



Evolution of a Median Algorithm

PETE ISENSEE



20
23



Output the median after
the samples are collected

Algorithms library

The algorithms library defines functions for a variety of purposes (e.g. searching, sorting, counting, manipulating) that operate on ranges of elements. Note that a range is defined as `[first, last)` where `last` refers to the element *past* the last element to inspect or modify.

Constrained algorithms

C++20 provides [constrained](#) versions of most algorithms in the namespace `std::ranges`. In these algorithms, a range can be specified as either an [iterator-sentinel](#) pair or as a single [range](#) argument, and projections and pointer-to-member callables are supported. Additionally, the [return types](#) of most algorithms have been changed to return all potentially useful information computed during the execution of the algorithm.

(since C++20)

```
std::vector<int> v {7, 1, 4, 0, -1};  
std::ranges::sort(v); // constrained algorithm
```

Execution policies

Most algorithms have overloads that accept execution policies. The standard library algorithms support several [execution policies](#), and the library provides corresponding execution policy types and objects. Users may select an execution policy statically by invoking a parallel algorithm with an [execution policy object](#) of

ChatGPT

Brainstorm names

for an orange cat we're adopting from th...

Plan a trip

to see the northern lights in Norway

Explain this code:

```
"cat config.yaml | awk NF"
```

Design a database schema

for an online merch store

How do I find the median of a container in C++?



Free Research Preview. ChatGPT may produce inaccurate information about people, places, or facts. [ChatGPT August 3 Version](#)

?

ChatGPT



To find the median of a container in C++, you can follow these steps:



1. Ensure that the container is sorted: The median is the middle value of a sorted dataset. If your container is not already sorted, you should sort it first. You can use the `std::sort` function from the `<algorithm>` header for this purpose.
2. Determine the median position: Depending on whether the number of elements in the container is odd or even, the median position can differ:
 - If the number of elements is odd, the median is the middle element.
 - If the number of elements is even, the median is the average of the two middle elements.
3. Access the median element(s): Once you've determined the median position, you can access the median element(s) using iterators or array indexing, depending on the type of container you are using.

Median

```
#include <algorithm>
#include <ranges>
#include <vector>

auto median(std::vector<int>& v) {
    std::ranges::sort(v);
    const auto medianIndex = std::size(v) / 2;
    return v[medianIndex];
}
```

Evolution of a Median Algorithm

Pete Isensee
Engineering Director



Goal

```
template <std::ranges::random_access_range Range>
constexpr auto median(Range&& rg) {
    // TODO
    // O(N)?
}
```


Stretch Goal

```
template <typename ExecPolicy,  
    std::ranges::random_access_range Range,  
    typename Comparison = std::ranges::less,  
    typename Projection = std::identity>  
constexpr auto median(  
    ExecPolicy&& execPolicy,  
    Range&& rg,  
    Comparison comp = {},  
    Projection proj = {}) {  
    // TODO  
}
```

Median

```
auto median(std::vector<int>& v) {  
    std::ranges::sort(v);  
    const auto medianIndex = std::size(v) / 2;  
    return v[medianIndex];  
}
```

Handle Empty Container

```
auto median(std::vector<int>& v) {  
    if (std::empty(v))  
        throw std::domain_error("requires non-empty range");  
  
    std::ranges::sort(v);  
    const auto medianIndex = std::size(v) / 2;  
    return v[medianIndex];  
}
```

What if Vector has 2 Elements?

```
auto median(std::vector<int>& v) {  
    if (std::empty(v))  
        throw std::domain_error("requires non-empty range");  
  
    std::ranges::sort(v);  
    const auto medianIndex = std::size(v) / 2;  
    return v[medianIndex];  
}
```

Wikipedia: Median

values

Finite data set of numbers [\[edit \]](#)

The median of a finite list of numbers is the "middle" number, when those numbers are listed in order from smallest to greatest.

If the data set has an odd number of observations, the middle one is selected. For example, the following list of seven numbers,

1, 3, 3, **6**, 7, 8, 9

has the median of 6, which is the fourth value.

If the data set has an **even number** of observations, there is no distinct middle value and the median is **usually defined to be the arithmetic mean of the two middle values.**^{[1][2]} For example, this data set of 8 numbers

1, 2, 3, **4**, **5**, 6, 8, 9

has a median value of 4.5, that is $(4 + 5)/2$. (In more technical terms, this interprets the median as the fully **trimmed mid-range**).

In general, with this convention, the median can be defined as follows: For a data set x of n elements, ordered from smallest to greatest,

Handle Even and Odd Lengths

```
auto median(std::vector<int>& v) {  
    if (std::empty(v))  
        throw std::domain_error("requires non-empty range");  
  
    std::ranges::sort(v);  
    const auto half = std::size(v) / 2;  
    if (std::size(v) % 2) // odd  
        return v[half];  
    return (v[half-1] + v[half]) / 2; // even  
}
```

Integer Division

```
auto median(std::vector<int>& v) {  
    if (std::empty(v))  
        throw std::domain_error("requires non-empty range");  
  
    std::ranges::sort(v);  
    const auto half = std::size(v) / 2;  
    if (std::size(v) % 2) // odd  
        return v[half];  
    return (v[half-1] + v[half]) / 2; // even  
}
```

Return Floating Point

```
auto median(std::vector<int>& v) {  
    if (std::empty(v))  
        throw std::domain_error("requires non-empty range");  
  
    std::ranges::sort(v);  
    const auto half = std::size(v) / 2;  
    if (std::size(v) % 2) // odd  
        return double(v[half]);  
    return (v[half-1] + v[half]) / 2.0; // even  
}
```


midpoint

```
auto median(std::vector<int>& v) {  
    if (std::empty(v))  
        throw std::domain_error("requires non-empty range");  
  
    std::ranges::sort(v);  
    const auto half = std::size(v) / 2;  
    if (std::size(v) % 2) // odd  
        return double(v[half]);  
    return std::midpoint<double>(v[half-1], v[half]); // even  
}
```

What About Performance

```
auto median(std::vector<int>& v) {  
    if (std::empty(v))  
        throw std::domain_error("requires non-empty range");  
  
    std::ranges::sort(v);  
    const auto half = std::size(v) / 2;  
    if (std::size(v) % 2) // odd  
        return double(v[half]);  
    return std::midpoint<double>(v[half-1], v[half]); // even  
}
```

What About Performance

```
auto median(std::vector<int>& v) {  
    if (std::empty(v))  
        throw std::domain_error("requires non-empty range");  
  
    std::ranges::sort(v); // n log n  
    const auto half = std::size(v) / 2;  
    if (std::size(v) % 2) // odd  
        return double(v[half]);  
    return std::midpoint<double>(v[half-1], v[half]); // even  
}
```

nth_element

```
auto median(std::vector<int>& v) {  
    if (std::empty(v))  
        throw std::domain_error("requires non-empty range");  
  
    const auto mid = std::begin(v) + (std::size(v) / 2);  
    std::ranges::nth_element(v, mid);  
    if (std::size(v) % 2) // odd  
        return double(*mid);  
    return std::midpoint<double>(*mid-1, *mid); // even  
}
```

Consider mid-1

```
auto median(std::vector<int>& v) {  
    if (std::empty(v))  
        throw std::domain_error("requires non-empty range");  
  
    const auto mid = std::begin(v) + (std::size(v) / 2);  
    std::ranges::nth_element(v, mid);  
    if (std::size(v) % 2) // odd  
        return double(*mid);  
    return std::midpoint<double>(*mid-1, *mid); // uh-oh  
}
```

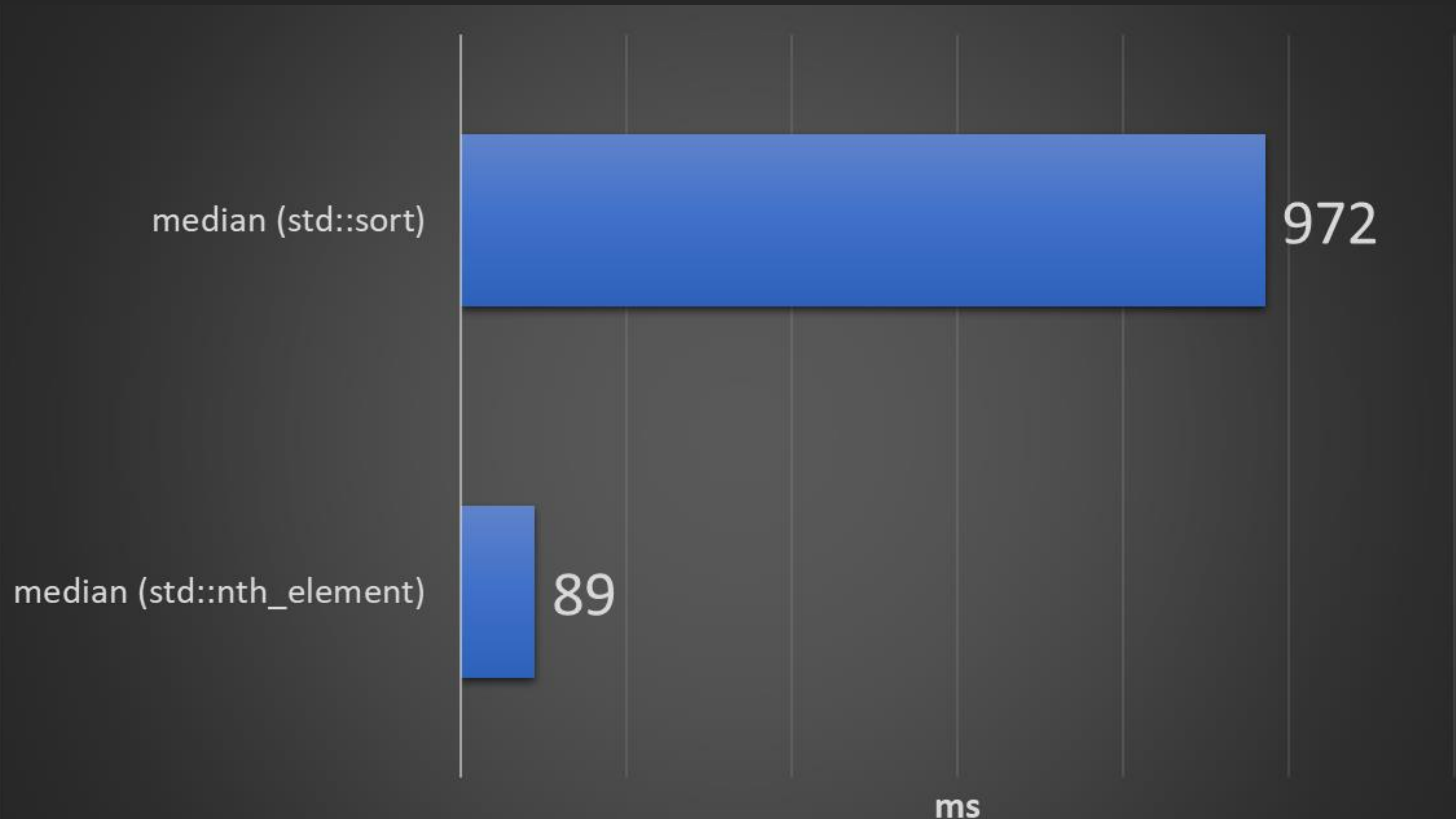
nth_element x2

```
auto median(std::vector<int>& v) {  
    using namespace std::ranges;  
    if (std::empty(v))  
        throw std::domain_error("requires non-empty range");  
  
    const auto mid = std::begin(v) + (std::size(v) / 2);  
    nth_element(v, mid);  
    const double midValue = *mid;  
    if (std::size(v) % 2) // odd  
        return midValue;  
    const auto midNeighbor = mid - 1; // even  
    nth_element(v, midNeighbor);  
    return std::midpoint<double>(*midNeighbor, midValue);  
}
```

max_element

```
auto median(std::vector<int>& v) {  
    using namespace std::ranges;  
    if (std::empty(v))  
        throw std::domain_error("requires non-empty range");  
  
    const auto mid = std::begin(v) + (std::size(v) / 2);  
    nth_element(v, mid);  
    if (std::size(v) % 2) // odd  
        return double(*mid);  
  
    const subrange lowerHalf{ std::begin(v), mid };  
    const auto midNeighbor = max_element(lowerHalf);  
    return std::midpoint<double>(*midNeighbor, *mid);  
}
```

Performance with 10M ints



Consider Flexibility

```
auto median(std::vector<int>& v) {  
    using namespace std::ranges;  
    if (std::empty(v))  
        throw std::domain_error("requires non-empty range");  
  
    const auto mid = std::begin(v) + (std::size(v) / 2);  
    nth_element(v, mid);  
    if (std::size(v) % 2) // odd  
        return double(*mid);  
  
    const subrange lowerHalf{ std::begin(v), mid }; // even  
    const auto midNeighbor = max_element(lowerHalf);  
    return std::midpoint<double>(*midNeighbor, *mid);  
}
```

Templatized Version

```
template <typename Container> auto median(Container& c) {  
    using namespace std::ranges;  
    if (std::empty(c))  
        throw std::domain_error("requires non-empty range");  
  
    const auto mid = std::begin(c) + (std::size(c) / 2);  
    nth_element(c, mid);  
    if (std::size(c) % 2) // odd  
        return double(*mid);  
  
    const subrange lowerHalf{ std::begin(c), mid }; // even  
    const auto midNeighbor = max_element(lowerHalf);  
    return std::midpoint<double>(*midNeighbor, *mid);  
}
```

Return Iterator

```
template <typename Container> auto median(Container& c) {  
    using namespace std::ranges;  
    if (std::empty(c))  
        throw std::domain_error("requires non-empty range");  
  
    const auto mid = std::begin(c) + (std::size(c) / 2);  
    nth_element(c, mid);  
    if (std::size(c) % 2) // odd  
        return mid;  
  
    const subrange lowerHalf{ std::begin(c), mid }; // even  
    const auto midNeighbor = max_element(lowerHalf);  
    return midNeighbor, mid; // hmm?  
}
```

Return Iterator Pairs

```
template <typename Container> auto median(Container& c) {  
    using namespace std::ranges;  
    if (std::empty(c))  
        throw std::domain_error("requires non-empty range");  
  
    const auto mid = std::begin(c) + (std::size(c) / 2);  
    nth_element(c, mid);  
    if (std::size(c) % 2) // odd  
        return std::pair{ mid, mid };  
  
    const subrange lowerHalf{ std::begin(c), mid }; // even  
    const auto midNeighbor = max_element(lowerHalf);  
    return std::pair{ midNeighbor, mid };  
}
```

Getting Closer

```
template <typename Container> auto median(Container& c) {  
    using namespace std::ranges;  
  
    const auto mid = std::begin(c) + (std::size(c) / 2);  
    nth_element(c, mid);  
    if (std::size(c) % 2) // odd  
        return std::pair{ mid, mid };  
  
    // even  
    const subrange lowerHalf{ std::begin(c), mid };  
    const auto midNeighbor = max_element(lowerHalf);  
    return std::pair{ midNeighbor, mid };  
}
```

Recall the Goal

```
template <std::ranges::random_access_range Range>
constexpr auto median(Range&& rg) {
    // TODO
    // O(N)
}
```

Concepts and Constexpr

```
template <std::ranges::random_access_range Range>
constexpr auto median(Range&& rg) {
    using namespace std::ranges;

    const auto mid = std::begin(rg) + (std::size(rg) / 2);
    nth_element(rg, mid);
    if (std::size(rg) % 2) // odd
        return std::pair{ mid, mid };

    // even
    const subrange lowerHalf{ std::begin(rg), mid };
    const auto midNeighbor = max_element(lowerHalf);
    return std::pair{ midNeighbor, mid };
}
```

Stretch Goal

```
template <typename ExecPolicy,  
    std::ranges::random_access_range Range,  
    typename Comparison = std::ranges::less,  
    typename Projection = std::identity>  
constexpr auto median(  
    ExecPolicy&& execPolicy,  
    Range&& rg,  
    Comparison comp = {},  
    Projection proj = {}) {  
    // TODO  
}
```


The Kitchen Sink

```
template <typename ExecPolicy,
          std::ranges::random_access_range Range,
          typename Comp = std::ranges::less>

auto median(ExecPolicy&& execPolicy, Range&& rg, Comp comp={}) {
    const auto begin = std::begin(rg);
    const auto end = std::end(rg);
    const auto mid = begin + (std::size(rg) / 2);

    std::nth_element(execPolicy, begin, mid, end, comp);
    if (std::size(rg) % 2)
        return std::pair{ mid, mid };

    const auto midNeighbor = std::max_element(execPolicy, begin, mid, comp);
    return std::pair{ midNeighbor, mid };
}
```

Plus Requirements

```
template <typename ExecPolicy,  
    std::ranges::random_access_range Range,  
    typename Comp = std::ranges::less>  
requires std::sortable<std::ranges::iterator_t<Range>, Comp>  
auto median(ExecPolicy&& execPolicy, Range&& rg, Comp comp={}) {  
    const auto begin = std::begin(rg);  
    const auto end = std::end(rg);  
    const auto mid = begin + (std::size(rg) / 2);  
  
    std::nth_element(execPolicy, begin, mid, end, comp);  
    if (std::size(rg) % 2)  
        return std::pair{ mid, mid };  
  
    const auto midNeighbor = std::max_element(execPolicy, begin, mid, comp);  
    return std::pair{ midNeighbor, mid };  
}
```

Better

```
template <typename ExecPolicy,
          std::ranges::random_access_range Range,
          typename Comp = std::ranges::less>
requires std::sortable<std::ranges::iterator_t<Range>, Comparison>
auto median(ExecPolicy&& execPolicy, Range&& rg, Comp comp={}) {
    const auto begin = std::begin(rg);
    const auto end = std::end(rg);
    const auto mid = begin + (std::size(rg) / 2);

    std::nth_element(execPolicy, begin, mid, end, comp);
    if (std::size(rg) % 2)
        return std::pair{ mid, mid };

    const auto midNeighbor = std::max_element(execPolicy, begin, mid, comp);
    return std::pair{ midNeighbor, mid };
}
```

Using Median is Complicated

```
// original
std::cout << median(rg);

// proposed
auto [midLow, midHigh] = median(rg);
if (midLow != rg.end()) {
    double medianValue = (*midLow + *midHigh) / 2.0; // midpoint, please!
    std::cout << medianValue;
}
else {
    std::cout << "empty";
}
```

median_result

```
template <typename Iterator, typename FP = double>
class median_result {
    Iterator low_, high_, end_;

public:
    Iterator mid_low() const { return low_; }
    Iterator mid_high() const { return high_; }

    std::optional<FP> value() const
    requires std::is_arithmetic_v<typename Iterator::value_type> {
        if (low_ == end_)
            return {};
        return std::midpoint<FP>(*low_, *high_);
    }
};
```

Best?

```
template <typename ExecPolicy,
          std::ranges::random_access_range Range,
          typename Comp = std::ranges::less>
requires std::sortable<std::ranges::iterator_t<Range>, Comp>
median_result<std::ranges::iterator_t<Range>>
median(ExecPolicy&& execPolicy, Range&& rg, Comp comp={}) {
    // get begin/mid/end iterators as before ...

    std::nth_element(execPolicy, begin, mid, end, comp);
    if (std::size(rg) % 2)
        return median_result{ mid, mid, end };

    const auto midNeighbor = std::max_element(execPolicy, begin, mid, comp);
    return median_result{ midNeighbor, mid, end };
}
```

Calling is Easier

```
// caller can use the std::optional<FP> for arithmetic types
auto middle = median(rg).value();
if (middle.has_value()) {
    std::cout << *middle;
}
else {
    std::cout << "empty";
}

// one line version if rg is not empty
std::cout << *median(rg).value();
```

Median Algorithm Takeaways

Pros

- Generic version possible
- Example w/ proven nth/max_element
- Much more efficient than sort
- Works for any sortable range
- Supports execution policies, comparison functions, concepts/requirements
- Hides confusing iterator pair
- Midpoint available for arithmetic types

Cons

- Surprising: permutes container
- Perf opportunities
- Doesn't handle quartiles, etc.
- Usage pattern good, not great
- What else?

Next Steps

- Version for any percentile range, e.g. { .25, .50, .75 }
- Support potential `std::multi_nth_element` [P2375](#)
- Investigate statistics proposals [P1708](#), [P2681](#)
 - Median/percentile for sorted ranges
 - Running medians
 - Align usage
- Create proposal for standardization

What Have We Learned

- AI is not always your friend
- Making algorithms generic is useful, challenging, rewarding
- Add midpoint and nth_element to your toolbox
- The caller is king
- median() should be standardized

Call to Action

- Send me your feedback, concerns, questions
- Help contribute, review, test
- pete.isensee@meta.com

PDF: <https://tinyurl.com/mediancppcon23>

Code: <https://godbolt.org/z/a4ooYbd1P>



Cppcon
The C++ Conference

2023



October 01-06

Aurora, Colorado, USA

Thanks

- Brian Budge
- Stephan T. Lavavej (STL)
- Alex Fix
- Forrest Smith
- Alex Lam
- The Test Yourself Tuesday crew

With multi_nth_element

```
// ... template specification as before ...
median_result<std::ranges::iterator_t<Range>> median(
    ExecPolicy&& execPolicy, Range&& rg, Comp comp = {}) {

    const auto sz = std::size(rg);
    const auto end = std::end(rg);
    const auto begin = std::begin(rg);
    const auto mid = begin + (sz / 2);
    if (sz % 2) { // odd
        std::nth_element(execPolicy, begin, mid, end, comp);
        return median_result{ mid, mid, end };
    }

    std::array nths = { mid-1, mid };
    std::multi_nth_element(execPolicy, begin, nths, end, comp);
    return median_result{ mid-1, mid, end };
}
```