable = static_cast<movable*>(static_cast<void*>(&storage));
   boost::uninitialized_move(&movable_array2[0], &movable_array2[ArraySize], raw_movable);
   assert(movable_array2[0].moved());
   assert(!raw_movable[0].moved());
   return 0;
}
```

Emulation limitations

Like any emulation effort, the library has some limitations users should take in care to achieve portable and efficient code when using the library with C++03 conformant compilers:

Initializing base classes

When initializing base classes in move constructors, users must cast the reference to a base class reference before moving it. Example:

```
//Portable and efficient
Derived(BOOST_RV_REF(Derived) x)             // Move ctor
   : Base(boost::move(static_cast<Base&>(x))),
     mem_(boost::move(x.mem_)) { }
```

If casting is not performed the emulation will not move construct the base class, because no conversion is available from `BOOST_RV_REF(Derived)` to `BOOST_RV_REF(Base)`. Without the cast we might obtain a compilation error (for non-copyable types) or a less-efficient move constructor (for copyable types):

```
//If Derived is copyable, then Base is copy-constructed.
//If not, a compilation error is issued
Derived(BOOST_RV_REF(Derived) x)             // Move ctor
   : Base(boost::move(x)),
     mem_(boost::move(x.mem_)) { }
```

Template parameters for perfect forwarding

The emulation can't deal with C++0x reference collapsing rules that allow perfect forwarding:

```
//C++0x
template<class T>
void forward_function(T &&t)
{  inner_function(std::forward<T>(t); }

//Wrong C++03 emulation
template<class T>
void forward_function(BOOST_RV_REF<T> t)
{  inner_function(boost::forward<T>(t); }
```

In C++03 emulation BOOST_RV_REF doesn't catch any const rlvalues. For more details on forwarding see Constructor Forwarding chapter.

Binding of rvalue references to lvalues

The first rvalue reference proposal allowed the binding of rvalue references to lvalues:

```
func(Type &&t);
//....

Type t;  //Allowed
func(t)
```

Later, as explained in *Fixing a Safety Problem with Rvalue References* this behaviour was considered dangerous and eliminated this binding so that rvalue references adhere to the principle of type-safe overloading: *Every function must be type-safe in isolation, without regard to how it has been overloaded*

Boost.Move can't emulate this type-safe overloading principle for C++03 compilers:

```
//Allowed by move emulation
movable m;
BOOST_RV_REF(movable) r = m;
```

Assignment operator in classes derived from or holding copyable and movable types

The macro BOOST_COPYABLE_AND_MOVABLE needs to define a copy constructor for `copyable_and_movable` taking a non-const parameter in C++03 compilers:

```
//Generated by BOOST_COPYABLE_AND_MOVABLE
copyable_and_movable &operator=(copyable_and_movable&){/**/}
```

Since the non-const overload of the copy constructor is generated, compiler-generated assignment operators for classes containing `copyable_and_movable` will get the non-const copy constructor overload, which will surely surprise users:

```
class holder
{
   copyable_and_movable c;
};

void func(const holder& h)
{
   holder copy_h(h); //<--- ERROR: can't convert 'const holder&' to 'holder&'
   //Compiler-generated copy constructor is non-const:
   // holder& operator(holder &)
   //!!!
}
```

This limitation forces the user to define a const version of the copy assignment, in all classes holding copyable and movable classes which might annoying in some cases.

An alternative is to implement a single `operator =()` for copyable and movable classes using "pass by value" semantics:

```
T& operator=(T x)    // x is a copy of the source; hard work already done
{
   swap(*this, x);  // trade our resources for x's
   return *this;    // our (old) resources get destroyed with x
}
```

However, "pass by value" is not optimal for classes (like containers, strings, etc.) that reuse resources (like previously allocated memory) when x is assigned from a lvalue.

How the library works

Boost.Move is based on macros that are expanded to true rvalue references in C++0x compilers and emulated rvalue reference classes and conversion operators in C++03 compilers.

In C++03 compilers **Boost.Move** defines a class named `::boost::rv`:

```
template <class T>
class rv : public T
{
   rv();
   ~rv();
   rv(rv const&);
   void operator=(rv const&);
};
```

which is convertible to the movable base class (usual C++ derived to base conversion). When users mark their classes as `BOOST_MOVABLE_BUT_NOT_COPYABLE` or `BOOST_COPYABLE_AND_MOVABLE`, these macros define conversion operators to references to `::boost::rv`:

```
#define BOOST_MOVABLE_BUT_NOT_COPYABLE(TYPE)\
   public:\
   operator ::boost::rv<TYPE>&() \
   {  return *static_cast< ::boost::rv<TYPE>* >(this);  }\
   operator const ::boost::rv<TYPE>&() const \
   {  return static_cast<const ::boost::rv<TYPE>* >(this);  }\
   private:\
   //More stuff...
```

`BOOST_MOVABLE_BUT_NOT_COPYABLE` also declares a private copy constructor and assignment. `BOOST_COPYABLE_AND_MOVABLE` defines a non-const copy constructor `TYPE &operator=(TYPE&)` that forwards to a const version:

```
#define BOOST_COPYABLE_AND_MOVABLE(TYPE)\
   public:\
   TYPE& operator=(TYPE &t)\
   {  this->operator=(static_cast<const ::boost::rv<TYPE> &>(const_cast<const TYPE &>(t))); re↵
turn *this;}\
   //More stuff...
```

In C++0x compilers `BOOST_COPYABLE_AND_MOVABLE` expands to nothing and `BOOST_MOVABLE_BUT_NOT_COPYABLE` declares copy constructor and assigment operator private.

When users define the `BOOST_RV_REF` overload of a copy constructor/assignment, in C++0x compilers it is expanded to a rvalue reference (`T&&`) overload and in C++03 compilers it is expanded to a `::boost::rv<T> &` overload:

```
#define BOOST_RV_REF(TYPE) ::boost::rv< TYPE >& \
```

When users define the `BOOST_COPY_ASSIGN_REF` overload, it is expanded to a usual copy assignment (`const T &`) overload in C++0x compilers and to a `const ::boost::rv &` overload in C++03 compilers:

```
#define BOOST_COPY_ASSIGN_REF(TYPE) const ::boost::rv< TYPE >&
```

As seen, in **Boost.Move** generates efficient and clean code for C++0x move semantics, without modifying any resolution overload. For C++03 compilers when overload resolution is performed these are the bindings:

- a) non-const rvalues (e.g.: temporaries), bind to `::boost::rv< TYPE >&`

- b) const rvalue and lvalues, bind to `const ::boost::rv< TYPE >&`
- c) non-const lvalues (e.g. non-const references) bind to `TYPE&`

The library does not define the equivalent of `BOOST_COPY_ASSIGN_REF` for copy construction (say, `BOOST_COPY_CTOR_REF`) because nearly all modern compilers implement RVO and this is much more efficient than any move emulation. `move` just casts `TYPE &` into `::boost::rv<TYPE> &`.

Here's an example that demostrates how different rlvalue objects bind to `::boost::rv` references in the presence of three overloads and the conversion operators in C++03 compilers:

```
#include <boost/move/core.hpp>
#include <iostream>

class sink_tester
{
   public: //conversions provided by BOOST_COPYABLE_AND_MOVABLE
   operator ::boost::rv<sink_tester>&()
      {  return *static_cast< ::boost::rv<sink_tester>* >(this);  }
   operator const ::boost::rv<sink_tester>&() const
      {  return *static_cast<const ::boost::rv<sink_tester>* >(this);  }
};

//Functions returning different r/lvalue types
      sink_tester    rvalue()       {  return sink_tester(); }
const sink_tester    const_rvalue() {  return sink_tester(); }
      sink_tester &  lvalue()       {  static sink_tester lv; return lv; }
const sink_tester &  const_lvalue() {  static const sink_tester clv = sink_tester(); return clv; }

//BOOST_RV_REF overload
void sink(::boost::rv<sink_tester> &)      { std::cout << "non-const rvalue ↵
catched" << std::endl; }
//BOOST_COPY_ASSIGN_REF overload
void sink(const ::boost::rv<sink_tester> &){ std::cout << "const (r-l)value ↵
catched" << std::endl; }
//Overload provided by BOOST_COPYABLE_AND_MOVABLE
void sink(sink_tester &)                   { std::cout << "non-const lvalue ↵
catched" << std::endl; }

int main()
{
   sink(const_rvalue());   //"const (r-l)value catched"
   sink(const_lvalue());   //"const (r-l)value catched"
   sink(lvalue());         //"non-const lvalue catched"
   sink(rvalue());         //"non-const rvalue catched"
   return 0;
}
```

Thanks and credits

Thanks to all that developed ideas for move emulation: the first emulation was based on Howard Hinnant emulation code for `unique_ptr`, David Abrahams suggested the use of `class rv`, and Klaus Triendl discovered how to bind const rlvalues using `class rv`.

Many thanks to all boosters that have tested, reviewed and improved the library.

Release Notes

Boost 1.53 Release

- Better header segregation (bug #6524).
- Small documentation fixes
- Replaced deprecated BOOST_NO_XXXX with newer BOOST_NO_CXX11_XXX macros.
- Fixed #7830, #7832.

Boost 1.51 Release

- Fixed bugs #7095, #7031.

Boost 1.49 Release

- Fixed bugs #6417, #6183, #6185, #6395, #6396,

Reference

Header <boost/move/algorithm.hpp>

```
namespace boost {
  template<typename I, typename O> O move(I, I, O);
  template<typename I, typename O> O move_backward(I, I, O);
  template<typename I, typename F> F uninitialized_move(I, I, F);
  template<typename I, typename F> F uninitialized_copy_or_move(I, I, F);
  template<typename I, typename F> F copy_or_move(I, I, F);
}
```

Function template move

boost::move

Synopsis

```
// In header: <boost/move/algorithm.hpp>


template<typename I, typename O> O move(I f, I l, O result);
```

Description

Effects: Moves elements in the range [first,last) into the range [result,result + (last - first)) starting from first and proceeding to last. For each non-negative integer n < (last-first), performs *(result + n) = boost::move (*(first + n)).

Effects: result + (last - first).

Requires: result shall not be in the range [first,last).

Complexity: Exactly last - first move assignments.

Function template move_backward

boost::move_backward

Synopsis

```
// In header: <boost/move/algorithm.hpp>


template<typename I, typename O> O move_backward(I f, I l, O result);
```

Description

Effects: Moves elements in the range [first,last) into the range [result - (last-first),result) starting from last - 1 and proceeding to first. For each positive integer n <= (last - first), performs *(result - n) = boost::move(*(last - n)).

Requires: result shall not be in the range [first,last).

Returns: result - (last - first).

Complexity: Exactly last - first assignments.

Function template uninitialized_move

boost::uninitialized_move — defined(BOOST_MOVE_USE_STANDARD_LIBRARY_MOVE)

Synopsis

```
// In header: <boost/move/algorithm.hpp>


template<typename I, typename F> F uninitialized_move(I f, I l, F r);
```

Description

Effects:

```
for (; first != last; ++result, ++first)
   new (static_cast<void*>(&*result))
      typename iterator_traits<ForwardIterator>::value_type(boost::move(*first));
```

Returns: result

Function template uninitialized_copy_or_move

boost::uninitialized_copy_or_move

Synopsis

```
// In header: <boost/move/algorithm.hpp>


template<typename I, typename F> F uninitialized_copy_or_move(I f, I l, F r);
```

Description

Effects:

```
for (; first != last; ++result, ++first)
   new (static_cast<void*>(&*result))
      typename iterator_traits<ForwardIterator>::value_type(*first);
```

Returns: result

Note: This function is provided because *std::uninitialized_copy* from some STL implementations is not compatible with `move_iterator`

Function template copy_or_move

boost::copy_or_move

Synopsis

```
// In header: <boost/move/algorithm.hpp>


template<typename I, typename F> F copy_or_move(I f, I l, F r);
```

Description

Effects:

```
for (; first != last; ++result, ++first)
   *result = *first;
```

Returns: result

Note: This function is provided because *std::uninitialized_copy* from some STL implementations is not compatible with *move_iterator*

Header <boost/move/core.hpp>

This header implements macros to define movable classes and move-aware functions

```


BOOST_MOVABLE_BUT_NOT_COPYABLE(TYPE)
BOOST_COPYABLE_AND_MOVABLE(TYPE)
BOOST_RV_REF(TYPE)
BOOST_RV_REF_BEG
BOOST_RV_REF_END
BOOST_COPY_ASSIGN_REF(TYPE)
BOOST_FWD_REF(TYPE)
```

```

template<typename T> struct has_move_emulation_enabled;
```

Struct template has_move_emulation_enabled

has_move_emulation_enabled

Synopsis

```
// In header: <boost/move/core.hpp>

template<typename T>
struct has_move_emulation_enabled {

  // public data members
  static const bool value;
};
```

Description

This trait yields to a compile-time true boolean if T was marked as BOOST_MOVABLE_BUT_NOT_COPYABLE or BOOST_COPYABLE_AND_MOVABLE and rvalue references are not available on the platform. False otherwise.

Macro BOOST_MOVABLE_BUT_NOT_COPYABLE

BOOST_MOVABLE_BUT_NOT_COPYABLE

Synopsis

```
// In header: <boost/move/core.hpp>

BOOST_MOVABLE_BUT_NOT_COPYABLE(TYPE)
```

Description

This macro marks a type as movable but not copyable, disabling copy construction and assignment. The user will need to write a move constructor/assignment as explained in the documentation to fully write a movable but not copyable class.

Macro BOOST_COPYABLE_AND_MOVABLE

BOOST_COPYABLE_AND_MOVABLE

Synopsis

```
// In header: <boost/move/core.hpp>

BOOST_COPYABLE_AND_MOVABLE(TYPE)
```

Description

This macro marks a type as copyable and movable. The user will need to write a move constructor/assignment and a copy assignment as explained in the documentation to fully write a copyable and movable class.

Macro BOOST_RV_REF

BOOST_RV_REF

Synopsis

```
// In header: <boost/move/core.hpp>

BOOST_RV_REF(TYPE)
```

Description

This macro is used to achieve portable syntax in move constructors and assignments for classes marked as BOOST_COPYABLE_AND_MOVABLE or BOOST_MOVABLE_BUT_NOT_COPYABLE

Macro BOOST_RV_REF_BEG

BOOST_RV_REF_BEG

Synopsis

```
// In header: <boost/move/core.hpp>

BOOST_RV_REF_BEG
```

Description

This macro is used to achieve portable syntax in move constructors and assignments for template classes marked as BOOST_COPYABLE_AND_MOVABLE or BOOST_MOVABLE_BUT_NOT_COPYABLE. As macros have problems with comma-separatd template arguments, the template argument must be preceded with BOOST_RV_REF_START and ended with BOOST_RV_REF_END

Macro BOOST_RV_REF_END

BOOST_RV_REF_END

Synopsis

```
// In header: <boost/move/core.hpp>

BOOST_RV_REF_END
```

Description

This macro is used to achieve portable syntax in move constructors and assignments for template classes marked as BOOST_COPYABLE_AND_MOVABLE or BOOST_MOVABLE_BUT_NOT_COPYABLE. As macros have problems with comma-separatd template arguments, the template argument must be preceded with BOOST_RV_REF_START and ended with BOOST_RV_REF_END

Macro BOOST_COPY_ASSIGN_REF

BOOST_COPY_ASSIGN_REF

Synopsis

```
// In header: <boost/move/core.hpp>

BOOST_COPY_ASSIGN_REF(TYPE)
```

Description

This macro is used to achieve portable syntax in copy assignment for classes marked as BOOST_COPYABLE_AND_MOVABLE.

Macro BOOST_FWD_REF

BOOST_FWD_REF

Synopsis

```
// In header: <boost/move/core.hpp>

BOOST_FWD_REF(TYPE)
```

Description

This macro is used to implement portable perfect forwarding as explained in the documentation.

Header <boost/move/iterator.hpp>

```
namespace boost {
  template<typename It> class move_iterator;
  template<typename C> class back_move_insert_iterator;
  template<typename C> class front_move_insert_iterator;
  template<typename C> class move_insert_iterator;
  template<typename It> move_iterator< It > make_move_iterator(const It &);
  template<typename C> back_move_insert_iterator< C > back_move_inserter(C &);
  template<typename C> 
    front_move_insert_iterator< C > front_move_inserter(C &);
  template<typename C> 
    move_insert_iterator< C > move_inserter(C &, typename C::iterator);
}
```

Class template move_iterator

boost::move_iterator

Synopsis

```
// In header: <boost/move/iterator.hpp>

template<typename It>
class move_iterator {
public:
  // types
  typedef It                                                          iterator_type;
  typedef std::iterator_traits< iterator_type >::value_type          value_type;
  typedef value_type &&                                               reference;
  typedef It                                                          pointer;
  typedef std::iterator_traits< iterator_type >::difference_type     difference_type;
  typedef std::iterator_traits< iterator_type >::iterator_category iterator_category;

  // construct/copy/destruct
  move_iterator();
  explicit move_iterator(It);
  template<typename U> move_iterator(const move_iterator< U > &);

  // public member functions
  iterator_type base() const;
  reference operator*() const;
  pointer operator->() const;
  move_iterator & operator++();
  move_iterator< iterator_type > operator++(int);
  move_iterator & operator--();
  move_iterator< iterator_type > operator--(int);
  move_iterator< iterator_type > operator+(difference_type) const;
  move_iterator & operator+=(difference_type);
  move_iterator< iterator_type > operator-(difference_type) const;
  move_iterator & operator-=(difference_type);
  reference operator[](difference_type) const;
};
```

Description

Class template move_iterator is an iterator adaptor with the same behavior as the underlying iterator except that its dereference operator implicitly converts the value returned by the underlying iterator's dereference operator to an rvalue reference. Some generic algorithms can be called with move iterators to replace copying with moving.

`move_iterator` public construct/copy/destruct

1.
```
move_iterator();
```

2.
```
explicit move_iterator(It i);
```

3.
```
template<typename U> move_iterator(const move_iterator< U > & u);
```

`move_iterator` public member functions

1.
```
iterator_type base() const;
```

2.
```
reference operator*() const;
```

3.
```
pointer operator->() const;
```

4.
```
move_iterator & operator++();
```

5.
```
move_iterator< iterator_type > operator++(int);
```

6.
```
move_iterator & operator--();
```

7.
```
move_iterator< iterator_type > operator--(int);
```

8.
```
move_iterator< iterator_type > operator+(difference_type n) const;
```

9.
```
move_iterator & operator+=(difference_type n);
```

10.
```
move_iterator< iterator_type > operator-(difference_type n) const;
```

11.
```
move_iterator & operator-=(difference_type n);
```

12.
```
reference operator[](difference_type n) const;
```

Class template back_move_insert_iterator

boost::back_move_insert_iterator

Synopsis

```
// In header: <boost/move/iterator.hpp>

template<typename C>
class back_move_insert_iterator :
  public std::iterator< std::output_iterator_tag, void, void, void, void >
{
public:
  // types
  typedef C             container_type;
  typedef C::value_type value_type;
  typedef C::reference  reference;

  // construct/copy/destruct
  explicit back_move_insert_iterator(C &);
  back_move_insert_iterator& operator=(reference);
  back_move_insert_iterator& operator=(BOOST_RV_REF(value_type));

  // public member functions
  back_move_insert_iterator & operator*();
  back_move_insert_iterator & operator++();
  back_move_insert_iterator & operator++(int);
};
```

Description

A move insert iterator that move constructs elements at the back of a container

`back_move_insert_iterator` public construct/copy/destruct

1.
```
explicit back_move_insert_iterator(C & x);
```

2.
```
back_move_insert_iterator& operator=(reference x);
```

3.
```
back_move_insert_iterator& operator=(BOOST_RV_REF(value_type) x);
```

`back_move_insert_iterator` public member functions

1.
```
back_move_insert_iterator & operator*();
```

2.
```
back_move_insert_iterator & operator++();
```

3.
```
back_move_insert_iterator & operator++(int);
```

Class template front_move_insert_iterator

boost::front_move_insert_iterator

Synopsis

```
// In header: <boost/move/iterator.hpp>

template<typename C>
class front_move_insert_iterator :
  public std::iterator< std::output_iterator_tag, void, void, void, void >
{
public:
  // types
  typedef C              container_type;
  typedef C::value_type value_type;
  typedef C::reference  reference;

  // construct/copy/destruct
  explicit front_move_insert_iterator(C &);
  front_move_insert_iterator& operator=(reference);
  front_move_insert_iterator& operator=(BOOST_RV_REF(value_type));

  // public member functions
  front_move_insert_iterator & operator*();
  front_move_insert_iterator & operator++();
  front_move_insert_iterator & operator++(int);
};
```

Description

A move insert iterator that move constructs elements int the front of a container

`front_move_insert_iterator` public construct/copy/destruct

1. ```
   explicit front_move_insert_iterator(C & x);
   ```

2. ```
   front_move_insert_iterator& operator=(reference x);
   ```

3. ```
   front_move_insert_iterator& operator=(BOOST_RV_REF(value_type) x);
   ```

`front_move_insert_iterator` public member functions

1. ```
   front_move_insert_iterator & operator*();
   ```

2. ```
   front_move_insert_iterator & operator++();
   ```

3. ```
   front_move_insert_iterator & operator++(int);
   ```

Class template move_insert_iterator

boost::move_insert_iterator

Synopsis

```
// In header: <boost/move/iterator.hpp>

template<typename C>
class move_insert_iterator :
  public std::iterator< std::output_iterator_tag, void, void, void, void >
{
public:
  // types
  typedef C              container_type;
  typedef C::value_type value_type;
  typedef C::reference  reference;

  // construct/copy/destruct
  explicit move_insert_iterator(C &, typename C::iterator);
  move_insert_iterator& operator=(reference);
  move_insert_iterator& operator=(BOOST_RV_REF(value_type));

  // public member functions
  move_insert_iterator & operator*();
  move_insert_iterator & operator++();
  move_insert_iterator & operator++(int);
};
```

Description

`move_insert_iterator` public construct/copy/destruct

1. ```
   explicit move_insert_iterator(C & x, typename C::iterator pos);
   ```

2. ```
   move_insert_iterator& operator=(reference x);
   ```

3. ```
   move_insert_iterator& operator=(BOOST_RV_REF(value_type) x);
   ```

`move_insert_iterator` public member functions

1. ```
   move_insert_iterator & operator*();
   ```

2. ```
   move_insert_iterator & operator++();
   ```

3. ```
   move_insert_iterator & operator++(int);
   ```

Function template make_move_iterator

boost::make_move_iterator

Synopsis

```
// In header: <boost/move/iterator.hpp>


template<typename It> move_iterator< It > make_move_iterator(const It & it);
```

Description

Returns: move_iterator<It>(i).

Function template back_move_inserter

boost::back_move_inserter

Synopsis

```
// In header: <boost/move/iterator.hpp>


template<typename C> back_move_insert_iterator< C > back_move_inserter(C & x);
```

Description

Returns: back_move_insert_iterator<C>(x).

Function template front_move_inserter

boost::front_move_inserter

Synopsis

```
// In header: <boost/move/iterator.hpp>


template<typename C> 
  front_move_insert_iterator< C > front_move_inserter(C & x);
```

Description

Returns: front_move_insert_iterator<C>(x).

Function template move_inserter

boost::move_inserter

Synopsis

```
// In header: <boost/move/iterator.hpp>


template<typename C>
  move_insert_iterator< C > move_inserter(C & x, typename C::iterator it);
```

Description

Returns: move_insert_iterator<C>(x, it).

Header <boost/move/move.hpp>

A general library header that includes the rest of top-level headers.

Header <boost/move/traits.hpp>

```
namespace boost {
  template<typename T> struct has_trivial_destructor_after_move;
  template<typename T> struct has_nothrow_move;
}
```

Struct template has_trivial_destructor_after_move

boost::has_trivial_destructor_after_move

Synopsis

```
// In header: <boost/move/traits.hpp>

template<typename T>
struct has_trivial_destructor_after_move :
  public boost::has_trivial_destructor< T >
{
};
```

Description

If this trait yields to true (*has_trivial_destructor_after_move <T>::value == true*) means that if T is used as argument of a move construction/assignment, there is no need to call T's destructor. This optimization tipically is used to improve containers' performance.

By default this trait is true if the type has trivial destructor, every class should specialize this trait if it wants to improve performance when inserted in containers.

Struct template has_nothrow_move

boost::has_nothrow_move

Synopsis

```
// In header: <boost/move/traits.hpp>

template<typename T>
struct has_nothrow_move {
};
```

Description

By default this traits returns false. Classes with non-throwing move constructor and assignment can specialize this trait to obtain some performance improvements.

Header <boost/move/utility.hpp>

```
namespace boost {
  template<typename T> struct enable_move_utility_emulation;
  template<typename T> rvalue_reference move(input_reference);
  template<typename T> output_reference forward(input_reference);
}
```

Struct template enable_move_utility_emulation

boost::enable_move_utility_emulation

Synopsis

```
// In header: <boost/move/utility.hpp>

template<typename T>
struct enable_move_utility_emulation {

  // public data members
  static const bool value;
};
```

Description

This trait's internal boolean `value` is false in compilers with rvalue references and true in compilers without rvalue references.

A user can specialize this trait for a type T to false to SFINAE out `move` and `forward` so that the user can define a different move emulation for that type in namespace boost (e.g. another Boost library for its types) and avoid any overload ambiguity.

Function template move

boost::move

Synopsis

```
// In header: <boost/move/utility.hpp>


template<typename T> rvalue_reference move(input_reference);
```

Description

This function provides a way to convert a reference into a rvalue reference in compilers with rvalue references. For other compilers converts T & into *::boost::rv<T> &* so that move emulation is activated.

Function template forward

boost::forward

Synopsis

```
// In header: <boost/move/utility.hpp>


template<typename T> output_reference forward(input_reference);
```

Description

This function provides limited form of forwarding that is usually enough for in-place construction and avoids the exponential overloading for achieve the limited forwarding in C++03.

For compilers with rvalue references this function provides perfect forwarding.

Otherwise:

- If input_reference binds to const ::boost::rv<T> & then it output_reference is ::boost::rv<T> &
- Else, output_reference is equal to input_reference.