STL sort

Murashko Ivan

Introduction

The most tricky thing in C++ is an error catch. Most part of errors are located in 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. seems to be error free but you got a SIGSEGV inside a system library. Most probably you just felt into a situation that is described in the article.

The examples in the article uses gcc 5.5.0 from the docker image gcc:5:

```
$ docker run gcc:5 gcc --version
gcc (GCC) 5.5.0
```

1 Example

Lets look at the following code:

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

void print(std::vector<int>& data) {
    for (int item : data) {
      std::cout << item << " ";
    }
    std::cout << std::endl;
}


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);
  print(data);
  return 0;
}
```

What's output does it produces? One can expect the following one:

```
$ ./src/sort
1 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 is produced by gcc 5.5.0

```
$ ./src/sort
0 1 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 be if every function in the code snapshot seems to be correct? The answer is below.

2 STL requirements

The STL library states the following requirements for std::sort [1](pp. 897-898): the compare function must follow so called strict weak ordering requirement. Especially it says that

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

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. The algorithm is called Quick sort [2].

Using the following command

```
docker run gcc:5 tail -n +1888 \
/usr/local/include/c++/5.5.0/bits/stl_algo.h | head -n 21
```

one can get the following code

```
  /// 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;
        }
    }
```

The pivot in the function is chosen via 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);
  }
```

i.e. the median element is chosen and moved to the first position as the pivot. After that 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) (gdb) p data.size()
$7 = 17
```

and when we pass through whole array and reach the end of the array we will found 0 on the next position:

```
(gdb) p *(&data[0] + 17)
$5 = 0
```

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

```
(gdb) p *(&data[0] + 18)
$1 = 61777
```

and as result the result be 2 positions away from the array end. This will be assigned as new end of the array and as result the element 0 will be included into the sort. The real sort will be done via partial sort that is really is a heap sort from the following function

```
/// 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;
      }
  }
```

As result the element (0) outside array will be placed into beginning for the list.

Conclusion

TBD

References

[1] ISO. ISO/IEC JTC1 SC22 WG21 N 3690: Programming languages — C++ / ISO. — 2013. — Sep. — P. 1359. — http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3690.pdf.

[2] Wikipedia contributors. Quicksort — Wikipedia, the free encyclopedia. — https://en.wikipedia.org/w/index.php?title=Quicksort&oldid=917319740. — 2019. — [Online; accessed 23-September-2019].