Painless C++ comparators

About me

- Yuri Gribov
- Compiler engineer
- Gmail: tetra2005
- t.me/the_real_yugr
- https://github.com/yugr
- https://www.linkedin.com/in/yugr/

Plan

- What is a comparator
- Example of UB
- Comparator axioms
- Common mistakes
- And ways to diagnose them

Comparators

- Generalization of `operator<`
- Predicate functor for comparing objects of some class/type
- Used by various STL algorithms and containers to sort and search objects

```
auto comp = [](T x, T y) { return pr(x) < pr(y); };
std::sort(begin, end, comp);

std::sort(begin, end);  // Uses operator<
```

Usage of comparators

- Standard containers:
  - `std::map, std::multimap`
  - `std::set, std::multiset`
- Standard algorithms:
  - `std::sort, std::stable_sort`
  - `std::binary_search, std::equal_range, std::{lower,upper}_bound`
  - `std::{min,max}_element, std::nth_element`
  - etc.

Example of comparator

```
int main() {
  std::vector<int> vals;
  srand(0);
  for (size_t i = 0; i < SIZE; ++i)
    vals.push_back((double)rand() / RAND_MAX * 10);
  std::sort(vals.begin(), vals.end(),
            [](int l, int r) { return l <= r; });
  for (auto v : vals)
    std::cout << v << "\n";
  return 0;
}
```

Does it work?

```
$ g++ -g -DSIZE=10 bad.cc && ./a.out
1
2
3
3
5
7
7
7
8
9
```

It does not…

```
$ g++ -g -DSIZE=50 bad.cc && ./a.out
1
4
1
1
9
2
5
...
double free or corruption (out)
Aborted
```

Buffer overflow!

```
$ g++ -g -DSIZE=50 -fsanitize=address -D_GLIBCXX_SANITIZE_VECTOR=1 bad.cc && ./a.out
=================================================================
==143607==ERROR: AddressSanitizer: container-overflow on address 0x611000000108
READ of size 4 at 0x611000000108 thread T0
    #0 0x55fa93254d5c in operator()<__gnu_cxx::__normal_iterator<int*, std::vector<int> >,
__gnu_cxx::__normal_iterator<int*, std::vector<int> > > /usr/include/c++/10/bits/predefined_ops.h:156
    #1 0x55fa93255164 in __unguarded_partition<__gnu_cxx::__normal_iterator<int*, std::vector<int> >,
__gnu_cxx::__ops::_Iter_comp_iter<main()::<lambda(int, int)> > > /usr/include/c++/10/bits/stl_algo.h:1904
    #2 0x55fa9325428b in __unguarded_partition_pivot<__gnu_cxx::__normal_iterator<int*, std::vector<int> >,
__gnu_cxx::__ops::_Iter_comp_iter<main()::<lambda(int, int)> > > /usr/include/c++/10/bits/stl_algo.h:1926
    #3 0x55fa93253d1f in __introsort_loop<__gnu_cxx::__normal_iterator<int*, std::vector<int> >, long int,
__gnu_cxx::__ops::_Iter_comp_iter<main()::<lambda(int, int)> > > /usr/include/c++/10/bits/stl_algo.h:1958
    #4 0x55fa93253d3d in __introsort_loop<__gnu_cxx::__normal_iterator<int*, std::vector<int> >, long int,
__gnu_cxx::__ops::_Iter_comp_iter<main()::<lambda(int, int)> > > /usr/include/c++/10/bits/stl_algo.h:1959
    #5 0x55fa93253d3d in __introsort_loop<__gnu_cxx::__normal_iterator<int*, std::vector<int> >, long int,
__gnu_cxx::__ops::_Iter_comp_iter<main()::<lambda(int, int)> > > /usr/include/c++/10/bits/stl_algo.h:1959
    #6 0x55fa93253a6f in __sort<__gnu_cxx::__normal_iterator<int*, std::vector<int> >,
__gnu_cxx::__ops::_Iter_comp_iter<main()::<lambda(int, int)> > > /usr/include/c++/10/bits/stl_algo.h:1974
    #7 0x55fa932537fb in sort<__gnu_cxx::__normal_iterator<int*, std::vector<int> >, main()::<lambda(int, int)> >
/usr/include/c++/10/bits/stl_algo.h:4894
    #8 0x55fa932534ca in main /home/yugr/tasks/CppRussia/bad.cc:11
    #9 0x7f9bba05ad09 in __libc_start_main ../csu/libc-start.c:308
    #10 0x55fa93253249 in _start (/home/yugr/tasks/CppRussia/a.out+0x2249)
```

Root cause

- Partitioning array by pivot element (main step of quicksort)

```
_RandomAccessIterator
__unguarded_partition(_RandomAccessIterator __first,
                      _RandomAccessIterator __last,
                      T __pivot, _Compare __comp) {
  while (true) {
    // There must be an element which is not less than __pivot
    while (__comp(*__first, __pivot))
      ++__first;
    --__last;
    while (__comp(__pivot, *__last))
      --__last;
    if (!(__first < __last))
      return __first;
    std::iter_swap(__first, __last);
    ++__first;
  }
}
```

Root cause

- `__pivot` is selected as median of first, second and last array elements
- So on loop entry we have `a` and `b` such that

  ```
  __comp(a, __pivot) && __comp(__pivot, b)
  ```
- This condition is a loop invariant
- In theory we can conclude a followup condition which guarantees loop termination:

  ```
  !__comp(__pivot, a) && !__comp(b, __pivot)
  ```

Simplification

Root cause

For our comparator

```
auto comp = [](int l, int r) { return l <= r; }
```

the implication is wrong:

```
comp(__pivot, b) ⇒ !comp(b, __pivot)
```

when `__pivot == b`.

Thus the loop invariant is violated which results in buffer overflow.

Comparator requirements

Comparator requirements

- To avoid errors in standard algorithms comparators must meet several requirements (axioms)
- These requirements are specified in Standard: bit.ly/3LpH5Nc
- Violation of axioms leads to Undefined Behavior (crashes, invalid results, hangs)
- Not specific to C++: C, Java, Lua, Swift, JavaScript и Rust

Strict partial ordering axioms

- Irreflexivity:

  ```
  !comp(a, a)
  ```

- Asymmetry:

  ```
  comp(a, b) ⇒ !comp(b, a)
  ```

- Transitivity:

  ```
  comp(a, b) && comp(b, c) ⇒ comp(a, c)
  ```

- In algebra such comparators are called *strict partial orderings* and corresponding sets/classes are called *partially ordered* (posets)

Equivalence relation

- Any comparator has a corresponding “equivalence function” (equivalence relation):

```
bool equiv(T a, T b) {
  return !comp(a, b) && !comp(b, a);
}
```

- Also known as “incomparability relation”
- Shows whether two elements are indiscernible by comparator
- Behaves similar to `operator==` but is different

Transitivity of equivalence

- Equivalence relation must be transitive:

  `equiv(a, b) && equiv(b, c) ⇒ equiv(a, c)`
- Objects of class can be partitioned to groups of equivalent elements
- Elements of the group behave similarly in comparisons:
  - Comparing any element of the group to any other element of the set gives the same result

Transitivity of equivalence

- Necessary condition for all “quick” sorting algorithms
  - Why do we need transitivity of equivalence
- Not all STL algorithms really need transitivity of equivalence!
  - E.g. partial ordering is enough for std::min/min_element
  - But Standard requires all 4 axioms for all algorithms (except for std::binary_search and friends)
  - Presumably to simplify things

Strict weak ordering

- Strict weak ordering
  - Partial ordering + transitivity of equivalence
- Excerpts from n4868:
  - alg.sorting:
    - For algorithms other than those described in [alg.binary.search], comp shall induce a **strict weak ordering** on the values.
  - utility.arg.requirements (Cpp17LessThanComparable):
    - < is a **strict weak ordering** relation

Common errors

Common errors: invalid lexicographical ordering

- Most common error
- Violation of asymmetry axiom:

```
A(100, 2) < A(200, 1) &&
A(200, 1) < A(100, 2)
```

```
bool operator<(const A &rhs) {
  if (x < rhs.x)
    return true;
  else if (y < rhs.y)
    return true;
  else
    return false;
}
```

Common errors: invalid lexicographical ordering

- Simple fix:

```
if (x < rhs.x)
  return true;
else if (rhs.x < x)
  return false;
else if (y < rhs.y)
  return true;
else
  return false;
```

Common errors: invalid lexicographical ordering

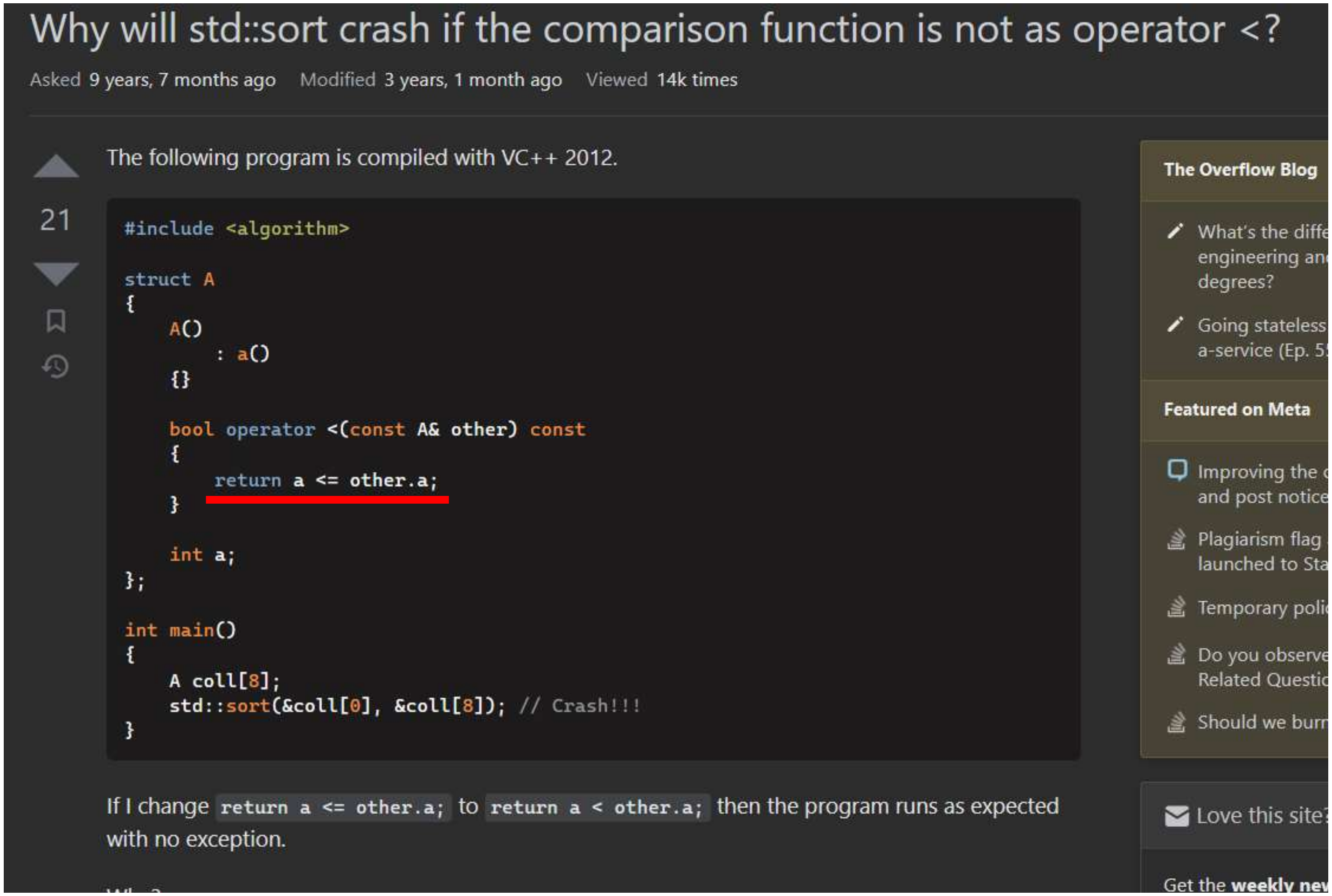
- Better options:
  - use `std::tie` and builtin comparison of tuples:

```
return std::tie(lhs.x, lhs.y) < std::tie(rhs.x, rhs.y);
```

  - (C++20) use default implementation of `operator<`:

```
bool operator<(const SomeClass &) const =
default;
```

Common errors: non-strict ordering



Why will std::sort crash if the comparison function is not as operator <?

Asked 9 years, 7 months ago Modified 3 years, 1 month ago Viewed 14k times

21

The following program is compiled with VC++ 2012.

```
#include <algorithm>

struct A
{
    A()
        : a()
    {}

    bool operator <(const A& other) const
    {
        return a <= other.a;
    }

    int a;
};

int main()
{
    A coll[8];
    std::sort(&coll[0], &coll[8]); // Crash!!!
}
```

If I change `return a <= other.a;` to `return a < other.a;` then the program runs as expected with no exception.

Violation of irreflexivity and asymmetry

Common errors: negation of strict ordering is non-strict

- Another variant of this error:

```
auto lt = std::less<int>();
auto inv_lt = std::not2(lt);
std::sort(..., ..., inv_lt);
```

- Negation of strict ordering is a *non-strict* ordering (and thus violates the asymmetry axioms)
- Real-world example:
  - Bug hunting fun with std::sort

Common errors: NaN

```
int main() {
  double a[] = {
    100, 5, 3, NAN, 200, 11
  };
  std::sort(&a[0], &a[std::size(a)]);
  for (auto x : a)
    std::cout << x << "\n";
  return 0;
}
```

Common errors: NaN

```
$ g++ bad.cc && ./a.out
3
5
100
nan
11
200
```

Common errors: NaN

- Floating-point types support special NaN values
- Generated during incorrect FP computations
  - E.g. sqrt of negative number or dividing `0/0`
- Comparison with NaN always returns false so NaNs are equivalent to all other numbers
- This causes intransitivity of equivalence:
  - `NAN ~ 1.0`
  - `NAN ~ 2.0`
  - But `! 1.0 ~ 2.0`

Common errors: NaN

- Get rid of NaNs before sorting via std::partition:

```
auto end = std::partition(&a[0],
                          &a[std::size(a)],
                          [](double x) {
                            return !isnan(x); });
std::sort(&a[0], end);
```

- In case of std::map wrap floats in a class with overloaded comparison

Common errors: approximate comparisons

```
bool cmp(double a, double b) {
    if (abs(a - b) < eps) return false;
    return a < b;
}
```

- Treat “close” numbers as equivalent
- But this violates transitivity of equivalence:

```
equiv(0, 0.5 * eps) == true
equiv(0.5 * eps, eps) == true
cmp(0, eps) == false
```

Common errors: special case

```
[](std::unique_ptr<SomeClass> a,
   std::unique_ptr<SomeClass> b) {
  if (!a.get())
    return true;
  else if (!b.get())
    return false;
  else
    return *a < *b;
}
```

Violation of irreflexivity and asymmetry when both operands are nullptrs

Common errors: special case

Generic pattern:

```
[](A lhs, A rhs) {
  if (pred(lhs))
    return true;
  ...
}
```

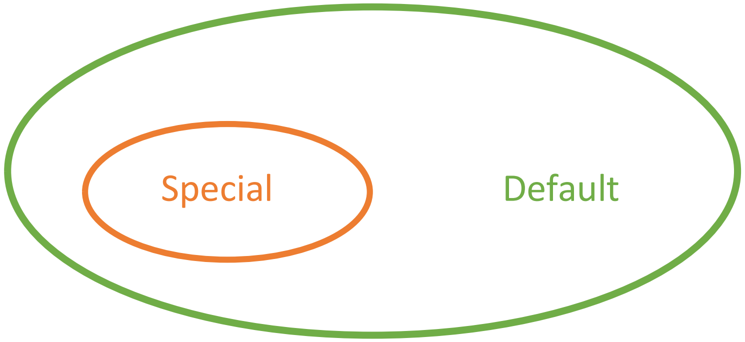
Generic fix:

```
[](A lhs, A rhs) {
  if (pred(lhs) || pred(rhs))
    return pred(rhs) < pred(lhs);
  ...
}
```

Common errors: custom comparator for “special” objects

Generic pattern:

```
auto comp = [](Object a, Object b) {
  if (is_special(a) && is_special(b))
    return comp_special(a, b);
  else
    return comp_default(a, b);
}
```



Common errors: custom comparator for “special” objects

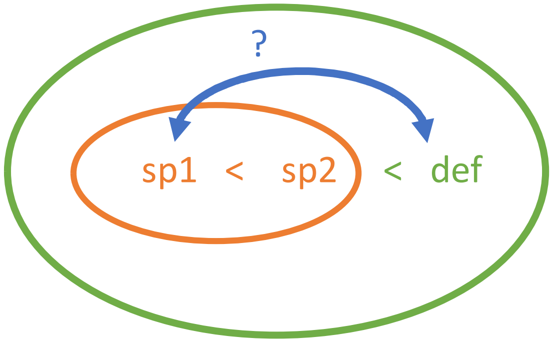
- Transitivity axiom:

```
comp(sp1, sp2) && comp(sp2, def) ⇒
comp(sp1, def)
```

- Substitute terms:

```
comp_special(sp1, sp2) &&
comp_default(sp2, def) ⇒
comp_default(sp1, def)
```

- Comp_special и comp_default are usually logically and algorithmically unrelated and implication does not hold



Common errors: custom comparator for “special” objects

- Violation of transitivity:
  - `A(true, 1, 3) < A(true, 2, 1)`
  - `A(true, 2, 1) < A(false, 1, 2)`
  - `! A(true, 1, 3) < A(false, 1, 2)`
- Real-world example:
  - [GCC Bugzilla #68988]

```
class A {
  bool special;
  int x, y;
  bool operator<(A rhs) {
    if (special && rhs.special)
      return x < rhs.x;
    return y < rhs.y;
  }
};
```

Common errors: custom comparator for “special” objects

Generic fix:

```
auto comp = [](Object a, Object b) {
  if (is_special(a) != is_special(b))
    return is_special(a) < is_special(b);
  else if (is_special(a) && is_special(b))
    return comp_special(a, b);
  else
    return comp_default(a, b);
}
```

Tooling

Debug mode in libstdc++

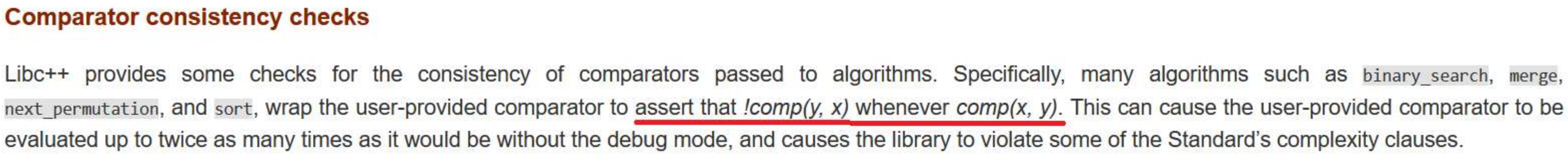
- Libstdc++ uses macro `-D_GLIBCXX_DEBUG` to enable irreflexivity checks
- Would have found error from the beginning of this presentation

```
$ cat /usr/include/c++/10/bits/stl_algo.h
...
inline void
sort(_RandomAccessIterator __first, _RandomAccessIterator __last,
     _Compare __comp)
{
  ...
  __glibcxx_requires_irreflexive_pred(__first, __last, __comp);

  std::__sort(__first, __last, __gnu_cxx::__ops::__iter_comp_iter(__comp));
}
...
```

Debug mode in libc++

- Libc++ uses macro `-D_LIBCPP_ENABLE_DEBUG_MODE` to enable asymmetry checks:



Comparator consistency checks

Libc++ provides some checks for the consistency of comparators passed to algorithms. Specifically, many algorithms such as `binary_search`, `merge`, `next_permutation`, and `sort`, wrap the user-provided comparator to assert that *!comp(y, x)* whenever *comp(x, y)*. This can cause the user-provided comparator to be evaluated up to twice as many times as it would be without the debug mode, and causes the library to violate some of the Standard's complexity clauses.

Debug mode

- Both options have significant (2x) overhead and should be used only for testing
- Checkers can not change the algorithmic complexity of std::sort
  - O(N*logN)
- Thus full correctness can not be checked
  - E.g. violation of transitivity axioms is O(N^3)

SortChecker++

- https://github.com/yugr/sortcheckxx
- Dynamic checker that verifies comparators in C++ code
- Intercepts and checks STL APIs like `std::sort` and `std::map`-like containers
- Based on source-to-source instrumentation via Clang
- Found 5 errors in OSS projects
- TODO: support all relevant algorithms (`nth_element`, etc.)

SortChecker

- https://github.com/yugr/sortcheck
- Dynamic checker that verifies comparators in C code
- Intercepts and checks libc API like `qsort` и `bsearch`
- Based on runtime instrumentation via `LD_PRELOAD`
- Found 15 errors in various OSS projects (GCC, Harfbuzz, etc.)

How to use SortChecker++

- Instrument code:

```
$ sortcheckxx/bin/SortChecker bad.cc -- -DSIZE=50
```

```
std::sort(begin, end, cmp);
```

→

```
sortcheck::sort_checked(begin, end,
  cmp, __FILE__, __LINE__);
```

- Compile and run instrumented code:

```
$ g++ -g -DSIZE=50 -Isortcheckxx/include bad.cc &&
./a.out
sortcheck: bad.cc:14: irreflexive comparator at
position 0
Aborted
```

Pseudocode

- Each run of `std::sort` (or similar APIs) is preceded by these checks:

```
for x in array
  if comp(x, x)
    error

for x, y in array
  if comp(x, y) != comp(y, x)
    error

for x, y, z in array
  if comp(x, y) && comp(y, z) && !comp(x, z)
    error

for x, y, z in array
  if equiv(x, y) && equiv(y, z) && !equiv(x, z)
    error
```

- Complexity is $O(N^3)$
- Too large even for `std::sort` so only small array prefix is verified (30 elements)

Fast verification algorithm

- https://github.com/danlark1/quadratic_strict_weak_ordering
- Proposed by D. Kutenin in January 2023
- Idea of the algorithm:
  - Sort array by stable algorithm
  - Go over prefixes of equivalent elements in sorted array
  - Verify their transitivity with remaining part of the array
- Complexity is $O(N^2)$ which is still larger than std::sort's O(N*logN)
- Will likely be integrated in debug libc++ (D150264) and SortChecker

Reading materials

- Danila Kutenin Changing std::sort at Google’s Scale and Beyond
- Jonathan Müller Mathematics behind Comparison

Recommendations

- Avoid common errors
- Turn on `_GLIBCXX_DEBUG` и `_LIBCPP_ENABLE_DEBUG_MODE` in your CI
- Apply SortChecker и SortChecker++ to your codebase
  - Bug reports and feedback are welcome
  - As well as stars on GitHub :)

Other types of errors in comparator API

- Unsorted arrays in `std::binary_search` and friends
  - Diagnosed by SortChecker/SortChecker++
- Relying on particular order of equivalent elements in sorted array
  - Diagnosed by debug libc++ (`−D_LIBCPP_ENABLE_DEBUG_MODE`) via randomization
  - Randomization also helps to provoke other types of bugs

Thank you for attending!

Spaceship operator and comparison categories

- Standard is changing towards explicit representation of orderings in the language
- C++20 has a new comparison operator: `operator<=>`
  - Reduces amount of code needed to implement all comparison operators (==, !=, <, >, <=, >=)
  - Allows for more efficient code
- Returns value of one of three types (comparison categories) depending on ordering type implemented by the class:
  - `std::partial_ordering`
  - `std::weak_ordering`
  - `std::strong_ordering`

Semantics of comparison categories?

- One could expect that selection of particular category gives guarantees about class behavior
  - `std::partial_ordering` – class is partially ordered ?
  - `std::weak_ordering` – class is weakly ordered ?
- In fact not guaranteed by the current Standard
- Selection of category does not give *any* guarantees of behavior
  - Implied meaning of ordering types

Common errors: invalid lexicographical ordering

Examples:

- https://stackoverflow.com/questions/48455244/bug-in-stdsort
- https://stackoverflow.com/questions/53712873/sorting-a-vector-of-a-custom-class-with-stdsort-causes-a-segmentation-fault
- https://stackoverflow.com/questions/68225770/sorting-vector-of-pair-using-lambda-predicate-crashing-with-memory-corruption
- https://stackoverflow.com/questions/72737018/stdsort-results-in-a-segfault
- https://stackoverflow.com/questions/33547566/strict-weak-ordering-operator-in-c

Common errors: non-strict ordering

- https://stackoverflow.com/questions/40483971/program-crash-in-stdsort-sometimes-cant-reproduce
- https://stackoverflow.com/questions/65468629/stl-sort-debug-assertion-failed
- https://stackoverflow.com/questions/18291620/why-will-stdsort-crash-if-the-comparison-function-is-not-as-operator
- https://stackoverflow.com/questions/19757210/stdsort-from-algorithm-crashes
- https://stackoverflow.com/questions/64014782/c-program-crashes-when-trying-to-sort-a-vector-of-strings
- https://stackoverflow.com/questions/70869803/c-code-crashes-when-trying-to-sort-2d-vector
- https://stackoverflow.com/questions/67553073/std-sort-sometimes-throws-seqmention-fault

Common errors: special case

- Example:
  - https://stackoverflow.com/questions/55815423/stdsort-crashes-with-strict-weak-ordering-comparing-with-garbage-values
  - https://stackoverflow.com/questions/48972158/crash-in-stdsort-sorting-without-strict-weak-ordering

Common errors: NaN

- Пример:
  - https://stackoverflow.com/questions/9244243/strict-weak-ordering-and-stdsort

Common errors: approximate comparison

- Example:
  - https://stackoverflow.com/questions/68114060/does-using-epsilon-in-comparison-of-floating-point-break-strict-weak-ordering