Inside STL sort algorithm

Ivan Murashko • 27 Apr 2024

Introduction

One of the trickiest thing in C++ is an error catch. Most part of errors are located on the application level and you can be quite sure that the library, especially the well known STL, is error free. But what should you do if your application code is trivial i.e. it seems to be error free but you get a SIGSEGV inside a system library. Most probably you just fell into a situation that is described in the article.

Note: All examples in the article use gcc 7.4.0 from the docker image gcc:7:

```
docker run gcc:7 gcc --version
gcc (GCC) 7.4.0
```

The source code for examples can be found on the article git repository https://github.com/ivanmurashko/articles in the folder **cppsort/src**.

Example

Lets look at the following code:

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>

int main(){
  std::vector<int> data = {1, 1, 1, 1, 1, 1, 1, 1, 1,
                           1, 1, 1, 1, 1, 1, 1, 1};
  auto comp = [](int i1, int i2) { return i1 <= i2; };
  std::sort(data.begin(), data.end(), comp);
  std::copy(data.begin(), data.end(),
            std::ostream_iterator<int>(std::cout, " ")
        );
  std::cout << std::endl;
```

```
  return 0;
}
```

What output does it produce? One can expect the following one:

```
$ docker run -v "$PWD":/tmp/test -w /tmp/test gcc:7 ./
src/sort
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
$
```

i.e. the original array has to be displayed. In reality the following output will be produced by gcc 7.4.0

```
$ docker run -v "$PWD":/tmp/test -w /tmp/test gcc:7 ./
src/sort
0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
$
```

As one can see the array is broken. How it can happen if every function in the code snapshot seems to be correct? The answer is below.

Inside STL sort

The STL library states the following requirements for **std::sort** ISO Programming Languages — C++(pp. 897-898): the compare function must follow so called strict weak ordering requirement. Especially it says that

```
comp(x, x) == false;
```

The requirement is violated by the $\leq$ operand used in our example.

Why the requirement is so important? The answer is in the algorithm that is used for sorting. STL uses an optimized version of Quicksort algorithm.

The code that produces the problem can be found via the following command

```
docker run gcc:7 tail -n +1893 \
/usr/local/include/c++/7.4.0/bits/stl_algo.h | head -n
21
```

and looks as follow

```
  /// This is a helper function...
  template<typename _RandomAccessIterator, typename
        _Compare>
    _RandomAccessIterator
    __unguarded_partition(_RandomAccessIterator
        __first,
                          _RandomAccessIterator
        __last,
                          _RandomAccessIterator
        __pivot, _Compare __comp)
    {
      while (true)
        {
          while (__comp(__first, __pivot))
            ++__first;
          --__last;
          while (__comp(__pivot, __last))
            --__last;
          if (!(__first < __last))
            return __first;
          std::iter_swap(__first, __last);
          ++__first;
        }
    }
```

As soon as our example has all elements equal to 1 one can assume that **__pivot** also keeps 1. In the function we start to compare all elements with the pivot. The used comparator will always produce **true** as soon as $1 \leq 1$. The size of the array is 17

```
(gdb) p data.size()
$1 = 17
```

and when we pass through whole array and reach its end we will find 0 on the last position (**__last**), i.e. 18th element of the array:

```
(gdb) p *(&data[0] + 17)
$2 = 0
```

The found element, that is located outside the array, will be included into the search as soon as it also satisfies the required property: $0 \leq 1$. The next element will be greater:

```
(gdb) p *(&data[0] + 18)
$3 = 61777
```

The element is 19th element of the array and is located 1 position after the original **__last** iterator. Thus the **__first** iterator will be 1 position behind the original **__last** one. The value will be returned into the following helper function

```
  /// This is a helper function...
  template<typename _RandomAccessIterator, typename
        _Compare>
    inline _RandomAccessIterator
    __unguarded_partition_pivot(_RandomAccessIterator
        __first,
                _RandomAccessIterator __last, _Compare
        __comp)
    {
      _RandomAccessIterator __mid = __first + (__last
        - __first) / 2;
      std::__move_median_to_first(__first, __first +
        1, __mid, __last - 1,
                  __comp);
      return std::__unguarded_partition(__first + 1,
        __last, __first, __comp);
    }
```

that will return the result as **__cut** in

```
  /// This is a helper function for the sort routine.
  template<typename _RandomAccessIterator, typename
        _Size, typename _Compare>
    void
    __introsort_loop(_RandomAccessIterator __first,
             _RandomAccessIterator __last,
             _Size __depth_limit, _Compare __comp)
    {
      while (__last - __first > int(_S_threshold))
    {
      if (__depth_limit == 0)
        {
          std::__partial_sort(__first, __last, __last,
        __comp);
          return;
        }
```

```
      --__depth_limit;
      _RandomAccessIterator __cut =
        std::__unguarded_partition_pivot(__first,
        __last, __comp);
      std::__introsort_loop(__cut, __last,
        __depth_limit, __comp);
      __last = __cut;
    }
    }
```

The **__last** iterator will be replaced with **__cut** that is **__first + 18** or **__last + 1**. The range stays stable until **__depth_limit** becomes 0.

Thus **std::__partial_sort** will get the range for sort that is greater (by 1) then original one. The element (0) outside array will be placed into beginning for the vector as **std::__partial_sort**'s application result.

Conclusion

As you can see, the comparator that violates strict weak ordering requirements will lead to array boundary condition violations. In our example we were "lucky" and just got incorrect data in the result. The SIGSEGV is more probable obvious result for such error. [1] The error can be easily fixed by using correct comparator, for instance the following one

```
auto comp = [](int i1, int i2) { return i1 < i2; };
```

The example shown in the article demonstrates the importance of requirements specified as C++ standard. Everyone who does not follow the requirements during an application development is in the risk of serious trouble with the application's undefined behaviour.

---

1. In real life the broken data is more probable but often are not recognized as an error because the result is hidden compare to SIGSEGV one that is visible immediately.↩