The Dark Corner of STL

MinMax Algorithms

ŠIMON TÓTH

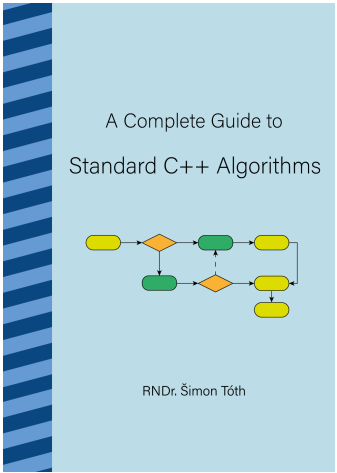
Permanent link

https://github.com/HappyCerberus/cppcon22-talk

Prior art

Walter E. Brown - Correctly calculating min, max and more

- how to correctly implement min, max algorithms
- nuances of less than comparison



- Free on GitHub:
  HappyCerberus/book-cpp-algorithms
- Donate to EFF on LeanPub:
  leanpub.com/cpp-algorithms-guide

How hard is it to call `std::min`?

```
auto min = std::min(1, 2);
```

```cpp
auto min = std::min(1, 2);

auto max = std::max(1, 2);
```

```cpp
auto min = std::min(1, 2);

auto max = std::max(1, 2);

auto clamped = std::clamp(0, 1, 2);
```

```cpp
auto min = std::min(1, 2);

auto max = std::max(1, 2);

auto clamped = std::clamp(0, 1, 2);

auto minmax = std::minmax(1, 2);
```

```
template< class T >
const T& min( const T& a, const T& b );

template< class T >
const T& max( const T& a, const T& b );

template< class T >
const T& clamp( const T& v, const T& lo, const T& hi );

template< class T >
std::pair<const T&,const T&> minmax( const T& a, const T& b );
```

```cpp
std::pair<const int&, const int&> a = std::minmax(1, 2);
// a.first, a.second are now dangling references
```

```cpp
std::pair<const int&, const int&> a = std::minmax(1, 2);
// a.first, a.second are now dangling references

const int& b = std::min(1, 2);
// b is now a dangling reference
```

```cpp
std::pair<const int&, const int&> a = std::minmax(1, 2);
// a.first, a.second are now dangling references

const int& b = std::min(1, 2);
// b is now a dangling reference

auto c = std::min(1, 2);
// decltype(c) == int
```

```cpp
std::pair<const int&, const int&> a = std::minmax(1, 2);
// a.first, a.second are now dangling references

const int& b = std::min(1, 2);
// b is now a dangling reference

auto c = std::min(1, 2);
// decltype(c) == int

auto d = std::minmax(1, 2);
// decltype(d) == std::pair<const int&, const int&>
```

```cpp
auto [x, y] = std::minmax(1, 2);
// decltype(x) == const int&, decltype(y) == const int&
```

```cpp
auto [x, y] = std::minmax(1, 2);
// decltype(x) == const int&, decltype(y) == const int&

std::pair<int, int> a = std::minmax(1, 2);
// OK, capture by-copy
```

```
g++ sample.cc -fsanitize=address -g && ./a.out
```

```
=================================================================
==119==ERROR: AddressSanitizer: stack-use-after-scope on address 0x7ffccaf6cd10
    at pc 0x55df7a7364b0 bp 0x7ffccaf6cce0 sp 0x7ffccaf6ccd0
READ of size 4 at 0x7ffccaf6cd10 thread T0
    #0 0x55df7a7364af in main /home/simon/main.cpp:7
    #1 0x7f61baf18d8f  (/lib/x86_64-linux-gnu/libc.so.6+0x29d8f)
    #2 0x7f61baf18e3f in __libc_start_main (/lib/x86_64-linux-gnu/libc.so.6+0x29e3f)
    #3 0x55df7a736264 in _start (/home/simon/a.out+0x1264)

Address 0x7ffccaf6cd10 is located in stack of thread T0 at offset 32 in frame
    #0 0x55df7a736338 in main /home/simon/main.cpp:5
```

Variants

Variants

- (C++20) range versions have identical behaviour

```
range_value_t<R> min(R&& r, Comp comp, Proj proj);
```

Variants

- (C++20) range versions have identical behaviour

```
range_value_t<R> min(R&& r, Comp comp, Proj proj);
```

- (C++14) `initializer_list` variants return by value

```cpp
auto x = std::min({1, 2});
// OK, decltype(x) == int

auto y = std::minmax({1, 2});
// OK, decltype(y) == std::pair<int,int>
```

```cpp
auto x = std::min({1, 2});
// OK, decltype(x) == int

auto y = std::minmax({1, 2});
// OK, decltype(y) == std::pair<int,int>

const int& z = std::min({1, 2});
// OK, lifetime extension
```

```
auto x = std::min({MoveOnly{}, MoveOnly{}});
// Wouldn't compile, can't move-out-of initializer_list.
```

```
auto x = std::min({MoveOnly{}, MoveOnly{}});
// Wouldn't compile, can't move-out-of initializer_list.

ExpensiveToCopy a, b;
auto y = std::min({ a, b });
// 3x copy
```

```
auto x = std::min({MoveOnly{}, MoveOnly{}});
// Wouldn't compile, can't move-out-of initializer_list.

ExpensiveToCopy a, b;
auto y = std::min({ a, b });
// 3x copy

auto z = std::min({ExpensiveToCopy{}, ExpensiveToCopy{}});
// 1x copy since C++17 (copy-initialization from prvalue)
```

const correctness

const correctness

```
MyType a, b;

if (b < a) {
    b.do_something();
} else {
    a.do_something();
}
```

const correctness

```
MyType a, b;

std::min(a,b).do_something();
```

const correctness

```
MyType a, b;

std::min(a,b).do_something();

const_cast<MyType&>(std::min(a,b)).do_something();
```

const correctness

```
const MyType a, b;

const_cast<MyType&>(std::min(a,b)).do_something();
// Undefined Behaviour (if do_something mutates state)
```

Can we fix it?

Target behaviour

- remove the need for `const_cast`
  when invoked with two mutable lvalue arguments
  $\Rightarrow$ *return (pair of) lvalue reference*

Target behaviour

- remove the need for `const_cast`
  when invoked with two mutable lvalue arguments
  $\Rightarrow$ *return (pair of) lvalue reference*
- remove the potential for dangling reference
  when either argument is prvalue
  $\Rightarrow$ *return by value*

Target behaviour

- remove the need for `const_cast`
  when invoked with two mutable lvalue arguments
  $\Rightarrow$ *return (pair of) lvalue reference*
- remove the potential for dangling reference
  when either argument is prvalue
  $\Rightarrow$ *return by value*
- avoid excessive copies
  only copy when returning by value
  and only the arguments that are returned

https://godbolt.org/z/ejvKx8YEh

```
// 1.
template <typename T> const T& min(const T& a, const T& b);
```

https://godbolt.org/z/ejvKx8YEh

```
// 1.
template <typename T> const T& min(const T& a, const T& b);

// 2.
template <typename T> T& min(T& a, T& b);
```

https://godbolt.org/z/ejvKx8YEh

```
// 1.
template <typename T> const T& min(const T& a, const T& b);

// 2.
template <typename T> T& min(T& a, T& b);

// 3.
template <typename T> T min(T&& a, const T& b);

// 4.
template <typename T> T min(const T& a, T&& b);
```

https://godbolt.org/z/ejvKx8YEh

```
// 1.
template <typename T> const T& min(const T& a, const T& b);

// 2.
template <typename T> T& min(T& a, T& b);

// 3.
template <typename T> T min(T&& a, const T& b);

// 4.
template <typename T> T min(const T& a, T&& b);

// 5.
template <typename T> T min(T&& a, T&& b);
```

```
int x = 1, y = 2;

int &a = min(x,y);
```

```
int x = 1, y = 2;

int &a = min(x,y);
// OK (lvalue, lvalue) → lvalue
```

```
int x = 1, y = 2;

int &a = min(x,y);
// OK (lvalue, lvalue) → lvalue

const int &b = min(10,20);
```

```
int x = 1, y = 2;

int &a = min(x,y);
// OK (lvalue, lvalue) → lvalue

const int &b = min(10,20);
// OK, lifetime extension
```

```
int x = 1, y = 2;

int &a = min(x,y);
// OK (lvalue, lvalue) → lvalue

const int &b = min(10,20);
// OK, lifetime extension

auto c = minmax(10,20);
```

```
int x = 1, y = 2;

int &a = min(x,y);
// OK (lvalue, lvalue) → lvalue

const int &b = min(10,20);
// OK, lifetime extension

auto c = minmax(10,20);
// OK, decltype(c) == std::pair<int,int>
```

Thanks to Luke D'Alessandro! https://godbolt.org/z/6qdGvczz3

Thanks to Luke D'Alessandro! https://godbolt.org/z/6qdGvczz3

```
// 1.
auto min(auto& x, auto& y) -> auto& {
    return y < x ? y : x;
}
```

Thanks to Luke D'Alessandro! https://godbolt.org/z/6qdGvczz3

```
// 1.
auto min(auto& x, auto& y) -> auto& {
    return y < x ? y : x;
}

// 2.
auto min(auto&& x, auto&& y) {
    return y < x ? y : x;
}
```

```
// 1.
auto min(auto& x, auto& y) -> auto& {
    if (y < x)
        return y;
    return x;
}
```

```cpp
// 1.
auto min(auto& x, auto& y) -> auto& {
    if (y < x)
        return y;
    return x;
}

auto min(auto& x, auto& y)
    -> std::common_reference_t<decltype(x),decltype(y)> {
    if (y < x)
        return y;
    return x;
}
```

```
// 1.
auto min(auto& x, auto& y) -> auto& {
    return y < x ? y : x;
}
```

```
// 1.
auto min(auto& x, auto& y) -> auto& {
    return y < x ? y : x;
}

auto min(auto& x, auto& y) -> auto&
requires std::is_same_v<std::remove_cvref_t<decltype(x)>,
                        std::remove_cvref_t<decltype(y)>> {
    return y < x ? y : x;
}
```

```
// 2.
auto min(auto&& x, auto&& y) {
    return y < x ? y : x;
}
```

```
// 2.
auto min(auto&& x, auto&& y) {
    return y < x ? y : x;
}

auto min(auto&& x, auto&& y) {
    return y < x ? std::forward<decltype(y)>(y) :
                   std::forward<decltype(x)>(x);
}
```

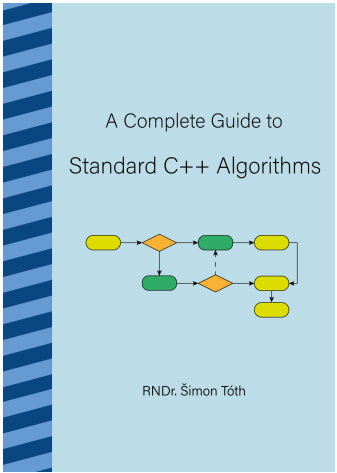
```cpp
// Min - rvalue variant
constexpr auto min(auto&& x, auto&& y)
    requires std::is_same_v<std::remove_cvref_t<decltype(x)>,
                            std::remove_cvref_t<decltype(y)>>
{
    return y < x ? std::forward<decltype(y)>(y) :
                   std::forward<decltype(x)>(x);
}

// Min - lvalue variant
constexpr auto min(auto& x, auto& y) -> auto&
    requires std::is_same_v<std::remove_cvref_t<decltype(x)>,
                            std::remove_cvref_t<decltype(y)>>
{
    return y < x ? y : x;
}
```

```cpp
// MinMax - lvalue variant
constexpr auto minmax(auto& x, auto& y)
    requires std::is_same_v<std::remove_cvref_t<decltype(x)>,
                            std::remove_cvref_t<decltype(y)>>
{
    using ref = std::common_reference_t<decltype(x),decltype(y)>;
    return y < x ? std::pair<ref,ref>{y,x} :
                   std::pair<ref,ref>{x,y};
}
```

How hard is it to call `std::min`?

How hard is it to call `std::min`?



- Free on GitHub:
  HappyCerberus/book-cpp-algorithms
- Donate to EFF on LeanPub:
  leanpub.com/cpp-algorithms-guide

References and links

- Demo of `std::minmax` with `-fsanitize=address`
- Temporary object lifetime
- `auto` type deduction
- `std::initializer_list`
- `const_cast`
- `std::common_reference`
- `requires` clause
- `std::is_same`
- All combinations for `std::common_reference_t`
- brute-force ”solution”
- C++20 ”solution”
- variadric version of min

Bonus slides

Performance-first alternative

```
auto min(auto& x, auto& y) -> auto& {
    return y < x ? y : x;
}

// Prohibit rvalues altogether.
auto min(auto&&, auto&&) = delete;
```

clamp

```
constexpr auto clamp(auto& v, auto& lo, auto& hi)
    → std::common_reference_t<decltype(v),decltype(lo),decltype(hi)>

requires
    std::is_same_v<std::remove_cvref_t<decltype(lo)>,
        std::remove_cvref_t<decltype(hi)>> &&
    std::is_same_v<std::remove_cvref_t<decltype(v)>,
        std::remove_cvref_t<decltype(hi)>>

{
    if (v < lo) return lo;
    if (v > hi) return hi;
    return v;
}
```