C++ Essentials: The Special Member Functions

2. The Special Member Functions

Klaus Iglberger
April, 15th, 2024

Content

1. The Special Member Functions
2. The Default Constructor
3. Copy Semantics
4. Copy Elision
5. Move Semantics
6. The Rule of 0/5

2.1. The Special Member Functions

The Compiler-Generated Functions

Quick Task: Name all compiler-generated functions!

```
class Widget
{
 public:
   Widget();                                  // Default constructor

   Widget( Widget const& );                   // Copy constructor

   Widget& operator=( Widget const& );        // Copy assignment operator

   ~Widget();                                 // Destructor

   Widget( Widget&& );  C++11                 // Move constructor

   Widget& operator=( Widget&& );  C++11      // Move assignment operator
};
```

The Special Member Functions

Quick Task: Name all special member functions (SMF)!

```
class Widget
{
 public:
   Widget();                                   // Default constructor

   Widget( Widget const& );                    // Copy constructor

   Widget& operator=( Widget const& );         // Copy assignment operator

   ~Widget();                                  // Destructor

   Widget( Widget&& );          C++11          // Move constructor

   Widget& operator=( Widget&& );     C++11    // Move assignment operator
};
```

2.2. The Default Constructor

The Default Constructor

The default constructor can be called without parameters. Its purpose is to default initialize the instance.

```
// User-defined default constructor
class Widget
{
 public:
   Widget();  // The default constructor
   Widget( /*value=default, …*/ );  // Also a default constructor


};

Widget w1;    // Compiler generated, ok
Widget w2{};  // Compiler generated, ok
```

The Default Constructor

The compiler generates a default constructor …

```
// Compiler-generated default constructor available
class Widget
{
 public:
   // ...



};

Widget w1;    // Compiler generated, ok
Widget w2{};  // Compiler generated, ok
```

The Default Constructor

The compiler generates a default constructor …

- if no constructor is explicitly declared, …

```
// No compiler-generated default constructor available
class Widget
{
 public:
   Widget( Widget const& ); // <- explicit declaration of the
   // ...                   //    copy ctor -> no default ctor
                            //    available

};

Widget w1;    // No default constructor, compilation failure
Widget w2{};  // No default constructor, compilation failure
```

The Default Constructor

The compiler generates a default constructor …

- if no constructor is explicitly declared, …
- all data members and base classes can be default constructed, and …

```
// No compiler-generated default constructor available
class Widget : public Base
{
 public:
   // ...
 private:
   NoDefaultCtor member_;  // Data member without default ctor

};

Widget w1;    // No default constructor, compilation failure
Widget w2{};  // No default constructor, compilation failure
```

The Default Constructor

The compiler generates a default constructor …

- if no constructor is explicitly declared, …
- all data members and base classes can be default constructed, and …
- if there are no const or reference data members in the class.

```
// No compiler-generated default constructor available
class Widget : public Base
{
 public:
   // ...
 private:
   T const member1_;  // Immutable data member
   T& member2_;       // Reference data member
};

Widget w1;    // No default constructor, compilation failure
Widget w2{};  // No default constructor, compilation failure
```

The Default Constructor

Task (2_The_Special_Member_Functions/MemberInitialization1): What is the initial value of the three data members `i`, `s`, and `pi`?

```
struct Widget
{

   int i;          // Uninitialized
   std::string s; // Default (i.e. empty string)
   int* pi;        // Uninitialized
};

int main()
{
   Widget w;       // Default initialization
}
```

The Default Constructor

The compiler generated default constructor …

- initializes all data members of class (user-defined) type …
- but not the data members of fundamental type or pointers.

```
struct Widget
{
   int i;            // Uninitialized
   std::string s;    // Default (i.e. empty string)
   int* pi;          // Uninitialized
};

int main()
{
   Widget w;         // Default initialization: Calls
}                    // the default constructor
```

Guidelines

Guideline: Remember that data members of fundamental type and pointers are by default not initialized.

The Default Constructor

Task (2_The_Special_Member_Functions/MemberInitialization2): What is the initial value of the three data members `i`, `s`, and `pi`?

```
struct Widget
{

   int i;            // Initialized to 0
   std::string s;    // Default (i.e. empty string)
   int* pi;          // Initialized to nullptr
};

int main()
{
   Widget w{};      // Value initialization
}
```

The Default Constructor

If no default constructor is declared, value initialization …

- zero-initializes the object
- and then default-initializes all non-trivial data members.

```
struct Widget
{

   int i;            // Initialized to 0
   std::string s; // Default (i.e. empty string)
   int* pi;          // Initialized to nullptr
};

int main()
{
   Widget w{};     // Value initialization: No default
}                   // ctor -> zero+default init
```

Guidelines

Guideline: Prefer to create default objects by means of an empty set of braces (value initialization).

The Default Constructor

Task (2_The_Special_Member_Functions/MemberInitialization3): What is the initial value of the three data members `i`, `s`, and `pi`?

```
struct Widget
{
   Widget() {}     // Explicit default constructor
   int i;          // Uninitialized
   std::string s;  // Default (i.e. empty string)
   int* pi;        // Uninitialized
};

int main()
{
   Widget w{};    // Value initialization
}
```

The Default Constructor

An empty default constructor …

- initializes all data members of class (user-defined) type …
- but not the data members of fundamental type or pointers.

```
struct Widget
{
   Widget() {}     // Explicit default constructor
   int i;          // Uninitialized
   std::string s;  // Default (i.e. empty string)
   int* pi;        // Uninitialized
};

int main()
{
   Widget w{};    // Value initialization: Declared
}                 // default ctor -> calls ctor
```

Guidelines

Guideline: Avoid writing an empty default constructor.

The Default Constructor

Via the default constructor, we can properly initialize all data members:

```
struct Widget
{
  Widget()
  {
    i  = 42;         // Initialize the int to 0
    s  = "CppCon";   // Initialize the string to ""
    pi = nullptr;    // Initialize the pointer to nullptr
  }


  int i;
  std::string s;
  int* pi;
};
```

The Default Constructor

Via the default constructor, we can properly initialize all data members:

```
struct Widget
{
  Widget()
  {
    i  = 42;         // Assignment, not initialization
    s  = "CppCon";   // Assignment, not initialization
    pi = nullptr;    // Assignment, not initialization
  }


  int i;
  std::string s;
  int* pi;
};
```

The Default Constructor

Via the default constructor, we can properly initialize all data members:

```
struct Widget
{
  Widget()
    : s{"CppCon"}   // Initialization of the string
                    // in the member initializer list
  {
    i  = 42;        // Assignment, not initialization
    pi = nullptr;   // Assignment, not initialization
  }

  int i;
  std::string s;
  int* pi;
};
```

The Default Constructor

Via the default constructor, we can properly initialize all data members:

```
struct Widget
{
  Widget()
    : i {42}         // Initializing to 42
    , s {"CppCon"}   // Initializing to "CppCon"
    , pi{}           // Initializing to nullptr
  {}



  int i;
  std::string s;
  int* pi;
};
```

Guidelines

Core Guideline C.47: Define and initialise member variables in the order of member declaration

Core Guideline C.49: Prefer initialization to assignment in constructors.

The Default Constructor

Let's assume that a colleague adds another constructor…

```
struct Widget
{
  Widget()
    : i {42}          // Initializing to 42
    , s {"CppCon"}  // Initializing to "CppCon"
    , pi{}            // Initializing to nullptr
  {}

  Widget( int j )
    : i {j}           // Initialization to j
  {}



  int i;
  std::string s;
  int* pi;
};
```

The Default Constructor

Let’s assume that a colleague adds another constructor…

```
struct Widget
{
  Widget()
    : i {42}           // Initializing to 42
    , s {"CppCon"}     // Initializing to "CppCon"
    , pi{}             // Initializing to nullptr
  {}

  Widget( int j )
    : i {j}            // Initialization to j
    , s {"CppCon"}     // Initialization to "CppCon"
    , pi{}             // Initialization to nullptr
  {}

  int i;
  std::string s;
  int* pi;
};
```

The Default Constructor

Let's assume that a colleague adds another constructor…

```
struct Widget
{
  Widget()
    : i {42}          // Initializing to 42
    , s {"CppCon"}   // Initializing to "CppCon"
    , pi{}            // Initializing to nullptr
  {}

  Widget( int j )
    : i {j}           // Initialization to j
    , s {"CppCon"}   // Initialization to "CppCon" (duplication)
    , pi{}            // Initialization to nullptr (duplication)
  {}

  int i;
  std::string s;
  int* pi;
};
```

The Default Constructor

Guideline: Avoid duplication to enable you to change everything in one place (the DRY principle).

Guideline: Design classes for easy change.

The Default Constructor

In order to reduce duplication, we could use delegating constructors …

```
struct Widget
{
  Widget()
    : Widget(42)  // Delegating constructor
  {}



  Widget( int j )
    : i {j}           // Initialization to j
    , s {"CppCon"}    // Initialization to "CppCon" (duplication)
    , pi{}            // Initialization to nullptr (duplication)
  {}

  int i;
  std::string s;
  int* pi;
};
```

The Default Constructor

Core Guideline C.51: Use delegating constructors to represent common actions for all constructors of a class

The Default Constructor

… or we could use in-class member initializers.

```
struct Widget
{
  Widget()
  {}

  Widget( int j )
    : i {j} // Initializing to j
  {}

  // Data members with in-class initializers
  int i{42};                        // initializing to 42
  std::string s{"CppCon"};          // initializing to "CppCon"
  int* pi{};                        // initialising to nullptr
};
```

In-class member initializers are used if the data member is not explicitly listed in the member initializer list.

The Default Constructor

… or we could use in-class member initializers.

```
struct Widget
{
  Widget() = default;


  Widget( int j )
    : i {j} // Initializing to j
  {}

  // Data members with in-class initializers
  int i{42};                        // initializing to 42
  std::string s{"CppCon"};  // initializing to "CppCon"
  int* pi{};                        // initialising to nullptr
};
```

In-class member initializers are used if the data member is not explicitly listed in the member initializer list.

Guidelines

Core Guideline C.48: Prefer in-class initializers to member initializers in constructors for constant initializers

Guideline: Prefer to initialize pointer members to nullptr with in-class member initializers.

Core Guideline C.44: Prefer default constructors to be simple and non-throwing

Uniform Initilization vs std::initializer_list

Guideline: Beware the difference between `()` and `{}` for container types (i.e. classes with a `std::initializer_list` constructor).

```
std::vector<int> v1( 3, 5 );  // Results in ( 5 5 5 )

std::vector<int> v2{ 3, 5 };  // Results in ( 3 5 )
```

2.3. Copy Semantics

The Signatures of the Copy Operations

The copy constructor can be called with one argument of the classes' type:

```
Widget( Widget const& );   // The default

~~Widget( Widget& );~~       // Possible, but very likely not
                           // reasonable

~~Widget( Widget );~~        // Not possible; recursive call
```

The same is true for the copy assignment operator:

```
Widget& operator=( Widget const& );   // The default

~~Widget& operator=( Widget& );~~       // Possible, but very likely
                                      // not reasonable

Widget& operator=( Widget );          // Reasonable; builds on the
                                      // copy constructor
```

The Signatures of the Copy Operations

The copy constructor can be called with one argument of the classes' type:

```
Widget( Widget const&, /*other_value = default, ...*/ );
```

~~`Widget( Widget&, /*other_value = default, ...*/ );`~~

~~`Widget( Widget, /*other_value = default, ...*/ );`~~

The same is true for the copy assignment operator:

```
Widget& operator=( Widget const&, /*other_value = default, ...*/ );
```

~~`Widget& operator=( Widget&, /*other_value = default, ...*/ );`~~

```
Widget& operator=( Widget, /*other_value = default, ...*/ );
```

The Copy Ctor and Copy Assignment Operator

The compiler **always** generates the copy operations …

```
// Compiler-generated copy ctor and copy assignment available
class Widget
{
 public:

   // ...


};

Widget w1{};
Widget w2( w1 );  // Compiler generated, ok
w1 = w2;          // Compiler generated, ok
```

The Copy Ctor and Copy Assignment Operator

The compiler **always** generates the copy operations …

- if they are not explicitly declared …

```
// Compiler-generated copy ctor and copy assignment not available
class Widget
{
 public:
   Widget( Widget const& );
   Widget& operator=( Widget const& );
   // ...

};

Widget w1{};
Widget w2( w1 );  // Explicitly defined, ok
w1 = w2;          // Explicitly defined, ok
```

The Copy Ctor and Copy Assignment Operator

The compiler **always** generates the copy operations …

- if they are not explicitly declared …
- if no move operation is declared … C++11

```
// Compiler-generated copy ctor and copy assignment not available
class Widget
{
 public:
   // Widget( Widget const& ) = delete;
   // Widget& operator=( Widget const& ) = delete;
   Widget( Widget&& w );

};

Widget w1{};
Widget w2( w1 );  // Compiler error: Copy constructor not available
w1 = w2;          // Compiler error: Copy assignment not available
```

The Copy Ctor and Copy Assignment Operator

The compiler **always** generates the copy operations …

- if they are not explicitly declared …
- if no move operation is declared … C++11
- if all data members and base classes can be copy constructed/assigned.

```
// Compiler-generated copy ctor and copy assignment not available
class Widget : public Base
{
 public:
   // Widget( Widget const& ) = delete;
   // Widget& operator=( Widget const& ) = delete;
 private:
   NonCopyable member_;  // Data member without copy operations
};

Widget w1{};
Widget w2( w1 );  // Compiler error: Copy constructor not available
w1 = w2;          // Compiler error: Copy assignment not available
```

The Default Implementation

```
class Widget : public Base
{
 public:
   Widget( Widget const& other )
      : Base{ other }           // The default copy constructor performs
      , i { other.i  }          // a member-wise copy construction of
      , s { other.s  }          // all bases and data members
      , pi{ other.pi }
   {}
   Widget& operator=( Widget const& other )
   {
      Base::operator=( other );   // The default copy assignment operator
      i  = other.i;               // performs a member-wise copy assignment
      s  = other.s;               // of all bases and data members
      pi = other.pi;
      return *this;
   }
   // ...

 private:                   // The three data members:
   int i;                   // - i as a representative of a fundamental type
   std::string s;           // - s as a representative of a class (user-defined) type
   int* pi{};               // - pi as representative of a possible resource
};
```

Programming Task

Task (2_The_Special_Member_Functions/ResourceOwner): Implement the copy operations of class `ResourceOwner`.

```
class ResourceOwner {
 public:
   // ...
   ResourceOwner( ResourceOwner const& );
   ResourceOwner& operator=( ResourceOwner const& );
   // ...
};
```

How to Disable Copy Operations

C++03

- Declare both the copy ctor and the copy assignment operator `private`
- Leave both operations undefined

```
class non_copyable
{
 protected:
   non_copyable() = default;

 private:
   non_copyable( non_copyable const& );
   non_copyable& operator=( non_copyable const& );
};
```

How to Disable Copy Operations

03

The NonCopyable class passes its non-copyable property on to deriving classes:

```
class Widget : private non_copyable
{
 public:
   // ...
};


Widget w1{};
Widget w2( w1 );  // Compilation error
w2 = w1;          // Compilation error
```

But note it is easily possible to reactivate copying by explicitly declaring a copy constructor and/or copy assignment operator within Widget!

How to Disable Copy Operations

C++11

- `delete` both the copy constructor and copy assignment operator
- Leave them in the `public` section

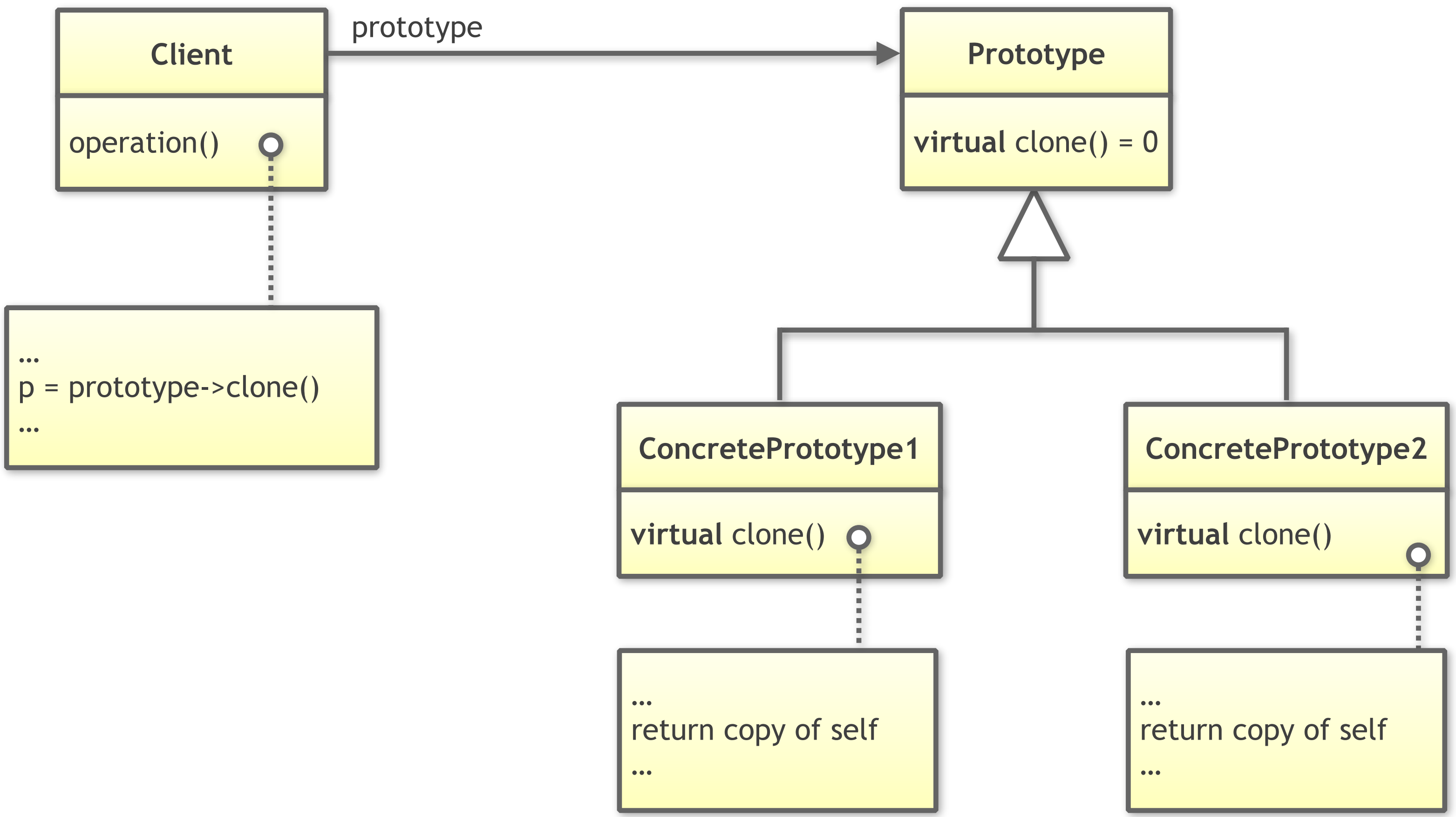
```
class Widget
{
 public:
   // ...
   Widget( Widget const& ) = delete;
   Widget & operator=( Widget const& ) = delete;
   // ...
};
```

How to Implement Virtual Copying

Use the prototype design pattern to implement virtual copying:

```
class Widget
{
 public:
   Widget( Widget const& ) = delete;
   Widget& operator=( Widget const& ) = delete;

   virtual Widget* clone() const = 0;

   ...
};
```

The Prototype Design Pattern



Guidelines

Guideline: Implement simple and intuitive copy operations and adhere to the expected semantics (deep copy rather than shallow copy, no changes in case of self-copy-assignment, …).

Guideline: Try to reduce the use of pointers!

swap(): The Secret 7. Special Member

```
class ResourceOwner
{
 public:                                    // The member function provides
                                            // access to the data members
   // ...
   void swap( ResourceOwner& other ) noexcept
   {
      using std::swap;                            // Prefer an unqualified call
      swap( m_id      , other.m_id       );       // to swap(), and make
      swap( m_name    , other.m_name     );       // std::swap() available via
      swap( m_resource, other.m_resource );       // using declaration
   }
   // ...

 private:
   int m_id{ 0 };
   std::string m_name{};
   Resource* m_resource{ nullptr };
};

void swap( ResourceOwner& a, ResourceOwner& b ) noexcept
{
   a.swap( b );                             // The free function is the primary
}                                           // customisation point
```

Guidelines

Core Guideline C.83: For value-like types, consider providing a `noexcept` `swap` function

Core Guideline C.84: A `swap` function must not fail

Core Guideline C.85: Make `swap` `noexcept`

2.4. Copy Elision

Copy Elision

The compiler is allowed to elide copies where results are “as if” copies were made. The Return Value Optimization (RVO) is one such instance:

- The caller allocates space on stack for the return value and passes the address to the callee;
- The callee constructs the result *directly* in that space.

Programming Task

Task (2_The_Special_Member_Functions/RVO1): Investigate, which of the special member functions are called when …

1. … creating a default `S`;
2. … creating an instance of `S` via the copy constructor;
3. … creating an instance of `S` via a function returning an `S`;

Copy Elision

Task: Given the following definition of `S`, what is printed in the `main()` function?

```
struct S {
  S() { puts("S()"); }
  S(S const&) { puts("S(const S&)"); }
  S& operator=(S const&) { puts("operator=(const S&)"); return *this; }
  ~S() { puts("~S()"); }
};
```

```
int main()
{
   S s{};


}
```

```
// Output:
//  S()
//  ~S()
```

Copy Elision

Task: Given the following definition of `S`, what is printed in the `main()` function?

```
struct S {
   S() { puts("S()"); }
   S(S const&) { puts("S(const S&)"); }
   S& operator=(S const&) { puts("operator=(const S&)"); return *this; }
   ~S() { puts("~S()"); }
};
```

```
int main()
{
    S s{};
    S s2{ s };

}
```

```
// Output:
//  S()
//  S(S const&)
//  ~S()
//  ~S()
```

Copy Elision

Task: Given the following definition of `S`, what is printed in the `main()` function?

```
struct S {
   S() { puts("S()"); }
   S(S const&) { puts("S(const S&)"); }
   S& operator=(S const&) { puts("operator=(const S&)"); return *this; }
   ~S() { puts("~S()"); }
};

S createS() { return S{}; }

int main()
{
    S s{ createS() };


}
```

```
// Output:
//  S()
//  ~S()
```

Copy Elision

Task: Given the following definition of `S`, what is printed in the `main()` function?

```
struct S {
  S() { puts("S()"); }
  S(S const&) { puts("S(const S&)"); }
  S& operator=(S const&) { puts("operator=(const S&)"); return *this; }
  ~S() { puts("~S()"); }
};

S createS() { return S{}; }

int main()
{
   S s{};

   s = createS();
}
```

```
// Output:
//  S()
//  S()
//  operator=(const S&)
//  ~S()
//  ~S()
```

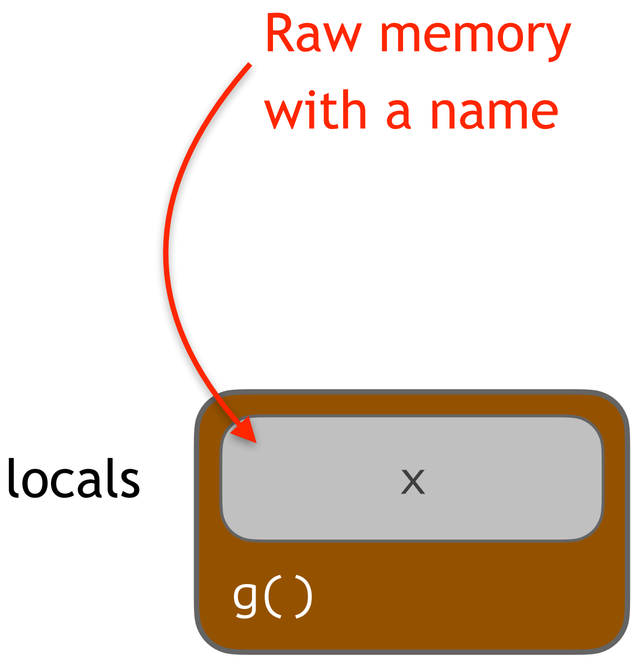
Copy Elision

Task: Given the following definition of `S`, what is printed in the `main()` function?

```
struct S {
  S() { puts("S()"); }
  S(S const&) { puts("S(const S&)"); }
  S& operator=(S const&) { puts("operator=(const S&)"); return *this; }
  ~S() { puts("~S()"); }
};

S createS() { return S{}; }

int main()
{
   S s{};
   S __tmp__{ createS() };
   s = __tmp__;
}
```

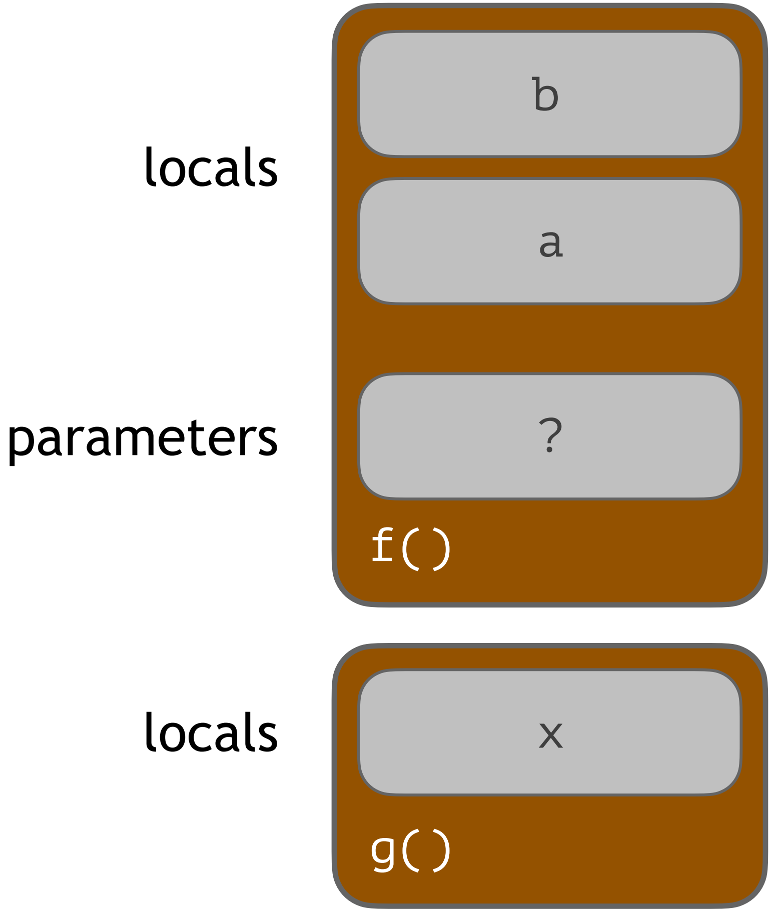
```
// Output:
//  S()
//  S()
//  operator=(const S&)
//  ~S()
//  ~S()
```

Unoptimized Return Value



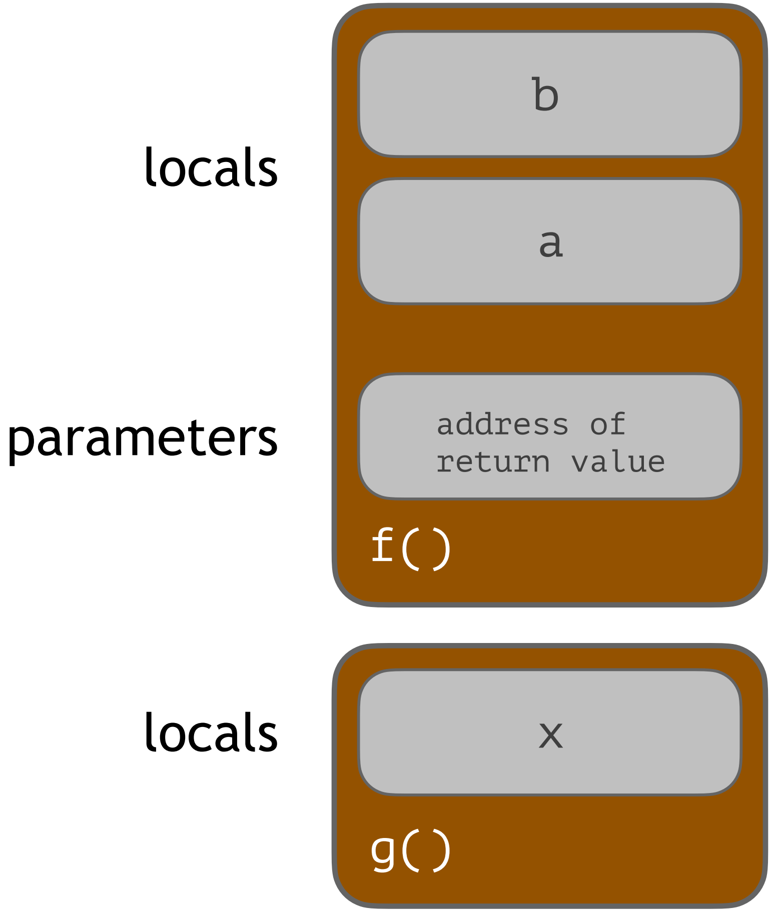

```
std::string f()
{
    std::string a{"A"};
    int b{23};
    // ...
    return a;
}

void g()
{
    std::string x{ f() };
}
```

Unoptimized Return Value



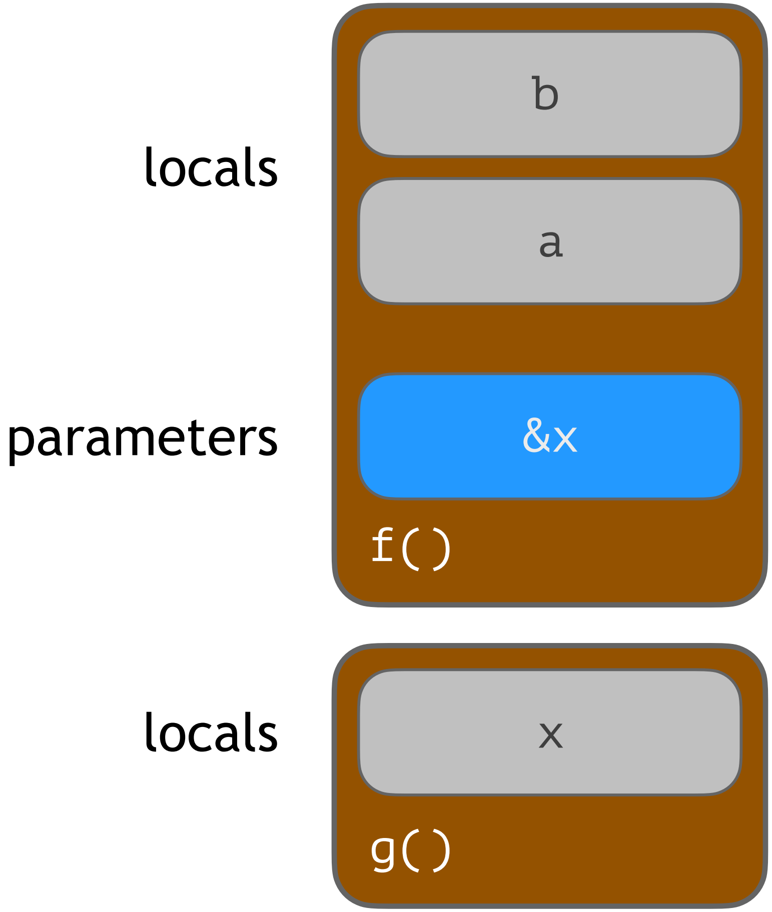

```
std::string f()
{
    std::string a{"A"};
    int b{23};
    // ...
    return a;
}

void g()
{
    std::string x{ f() };
}
```

Unoptimized Return Value



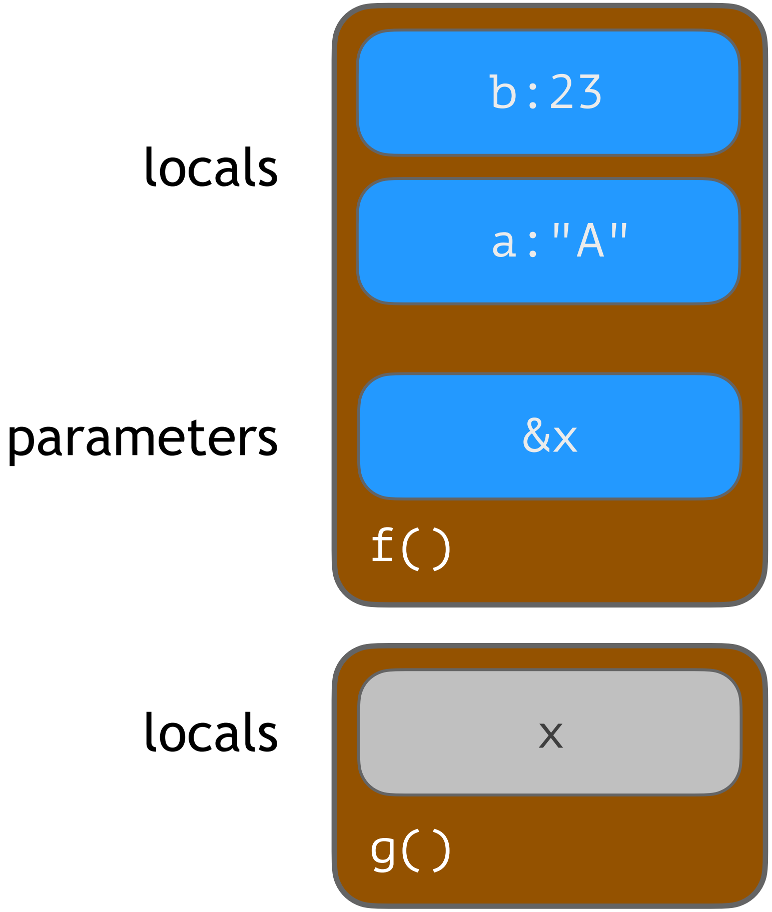

```
std::string f()
{
    std::string a{"A"};
    int b{23};
    // ...
    return a;
}

void g()
{
    std::string x{ f() };
}
```

Unoptimized Return Value



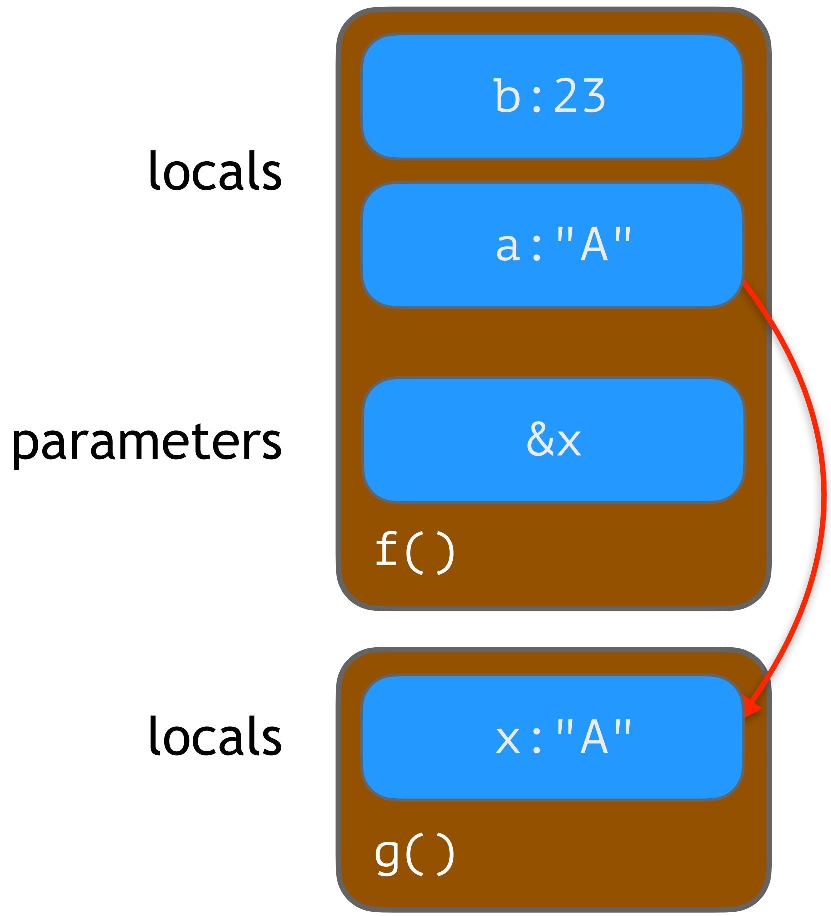

```
std::string f()
{
    std::string a{"A"};
    int b{23};
    // ...
    return a;
}

void g()
{
    std::string x{ f() };
}
```

Unoptimized Return Value




```
std::string f()
{
    std::string a{"A"};
    int b{23};
    // ...
    return a;
}

void g()
{
    std::string x{ f() };
}
```

Unoptimized Return Value

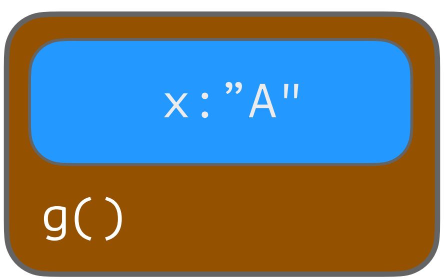



```
std::string f()
{
    std::string a{"A"};
    int b{23};
    // ...
    return a;
}

void g()
{
    std::string x{ f() };
}
```

Unoptimized Return Value

```
std::string f()
{
    std::string a{"A"};
    int b{23};
    // ...
    return a;
}

void g()
{
    std::string x{ f() };
}
```

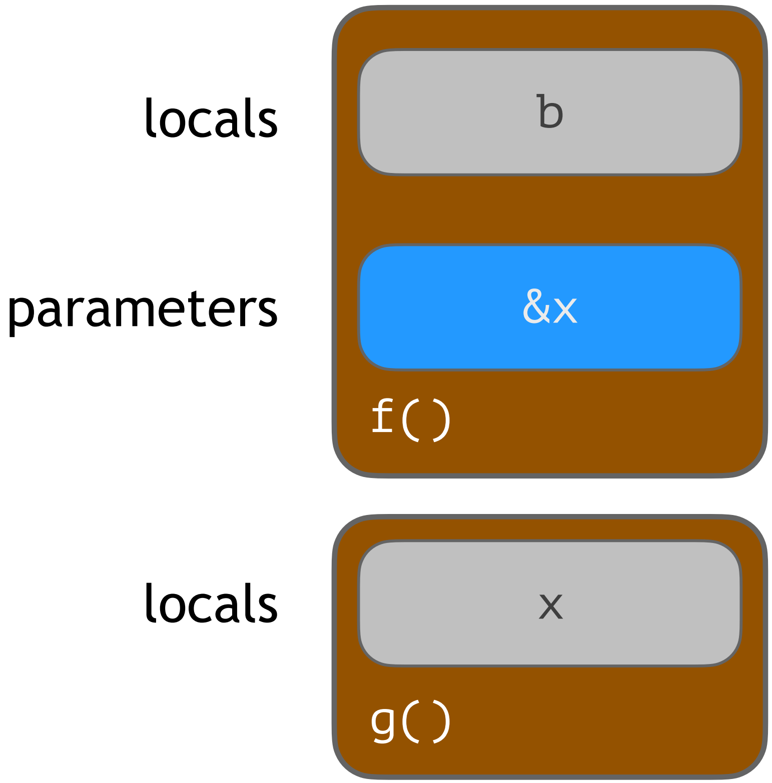
locals



Return Value Optimization (RVO)

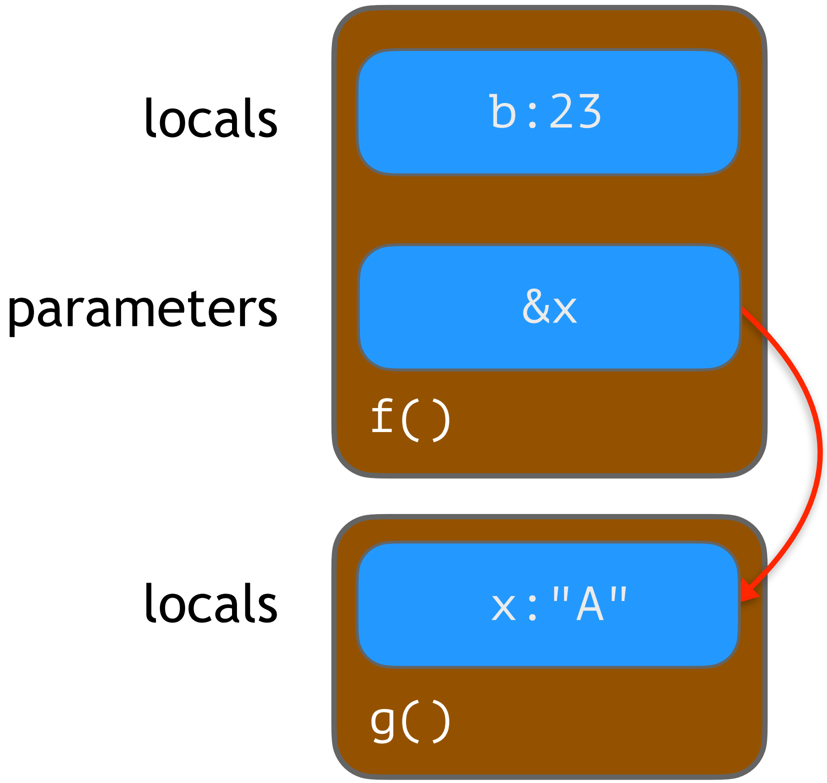
```
std::string f()
{
   std::string a{"A"};
   int b{23};
   // ...
   return a;
}

void g()
{
   std::string x{ f() };
}
```

Return Value Optimization (RVO)



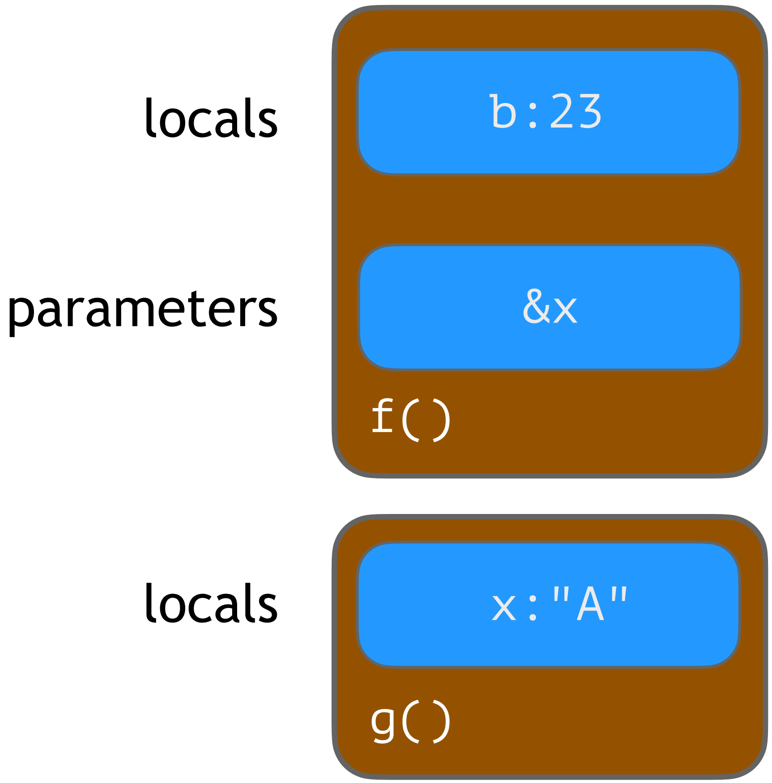

```
std::string f()
{
    std::string a{"A"};
    int b{23};
    // ...
    return a;
}

void g()
{
    std::string x{ f() };
}
```

Return Value Optimization (RVO)




```
std::string f()
{
    std::string a{"A"};
    int b{23};
    // ...
    return a;
}

void g()
{
    std::string x{ f() };
}
```

Return Value Optimization (RVO)

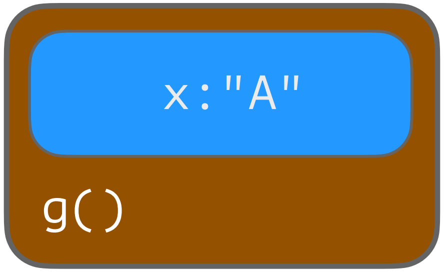



```
std::string f()
{
    std::string a{"A"};
    int b{23};
    // ...
    return a;  // No-op
}

void g()
{
    std::string x{ f() };
}
```

Return Value Optimization (RVO)

```
std::string f()
{
    std::string a{"A"};
    int b{23};
    // ...
    return a;
}

void g()
{
    std::string x{ f() };
}
```

locals



Programming Task

Task (2_The_Special_Member_Functions/RVO2): Evaluate the given code examples. Will the functions apply copy elision (aka RVO)?

Further Reading

- Copy Elision on CppReference: https://en.cppreference.com/w/cpp/language/copy_elision
- Wikipedia: https://en.wikipedia.org/wiki/Copy_elision

Guidelines

Guideline: Prefer to return by value (relying on copy elision or move).

Core Guideline F.20: For “out” output values, prefer return values to output parameters

2.5. Move Semantics

Programming Task

Task (2_The_Special_Member_Functions/CreateStrings): Benchmark the given code example to create a performance base line!

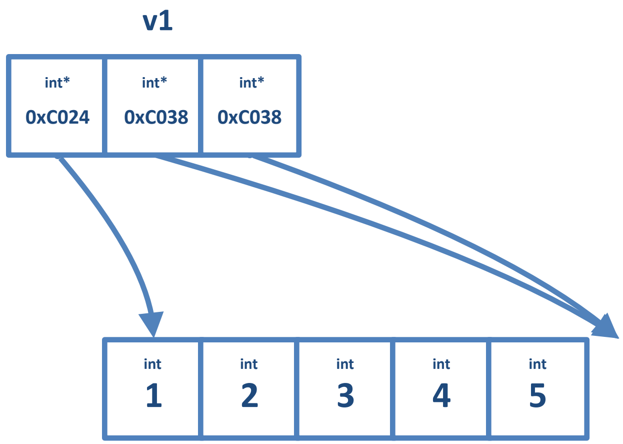
The Basics of Move Semantics

The Basics of Move Semantics

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

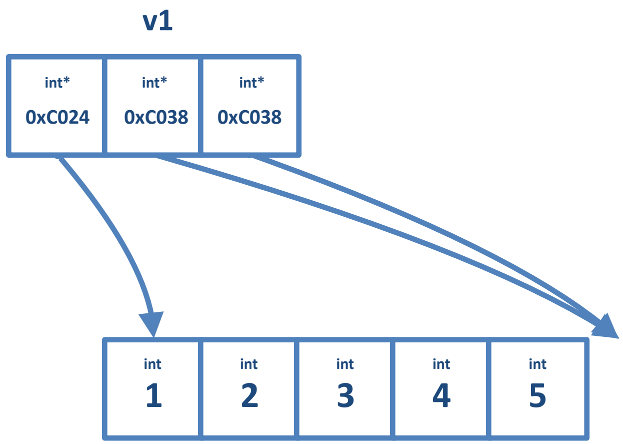
The Basics of Move Semantics

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



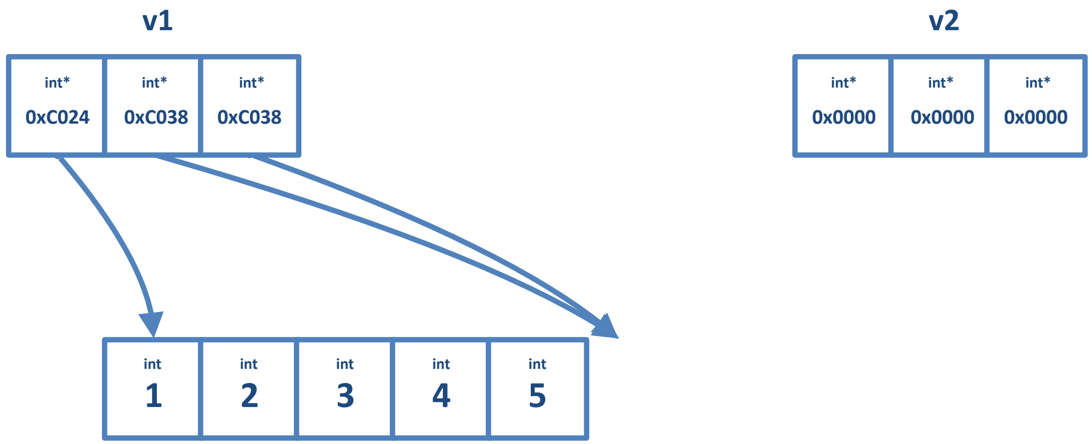
The Basics of Move Semantics

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



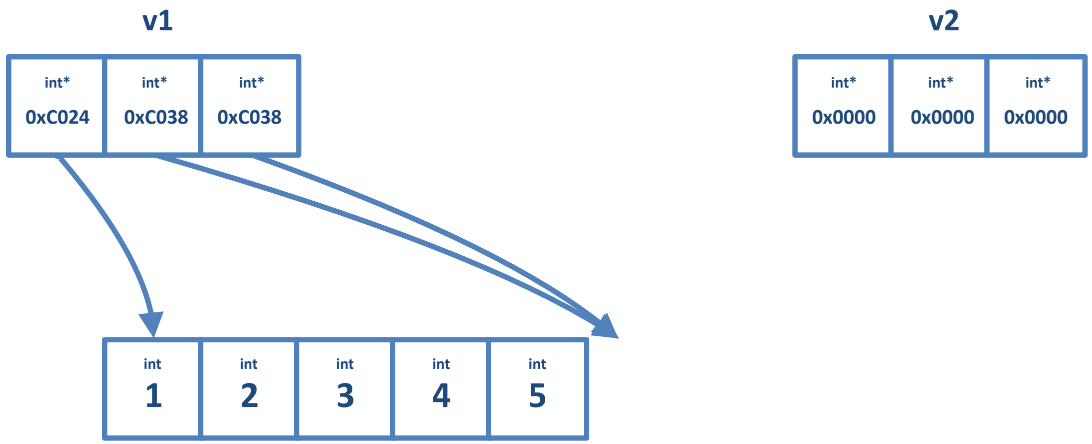
The Basics of Move Semantics

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



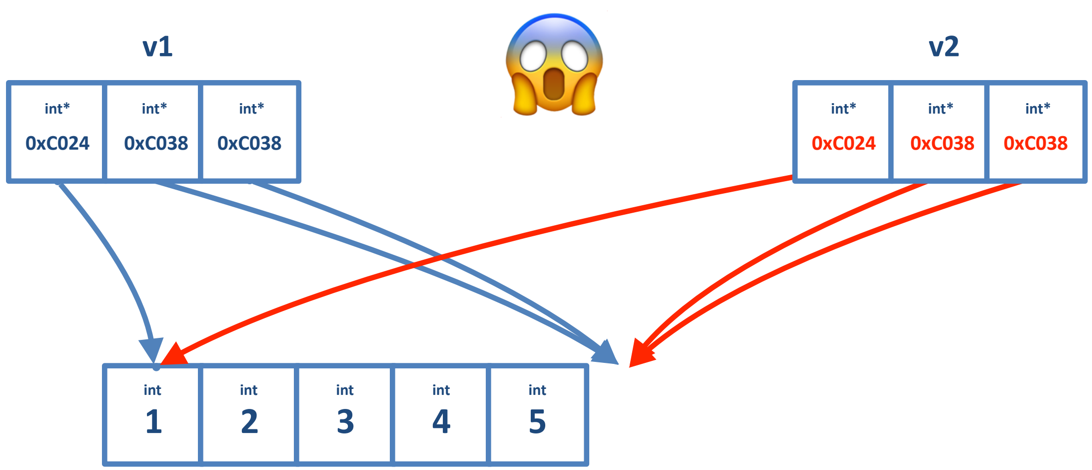
The Basics of Move Semantics

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

v2 = v1;
```



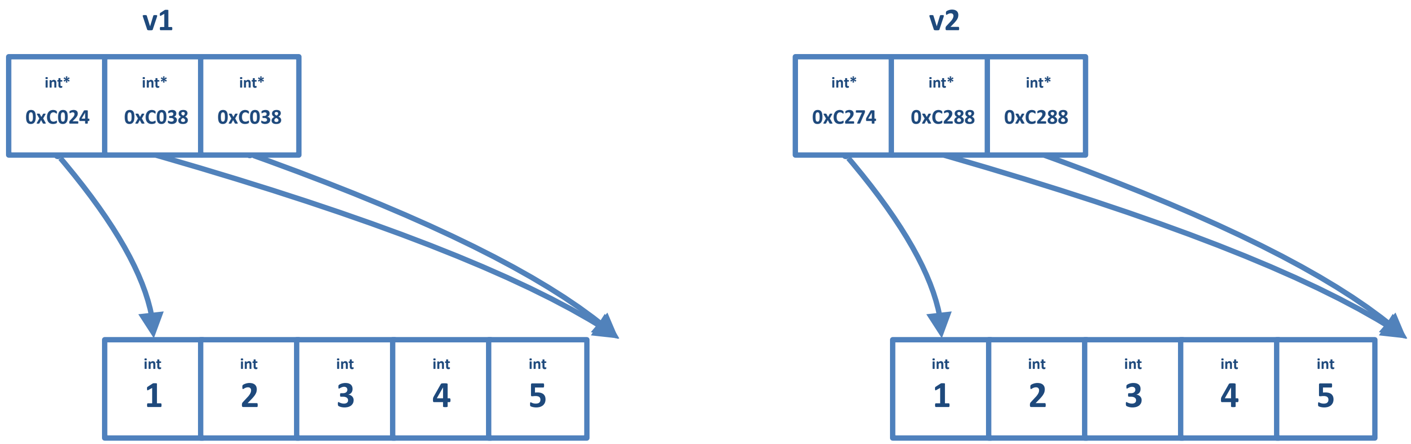
The Basics of Move Semantics

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

v2 = v1;
```



The Basics of Move Semantics

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

v2 = v1;
```

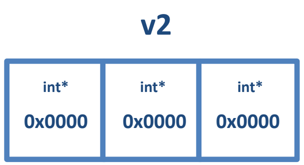


The Basics of Move Semantics

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

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

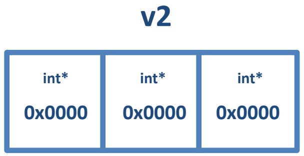
The Basics of Move Semantics

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

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



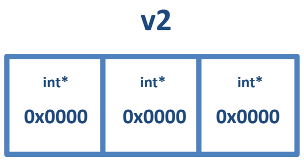
The Basics of Move Semantics

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

std::vector<int> v2{};

v2 = createVector();
```



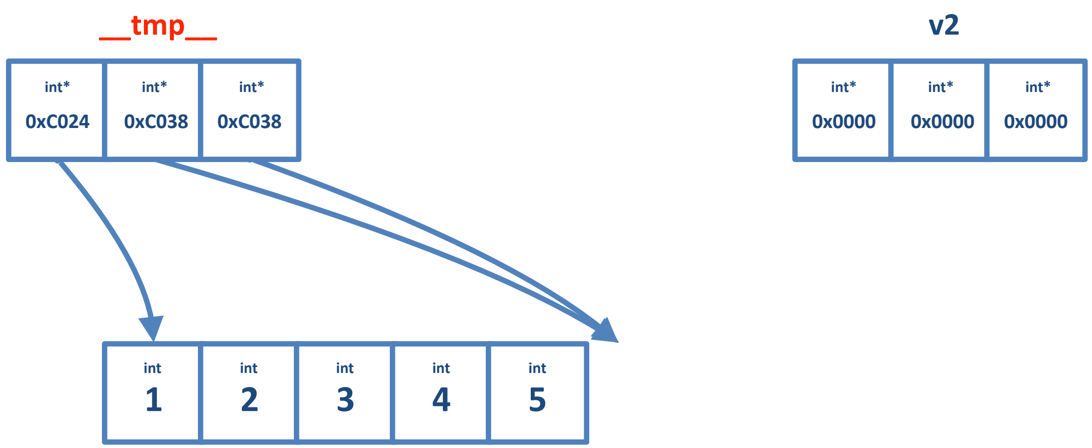
The Basics of Move Semantics

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

std::vector<int> v2{};

v2 = createVector();  // Due to copy elision equivalent to:
                      // auto __tmp__{ createVector() };
                      // v2 = __tmp__;
```



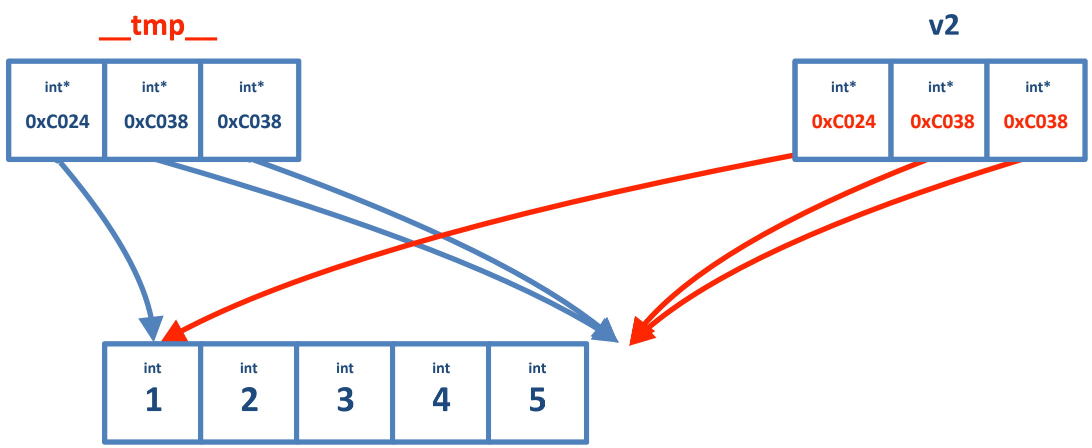
The Basics of Move Semantics

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

std::vector<int> v2{};

auto __tmp__{ createVector() };  // Copy Elision
v2 = __tmp__;
```



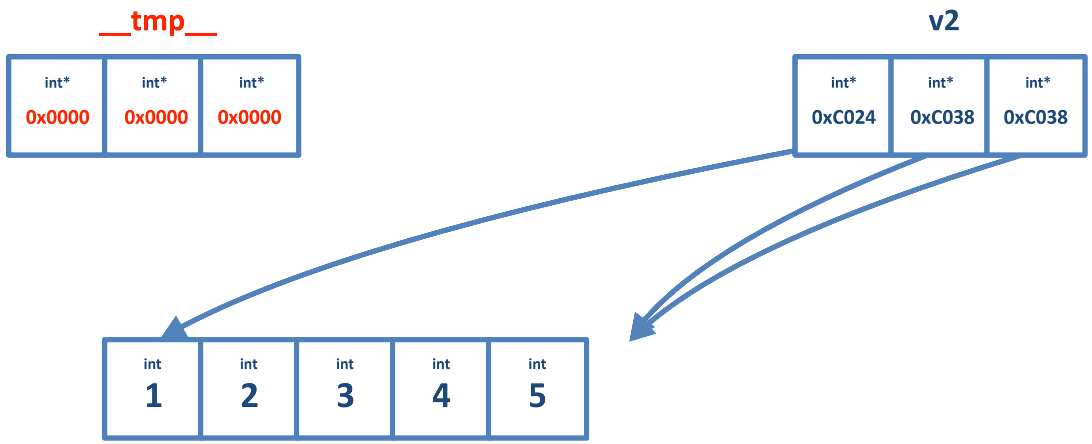
The Basics of Move Semantics

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

std::vector<int> v2{};

auto __tmp__{ createVector() };  // Copy Elision
v2 = __tmp__;
```



The Basics of Move Semantics

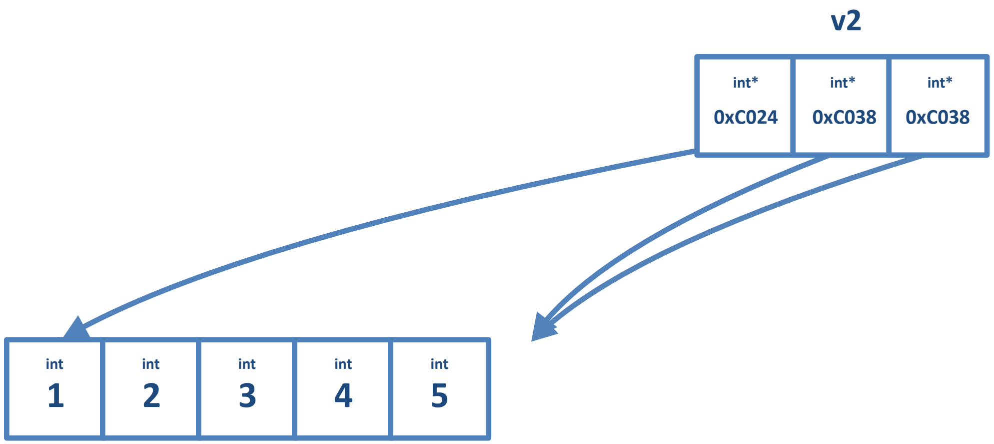
```
std::vector<int> createVector() {
   return std::vector<int>{ 1, 2, 3, 4, 5 };
}

std::vector<int> v2{};

auto __tmp__{ createVector() };  // Copy Elision
v2 = __tmp__;
```



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

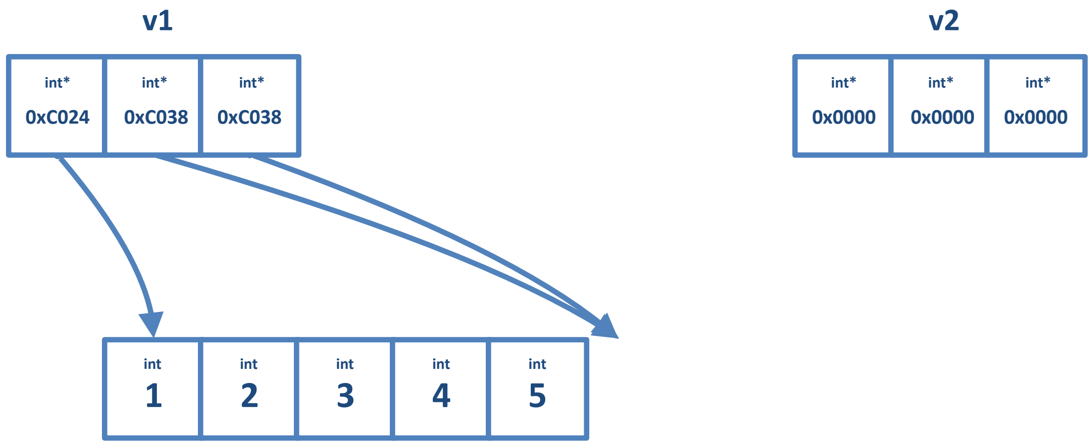
The Basics of Move Semantics

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

std::vector<int> v2{};

auto __tmp__{ createVector() };  // Copy Elision
v2 = __tmp__;
```



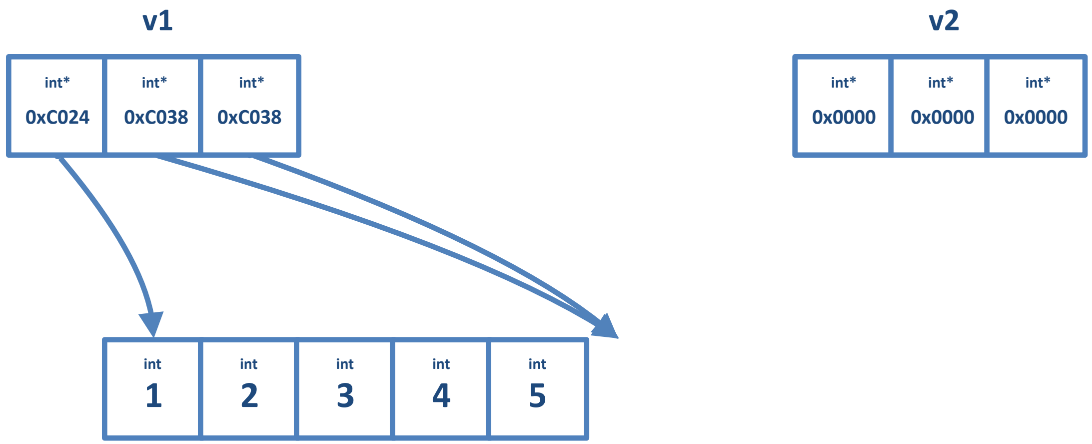
The Basics of Move Semantics

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

v2 = v1;
```



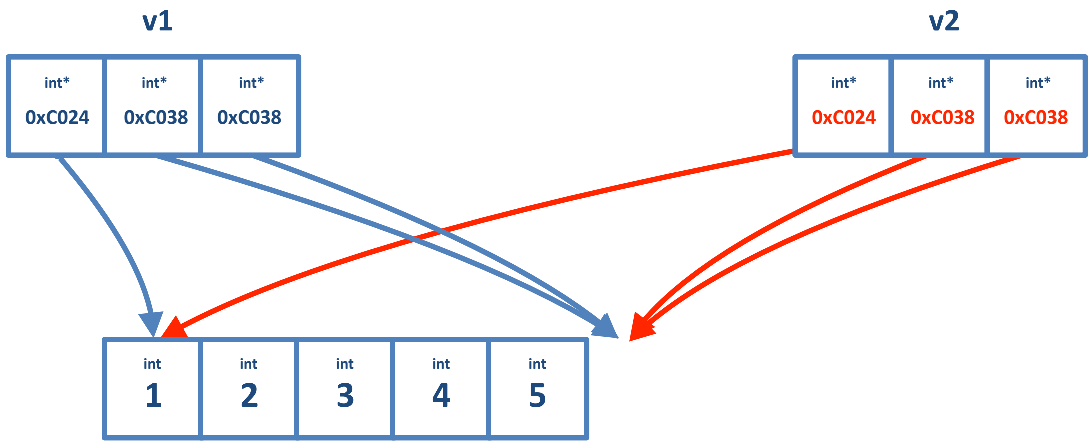
The Basics of Move Semantics

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

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



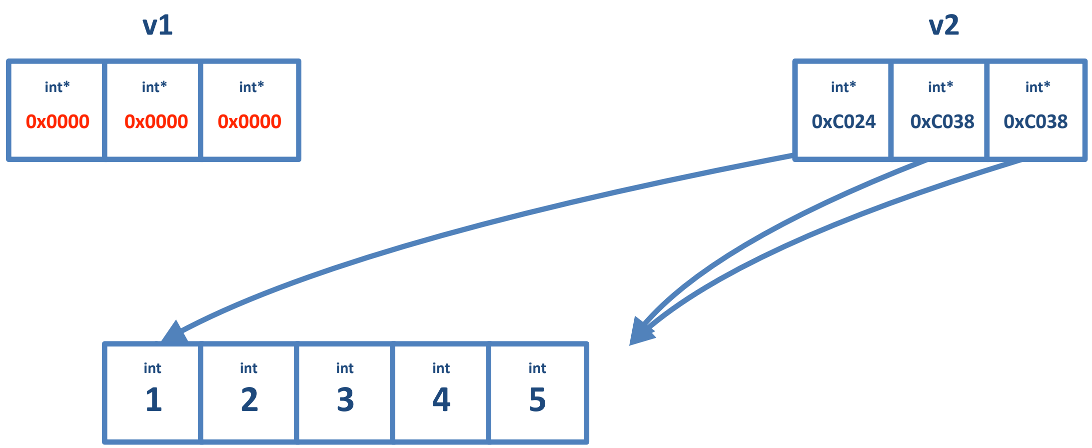
The Basics of Move Semantics

```
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

```cpp
template< typename T
        , typename A = /*...*/ >
class vector
{
 public:
   // ...
   // Copy assignment operator

   vector&
     operator=(vector const& other);




   // ...
};
```

```cpp
std::vector<int> v1{ ... };

std::vector<int> v2{};

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


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=(vector const& other);





   // ...
};
```

```
std::vector<int> v1{ ... };

std::vector<int> v2{};

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


v2 = v1;   // Lvalue


v2 = createVector();


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

Lvalues and Rvalues

```
l = r;
```

Lvalues and Rvalues

Lvalue → `l = r;`

(“left”)

Lvalues and Rvalues

```
l = r;
```

Rvalue

(“**right**”, “**read-only**”)

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

Lvalues and Rvalues

```
~~l = r;~~
```

```
std::string s{};

s + s = s;
```

Lvalue

(has a **name**)

Lvalues and Rvalues

```
l = r;
```

```
std::string s{};

s + s = s;
```

Rvalue

(has **no name**)

Lvalues and Rvalues

~~`l = r;`~~

```
std::string s{};

s + s = s;
```

Rvalue
(has **no name**)

Lvalue
(has a **name**)

The Basics of Move Semantics

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

   // ...
};
```

```
std::vector<int> v1{ ... };

std::vector<int> v2{};

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

v2 = v1;   // Lvalue

v2 = createVector();

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

The Basics of Move Semantics

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



   // ...
};
```

(pre C++11)

```cpp
std::vector<int> v1{ ... };

std::vector<int> v2{};

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

v2 = v1;   // Lvalue

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=(vector const& other);

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

   // ...
};
```

```
std::vector<int> v1{ ... };

std::vector<int> v2{};

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


v2 = v1;   // Lvalue


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=(vector const& other);

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

   // ...
};
```

```
std::vector<int> v1{ ... };

std::vector<int> v2{};

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

v2 = v1;    // Lvalue

v2 = createVector();   // Rvalue

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

std::move

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

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

The Basics of Move Semantics

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

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

   // ...
};
```

```
std::vector<int> v1{ ... };

std::vector<int> v2{};

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


v2 = v1;   // Lvalue


v2 = createVector();   // Rvalue


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

Things to remember

- An rvalue reference is a reference to a temporary object created by the compiler
- An rvalue reference is unique, i.e. no-one else holds a reference to the same object
- Therefore, an object may be modified through an rvalue reference without changing program correctness
- Alternatively, the programmer may deliberately decide that the above is true by applying `std::move`
- Move semantics is primarily an optimization feature in order to avoid unnecessary expensive deep copies
- Additionally, there is the semantical part of move semantics, which allows you to express transfer of ownership explicitly

Programming Task

Task (2_The_Special_Member_Functions/CreateStrings): Improve the performance of the given code by refactoring. After each modification, first predict how performance is affected and then benchmark the actual effect. Explain why performance was affected accordingly. Note that we assume that the `createStrings()` function does not produce a predictable result!

The Move Ctor and Move Assignment Operator

The Move Ctor and Move Assignment Operator

In order to enable move semantics your class requires …

- … a move constructor …
- … a move assignment operator …

… where the move version is optimized to …

- … steal the contents from the passed object;
- … set the passed object to a valid but unspecified state!

```
class Widget {
 public:
   // ...
   Widget( Widget&& );                    // Move constructor
   Widget& operator=( Widget&& );   // Move assignment operator
   // ...
};
```

The Signatures of the Move Operations

The signature of the move constructor:

```
Widget( Widget&& );                // The default
~~Widget( Widget const&& );~~      // Possible, but very uncommon
```

The signature of the move assignment operator:

```
Widget& operator=( Widget&& );              // The default
~~Widget& operator=( Widget const&& );~~    // Possible, but very uncommon
```

The Move Ctor and Move Assignment Operator

The compiler generates the move operations …

```
// Compiler-generated move ctor and move assignment available
class Widget
{
 public:

   // ...

};

Widget w1{};
Widget w2( std::move(w1) );  // Compiler generated, ok
w1 = std::move(w2);          // Compiler generated, ok
```

The Move Ctor and Move Assignment Operator

The compiler generates the move operations …

- if they are not explicitly declared …

```
// Compiler-generated move ctor and move assignment available
class Widget
{
 public:
   Widget( Widget&& );
   Widget& operator=( Widget&& );
   // ...
};

Widget w1{};
Widget w2( std::move(w1) );  // Explicitly defined, ok
w1 = std::move(w2);          // Explicitly defined, ok
```

The Move Ctor and Move Assignment Operator

The compiler generates the move operations …

- if they are not explicitly declared …
- if no destructor and no copy operation is declared …

```
// Compiler-generated move ctor and move assignment not available
class Widget
{
 public:
   Widget( Widget const& );   // or alternatively declaration of
   // ...                     //    destructor or copy assignment

};

Widget w1{};
Widget w2( std::move(w1) );   // Copy ctor instead of move ctor
w1 = std::move(w2);           // Copy assign instead of move assign
```

The Move Ctor and Move Assignment Operator

The compiler generates the move operations …

- if they are not explicitly declared …
- if no destructor and no copy operation is declared …
- if all bases/data members can be copy or move constructed/assigned.

```
// Compiler-generated copy ctor and copy assignment available
class Widget : public Base
{
 public:
   // ...
 private:
   NonCopyable member_;  // Data member without copy operations
};

Widget w1{};
Widget w2( std::move(w1) );  // Compiler generated, ok
w1 = std::move(w2);          // Compiler generated, ok
```

The Move Ctor and Move Assignment Operator

The compiler generates the move operations …

- if they are not explicitly declared …
- if no destructor and no copy operation is declared …
- if all bases/data members can be copy or move constructed/assigned.

```
// Compiler-generated copy ctor and copy assignment available
class Widget : public Base
{
 public:
   // ...
 private:
   NonMovable member_;  // Data member without move operations
};

Widget w1{};
Widget w2( std::move(w1) );  // Copy ctor instead of move ctor
w1 = std::move(w2);          // Copy assign instead of move assign
```

The Move Ctor and Move Assignment Operator

The compiler generates the move operations …

- if they are not explicitly declared …
- if no destructor and no copy operation is declared …
- if all bases/data members can be copy or move constructed/assigned.

```
// Compiler-generated copy ctor and copy assignment not available
class Widget : public Base
{
 public:
   // ...
 private:
   Immobile member_;  // Data member without copy AND move ops
};

Widget w1{};
Widget w2( std::move(w1) );  // Compiler error: No move ctor
w1 = std::move(w2);          // Compiler error: No move assignment
```

The Default Implementation

```
class Widget : public Base
{
 public:
   Widget( Widget&& other )
      : Base{ std::move(other) }   // The default move constructor performs
      , i { std::move(other.i)  }  // a member-wise move construction of
      , s { std::move(other.s)  }  // all bases and data members
      , pi{ std::move(other.pi) }
   {}
   Widget& operator=( Widget&& other )
   {
      Base::operator=( std::move(other) );
      i  = std::move(other.i);     // The default move assignment operator
      s  = std::move(other.s);     // performs a member-wise move assignment
      pi = std::move(other.pi);    // of all bases and data members
      return *this;
   }
   // ...

 private:              // The three data members:
   int i;              // - i as a representative of a fundamental type
   std::string s;      // - s as a representative of a class (user-defined) type
   int* pi{};          // - pi as representative of a possible resource
};
```

Programming Task

Task (2_The_Special_Member_Functions/ResourceOwner): Implement the move operations of class `ResourceOwner`.

```
class ResourceOwner {
 public:
   // ...
   ResourceOwner( ResourceOwner&& );
   ResourceOwner& operator=( ResourceOwner&& );
   // ...
};
```

Further Reading

Core Guideline C.65: Make move assignment safe for self-assignment

- https://stackoverflow.com/questions/9322174/move-assignment-operator-and-if-this-rhs
- https://scottmeyers.blogspot.com/2014/06/the-drawbacks-of-implementing-move.html
- http://www.open-std.org/jtc1/sc22/wg21/docs/lwg-defects.html#1204

Programming Task

Task (2_The_Special_Member_Functions/ResourceOwner): Refactor the `ResourceOwner` to remove as many of the special member functions as possible without changing the interface or behavior.

Valid, but Unspecified?

Valid, but Unspecified?

A moved-from object …

- … must be in a valid state (i.e. all invariants must be fulfilled)
- … is not necessarily in the same state as before the move

Special examples are `std::vector` and `std::unique_ptr`. These two promise to be in a default state after the move.

As a counter example: `std::string` does not promise to be in a default state after the move.

Valid, but Unspecified?

Move, simply – Sutter's Mill 25.02.24, 12:46

Sutter's Mill

Herb Sutter on software development

Move, simply

Herb Sutter 2020-02-172020-02-21 9 Minutes

C++ "move" semantics are simple, and unchanged since C++11. But they are still widely misunderstood, sometimes because of unclear teaching and sometimes because of a desire to view move as something else instead of what it is. This post is an attempt to shed light on that situation. Thank you to the following for their feedback on drafts of this material: Howard Hinnant (lead designer and author of move semantics), Jens Maurer, Arthur O'Dwyer, Geoffrey Romer, Bjarne Stroustrup, Andrew Sutton, Ville Voutilainen, Jonathan Wakely.

Edited to add: Formatting, added [basic.life] link, and reinstated a "stateful type" Q&A since the question was asked in comments.

Programming Task

Task (2_The_Special_Member_Functions/ResourceOwner): Assume the invariant that the `m_resource` pointer must never be a `nullptr`. What changes to the implementation of the special member functions are necessary?

Programming Task

Task (2_The_Special_Member_Functions/EmailAddress): Implement the special member functions for the given `EmailAddress` class. Note that `EmailAddress` must contain a valid email address at all times!

A Valid Moved-From Email Address

```
class EmailAddress
{
 public:
   explicit EmailAddress( std::string address )
      : address_{std::move(address)}
   {
      if( !is_valid() ) {
         throw std::invalid_argument( "Invalid email address" );
      }
   }

   ~EmailAddress() = default;
   EmailAddress( EmailAddress const& ) = default;
   EmailAddress& operator=( EmailAddress const& ) = default;
   // Move constructor explicitly omitted
   // Move assignment operator explicitly omitted

   std::string const& value() const { return address_; }
   bool is_valid() const { return /* checking the email address */; }

 private:
   std::string address_;
};
```

The move operations might empty the string. This would not represent a valid email address. Therefore they are explicitly omitted and copy is used by default. This has of course unfortunate performance repercussions.

Valid, but Unspecified?

Sean Parent

About Sean Parent Papers and Presentations

Move Annoyance

Mar 31, 2021

[Written for Lakos, J., Romeo, V., Khlebnikov, R., & Meredith, A. (2021). *Embracing Modern C++ Safely*. Addison-Wesley Professional. Reproduced with permission.]

Annoyances

Required Postconditions of a Moved-From Object Are Overly Strict

Given an object, `rv`, which has been moved from, the C++20[1] Standard specifies the required postconditions of a moved-from object:

Copy Control

Task (2_The_Special_Member_Functions/CopyControl): Assuming that each of the following classes `A` to `F` should be copyable and moveable and that all given data members are in `private` sections, for which of the classes do you have to explicitly define a copy constructor, a move constructor, a destructor, a copy assignment operator, and a move assignment operator? Check the final solution with AddressSanitizer (see https://en.wikipedia.org/wiki/AddressSanitizer).

The Move Operations and noexcept

Task (2_The_Special_Member_Functions/MoveNoexcept): Examine the influence of declaring the move operations `noexcept` by means of creating a `std::vector` of strings.

Special Member Functions: Guidelines

Guideline: Provide the move operations for your value type.

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

Guideline: Adhere to the *Rule of Five*, but strive for the *Rule of Zero*.

Guideline: Use `=default` and `=delete` liberally in order to specify and document your intent.

Qualified/Modified Member Data

Guideline: Remember that a class with const or reference data member cannot be copy/move assigned by default.

Guideline: Strive for symmetry between the copy operations and the move operations (i.e. avoid const and reference member data).

2. The Special Member Functions – Move Semantics

2.6. The Rule of 0/5

Generating the Move Operations

- Default move operations are NOT generated
  if any copy operation or the destructor is user-defined.
- Default copy operations are NOT generated (implicitly deleted)
  if any 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;

   // ...
};
```

Guidelines

Core Guideline C.21: If you define or `=delete` any copy, move, or destructor function, define or `=delete` them all

Generating the Move Operations

Note that it makes a difference whether you don't provide or explicitly delete the move operations:

- **Move operations not provided:** When an object is moved, copy serves as a fallback
- **Move operations deleted:** Moving an object results in a compilation error

```
class X {
 public:
   virtual ~X() = default;

   X( X&& ) = delete;
   X& operator=( X&& ) = delete;

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

   // ...
};
```

Special Member Functions: Guidelines

Guideline: Note that omitting a copy or move operation, defaulting it or deleting it has different meaning. Stick either to the *Rule of Zero* or the *Rule of Five*.

Guideline: Adhere to the *Rule of Five* if you want to default or delete the move operations, but omit the move operations if you want to copy instead of move.

Special Member Functions: Guidelines

Guideline: Do not define empty destructors in derived classes.

```
class AbstractBase {
   // ...
};

class Derived : public AbstractBase {
 public:
   virtual ~Derived() = default;  // Disables move operations
 private:
   std::vector<int> v;
};
```

Guidelines

Guideline: Take care of the „Rule of Five“: When you require a destructor, any of the copy operations, or any of the move operations, you most likely also require the other four functions.

```
class Widget
{
 public:
   ~Widget();                                  // Destructor

   Widget( Widget const& );                    // Copy constructor

   Widget& operator=( Widget const& );         // Copy assignment operator

   Widget( Widget&& );                         // Move constructor

   Widget& operator=( Widget&& );              // Move assignment operator
};
```

Guidelines

Guideline: Strive for the „Rule of Zero“: Classes that don‘t require an explicit copy ctor, copy assignment operator, move ctor, move assignment operator and destructor are much (!) easier to handle.

```
class Widget
{
 public:
   Widget( size_t size )
      : vec_( size )
   {}

   // ...

 private:
   std::vector<int> vec_;
};
```

Howard Hinnant's Summary Table

User declares \ Compiler implicitly declares	Default constructor	*destructor*	*copy constructor*	*copy assignment*	*move constructor*	*move assignment*
nothing	defaulted	defaulted	defaulted	defaulted	defaulted	defaulted
any constructor	not declared	defaulted	defaulted	defaulted	defaulted	defaulted
default constructor	user-declared	defaulted	defaulted	defaulted	defaulted	defaulted
destructor	defaulted	user-declared	defaulted	defaulted	not declared	not declared
copy constructor	not declared	defaulted	user-declared	defaulted	not declared	not declared
copy assignment	defaulted	defaulted	defaulted	user-declared	not declared	not declared
move constructor	not declared	defaulted	deleted	deleted	user-declared	not declared
move assignment	defaulted	defaulted	deleted	deleted	not declared	user-declared

Table taken from Howard Hinnant's talk "Everything You Ever Wanted To Know About Move Semantics (and then some)" (https://www.youtube.com/watch?v=vLinb2fgkHk)

The Rule of 6?

The Rule of 6 = Rule of 5 + the default constructor.

The Rule of 6 implies that you should write all six special member functions, instead of only 5.

In practice, this is not true. The default constructor …

- … does not (**technically**) affect any of the other special member function;
- … does not necessarily require to write the other special member functions (although it is **semantically** likely);
- … does not have to be written if there is no (reasonable) default.

Further Reading

- Phil Nash, “The Rules of Three, Five and Zero”. Sonar Blog Post (https://www.sonarsource.com/blog/the-rules-of-three-five-and-zero/)

Guidelines

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

Core Guideline C.21: If you define or `=delete` any copy, move, or destructor function, define or `=delete` them all

Core Guideline C.80: Use `=default` if you have to be explicit about using the default semantics

Core Guideline C.81: Use `=delete` when you want to disable default behavior (without wanting an alternative)

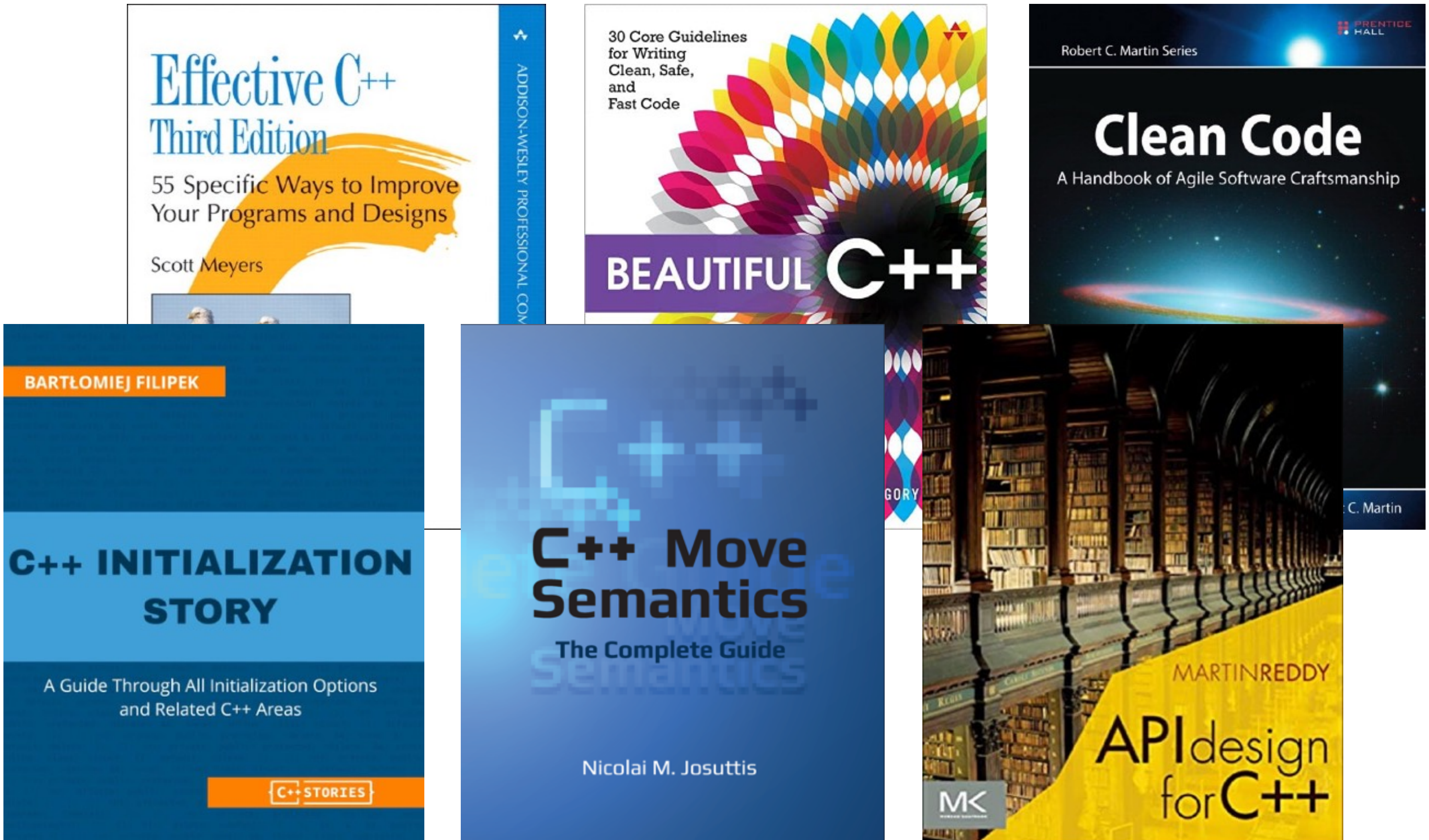
Guidelines

Core Guideline C.130: For making deep copies of polymorphic classes prefer a virtual `clone` function instead of copy construction/assignment.

Things to Remember

- Remember the default initialization of the default constructor
- Adhere to the expected copy semantics
- Implement the move operations for your value types
- Remember the “Rule of 0” and “Rule of 5”
- Use `=default` and `=delete` liberally

Literature


Effective C++
Third Edition
55 Specific Ways to Improve Your Programs and Designs
Scott Meyers
30 Core Guidelines for Writing Clean, Safe, and Fast Code
BEAUTIFUL C++
Robert C. Martin Series
Clean Code
A Handbook of Agile Software Craftsmanship
BARTŁOMIEJ FILIPEK
C++ INITIALIZATION STORY
A Guide Through All Initialization Options and Related C++ Areas
C++ STORIES
C++ Move Semantics
The Complete Guide
Nicolai M. Josuttis
MARTINREDDY
APIdesign for C++

References

- Klaus Iglberger, “Back to Basics: Designing Classes (Part 1 of 2)”. CppCon 2021 (TBA)
- Klaus Iglberger, “Back to Basics: The Special Member Functions”. CppCon 2021 (TBA)
- Klaus Iglberger, “Back to Basics: Exception Safety”. CppCon 2020 (https://www.youtube.com/watch?v=0ojB8c0xUd8)
- Arthur O’Dwyer, “Back to Basics: RAII and the Rule of Zero”, CppCon 2019 (https://www.youtube.com/watch?v=7Qgd9B1KuMQ)
- Kate Gregory, “What Do We Mean When We Say Nothing At All”. CppCon 2018 (https://www.youtube.com/watch?v=kYVxGyido9g)
- Klaus Iglberger, “Back to Basics: Move Semantics (part 1 of 2)”. CppCon 2019 (https://www.youtube.com/watch?v=St0MNEU5b0o&t=15s)
- David Olsen, “Back to Basics: Move Semantics”. CppCon 2020 (https://www.youtube.com/watch?v=ZG59Bqo7qX4)
- Nicolai Josuttis, “The Nightmare of Move Semantics for Trivial Classes”. CppCon 2017 (https://www.youtube.com/watch?v=PNRju6_yn3o)
- Nicolai Josuttis, “The Hidden Features of Move Semantics”. CppCon 2020 (https://www.youtube.com/watch?v=TFMKjL38xAI)

email: klaus.iglberger@gmx.de

LinkedIn: linkedin.com/in/klaus-iglberger-2133694/

Xing: xing.com/profile/Klaus_Iglberger/cv