Cpp North 2025

On coding guidelines, class invariants, and special member functions

Olivia Wasalski

About me

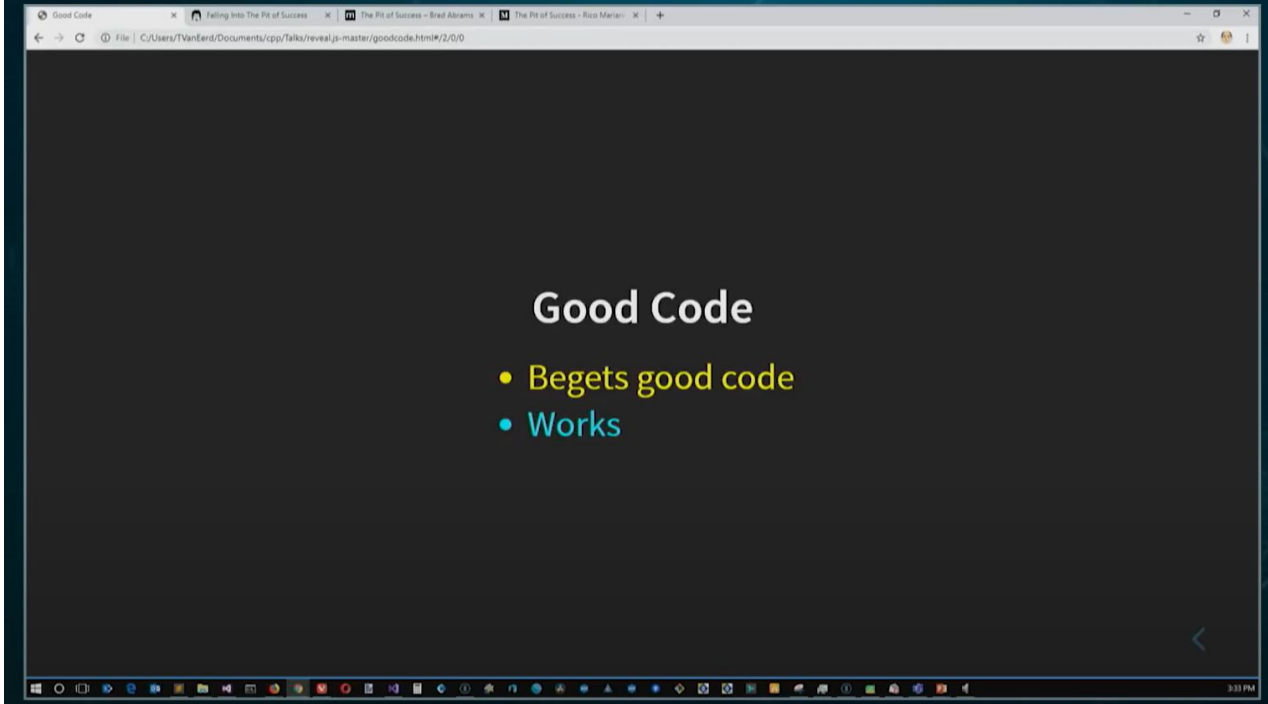
- Programming in C++ for 14 years
- Working at Safe Software for the last 10 years, specializing in computational geometry
- Worked with over a dozen new C++ developers

safe.com/careers

What is good code?

CppCon 2019: Tony Van Eerd Objects vs Values: Value Oriented Programming in an Object Oriented World

Cppcon The C++ Conference 2019 cppcon.org



Objects vs Values: Value Oriented Programming in an Object Oriented World

Video Sponsorship Provided By:

ansatz

This talk is about

- 5 specific coding guidelines
- All applied to the same simple example
- Explore the way special member functions interact with class invariants in C++
- The design of new two classes in the C++ standard library
- What guidelines I currently teach and why

Why this talk exists

- Are there commonly taught C++ guidelines that are incompatible in practice?
- What are the role of class invariants in C++?
- What guidelines should we teach?
- How do we write good code?

C++ Core Guideline I.10:

Use exceptions to signal a failure to perform a required task

I.10: Use exceptions to signal a failure to perform a required task

Reason

It should not be possible to ignore an error because that could leave the system or a computation in an undefined (or unexpected) state. This is a major source of errors.

Example

```
int printf(const char* ...);    // bad: return negative number if output fails

template<class F, class ...Args>
// good: throw system_error if unable to start the new thread
explicit thread(F&& f, Args&&... args);
```

Note

What is an error?

An error means that the function cannot achieve its advertised purpose (including establishing postconditions). Calling code that ignores an error could lead to wrong results or undefined systems state. For example, not being able to connect to a remote server is not by itself an error: the server can refuse a connection for all kinds of reasons, so the natural thing is to return a result that the caller should always check. However, if failing to make a connection is considered an error, then a failure should throw an exception.

```cpp
#include  <algorithm>

template <class T>
struct interval {
  T low;
  T high;
};

template <class T>
const T& clamp(const T& value, const interval<T>& interval) {
  return std::clamp(value, interval.low, interval.high);
}
```

https://godbolt.org/z/jTxx7cG47

std::clamp

Defined in header <algorithm>

```
template< class T >
constexpr const T& clamp( const T& v, const T& lo, const T& hi );
```
(1) (since C++17)

```
template< class T, class Compare >
constexpr const T& clamp( const T& v, const T& lo, const T& hi,
                          Compare comp );
```
(2) (since C++17)

If the value of `v` is within [`lo` , `hi`], returns `v` ; otherwise returns the nearest boundary.

1) Uses `operator<`(until C++20) `std::less{}` (since C++20) to compare the values.

   If T is not *LessThanComparable*, the behavior is undefined.[1]

2) Uses the comparison function `comp` to compare the values.

If `lo` is greater than `hi` , the behavior is undefined.

1. ↑ If NaN is avoided, T can be a floating-point type.

https://en.cppreference.com/w/cpp/algorithm/clamp.html

```cpp
#include <algorithm>
#include <stdexcept>

template <class T>
struct interval {
  T low;
  T high;
};

template <class T>
const T& clamp(const T& value, const interval<T>& interval) {
  if (!(interval.low <= interval.high)) {
    throw std::runtime_error("invalid interval");
  }
  return std::clamp(value, interval.low, interval.high);
}
```

https://godbolt.org/z/v7nKd1v1b

C++ Core Guideline C.2:

Use class if the class has an invariant; use struct if the data members can vary independently

C.2: Use `class` if the class has an invariant; use `struct` if the data members can vary independently

Reason

Readability. Ease of comprehension. The use of `class` alerts the programmer to the need for an invariant. This is a useful convention.

Note

An invariant is a logical condition for the members of an object that a constructor must establish for the public member functions to assume. After the invariant is established (typically by a constructor) every member function can be called for the object. An invariant can be stated informally (e.g., in a comment) or more formally using `Expects`.

If all data members can vary independently of each other, no invariant is possible.

In other words

A class invariant is a property that:

- Is established by the constructor
- Is assumed to be true at the start of each public member function
- Is re-established at the end of each public member function

```cpp
#include <algorithm>
#include <stdexcept>

template <class T>
struct interval {
  T low;
  T high;
};

template <class T>
const T& clamp(const T& value, const interval<T>& interval) {
  if (!(interval.low <= interval.high)) {
    throw std::runtime_error("invalid interval");
  }
  return std::clamp(value, interval.low, interval.high);
}
```

https://godbolt.org/z/v7nKd1v1b

```cpp
template <class T>
class interval {
private:
  T low_;
  T high_;

public:
  interval(T low, T high) : low_(std::move(low)), high_(std::move(high)) {
    if (!(low_ <= high_)) {
      throw std::runtime_error("invalid interval");
    }
  }

  const T& low() const { return low_; }
  const T& high() const { return high_; }
};

template <class T>
const T& clamp(const T& value, const interval<T>& interval) {
  return std::clamp(value, interval.low(), interval.high());
}
```

https://godbolt.org/z/68oveWWhn

C++ Core Guideline C.20:

If you can avoid defining default operations, do (a.k.a rule of zero)

master CppCoreGuidelines / CppCoreGuidelines.md ↑ Top

Preview Code Blame Raw

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

Reason

It's the simplest and gives the cleanest semantics.

Example

```
struct Named_map {
public:
    explicit Named_map(const string& n) : name(n) {}
    // no copy/move constructors
    // no copy/move assignment operators
    // no destructor
private:
    string name;
    map<int, int> rep;
};

Named_map nm("map"); // construct
Named_map nm2 {nm};  // copy construct
```

Since `std::map` and `string` have all the special functions, no further work is needed.

Note

This is known as "the rule of zero".

Special member functions

Add languages

Article Talk Read Edit View history Tools

From Wikipedia, the free encyclopedia

In the C++ programming language, **special member functions**[1] are functions which the compiler will automatically generate if they are used, but not declared explicitly by the programmer. The automatically generated special member functions are:

- Default constructor if no other constructor is explicitly declared.
- Copy constructor if no move constructor and move assignment operator are explicitly declared.

  If a destructor is declared generation of a copy constructor is deprecated (C++11, proposal N3242[2]).
- Move constructor if no copy constructor, copy assignment operator, move assignment operator and destructor are explicitly declared.
- Copy assignment operator if no move constructor and move assignment operator are explicitly declared.

  If a destructor is declared, generation of a copy assignment operator is deprecated.
- Move assignment operator if no copy constructor, copy assignment operator, move constructor and destructor are explicitly declared.
- Destructor

In these cases the compiler generated versions of these functions perform a *memberwise* operation. For example, the compiler generated destructor will destroy each sub-object (base class or member) of the object.

The compiler generated functions will be `public`, non-virtual[3] and the copy constructor and assignment operators will receive `const&` parameters (and not be of the alternative legal forms).[4]

https://en.wikipedia.org/wiki/Special_member_functions

“Rule of Zero”
by R. Martinho Fernandes

As an example, let’s write a class that owns a `HMODULE` resource from the Windows API. A `HMODULE` (a type alias for `void*`) is a handle to a loaded module, usually a DLL. It must be released by passing it to the `FreeLibrary` function.

One option would be to implement a class that follows the rule of three (or five).

```
class module {
public:
    explicit module(std::wstring const& name)
    : handle { ::LoadLibrary(name.c_str()) } {}

    // move constructor
    module(module&& that)
    : handle { that.handle } {
        that.handle = nullptr;
    }
    // copy constructor is implicitly forbidden due to user-defined move constructor

    // move assignment
    module& operator=(module&& that) {
        module copy { std::move(that) };
        std::swap(handle, copy.handle);
        return *this;
    }
    // copy assignment is implicitly forbidden due to user-defined move assignment

    // destructor
    ~module() {
        ::FreeLibrary(handle);
    }

    // other module related functions go here

private:
    HMODULE handle;
};
```

https://web.archive.org/web/20121127171954/http://rmartinho.github.com/cxx11/2012/08/15/rule-of-zero.html

“Rule of Zero”
by R. Martinho Fernandes

```
class module {
public:
    explicit module(std::wstring const& name)
    : handle { ::LoadLibrary(name.c_str()), &::FreeLibrary } {}

    // other module related functions go here

private:
    using module_handle = std::unique_ptr<void, decltype(&::FreeLibrary)>;

    module_handle handle;
};
```

Believe it or not, this does exactly the same thing as the first version, but all lifetime members have been implicitly defined by the compiler. This has several advantages:

- No duplication of the ownership logic;
- No mixing of ownership with other concerns;
- Less code, less opportunity for error; no code to review/change when the class changes;
- And the kicker is, we got exception safety *for free*.

https://web.archive.org/web/20121127171954/http://rmartinho.github.com/cxx11/2012/08/15/rule-of-zero.html

“Rule of Zero”
by R. Martinho Fernandes

So, we have arrived at the Rule of Zero (which is actually a particular instance of the *Single Responsibility Principle*):

> 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.

https://web.archive.org/web/20121127171954/http://rmartinho.github.com/cxx11/2012/08/15/rule-of-zero.html

```
5   template <class T>
6   class interval {
7   private:
8     T low_;
9     T high_;
10
11  public:
12    interval(T low, T high) : low_(std::move(low)), high_(std::move(high)) {
13      if (!(low_ <= high_)) {
14        throw std::runtime_error("invalid interval");
15      }
16    }
17
18    const T& low() const { return low_; }
19    const T& high() const { return high_; }
20  };
21
22  template <class T>
23  const T& clamp(const T& value, const interval<T>& interval) {
24    return std::clamp(value, interval.low(), interval.high());
25  }
```

https://godbolt.org/z/68oveWWhn

C++ Core Guideline C.64:

A move operation should move and leave its source in a valid state

master CppCoreGuidelines / CppCoreGuidelines.md ↑ Top

Preview Code Blame Raw

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

Reason

That is the generally assumed semantics. After `y = std::move(x)` the value of `y` should be the value `x` had and `x` should be in a valid state.

<example redacted to fit on slide>

Note

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

```
5   template <class T>
6   class interval {
7   private:
8     T low_;
9     T high_;
10
11  public:
12    interval(T low, T high) : low_(std::move(low)), high_(std::move(high)) {
13      if (!(low_ <= high_)) {
14        throw std::runtime_error("invalid interval");
15      }
16    }
17
18    const T& low() const { return low_; }
19    const T& high() const { return high_; }
20  };
21
22  template <class T>
23  const T& clamp(const T& value, const interval<T>& interval) {
24    return std::clamp(value, interval.low(), interval.high());
25  }
```

https://godbolt.org/z/68oveWWhn

```
23  int main() {
24    interval<std::string> a("apple", "banana clementine durian eggfruit fig");
25    interval<std::string> b(std::move(a));
26    assert(a.low() <= a.high());
27    return 0;
28  }
```

https://godbolt.org/z/878Gqx5K3

```
23  int main() {
24    interval<std::string> a("apple", "banana clementine durian eggfruit fig");
25    interval<std::string> b(std::move(a));
26    assert(a.low() <= a.high());
27    return 0;
28  }
```

Works! Tested with:

- gcc 15.01
- clang 20.1.0
- msvc 19.43 VX 17.13

https://godbolt.org/z/878Gqx5K3

```cpp
int main() {
  interval<std::pair<std::string, int>> a(
    {"apple", 5},
    {"banana clementine durian eggfruit fig", 2});

  interval<std::pair<std::string, int>> b(std::move(a));

  assert(a.low() <= a.high());
  return 0;
}
```

https://godbolt.org/z/YxfPschr9

```cpp
int main() {
  interval<std::pair<std::string, int>> a(
    {"apple", 5},
    {"banana clementine durian eggfruit fig", 2});

  interval<std::pair<std::string, int>> b(std::move(a));

  assert(a.low() <= a.high());
  return 0;
}
```

x86-64 gcc 15.1 | Compiler options...

```
Program returned: 139
Program stderr
output.s: /app/example.cpp:36: int main(): Assertion `a.low() <= a.high()' failed.
Program terminated with signal: SIGSEGV
```

https://godbolt.org/z/YxfPschr9

Option A

Add constraints to supported types and/or behaviour

```
5   template <class T>
6   class interval {
7   private:
8     T low_;
9     T high_;
10
11  public:
12    interval(T low, T high) : low_(std::move(low)), high_(std::move(high)) {
13      if (!(low_ <= high_)) {
14        throw std::runtime_error("invalid interval");
15      }
16    }
17
18    // Disable moves to ensure invariant is preseved
19    ~interval() = default;
20    interval(const interval&) = default;
21    interval& operator=(const interval&) = default;
22
23    const T& low() const { return low_; }
24    const T& high() const { return high_; }
25  };
```

https://godbolt.org/z/K4qsb69z8

```cpp
// Constrain T so that moved from objects are in valid states
template <class T> requires std::is_arithmetic_v<T>
class interval {
private:
  T low_;
  T high_;

public:
  interval(T low, T high) : low_(std::move(low)), high_(std::move(high)) {
    if (!(low_ <= high_)) {
      throw std::runtime_error("invalid interval");
    }
  }

  const T& low() const { return low_; }
  const T& high() const { return high_; }
};
```

https://godbolt.org/z/1qPzP3qna

```
5   // PRECONDITION: Moving from Ts must either leave the objects unchanged
6   // or must leave all Ts in the same empty state
7   template <class T>
8   class interval {
9   private:
10    T low_;
11    T high_;
12
13  public:
14    interval(T low, T high) : low_(std::move(low)), high_(std::move(high)) {
15      if (!(low_ <= high_)) {
16        throw std::runtime_error("invalid interval");
17      }
18    }
19
20    const T& low() const { return low_; }
21    const T& high() const { return high_; }
22  };
```

https://godbolt.org/z/cEsoMEhTc

Option B

Implement move operations by hand

```
18  // other.low_ and other.high_ will be replaced with default constructed
19  // values to maintain class invariant low_ <= high_
20  interval(interval&& other)
21    : low_(std::exchange(other.low_, {})),
22      high_(std::exchange(other.high_, {})) {
23  }
24
25  // other.low_ and other.high_ will be replaced with default constructed
26  // values to maintain class invariant low_ <= high_
27  interval& operator=(interval&& other) {
28    low_ = std::exchange(other.low_, {});
29    high_ = std::exchange(other.high_, {});
30    return *this;
31  }
```

https://godbolt.org/z/rn7MsPv6d

```
 7  template <class T>
 8  requires std::is_nothrow_default_constructible_v<T> &&
 9           std::is_nothrow_move_constructible_v<T> &&
10           std::is_nothrow_move_assignable_v<T>
11  class interval {
12  private:
13    T low_;
14    T high_;
15
16  public:
17    interval(T low, T high) : low_(std::move(low)), high_(std::move(high)) {…
21    }
22
23    // other.low_ and other.high_ will be replaced with default constructed
24    // values to maintain class invariant low_ <= high_
25    interval(interval&& other) noexcept…
28    }
29
30    // other.low_ and other.high_ will be replaced with default constructed
31    // values to maintain class invariant low_ <= high_
32    interval& operator=(interval&& other) noexcept {…
36    }
```

https://godbolt.org/z/7bjqMszEY

```cpp
template <class T>
requires std::is_nothrow_default_constructible_v<T> &&
         std::is_nothrow_move_constructible_v<T> &&
         std::is_nothrow_move_assignable_v<T>
class interval {
private:
  T low_;
  T high_;

public:
  interval(T low, T high) : low_(std::move(low)), high_(std::move(high)) {
    if (!(low_ <= high_)) {
      throw std::runtime_error("invalid interval");
    }
  }

  // other.low_ and other.high_ will be replaced with default constructed
  // values to maintain class invariant low_ <= high_
  interval(interval&& other) noexcept
    : low_(std::exchange(other.low_, {})),
      high_(std::exchange(other.high_, {})) {
  }

  // other.low_ and other.high_ will be replaced with default constructed
  // values to maintain class invariant low_ <= high_
  interval& operator=(interval&& other) noexcept {
    low_ = std::exchange(other.low_, {});
    high_ = std::exchange(other.high_, {});
    return *this;
  }

  ~interval() = default;
  interval(const interval&) = default;
  interval& operator=(const interval&) = default;

  const T& low() const { return low_; }
  const T& high() const { return high_; }
};

template <class T>
const T& clamp(const T& value, const interval<T>& interval) {
  return std::clamp(value, interval.low(), interval.high());
}
```

https://godbolt.org/z/7bjqMszEY

```
5   template <class T>
6   class interval {
7   private:
8     T low_;
9     T high_;
10
11  public:
12    interval(T low, T high) : low_(std::move(low)), high_(std::move(high)) {
13      if (!(low_ <= high_)) {
14        throw std::runtime_error("invalid interval");
15      }
16    }
17
18    const T& low() const { return low_; }
19    const T& high() const { return high_; }
20  };
21
22  template <class T>
23  const T& clamp(const T& value, const interval<T>& interval) {
24    return std::clamp(value, interval.low(), interval.high());
25  }
```

https://godbolt.org/z/68oveWWhn

https://godbolt.org/z/7bjqMszEY

```
6   template <class T>
7   requires std::is_nothrow_default_constructible_v<T> &&
8            std::is_nothrow_move_constructible_v<T> &&
9            std::is_nothrow_move_assignable_v<T>
10  class interval {
11  private:
12    T low_;
13    T high_;
14
15  public:
16    interval(T low, T high) : low_(std::move(low)), high_(std::move(high)) {
17      if (!(low_ <= high_)) {
18        throw std::runtime_error("invalid interval");
19      }
20    }
21
22    // other.low_ and other.high_ will be replaced with default constructed
23    // values to maintain class invariant low_ <= high_
24    interval(interval&& other) noexcept
25      : low_(std::exchange(other.low_, {})),
26        high_(std::exchange(other.high_, {})) {
27    }
28
29    // other.low_ and other.high_ will be replaced with default constructed
30    // values to maintain class invariant low_ <= high_
31    interval& operator=(interval&& other) noexcept {
32      low_ = std::exchange(other.low_, {});
33      high_ = std::exchange(other.high_, {});
34      return *this;
35    }
36
37    ~interval() = default;
38    interval(const interval&) = default;
39    interval& operator=(const interval&) = default;
40
41    const T& low() const { return low_; }
42    const T& high() const { return high_; }
43  };
44
45  template <class T>
46  const T& clamp(const T& value, const interval<T>& interval) {
47    return std::clamp(value, interval.low(), interval.high());
48  }
```

Option C

Weaken or abandon class invariant

```
 6   template <class T>
 7   class interval {
 8   private:
 9     bool valueless_by_move_;
10     T low_;
11     T high_;
12
13   public:
14     interval(T low, T high)
15       : valueless_by_move_(false),
16         low_(std::move(low)),
17         high_(std::move(high)) {
18       if (!(low_ <= high_)) {
19         throw std::runtime_error("invalid interval");
20       }
21     }
```

https://godbolt.org/z/v8MMdoEzc

```
42      // If !valueless_by_move(), then low() <= high() is always true
43      bool valueless_by_move() const { return valueless_by_move_; }
44      const T& low() const { return low_; }
45      const T& high() const { return high_; }
46  };
47
48  template <class T>
49  const T& clamp(const T& value, const interval<T>& interval) {
50      if (interval.valueless_by_move()) {
51          throw std::runtime_error("moved from interval");
52      }
53      return std::clamp(value, interval.low(), interval.high());
54  }
```

https://godbolt.org/z/v8MMdoEzc

```
23  interval(interval&& other) noexcept(std::is_nothrow_move_constructible_v<T>) :
24    valueless_by_move_(std::exchange(other.valueless_by_move_, true)),
25    low_(std::move(other.low_)),
26    high_(std::move(other.high_)) {
27  }
28
29  interval& operator=(interval&& other) noexcept(std::is_nothrow_move_assignable_v<T>) {
30    valueless_by_move_ = true;
31    const bool valueless_by_move = std::exchange(other.valueless_by_move_, true);
32    low_ = std::move(other.low_);
33    high_ = std::move(other.high_);
34    valueless_by_move_ = valueless_by_move;
35    return *this;
36  }
```

https://godbolt.org/z/v8MMdoEzc

```
23  interval(interval&& other) noexcept(std::is_nothrow_move_constructible_v<T>) : ···
27  }
28
29  interval& operator=(interval&& other) noexcept(std::is_nothrow_move_assignable_v<T>) {···
36  }
37
38  ~interval() = default;
39  interval(const interval&) = default;
40  interval& operator=(const interval&) = default;
```

https://godbolt.org/z/v8MMdoEzc

```cpp
template <class T>
class interval {
private:
  bool valueless_by_move_;
  T low_;
  T high_;

public:
  interval(T low, T high)
    : valueless_by_move_(false),
      low_(std::move(low)),
      high_(std::move(high)) {
    if (!(low_ <= high_)) {
      throw std::runtime_error("invalid interval");
    }
  }

  interval(interval&& other) noexcept(std::is_nothrow_move_constructible_v<T>) :
    valueless_by_move_(std::exchange(other.valueless_by_move_, true)),
    low_(std::move(other.low_)),
    high_(std::move(other.high_)) {
  }

  interval& operator=(interval&& other) noexcept(std::is_nothrow_move_assignable_v<T>) {
    valueless_by_move_ = true;
    const bool valueless_by_move = std::exchange(other.valueless_by_move_, true);
    low_ = std::move(other.low_);
    high_ = std::move(other.high_);
    valueless_by_move_ = valueless_by_move;
    return *this;
  }

  ~interval() = default;
  interval(const interval&) = default;
  interval& operator=(const interval&) = default;

  // If !valueless_by_move(), then low() <= high() is always true
  bool valueless_by_move() const { return valueless_by_move_; }
  const T& low() const { return low_; }
  const T& high() const { return high_; }
};

template <class T>
const T& clamp(const T& value, const interval<T>& interval) {
  if (interval.valueless_by_move()) {
      throw std::runtime_error("moved from interval");
  }
  return std::clamp(value, interval.low(), interval.high());
}
```

https://godbolt.org/z/v8MMdoEzc

```cpp
template <class T>
class interval {
private:
  T low_;
  T high_;

public:
  interval(T low, T high) : low_(std::move(low)), high_(std::move(high)) {
    if (!(low_ <= high_)) {
      throw std::runtime_error("invalid interval");
    }
  }

  const T& low() const { return low_; }
  const T& high() const { return high_; }
};

template <class T>
const T& clamp(const T& value, const interval<T>& interval) {
  return std::clamp(value, interval.low(), interval.high());
}
```

https://godbolt.org/z/68oveWWhn

https://godbolt.org/z/v8MMdoEzc

```cpp
template <class T>
class interval {
private:
  bool valueless_by_move_;
  T low_;
  T high_;

public:
  interval(T low, T high)
    : valueless_by_move_(false),
      low_(std::move(low)),
      high_(std::move(high)) {
    if (!(low_ <= high_)) {
      throw std::runtime_error("invalid interval");
    }
  }

  interval(interval&& other) noexcept(std::is_nothrow_move_constructible_v<T>) :
    valueless_by_move_(std::exchange(other.valueless_by_move_, true)),
    low_(std::move(other.low_)),
    high_(std::move(other.high_)) {
  }

  interval& operator=(interval&& other) noexcept(std::is_nothrow_move_assignable_v<T>) {
    valueless_by_move_ = true;
    const bool valueless_by_move = std::exchange(other.valueless_by_move_, true);
    low_ = std::move(other.low_);
    high_ = std::move(other.high_);
    valueless_by_move_ = valueless_by_move;
    return *this;
  }

  ~interval() = default;
  interval(const interval&) = default;
  interval& operator=(const interval&) = default;

  // If !valueless_by_move(), then low() <= high() is always true
  bool valueless_by_move() const { return valueless_by_move_; }
  const T& low() const { return low_; }
  const T& high() const { return high_; }
};

template <class T>
const T& clamp(const T& value, const interval<T>& interval) {
  if (interval.valueless_by_move()) {
    throw std::runtime_error("moved from interval");
  }
  return std::clamp(value, interval.low(), interval.high());
}
```

<C++ standard library aside>

std::indirect from C++26

std::indirect

Defined in header <memory>

```
template< class T, class Allocator = std::allocator<T> >
class indirect;
```
(1) (since C++26)

```
namespace pmr {
    template< class T >
    using indirect = std::indirect<T, std::pmr::polymorphic_allocator<T>>;
}
```
(2) (since C++26)

1) `std::indirect` is a wrapper containing dynamically-allocated object with value-like semantics.
2) `std::pmr::indirect` is an alias template that uses a polymorphic allocator.

An `std::indirect` object manages the lifetime of an owned object. An `std::indirect` object can only have no owned object after it has been moved from, in this case it is *valueless*.

Every object of type `std::indirect<T, Allocator>` uses an object of type `Allocator` to allocate and free storage for the owned object as needed.

If a program declares an explicit or partial specialization of `std::indirect`, the behavior is undefined.

https://en.cppreference.com/w/cpp/memory/indirect.html

std::indirect<T, Allocator>::valueless_after_move

```
constexpr bool valueless_after_move() const noexcept;
```
(since C++26)

Checks whether `*this` is valueless.

Return value

`true` if `*this` is valueless, otherwise `false`.

https://en.cppreference.com/w/cpp/memory/indirect/valueless_after_move.html

std::indirect<T, Allocator>::operator->, std::indirect<T, Allocator>::operator*

`constexpr const_pointer operator->() const noexcept;`	(1)	(since C++26)
`constexpr pointer operator->() noexcept;`	(2)	(since C++26)
`constexpr const T& operator*() const& noexcept;`	(3)	(since C++26)
`constexpr T& operator*() & noexcept;`	(4)	(since C++26)
`constexpr const T&& operator*() const&& noexcept;`	(5)	(since C++26)
`constexpr T&& operator*() && noexcept;`	(6)	(since C++26)

Accesses the owned value.

1,2) Returns a pointer to the owned value.

3-6) Returns a reference to the owned value.

If `*this` is valueless, the behavior is undefined.

https://en.cppreference.com/w/cpp/memory/indirect/operator%252A.html

</C++ standard library aside>

```cpp
template <class T>
class interval {
private:
  T low_;
  T high_;

public:
  interval(T low, T high) : low_(std::move(low)), high_(std::move(high)) {
    if (!(low_ <= high_)) {
      throw std::runtime_error("invalid interval");
    }
  }

  const T& low() const { return low_; }
  const T& high() const { return high_; }
};

template <class T>
const T& clamp(const T& value, const interval<T>& interval) {
  if (!(interval.low() <= interval.high())) {
    throw std::runtime_error("invalid interval");
  }
  return std::clamp(value, interval.low(), interval.high());
}
```

https://godbolt.org/z/MPef5bGqz

None of these options are ideal

I could see an argument for using each of these in different contexts

- Option A: Add constraints to supported types and/or behaviour
- Option B: Implement move operations by hand (and still add type constraints)
- Option C: Weaken or abandon class invariant

Let’s assume that this is our context

- We want to have a class invariant
- We are willing to constrain supported types (we want to use arithmetic types and strings)
- We want the requirements on supported types to be checked with the type system
- We want our move operations to be reasonably efficient

- Move operations may do unnecessary writes - hopefully the optimizer will help
- Lost the rule of zero - but maybe managing the invariant is the single responsibility of this class?

```cpp
template <class T>
requires std::is_nothrow_default_constructible_v<T> &&
         std::is_nothrow_move_constructible_v<T> &&
         std::is_nothrow_move_assignable_v<T>
class interval {
private:
  T low_;
  T high_;

public:
  interval(T low, T high) : low_(std::move(low)), high_(std::move(high)) {
    if (!(low_ <= high_)) {
      throw std::runtime_error("invalid interval");
    }
  }

  // other.low_ and other.high_ will be replaced with default constructed
  // values to maintain class invariant low_ <= high_
  interval(interval&& other) noexcept
    : low_(std::exchange(other.low_, {})),
      high_(std::exchange(other.high_, {})) {
  }

  // other.low_ and other.high_ will be replaced with default constructed
  // values to maintain class invariant low_ <= high_
  interval& operator=(interval&& other) noexcept {
    low_ = std::exchange(other.low_, {});
    high_ = std::exchange(other.high_, {});
    return *this;
  }

  ~interval() = default;
  interval(const interval&) = default;
  interval& operator=(const interval&) = default;

  const T& low() const { return low_; }
  const T& high() const { return high_; }
};

template <class T>
const T& clamp(const T& value, const interval<T>& interval) {
  return std::clamp(value, interval.low(), interval.high());
}
```

https://godbolt.org/z/7bjqMszEY

Key takeaway

The “valid but unspecified state” is not composable

Always provide at least the basic exception guarantee

Exception safety

After the error condition is reported by a function, additional guarantees may be provided with regards to the state of the program. The following four levels of exception guarantee are generally recognized[4][5][6], which are strict supersets of each other:

1. *Nothrow (or nofail) exception guarantee* — the function never throws exceptions. Nothrow (errors are reported by other means or concealed) is expected of destructors and other functions that may be called during stack unwinding. The destructors are `noexcept` by default.(since C++11) Nofail (the function always succeeds) is expected of swaps, move constructors, and other functions used by those that provide strong exception guarantee.
2. *Strong exception guarantee* — If the function throws an exception, the state of the program is rolled back to the state just before the function call (for example, `std::vector::push_back`).
3. *Basic exception guarantee* — If the function throws an exception, the program is in a valid state. No resources are leaked, and all objects' invariants are intact.
4. *No exception guarantee* — If the function throws an exception, the program may not be in a valid state: resource leaks, memory corruption, or other invariant-destroying errors may have occurred.

4. ↑ B. Stroustrup (2000), "The C++ Programming Language" Appendix E
5. ↑ H. Sutter (2000) "Exceptional C++"
6. ↑ D. Abrahams (2001) "Exception Safety in Generic Components"

https://en.cppreference.com/w/cpp/language/exceptions.html

In Each Function, Give the Strongest Safety Guarantee that Won't Penalize Callers Who Don't Need It, But Always Give at Least the Basic Guarantee

The summary:

- Ensure that errors always leave your program in a valid state. This is the basic guarantee. Beware of invariant-destroying errors (including, but not limited to, leaks), which are just plain bugs.
- Prefer to additionally guarantee that the final state is either the original state (if there was an error the operation was rolled back) or the intended target state (if there was no error the operation was committed). This is the strong guarantee.
- Prefer to additionally guarantee that the operation can never fail at all. Although this is not possible for most functions, it is required for certain functions like destructors, deallocation functions, and swap functions. This is the nofail guarantee.

"When and How to Use Exceptions", Herb Sutter, August 2004

https://www.drdobbs.com/when-and-how-to-use-exceptions/184401836

That's it; there is no lower level. A failure to meet at least the basic guarantee is always a program bug. Correct programs meet at least the basic guarantee for all functions; even those rare correct programs that deliberately leak resources by design, particularly in situations where the program immediately aborts, do so knowing that they will be reclaimed by the operating system. Always structure code so that resources are correctly freed and data is in a consistent state even in the presence of errors (unless the error is so severe that graceful or ungraceful termination is the only option).

“When and How to Use Exceptions”, Herb Sutter, August 2004

https://www.drdobbs.com/when-and-how-to-use-exceptions/184401836

```
7   template <class T>
8   requires std::is_nothrow_default_constructible_v<T> &&
9            std::is_nothrow_move_constructible_v<T> &&
10           std::is_nothrow_move_assignable_v<T>
11  class interval {
12  private:
13    T low_;
14    T high_;
15
16  public:
17    interval(T low, T high) : low_(std::move(low)), high_(std::move(high)) {
18      if (!(low_ <= high_)) {
19        throw std::runtime_error("invalid interval");
20      }
21    }
```

https://godbolt.org/z/7bjqMszEY

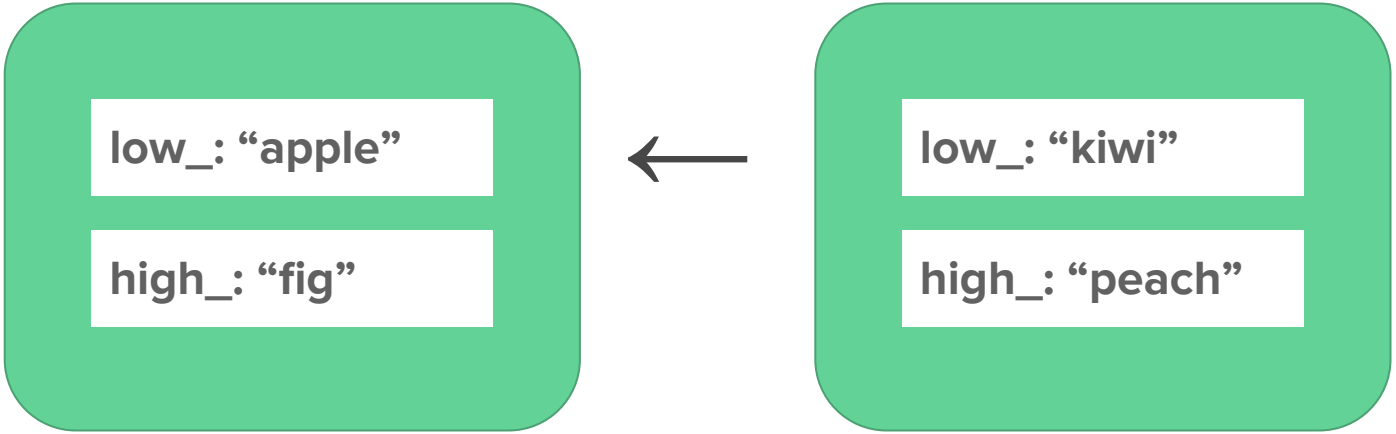
```
23    // other.low_ and other.high_ will be replaced with default constructed
24    // values to maintain class invariant low_ <= high_
25    interval(interval&& other) noexcept
26      : low_(std::exchange(other.low_, {})),
27        high_(std::exchange(other.high_, {})) {
28    }
29
30    // other.low_ and other.high_ will be replaced with default constructed
31    // values to maintain class invariant low_ <= high_
32    interval& operator=(interval&& other) noexcept {
33      low_ = std::exchange(other.low_, {});
34      high_ = std::exchange(other.high_, {});
35      return *this;
36    }
```

https://godbolt.org/z/7bjqMszEY

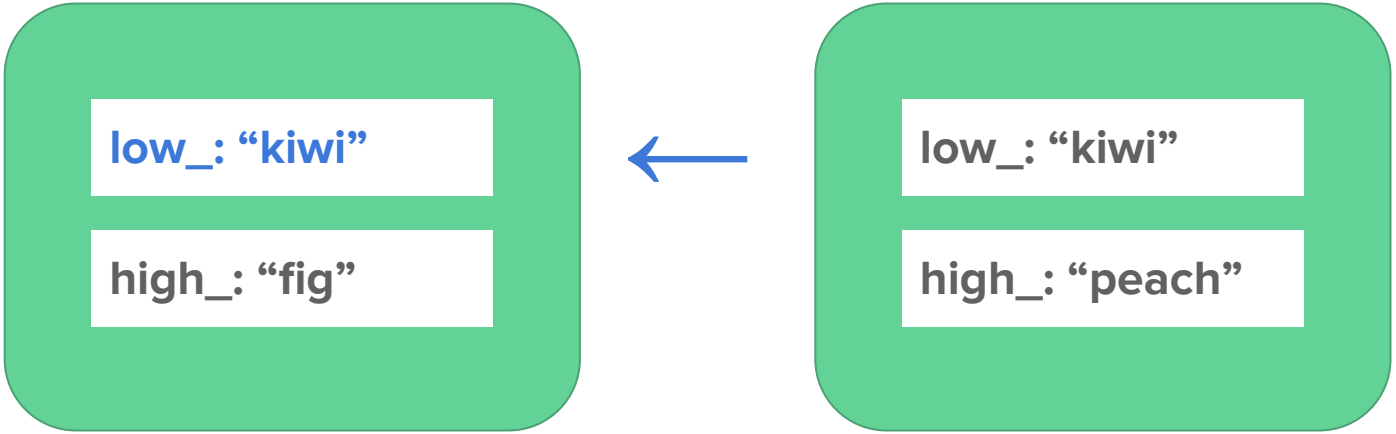
```
38      ~interval() = default;
39      interval(const interval&) = default;
40      interval& operator=(const interval&) = default;
41
42      const T& low() const { return low_; }
43      const T& high() const { return high_; }
44  };
```

https://godbolt.org/z/7bjqMszEY

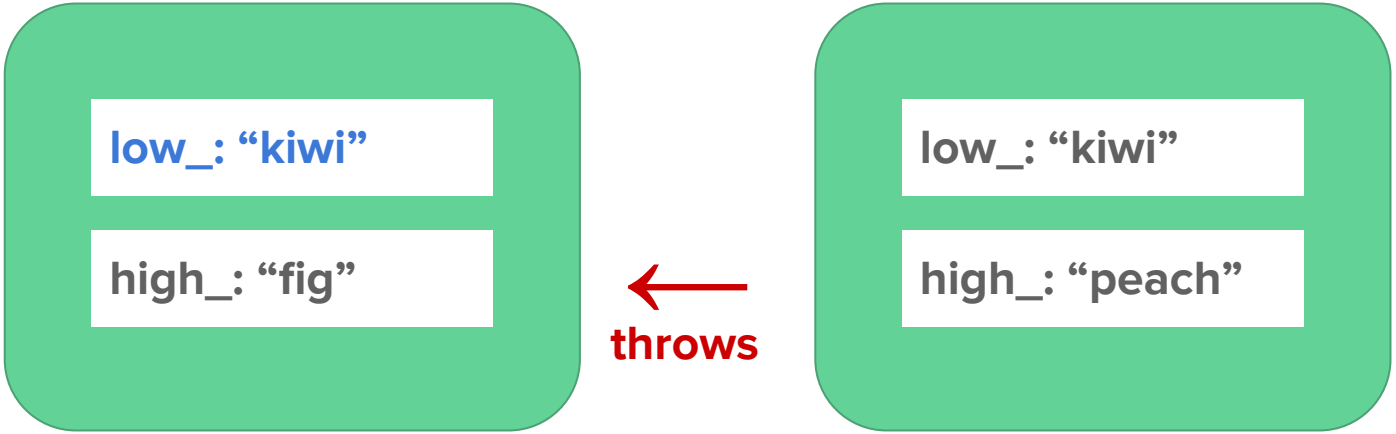
```
// assume there's no short string optimization
interval<std::string> a{"apple", "fig"};
interval<std::string> b{"kiwi", "peach"};
a = b; // throws during copy of "peach"
```



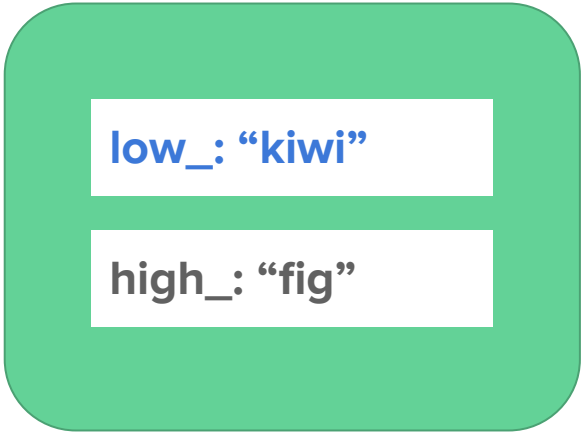
```
// assume there's no short string optimization
interval<std::string> a{"apple", "fig"};
interval<std::string> b{"kiwi", "peach"};
a = b; // throws during copy of "peach"
```

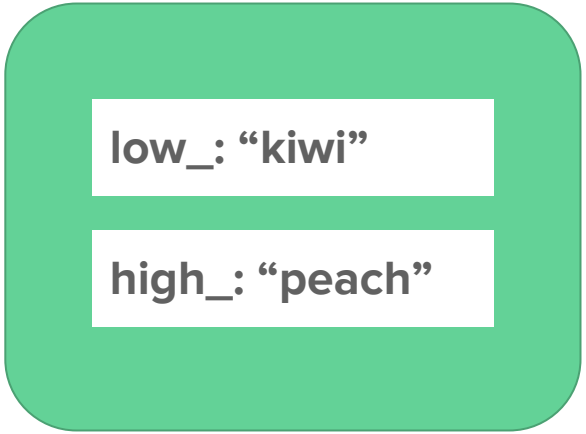


```
// assume there's no short string optimization
interval<std::string> a{"apple", "fig"};
interval<std::string> b{"kiwi", "peach"};
a = b; // throws during copy of "peach"
```



```
// assume there's no short string optimization
interval<std::string> a{"apple", "fig"};
interval<std::string> b{"kiwi", "peach"};
a = b; // throws during copy of "peach"
```





interval a is left in an invalid state

2. What is the canonical form of strongly exception-safe copy assignment?

It involves two steps: First, provide a nonthrowing Swap() function that swaps the guts (state) of two objects:

```
void T::Swap( T& other ) throw()
{
  // ...swap the guts of *this and other...
}
```

Second, implement operator=() using the "create a temporary and swap" idiom:

```
T& T::operator=( const T& other )
{
  T temp( other ); // do all the work off to the side

  Swap( temp );    // then "commit" the work using
  return *this;    //  nonthrowing operations only
}
```

"Exception-Safe Class Design, Part 1: Copy Assignment", Herb Sutter, July 1999

http://gotw.ca/gotw/059.htm

```
41  // Use the copy-and-move idiom to provide the strong exception guarantee
42  // The strong exception guarantee automatically provides the basic
43  // exception guarantee
44  interval& operator=(const interval& o) noexcept(std::is_nothrow_copy_constructible_v<T>) {
45    interval tmp(o);
46    *this = std::move(tmp);
47    return *this;
48  }
```

https://godbolt.org/z/nhKb59d38

```
41  // Use the copy-and-move idiom to provide the strong exception guarantee
42  // The strong exception guarantee automatically provides the basic
43  // exception guarantee
44  interval& operator=(const interval& o) noexcept(std::is_nothrow_copy_constructible_v<T>) {
45    interval tmp(o);
46    *this = std::move(tmp);
47    return *this;
48  }
```

Disadvantages:

- Requires manual implementation
- Slower than memberwise copy assignment

https://godbolt.org/z/nhKb59d38

<C++ standard library aside>

std::flat_set from C++23

std::flat_set

Defined in header <flat_set>

```
template<
    class Key,
    class Compare = std::less<Key>,          (since C++23)
    class KeyContainer = std::vector<Key>
> class flat_set;
```

The flat set is a container adaptor that gives the functionality of an associative container that stores a sorted set of unique objects of type Key. Sorting is done using the key comparison function Compare.

The class template `flat_set` acts as a wrapper to the underlying sorted container passed as object of type `KeyContainer`.

Everywhere the standard library uses the *Compare* requirements, uniqueness is determined by using the equivalence relation. Informally, two objects `a` and `b` are considered equivalent if neither compares less than the other: `!comp(a, b) && !comp(b, a)`.

`std::flat_set` meets the requirements of *Container*, *ReversibleContainer*, optional container requirements, and all requirements of *AssociativeContainer* (including logarithmic search complexity), except that:

- requirements related to nodes are not applicable,
- iterator invalidation requirements differ,
- the complexity of insertion and erasure operations is linear.

A flat set supports most *AssociativeContainer*'s operations that use unique keys.

https://en.cppreference.com/w/cpp/container/flat_set.html

4 Descriptions are provided here only for operations on `flat_set` that are not described in one of those sets of requirements or for operations where there is additional semantic information.

5 A `flat_set` maintains the invariant that the keys are sorted with respect to the comparison object.

6 If any member function in [flatset.defn] exits via an exception the invariant is restored. [*Note*: This may result in the `flat_set`'s being emptied. — *end note*]

7 Any sequence container ([sequence.reqmts]) supporting *Cpp17RandomAccessIterator* can be used to instantiate `flat_set`. In particular, `vector` ([vector]) and `deque` ([deque]) can be used. [*Note*: `vector<bool>` is not a sequence container. — *end note*]

https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2022/p1222r4.pdf

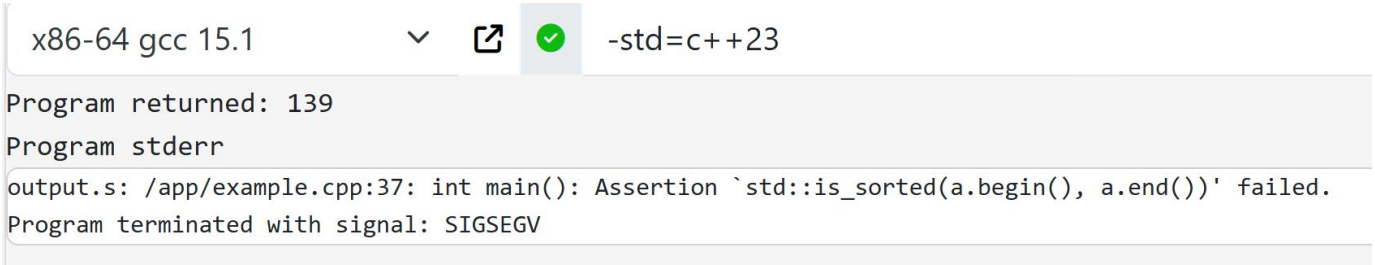
```cpp
struct Thrower {
  int value = 0;

  Thrower(int v) : value(v) {}

  ~Thrower() = default;
  Thrower(Thrower&&) = default;
  Thrower(const Thrower& other) = default;
  Thrower& operator=(Thrower&&) = default;

  Thrower& operator=(const Thrower& other) {
    static int count = 0;
    if (++count == 3) {
      throw std::bad_alloc();
    }

    value = other.value;
    return *this;
  }

  auto operator<=>(const Thrower&) const = default;
};
```

https://godbolt.org/z/aTq3bnWox

```cpp
int main() {
  std::flat_set<Thrower> a{1, 2, 3};
  std::flat_set<Thrower> b{4, 5, 6};

  try {
    a = b;
  } catch(...) {
  }

  assert(std::is_sorted(a.begin(), a.end()));
  return 0;
}
```

https://godbolt.org/z/aTq3bnWox

```cpp
int main() {
  std::flat_set<Thrower> a{1, 2, 3};
  std::flat_set<Thrower> b{4, 5, 6};

  try {
    a = b;
  } catch(...) {
  }

  assert(std::is_sorted(a.begin(), a.end()));
  return 0;
}
```



https://godbolt.org/z/aTq3bnWox

</C++ standard library aside>

We have applied all
5 guidelines!

```cpp
#include  <algorithm>

template <class T>
struct interval {
  T low;
  T high;
};

template <class T>
const T& clamp(const T& value, const interval<T>& interval) {
  return std::clamp(value, interval.low, interval.high);
}
```

```cpp
template <class T>
requires std::is_nothrow_default_constructible_v<T> &&
         std::is_nothrow_move_constructible_v<T> &&
         std::is_nothrow_move_assignable_v<T>
class interval {
private:
  T low_;
  T high_;

public:
  interval(T low, T high) : low_(std::move(low)), high_(std::move(high)) {
    if (!(low_ <= high_)) {
      throw std::runtime_error("invalid interval");
    }
  }

  ~interval() = default;

  // other.low_ and other.high_ will be replaced with default constructed
  // values to maintain class invariant low_ <= high_
  interval(interval&& other) noexcept
    : low_(std::exchange(other.low_, {})),
      high_(std::exchange(other.high_, {})) {
  }

  // other.low_ and other.high_ will be replaced with default constructed
  // values to maintain class invariant low_ <= high_
  interval& operator=(interval&& other) noexcept {
    low_ = std::exchange(other.low_, {});
    high_ = std::exchange(other.high_, {});
    return *this;
  }

  interval(const interval&) = default;

  // Use the copy-and-move idiom to provide the strong exception guarantee
  // The strong exception guarantee automatically provides the basic
  // exception guarantee
  interval& operator=(const interval& o) noexcept(std::is_nothrow_copy_constructible_v<T>) {
    interval tmp(o);
    *this = std::move(tmp);
    return *this;
  }

  const T& low() const { return low_; }
  const T& high() const { return high_; }
};

template <class T>
const T& clamp(const T& value, const interval<T>& interval) {
  return std::clamp(value, interval.low(), interval.high());
}
```

https://godbolt.org/z/jTxx7cG47

https://godbolt.org/z/nhKb59d38

Is it possible to have all of these at once?

valid but unspecified
moved from objects

“true” class invariants

optimal performance

rule of zero

fully generic code

catching and resolving exceptions with the basic exception guarantee

Is it possible to use the rule of zero with “true” class invariants?

What I currently teach

C++ Core Guideline C.20:

If you can avoid defining default operations, do (a.k.a rule of zero)

```
3   class SomeClass {
4   public:
5     SomeClass(IDependency* dependency) : dependency_(dependency) {}
6
7     ~SomeClass() { delete dependency_; }
8
9     // SomeClass member functions...
10
11  private:
12    // Disable copy constructor and assignment operator
13    SomeClass(const SomeClass&);
14    SomeClass& operator=(const SomeClass&);
15
16    IDependency* dependency_;
17  };
```

https://godbolt.org/z/bnzfMs6zv

Rule of three

https://drdobbs.com/c-made-easier-the-rule-of-three/184401400

```
3   class SomeClass {
4   public:
5     SomeClass(IDependency* dependency) : dependency_(dependency) {}
6
7     ~SomeClass() { delete dependency_; }
8
9     // SomeClass member functions...
10
11  private:
12    // Disable copy constructor and assignment operator
13    SomeClass(const SomeClass&);
14    SomeClass& operator=(const SomeClass&);
15
16    IDependency* dependency_;
17  };
```

https://godbolt.org/z/bnzfMs6zv

```
5   class SomeClass {
6   public:
7     SomeClass(std::unique_ptr<IDependency> dependency)
8       : dependency_(std::move(dependency)) {}
9
10    // SomeClass member functions...
11
12  private:
13    std::unique_ptr<IDependency> dependency_;
14  };
```

https://godbolt.org/z/4Y5EEde74

C++ Core Guideline C.2:

Use class if the class has an invariant; use struct if the data members can vary independently

```cpp
#include <cstdint>

template <class T>
class my_vector {
private:
  T* data_;
  std::size_t size_;
  std::size_t capacity_;

  // INVARIANT: size_ <= capacity_
  // INVARIANT: capacity_ == 0 ? data_ == nullptr : data_ != nullptr

public:
  // my_vector implementation assumes invariants
};
```

https://godbolt.org/z/TrcExdq8s

~~C++ Core Guideline C.64:~~

~~A move operation should move and leave its source in a valid state~~

Do not read from moved-from objects

Just like it is an error to read from uninitialized variables

Extra Clang Tools 22.0.0git documentation

CLANG-TIDY - BUGPRONE-USE-AFTER-MOVE

bugprone-use-after-move

Warns if an object is used after it has been moved, for example:

```
std::string str = "Hello, world!\n";
std::vector<std::string> messages;
messages.emplace_back(std::move(str));
std::cout << str;
```

The last line will trigger a warning that `str` is used after it has been moved.

https://clang.llvm.org/extra/clang-tidy/checks/bugprone/use-after-move.html

If you are implementing move operations:

- Moved from objects must be destructible
- Moved from objects must be ready to be assigned into
- It must be possible to assign a moved from object into another object of the same type

If you are implementing move operations:

- Moved from objects must be destructible
- Moved from objects must be ready to be assigned into
- It must be possible to assign a moved from object into another object of the same type

These rules compose well with defaulted memberwise special member functions!

~~Always provide at least the basic exception guarantee~~

~~C++ Core Guideline I.10:~~

~~Use exceptions to signal a failure to perform a required task~~

Different error handling strategies have different strengths and weaknesses

Use destructors for cleanup to prevent resource leaks. Objects that throw exceptions must still be destructible.

Different error handling strategies are stronger or weaker depending on the context. Returning error codes, exceptions, and narrow contracts can all be considered.

Throwing an exception is fine. Catching and handling an exception should be done with care, because it is not guaranteed that the program’s invariants are still true.

Why this talk exists

- Are there commonly taught C++ guidelines that are incompatible in practice?
- What are the role of class invariants in C++?
- What guidelines should we teach?
- How do we write good code?

Thank you!

Questions?