# Impact of C++11 Move Semantics on Performance

Francisco Almeida

# Move semantics and performance

- A little bit of background.
  - What are R-value references?
  - What is it good for?

- What about compiler optimizations?
  - Copy elision/return value optimization.

- Execution time comparisons (STL).
  - Using GCC 4.7.0

# L-Value References

- What do we mean by L-value reference?

```
std::string becpp = "BeCppUG";
```

L-value (named object)

`std::string&`

# R-Value References

- What do we mean by R-value reference?

```
std::string GetGroupName()
{
    return std::string("BECppUG");
}
```

R-value (unnamed object)

`std::string&&`

# Construct by moving

- Default constructor
- Parameterized constructor
- Copy constructor
- **Move constructor**

```
MyClass::MyClass(MyClass&& other)
{
   data = other.data;
}
```

# std::move

```
MyClass::MyClass(MyClass&& other)
{
   data = other.data;
}
```

*L-value*

```
MyClass::MyClass(MyClass&& other)
{
   data = std::move(other.data);
}
```

*R-value*

# std::move

```
MyClass& MyClass::operator=(MyClass&& other)
{
   data = other.data;   L-value
   return *this;
}
```

```
MyClass& MyClass::operator=(MyClass&& other)
{
   data = std::move(other.data);   R-value
   return *this;
}
```

# std::move definition

```
template<typename T>
inline typename std::remove_reference<T>::type&&
move(T&& t)
{
   return
      static_cast<typename std::remove_reference<T>::type&&>(t);
}
```

# By the way…

- Even if you don't use it, the STL will!

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

# Default move constructor

- The compiler will provide your class an implicit move constructor if:
  - No user-defined copy constructor or assignment.
  - No user-defined destructor.
  - No user-defined move assignment.

- Your class will also get an implicit move assignment if:
  - No user-defined copy constructor or assignment.
  - No user-defined destructor.
  - No user-define move constructor.

- Strong guarantee required:
    - Copy constructor and destructor have no side effects.
    - Constructors do not throw (tell the compiler `noexcept`).

# Imperfect forwarding

- How to ensure that a reference type is always correctly forwarded?

```
template<typename T, typename U>
shared_ptr<T> create_shared(U& arg)
{
   return shared_ptr<T>(new T(arg));
}
```

```
template<typename T, typename U>
shared_ptr<T> create_shared(const U& arg)
{
   return shared_ptr<T>(new T(arg));
}
```

# C++11 Reference Collapsing Rules

| When passing… | … it becomes… |
|---|---|
| A& & | A& |
| A& && | A& |
| A&& & | A& |
| A&& && | A&& |

```
template<typename T>
inline T&& forward(typename std::remove_reference<T>::type& t) noexcept
{
   return static_cast<T&&>(t);
}
```

# Perfect forwarding

- "One overload to forward them all"

```
template<typename T, typename U>
shared_ptr<T> create_shared(U&& arg)
{
   return shared_ptr<T>(new T(std::forward(arg)));
}
```

# Doesn’t the compiler do all this, anyway?

# Return Value Optimization

```
std::string GetGroupName()
{
   return std::string("BECppUG");
}

// ...

std::string name = GetGroupName();
```

*No copying!*

# Return Value Optimization

- Compiler skips object copying (*elides copy*)
  - Stack frame optimization technique.
  - First introduced by Walter Bright, in the Zortech C ++ Compiler.
- Compiler dependent, and not always guaranteed.

I don’t need “move semantics” to move!

# Explicit swaps do get most of the job done…

```
void MyClass::Swap(MyClass& other)
{
   std::swap(data, other.data);
}
```

```
MyClass obj1;
obj1.Swap(temp);
```

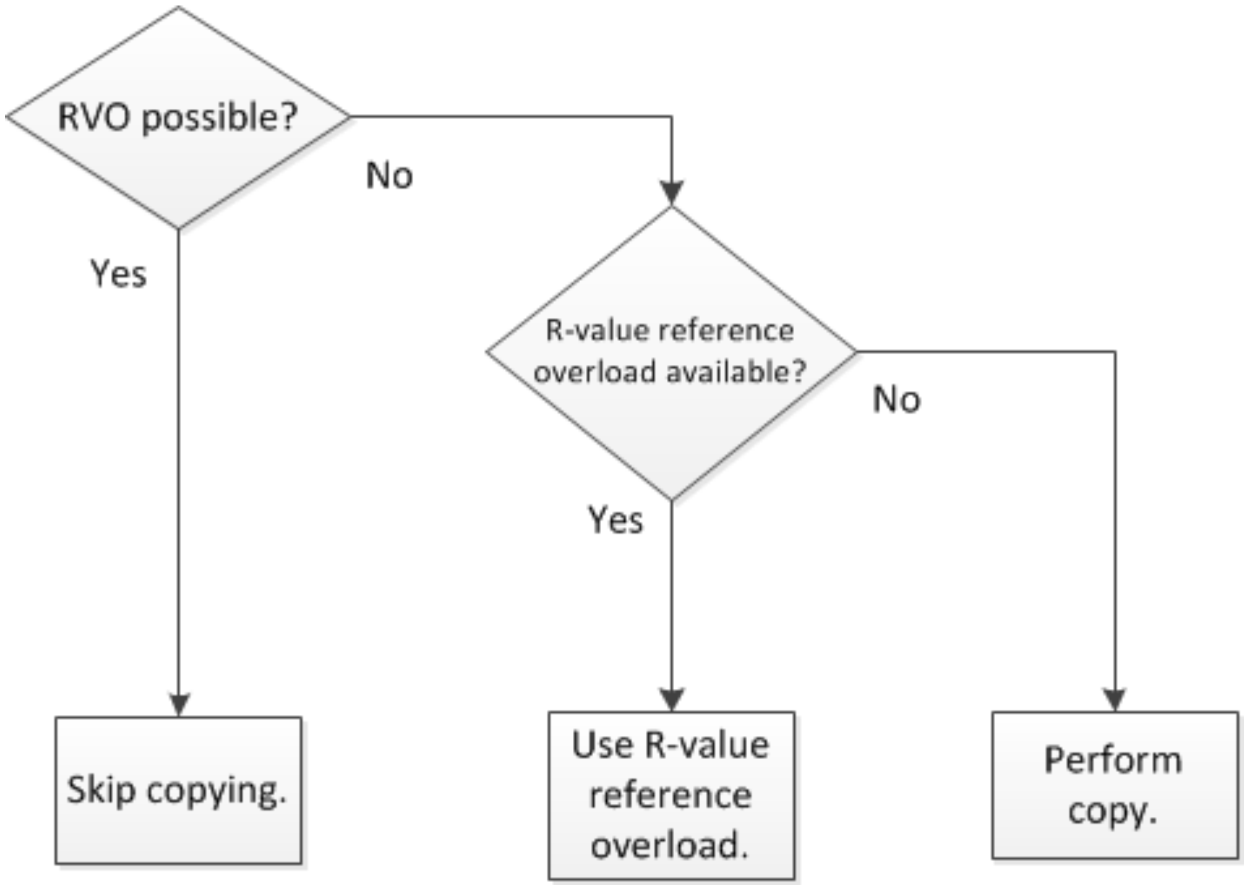
# …and return value optimization does the rest…

- *Copy-and-Swap* idiom:

```
MyClass& MyClass::operator=(MyClass other)
{
    Swap(other);
    return *this;
}
```

…but it is not portable, nor guaranteed to always work.

# Optimal use of copy and move semantics



But does it really make any difference?

# A simple example

- "Benchmark" for comparing move-enabled STL to move-disabled STL performance.

- Tested using GCC 4.7.0
  - Without -std=c++0x (C++98 rules, no move)
  - With -std=c++0x (C++11, use move in STL)

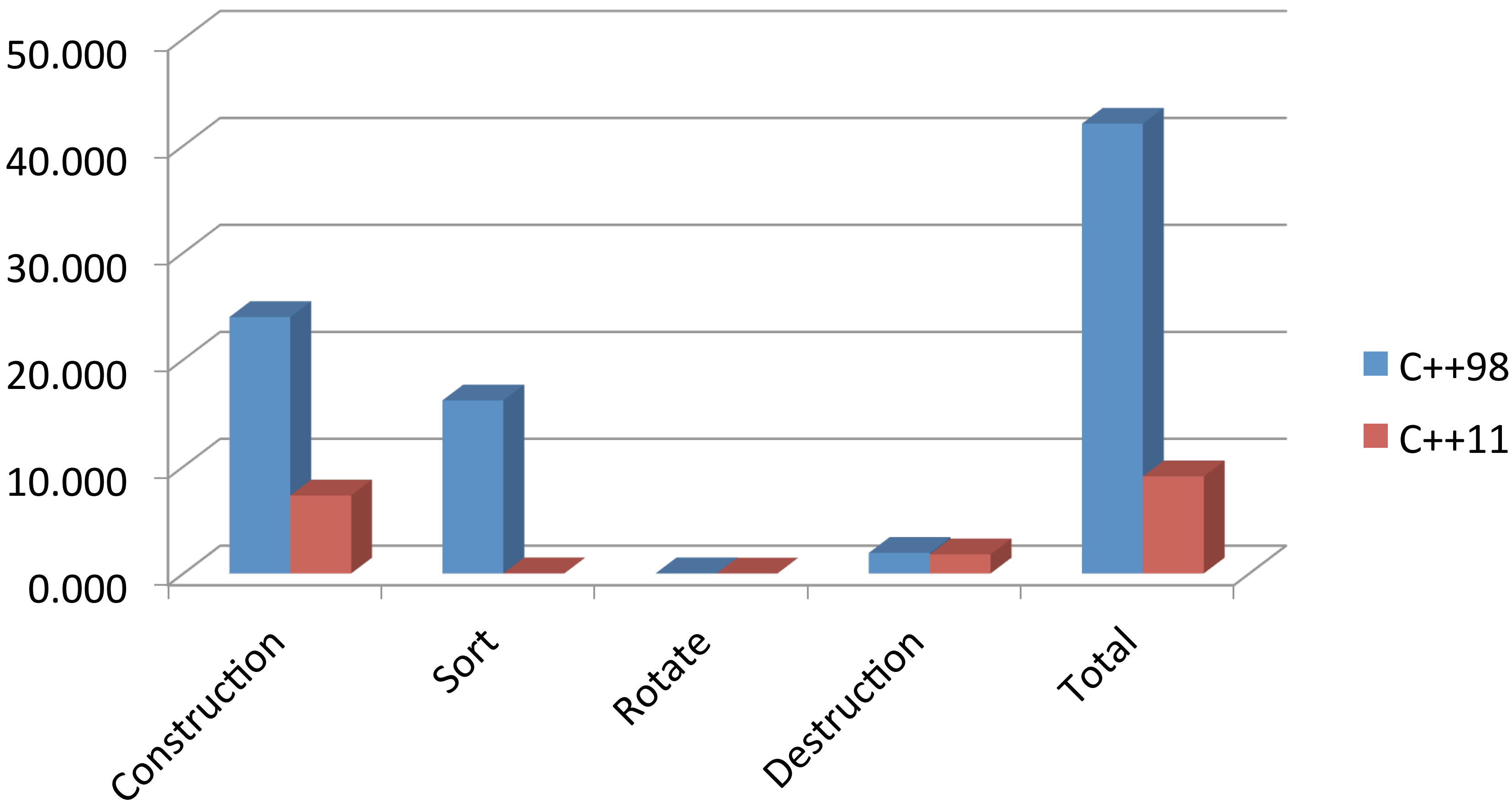
- Refer to:

  http://cpp-next.com/archive/2010/10/howards-stl-move-semantics-benchmark/

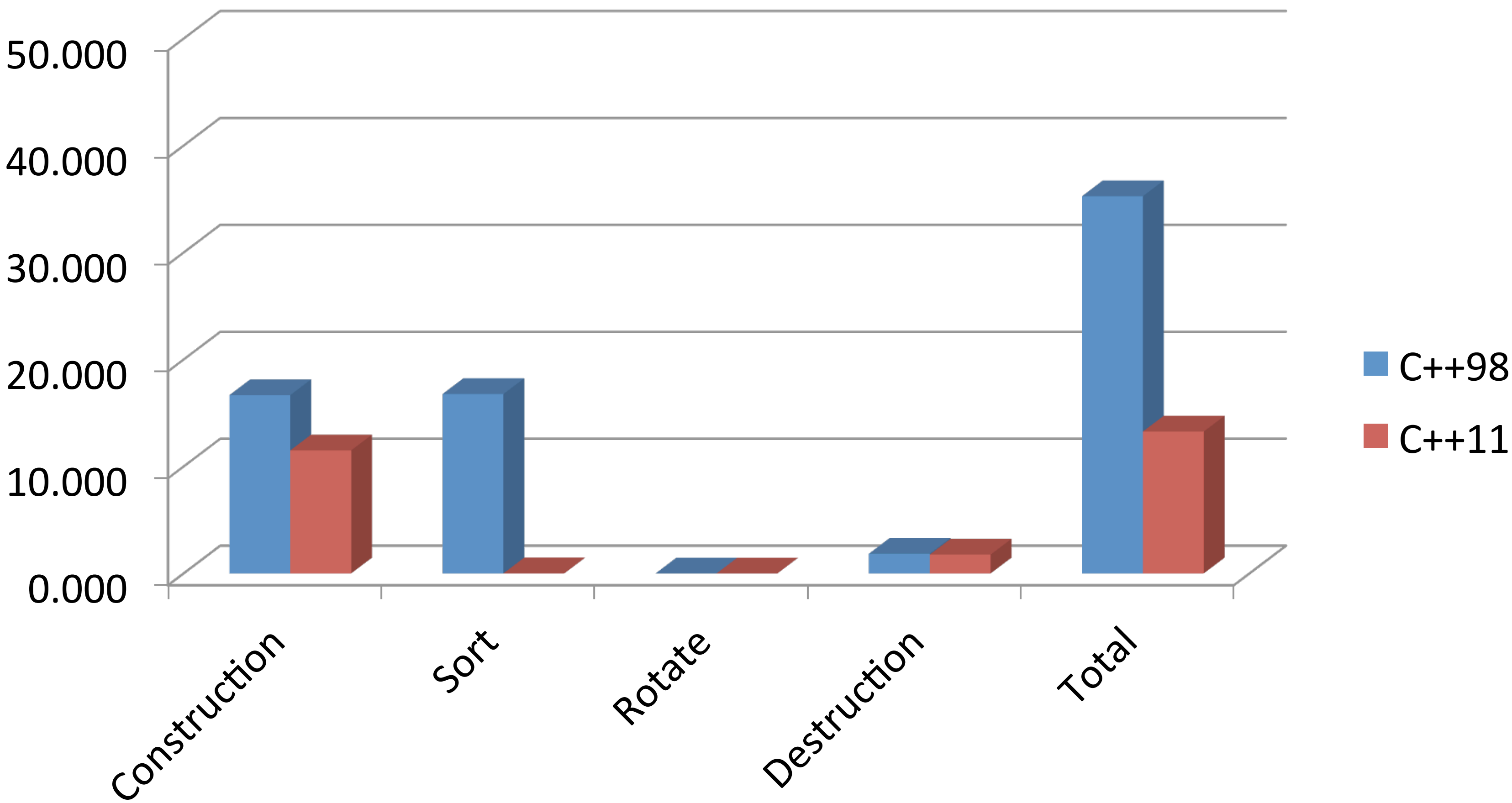
# Howard’s STL move semantics “benchmark”

- Fill a `std::vector` with N `std::set`s of N randomly generated values.
  - We will use N = 5001 here.
- Sort the `std::vector`.
- Rotate the `std::vector` by half its size.

# Execution times comparison (containers passed by value)



# Execution times comparison (containers passed by reference)



# Conclusions

- Move semantics are another optimization tool in the C++ arsenal.
- Profile your code and compare approaches.
  - Use `noexcept` wherever possible.
  - Do not overuse move semantics, you may actually lose performance.
- STL is move-enabled “out of the box”
  - Optimizations for free