The Canonical Class

Understanding what goes into a C++11 Class

Michael Caisse

`mcaisse@ciere.com` | follow `@MichaelCaisse`

std::disclaimer

- The example code is 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

1. A Simpler World
   - Intro
   - Default
   - The Three
2. Moving on
   - Move Basics
3. Tidbits
   - Perfect Forwarding
4. Observations
   - Observations

ciere.com

The Question

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

The Question

What is the canonical form of a class?

ciere.com

What they Mean

Please provide us with a list of rules to follow.

Rules

Some proposals:

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

ciere.com

Rules

Some proposals:

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

Rules

Some proposals:

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

Rules

Some proposals:

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

Rules $:-/$

- I dislike fabricated rules

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

Rules :-/

- I dislike fabricated rules
- I don’t mind guidelines - if the brain is engaged
- The average team needs guidelines
- Rules/guidelines are not a replacement for mentoring via code review

Rules $:-/$

- I dislike fabricated rules
- I don't mind guidelines - if the brain is engaged
- The average team needs guidelines
- Rules/guidelines are not a replacement for mentoring via code review

Rules $: -/$

- I dislike fabricated rules
- I don’t mind guidelines - if the brain is engaged
- The average team needs guidelines
- Rules/guidelines are not a replacement for mentoring via code review

The Question

So, how should we write a simple class for C++11?

Canonical Class

In C++98:

- Default Constructor
- Copy Constructor
- Assignment Operator
- Destructor

Canonical Class

In C++11/14:

- Default Constructor
- Copy Constructor
- Copy Assignment
- Destructor
- Move Constructor
- Move Assignment

- Custom `swap`
- `initializer_list`
- `noexcept`
- `constexpr`

ciere.com

Canonical Class

In C++11/14:

- Default Constructor
- Copy Constructor
- Copy Assignment
- Destructor
- Move Constructor
- Move Assignment

- Custom **`swap`**
- **`initializer_list`**
- **`noexcept`**
- **`constexpr`**

Outline

1 A Simpler World
- Intro
- Default
- The Three

2 Moving on
- Move Basics

3 Tidbits
- Perfect Forwarding

4 Observations
- Observations

ciere.com

Default Constructor

What does it mean to default construct an object?

Default Constructor

What does it mean to default construct?

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

ciere.com

Default Constructor

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 Constructor

What does it mean to default construct?

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

ciere.com

Default Constructor

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 Constructor

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 Constructor

Is your type default constructable?

Default Constructor

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

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

Default Constructor

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

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

Default Constructor

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

ciere.com

Default Constructor

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

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

Default Constructor

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

Special Member Methods

- Implicit - Do nothing and take the compiler implementation
- Explicit - Define what you want
  - User supplied
  - Deleted
  - Default

Special Member Methods

- Implicit - Do nothing and take the compiler implementation
- Explicit - Define what you want
  - User supplied
  - Deleted
  - Default

Outline

1. A Simpler World
   - Intro
   - Default
   - The Three
2. Moving on
   - Move Basics
3. Tidbits
   - Perfect Forwarding
4. Observations
   - Observations

ciere.com

circular

```
class circular
{
public:
   circular(std::size_t i=20)
      : buffer( new uint8_t[i] )
      , head(buffer), tail(buffer)
   {}

   ~circular()
   {
      delete [] buffer;
   }

private:
   uint8_t *buffer, *head, *tail;
};
```

circular

```
circular a;
circular b = a;
```

```
class circular
{
public:
   circular(std::size_t i=20)
      : buffer( new uint8_t[i] )
      , head(buffer), tail(buffer)
   {}

   ~circular()
   {
      delete [] buffer;
   }

private:
   uint8_t *buffer, *head, *tail;
};
```

circular

```
circular a;
circular b = a;


class circular
{
public:
   circular(std::size_t i=20)
      : buffer( new uint8_t[i] )
      , head(buffer), tail(buffer)
   {}

   ~circular()
   {
      delete [] buffer;
   }

private:
   uint8_t *buffer, *head, *tail;
};
```

Whoops

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

Fixing circular - Remove destructor

```
class circular
{
public:
   circular(std::size_t i=20)
      : buffer( new uint8_t[i] )
      , head(buffer), tail(buffer)
   {}

private:
   uint8_t *buffer, *head, *tail;
};
```

Fixing circular - Delete Special Members

```
class circular
{
public:
   circular(std::size_t i=20)
      : buffer( new uint8_t[i] )
      , head(buffer), tail(buffer)
   {}

   ~circular()
   {
      delete [] buffer;
   }

   circular(circular const &) = delete;
   void operator=(circular const &) = delete;

private:
   uint8_t *buffer, *head, *tail;
};
```

Copy Constructor

What should it look like?

```
class circular
{
public:
   circular(std::size_t i=20)
      : buffer(new uint8_t[i])
      , head{buffer}, tail{buffer}, size{i}
   {}

   ~circular() { delete [] buffer; }

private:
   uint8_t *buffer, *head, *tail;
   std::size_t size;
};
```

Copy Constructor

```
class circular
{
public:
   circular(std::size_t i=20)
      : buffer(new uint8_t[i])
      , head{buffer}, tail{buffer}, size{i}
   {}

   ~circular() { delete [] buffer; }

   circular(circular const & rhs)
      : buffer{new uint8_t[rhs.size]}
      , head{0}, tail{0}, size(rhs.size)
   {
      std::copy(rhs.head,rhs.tail,buffer);
      head = buffer + (rhs.head - rhs.buffer);
      tail = buffer + (rhs.tail - rhs.buffer);
   }

private:
   uint8_t *buffer, *head, *tail;
   std::size_t size;
};
```

Copy Assignment

Why not the same as the Copy Constructor?

```
circular & operator=(circular const & rhs)
{
   if(&rhs != 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 did we need to add a copy constructor and assignment operator?

What in `circular` required it?

circular

```
class circular
{
public:
   circular(std::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 destructor implies changing state 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 constructor and/or assignment operator require a destructor?

circular array

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

circular array

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

circular array

```
class circular
{
public:
   circular() : head(buffer), tail(buffer)  {}

   circular(circular const & rhs)
   {
      assign(rhs);
   }

   circular & operator=(circular const & rhs)
   {
      if(&rhs != this) assign(rhs);
      return *this;
   }

private:
   inline void assign(circular const & 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

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	constr	copy constr	copy assign	destr	move constr	move assign
default constr	default			default	default		default
copy constr	default	default	user	default	default	delete	delete
copy assign	default	default	default	user	default	delete	delete
destr	default	default	default	default	user	default	default
move constr	default	default				user	
move assign	default	default					user

Outline

1 A Simpler World
- Intro
- Default
- The Three

2 Moving on
- Move Basics

3 Tidbits
- Perfect Forwarding

4 Observations
- Observations

ciere.com

Special Member Functions

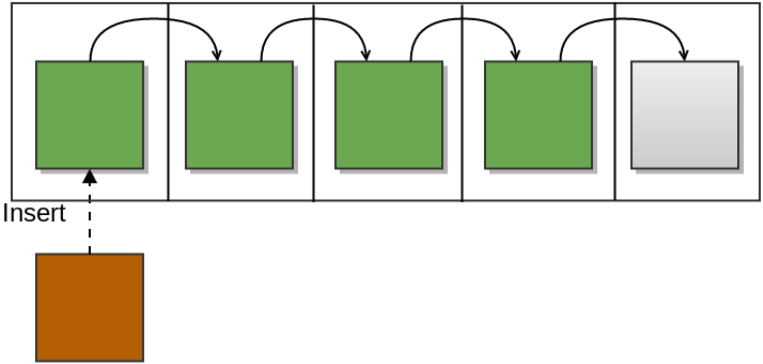
Moving on with:

- Move Constructor
- 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 cirque;
get_circus(cirque);
```

Motivation for Move

This makes sense in our value semantic language.

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

circular cirque = get_circus();
```

ciere.com

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

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(bar const & b);     // lvalue reference


bar z;
foo(z);
```

Motivation for Move

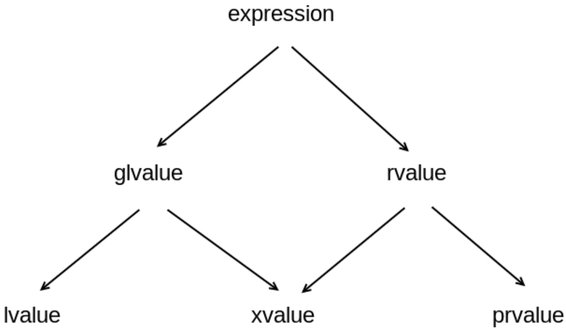
```
foo(bar && b);          // rvalue reference
foo(bar const & b);     // lvalue reference


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

foo(get_bar());
```

Motivation for Move

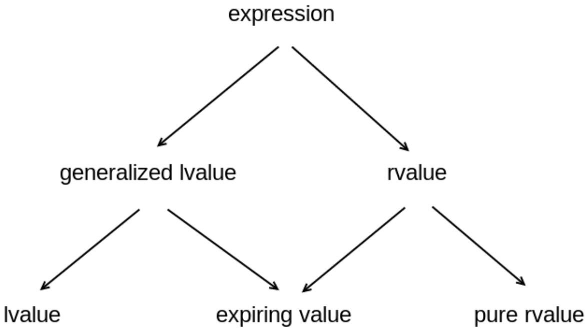
What about this?

```
my_special_type o;

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

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

Motivation for Move



Motivation for Move



expression

generalized lvalue

rvalue

lvalue

expiring value

pure rvalue

Motivation for Move

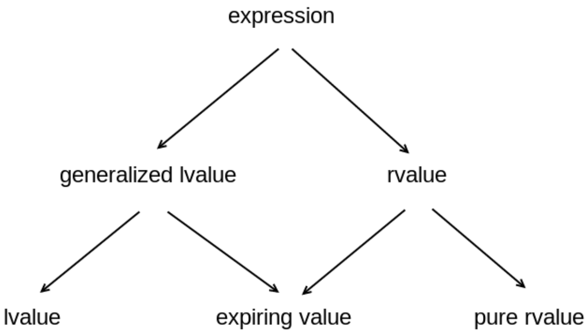
```
my_special_type o;

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

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

What is **std::move**

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

Motivation for Move

The **rvalue** from an **lvalue** is an **xvalue.**



Move Special Member functions

What do you notice about the copy declarations versus the move declarations?

```cpp
class circular
{
public:
   circular(std::size_t i=20);
   ~circular();

   circular(circular const & rhs);
   circular & operator=(circular const & rhs);

   circular(circular && rhs);
   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?

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;
   std::size_t size;
};
```

Move Constructor

```
class circular
{
public:

   circular(circular && rhs)

      : buffer(rhs.buffer)
      , head(rhs.head), tail(rhs.tail)
      , size(rhs.size)
   {
      rhs.buffer = nullptr;
   }

private:
   uint8_t *buffer, *head, *tail;
   std::size_t size;
};
```

Move Constructor

```
class circular
{
public:

   circular(circular && rhs)

      : buffer(rhs.buffer)
      , head(rhs.head), tail(rhs.tail)
      , size(rhs.size)
   {
      rhs.buffer = nullptr;
   }

private:
   uint8_t *buffer, *head, *tail;
   std::size_t size;
};
```

Move Assignment

```
class circular
{
public:
   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 = 0;
      }
      return *this;
   }
private:
   uint8_t *buffer, *head, *tail;
   std::size_t size;
};
```

Move Assignment

```
class circular
{
public:
   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 = 0;
      }
      return *this;
   }
private:
   uint8_t *buffer, *head, *tail;
   std::size_t size;
};
```

Move Assignment

```
class circular
{
public:

    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;

    }

private:
    uint8_t *buffer, *head, *tail;
    std::size_t size;
};
```

Move Assignment

```
class circular
{
public:

   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;

   }

private:
   uint8_t *buffer, *head, *tail;
   std::size_t size;
};
```

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;
}
```

Move Intrumented

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

```
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 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;
}
```

Move Instrumented

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

```cpp
circular amazing_stuff()
{
   circular circus;
   // ...
   return circus;
}


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

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 object have **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 ...**”

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 object have **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
Return **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;
}
```

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;
}
```

Breaking RVO

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

```
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;
}
```

Breaking RVO

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

Breaking RVO

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

Move is Assign?

Move our `circular` type with the array.

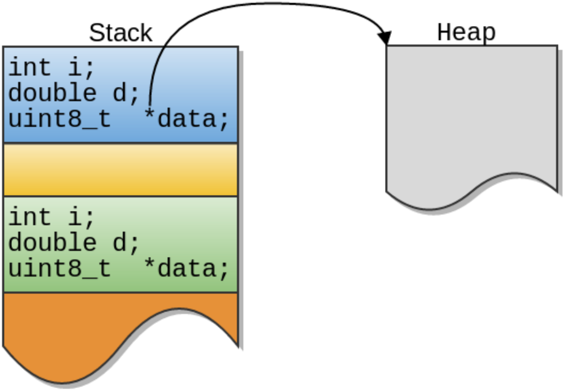
```
class circular
{

private:
   uint8_t buffer[10];
   uint8_t *head, *tail;
};
```

Move is not Magic

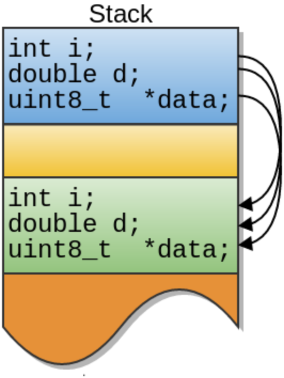
Move our `circular` type with the array.

```
class circular
{

private:
   uint8_t buffer[10];
   uint8_t *head, *tail;
};
```

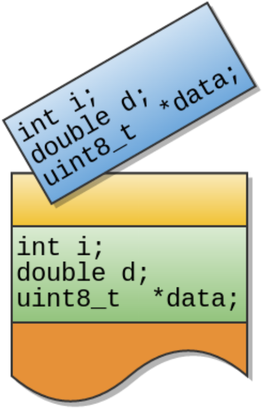
Move is not Magic



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 a *strong exception guarantee* for certain operations.

For example: `std::vector::insert`

noexcept

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

For example: **`std::vector::insert`**

noexcept

```
circular(circular && rhs)  noexcept
   : buffer(rhs.buffer)
   , head(rhs.head), tail(rhs.tail)
   , size(rhs.size)
{
   rhs.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 <class 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;
   //...
}
```

Outline

1. A Simpler World
   - Intro
   - Default
   - The Three
2. Moving on
   - Move Basics
3. Tidbits
   - Perfect Forwarding
4. Observations
   - Observations

ciere.com

Perfect Forwarding

```
class circular
{
public:

   circular(std::size_t i=20);
   circular(circular const & rhs);
   circular(circular && rhs);

   template <typename T>
   circular(T && rhs);

};
```

Perfect Forwarding

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

```
int i = 10;
circular c(i);
```

```
circular c(8);
```

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

```
circular a;
circular b = a;
```

Perfect Forwarding

```
int i = 10;
circular c(i);
```

universal

```
circular c(8);
```

universal

Perfect Forwarding

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

default
universal

```
circular a;
circular b = a;
```

default
universal

Initializer List

```
class circular
{
public:
   circular(std::size_t i=20);
   circular(circular const & rhs);
   circular(circular && rhs);

   circular(std::initializer_list<circular> rhs);

};
```

Initializer List

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

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

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

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

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

Initializer List

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

default
initializer

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

default
initializer

Initializer List

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

default
copy
initializer

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

default
copy

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

default
copy
initializer

Outline

1. A Simpler World
   - Intro
   - Default
   - The Three
2. Moving on
   - Move Basics
3. Tidbits
   - Perfect Forwarding
4. Observations
   - Observations

ciere.com

Rules

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

Rules

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

Rules

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

Compiler Implicit Rules

User does

Compiler does

	nothing	constr	copy constr	copy assign	destr	move constr	move assign
default constr	default			default	default		default
copy constr	default	default	user	default	default	delete	delete
copy assign	default	default	default	user	default	delete	delete
destr	default	default	default	default	user	default	default
move constr	default	default				user	
move assign	default	default					user

Rules

"Rules" ....

Rules

Be a programmer!

What to do

- Give careful attention to each and every special member
  - Default
  - Destructor
  - Copy Constructor
  - Copy Assignment
  - Move Constructor
  - Move Assignment
- Consider the Semantics!
- Consider performance : runtime and corporate

ciere.com

What to do

- Give careful attention to each and every special member
  - Default
  - Destructor
  - Copy Constructor
  - Copy Assignment
  - Move Constructor
  - Move Assignment
- Consider the Semantics!
- Consider performance : runtime and corporate

ciere.com

What to do

- Give careful attention to each and every special member
  - Default
  - Destructor
  - Copy Constructor
  - Copy Assignment
  - Move Constructor
  - Move Assignment
- Consider the Semantics!
- Consider performance : runtime and corporate

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!

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!

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!

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!

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!

Think