The Canonical Class

Understanding what goes into a C++11 Class

Attribution

- Copyright © 2014
- You can follow him @MichaelCaisse
- CppCon 2014

std::disclaimer

- The example code for demonstrating a purpose
- Please do not assume styles are considered good practice
- Please, never `using std;` in your own code
- Please, always use scoping and namespaces properly

Outline

- A Simpler World
- Moving On
- Tidbits
- Observations

How should we write a simple class for C++11?

What is the canonical form of a class?

§

Please provide us with a list of rules to follow.

§

Some proposals:

- Rule of 3
- Rule of 4
- Rule of 5
- Rule of Zero

http://en.cppreference.com/w/cpp/language/rule_of_three

§

Well, some dislike fabricated rules.

§

Effective C++ Item 7: Declare destructors virtual in polymorphic base classes.

- Polymorphic base classes should declare virtual destructors. If a class has any virtual functions, it should have a virtual destructor.
- Classes not designed to be base classes or not designed to be used polymorphically should not declare virtual destructors.

Scott Meyers

- Some dislike fabricated rules. (Includes Michael Caisse and me.)
- We don’t mind guidelines – if brain is engaged
- The average team needs guidelines
- Rules/guidelines are not a replacement for mentoring via code review

How should we write a simple class for C++11?

In C++98:

- Default c’tor
- Copy c’tor
- Assignment operator
- D’tor

In C++11/14:

- Default c’tor
- Copy c’tor
- *Copy assignment*
- D’tor
- *Move c’tor*
- *Move assignment*

In C++98:

- Default c’tor
- Copy c’tor
- Assignment operator
- D’tor

In C++11/14:

- Default c’tor
- Copy c’tor
- Copy assignment
- D’tor
- Move c’tor
- Move assignment
- Custom `swap`
- `initializer_list`
- `noexcept`
- `constexpr`

Default c’tor

What does it mean to default construct an object?

Default c’tor

What does it mean to default construct?

- `std::vector` .. what happens if you call `front()`?
- `std::thread` ..
- `std::unique_ptr(constexpr)`
- `std::reference_wrapper`

Default c’tor

Is your type default constructible?

Default c’tor

```
class foo
{
public:
    foo() = delete;
};

error: use of deleted function 'foo::foo()'
    foo f;
        ^
```

Default c’tor

```
class foo
{
private:
     int i;
     bar b;
};
```

Default c’tor

```
class foo
{
public:
    inline foo() : i(), b() {}

private:
    int i;
    bar b;
};
```

Default c’tor

```
class foo
{
public:
    foo() = default;
};
```

Special Member Methods

- Implicit – Do nothing and take the compiler's implementation
- Explicit – Define what you want
  - user supplied `{ /* ... */ }`
  - `= delete;`
  - `= default;`

The Three

```
class circular
{
public:
     circular(size_t i=20)
          : buffer(new uint8_t[i]),
            head(buffer), tail(buffer) {}
     ~circular() { delete[] buffer; }
private:
     uint8_t *buffer, *head, *tail;
};
```

```
circular a;
circular b = a;

class circular
{
public:
    circular(size_t i=20)
        : buffer(new uint8_t[i]),
          head(buffer), tail(buffer) {}
    ~circular() { delete[] buffer; }
private:
    uint8_t *buffer, *head, *tail;
};
```

```
*** glibc detected *** \bin/start.test/gcc-4.8.1/debug/thread
\double free or corruption (faststop): 0x0000000001a4e010
***
```

```
class circular
{
public:
    circular(size_t i=20)
        : buffer(new uint8_t[i]),
          head(buffer), tail(buffer) {}
    ~circular() { delete[] buffer; }
private:
    uint8_t *buffer, *head, *tail;
};
```

```
class circular
{
public:
     circular(size_t i=20)
          : buffer(new uint8_t[i]),
            head(buffer), tail(buffer) {}
     ~circular() { delete[] buffer; }
private:
     uint8_t *buffer, *head, *tail;
};
```

```
class circular
{
public:
    circular(size_t i=20)
        : buffer(new uint8_t[i]),
          head(buffer), tail(buffer) {}
    ~circular() { delete[] buffer; }
    circular(const circular&) = delete;
    circular& operator=(const circular&) = delete;
private:
    uint8_t *buffer, *head, *tail;
};
```

Copy c'tor – What should it look like?

```
class circular
{
public:
     circular(size_t i=20)
          : buffer(new uint8_t[i]),
            head(buffer), tail(buffer) {}
     ~circular() { delete[] buffer; }
private:
     uint8_t *buffer, *head, *tail;
};
```

```
class circular
{
public:
	// ...
	circular(const circular& other)
		: buffer(new uint8_t[other.size]),
		  head(nullptr), tail(nullptr), size(other.size)
	{
		std::copy(other.head, other.tail, buffer);
		head = buffer + (other.head - other.buffer);
		tail = buffer + (other.tail - other.buffer);
	}
private:
	uint8_t *buffer, *head, *tail;
	size_t size;
};
```

Copy assignment – Why not the same as copy c'tor?

```
circular& operator=(const circular& rhs)
{
        if (&rhs == this) { return *this; }
        if (rhs.size > size)
        {
                delete[] buffer;
                buffer = new uint8_t[size];
        }
        size = rhs.size;
        std::copy(rhs.head, rhs.tail, buffer);
        head = buffer + (rhs.head - rhs.buffer);
        tail = buffer + (rhs.tail - rhs.buffer);
        return *this;
}
```

Why?

Why do we need a copy c'tor and assignment operator?

What in `circular` required it?

```
class circular
{
public:
    circular(size_t i=20)
        : buffer(new uint8_t[i]),
          head(buffer), tail(buffer) {}
    ~circular() { delete[] buffer; }
private:
    uint8_t *buffer, *head, *tail;
};
```

Bad compiler

A d’tor implies state change outside of the object being destroyed.

If the object is performing resource allocation, then the compiler generated/supplied methods will be wrong!

Opposite

Is the opposite true?

Does the existence of a copy c’tor and/or assignment operator require a d’tor?

Circular array

```
class circular
{
public:
    // ...
private:
    uint8_t buffer[10];
    uint8_t* head;
    uint8_t* tail;
};
```

Circular array

```
class circular
{
public:
    // ...
private:
    using buffer_type = std::vector<uint8_t>;
    buffer_type buffer;
    buffer_type::iterator head;
    buffer_type::iterator tail;
};
```

```
class circular
{
public:
        circular() : head(buffer), tail(buffer) {}
        circular(const circular& other) { assign(other); }
        circular& operator=(const circular& rhs)
        {
                if (&rhs != this) { assign(rhs); }
                return *this;
        }
private:
        inline void assign(const circular& rhs)
        {
                std::copy(rhs.buffer, rhs.buffer+10, buffer);
                head = buffer + (rhs.head - rhs.buffer);
                tail = buffer + (rhs.tail - rhs.buffer);
        }
        uint8_t buffer[10], *head, *tail;
};
```

Rule of 3

C++ Made Easier: The Rule of Three

By Andrew Koenig and Barbara E. Moo, June 01, 2001

Dr. Dobb's

- If a class has a nonempty destructor, it almost always needs a copy constructor and an assignment operator.
- If a class has a nontrivial copy constructor or assignment operator, it usually needs both of these members and a destructor as well.

Why the rule?

The compiler’s implicit help will hurt us.

Miranda Warning

“If you cannot afford a lawyer, one will be appointed, at public expense”

Compiler spin

"If you do not provide a copy or assignment operator, one will be appointed by the compiler, at the expense of your schedule and sanity."

Compiler Implicit Help

User does

Compiler does

	nothing	c'tor	copy c'tor	copy assign	d'tor	move c'tor	move assign
default c'tor	default			default	default		default
copy c'tor	default	default	user	default	default	delete	delete
copy assign	default	default	default	user	default	delete	delete
d'tor	default	default	default	default	user	default	default
move c'tor	default	default				user	
move assign	default	default					user

Moving on

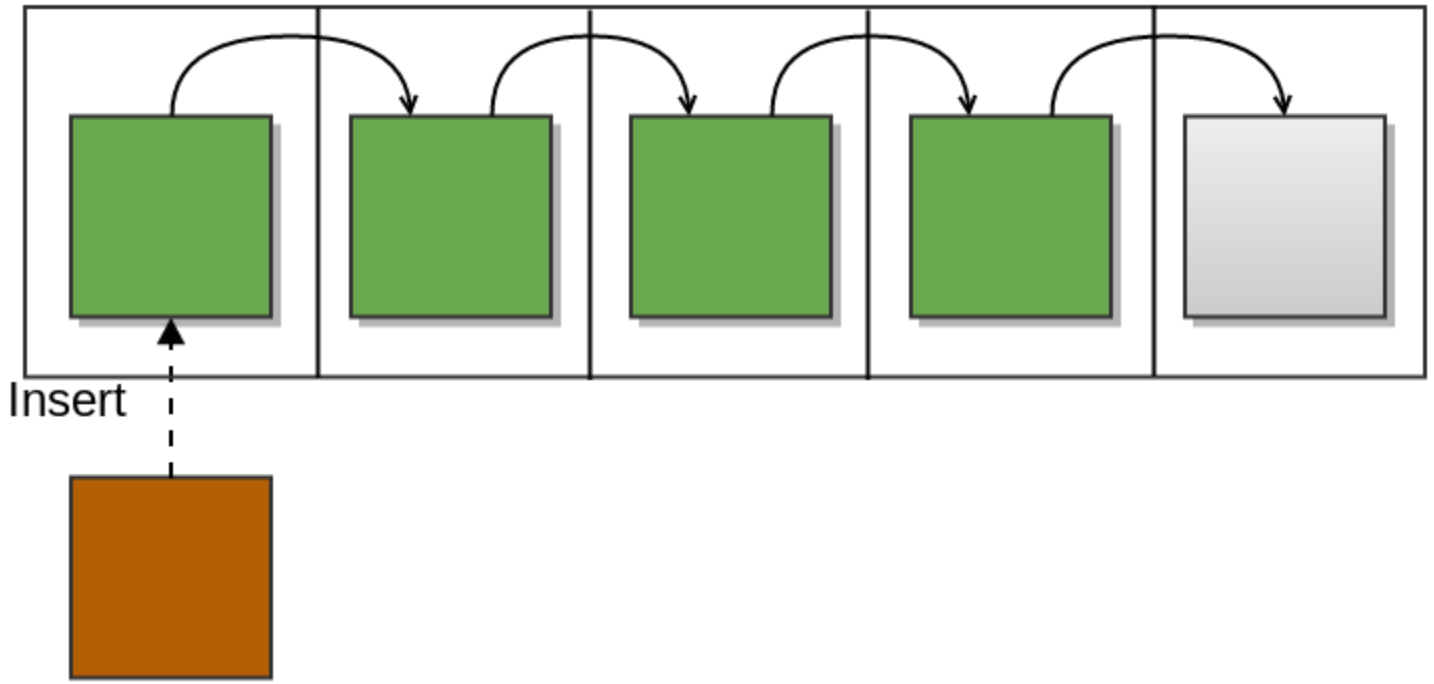
Moving on with:

- Move c’tor
- Move assignment

Motivation for Move

```
std::string s(' ', 1000);
std::vector<std::string> v(1000, s);
v.insert(v, v.begin());
```

Motivation for Move



Motivation for Move

This is ugly:

```
void get_circus(circular& cirque)
{
    // create a circus
}
circular c;
get_circus(c);
```

Motivation for Move

This makes sense in our value semantic language:

```
circular get_circus()
{
    circular cirque;
    // create a circus
    return cirque;
}
circular c = get_circus();
```

Motivation for Move

```
my_special_type o;
// manipulate and do things with o
// ..

// store for later use
storage.push_back(o);
```

Motivation for Move

What is it about copying in the previous examples that we don't like?

We want to reuse the guts from the source object to populate the destination object.

Motivation for Move

Optimization:

- ability to recognize the object is a temporary
- ability to indicate that the object is no longer needed... it is expiring

Move only types:

- name some

Motivation for Move

Bind to a rvalue:

```
foo(bar&& b); // rvalue reference
```

Motivation for Move

```
foo(bar&& b); // rvalue reference
foo(const bar& b); // lvalue reference

bar z;
foo(z);
```

Motivation for Move

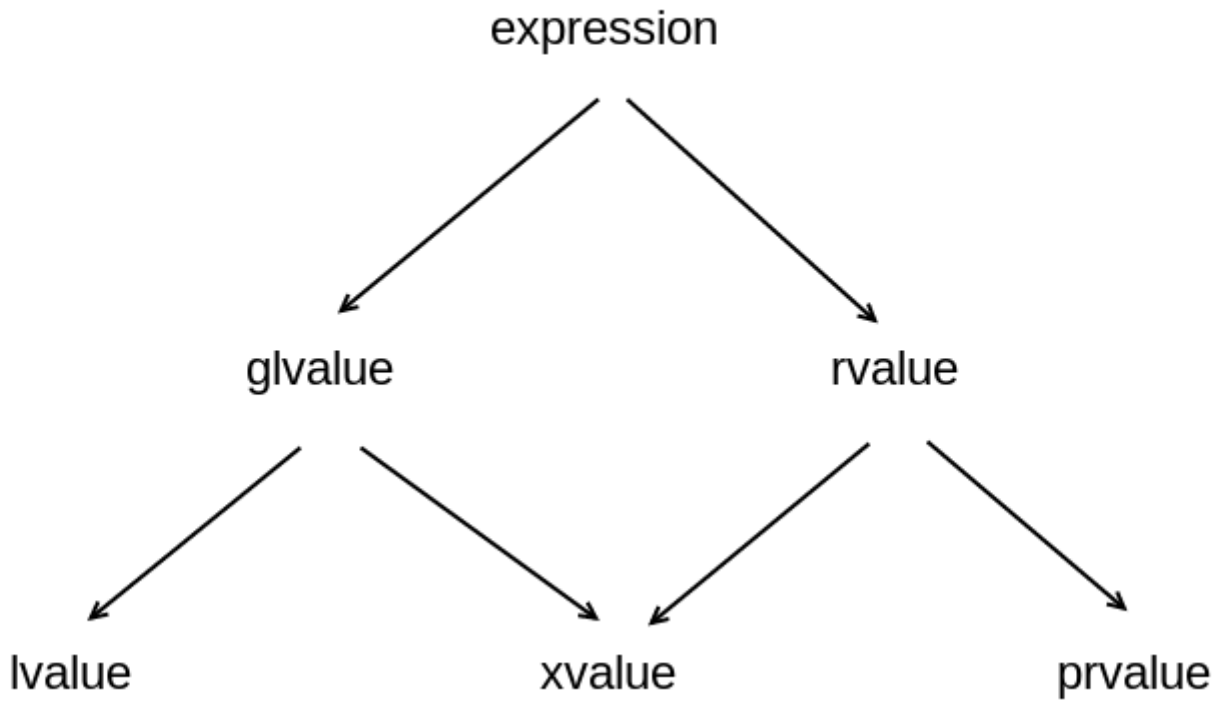
```
foo(bar&& b); // rvalue reference
foo(const bar& b); // lvalue reference

bar get_bar()
{
     bar b;
     // ..
     return b;
}

foo(get_bar());
```

Motivation for Move

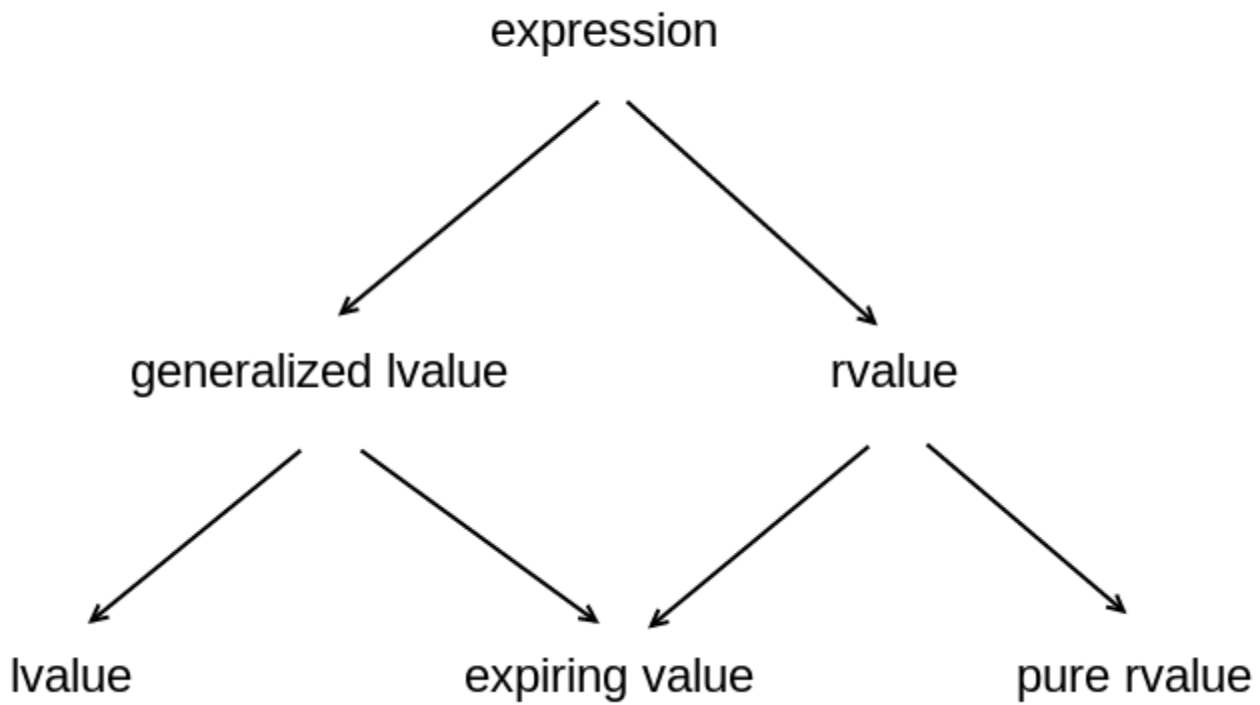
What about this?

```
my_special_type o;

// manipulate and do things with o
// ..

// store for later use
storage.push_back(o);
```

Motivation for Move



Motivation for Move



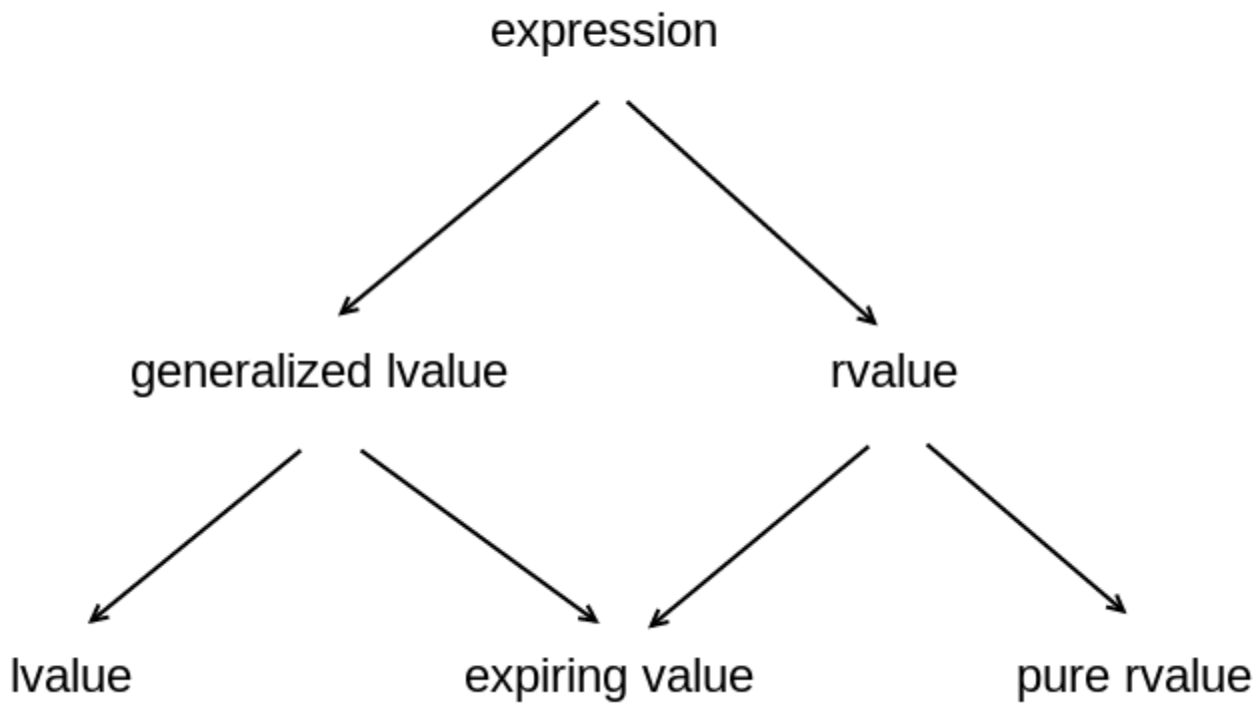
Motivation for Move

```
my_special_type o;

// manipulate and do things with o
// ..

// store for later use
storage.push_back(std::move(o));
```

What is `std::move`

```
template <typename T>
inline
T&& move(T&& x)
{
    return static_cast<T&&>(x);
}
```

Motivation for Move



Move Special Member functions

What do you notice about the copy declarations vs. the move declarations?

```
class circular
{
public:
     circular(size_t i=20);
     ~circular();
     circular(const circular& other);
     circular& operator=(const circular& rhs);
     circular(circular&& other);
     circular& operator=(circular&& rhs);
};
```

Note

“A move on the other hand leaves the source in a state defined differently for each type. The state of the source may be unchanged, or it may be radically different.

The only requirement is that the object remain in a self consistent state (all internal invariants are still intact).

From a client code point of view, choosing move instead of copy means that you don’t care what happens to the state of the source.”

Semantics of a Move

Ask yourself:

- What does it mean for your class to be moved?
- What are the post-move requirements on your class?

circular

```
class circular
{
public:
private:
    uint8_t *buffer, *head, *tail;
    size_t size;
};
```

circular

```
class circular
{
public:
	circular(circular&& other)
		: buffer(other.head),
		  head(other.head), tail(other.tail),
		  size(other.size)
	{ other.buffer = nullptr }
private:
	uint8_t *buffer, *head, *tail;
	size_t size;
};
```

circular

```
class circular
{
public:
      circular(circular&& other)
            : buffer(other.head),
              head(other.head), tail(other.tail),
              size(other.size)
      { other.buffer = nullptr }
private:
      uint8_t *buffer, *head, *tail;
      size_t size;
};
```

```
circular& operator=(circular&& rhs)
{
	if (&rhs != this)
	{
		delete[] buffer;
		size = rhs.size;
		buffer = rhs.buffer;
		head = rhs.head;
		tail = rhs.tail;

		rhs.buffer = nullptr;
		rhs.head = nullptr;
		rhs.tail = nullptr;
		rhs.size = 0U;
	}
	return *this;
}
```

```
circular& operator=(circular&& rhs)
{
    if (&rhs != this)
    {
        using std::swap;
        swap(buffer, rhs.buffer);
        swap(head, rhs.head);
        swap(tail, rhs.tail);
        swap(size, rhs.size);
    }
    return *this;
}
```

```
circular& operator=(circular&& rhs)
{
    using std::swap;
    swap(buffer, rhs.buffer);
    swap(head, rhs.head);
    swap(tail, rhs.tail);
    swap(size, rhs.size);
    return *this;
}
```

Move instrumented

```
circular amazing_stuff()
{
      circular circus;
      // ..
      return circus;
}
{
      std::cout << "-> start 1" << std::endl;
      circular a;
      a = amazing_stuff();
      std::cout << "<- end 1" << std::endl;
}
```

Move instrumented

```
circular amazing_stuff()
{
      circular circus;
      // ..
      return circus;
}
{
      std::cout << "-> start 1" << std::endl;
      circular a;
      a = amazing_stuff();
      std::cout << "<- end 1" << std::endl;
}
```

```
-> start 1
circular default constructor
circular default constructor
circular move assignment
circular destructor
<- end 1
circular destructor
```

Move instrumented

```
circular amazing_stuff()
{
     circular circus;
     // ..
     return circus;
}
{
     std::cout << "-> start 2" << std::endl;
     circular a = amazing_stuff();
     std::cout << "<- end 2" << std::endl;
}
```

Move instrumented

```
circular amazing_stuff()
{
     circular circus;
     // ..
     return circus;
}
{
     std::cout << "-> start 2" << std::endl;
     circular a = amazing_stuff();
     std::cout << "<- end 2" << std::endl;
}
```

```
-> start 2
circular default constructor
<- end 2
circular destructor
```

Standard 12.8 [31]

"When certain criteria are met, an implementation is allowed to omit the copy/move construction of a class object, even if the copy/move constructor and/or destructor for the objct has side effects.

In such cases, the implementation treats the source and target of the omitted copy/move operation as simply two different ways of referring to the same object..."

RVO/NRVO

What costs less than a move?

RVO/NRVO

Don't break **R**eturn **V**alue **O**ptimization

or

Named **R**eturn **V**alue **O**ptimization.

Breaking RVO

```
circular amazing_broken_stuff()
{
     circular circus;
     // ...
     return std::move(circus);
}
{
     std::cout << "-> start 3" << std::endl;
     circular a = amazing_broken_stuff();
     std::cout << "<- end 3" << std::endl;
}
```

```
-> start 3
circular default constructor
circular move constructor
circular destructor
<- end 3
circular destructor
```

Breaking RVO

```
circular broken_choice_stuff()
{
    circular soleil;
    circular ringling;
    bool wants_animal = false;
    // ...
    return wants_animal ? ringling : soleil;
}
```

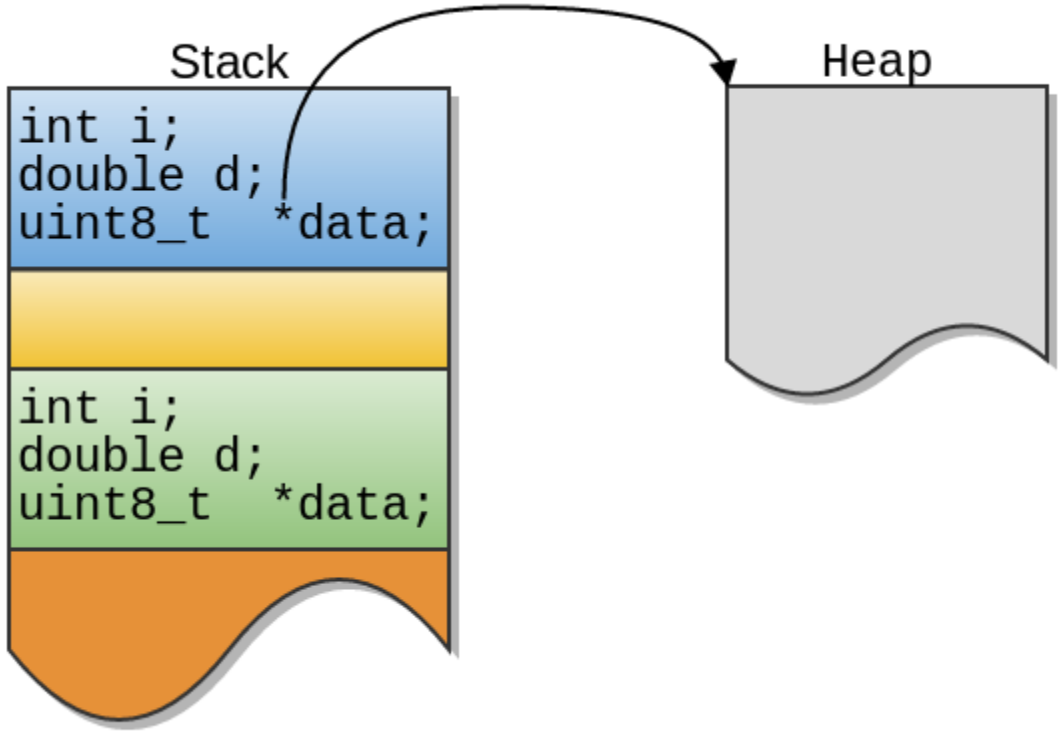
Breaking RVO

```
circular amazing_conversion_stuff()
{
    // ...
    return 42;
}
```

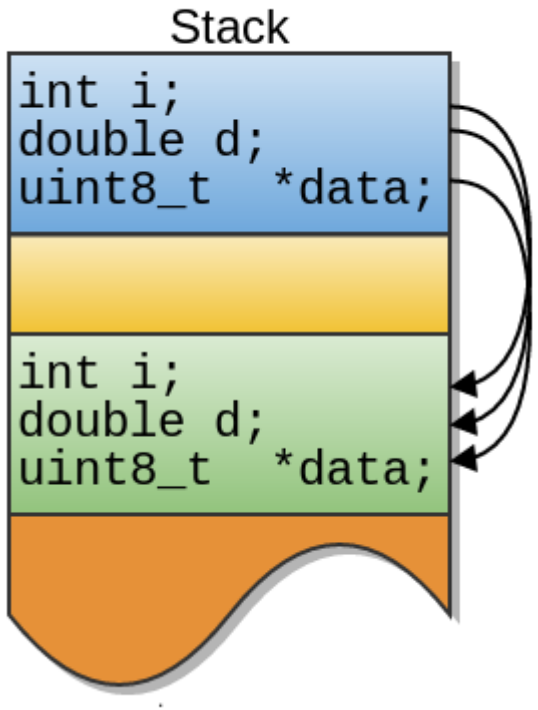
Move is not Magic

```
class circular
{
    // ...
private:
    uint8_t buffer[10];
    uint8_t *head, *tail;
}
```

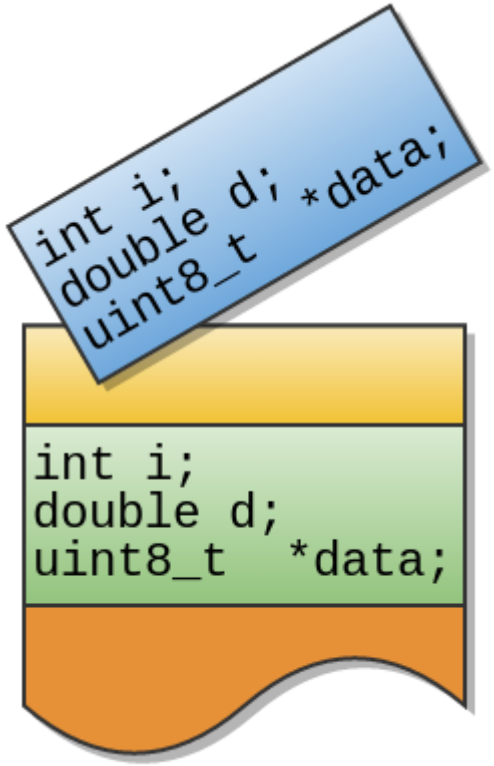
Move is not Magic



Move is not Magic



Move is not Magic



Move is not Magic

Move is copy if there are no externally managed resources.

`noexcept`

Some containers have *strong exception guarantee* for certain operations.

For example `std::vector::insert`

noexcept

```
circular(circular&& other) noexcept
    : buffer(other.buffer),
      head(other.head), tail(other.tail),
      size(other.size)
{
    other.buffer = nullptr;
}
```

noexcept

```
circular& operator=(circular&& rhs) noexcept
{
    using std::swap;
    swap(buffer, rhs.buffer);
    swap(head, rhs.head);
    swap(tail, rhs.tail);
    swap(size, rhs.size);
    return *this;
}
```

noexcept

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

swap

```
friend void swap(circular& a, circular& b)
{
    using std::swap;
    // ...
}
```

Perfect forwarding

```
class circular
{
public:
    circular(size_t i=20);
    circular(const circular& other);
    circular(circular&& other);
    template <typename T> circular(T&& t);
};
```

Perfect forwarding

```
template <typename T> circular(T&& t); // Uref
// or universal reference
```

Scott Meyers

```
class circular
{
public:
    circular(size_t i=20);
    circular(const circular& other);
    circular(circular&& other);
    template <typename T> circular(T&& t);
};
int i = 10;
circular c(i);
```

```
class circular
{
public:
    circular(size_t i=20);
    circular(const circular& other);
    circular(circular&& other);
    template <typename T> circular(T&& t);
};
int i = 10;
circular c(i);
```

universal

```
class circular
{
public:
    circular(size_t i=20);
    circular(const circular& other);
    circular(circular&& other);
    template <typename T> circular(T&& t);
};

circular c(8);
```

```
class circular
{
public:
    circular(size_t i=20);
    circular(const circular& other);
    circular(circular&& other);
    template <typename T> circular(T&& t);
};

circular c(8);
```

universal

```
class circular
{
public:
    circular(size_t i=20);
    circular(const circular& other);
    circular(circular&& other);
    template <typename T> circular(T&& t);
};

circular a;
circular b(a);
```

```
class circular
{
public:
    circular(size_t i=20);
    circular(const circular& other);
    circular(circular&& other);
    template <typename T> circular(T&& t);
};

circular a;
circular b(a);
```

default
universal

```
class circular
{
public:
    circular(size_t i=20);
    circular(const circular& other);
    circular(circular&& other);
    template <typename T> circular(T&& t);
};

circular a;
circular b = a;
```

```
class circular
{
public:
    circular(size_t i=20);
    circular(const circular& other);
    circular(circular&& other);
    template <typename T> circular(T&& t);
};

circular a;
circular b = a;
```

default
universal

Perfect forwarding

Takeaway:

- difficult in overloaded contexts (as with c’tor)
- binds to nearly EVERYTHING

Initializer list

```
class circular
{
public:
     circular(size_t i=20);
     circular(const circular& other);
     circular(circular&& other);
     circular(std::initializer_list<circular> cs);
};
```

```
class circular
{
public:
     circular(size_t i=20);
     circular(const circular& other);
     circular(circular&& other);
     circular(std::initializer_list<circular> cs);
};


int i = 10;
circular c{i};
```

```
class circular
{
public:
     circular(size_t i=20);
     circular(const circular& other);
     circular(circular&& other);
     circular(std::initializer_list<circular> cs);
};
```

```
int i = 10;
circular c{i};
```

default initializer

```
class circular
{
public:
     circular(size_t i=20);
     circular(const circular& other);
     circular(circular&& other);
     circular(std::initializer_list<circular> cs);
};



circular c{8};
```

```
class circular
{
public:
     circular(size_t i=20);
     circular(const circular& other);
     circular(circular&& other);
     circular(std::initializer_list<circular> cs);
};
```

```
circular c{8};
```

default initializer

```
class circular
{
public:
     circular(size_t i=20);
     circular(const circular& other);
     circular(circular&& other);
     circular(std::initializer_list<circular> cs);
};


circular a;
circular b{a};
```

```
class circular
{
public:
     circular(size_t i=20);
     circular(const circular& other);
     circular(circular&& other);
     circular(std::initializer_list<circular> cs);
};
```

```
circular a;
circular b{a};
```

default
copy
initializer

```
class circular
{
public:
     circular(size_t i=20);
     circular(const circular& other);
     circular(circular&& other);
     circular(std::initializer_list<circular> cs);
};


circular a;
circular b(a);
```

```
class circular
{
public:
     circular(size_t i=20);
     circular(const circular& other);
     circular(circular&& other);
     circular(std::initializer_list<circular> cs);
};


circular a;
circular b(a);
```

default
copy

```
class circular
{
public:
     circular(size_t i=20);
     circular(const circular& other);
     circular(circular&& other);
     circular(std::initializer_list<circular> cs);
};


circular a;
circular b = {a};
```

```
class circular
{
public:
     circular(size_t i=20);
     circular(const circular& other);
     circular(circular&& other);
     circular(std::initializer_list<circular> cs);
};
```

```
circular a;
circular b = {a};
```

default
copy
initializer

Initializer list

Takeaway:

- difficult to read with uniform initialization
- again, be careful and read about the implications

Observations

- We are through; what do we take home?

Observations – Man Made Rules

- Rule of 3
- Rule of 4/5
- Rule of Zero

Observations – Compiler Implicit Rules

User does

Compiler does

	nothing	c'tor	copy c'tor	copy assign	d'tor	move c'tor	move assign
default c'tor	default			default	default		default
copy c'tor	default	default	user	default	default	delete	delete
copy assign	default	default	default	user	default	delete	delete
d'tor	default	default	default	default	user	default	default
move c'tor	default	default				user	
move assign	default	default					user

Observations

- Rules...

Observations

- Be a programmer... Think!

What to do

- Give careful attention to each and every special member
  - Default c'tor
  - d'tor
  - Copy c'tor
  - Copy assignment
  - Move c'tor
  - Move assignment
- Consider the semantics
- Consider performance: runtime and "corporate"
- Avoid premature optimization

What to do

- Consider a customized `swap`
- Consider but don’t fret about `noexcept`
- Determine `move` guidelines for your team
- Prefer explicit handling of special members
- Think!