Back to Basics:

Move Semantics

(part 1 of 2)

Klaus Iglberger, CppCon 2019

C++ Trainer since 2016

Senior Software Engineer at Siemens

Author of the blaze C++ math library

(Co-)Organizer of the Munich C++ user group

Regular presenter at C++ conferences

Klaus Iglberger

Content

Back to Basics: Move Semantics (Part 1)

- The Basics of Move Semantics
- The New Special Member Functions
  - The Move Constructor
  - The Move Assignment Operator
- Parameter Conventions

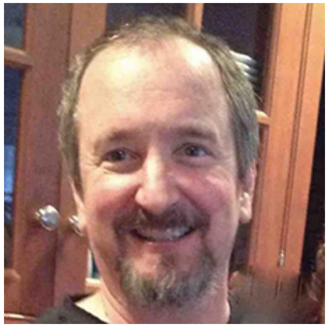
Back to Basics: Move Semantics (Part 2)

- Forwarding References
  - Perfect Forwarding
  - The Perils of Forwarding References
  - Overloading with Forwarding References
- Move Semantics Pitfalls

Acknowledgements

Scott Meyers

Nicolai Josuttis



Howard Hinnant

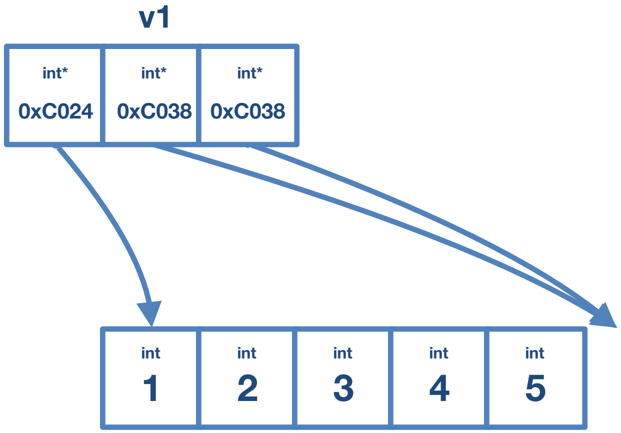
The Basics of Move Semantics

The Basics of Move Semantics

```
std::vector<int> v1{ 1, 2, 3, 4, 5 };
```

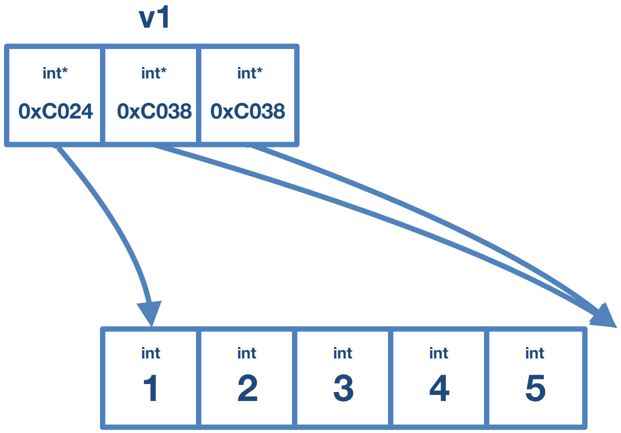
The Basics of Move Semantics

```
std::vector<int> v1{ 1, 2, 3, 4, 5 };
```



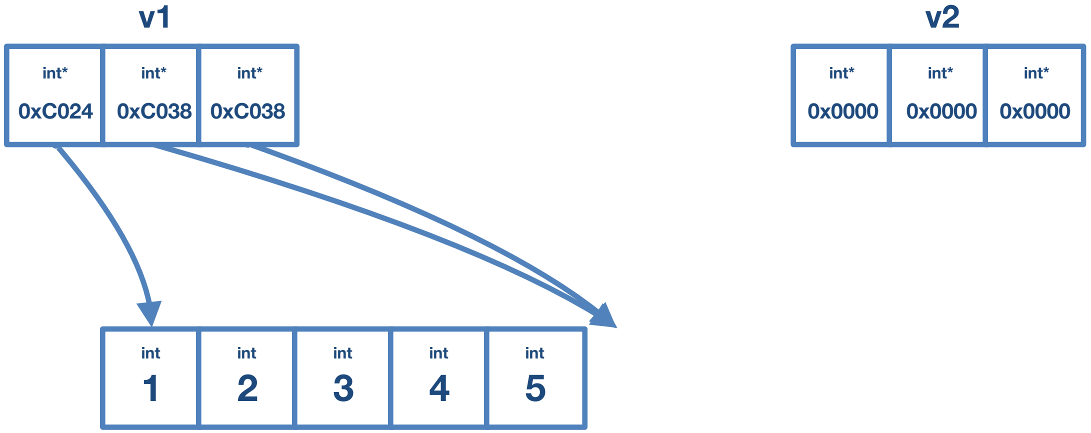
The Basics of Move Semantics

```
std::vector<int> v1{ 1, 2, 3, 4, 5 };
std::vector<int> v2{};
```



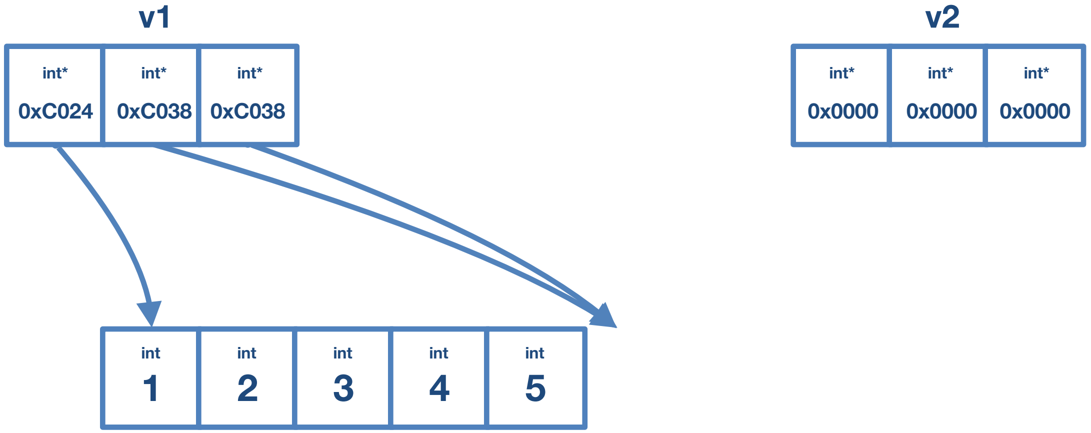
The Basics of Move Semantics

```
std::vector<int> v1{ 1, 2, 3, 4, 5 };
std::vector<int> v2{};
```



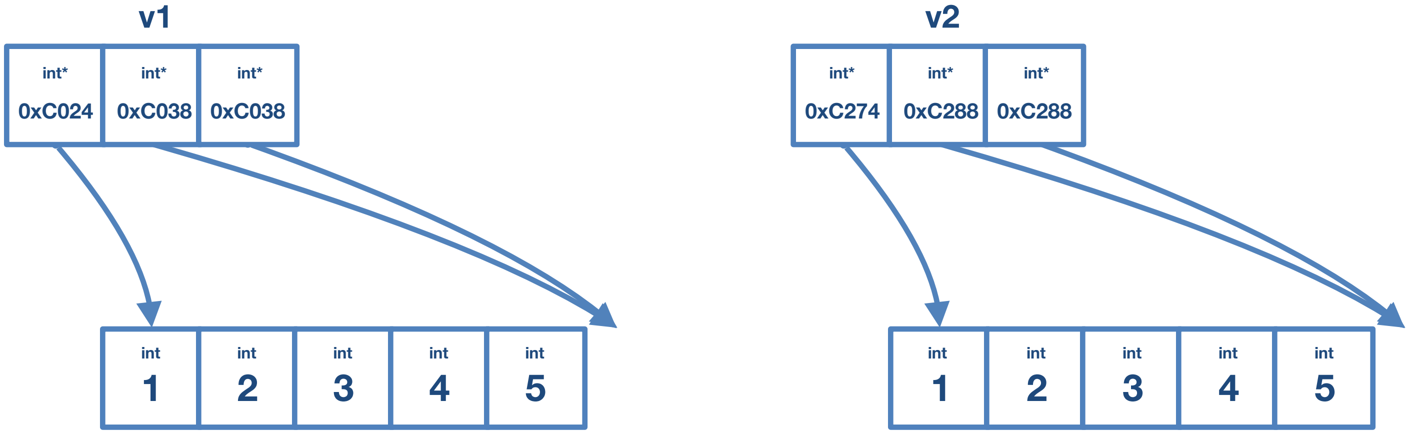
The Basics of Move Semantics

```
std::vector<int> v1{ 1, 2, 3, 4, 5 };
std::vector<int> v2{};

v2 = v1;
```



The Basics of Move Semantics

```
std::vector<int> v1{ 1, 2, 3, 4, 5 };
std::vector<int> v2{};

v2 = v1;
```

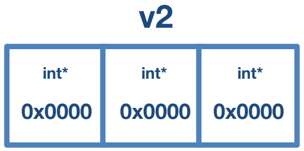


The Basics of Move Semantics

```
std::vector<int> createVector() {
   return std::vector<int>{ 1, 2, 3, 4, 5 };
}

std::vector<int> v2{};
```

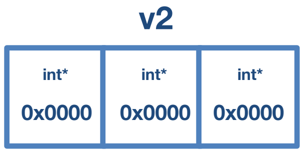
The Basics of Move Semantics

```
std::vector<int> createVector() {
   return std::vector<int>{ 1, 2, 3, 4, 5 };
}

std::vector<int> v2{};
```



The Basics of Move Semantics

```
std::vector<int> createVector() {
   return std::vector<int>{ 1, 2, 3, 4, 5 };
}

std::vector<int> v2{};

v2 = createVector();
```

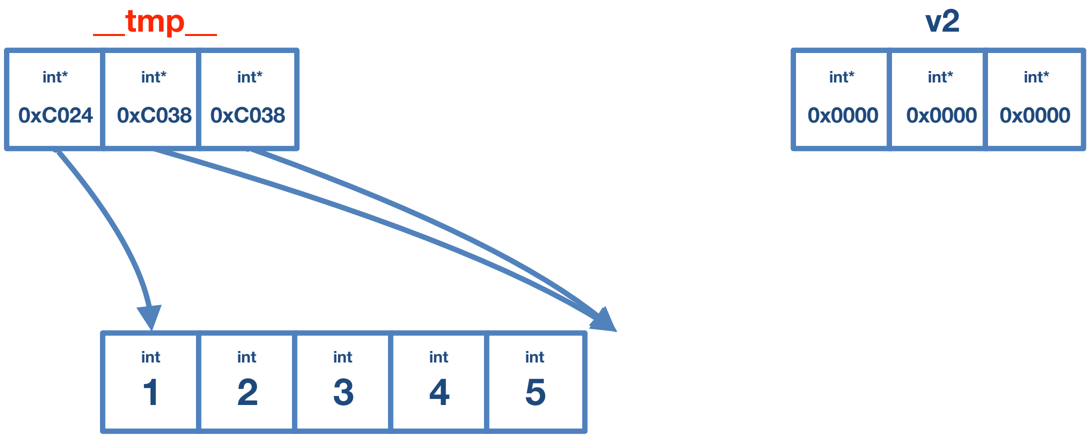


v2

int*	int*	int*
0x0000	0x0000	0x0000

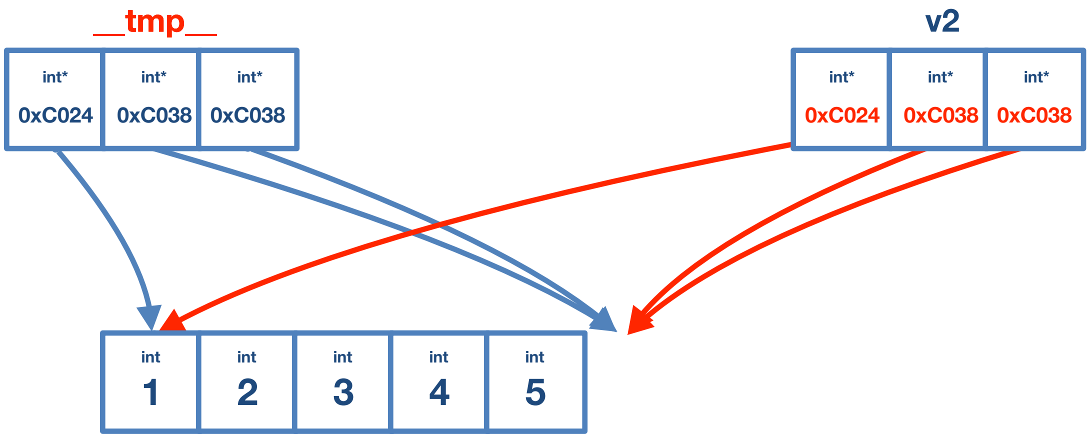
The Basics of Move Semantics

```
std::vector<int> createVector() {
   return std::vector<int>{ 1, 2, 3, 4, 5 };
}

std::vector<int> v2{};

v2 = createVector();
```



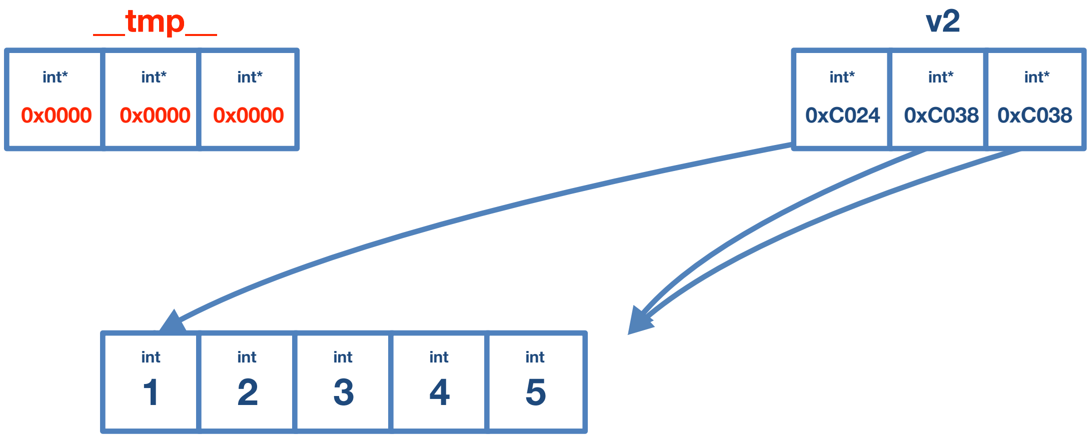
The Basics of Move Semantics

```
std::vector<int> createVector() {
   return std::vector<int>{ 1, 2, 3, 4, 5 };
}

std::vector<int> v2{};

v2 = createVector();
```



The Basics of Move Semantics

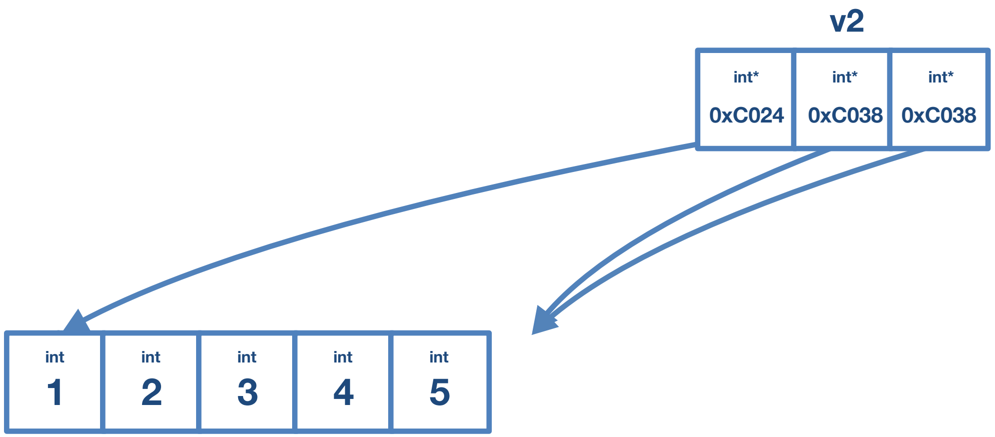
```
std::vector<int> createVector() {
   return std::vector<int>{ 1, 2, 3, 4, 5 };
}

std::vector<int> v2{};

v2 = createVector();
```



Note: This is only possible no one else holds a reference to tmp!

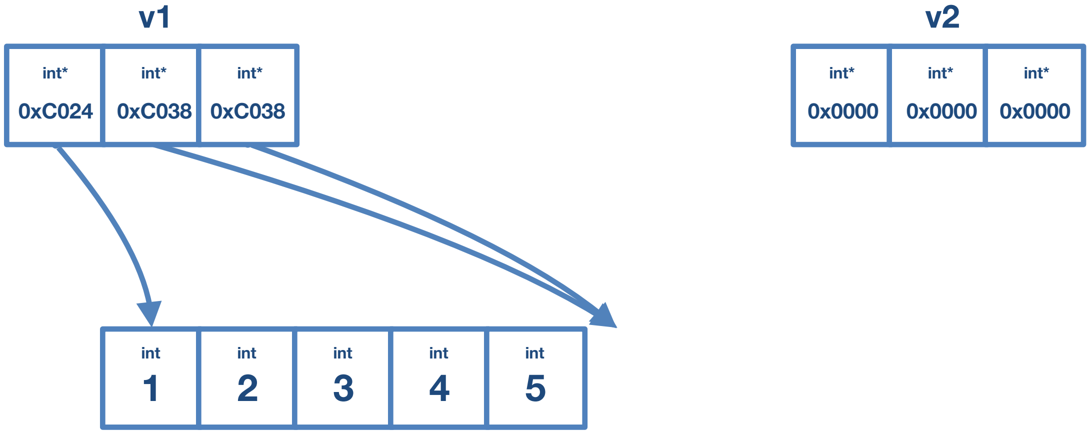
The Basics of Move Semantics

```
std::vector<int> createVector() {
   return std::vector<int>{ 1, 2, 3, 4, 5 };
}

std::vector<int> v2{};

v2 = createVector();
```



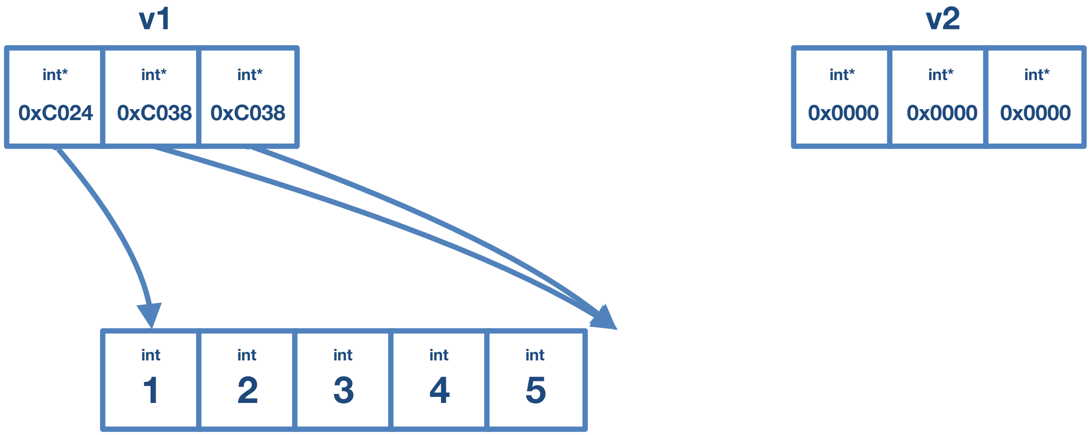
The Basics of Move Semantics

```
std::vector<int> v1{ 1, 2, 3, 4, 5 };
std::vector<int> v2{};

v2 = v1;
```



The Basics of Move Semantics

```
std::vector<int> v1{ 1, 2, 3, 4, 5 };
std::vector<int> v2{};

v2 = std::move(v1);
```



v1

int* 0xC024 | int* 0xC038 | int* 0xC038

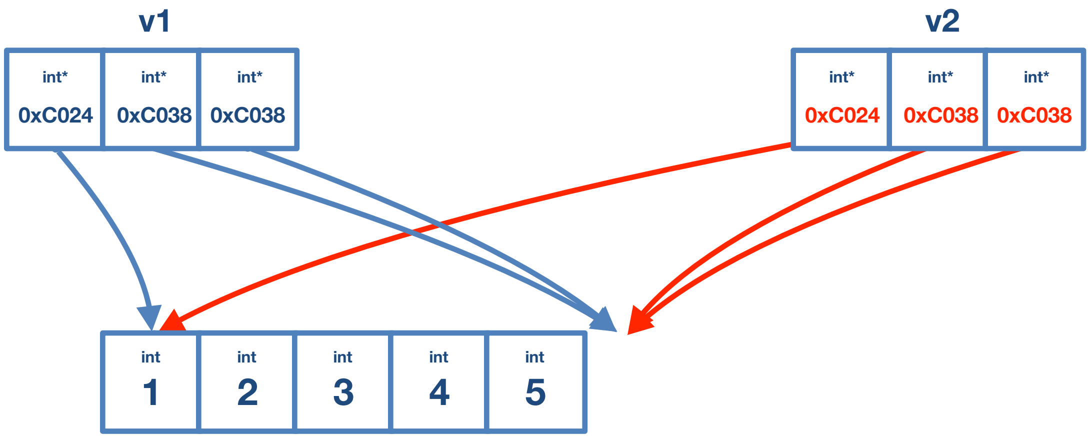
v2

int* 0x0000 | int* 0x0000 | int* 0x0000

int 1 | int 2 | int 3 | int 4 | int 5

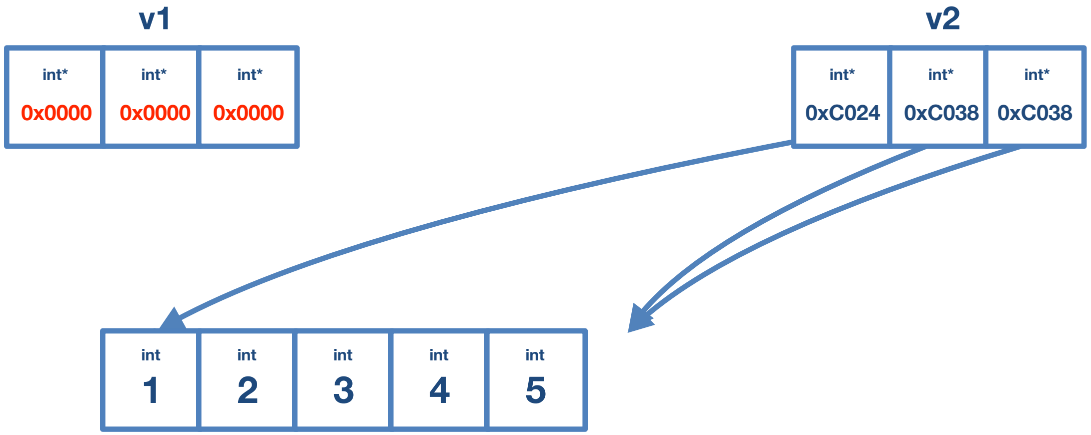
The Basics of Move Semantics

```
std::vector<int> v1{ 1, 2, 3, 4, 5 };
std::vector<int> v2{};

v2 = std::move(v1);
```



The Basics of Move Semantics

```
std::vector<int> v1{ 1, 2, 3, 4, 5 };
std::vector<int> v2{};

v2 = std::move(v1);
```



The Basics of Move Semantics

```
template< typename T
        , typename A = … >
class vector
{
 public:
   …
   // Copy assignment operator
   //   (takes an lvalue)
   vector&
     operator=(const vector& rhs);







   …
};
```

```
std::vector<int> v1{ … };

std::vector<int> createVector() {
   return std::vector<int>{ … };
}

std::vector<int> v2{};


v2 = v1;


v2 = createVector();


v2 = std::move(v1);
```

The Basics of Move Semantics

```
template< typename T
        , typename A = … >
class vector
{
 public:
   …
   // Copy assignment operator
   //   (takes an lvalue)
   vector&
     operator=(const vector& rhs);






   …
};
```

```
std::vector<int> v1{ … };

std::vector<int> createVector() {
   return std::vector<int>{ … };
}

std::vector<int> v2{};


v2 = v1;   // Lvalue


v2 = createVector();


v2 = std::move(v1);
```

Lvalues and Rvalues

```
l = r;
```

Lvalues and Rvalues

Lvalue → `l = r;`

Lvalues and Rvalues

`l = r;` ← **Rvalue**

Lvalues and Rvalues

~~l = r;~~

```
std::string s{};

s + s = s;
```

Lvalues and Rvalues

~~l = r;~~

```
std::string s{};

s + s = s;
```

Lvalue

Lvalues and Rvalues

~~`l = r;`~~

```
std::string s{};

s + s = s;
```

Rvalue

The Basics of Move Semantics

```
template< typename T
        , typename A = … >
class vector
{
 public:
   …
   // Copy assignment operator
   //   (takes an lvalue)
   vector&
     operator=(const vector& rhs);




   …
};
```

```
std::vector<int> v1{ … };

std::vector<int> createVector() {
   return std::vector<int>{ … };
}

std::vector<int> v2{};


v2 = v1;   // Lvalue


v2 = createVector();


v2 = std::move(v1);
```

The Basics of Move Semantics

```
template< typename T
        , typename A = … >
class vector
{
 public:
   …
   // Copy assignment operator
   //   (takes an lvalue)
   vector&
     operator=(const vector& rhs);



   …
};
```

```
std::vector<int> v1{ … };

std::vector<int> createVector() {
   return std::vector<int>{ … };
}

std::vector<int> v2{};


v2 = v1;


v2 = createVector();   // Rvalue


v2 = std::move(v1);
```

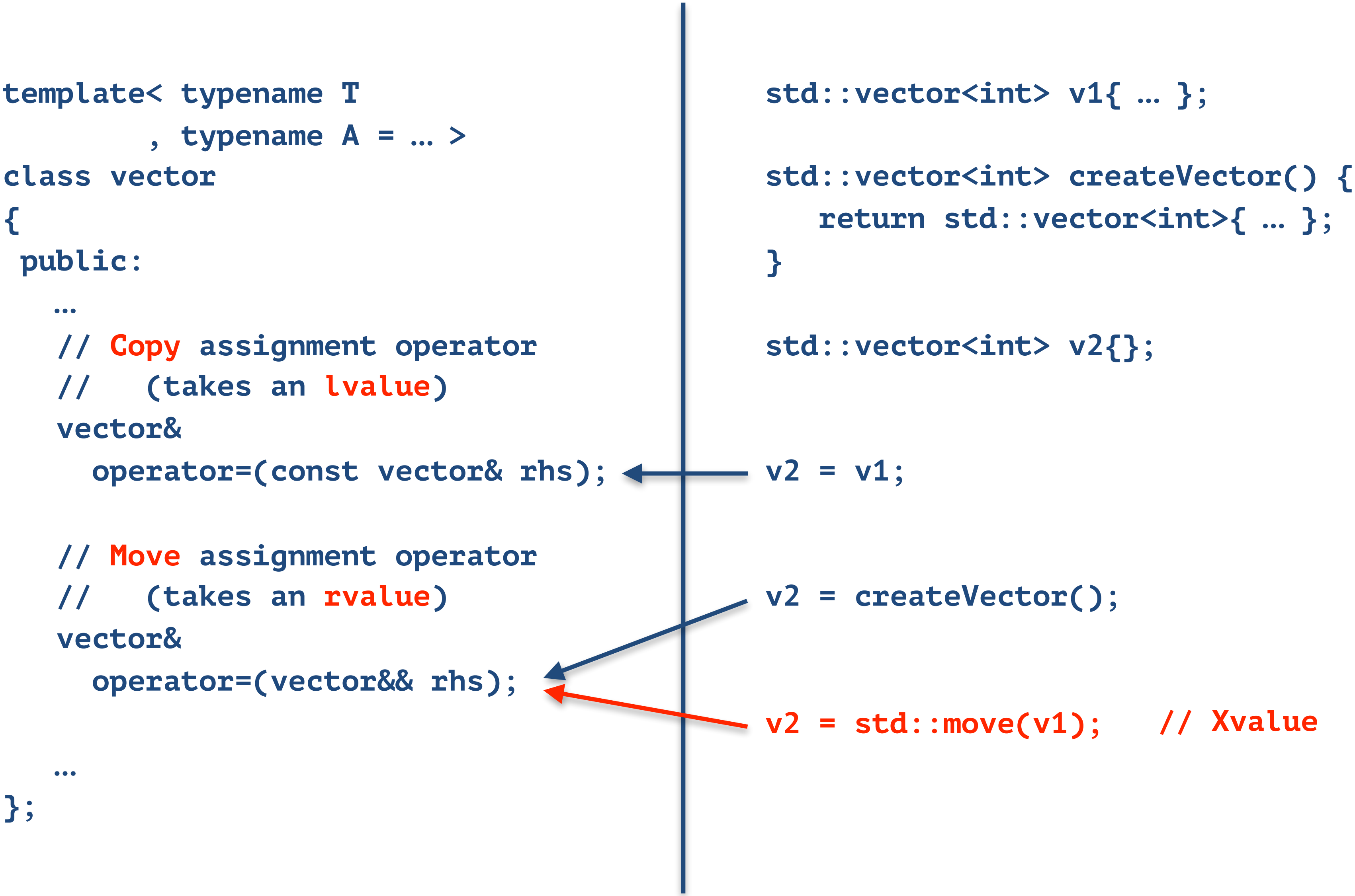
(pre C++11)

The Basics of Move Semantics

```
template< typename T
        , typename A = … >
class vector
{
 public:
   …
   // Copy assignment operator
   //   (takes an lvalue)
   vector&
     operator=(const vector& rhs);

   // Move assignment operator
   //   (takes an rvalue)
   vector&
     operator=(vector&& rhs);

   …
};
```

```
std::vector<int> v1{ … };

std::vector<int> createVector() {
   return std::vector<int>{ … };
}

std::vector<int> v2{};


v2 = v1;


v2 = createVector();   // Rvalue


v2 = std::move(v1);
```

The Basics of Move Semantics



```
template< typename T
        , typename A = … >
class vector
{
 public:
   …
   // Copy assignment operator
   //   (takes an lvalue)
   vector&
     operator=(const vector& rhs);

   // Move assignment operator
   //   (takes an rvalue)
   vector&
     operator=(vector&& rhs);

   …
};
```

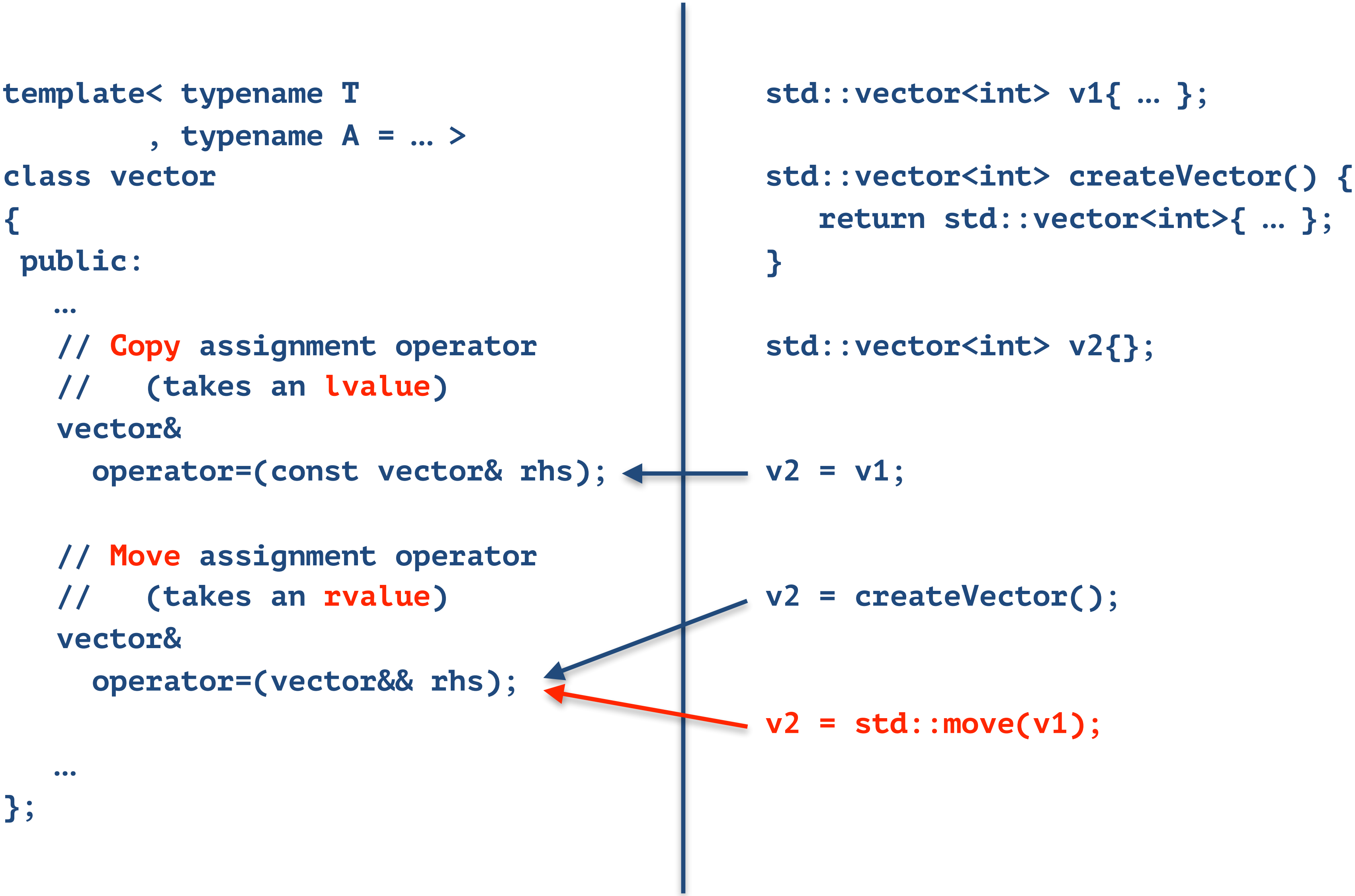
```
std::vector<int> v1{ … };

std::vector<int> createVector() {
   return std::vector<int>{ … };
}

std::vector<int> v2{};


v2 = v1;


v2 = createVector();


v2 = std::move(v1);   // Xvalue
```

std::move

- std::move unconditionally casts its input into an rvalue reference
- std::move does not move anything

```
template< typename T >
std::remove_reference_t<T>&&
   move( T&& t ) noexcept
{
   return static_cast<std::remove_reference_t<T>&&>( t );
}
```

The Basics of Move Semantics



```
template< typename T
        , typename A = … >
class vector
{
 public:
   …
   // Copy assignment operator
   //   (takes an lvalue)
   vector&
     operator=(const vector& rhs);

   // Move assignment operator
   //   (takes an rvalue)
   vector&
     operator=(vector&& rhs);

   …
};
```

```
std::vector<int> v1{ … };

std::vector<int> createVector() {
   return std::vector<int>{ … };
}

std::vector<int> v2{};


v2 = v1;


v2 = createVector();


v2 = std::move(v1);
```

Summary

- Containers in C++ employ value semantics
- In pre-C++11 this leads to unnecessary (expensive) copy operations
- C++11 introduces rvalue references to distinguish between lvalues and rvalues
- rvalue references represent modifiable objects that are no longer needed

The New Special Member Functions

The New Special Member Functions

```
class Widget {
 private:
   int i{ 0 };
   std::string s{};
   unique_ptr<int> pi{};

 public:
   // …

   // …
};
```

The New Special Member Functions

```
class Widget {
 private:
   int i{ 0 };
   std::string s{};
   unique_ptr<int> pi{};

 public:
   // …
   // Move constructor
   Widget( Widget&& w ) = default;

   // Move assignment operator
   Widget& operator=( Widget&& w ) = default;






   // …
};
```

The Move Constructor

Core Guideline C.20: If you can avoid defining default operations, do

Note This is known as the “rule of zero”.

The New Special Member Functions

```
class Widget {
 private:
   int i{ 0 };
   std::string s{};
   unique_ptr<int> pi{};

 public:
   // …
   // Move constructor
   Widget( Widget&& w ) = default;

   // Move assignment operator
   Widget& operator=( Widget&& w ) = default;




   // …
};
```

The New Special Member Functions

```
class Widget {
 private:
   int i{ 0 };
   std::string s{};
   int* pi{ nullptr };

 public:
   // …
   // Move constructor
   Widget( Widget&& w ) = default;

   // Move assignment operator
   Widget& operator=( Widget&& w ) = default;





   // …
};
```

The New Special Member Functions

```
class Widget {
 private:
   int i{ 0 };
   std::string s{};
   int* pi{ nullptr };

 public:
   // …
   // Move constructor
   Widget( Widget&& w );

   // Move assignment operator
   Widget& operator=( Widget&& w );

   // …
};
```

The Move Constructor

```
class Widget {
 private:
   int i{ 0 };
   std::string s{};
   int* pi{ nullptr };

 public:
   // …
   // Move constructor
   Widget( Widget&& w )



   {


   }


   // …
};
```

The Goal

- Transfer the content of `w` into `this`
- Leave `w` in a valid but undefined state

The Move Constructor

```
class Widget {
 private:
   int i{ 0 };
   std::string s{};
   int* pi{ nullptr };

 public:
   // …
   // Move constructor
   Widget( Widget&& w )
      : i ( w.i )


   {


   }


   // …
};
```

The Goal

- Transfer the content of `w` into `this`
- Leave `w` in a valid but undefined state

The Move Constructor

```
class Widget {
 private:
   int i{ 0 };
   std::string s{};
   int* pi{ nullptr };

 public:
   // …
   // Move constructor
   Widget( Widget&& w )
      : i ( w.i )
      , s ( w.s )

   {


   }


   // …
};
```

The Goal

- Transfer the content of `w` into `this`
- Leave `w` in a valid but undefined state

The Move Constructor

```
class Widget {
 private:
   int i{ 0 };
   std::string s{};
   int* pi{ nullptr };

 public:
   // …
   // Move constructor
   Widget( Widget&& w )
      : i ( w.i )
      , s ( w.s )

   {


   }


   // …
};
```

⚠

The Goal

- ○ Transfer the content of `w` into `this`
- ○ Leave `w` in a valid but undefined state

The Move Constructor

```
class Widget {
 private:
   int i{ 0 };
   std::string s{};
   int* pi{ nullptr };

 public:
   // …
   // Move constructor
   Widget( Widget&& w )
      : i ( w.i )
      , s ( w.s )  // Copies the string, w is an lvalue!!!

   {


   }


   // …
};
```

The Goal

- ○ Transfer the content of `w` into `this`
- ○ Leave `w` in a valid but undefined state

The Move Constructor

```
class Widget {
 private:
   int i{ 0 };
   std::string s{};
   int* pi{ nullptr };

 public:
   // …
   // Move constructor
   Widget( Widget&& w )
      : i ( w.i )
      , s ( std::move(w.s) )  // Moves the string

   {


   }


   // …
};
```

The Goal

- Transfer the content of w into this
- Leave w in a valid but undefined state

The Move Constructor

```
class Widget {
 private:
   int i{ 0 };
   std::string s{};
   int* pi{ nullptr };

 public:
   // …
   // Move constructor
   Widget( Widget&& w )
      : i ( std::move(w.i) )  // Correct, but no speed up
      , s ( std::move(w.s) )

   {


   }


   // …
};
```

The Goal

○ Transfer the content of `w` into `this`

○ Leave `w` in a valid but undefined state

The Move Constructor

```
class Widget {
 private:
   int i{ 0 };
   std::string s{};
   int* pi{ nullptr };

 public:
   // …
   // Move constructor
   Widget( Widget&& w )
      : i ( std::move(w.i) )
      , s ( std::move(w.s) )
      , pi( std::move(w.pi) )
   {


   }


   // …
};
```

The Goal

- Transfer the content of `w` into `this`
- Leave `w` in a valid but undefined state

The Move Constructor

```
class Widget {
 private:
   int i{ 0 };
   std::string s{};
   int* pi{ nullptr };

 public:
   // …
   // Move constructor
   Widget( Widget&& w )
      : i ( std::move(w.i) )
      , s ( std::move(w.s) )
      , pi( std::move(w.pi) )
   {
      w.pi = nullptr;

   }


   // …
};
```

The Goal

✅ Transfer the content of `w` into `this`

○ Leave `w` in a valid but undefined state

The Move Constructor

```
class Widget {
 private:
   int i{ 0 };
   std::string s{};
   int* pi{ nullptr };

 public:
   // …
   // Move constructor
   Widget( Widget&& w )
      : i ( std::move(w.i) )
      , s ( std::move(w.s) )
      , pi( std::move(w.pi) )
   {
      w.pi = nullptr;

   }


   // …
};
```

The Goal

✅ Transfer the content of `w` into `this`

○ Leave `w` in a valid but undefined state

The Move Constructor

```
class Widget {
 private:
   int i{ 0 };
   std::string s{};
   int* pi{ nullptr };

 public:
   // …
   // Move constructor
   Widget( Widget&& w )
      : i ( std::move(w.i) )
      , s ( std::move(w.s) )
      , pi( std::exchange(w.pi,nullptr) )
   {


   }


   // …
};
```

The Goal

✅ Transfer the content of `w` into `this`

○ Leave `w` in a valid but undefined state

The Move Constructor

```
class Widget {
 private:
   int i{ 0 };
   std::string s{};
   int* pi{ nullptr };

 public:
   // …
   // Move constructor
   Widget( Widget&& w )
      : i ( std::move(w.i) )
      , s ( std::move(w.s) )
      , pi( std::move(w.pi) )
   {
      w.pi = nullptr;

   }


   // …
};
```

The Goal

✅ Transfer the content of `w` into `this`

○ Leave `w` in a valid but undefined state

The Move Constructor

Core Guideline C.66: Make move operations `noexcept`.

The Move Constructor

```
class Widget {
 private:
   int i{ 0 };
   std::string s{};
   int* pi{ nullptr };

 public:
   // …
   // Move constructor
   Widget( Widget&& w ) noexcept
      : i ( std::move(w.i) )
      , s ( std::move(w.s) )
      , pi( std::move(w.pi) )
   {
      w.pi = nullptr;

   }


   // …
};
```

The Goal

✅ Transfer the content of `w` into `this`

○ Leave `w` in a valid but undefined state

Benchmarking the Move Constructor

```
int main()
{
  std::string s( "Long string that needs to be copied" );
  std::vector<Widget> v{};

  constexpr size_t N( 10000 );

  std::chrono::time_point<std::chrono::high_resolution_clock> start, end;
  start = std::chrono::high_resolution_clock::now();

  for( size_t i=0UL; i<N; ++i ) {
    Widget w{ 1, s, nullptr };
    v.push_back( std::move(w) );
  }

  end = std::chrono::high_resolution_clock::now();
  const std::chrono::duration<double> elapsedTime( end - start );
  const double seconds( elapsedTime.count() );

  std::cout << " Runtime: " << seconds << “s\n\n”;
}
```

Benchmarking the Move Constructor

```
int main()
{
  std::string s( "Long string that needs to be copied" );
  std::vector<Widget> v{};

  constexpr size_t N( 10000 );

  std::chrono::time_point<std::chrono::high_resolution_clock> start, end;
  start = std::chrono::high_resolution_clock::now();

  for( size_t i=0UL; i<N; ++i ) {
    Widget w{ 1, s, nullptr };
    v.push_back( std::move(w) );
  }

  end = std::chrono::high_resolution_clock::now();
  const std::chrono::duration<double> elapsedTime( end - start );
  const double seconds( elapsedTime.count() );

  std::cout << " Runtime: " << seconds << “s\n\n”;
}
```

Benchmarking the Move Constructor

```
int main()
{
  std::string s( "Long string that needs to be copied" );
  std::vector<Widget> v{};

  constexpr size_t N( 10000 );

  std::chrono::time_point<std::chrono::high_resolution_clock> start, end;
  start = std::chrono::high_resolution_clock::now();

  for( size_t i=0UL; i<N; ++i ) {
    Widget w{ 1, s, nullptr };
    v.push_back( std::move(w) );
  }

  end = std::chrono::high_resolution_clock::now();
  const std::chrono::duration<double> elapsedTime( end - start );
  const double seconds( elapsedTime.count() );

  std::cout << " Runtime: " << seconds << “s\n\n”;
}
```

Benchmarking the Move Constructor

```
int main()
{
  std::string s( "Long string that needs to be copied" );
  std::vector<Widget> v{};

  constexpr size_t N( 10000 );

  std::chrono::time_point<std::chrono::high_resolution_clock> start, end;
  start = std::chrono::high_resolution_clock::now();

  for( size_t i=0UL; i<N; ++i ) {
    Widget w{ 1, s, nullptr };
    v.push_back( std::move(w) );
  }

  end = std::chrono::high_resolution_clock::now();
  const std::chrono::duration<double> elapsedTime( end - start );
  const double seconds( elapsedTime.count() );

  std::cout << " Runtime: " << seconds << “s\n\n”;
}
```

Benchmarking the Move Constructor

```
int main()
{
  std::string s( "Long string that needs to be copied" );
  std::vector<Widget> v{};

  constexpr size_t N( 10000 );

  std::chrono::time_point<std::chrono::high_resolution_clock> start, end;
  start = std::chrono::high_resolution_clock::now();

  for( size_t i=0UL; i<N; ++i ) {
    Widget w{ 1, s, nullptr };
    v.push_back( std::move(w) );
  }

  end = std::chrono::high_resolution_clock::now();
  const std::chrono::duration<double> elapsedTime( end - start );
  const double seconds( elapsedTime.count() );

  std::cout << " Runtime: " << seconds << “s\n\n”;
}
```

Runtime without noexcept (Clang & GCC): 0.005s

Benchmarking the Move Constructor

```
int main()
{
  std::string s( "Long string that needs to be copied" );
  std::vector<Widget> v{};

  constexpr size_t N( 10000 );

  std::chrono::time_point<std::chrono::high_resolution_clock> start, end;
  start = std::chrono::high_resolution_clock::now();

  for( size_t i=0UL; i<N; ++i ) {
    Widget w{ 1, s, nullptr };
    v.push_back( std::move(w) );
  }

  end = std::chrono::high_resolution_clock::now();
  const std::chrono::duration<double> elapsedTime( end - start );
  const double seconds( elapsedTime.count() );

  std::cout << " Runtime: " << seconds << “s\n\n”;
}
```

Runtime without noexcept (Clang & GCC): 0.005s

Runtime with noexcept (Clang & GCC): 0.002s (-60%)

The Move Constructor

```
class Widget {
 private:
   int i{ 0 };
   std::string s{};
   int* pi{ nullptr };

 public:
   // …
   // Move constructor
   Widget( Widget&& w ) noexcept
      : i ( std::move(w.i) )
      , s ( std::move(w.s) )
      , pi( std::move(w.pi) )
   {
      w.pi = nullptr;

   }


   // …
};
```

The Goal

✅ Transfer the content of `w` into `this`

○ Leave `w` in a valid but undefined state

The Move Constructor

```
class Widget {
 private:
   int i{ 0 };
   std::string s{};
   int* pi{ nullptr };

 public:
   // …
   // Move constructor
   Widget( Widget&& w ) noexcept
      : i ( std::move(w.i) )
      , s ( std::move(w.s) )
      , pi( std::move(w.pi) )
   {
      w.pi = nullptr;

   }


   // …
};
```

The Goal

✅ Transfer the content of `w` into `this`

○ Leave `w` in a valid but undefined state

What about w.i?

The Move Constructor

Core Guideline C.64: A move operation should move and leave its source in a valid state

Note Ideally, that moved-from should be the default value of the type. Ensure that unless there is an exceptionally good reason not to. However, not all types have a default value and for some types establishing the default value can be expensive. The standard requires only that the moved-from object can be destroyed. Often, we can easily and cheaply do better: The standard library assumes that it is possible to assign to a moved-from object. Always leave the moved-from object in some (necessarily specified) valid state.

The Move Constructor

Core Guideline C.64: A move operation should move and leave its source in a valid state

Note Ideally, that moved-from should be the default value of the type. Ensure that unless there is an exceptionally good reason not to. However, not all types have a default value and for some types establishing the default value can be expensive. The standard requires only that the moved-from object can be destroyed. Often, we can easily and cheaply do better: The standard library assumes that it is possible to assign to a moved-from object. Always leave the moved-from object in some (necessarily specified) valid state.

The Move Constructor

```
class Widget {
 private:
   int i{ 0 };
   std::string s{};
   int* pi{ nullptr };

 public:
   // …
   // Move constructor
   Widget( Widget&& w ) noexcept
      : i ( std::move(w.i) )
      , s ( std::move(w.s) )
      , pi( std::move(w.pi) )
   {
      w.pi = nullptr;
      w.i = 0;  // Purely optional, not done by default!
   }


   // …
};
```

The Goal

✅ Transfer the content of `w` into `this`

✅ Leave `w` in a valid but undefined state

The Move Constructor

```
class Widget {
 private:
   int i{ 0 };
   std::string s{};
   int* pi{ nullptr };

 public:
   // …
   // Move constructor
   Widget( Widget&& w ) noexcept
      : i ( std::move(w.i) )
      , s ( std::move(w.s) )
      , pi( std::move(w.pi) )
   {
      w.pi = nullptr;

   }


   // …
};
```

The Goal

- ✅ Transfer the content of `w` into `this`
- ✅ Leave `w` in a valid but undefined state

The Move Constructor

```
class Widget {
 private:
   int i{ 0 };
   std::string s{};
   int* pi{ nullptr };

 public:
   // …
   // Move constructor
   Widget( Widget&& w ) noexcept
      : i ( std::move(w.i) )
      , s ( std::move(w.s) )
      , pi( std::move(w.pi) )
   {
      w.pi = nullptr;

   }


   // …
};
```

The Goal

✅ Transfer the content of `w` into `this`

✅ Leave `w` in a valid but undefined state

Phase 1: Member-wise move

Phase 2: Reset

The Move Constructor

```
class Widget {
 private:
   int i{ 0 };
   std::string s{};
   unique_ptr<int> pi{};

 public:
   // …
   // Move constructor
   Widget( Widget&& w ) noexcept
      : i ( std::move(w.i) )
      , s ( std::move(w.s) )
      , pi( std::move(w.pi) )
   {
      w.pi = nullptr;

   }


   // …
};
```

The Goal

✅ Transfer the content of `w` into `this`

✅ Leave `w` in a valid but undefined state

Phase 1: Member-wise move

Phase 2: Reset

The Move Constructor

```
class Widget {
 private:
   int i{ 0 };
   std::string s{};
   unique_ptr<int> pi{};

 public:
   // …
   // Move constructor
   Widget( Widget&& w ) noexcept
      : i ( std::move(w.i) )
      , s ( std::move(w.s) )
      , pi( std::move(w.pi) )
   {


   }


   // …
};
```

The Goal

✅ Transfer the content of `w` into `this`

✅ Leave `w` in a valid but undefined state

Phase 1: Member-wise move

The Move Constructor

```
class Widget {
 private:
   int i{ 0 };
   std::string s{};
   unique_ptr<int> pi{};

 public:
   // …
   // Move constructor
   Widget( Widget&& ) = default;  // Note: also noexcept!










   // …
};
```

The Goal

✅ Transfer the content of `w` into `this`

✅ Leave `w` in a valid but undefined state

The Move Constructor

```
class Widget {
 private:
   int i{ 0 };
   std::string s{};
   int* pi{ nullptr };

 public:
   // …
   // Move constructor
   Widget( Widget&& w ) noexcept
      : i ( std::move(w.i) )
      , s ( std::move(w.s) )
      , pi( std::move(w.pi) )
   {
      w.pi = nullptr;

   }


   // …
};
```

The Goal

✅ Transfer the content of `w` into `this`

✅ Leave `w` in a valid but undefined state

The Move Constructor

"… I think the most important take-away is that programmers should be leery of following patterns without thought. …" (Howard Hinnant)

The Move Assignment Operator

```
class Widget {
 private:
   int i{ 0 };
   std::string s{};
   int* pi{ nullptr };

 public:
   // …
   // Move assignment operator
   Widget& operator=( Widget&& w )
   {







      return *this;
   }
   // …
};
```

The Goal

- ○ Clean up all visible resources
- ○ Transfer the content of `w` into `this`
- ○ Leave `w` in a valid but undefined state

The Move Assignment Operator

```
class Widget {
 private:
   int i{ 0 };
   std::string s{};
   int* pi{ nullptr };

 public:
   // …
   // Move assignment operator
   Widget& operator=( Widget&& w )
   {

      i  = std::move(w.i);




      return *this;
   }
   // …
};
```

The Goal

○ Clean up all visible resources

○ Transfer the content of `w` into `this`

○ Leave `w` in a valid but undefined state

The Move Assignment Operator

```
class Widget {
 private:
   int i{ 0 };
   std::string s{};
   int* pi{ nullptr };

 public:
   // …
   // Move assignment operator
   Widget& operator=( Widget&& w )
   {

      i  = std::move(w.i);
      s  = std::move(w.s);



      return *this;
   }
   // …
};
```

The Goal

- Clean up all visible resources
- Transfer the content of `w` into `this`
- Leave `w` in a valid but undefined state

The Move Assignment Operator

```
class Widget {
 private:
   int i{ 0 };
   std::string s{};
   int* pi{ nullptr };

 public:
   // …
   // Move assignment operator
   Widget& operator=( Widget&& w )
   {

      i  = std::move(w.i);
      s  = std::move(w.s);
      pi = std::move(w.pi);


      return *this;
   }
   // …
};
```

The Goal

- Clean up all visible resources
- Transfer the content of `w` into `this`
- Leave `w` in a valid but undefined state

The Move Assignment Operator

```
class Widget {
 private:
   int i{ 0 };
   std::string s{};
   int* pi{ nullptr };

 public:
   // …
   // Move assignment operator
   Widget& operator=( Widget&& w )
   {
      delete pi;
      i  = std::move(w.i);
      s  = std::move(w.s);
      pi = std::move(w.pi);


      return *this;
   }
   // …
};
```

The Goal

- ✅ Clean up all visible resources
- ○ Transfer the content of `w` into `this`
- ○ Leave `w` in a valid but undefined state

The Move Assignment Operator

```
class Widget {
 private:
   int i{ 0 };
   std::string s{};
   int* pi{ nullptr };

 public:
   // …
   // Move assignment operator
   Widget& operator=( Widget&& w )
   {
      delete pi;
      i  = std::move(w.i);
      s  = std::move(w.s);
      pi = std::move(w.pi);
      w.pi = nullptr;

      return *this;
   }
   // …
};
```

The Goal

✅ Clean up all visible resources

○ Transfer the content of `w` into `this`

○ Leave `w` in a valid but undefined state

The Move Assignment Operator

```
class Widget {
 private:
   int i{ 0 };
   std::string s{};
   int* pi{ nullptr };

 public:
   // …
   // Move assignment operator
   Widget& operator=( Widget&& w )
   {
      delete pi;
      i  = std::move(w.i);
      s  = std::move(w.s);
      pi = std::move(w.pi);
      w.pi = nullptr;

      return *this;
   }
   // …
};
```

The Goal

- ✅ Clean up all visible resources
- ✅ Transfer the content of `w` into `this`
- ○ Leave `w` in a valid but undefined state

The Move Assignment Operator

```
class Widget {
 private:
   int i{ 0 };
   std::string s{};
   int* pi{ nullptr };

 public:
   // …
   // Move assignment operator
   Widget& operator=( Widget&& w )
   {
      delete pi;
      i  = std::move(w.i);
      s  = std::move(w.s);
      pi = std::exchange(w.pi,nullptr);


      return *this;
   }
   // …
};
```

The Goal

✅ Clean up all visible resources

✅ Transfer the content of `w` into `this`

○ Leave `w` in a valid but undefined state

The Move Assignment Operator

```
class Widget {
 private:
   int i{ 0 };
   std::string s{};
   int* pi{ nullptr };

 public:
   // …
   // Move assignment operator
   Widget& operator=( Widget&& w )
   {
      delete pi;
      i  = std::move(w.i);
      s  = std::move(w.s);
      pi = std::move(w.pi);
      w.pi = nullptr;

      return *this;
   }
   // …
};
```

The Goal

✅ Clean up all visible resources

✅ Transfer the content of `w` into `this`

○ Leave `w` in a valid but undefined state

The Move Assignment Operator

```
class Widget {
 private:
   int i{ 0 };
   std::string s{};
   int* pi{ nullptr };

 public:
   // …
   // Move assignment operator
   Widget& operator=( Widget&& w )
   {
      delete pi;
      i  = std::move(w.i);
      s  = std::move(w.s);
      pi = std::move(w.pi);
      w.pi = nullptr;

      return *this;
   }
   // …
};
```

The Goal

✅ Clean up all visible resources

✅ Transfer the content of w into this

○ Leave w in a valid but undefined state

The Move Assignment Operator

```
class Widget {
 private:
   int i{ 0 };
   std::string s{};
   int* pi{ nullptr };

 public:
   // …
   // Move assignment operator
   Widget& operator=( Widget&& w )
   {

      i  = std::move(w.i);
      s  = std::move(w.s);
      std::swap(pi,w.pi);   // Less deterministic


      return *this;
   }
   // …
};
```

The Goal

✅ Clean up all visible resources

✅ Transfer the content of w into this

○ Leave w in a valid but undefined state

The Move Assignment Operator

```
class Widget {
 private:
   int i{ 0 };
   std::string s{};
   int* pi{ nullptr };

 public:
   // …
   // Move assignment operator
   Widget& operator=( Widget&& w )
   {
      delete pi;
      i  = std::move(w.i);
      s  = std::move(w.s);
      pi = std::move(w.pi);
      w.pi = nullptr;

      return *this;
   }
   // …
};
```

The Goal

- ✅ Clean up all visible resources
- ✅ Transfer the content of `w` into `this`
- ○ Leave `w` in a valid but undefined state

The Move Assignment Operator

```
class Widget {
 private:
   int i{ 0 };
   std::string s{};
   int* pi{ nullptr };

 public:
   // …
   // Move assignment operator
   Widget& operator=( Widget&& w ) noexcept
   {
      delete pi;
      i  = std::move(w.i);
      s  = std::move(w.s);
      pi = std::move(w.pi);
      w.pi = nullptr;

      return *this;
   }
   // …
};
```

The Goal

- ✅ Clean up all visible resources
- ✅ Transfer the content of `w` into `this`
- ○ Leave `w` in a valid but undefined state

The Move Assignment Operator

```
class Widget {
 private:
   int i{ 0 };
   std::string s{};
   int* pi{ nullptr };

 public:
   // …
   // Move assignment operator
   Widget& operator=( Widget&& w ) noexcept
   {
      delete pi;
      i  = std::move(w.i);
      s  = std::move(w.s);
      pi = std::move(w.pi);
      w.pi = nullptr;
      w.i  = 0;  // Purely optional, not done by default!
      return *this;
   }
   // …
};
```

The Goal

✅ Clean up all visible resources

✅ Transfer the content of `w` into `this`

✅ Leave `w` in a valid but undefined state

The Move Assignment Operator

```
class Widget {
 private:
   int i{ 0 };
   std::string s{};
   int* pi{ nullptr };

 public:
   // …
   // Move assignment operator
   Widget& operator=( Widget&& w ) noexcept
   {
      delete pi;
      i  = std::move(w.i);
      s  = std::move(w.s);
      pi = std::move(w.pi);
      w.pi = nullptr;

      return *this;
   }
   // …
};
```

The Goal

✅ Clean up all visible resources

✅ Transfer the content of `w` into `this`

✅ Leave `w` in a valid but undefined state

The Move Assignment Operator

```
class Widget {
 private:
   int i{ 0 };
   std::string s{};
   int* pi{ nullptr };

 public:
   // …
   // Move assignment operator
   Widget& operator=( Widget&& w ) noexcept
   {
      delete pi;
      i  = std::move(w.i);
      s  = std::move(w.s);
      pi = std::move(w.pi);
      w.pi = nullptr;

      return *this;
   }
   // …
};
```

The Goal

✅ Clean up all visible resources

✅ Transfer the content of `w` into `this`

✅ Leave `w` in a valid but undefined state

Phase 1: Cleanup

Phase 2: Member-wise move

Phase 3: Reset

The Move Assignment Operator

```
class Widget {
 private:
   int i{ 0 };
   std::string s{};
   unique_ptr<int> pi{};

 public:
   // …
   // Move assignment operator
   Widget& operator=( Widget&& w ) noexcept
   {
      delete pi;
      i  = std::move(w.i);
      s  = std::move(w.s);
      pi = std::move(w.pi);
      w.pi = nullptr;

      return *this;
   }
   // …
};
```

The Goal

✅ Clean up all visible resources

✅ Transfer the content of `w` into `this`

✅ Leave `w` in a valid but undefined state

Phase 1: Cleanup

Phase 2: Member-wise move

Phase 3: Reset

The Move Assignment Operator

```
class Widget {
 private:
   int i{ 0 };
   std::string s{};
   unique_ptr<int> pi{};

 public:
   // …
   // Move assignment operator
   Widget& operator=( Widget&& w ) noexcept
   {

      i  = std::move(w.i);
      s  = std::move(w.s);
      pi = std::move(w.pi);


      return *this;
   }
   // …
};
```

The Goal

✅ Clean up all visible resources

✅ Transfer the content of `w` into `this`

✅ Leave `w` in a valid but undefined state

Phase 2: Member-wise move

The Move Assignment Operator

```
class Widget {
 private:
   int i{ 0 };
   std::string s{};
   unique_ptr<int> pi{};

 public:
   // …
   // Move assignment operator
   Widget& operator=( Widget&& w ) = default;
                                        // Note: also noexcept!

   // …
};
```

The Goal

✅ Clean up all visible resources

✅ Transfer the content of `w` into `this`

✅ Leave `w` in a valid but undefined state

The Move Assignment Operator

```
class Widget {
 private:
   int i{ 0 };
   std::string s{};
   int* pi{ nullptr };

 public:
   // …
   // Copy+move assignment operator
   Widget& operator=( Widget w )
   {
      swap( w );
      return *this;
   }

   void swap( Widget& w )
   {
      // …
   }
   // …
};
```

The Goal

✅ Clean up all visible resources

✅ Transfer the content of `w` into `this`

✅ Leave `w` in a valid but undefined state

The New Special Member Functions

- The default move operations are generated
  if no copy operation or destructor is user-defined
- The default copy operations are generated
  if no move operation is user-defined
- Note: =default and =delete count as user-defined!

```
class X {
 public:
   // …

   virtual ~X() = default;






   // …
};
```

The New Special Member Functions

- The default move operations are generated
  if no copy operation or destructor is user-defined
- The default copy operations are generated
  if no move operation is user-defined
- Note: `=default` and `=delete` count as user-defined!

```
class X {
 public:
   // …

   virtual ~X() = default;

   X( X&& ) = default;
   X& operator=( X&& ) = default;



   // …
};
```

The New Special Member Functions

- The default move operations are generated

  if no copy operation or destructor is user-defined
- The default copy operations are generated

  if no move operation is user-defined
- Note: `=default` and `=delete` count as user-defined!

```
class X {
 public:
   // …

   virtual ~X() = default;

   X( X&& ) = default;
   X& operator=( X&& ) = default;

   X( X const& ) = default;
   X& operator=( X const& ) = default;

   // …
};
```

The New Special Member Functions

Core Guideline C.21: If you define or =delete any default operation, define or =delete them all

Note This is known as the “rule of five” or “rule of six”, depending on whether you count the default constructor.

The New Special Member Functions

- The default move operations are generated

  if no copy operation or destructor is user-defined

- The default copy operations are generated

  if no move operation is user-defined

- Note: `=default` and `=delete` count as user-defined!

```
class X {
 public:
   // …

   virtual ~X() = default;

   X( X&& ) = default;
   X& operator=( X&& ) = default;

   X( X const& ) = default;
   X& operator=( X const& ) = default;

   // …
};
```

Parameter Conventions

Parameter Conventions

Core Guideline C.15: Prefer simple and conventional ways of passing information

Parameter Conventions

	Cheap or impossible to copy (e.g., int, unique_ptr)	**Cheap to move** (e.g., vector<T>, string) or **Moderate cost to move** (e.g., array<vector>, BigPOD) or **Don‘t know** (e.g., unfamilar type, template)	**Expensive to move** (e.g., BigPOD[], array<BigPOD>)
Out	`X f()`		`f( X& )`
In/Out	`f( X& )`		
In	`f( X )`	`f( const X& )`	

In & retain copy		`f( const X& ) +`
		`f( X&& ) & move`
			`f( const X& )`

In & move from	`f( X&& )`		

Parameter Conventions

<table>
<tr><th></th><th><b>Cheap or impossible to copy</b>
(e.g., int, unique_ptr)</th><th><b>Cheap to move</b> (e.g., vector<T>, string) or <b>Moderate cost to move</b> (e.g., array<vector>, BigPOD) or <b>Don‘t know</b> (e.g., unfamilar type, template)</th><th><b>Expensive to move</b> (e.g., BigPOD[], array<BigPOD&gt;)</th></tr>
<tr><td><b>Out</b></td><td colspan="2"><code>X f()</code></td><td><code>f(X&amp;)</code></td></tr>
<tr><td><b>In/Out</b></td><td colspan="3"><code>f(X&amp;)</code></td></tr>
<tr><td><b>In</b></td><td rowspan="2"><code>f(X)</code></td><td colspan="2"><code>f(const X&amp;)</code></td></tr>
<tr><td><b>In &amp; retain copy</b></td><td><code>f(X)</code></td><td><code>f(const X&amp;)</code></td></tr>
<tr><td><b>In &amp; move from</b></td><td colspan="3"><code>f(X&amp;&amp;)</code></td></tr>
</table>

klaus.iglberger@gmx.de