Как правильно писать компараторы

Обо мне

- Юрий Грибов
- Инженер-компиляторщик
- Gmail: tetra2005
- t.me/the_real_yugr
- https://github.com/yugr
- https://www.linkedin.com/in/yugr/

План доклада

- Что же такое компаратор
- Пример UB
- Аксиоматика компараторов
- Самые частые ошибки
- И средства их обнаружения

Компараторы

- Обобщение `operator<`
- Функторы-предикаты для сравнения элементов какого-либо типа
- Используются различными алгоритмами и контейнерами стандартной библиотеки для упорядочения/поиска объектов

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

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

Использование компараторов

- Стандартные контейнеры:
  - `std::map, std::multimap`
  - `std::set, std::multiset`
- Стандартные алгоритмы:
  - `std::sort, std::stable_sort`
  - `std::binary_search`
  - `std::equal_range, std::lower_bound, std::upper_bound`
  - `std::min_element, std::max_element, std::nth_element`
  - etc.

Пример использования компаратора

```
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;
}
```

Программа работает?

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

Или нет...

```
$ 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)
```

Причина ошибки

- Разбиение массива по опорному элементу (основной шаг быстрой сортировки)

```
_RandomAccessIterator
__unguarded_partition(_RandomAccessIterator __first,
                      _RandomAccessIterator __last,
                      T __pivot, _Compare __comp) {
  while (true) {
    // Должен найтись элемент, не меньший __pivot
    while (__comp(*__first, __pivot))
      ++__first;
    --__last;
    while (__comp(__pivot, *__last))
      --__last;
    if (!(__first < __last))
      return __first;
    std::iter_swap(__first, __last);
    ++__first;
  }
}
```

Причина ошибки

- Опорный элемент `__pivot` выбирается как медиана первого, среднего и последнего элемента массива
- Поэтому при входе в цикл всегда существуют `a` и `b`, такие что

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

- И этот инвариант сохраняется в ходе выполнения внешнего цикла
- Из этого по идее следует условие, гарантирующее отсутствие выхода за границы массива:

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

Пропедевтическое упрощение

Причина ошибки

Для компаратора

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

не выполняется условие

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

в случае `__pivot == b`.

Нарушается необходимый инвариант цикла и просходит переполнение буфера.

Требования к компараторам

Требования к компараторам

- Для избежания ошибок в работе алгоритмов сортировки компараторы должны удовлятворять набору правил (аксиом)
- Правила указаны в стандарте языка: bit.ly/3LpH5Nc

- Нарушение аксиом приводит к Undefined Behavior (аварийные завершения, некорректные результаты, зависания)
- Не специфичны для C++: C, Java, Lua, Swift, JavaScript и Rust

Аксиомы строгого частичного порядка

- Иррефлексивность:

  `!comp(a, a)`

- Антисимметричность:

  `comp(a, b) ⇒ !comp(b, a)`

- Транзитивность:

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

- В алгебре такие компараторы называют *строгими частичными порядками*, а соответствующие множества – *частично упорядоченными* (partially ordered)
  - Кратко: ЧУМ или poset

Отношение эквивалентности

- С каждым компаратором связана ещё одна функция (“отношение” в терминах алгебры):

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

- Отношение эквивалентности или несравнимости (incomparability)
- Показывает что два элемента “неразличимы” с точки зрения компаратора
- Похоже на оператор равенства, но вообще говоря отличается от `operator==`

Транзитивность эквивалентности

- Транзитивность эквивалентности:

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

- Сортируемое множество можно разбить на группы "равных" элементов
- Эти группы будут вести себя одинаково в сравнениях:
  - Сравнение любого экземпляра группы с другими элементами множества будет давать одинаковый результат независимо от выбора экземпляра

Транзитивность эквивалентности

- Необходимое условие для всех “быстрых” алгоритмов сортировки
  - Why do we need transitivity of equivalence
- Не всем алгоритмам STL требуется транзитивность эквивалентности!
  - Например для std::min/min_element достаточно частичного порядка
  - Но Стандарт требует выполнения четырёх аксиом для всех алгоритмов (вероятно для упрощения)

Strict weak ordering

- Строгий слабый порядок (strict weak ordering)
  - Частичный порядок + транзитивность эквивалентности
- Выдержки из 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

Частые ошибки

Частые ошибки: неправильный лексикографический порядок

- Самая частая ошибка при написании компараторов
- Нарушена аксиома антисимметричности:

```
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;
}
```

Частые ошибки: лексикографический порядок

- Простое исправление:

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

Частые ошибки: лексикографический порядок

- Но лучше:
  - использовать `std::tie` и встроенный оператор сравнения кортежей:

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

  - (C++20) использовать реализацию `operator<` по умолчанию:

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

Частые ошибки: нестрогий порядок

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.

The Overflow Blog

What's the diffe… engineering an… degrees?

Going stateless… a-service (Ep. 5…

Featured on Meta

Improving the … and post notice…

Plagiarism flag … launched to Sta…

Temporary poli…

Do you observe… Related Questio…

Should we burn…

Love this site?

Get the weekly new…

Нарушена иррефлексивность и антисимметричность

Частые ошибки: отрицание строгого порядка не является строгим порядком

- Другая вариация той же ошибки:

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

- Отрицание строгого порядка является *нестрогим* порядком (и нарушает аксиому антисимметричности)
- Пример из жизни:
  - Bug hunting fun with std::sort

Частые ошибки: 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;
}
```

Частые ошибки: NaN

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

Частые ошибки: NaN

- Типы с плавающей точкой поддерживают специальные значения NaN, которые возникают в результате некорректных вычислений
  - например извлечения корня из отрицательного числа или деления `0/0`
- Сравнение с NaN всегда возвращает false, поэтому NaN эквивалентен всем остальным числам
- Это приводит к нарушению транзитивности эквивалентности (4 аксиома):
  - `NAN ~ 1.0`
  - `NAN ~ 2.0`
  - Но НЕ `1.0 ~ 2.0`

Частые ошибки: NaN

- Достаточно перед сортировкой избавиться от NaN'ов с помощью std::partition:

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

- В случае std::map сделать классс-обёртку над float

Частые ошибки: некорректная обработка специального случая

```
[](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;
}
```

Нарушена иррефлексивность и антисимметричность если оба операнда нулевые

Частые ошибки: некорректная обработка специального случая

Общий паттерн ошибки:

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

Исправление:

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

Частые ошибки: сравнение особых объектов отдельным алгоритмом

- Нарушена транзитивность:
  - `A(true, 1, 2) < A(true, 2, 1)`
  - `A(true, 2, 1) < A(false, 1, 2)`
  - `! A(true, 1, 2) < A(false, 1, 2)`
- Пример из жизни:
  - 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;
  }
};
```

Частые ошибки: сравнение особых объектов отдельным алгоритмом

Общий паттерн ошибки:

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

Частые ошибки: сравнение особых объектов отдельным алгоритмом

Исправление:

```
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);
}
```

Частые ошибки: приближенные сравнения

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

- Программист хотел чтобы "близкие" элементы рассматривались как эквивалентные
- Но при этом нарушил аксиому транзитивности эквивалентности:

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

Инструменты

Отладочные средства в тулчейнах: libstdc++

- В libstdc++ с помощью макроса `-D_GLIBCXX_DEBUG` можно включить дополнительную проверку иррефлексивности
- Она бы нашла ошибку из начала презентации

```
$ 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));
}
...
```

Отладочные средства в тулчейнах: libc++

- В libc++ с помощью макроса `-D_LIBCPP_ENABLE_DEBUG_MODE` можно включить проверку асимметричности:

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.

Отладочные средства в тулчейнах

- Обе опции имеют существенные (2x) накладные расходы
- Рекомендуется использовать только для тестирования
- Чекеры компараторов не должны менять алгоритмическую сложность алгоритма O(N*logN)
- И поэтому не могут провести полную проверку корректности
  - Например проверку аксиом транзитивности

SortChecker++

- https://github.com/yugr/sortcheckxx
- Динамический чекер для проверки компараторов в программах на C++
- Перехватывает и проверяет STL API типа `std::sort` и контейнеры типа `std::map`
- Основан на source-to-source инструментации (Clang-based)
- 5 ошибок в различных OSS проектах
- TODO: поддержать все релевантные алгоритмы (`nth_element`, etc.)

SortChecker

- https://github.com/yugr/sortcheck
- Динамический чекер для проверки компараторов в программах на Си
- Перехватывает и проверяет libc API типа `qsort` и `bsearch`
- Основан на динамической инструментации (`LD_PRELOAD`)
- Нашёл 15 ошибок в различных OSS проектах (GCC, Harfbuzz, etc.)

Как использовать SortChecker++

- Вначале инструментируем код:

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

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

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

- Скомпилируем и запустим инструментированный код из начала презентации:

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

Псевдокод

- Каждый запуск `std::sort` и аналогичных API предваряется проверками:

```
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
```

- Сложность проверок составляет O(N^3)
- Существенно превосходит даже `std::sort`, не говоря о более быстрых алгоритмах (`std::max_element`, etc.)
- На практике обходится не весь массив, а его небольшое подмножество (20-30 элементов)

Быстрый алгоритм проверки

- https://github.com/danlark1/quadratic_strict_weak_ordering
- Предложен Д. Кутениным в начале 2023 года
- Идея алгоритма:
  - Предварительно отсортировать массив устойчивым алгоритмом
  - Выделять в отсортированном массиве префиксы эквивалентных элементов
  - И проверять их на транзитивность с оставшейся частью массива
- Снижает сложность до O(N^2) (по прежнему превосходит сложность проверяемых алгоритмов)
- Возможно будет интегрирован в libc++ (D150264) и SortChecker

Что почитать

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

Рекомендации

- Избегайте типичных ошибок в работе
- Включите `_GLIBCXX_DEBUG` и `_LIBCPP_ENABLE_DEBUG_MODE` в своём CI
- Примените SortChecker и SortChecker++ к своему коду
  - Сообщения об ошибках и дополнения приветствуются!

Другие типы ошибок в компараторных API

- Неотсортированные массивы в API типа `std::binary_search`
  - поддерживается в SortChecker/SortChecker++
- Расчёт на определённый порядок сортировки эквивалентных элементов
  - проверяется в отладочной libc++ (`-D_LIBCPP_ENABLE_DEBUG_MODE`) с помощью рандомизации

Спасибо за внимание!

Spaceship-оператор и comparison categories

- Стандарт движется в сторону явного представления понятия порядка в языке
- В C++20 введён новый тип оператор сравнения: `operator<=>`
  - Сокращает объём кода для реализации всех операторов сравнения (==, !=, <, >, <=, >=)
- Может возвращать значение одного из 3 типов (comparison categories) в зависимости от вида порядка, реализуемого классом:
  - `std::partial_ordering`
  - `std::weak_ordering`
  - `std::strong_ordering`

Семантика comparison categories?

- Можно было бы предположить что наличие категории даёт гарантии о поведении класса, например
  - `std::partial_ordering` – класс является частичным порядком ?
  - `std::weak_ordering` – класс является слабым порядком ?
- Но на данный момент это не гарантируется Стандартом
- Выбор той или иной категории не даёт *никаких* гарантий поведения и служит скорее для документирования
  - Implied meaning of ordering types

Частые ошибки: неправильный лексикографический порядок

Примеры:

- 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

Частые ошибки: нестрогий порядок

- 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

Частые ошибки: некорректная обработка специального случая

- Примеры:
  - 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

Частые ошибки: NaN

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

Частые ошибки: приближенные сравнения

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