“Mostly Invalid”

Container adaptors, exception guarantees, and the STL

Arthur O’Dwyer
2019-09-18

Outline

This talk grew out of my blog post
"Fetishizing class invariants" (Feb 2019)

- A fun example [3–35]
- Another fun example [36–52]
- Another (formerly) fun example [53–63]
- Relation to `flat_set` and `flat_map` [64–67]
- LWG 2189 and Billy O'Neal's P1843 [68–69]
- Vague complaints and speculations [70–77]
- Questions?

https://godbolt.org/z/bPo1Gf

PQ with a throwing comparator

Create a priority queue with a custom comparator:

```
using Cmp = std::function<bool(int, int)>;
using PQ = std::priority_queue<int, std::vector<int>, Cmp>;

PQ pq( [](int a, int b) {
    if (a == 2 && b == 3) throw "oops"; return (a < b);
});
```

Print the elements of a priority queue:

```
puts("Elements from highest to lowest:");
while (!pq.empty()) { printf("%d\n", pq.top()); pq.pop(); }
```

https://godbolt.org/z/bPo1Gf

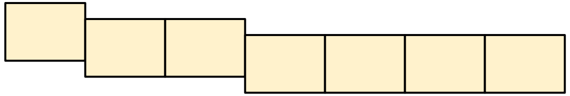
PQ with a throwing comparator

```
std::priority_queue<int, std::vector<int>, Cmp> pq( [](int a, int b) {
    if (a == 2 && b == 3) throw "oops"; return (a < b);
});

pq.push(2);
pq.push(2);
pq.push(1);
try { pq.push(3); } catch (...) {}
try { pq.push(3); } catch (...) {}
pq.push(4);

puts("Elements from highest to lowest:");
while (!pq.empty()) { printf("%d\n", pq.top()); pq.pop(); }
```

https://godbolt.org/z/bPo1Gf

PQ with a throwing comparator

```
std::priority_queue<int, std::vector<int>, Cmp> pq( [](int a, int b) {
    if (a == 2 && b == 3) throw "oops"; return (a < b);
});
```

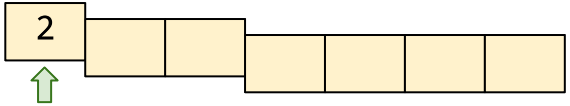


```
pq.push(2);
pq.push(2);
pq.push(1);
try { pq.push(3); } catch (...) {}
try { pq.push(3); } catch (...) {}
pq.push(4);

puts("Elements from highest to lowest:");
while (!pq.empty()) { printf("%d\n", pq.top()); pq.pop(); }
```

https://godbolt.org/z/bPo1Gf

PQ with a throwing comparator

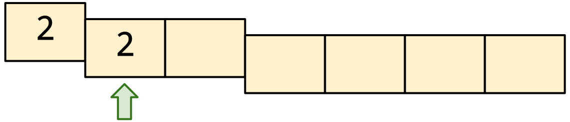
```
std::priority_queue<int, std::vector<int>, Cmp> pq( [](int a, int b) {
    if (a == 2 && b == 3) throw "oops"; return (a < b);
});

pq.push(2);
pq.push(2);
pq.push(1);
try { pq.push(3); } catch (...) {}
try { pq.push(3); } catch (...) {}
pq.push(4);

puts("Elements from highest to lowest:");
while (!pq.empty()) { printf("%d\n", pq.top()); pq.pop(); }
```



https://godbolt.org/z/bPo1Gf

PQ with a throwing comparator

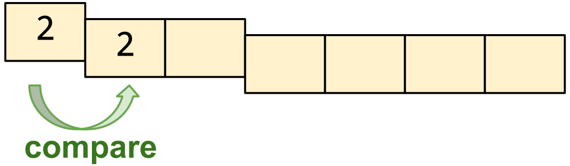
```
std::priority_queue<int, std::vector<int>, Cmp> pq( [](int a, int b) {
    if (a == 2 && b == 3) throw "oops"; return (a < b);
});

pq.push(2);
pq.push(2);
pq.push(1);
try { pq.push(3); } catch (...) {}
try { pq.push(3); } catch (...) {}
pq.push(4);

puts("Elements from highest to lowest:");
while (!pq.empty()) { printf("%d\n", pq.top()); pq.pop(); }
```



https://godbolt.org/z/bPo1Gf

PQ with a throwing comparator

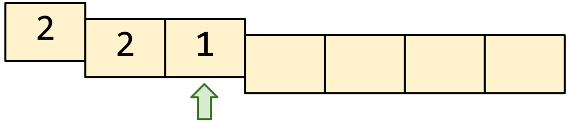
```
std::priority_queue<int, std::vector<int>, Cmp> pq( [](int a, int b) {
    if (a == 2 && b == 3) throw "oops"; return (a < b);
});

pq.push(2);
pq.push(2);
pq.push(1);
try { pq.push(3); } catch (...) {}
try { pq.push(3); } catch (...) {}
pq.push(4);

puts("Elements from highest to lowest:");
while (!pq.empty()) { printf("%d\n", pq.top()); pq.pop(); }
```



https://godbolt.org/z/bPo1Gf

PQ with a throwing comparator

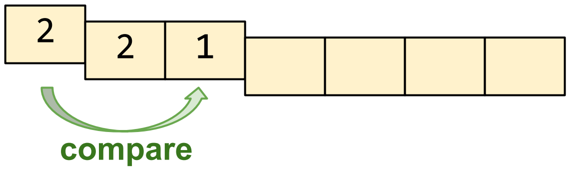
```
std::priority_queue<int, std::vector<int>, Cmp> pq( [](int a, int b) {
    if (a == 2 && b == 3) throw "oops"; return (a < b);
});

pq.push(2);
pq.push(2);
pq.push(1);
try { pq.push(3); } catch (...) {}
try { pq.push(3); } catch (...) {}
pq.push(4);

puts("Elements from highest to lowest:");
while (!pq.empty()) { printf("%d\n", pq.top()); pq.pop(); }
```



https://godbolt.org/z/bPo1Gf

PQ with a throwing comparator

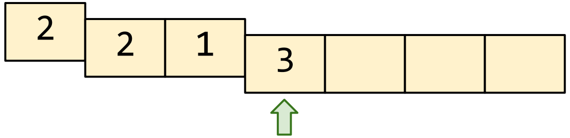
```
std::priority_queue<int, std::vector<int>, Cmp> pq( [](int a, int b) {
    if (a == 2 && b == 3) throw "oops"; return (a < b);
});

pq.push(2);
pq.push(2);
pq.push(1);
try { pq.push(3); } catch (...) {}
try { pq.push(3); } catch (...) {}
pq.push(4);

puts("Elements from highest to lowest:");
while (!pq.empty()) { printf("%d\n", pq.top()); pq.pop(); }
```



https://godbolt.org/z/bPo1Gf

PQ with a throwing comparator

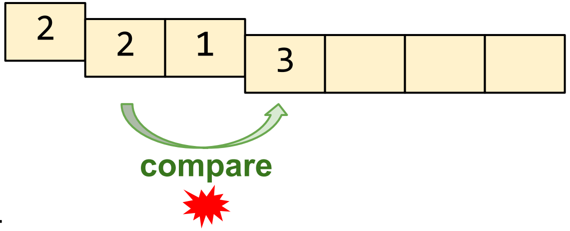
```
std::priority_queue<int, std::vector<int>, Cmp> pq( [](int a, int b) {
    if (a == 2 && b == 3) throw "oops"; return (a < b);
});

pq.push(2);
pq.push(2);
pq.push(1);
try { pq.push(3); } catch (...) {}
try { pq.push(3); } catch (...) {}
pq.push(4);

puts("Elements from highest to lowest:");
while (!pq.empty()) { printf("%d\n", pq.top()); pq.pop(); }
```



https://godbolt.org/z/bPo1Gf

PQ with a throwing comparator

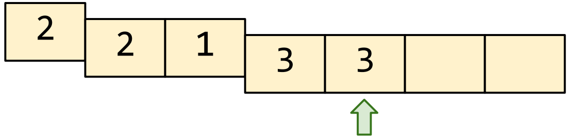
```
std::priority_queue<int, std::vector<int>, Cmp> pq( [](int a, int b) {
    if (a == 2 && b == 3) throw "oops"; return (a < b);
});

pq.push(2);
pq.push(2);
pq.push(1);
try { pq.push(3); } catch (...) {}
try { pq.push(3); } catch (...) {}
pq.push(4);

puts("Elements from highest to lowest:");
while (!pq.empty()) { printf("%d\n", pq.top()); pq.pop(); }
```



https://godbolt.org/z/bPo1Gf

PQ with a throwing comparator

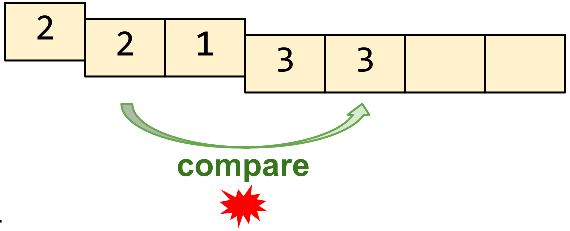
```
std::priority_queue<int, std::vector<int>, Cmp> pq( [](int a, int b) {
    if (a == 2 && b == 3) throw "oops"; return (a < b);
});

pq.push(2);
pq.push(2);
pq.push(1);
try { pq.push(3); } catch (...) {}
try { pq.push(3); } catch (...) {}
pq.push(4);

puts("Elements from highest to lowest:");
while (!pq.empty()) { printf("%d\n", pq.top()); pq.pop(); }
```



https://godbolt.org/z/bPo1Gf

PQ with a throwing comparator

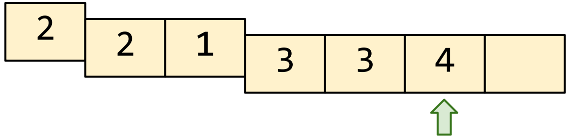
```
std::priority_queue<int, std::vector<int>, Cmp> pq( [](int a, int b) {
    if (a == 2 && b == 3) throw "oops"; return (a < b);
});

pq.push(2);
pq.push(2);
pq.push(1);
try { pq.push(3); } catch (...) {}
try { pq.push(3); } catch (...) {}
pq.push(4);

puts("Elements from highest to lowest:");
while (!pq.empty()) { printf("%d\n", pq.top()); pq.pop(); }
```



https://godbolt.org/z/bPo1Gf

PQ with a throwing comparator

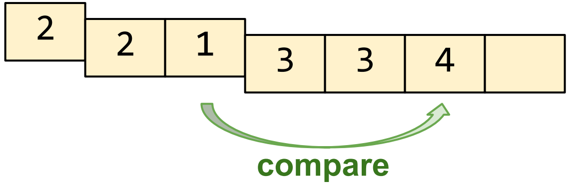
```
std::priority_queue<int, std::vector<int>, Cmp> pq( [](int a, int b) {
    if (a == 2 && b == 3) throw "oops"; return (a < b);
});

pq.push(2);
pq.push(2);
pq.push(1);
try { pq.push(3); } catch (...) {}
try { pq.push(3); } catch (...) {}
pq.push(4);

puts("Elements from highest to lowest:");
while (!pq.empty()) { printf("%d\n", pq.top()); pq.pop(); }
```



https://godbolt.org/z/bPo1Gf

PQ with a throwing comparator

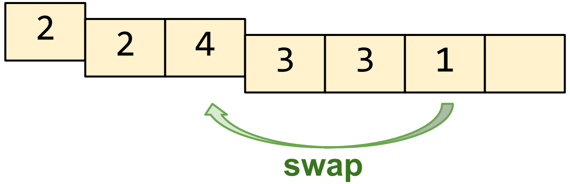
```
std::priority_queue<int, std::vector<int>, Cmp> pq( [](int a, int b) {
    if (a == 2 && b == 3) throw "oops"; return (a < b);
});

pq.push(2);
pq.push(2);
pq.push(1);
try { pq.push(3); } catch (...) {}
try { pq.push(3); } catch (...) {}
pq.push(4);

puts("Elements from highest to lowest:");
while (!pq.empty()) { printf("%d\n", pq.top()); pq.pop(); }
```



https://godbolt.org/z/bPo1Gf

PQ with a throwing comparator

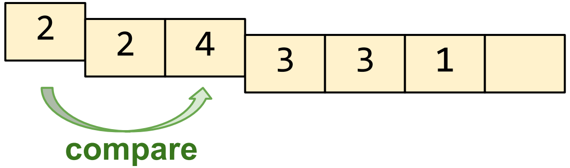
```
std::priority_queue<int, std::vector<int>, Cmp> pq( [](int a, int b) {
    if (a == 2 && b == 3) throw "oops"; return (a < b);
});

pq.push(2);
pq.push(2);
pq.push(1);
try { pq.push(3); } catch (...) {}
try { pq.push(3); } catch (...) {}
pq.push(4);

puts("Elements from highest to lowest:");
while (!pq.empty()) { printf("%d\n", pq.top()); pq.pop(); }
```



https://godbolt.org/z/bPo1Gf

PQ with a throwing comparator

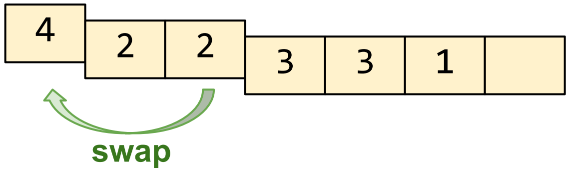
```
std::priority_queue<int, std::vector<int>, Cmp> pq( [](int a, int b) {
    if (a == 2 && b == 3) throw "oops"; return (a < b);
});

pq.push(2);
pq.push(2);
pq.push(1);
try { pq.push(3); } catch (...) {}
try { pq.push(3); } catch (...) {}
pq.push(4);

puts("Elements from highest to lowest:");
while (!pq.empty()) { printf("%d\n", pq.top()); pq.pop(); }
```



https://godbolt.org/z/bPo1Gf

PQ with a throwing comparator

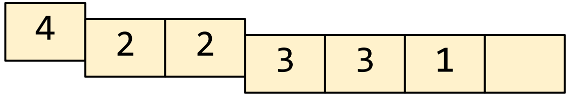
```
std::priority_queue<int, std::vector<int>, Cmp> pq( [](int a, int b) {
    if (a == 2 && b == 3) throw "oops"; return (a < b);
});

pq.push(2);
pq.push(2);
pq.push(1);
try { pq.push(3); } catch (...) {}
try { pq.push(3); } catch (...) {}
pq.push(4);

puts("Elements from highest to lowest:");
while (!pq.empty()) { printf("%d\n", pq.top()); pq.pop(); }
```



https://godbolt.org/z/bPo1Gf

PQ with a throwing comparator

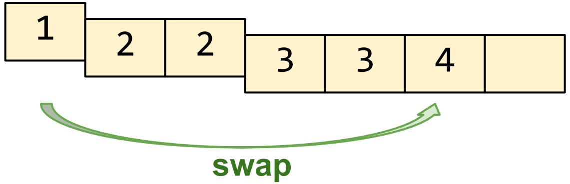
```
std::priority_queue<int, std::vector<int>, Cmp> pq( [](int a, int b) {
    if (a == 2 && b == 3) throw "oops"; return (a < b);
});

pq.push(2);
pq.push(2);
pq.push(1);
try { pq.push(3); } catch (...) {}
try { pq.push(3); } catch (...) {}
pq.push(4);

puts("Elements from highest to lowest:");
while (!pq.empty()) { printf("%d\n", pq.top()); pq.pop(); }
```



https://godbolt.org/z/bPo1Gf

PQ with a throwing comparator

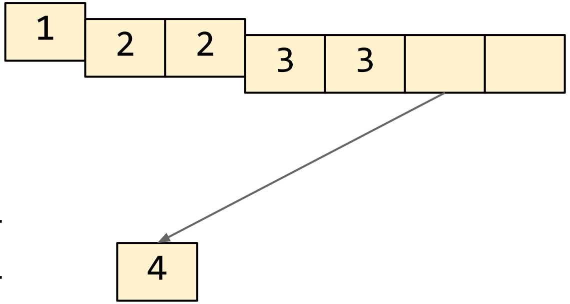
```
std::priority_queue<int, std::vector<int>, Cmp> pq( [](int a, int b) {
    if (a == 2 && b == 3) throw "oops"; return (a < b);
});

pq.push(2);
pq.push(2);
pq.push(1);
try { pq.push(3); } catch (...) {}
try { pq.push(3); } catch (...) {}
pq.push(4);

puts("Elements from highest to lowest:");
while (!pq.empty()) { printf("%d\n", pq.top()); pq.pop(); }
```



https://godbolt.org/z/bPo1Gf

PQ with a throwing comparator

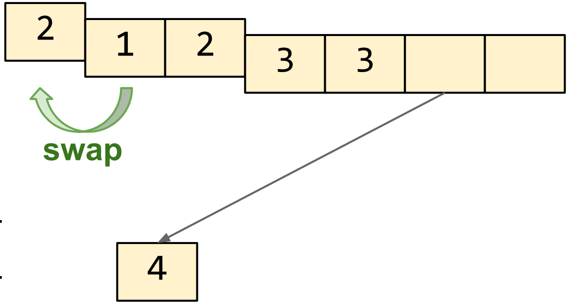
```
std::priority_queue<int, std::vector<int>, Cmp> pq( [](int a, int b) {
    if (a == 2 && b == 3) throw "oops"; return (a < b);
});

pq.push(2);
pq.push(2);
pq.push(1);
try { pq.push(3); } catch (...) {}
try { pq.push(3); } catch (...) {}
pq.push(4);

puts("Elements from highest to lowest:");
while (!pq.empty()) { printf("%d\n", pq.top()); pq.pop(); }
```



https://godbolt.org/z/bPo1Gf

PQ with a throwing comparator

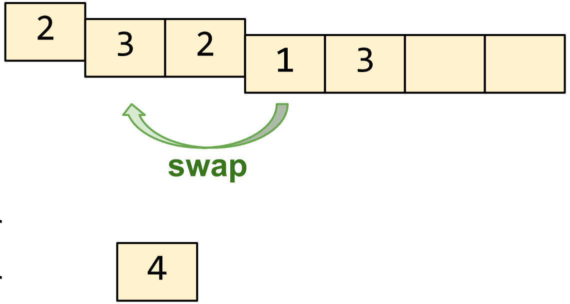
```
std::priority_queue<int, std::vector<int>, Cmp> pq( [](int a, int b) {
    if (a == 2 && b == 3) throw "oops"; return (a < b);
});

pq.push(2);
pq.push(2);
pq.push(1);
try { pq.push(3); } catch (...) {}
try { pq.push(3); } catch (...) {}
pq.push(4);

puts("Elements from highest to lowest:");
while (!pq.empty()) { printf("%d\n", pq.top()); pq.pop(); }
```



https://godbolt.org/z/bPo1Gf

PQ with a throwing comparator

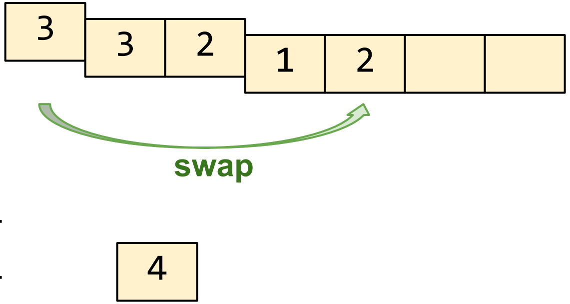
```
std::priority_queue<int, std::vector<int>, Cmp> pq( [](int a, int b) {
    if (a == 2 && b == 3) throw "oops"; return (a < b);
});

pq.push(2);
pq.push(2);
pq.push(1);
try { pq.push(3); } catch (...) {}
try { pq.push(3); } catch (...) {}
pq.push(4);

puts("Elements from highest to lowest:");
while (!pq.empty()) { printf("%d\n", pq.top()); pq.pop(); }
```



https://godbolt.org/z/bPo1Gf

PQ with a throwing comparator

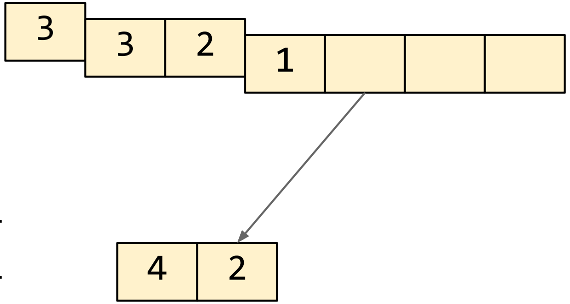
```
std::priority_queue<int, std::vector<int>, Cmp> pq( [](int a, int b) {
    if (a == 2 && b == 3) throw "oops"; return (a < b);
});

pq.push(2);
pq.push(2);
pq.push(1);
try { pq.push(3); } catch (...) {}
try { pq.push(3); } catch (...) {}
pq.push(4);

puts("Elements from highest to lowest:");
while (!pq.empty()) { printf("%d\n", pq.top()); pq.pop(); }
```



https://godbolt.org/z/bPo1Gf

PQ with a throwing comparator

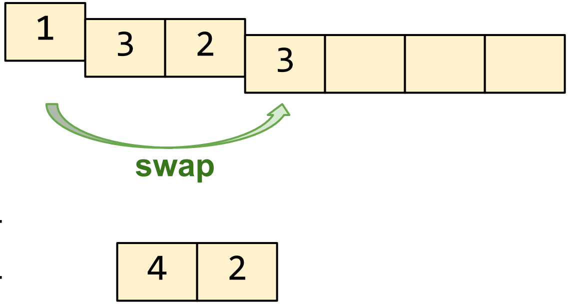
```
std::priority_queue<int, std::vector<int>, Cmp> pq( [](int a, int b) {
    if (a == 2 && b == 3) throw "oops"; return (a < b);
});

pq.push(2);
pq.push(2);
pq.push(1);
try { pq.push(3); } catch (...) {}
try { pq.push(3); } catch (...) {}
pq.push(4);

puts("Elements from highest to lowest:");
while (!pq.empty()) { printf("%d\n", pq.top()); pq.pop(); }
```



https://godbolt.org/z/bPo1Gf

PQ with a throwing comparator

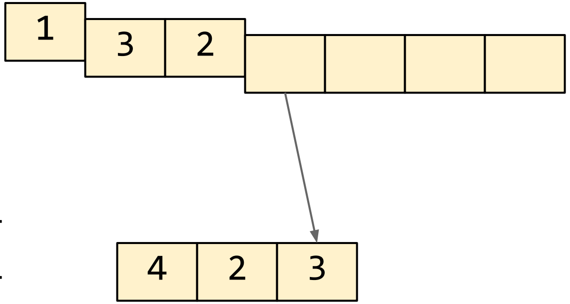
```
std::priority_queue<int, std::vector<int>, Cmp> pq( [](int a, int b) {
    if (a == 2 && b == 3) throw "oops"; return (a < b);
});

pq.push(2);
pq.push(2);
pq.push(1);
try { pq.push(3); } catch (...) {}
try { pq.push(3); } catch (...) {}
pq.push(4);

puts("Elements from highest to lowest:");
while (!pq.empty()) { printf("%d\n", pq.top()); pq.pop(); }
```



https://godbolt.org/z/bPo1Gf

PQ with a throwing comparator

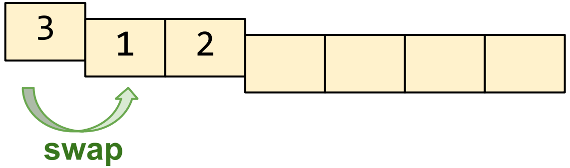
```
std::priority_queue<int, std::vector<int>, Cmp> pq( [](int a, int b) {
    if (a == 2 && b == 3) throw "oops"; return (a < b);
});

pq.push(2);
pq.push(2);
pq.push(1);
try { pq.push(3); } catch (...) {}
try { pq.push(3); } catch (...) {}
pq.push(4);

puts("Elements from highest to lowest:");
while (!pq.empty()) { printf("%d\n", pq.top()); pq.pop(); }
```

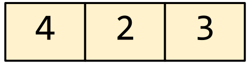


https://godbolt.org/z/bPo1Gf

PQ with a throwing comparator

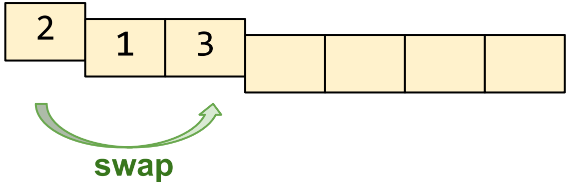
```
std::priority_queue<int, std::vector<int>, Cmp> pq( [](int a, int b) {
    if (a == 2 && b == 3) throw "oops"; return (a < b);
});

pq.push(2);
pq.push(2);
pq.push(1);
try { pq.push(3); } catch (...) {}
try { pq.push(3); } catch (...) {}
pq.push(4);

puts("Elements from highest to lowest:");
while (!pq.empty()) { printf("%d\n", pq.top()); pq.pop(); }
```

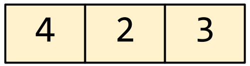




PQ with a throwing comparator

```
std::priority_queue<int, std::vector<int>, Cmp> pq( [](int a, int b) {
    if (a == 2 && b == 3) throw "oops"; return (a < b);
});

pq.push(2);
pq.push(2);
pq.push(1);
try { pq.push(3); } catch (...) {}
try { pq.push(3); } catch (...) {}
pq.push(4);

puts("Elements from highest to lowest:");
while (!pq.empty()) { printf("%d\n", pq.top()); pq.pop(); }
```

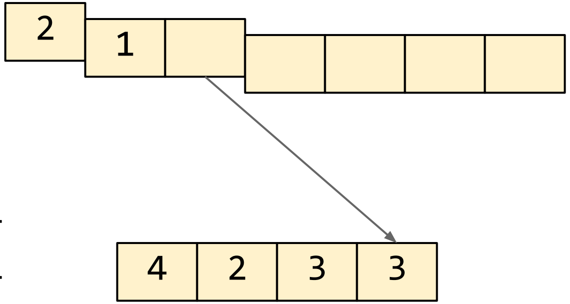




https://godbolt.org/z/bPo1Gf

PQ with a throwing comparator

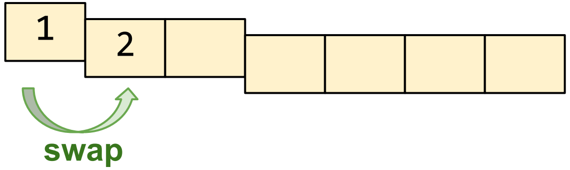
```
std::priority_queue<int, std::vector<int>, Cmp> pq( [](int a, int b) {
    if (a == 2 && b == 3) throw "oops"; return (a < b);
});

pq.push(2);
pq.push(2);
pq.push(1);
try { pq.push(3); } catch (...) {}
try { pq.push(3); } catch (...) {}
pq.push(4);

puts("Elements from highest to lowest:");
while (!pq.empty()) { printf("%d\n", pq.top()); pq.pop(); }
```

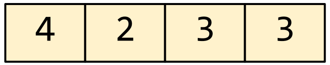


https://godbolt.org/z/bPo1Gf

PQ with a throwing comparator

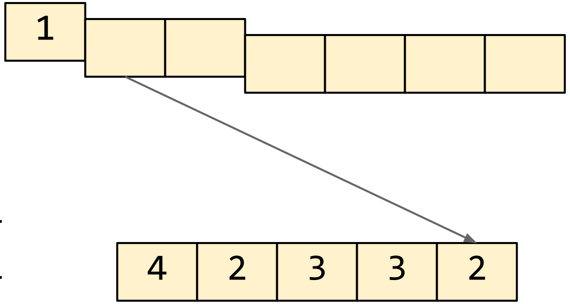
```
std::priority_queue<int, std::vector<int>, Cmp> pq( [](int a, int b) {
    if (a == 2 && b == 3) throw "oops"; return (a < b);
});

pq.push(2);
pq.push(2);
pq.push(1);
try { pq.push(3); } catch (...) {}
try { pq.push(3); } catch (...) {}
pq.push(4);

puts("Elements from highest to lowest:");
while (!pq.empty()) { printf("%d\n", pq.top()); pq.pop(); }
```





PQ with a throwing comparator

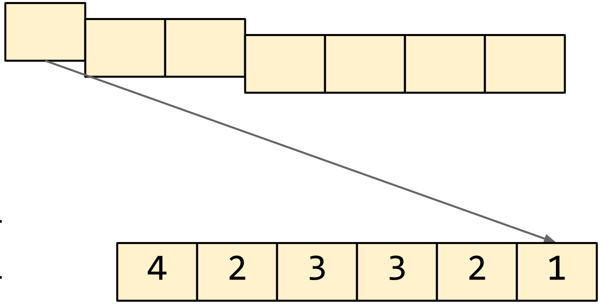
```
std::priority_queue<int, std::vector<int>, Cmp> pq( [](int a, int b) {
    if (a == 2 && b == 3) throw "oops"; return (a < b);
});

pq.push(2);
pq.push(2);
pq.push(1);
try { pq.push(3); } catch (...) {}
try { pq.push(3); } catch (...) {}
pq.push(4);

puts("Elements from highest to lowest:");
while (!pq.empty()) { printf("%d\n", pq.top()); pq.pop(); }
```



https://godbolt.org/z/bPo1Gf

PQ with a throwing comparator

```
std::priority_queue<int, std::vector<int>, Cmp> pq( [](int a, int b) {
    if (a == 2 && b == 3) throw "oops"; return (a < b);
});

pq.push(2);
pq.push(2);
pq.push(1);
try { pq.push(3); } catch (...) {}
try { pq.push(3); } catch (...) {}
pq.push(4);

puts("Elements from highest to lowest:");
while (!pq.empty()) { printf("%d\n", pq.top()); pq.pop(); }
```



This is with libc++. libstdc++ produces “4 2 2 3 3 1.”

Throwing comparator breaks PQ’s invariant

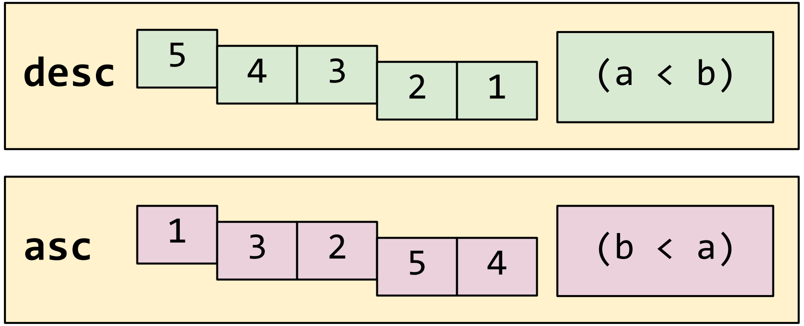
- `std::priority_queue` has a ***class invariant*** that its vector is always sorted by the comparator.
- The point of a class invariant is that it should invariably be satisfied (except perhaps briefly inside a member function).
- A throwing comparator can force `priority_queue::push()` to exit after doing some work but ***before*** restoring the invariant.
- This puts the `priority_queue` into a very bad state.
  - Its class invariant is broken!

There are other ways to break PQ's invariant

https://godbolt.org/z/8pJ4JE

- A comparator whose assignment operator throws is also trouble for `priority_queue`

```
struct X {
    bool up;
    X& operator=(const X&) { throw "oops"; }
    bool operator()(int a, int b) const {
        return up ? (b < a) : (a < b);
    }
};
```



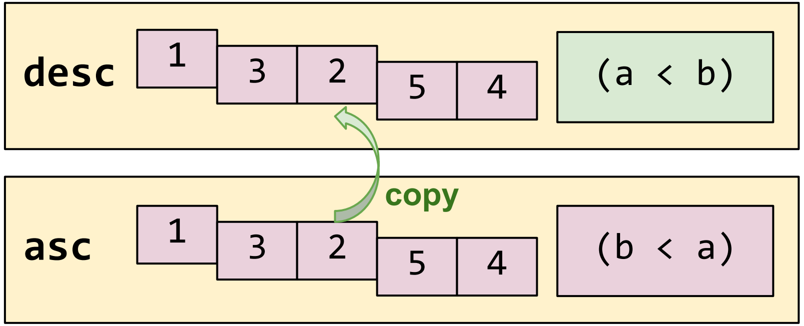

```
std::priority_queue<int, std::vector<int>, X> desc(X{false}, {5,4,3,2,1});
std::priority_queue<int, std::vector<int>, X> asc(X{true}, {1,3,2,5,4});
try { desc = asc; } catch (...) {}
```

There are other ways to break PQ's invariant

https://godbolt.org/z/8pJ4JE

- A comparator whose assignment operator throws is also trouble for `priority_queue`

```
struct X {
    bool up;
    X& operator=(const X&) { throw "oops"; }
    bool operator()(int a, int b) const {
        return up ? (b < a) : (a < b);
    }
};
```



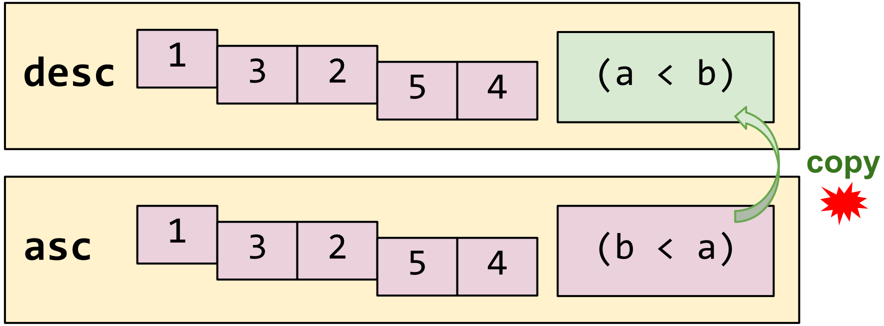

```
std::priority_queue<int, std::vector<int>, X> desc(X{false}, {5,4,3,2,1});
std::priority_queue<int, std::vector<int>, X> asc(X{true}, {1,3,2,5,4});
try { desc = asc; } catch (...) {}
```

There are other ways to break PQ's invariant

https://godbolt.org/z/8pJ4JE

- A comparator whose assignment operator throws is also trouble for `priority_queue`

```
struct X {
    bool up;
    X& operator=(const X&) { throw "oops"; }
    bool operator()(int a, int b) const {
        return up ? (b < a) : (a < b);
    }
};
```



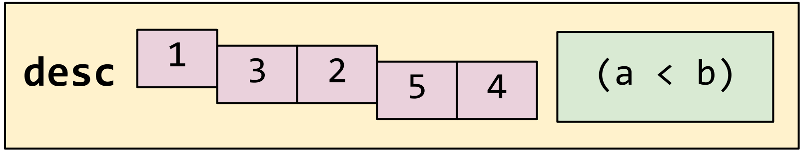

```
std::priority_queue<int, std::vector<int>, X> desc(X{false}, {5,4,3,2,1});
std::priority_queue<int, std::vector<int>, X> asc(X{true}, {1,3,2,5,4});
try { desc = asc; } catch (...) {}
```

There are other ways to break PQ's invariant

https://godbolt.org/z/8pJ4JE

- A comparator whose assignment operator throws is also trouble for `priority_queue`

```
struct X {
    bool up;
    X& operator=(const X&) { throw "oops"; }
    bool operator()(int a, int b) const {
        return up ? (b < a) : (a < b);
    }
};
```



Now to do the popping and printing...

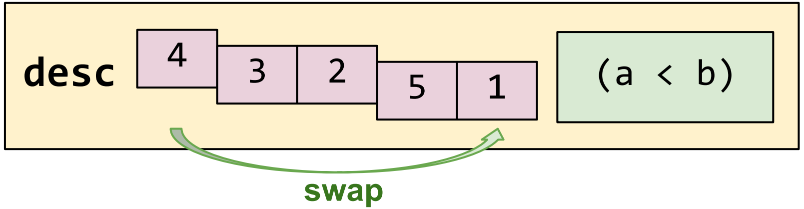
```
std::priority_queue<int, std::vector<int>, X> desc(X{false}, {5,4,3,2,1});
std::priority_queue<int, std::vector<int>, X> asc(X{true}, {1,3,2,5,4});
try { desc = asc; } catch (...) {}
```

There are other ways to break PQ's invariant

https://godbolt.org/z/8pJ4JE

- A comparator whose assignment operator throws is also trouble for `priority_queue`

```
struct X {
    bool up;
    X& operator=(const X&) { throw "oops"; }
    bool operator()(int a, int b) const {
        return up ? (b < a) : (a < b);
    }
};
```



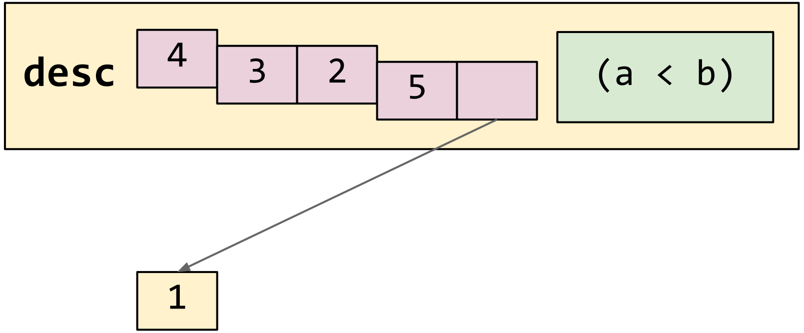

```
std::priority_queue<int, std::vector<int>, X> desc(X{false}, {5,4,3,2,1});
std::priority_queue<int, std::vector<int>, X> asc(X{true}, {1,3,2,5,4});
try { desc = asc; } catch (...) {}
```

There are other ways to break PQ's invariant

https://godbolt.org/z/8pJ4JE

- A comparator whose assignment operator throws is also trouble for `priority_queue`

```
struct X {
    bool up;
    X& operator=(const X&) { throw "oops"; }
    bool operator()(int a, int b) const {
        return up ? (b < a) : (a < b);
    }
};
```



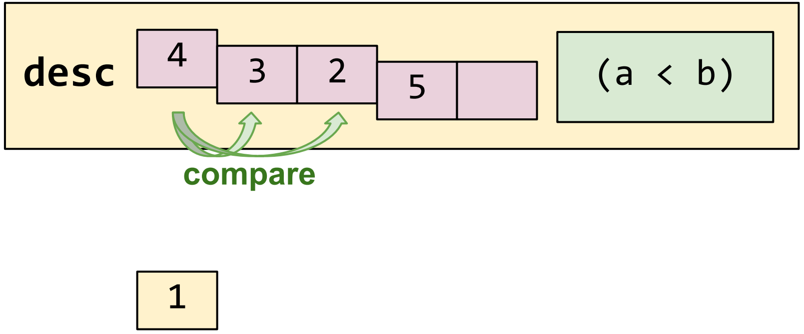

```
std::priority_queue<int, std::vector<int>, X> desc(X{false}, {5,4,3,2,1});
std::priority_queue<int, std::vector<int>, X> asc(X{true}, {1,3,2,5,4});
try { desc = asc; } catch (...) {}
```

There are other ways to break PQ's invariant

https://godbolt.org/z/8pJ4JE

- A comparator whose assignment operator throws is also trouble for `priority_queue`

```
struct X {
    bool up;
    X& operator=(const X&) { throw "oops"; }
    bool operator()(int a, int b) const {
        return up ? (b < a) : (a < b);
    }
};
```



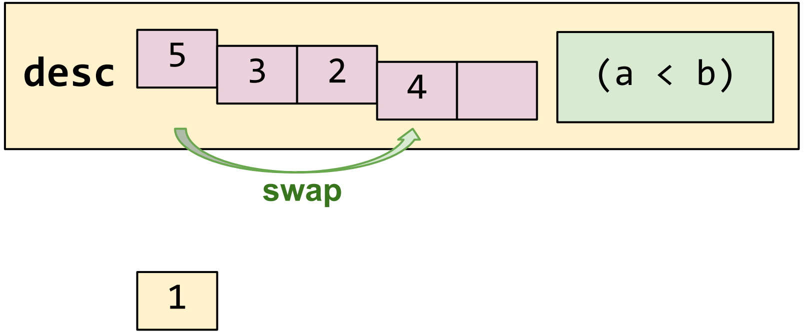

```
std::priority_queue<int, std::vector<int>, X> desc(X{false}, {5,4,3,2,1});
std::priority_queue<int, std::vector<int>, X> asc(X{true}, {1,3,2,5,4});
try { desc = asc; } catch (...) {}
```

There are other ways to break PQ's invariant

https://godbolt.org/z/8pJ4JE

- A comparator whose assignment operator throws is also trouble for `priority_queue`

```
struct X {
    bool up;
    X& operator=(const X&) { throw "oops"; }
    bool operator()(int a, int b) const {
        return up ? (b < a) : (a < b);
    }
};
```

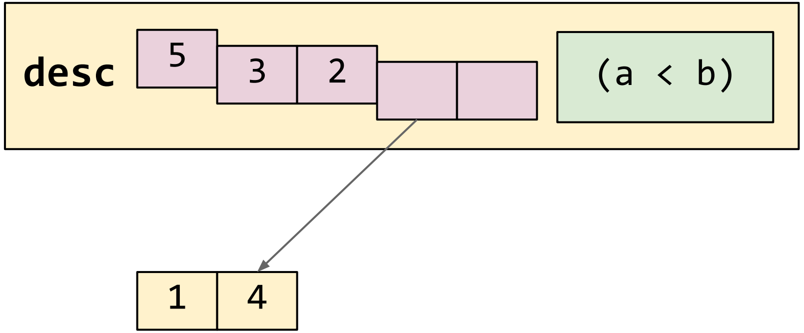



```
std::priority_queue<int, std::vector<int>, X> desc(X{false}, {5,4,3,2,1});
std::priority_queue<int, std::vector<int>, X> asc(X{true}, {1,3,2,5,4});
try { desc = asc; } catch (...) {}
```

There are other ways to break PQ's invariant https://godbolt.org/z/8pJ4JE

- A comparator whose assignment operator throws is also trouble for `priority_queue`

```
struct X {
    bool up;
    X& operator=(const X&) { throw "oops"; }
    bool operator()(int a, int b) const {
        return up ? (b < a) : (a < b);
    }
};
```



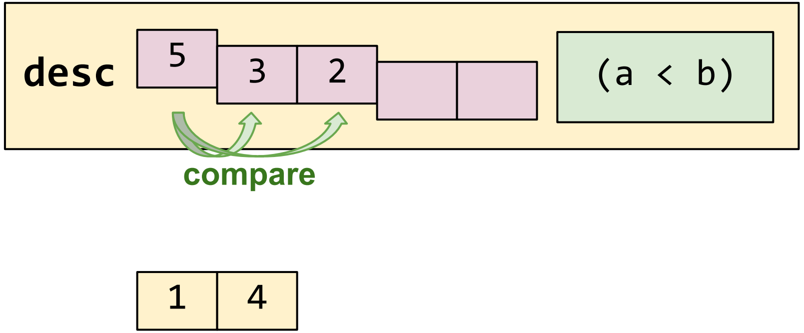

```
std::priority_queue<int, std::vector<int>, X> desc(X{false}, {5,4,3,2,1});
std::priority_queue<int, std::vector<int>, X> asc(X{true}, {1,3,2,5,4});
try { desc = asc; } catch (...) {}
```

There are other ways to break PQ's invariant

https://godbolt.org/z/8pJ4JE

- A comparator whose assignment operator throws is also trouble for `priority_queue`

```
struct X {
    bool up;
    X& operator=(const X&) { throw "oops"; }
    bool operator()(int a, int b) const {
        return up ? (b < a) : (a < b);
    }
};
```



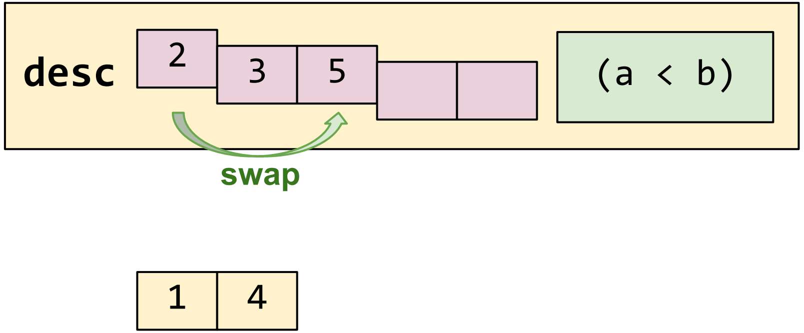

```
std::priority_queue<int, std::vector<int>, X> desc(X{false}, {5,4,3,2,1});
std::priority_queue<int, std::vector<int>, X> asc(X{true}, {1,3,2,5,4});
try { desc = asc; } catch (...) {}
```

There are other ways to break PQ's invariant

https://godbolt.org/z/8pJ4JE

- A comparator whose assignment operator throws is also trouble for `priority_queue`

```
struct X {
    bool up;
    X& operator=(const X&) { throw "oops"; }
    bool operator()(int a, int b) const {
        return up ? (b < a) : (a < b);
    }
};
```



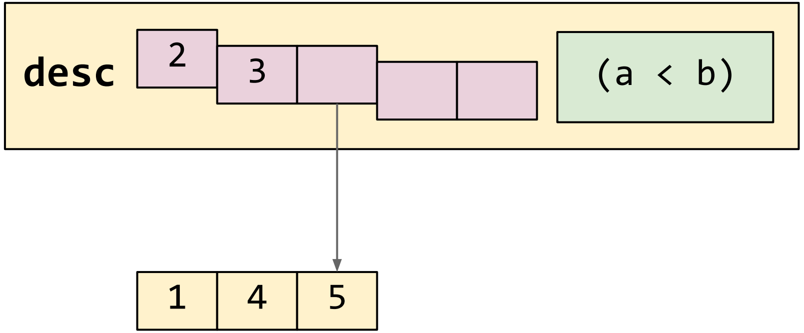

```
std::priority_queue<int, std::vector<int>, X> desc(X{false}, {5,4,3,2,1});
std::priority_queue<int, std::vector<int>, X> asc(X{true}, {1,3,2,5,4});
try { desc = asc; } catch (...) {}
```

There are other ways to break PQ's invariant

https://godbolt.org/z/8pJ4JE

- A comparator whose assignment operator throws is also trouble for `priority_queue`

```
struct X {
    bool up;
    X& operator=(const X&) { throw "oops"; }
    bool operator()(int a, int b) const {
        return up ? (b < a) : (a < b);
    }
};
```



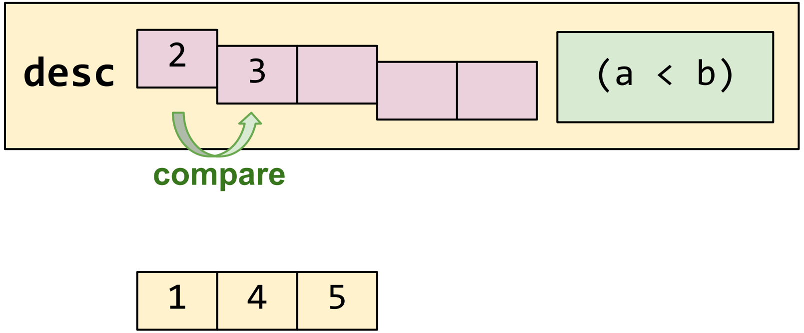

```
std::priority_queue<int, std::vector<int>, X> desc(X{false}, {5,4,3,2,1});
std::priority_queue<int, std::vector<int>, X> asc(X{true}, {1,3,2,5,4});
try { desc = asc; } catch (...) {}
```

There are other ways to break PQ's invariant

https://godbolt.org/z/8pJ4JE

- A comparator whose assignment operator throws is also trouble for `priority_queue`

```
struct X {
    bool up;
    X& operator=(const X&) { throw "oops"; }
    bool operator()(int a, int b) const {
        return up ? (b < a) : (a < b);
    }
};
```



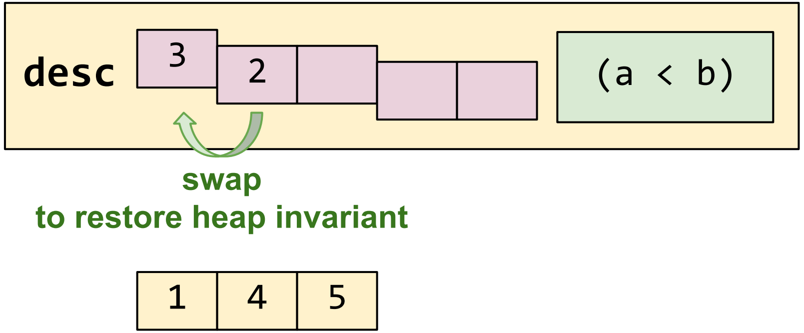

```
std::priority_queue<int, std::vector<int>, X> desc(X{false}, {5,4,3,2,1});
std::priority_queue<int, std::vector<int>, X> asc(X{true}, {1,3,2,5,4});
try { desc = asc; } catch (...) {}
```

There are other ways to break PQ's invariant

https://godbolt.org/z/8pJ4JE

- A comparator whose assignment operator throws is also trouble for `priority_queue`

```
struct X {
    bool up;
    X& operator=(const X&) { throw "oops"; }
    bool operator()(int a, int b) const {
        return up ? (b < a) : (a < b);
    }
};
```



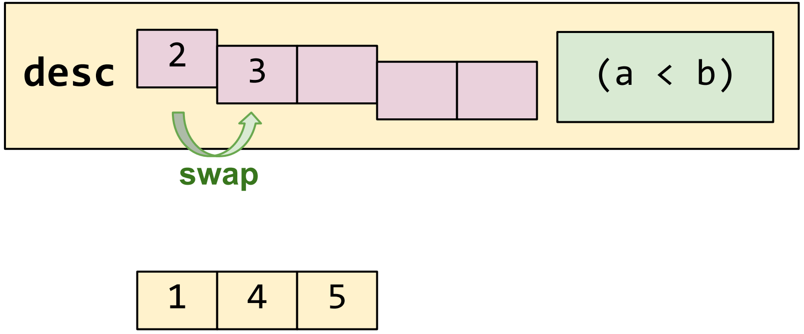

```
std::priority_queue<int, std::vector<int>, X> desc(X{false}, {5,4,3,2,1});
std::priority_queue<int, std::vector<int>, X> asc(X{true}, {1,3,2,5,4});
try { desc = asc; } catch (...) {}
```

There are other ways to break PQ's invariant

https://godbolt.org/z/8pJ4JE

- A comparator whose assignment operator throws is also trouble for `priority_queue`

```
struct X {
    bool up;
    X& operator=(const X&) { throw "oops"; }
    bool operator()(int a, int b) const {
        return up ? (b < a) : (a < b);
    }
};
```



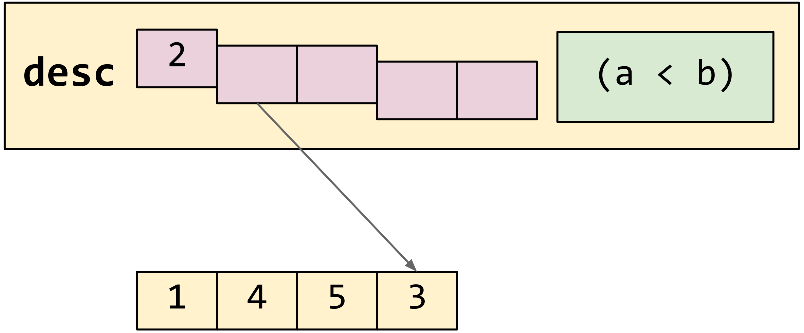

```
std::priority_queue<int, std::vector<int>, X> desc(X{false}, {5,4,3,2,1});
std::priority_queue<int, std::vector<int>, X> asc(X{true}, {1,3,2,5,4});
try { desc = asc; } catch (...) {}
```

There are other ways to break PQ's invariant

https://godbolt.org/z/8pJ4JE

- A comparator whose assignment operator throws is also trouble for `priority_queue`

```
struct X {
    bool up;
    X& operator=(const X&) { throw "oops"; }
    bool operator()(int a, int b) const {
        return up ? (b < a) : (a < b);
    }
};
```



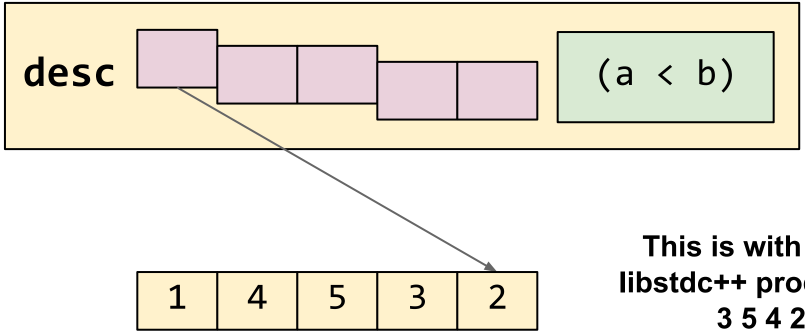

```
std::priority_queue<int, std::vector<int>, X> desc(X{false}, {5,4,3,2,1});
std::priority_queue<int, std::vector<int>, X> asc(X{true}, {1,3,2,5,4});
try { desc = asc; } catch (...) {}
```

There are other ways to break PQ's invariant

https://godbolt.org/z/8pJ4JE

- A comparator whose assignment operator throws is also trouble for `priority_queue`

```
struct X {
    bool up;
    X& operator=(const X&) { throw "oops"; }
    bool operator()(int a, int b) const {
        return up ? (b < a) : (a < b);
    }
};
```



This is with libc++. libstdc++ produces "1 3 5 4 2."

```
std::priority_queue<int, std::vector<int>, X> desc(X{false}, {5,4,3,2,1});
std::priority_queue<int, std::vector<int>, X> asc(X{true}, {1,3,2,5,4});
try { desc = asc; } catch (...) {}
```

https://rextester.com/XNXP31788

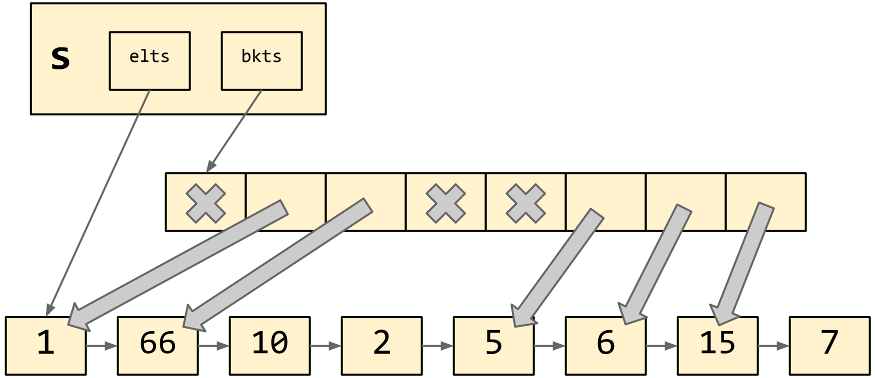
MSVC’s `unordered_set`

- Hat tip to Billy O’Neal for this example
- Billy reports that this is ***fixed*** in latest MSVC STL
  - which was just released on GitHub, by the way!
- MSVC’s `unordered_set` is (still) implemented by composition of two more primitive containers:
  - A linked list of elements — from `us.begin()` to `us.end()`
  - A vector of list iterators — `local_iterator begin(0), begin(1),` etc.
- These two components can get out of sync in the same way as `priority_queue`’s two components

https://rextester.com/XNXP31788

MSVC's unordered_set

```
struct Hasher {
    size_t operator()(int i) const {
        return i;
    }
};

std::unordered_set<int, Hasher> s =
    {1, 2, 5, 6, 7, 10, 15, 66};
```



Looking up an element involves hashing it (to give h), accessing `bkts[h]`, and then walking the linked list until you reach `bkts[h+1]`.

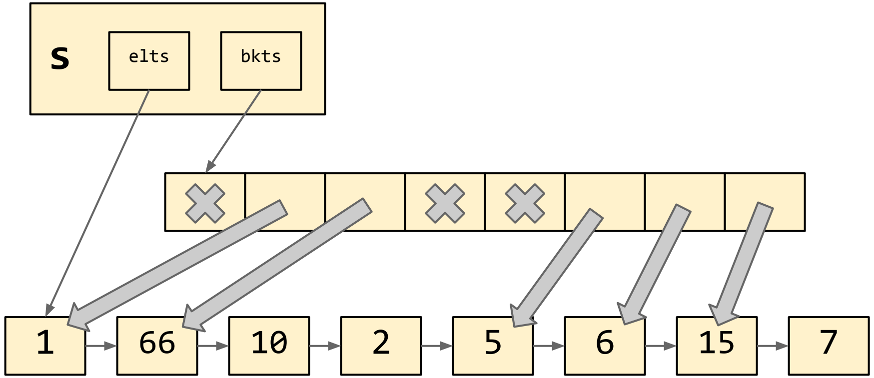
```
auto it = s.find(10);
```

Details of the linked-list walk have been slightly simplified for presentation.

https://rextester.com/XNXP31788

MSVC's `unordered_set`

When the `unordered_set`'s load factor gets too high, we resize the `bkts` vector and rehash all the existing elements.



In this particular example, inserting a new element with value `9` will trigger a rehash from 8 buckets to 64 buckets.

During the rehash, we'll update the iterators in `bkts` to point to the first element of each new bucket.

We may need to shuffle the list to regroup ranges of elements whose hash values are equal mod 8 but not equal mod 64. (For example, `66 10 2` becomes `2 66 10`.)

https://rextester.com/XNXP31788

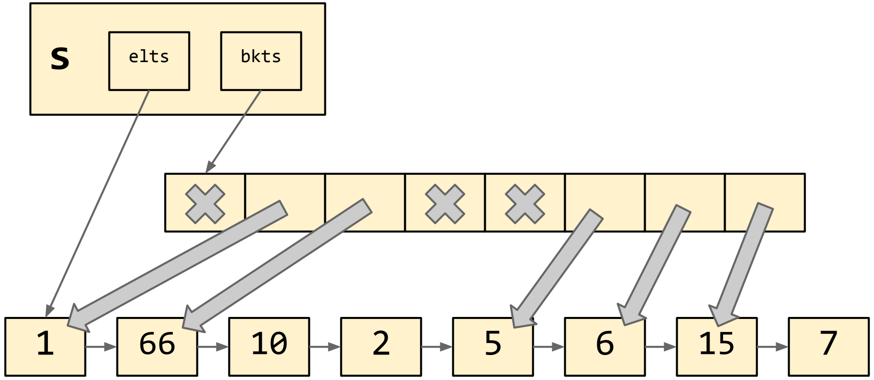
hasher::operator() can throw!

```
static int throw_on = 0;

struct Hasher {
    size_t operator()(int i) const {
        if (throw_on == i) throw "oops";
        return i;
    }
};

int main() {
    std::unordered_set<int, Hasher> s = {1, 2, 5, 6, 7, 10, 15, 66};
    throw_on = 5;
    try { s.emplace(9); } catch (...) {}
```

https://rextester.com/XNXP31788

hasher::operator() can throw

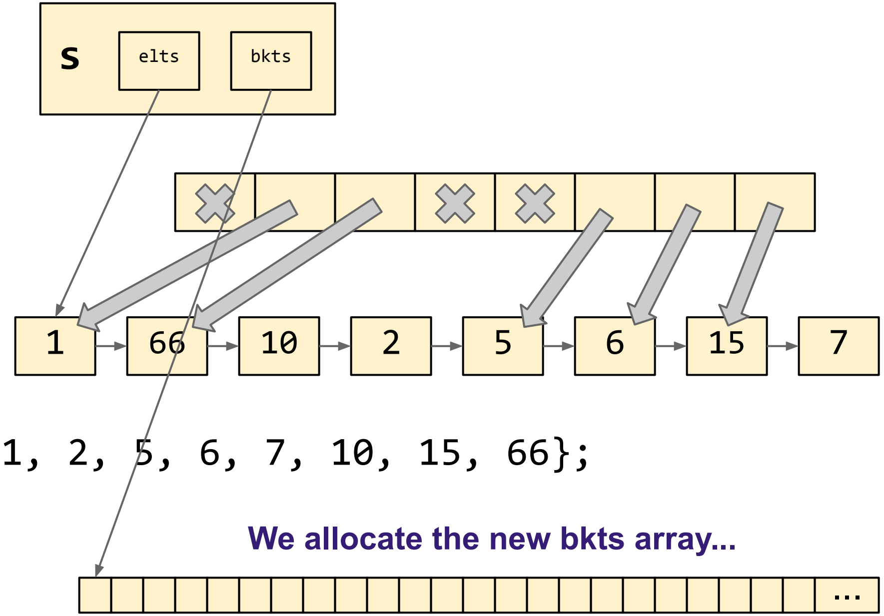
```
static int throw_on = 0;

struct Hasher {
    size_t operator()(int i) const {
        if (throw_on == i) throw "oops";
        return i;
    }
};

int main() {
    std::unordered_set<int, Hasher> s = {1, 2, 5, 6, 7, 10, 15, 66};
    throw_on = 5;
    try { s.emplace(9); } catch (...) {}
```



https://rextester.com/XNXP31788

hasher::operator() can throw

```
static int throw_on = 0;

struct Hasher {
    size_t operator()(int i) const {
        if (throw_on == i) throw "oops";
        return i;
    }
};

int main() {
    std::unordered_set<int, Hasher> s = {1, 2, 5, 6, 7, 10, 15, 66};
    throw_on = 5;
    try { s.emplace(9); } catch (...) {}
```

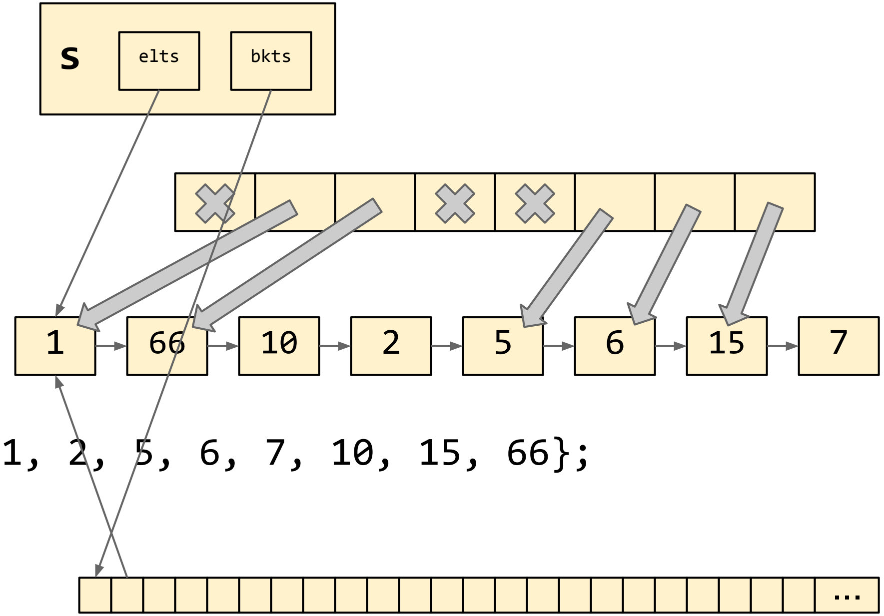


We allocate the new bkts array...

...and we begin the rehash.

hasher::operator() can throw

```
static int throw_on = 0;

struct Hasher {
    size_t operator()(int i) const {
        if (throw_on == i) throw "oops";
        return i;
    }
};

int main() {
    std::unordered_set<int, Hasher> s = {1, 2, 5, 6, 7, 10, 15, 66};
    throw_on = 5;
    try { s.emplace(9); } catch (...) {}
```

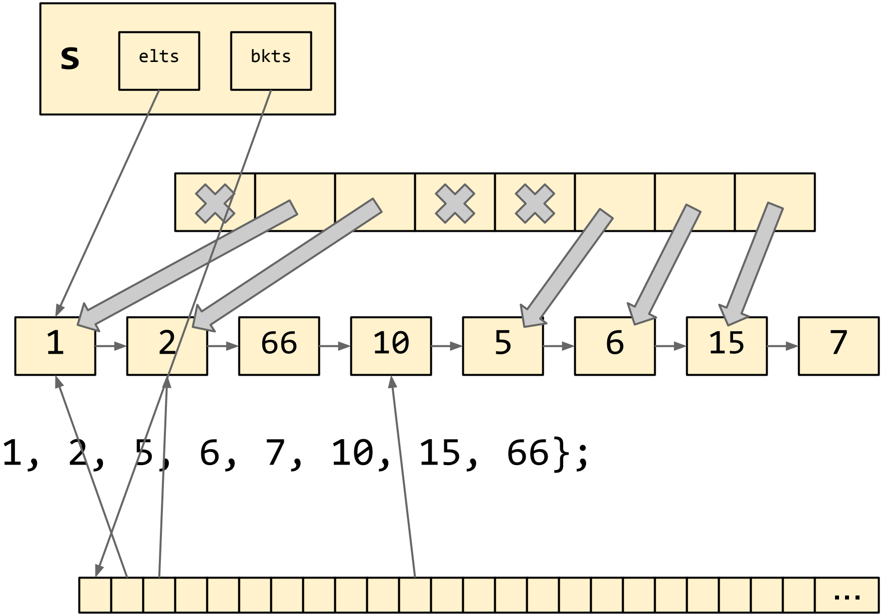


Hash 1.

https://rextester.com/XNXP31788

hasher::operator() can throw

```
static int throw_on = 0;

struct Hasher {
    size_t operator()(int i) const {
        if (throw_on == i) throw "oops";
        return i;
    }
};

int main() {
    std::unordered_set<int, Hasher> s = {1, 2, 5, 6, 7, 10, 15, 66};
    throw_on = 5;
    try { s.emplace(9); } catch (...) {}
```

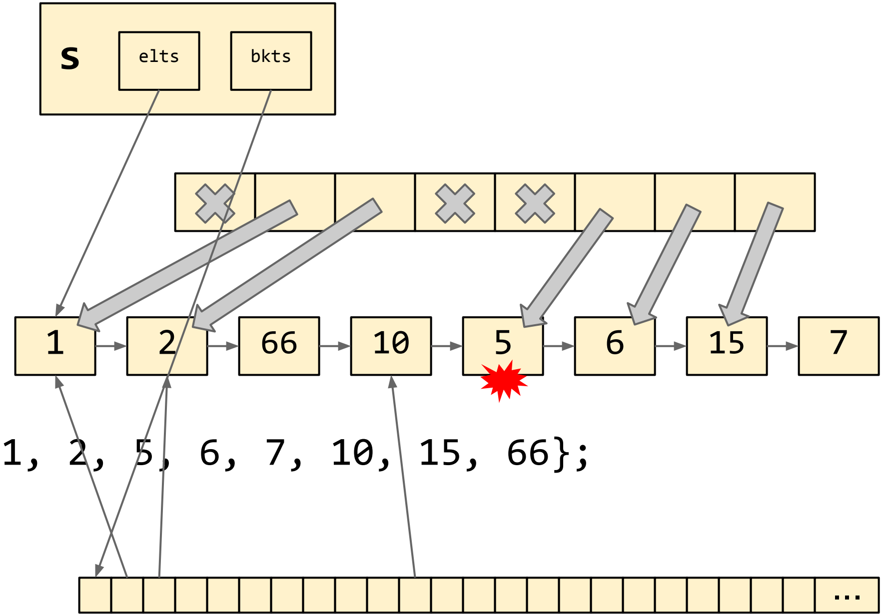


Hash 66, 10, and 2.
Reorder them appropriately.
(Some details omitted.)

https://rextester.com/XNXP31788

hasher::operator() can throw

```
static int throw_on = 0;

struct Hasher {
    size_t operator()(int i) const {
        if (throw_on == i) throw "oops";
        return i;
    }
};

int main() {
    std::unordered_set<int, Hasher> s = {1, 2, 5, 6, 7, 10, 15, 66};
    throw_on = 5;
    try { s.emplace(9); } catch (...) {}
```



Hash 5.
This throws an exception!
The bkts vector is left half-unfilled.

Here’s the punch line:

```
static int throw_on = 0;

struct Hasher {
    size_t operator()(int i) const {
        if (throw_on == i) throw "oops";
        return i;
    }
};

int main() {
    std::unordered_set<int, Hasher> s = {1,2,3,4,5,6,7,8};
    throw_on = 5;
    try { s.emplace(9); } catch (...) {}
    auto it1 = std::find(s.begin(), s.end(), 6);          // linear search is OK
    auto it2 = s.find(6);                          // yet the bucket appears empty
    assert(it1 != it2);  // Surprise!
}
```

“Fixed in master.”

```
static int throw_on = 0;

struct Hasher {
    size_t operator()(int i) const {
        if (throw_on == i) throw "oops";
        return i;
    }
};

int main() {
    std::unordered_set<int, Hasher> s = {1,2,3,4,5,6,7,8};
    throw_on = 5;
    try { s.emplace(9); } catch (...) {}

    assert(s.size() == 0);  // Um, still surprise?
}
```

What about `flat_set` and `flat_map`?

- P1221 `flat_set` has this kind of problem in abundance
  - If container assignment succeeds but comparator assignment throws (or v/v)
  - If comparison throws, during insertion/deletion (our PQ example)
  - If element assignment/swap throws, during insertion/deletion
- Due to its unusual number of “cross-component” invariants
  - The container must be sorted by the comparator
  - The container must not contain duplicates
- P0429 `flat_map` has the same issues, plus more
  - It has one ***more*** cross-component invariant:
  - The key container and the value container must be in sync
  - If key insertion succeeds but value insertion throws...

What about `flat_set` and `flat_map`?

A pernicious case for P0429 is `flat_map::extract` —

`containers extract() &&`

Returns: `std::move(c)`

Effects: `*this` is emptied, even if the function is exited via exception.

Moving-out-of `c` leaves `c.keys` and `c.values` in their moved-from states. These states might not be compatible. The overall `flat_map` might be in an *invalid* state.

So P0429 mandates that `extract()` must take some extra cycles to ensure that both containers are actually cleared *after* being moved-from. This is contrary to the “move is fast” philosophy.

What about `flat_set` and `flat_map`?

Another pernicious case is `flat_map::insert`.

```
template<class InputIterator>
void insert(InputIterator first, InputIterator last);
```

Effects: Adds elements to `c` as if by

```
    for (; first != last; ++first) {
        c.keys.insert(std::end(c.keys), first->first);
        c.values.insert(std::end(c.values), first->second);
    }
```

; sorts the range of newly inserted elements with respect to `value_comp()`;
merges the resulting sorted range and the sorted range of pre-existing elements into a single sorted range;
and finally erases the range [`ranges::unique(*this, key_equiv(compare))`, `end()`).

Typical LWG response:

“Restore the class invariant by any means necessary.”

P1843 “Comparison and Hasher Req’ts”

LWG 2189 “Throwing swap breaks unordered containers’ state.”

Billy O’Neal (who helped greatly with this talk — but all mistakes and misrepresentations are my own) wrote the paper P1843, which proposes, in part:

In [priqueue.members], add:

`void swap(priority_queue& q) noexcept(is_nothrow_swappable_v<Container>)`

-?- *Constraints:* `is_swappable_v<Container>` is `true` and `is_swappable_v<Compare>` is `true`.

-?- *Expects*: If swapping `this->c` with `q.c` throws an exception, either there are no effects on the containers, or they both contain 0 elements. Swapping `this->comp` and `q.comp` shall not exit via an exception.

-?- *Effects:* Exchanges the contents of `*this` and `q` by: `using std::swap; swap(c, q.c); swap(comp, q.comp);`

P1843 “Comparison and Hasher Req’ts”

LWG 2189 “Throwing swap breaks unordered containers’ state.”

Billy O’Neal (who helped greatly with this talk — but all mistakes and misrepresentations are my own) wrote the paper P1843, which proposes, in part:

In [priqueue.members], add:

`void swap(priority_queue& q) noexcept(is_nothrow_swappable_v<Container>)`

-?- *Constraints:* `is_swappable_v<Container>` is `true` and `is_swappable_v<Compare>` is `true`.

-?- *Expects*: If swapping `this->c` with `q.c` throws an exception, either there are no effects on the containers, or they both contain 0 elements. Swapping `this->comp` and `q.comp` shall not exit via an exception.

-?- *Effects:* Exchanges the contents of `*this` and `q` by: `using std::swap; swap(c, q.c); swap(comp, q.comp);`

“Nuke it from orbit” seems user-hostile

The current “buggy” behavior of `priority_queue` is ***exactly what any working programmer would expect***, based on an “STL 101” explanation of how `priority_queue` works, plus their knowledge of the Rule of Zero.

```
template<class Container, class Comparator>
class priority_queue {
protected:
    Container c; Comparator comp;
public:
    priority_queue& operator=(const priority_queue&) = default;
    priority_queue& operator=(priority_queue&&) = default;
```

“Nuke it from orbit” seems user-hostile

Clearing the container certainly restores `priority_queue`’s invariant...
...but it also destroys all the programmer’s data!

And adds effectively dead code. And nukes the hope of trivial copyability (except via ugly metaprogramming).

```
priority_queue& operator=(const priority_queue& rhs) {
    c = rhs.c;
    try {
        comp = rhs.comp;
    } catch (...) {
        c.clear();            // Yuck!
        throw;
    }
}
```

We don’t have many tools at our disposal

How `flat_map` discussion inevitably goes:

“When X happens, we must either break the container invariant, or nuke the container.”

“Hmm, those are both terrible. Let’s spend 10 minutes thinking about clever algorithmic hacks that might prevent X from happening in the first place.”

- Can we front-load all the possibly failing operations?
- Can we rely on nothrow swap, nothrow move-assignment, etc.?

We don’t have many tools at our disposal

P0429R6 (now superseded) had even tried this:

21.6.8.8 Specialized algorithms [flatmap.special]

```
template<class Key, class T, class Compare, class KeyContainer, class MappedContainer>
  void swap(flat_map<Key, T, Compare, KeyContainer, MappedContainer>& x,
            flat_map<Key, T, Compare, KeyContainer, MappedContainer>& y) noexcept;
```

1 *Constraints:* `is_nothrow_swappable_v<KeyContainer> && is_nothrow_swappable_v<MappedContainer> && is_nothrow_swappable_v<Compare>` is `true`.

2 *Effects:* Equivalent to: `x.swap(y)`.

“If any of my components seem like they ***might*** throw during swap, then I simply won’t provide swappability at all.”

(This wording has, thankfully, vanished from P0429R7.)

At first glance, this is likely a job for UB

- How often is an exception ever thrown from `hasher::operator()`, `compare::operator()`, etc?
- We should probably just say that programs which throw from those functions have undefined behavior.
  - Do not require `std::hash<T>::operator()` to be marked `noexcept`. That would be a breaking change for much real-world code.

- On the other hand, copy-assignment can quite plausibly throw. If an exception is thrown by `hasher::operator=`, maybe we would be justified in nuking `lhs`.

Exception guarantees in the standard?

“Strong guarantee” — Either the operation succeeds, or there is no effect.

- This would be a reliable building block.
- But the STL has no way for users to indicate “I provide this!”

“Basic guarantee” — Either the operation succeeds, or the component enters a valid but otherwise unspecified state.

- This is generally the default for STL objects after a throw.
- But this is not a reliable building block.

Exception guarantees in the standard?

"No guarantee" — Either the operation succeeds, or the component enters an unspecified and possibly broken state.

- If any of your components give only the basic guarantee, ***and*** you have cross-component invariants, then you end up here.
- If an exception is thrown ***from*** a user-provided component ***through*** an STL object with cross-component invariants, then:
  - The STL doesn't know the user-provided component's state (unless the user-provided component clearly gives the strong guarantee)
  - Therefore the STL object likely ends up in an "invalid" state
  - Until it takes off and nukes its contents from orbit

Is there a way forward?

- I suspect the answer will involve codifying the notion of strong exception guarantee into the standard library clauses.
- Certain user-provided operations (e.g. `hasher::operator=`) should be required to provide the strong exception guarantee.
  - When the STL provides a class that is likely to be used in such a role (e.g. `std::function`), that class's relevant operations should come with the strong exception guarantee.
- We already require the ***no-throw guarantee*** of allocators' relevant operations (e.g. move, copy, swap).
  - Should probably also require it of `hasher::operator()` etc.

Questions?