The Rule of Seven

(plus or minus two)

or,

The Canonical C++ Class

http://www.club.cc.cmu.edu/~ajo/disseminate/C++Now2015.pdf

The Rule of Zero

Write your classes in a way that you do not need to declare/define neither a destructor, nor a copy/move constructor or copy/move assignment operator.

— Peter Sommerlad, 2013

http://wiki.hsr.ch/PeterSommerlad/files/MeetingCPP2013_SimpleC++.pdf

What are the special member functions?

- Copy constructor
- Move constructor
- Copy assignment operator
- Move assignment operator
- Destructor

The Rule of Zero (SRP)

Classes that have custom destructors, copy/move constructors or copy/move assignment operators should deal exclusively with ownership.

Other classes should not have custom destructors, copy/move constructors or copy/move assignment operators.

— R. Martinho Fernandes

http://flamingdangerzone.com/cxx11/rule-of-zero/
http://accu.org/index.php/journals/1896

NOT special member functions

- Default constructor
- Other constructors
- Other assignment operators
- Member swap()
- Non-member swap()

NOT special member functions

- Default constructor
- Other constructors
- Other assignment operators
- Member swap()
- Non-member swap()

We'll talk about these too, but not quite yet.

Sample code for testing

```
#define CE constexpr // Clang doesn't like this, but GCC's okay with it
#define NE noexcept

struct Instruments {
    CE Instruments() NE { puts("Instruments default-ctor"); }
    CE Instruments(const Instruments&) NE { puts("Instruments copy-ctor"); }
    CE Instruments(Instruments&&) NE { puts("Instruments move-ctor"); }
    CE Instruments& operator=(const Instruments&) NE
        { puts("Instruments copy-assignment"); return *this; }
    CE Instruments& operator=(Instruments&&) NE
        { puts("Instruments move-assignment"); return *this; }
    ~Instruments() NE = default; // non-trivial dtor makes a class non-literal
};

class MyWidget : public Instruments { ... };
```

http://tiny.cc/Instrument

The default constructor

This is *not* a special member function.

Just to give you a taste of this talk's theme...

What are some things that could go wrong?

The default constructor

```
#include <string>

template<class T> struct Container {
    T t;
    Container() = default;
};

template<class T>
void UNIT_TEST_COPY_ASSIGNMENT() {
    T t;
    T const ct;
    t = ct;  // check this compiles
}

int main() {
    UNIT_TEST_COPY_ASSIGNMENT<Container<std::string>>();
}
```

http://www.open-std.org/jtc1/sc22/wg21/docs/cwg_active.html#253

The default constructor

```
#include <string>

template<class T> struct Container {
    T t;
    Container() = default;
};

template<class T>
void UNIT_TEST_COPY_ASSIGNMENT() {
    T t;
    T const ct;
    t = ct;  // check this compiles
}

int main() {
    UNIT_TEST_COPY_ASSIGNMENT<Container<int>>();
}
```

http://www.open-std.org/jtc1/sc22/wg21/docs/cwg_active.html#253

wat

```
prog.cc: In instantiation of 'void UNIT_TEST_COPY_ASSIGNMENT() [with T =
Container<int>]':
prog.cc:16:47:   required from here
prog.cc:11:13: error: uninitialized const 'ct' [-fpermissive]
     T const ct;
             ^
prog.cc:3:26: note: 'const struct Container<int>' has no user-provided
default constructor
 template<class T> struct Container {
                          ^
prog.cc:5:5: note: constructor is not user-provided because it is
explicitly defaulted in the class body
     Container() = default;
     ^
prog.cc:4:7: note: and the implicitly-defined constructor does not
initialize 'int Container<int>::t'
     T t;
       ^
```

http://www.open-std.org/jtc1/sc22/wg21/docs/cwg_active.html#253

Don’t =default your default ctor

> If a program calls for the default initialization of an object of a const-qualified type T,
> T shall be a class type with a user-provided default constructor. — *Draft Standard N4296 § 8.5 [dcl.init] ¶ 7*

- Defect Report 253 (July 11, 2000)
- Try not to nerf `const Widget cw;` for your users.

http://www.open-std.org/jtc1/sc22/wg21/docs/cwg_active.html#253

What about the destructor?

Supplying your own destructor *immediately* breaks the Rule of Zero in just about every possible way.

- No implicitly defined move ops
- Yes implicitly defined copy ops

  (but that's deprecated in C++14)

Puzzle: What does this print?

```
#include "Instrument.h"  // http://tiny.cc/Instrument
#include <vector>

struct Widget : public Instruments {
    std::vector<int> v;
    ~Widget() { assert(v.empty()); }
};

int main() {
    Widget w, w2;
    w = std::move(w2);  // what does this print?
}
```

Our move ops have disappeared!

```
#include "Instrument.h"  // http://tiny.cc/Instrument
#include <vector>

struct Widget : public Instruments {
    std::vector<int> v;
    ~Widget() { assert(v.empty()); }
};

int main() {
    Widget w, w2;
    w = std::move(w2);  // “Instruments copy-assignment”
}
```

Do I need a virtual destructor?

If A is intended to be used as a base class, and if callers should be able to destroy polymorphically, then make A::~A public and virtual.

Otherwise make it protected (and not virtual).

— Herb Sutter, C++ Coding Standards

BUT.

http://accu.org/index.php/journals/1896#[Sutter1]

Polymorphism and value semantics don’t play well together.

Polymorphism + value semantics

```
class Dog { ... };
class FoxTerrier : public Dog { ... };
class DireWolf : public Dog { ... };

void mixupAtTheDogPark(Dog& one, Dog& two)
{
    std::swap(one, two);  // value semantics, right?
}

int main() {
    auto asta = std::make_shared<FoxTerrier>();
    auto nymeria = std::make_shared<DireWolf>();
    mixupAtTheDogPark(*asta, *nymeria);
}
```

Not convinced yet?

It is unspecified whether subobjects representing virtual base classes are assigned more than once by the implicitly-defined copy/move assignment operator.

— Draft Standard N4296 § 12.8 [class.copy] ¶ 28

So don’t rely on those assignment operators.
Implement them yourself, or... just `=delete` them!

From here on, we assume *no inheritance hierarchy* is involved.

What about copy operations?

```
#include "Instrument.h"  // http://tiny.cc/Instrument
#include <vector>

struct Widget : public Instruments {
    std::vector<int> v;
    Widget() : v() {}  // the AP Comp Sci approach, circa 2001
    Widget(const Widget& rhs) : v(rhs.v) {}
    Widget& operator=(const Widget& rhs) { v = rhs.v; return *this; }
};

int main() {
    Widget w, w2;
    w = std::move(w2);  // what does this print?
}
```

What about copy operations?

```
#include "Instrument.h"  // http://tiny.cc/Instrument
#include <vector>

struct Widget : public Instruments {
    std::vector<int> v;
    Widget() = default;  // the modern equivalent
    Widget(const Widget&) = default;
    Widget& operator=(const Widget&) = default;
};

int main() {
    Widget w, w2;
    w = std::move(w2);  // what does this print?
}
```

What about copy operations?

```
#include "Instrument.h"  // http://tiny.cc/Instrument
#include <vector>

struct Widget : public Instruments {
    std::vector<int> v;
    Widget() = default;  // the modern equivalent
    Widget(const Widget&) = default;
    Widget& operator=(const Widget&) = default;
};

int main() {
    Widget w, w2;
    w = std::move(w2);  // “Instruments copy-assignment”
}
```

What about copy operations?

- Certainly don't implement the defaults yourself!
- Even explicitly `=default`'ing will trash your implicitly defined move ops.
- Explicitly `=delete`'ing will ***also*** trash the move ops: that's ***not*** how you make a "move-only" class!
- There are a lot of ways to make a class "copy-only." Beware.

https://groups.google.com/a/isocpp.org/forum/#!msg/std-proposals/HpxOPebDaMA/Q-Y4922iHXsJ

The implicit definitions are often correct!

- The implicitly defined (defaulted) copy constructor will copy-construct all the members.
- The implicitly defined (defaulted) copy assignment operator will copy-assign all the members.

What are some ways that could go wrong?

Exception-safety!

```
struct Name {
    std::string first, middle, last;
};

int main() {
    Name v { "Jerome", "K", "Jerome" };
    Name *n = &v;
    try {
        while (true) {
            n = new Name();  // n->first == n->last == ""
            *n = v;  // n->first == n->last == "Jerome"
        }
    } catch (std::bad_alloc) {
        assert(n->first == n->last);  // right?
    }
}
```

Exception-safety!

```
struct Name {
    std::string first, middle, last;

    Name(const Name& rhs) :
        first(rhs.first),
        middle(rhs.middle),
        last(rhs.last)
    {}              // The implicitly defaulted copy constructor is fine, but...

    Name& operator=(const Name& rhs) {
        first = rhs.first;
        middle = rhs.middle;  // What if this assignment throws std::bad_alloc?
        last = rhs.last;      // We lack the strong exception guarantee.
    }
};
```

http://c2.com/cgi/wiki?ExceptionGuarantee

Copy-and-swap idiom

```
struct Name {
    std::string first, middle, last;

    Name(const Name&) = default;

    Name& operator=(const Name& rhs) {
        Name temp = rhs;    // copy-construct first (might throw but won't leak)
        swap(*this, temp);  // swap second (cannot throw)
        return *this;
    }


    // We'll get to the move operations in a moment.
    Name(Name&&) = ...;
    Name& operator=(Name&&) = ...;
};
```

Copy-and-swap idiom

```
struct Name {
    std::string first, middle, last;

    Name(const Name&) = default;
    Name(Name&&) = default;

    Name& operator=(const Name& rhs) {
        Name temp = rhs;    // copy-construct first (might throw but won't leak)
        swap(*this, temp);  // swap second (cannot throw)
        return *this;
    }

    Name& operator=(Name&& rhs) {
        swap(*this, rhs);  // rhs is going to get destroyed anyway
        return *this;
    }
};
```

Copy-and-swap idiom

```
struct Name {
    std::string first, middle, last;

    Name(const Name&) = default;
    Name(Name&&) = default;

    Name& operator=(const Name& rhs) {
        Name temp = rhs;    // copy-construct first (might throw but won't leak)
        swap(*this, temp);  // swap second (cannot throw)
        return *this;
    }

    Name& operator=(Name&& rhs) {
        swap(*this, rhs);  // rhs is going to get destroyed anyway
        return *this;
    }
};
```

What could go wrong?

The funniest way for things to go wrong

```
using std::swap;

struct Name {
    std::string first, middle, last;
    Name(const Name&) = default;
    Name(Name&&) = default;
    Name& operator=(const Name& rhs) {
        Name temp = rhs;
        swap(*this, temp);
        return *this;
    }
    Name& operator=(Name&& rhs) {
        swap(*this, rhs);
        return *this;
    }
};

int main() {
    Name a, b;
    a = std::move(b);
}
```

http://melpon.org/wandbox/permlink/BUlkY1b12S4tlWXR

The funniest way for things to go wrong

```
using std::swap;

struct Name {
    std::string first, middle, last;
    Name(const Name&) = default;
    Name(Name&&) = default;
    Name& operator=(const Name& rhs) {
        Name temp = rhs;
        swap(*this, temp);
        return *this;
    }
    Name& operator=(Name&& rhs) {
        swap(*this, rhs);
        return *this;
    }
};

int main() {
    Name a, b;
    a = std::move(b);  // Segmentation fault
}
```

http://melpon.org/wandbox/permlink/BUlkY1b12S4tlWXR

What does `std::swap` do?

20.2.2 swap [utility.swap]

```
template<class T> void swap(T& a, T& b) noexcept(see below);
```

1 *Remark:* The expression inside `noexcept` is equivalent to:

```
is_nothrow_move_constructible<T>::value &&
is_nothrow_move_assignable<T>::value
```

2 *Requires:* Type `T` shall be `MoveConstructible` (Table 20) and `MoveAssignable` (Table 22).

3 *Effects:* Exchanges values stored in two locations.

This is everything N4296 has to say about the semantics of the `std::swap` template.

In practice:

```
template<class T>
void swap(T& a, T& b) noexcept(...)
{
    T t(std::move(a));
    a = std::move(b);
    b = std::move(t);
}
```

The funniest way for things to go wrong

```
using std::swap;  // Wrong! We'll show how to implement swap in a moment.

struct Name {
    std::string first, middle, last;
    Name(const Name&) = default;
    Name(Name&&) = default;
    Name& operator=(const Name& rhs) {
        Name temp = rhs;
        swap(*this, temp);
        return *this;
    }
    Name& operator=(Name&& rhs) {
        swap(*this, rhs);
        return *this;
    }
};

int main() {
    Name a, b;
    a = std::move(b);  // Move implemented as swap; swap implemented as move
}
```

http://melpon.org/wandbox/permlink/BUlkY1b12S4tIWXR

Let the compiler make the copy?

```
struct Name {
    std::string first, middle, last;
    Name(const Name&) = default;
    Name(Name&&) = default;

    // an atypical but valid copy-assignment operator
    Name& operator=(Name temp) {
        swap(*this, temp);
        return *this;
    }
};

Name X, L, R();

X = L;            // copy-construct temp from L; swap X with temp; destroy temp
X = std::move(L); // move-construct temp from L; swap X with temp; destroy temp
X = R();          // move-construct temp from R(); swap X with temp; destroy temp
```

http://definedbehavior.blogspot.com/2011/09/rule-of-3-4-swap.html

Let the compiler make the copy?

```
struct Name {
    std::string first, middle, last;
    Name(const Name&) = default;
    Name(Name&&) = default;

    // an atypical but valid copy-assignment operator
    Name& operator=(Name temp) {
        swap(*this, temp);
        return *this;
    }
};

Name X, L, R();

X = L;            // copy-construct temp from L; swap X with temp; destroy temp
X = std::move(L); // move-construct temp from L; swap X with temp; destroy temp
X = R();          // move-construct temp from R(); swap X with R();  destroy temp
```

http://definedbehavior.blogspot.com/2011/09/rule-of-3-4-swap.html

Yes, this trick is valid.

> A user-declared *copy* assignment operator `X::operator=` is a non-static non-template member function of class X with exactly one parameter of type `X`, `X&`, `const X&`, `volatile X&`, or `const volatile X&`.
>
> — *Draft Standard N4296 § 12.8 [class.copy] ¶ 17*

I'm not sold on it yet, but it's certainly interesting.

http://definedbehavior.blogspot.com/2011/09/rule-of-3-4-swap.html

So what do we do about `swap`?

- Specialize `std::swap<Widget>`?
- *Overload* `std::swap` for arguments of type `Widget`?
- Provide a free function `swap` located via ADL?
- Provide a member function `swap` (a la `std::vector`)?
- Something else?

Basically, what construct(s) do users ***expect*** will work?

```
using std::swap; swap(widget1, widget2); // 1,2,3
std::swap(widget1, widget2);             // 1,2
widget1.swap(widget2);                   // 4
```

Do not intrude on `namespace std`

> The behavior of a C++ program is undefined if it adds declarations or definitions to `namespace std` or to a namespace within `namespace std` unless otherwise specified. A program may add a template specialization for any standard library template to `namespace std` only if the declaration depends on a user-defined type and the specialization meets the standard library requirements for the original template and is not explicitly prohibited.
>
> *— Draft Standard N4296 § 17.6.4.2.1 [namespace.std] ¶ 1*

```
namespace std {
    // overload: this produces undefined behavior
    void swap(Widget& a, Widget& b) { ... }
};

namespace std {
    // specialization: this is allowed, but a Bad Idea
    template<> void swap(Widget& a, Widget& b) { ... }
};
```

http://www.gotw.ca/publications/mill17.htm

Specializing std::swap: a Bad Idea

```
class Widget;
namespace std {
    // specialization: this is allowed, but a Bad Idea
    template<> void swap(Widget& a, Widget& b);
};

template<class T> class Gadget;
namespace std {
    // partial specialization is impossible!
    template<class T> void swap(Gadget<T>& a, Gadget<T>& b);
};
```

http://www.gotw.ca/publications/mill17.htm

Provide an ADL swap (with a wrinkle)

```
class Name {
    std::string first, middle, last;

    // ...

    friend void swap(Name&, Name&);
};

void swap(Name& a, Name& b) {
    a.first.swap(b.first);
    a.middle.swap(b.middle);
    a.last.swap(b.last);
}
```

http://b.atch.se/posts/non-constant-constant-expressions/#friends

Provide an ADL swap (with a wrinkle)

```
class Name {
    std::string first, middle, last;


    // This version of the swap function is "hidden" from everyone except ADL.
    friend void swap(Name& a, Name& b) {
        a.first.swap(b.first);
        a.middle.swap(b.middle);
        a.last.swap(b.last);
    }
};

using std::swap; swap(widget1, widget2); // works
std::swap(widget1, widget2);             // essentially fails (does 3 moves)
widget1.swap(widget2);                   // not supported, which is fine
```

http://b.atch.se/posts/non-constant-constant-expressions/#friends

Rule of Zero asplode

```
template<typename Sequence>
void non_uniform_shuffle(Sequence& s) {
    for (auto& elt : s) {
        // pick a random element and swap with it
        std::swap(elt, s[rand() % s.size()]);  // or use ADL; there is no other swap in scope
    }
}

struct Person {
    std::vector<std::string> names;  // What could possibly go wrong?
};

int main() {
    std::vector<Person> vec = {
        {{"Alice", "Murphy"}},
        {{"Bob", "Dobbs"}},
        {{"Carl", "Carlson"}},
    };
    non_uniform_shuffle(vec);
}
```

http://tiny.cc/rule-of-zero-asplode

Rule of Zero asplode

```
% g++ prog.cc -Wall -Wextra -std=c++11 -D_GLIBCXX_DEBUG
% ./a.out

/usr/local/gcc-head/include/c++/6.0.0/debug/safe_container.h:86:error:
    attempt to self move assign.

Objects involved in the operation:
sequence "this" @ 0x0x1b90cc8 {
  type =
N11__gnu_debug15_Safe_containerINSt7__debug6vectorINSt7__cxx1112basic_stringIcSt11
char_traitsIcESaIcEEESaIS8_EEES9_NS_14_Safe_sequenceELb1EEE;
}

Aborted
```

http://tiny.cc/rule-of-zero-asplode

Self-move and self-swap

```
template<class T>
void swap(T& a, T& b) noexcept(...)
{
    T t(std::move(a));
    a = std::move(b);
    b = std::move(t);
}
```

Self-move and self-swap

```
template<class T>
void swap(T& x, T& x) noexcept(...)
{
    T t(std::move(x));
    x = std::move(x);
    x = std::move(t);
}
```

Self-move and self-swap

```
template<class T>
void swap(T& x, T& x) noexcept(...)
{
    T t(std::move(x));
    x = std::move(x);  // BOOM
    x = std::move(t);
}
```

`std` types don't support self-move

Each of the following applies to **all arguments** to functions defined in the C++ standard library, unless explicitly stated otherwise. ...

> — **If a function argument binds to an rvalue reference parameter, the implementation may assume that this parameter is a unique reference to this argument.**
> [*Note:* If the parameter is a generic parameter of the form T&& and an lvalue of type A is bound, the argument binds to an lvalue reference and thus is not covered by the previous sentence.]
> [*Note:* If a program casts an lvalue to an xvalue while passing that lvalue to a library function (e.g. by calling the function with the argument `move(x)`), the program is effectively asking that function to treat that lvalue as a temporary. **The implementation is free to optimize away aliasing checks** which might be needed if the argument was an lvalue.]
>
> *— Draft Standard N4296 § 17.6.4.9 [res.on.arguments] ¶ 1.3*

`vector::operator=(vector&&)` is a function defined in the C++ standard library.
One of its parameters is an rvalue reference parameter.
...Uh-oh.

Exceptions to the rule?

- **`std::unique_ptr`** (N4296 § 20.8.2.1.3 [util.smartptr.unique.assign] ¶ 6)
  "Transfers ownership as if by calling `reset(u.release())` followed by `get_deleter() = std::forward<D>(u.get_deleter())`."
- **`std::shared_ptr`** (N4296 § 20.8.2.2.3 [util.smartptr.shared.assign] ¶ 4)
  "Equivalent to `shared_ptr(std::move(r)).swap(*this)`."
- **`std::weak_ptr`** (N4296 § 20.8.2.3.3 [util.smartptr.weak.assign] ¶ 4)
  "Equivalent to `weak_ptr(std::move(r)).swap(*this)`."
- **`std::string`** (N4296 § 21.4.2 [string.cons] ¶ 22)
  "If `*this` and `str` are the same object, the member has no effect."

http://cplusplus.github.io/LWG/lwg-active.html#2063

Exceptions to the rule?

- **`std::unique_ptr`** (N4296 § 20.8.2.1.3 [util.smartptr.unique.assign] ¶ 6)
  "Transfers ownership as if by calling `reset(u.release())` followed by `get_deleter() = std::forward<D>(u.get_deleter())`."
- **`std::shared_ptr`** (N4296 § 20.8.2.2.3 [util.smartptr.shared.assign] ¶ 4)
  "Equivalent to `shared_ptr(std::move(r)).swap(*this)`."
- **`std::weak_ptr`** (N4296 § 20.8.2.3.3 [util.smartptr.weak.assign] ¶ 4)
  "Equivalent to `weak_ptr(std::move(r)).swap(*this)`."
- ~~**`std::string`** (N4296 § 21.4.2 [string.cons] ¶ 22)~~
  "If `*this` and `str` are the same object, the member has no effect."

DR 2063 "Contradictory requirements for string move assignment"
Resolved in Lenexa last week by ***removing § 21.4.2 [string.cons] ¶ 22***

http://cplusplus.github.io/LWG/lwg-active.html#2063

The default move-assignment operators cannot be trusted!

I call this the Rule of At Least One.

The default move-assignment operators cannot be trusted!

I call this the Rule of At Least ~~One~~ Two

The default move-assignment operators cannot be trusted!

I call this the Rule of At Least ~~One~~ ~~Two~~ Four

“Oh, but I provide an ADL swap.”

```
class Cat {
    std::vector<std::string> names;
    // let the move-assignment operator implicitly default
    friend void swap(Cat& a, Cat& b) {
        if (&a != &b) std::swap(a, b);
    }
};

Cat x;
swap(x,x);  // saved by ADL: ha, my code is foolproof!

x = std::move(x);  // “Oh, I would never write that.”
std::swap(x,x);    // “Nor that. I know my stuff.”
```

Have you seen code that makes *qualified* calls to `swap` in a template, like `std::swap( a, b );`? Congratulations, you have probably found a bug.

— *Eric Niebler, 2014*

http://ericniebler.com/2014/10/21/customization-point-design-in-c11-and-beyond/

...until, one day...

```
#include "Cat.h"

struct BagOfCats {
    Cat first, second;
};

// BagOfCats' implementor omits to write an ADL swap()

BagOfCats bag;
swap(bag,bag);  // ADL finds std::swap<BagOfCats>
                // and everything blows up
```

What we've come up with today

```
class Cat {
    std::vector<std::string> names; int lives;
  public:
    Cat() {}
    Cat(const Cat&) = default;
    Cat(Cat&&) = default;
    Cat& operator=(const Cat& rhs) {
        Cat temp = rhs;
        swap(*this, temp);
        return *this;
    }
    Cat& operator=(Cat&& rhs) { swap(*this, rhs); return *this; }

    friend void swap(Cat& a, Cat& b) {
        using std::swap; swap(a.names, b.names); swap(a.lives, b.lives);
    }
};
```

One slide on `constexpr`

GCC rejects constexpr member functions of any class with a non-trivial destructor

> The definition of a constexpr function shall satisfy the following constraints:
> — each of its parameter types shall be a literal type;
> ...
> For a non-template, non-defaulted constexpr function ... if no argument values exist such that an invocation of the function ... could be an evaluated subexpression of a core constant expression ... the program is ill-formed; no diagnostic required.
>
> *— Draft Standard N4296 § 7.1.5 [dcl.constexpr] ¶ 3.3 and ¶ 5*

Trivialness of destructor is very hard to enforce!

The one place noexcept matters

```
struct BadCat : Instruments {
    std::vector<std::string> names;
    BadCat() = default;
    BadCat(const BadCat&) = default;
    BadCat(BadCat&& rhs) : names(std::move(rhs.names)) {}
};

int main() {
    std::vector<BadCat> badcats;
    for (int i=0; i < 100; ++i) {
        badcats.emplace_back();  // watch for the resize
    }
}
```

http://tiny.cc/Instrument

```
template <class T> constexpr conditional_t<
  !is_nothrow_move_constructible<T>::value && is_copy_constructible<T>::value,
  const T&, T&&> move_if_noexcept(T& x) noexcept;
```

9 *Returns:* `std::move(x)`

— Draft Standard N4296 § 20.2.4 [forward] ¶ 9

Remarks: Causes reallocation if the new size is greater than the old capacity. If no reallocation happens, all the iterators and references before the insertion point remain valid. If an exception is thrown other than by the copy constructor, move constructor, assignment operator, or move assignment operator of T or by any `InputIterator` operation there are no effects. If an exception is thrown while inserting a single element at the end and T is `CopyInsertable` or `is_nothrow_move_constructible<T>::value` is `true`, there are no effects. Otherwise, if an exception is thrown by the move constructor of a non-`CopyInsertable` T, the effects are unspecified.

— Draft Standard N4296 § 23.3.6.5 [vector.modifiers] ¶ 1, talking about `push_back`

Questions?

Bonus material: `std::compare`

```
class Cat {
    ...

    friend void swap(Cat& a, Cat& b) { ... }
    ...

    friend int compare(const Cat& a, const Cat& b) { ... }

    friend bool operator<(const Cat& a, const Cat& b)  { return compare(a,b) < 0; }
    friend bool operator<=(const Cat& a, const Cat& b) { return compare(a,b) <= 0; }
    friend bool operator>(const Cat& a, const Cat& b)  { return compare(a,b) > 0; }
    friend bool operator>=(const Cat& a, const Cat& b) { return compare(a,b) >= 0; }
    friend bool operator==(const Cat& a, const Cat& b) { return compare(a,b) == 0; }
    friend bool operator!=(const Cat& a, const Cat& b) { return compare(a,b) != 0; }
};
```

http://www.club.cc.cmu.edu/~ajo/disseminate/ACCU_std_spaceship.pdf

Bonus material: the PIMPL pitfall

<<<Widget.h>>>

```
class Widget {
  public:
    Widget();


  private:
    struct Impl;
    unique_ptr<Impl> pImpl;
};
...
```

<<<Widget.cpp>>>

```
struct Widget::Impl {
    ...
};

Widget::Widget()
  : pImpl(make_unique<Impl>())
{}
```

```
Widget w; // OOPS
```

http://www.club.cc.cmu.edu/~ajo/disseminate/EMC++-Chapter4-SmartPointers.pdf

Bonus material: the PIMPL pitfall

<<<Widget.h>>>

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

  private:
    struct Impl;
    unique_ptr<Impl> pImpl;
};
...
```

<<<Widget.cpp>>>

```
struct Widget::Impl {
    ...
};

Widget::Widget()
  : pImpl(make_unique<Impl>())
{}

Widget::~Widget()
{}
```

```
Widget foo() { return Widget(); } // OOPS
```

Bonus material: the PIMPL pitfall

<<<Widget.h>>>

```
class Widget {
  public:
    Widget();
    ~Widget();
    Widget(Widget&&) = ...
  private:
    struct Impl;
    unique_ptr<Impl> pImpl;
};
...
```

<<<Widget.cpp>>>

```
struct Widget::Impl {
    ...
};

Widget::Widget()
  : pImpl(make_unique<Impl>())
{}

Widget::~Widget()
{}
```

The Rule of Five strikes again!