Aliasing

Risks, Opportunities and Techniques

ROI BARKAN

Link to Slides

“All problems in computer science can be solved by another level of indirection"
"... except for the problem of too many levels of indirection"

- David Wheeler

levelofindirection.com

Hi, I'm Roi

- Roi Barkan (he/him) - רועי ברקן
- I live in Tel Aviv, Israel
- C++ developer since 2000
- VP Technologies @ Istra Research
  - Finance, Low Latency, in Israel
  - careers@istraresearch.com
- Not an expert - happy to learn
  - Please - ask questions, make comments

What is Aliasing?

- Definition: two (or more) variables which refer to the same memory location being used together.
- Example:

```
std::string s{“hello, ”};
s += s;
```

- Causes dependencies to exist where the code seems independant
  - Aliasing is NOT about threads and volatile data
  - Reasoning about aliasing can be similar to reasoning about race conditions
- Aliasing considerations impact code correctness and efficiency/speed

Talk Outline

- Examples
  - Correctness, Performance
- Aliasing and the C++ Standard
- Dealing with aliasing pitfalls
  - APIs and implementations
  - Standard vs. compiler specific
- Future of aliasing
- Aliasing based design

Example: Aliased Function Arguments

- Pointers:

```
auto minmax = [](const string& i, const string& j,
                 string* out_min, string* out_max) {
    *out_min = min(i, j); *out_max = max(i, j);
};
array<string, 2> arr{"22222", "11111"};
minmax(arr[0], arr[1], &arr[0], &arr[1]); // try to sort
```

- References:

```
auto concat = [](string& result, const auto&... args) {
    ((result += args), ...);
};
string x{"hello "}, y{"world "};
concat(x, y, x);
```

```cpp
int main() {
    auto minmax = [](const string& i, const string& j, string* out_min,
                     string* out_max) {
        *out_min = min(i, j);
        *out_max = max(i, j);
    };
    array<string, 2> arr{"22222", "11111"};
    // try to sort
    minmax(arr[0], arr[1], &arr[0], &arr[1]);
    cout << "expect 22222 and get " << arr[1] << "\n";
    auto concat = [](string& result, const auto&... args) {
        ((result += args), ...);
    };
    string x{"hello "}, y{"world "};
    concat(x, y, x);
    cout << "expect [hello world hello ] and get [" << x << "]\n";
    return 0;
```

Executor x86-64 clang 14.0.0 (C++, Editor #1)

A | Wrap lines | Libraries (1) | Compilation | Arguments | Stdin | Compiler output

x86-64 clang 14.0.0 | -std=c++20 -O3

```
Program returned: 0
Program stdout
expect 22222 and get 11111
expect [hello world hello ] and get [hello world hello world ]
```

Example: Not Only Arguments

- Member variables:

```
complex<int> x{2, 2};
x *= reinterpret_cast<int*>(&x)[0];  // multiply by real part
```

- Lambda closures:

```
auto add_to_all = [](auto& v, const auto& val) {
    for_each(begin(v), end(v), [&](auto& x) { x += val; });
};
vector<int> v{1, 2, 3};
add_to_all(v, v[0]);
```

```
12  //    members();
13      complex<int> x{2, 2};
14      x *= reinterpret_cast<int*>(&x)[0];  // multiply by real part
15      cout << "expect (4,4) and get " << x << "\n";
16  //    lambdas();
17      auto add_to_all = [](auto& v, const auto& suffix) {
18          for_each(begin(v), end(v), [&](auto& x) { x += suffix; });
19      };
20      vector<int> v{1, 2, 3};
21      add_to_all(v, v[0]);
22      cout << "expected [2,3,4] and got [" << v[0] << "," << v[1] << "," << v[2]
23           << "]\n";
```

Executor x86-64 clang 14.0.0 (C++, Editor #1)

A ▾ | Wrap lines | Libraries (1) | Compilation | Arguments | Stdin | Compiler output

x86-64 clang 14.0.0 ▾ | -std=c++20 -O3

```
Program returned: 0
Program stdout
expect (4,4) and get (4,8)
expected [2,3,4] and got [2,4,5]
```

Example: Aliased Buffers

```
void loopcpy(char* dst, const char* src, int size) {
    while (size--) *dst++ = *src++;
}

using namespace std;
using namespace std::literals;
template <typename Fun>
void test(string_view name, Fun F) {
    char buffer[50] = "hello ";
    F(buffer + 1, buffer, 6);
    buffer[0] = ' ';
    cout << name << " [" << buffer << "] "
         << (" hello "sv == buffer ? "Good\n" : "Bad\n");
}
```


https://godbolt.org/z/Er4dfPPqb

Example: Aliased Buffers

```
void loopcpy(char* dst, const char* src, int size) {
    while (size--) *dst++ = *src++;
}
```

```
test("loopcpy", loopcpy);
test("strcpy ", [](auto dst, auto src, au
                    strcpy(dst, src); });
test("strncpy", strncpy);
test("memcpy ", memcpy);
test("memmove", memmove);
test("copy_n ",
     [](auto dst, auto src, auto size) {
                   copy_n(src, size, dst); });
```

Clang14	ICC 2021.5.0	Standard
loopcpy [hhhhhh] Bad	[hhhhhh] Bad	Bad
strcpy [helll] Bad	[helll] Bad	UB
strncpy [hello] Good	[hello] Good	UB
memcpy [hello] Good	[helll] Bad	UB
memmove [hello] Good	[hello] Good	Good
copy_n [hello] Good	[hello] Good	ID

https://godbolt.org/z/Er4dfPPqb

Example: STL Algorithms

- Erase (or Erase-Remove) max element with duplicates

```
erase(v, *max_element(begin(v), end(v)));
```

  or (C++20 ranges)

```
erase(v, *ranges::max_element(v));
```

  - (**remove** has documentation about this, **erase** doesn't)
- Copy/Move overlapping regions

```
copy(begin(v),end(v)-1, begin(v)+1);
```

  - (Documented as faulty, **copy_backward** recommended instead)
- Iterators can cause aliasing

```
auto max = ranges::max_element(a);
stable_partition(begin(a),end(a),[=](const auto&x) {return x != *max;});
```

  - (Predicates which modify their argument or the sequence are UB, this case isn't).

```cpp
void erase() {
    vector<int> v{1, 4, 2, 1, 4, 3, 4};
    erase(v, *max_element(begin(v), end(v)));
    cout << "erase_max         expected [1,2,1,3]     and got [";
    copy(begin(v), end(v) - 1, ostream_iterator<int>(cout, ","));
    cout << v.back() << "]\n";
    erase(v, *ranges::max_element(v));
    cout << "erase_ranges::max expected [1,2,1,3]     and got [";
    copy(begin(v), end(v) - 1, ostream_iterator<int>(cout, ","));
    cout << v.back() << "]\n";
}
```

Executor x86-64 clang 14.0.0 (C++, Editor #1)

A | Wrap lines | Libraries (1) | Compilation | Arguments | Stdin | Compiler outp

x86-64 clang 14.0.0 | -std=c++20 -O3

```
Program returned: 0
Program stdout
erase_max         expected [1,2,1,3]     and got [1,2,1,4,3,4]
erase_ranges::max expected [1,2,1,3]     and got [1,2,1,3,4]
```

```cpp
void copy() {
    vector<string> v{"b", "c", "d", "e"};
    copy(begin(v), end(v) - 1, begin(v) + 1);
    v[0] = "a";
    cout << "copy              expected [a,b,c,d]     and got [";
    copy(begin(v), end(v) - 1, ostream_iterator<string>(cout, ","));
    cout << v.back() << "]\n";
}

void partition() {
    array a = {1, 4, 2, 4, 3, 4};
    auto max = ranges::max_element(a);
    stable_partition(begin(a), end(a),
                     [=](const auto& x) { return x != *max; });
    cout << "stable_partition  expected [1,2,3,4,4,4] and got [";
    copy(begin(a), end(a) - 1, ostream_iterator<int>(cout, ","));
    cout << a.back() << "]\n";
```

Executor x86-64 clang 14.0.0 (C++, Editor #1)

A | Wrap lines | Libraries (1) | Compilation | Arguments | Stdin | Compiler output

x86-64 clang 14.0.0 | -std=c++20 -O3

```
Program returned: 0
Program stdout
copy              expected [a,b,c,d]     and got [a,b,b,b]
stable_partition  expected [1,2,3,4,4,4] and got [1,2,4,3,4,4]
```

Performance Effect of Aliasing

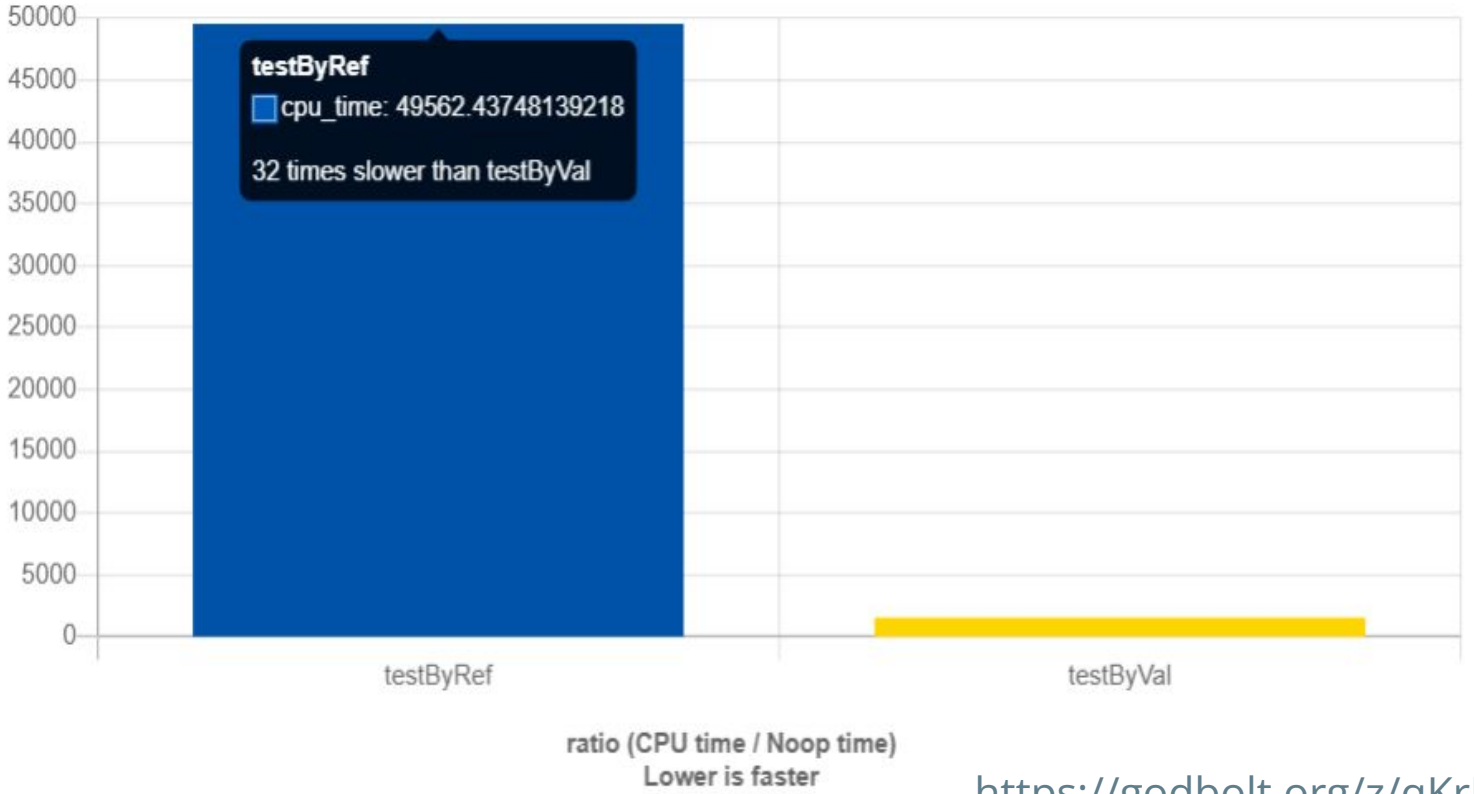
- Extreme example

```
void foo(std::vector<double>& v, const double& coeff) {
  for (auto& item : v) item *= std::sin(coeff);
}
```

- Compiler's missed opportunities:
  - Register <-> memory
  - Vectorization
  - Expression hoisting
- How important can it be...

https://godbolt.org/z/qKrEfWe7Y

Performance Benchmark Results



https://godbolt.org/z/qKrEfWe7Y

Lesson Learned - Aliasing is Tricky

- Humans rarely consider it → Strange unexpected bugs
  - We expect independence of different variables
- Compilers can't ignore it → Unexpected performance loss
  - Learn more in Ofek Shilon's talk about optimization remarks from Wednesday
- Library writers should document it → users should read documentation
  - Misuse often leads to 'happens to work' code
- *"All problems in computer science can be solved by another level of indirection"*
  "... except for the problem of too many levels of indirection"

- Questions and comments are welcome…

Aliasing in Other Languages

- The C language had a surge of (non-assembly) aliasing issues
  - Pointers were used as primitive substitutes to arrays, matrices, strings
  - C99 introduced the `restrict` keyword
    - A code block with a `restrict` pointer/array can only change the pointed data through that pointer/array. Otherwise: undefined behavior
    - Most C++ compilers have some non-standard support for `restrict`
- Fortran typically treats aliases as undefined behavior
  - with compiler switches to assume aliasing
- Swift and Rust track reference creation aiming to prohibit the risk of aliasing

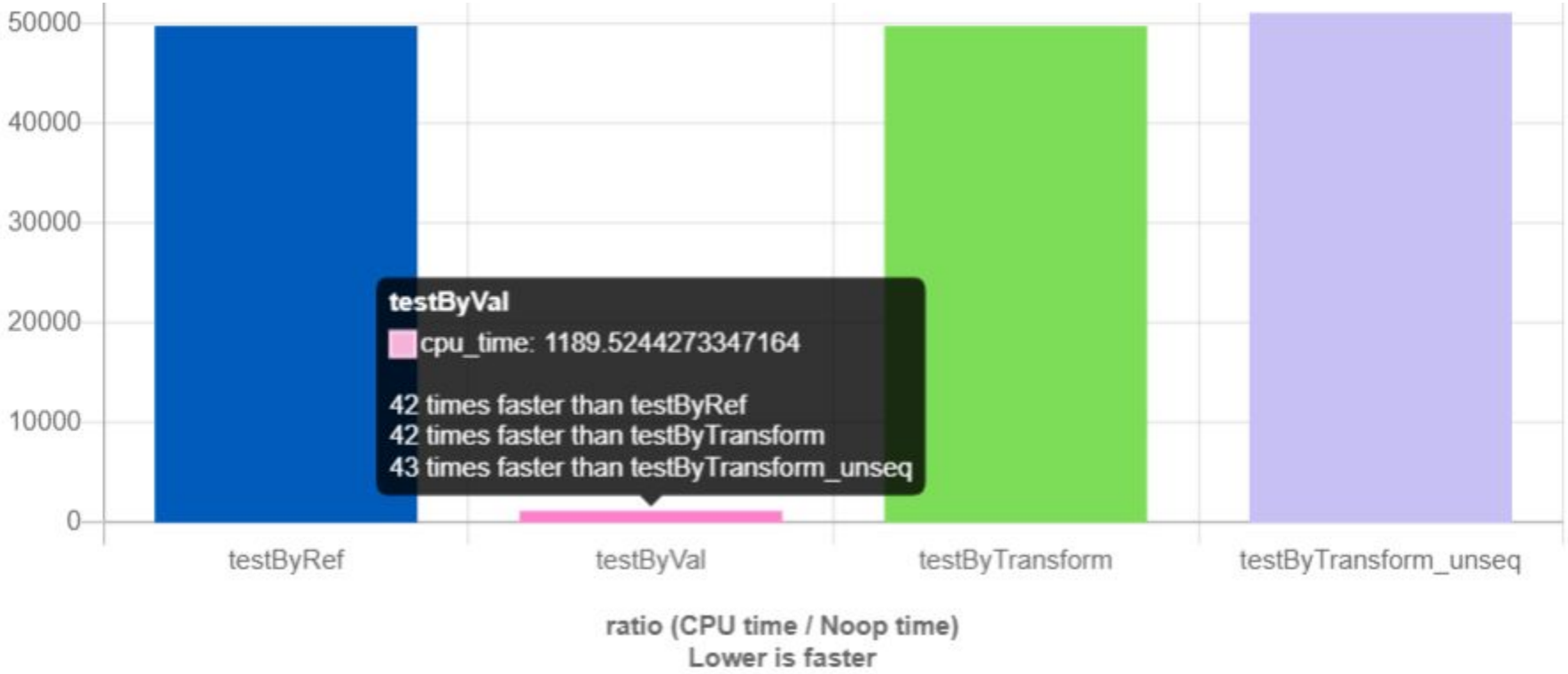
Aliasing in the C++ Standard

- C++ hasn't adopted the `restrict` keyword (yet?)
  - Seems more tricky: function-signature qualifiers, templates, functors/lambdas
- Aliasing *should* be type-based - known as "strict aliasing"
  - Only similar types are technically allowed to alias each other (and `char`, `std::byte`)
    - Similar types - changes to const/volatile/signed, or base-derived relationship
    - Otherwise - undefined behavior
  - Strong-typedefs can reduce risk and improve performance !
  - Most compiler-optimizers relax the rules - favoring predictability over performance
    - Still - compilers try to prove whether aliasing is impossible.
- Some objects are easier to reason about
  - Local variables - locally live on the stack
  - Temporary values

Aliasing in the C++ Standard Library (STL)

- The STL tries to document the effect of aliasing and sometimes mitigates them
  - `vec.push_back(v.front());` always works (with a performance cost)
  - `std::bind()` holds its 'closure' by-value and avoids aliasing
- **`std::valarray`** is specifically required to have no aliasing
  - The expression `addressof(a[i]) != addressof(b[j])` evaluates to `true` for any two arrays `a` and `b` and for any `size_t i` and `size_t j` such that `i < a.size()` and `j < b.size()`. [valarray.access]
- **`std::execution::par/unseq`** inherently (implicitly) treats many forms of aliasing as *undefined behavior*.

Performance - `std::execution::unseq`



https://godbolt.org/z/9qsafxfvb

Strong Typedefs

- Types that encapsulate and behave like other types, but are different and don't automatically convert to/from them
  - No standard implementation, but a few libraries mimic the behavior
- Motivating example:

```
struct A { int i; };
struct B { int i; };

int mayAlias(auto& a, const auto& b) {
    a.i += b.i;
    if (b.i == 2) return 0;
    return 1;
}

template int mayAlias(A&, const A&);
template int mayAlias(A&, const B&);
```

```
int mayAlias<A, A>(A&, A const&):
        mov     eax, DWORD PTR [rsi]
        add     DWORD PTR [rdi], eax
        xor     eax, eax
        cmp     DWORD PTR [rsi], 2
        setne   al
        ret
int mayAlias<A, B>(A&, B const&):
        mov     eax, DWORD PTR [rsi]
        add     DWORD PTR [rdi], eax
        cmp     eax, 2
        setne   al
        movzx   eax, al
        ret
```

https://godbolt.org/z/4dKfah7c5

How to Avoid Aliasing Pitfalls

- Pass arguments by value
  - Value semantics are all the rage
  - Move semantics and copy-elision can make this relatively cheap
  - Consider supporting **`std::reference_wrapper`** (i.e. **`std::ref()`**)
- Use strong typedefs and unit libraries
  - clearer code for humans, compilers might optimize it as well
- Document your code's aliasing assumptions (contract)
  - Read other people's documentation
- For a large user base - write defensive code
  - Verify your contract - assert/throw/etc.
  - Widen your contract (e.g. **`vec.push_back(v.front())`**)
  - Let users control the contract

Defensive Code

- Basic function

```
template <typename Value, typename BinOp>
void unsafe_apply(std::span<Value> s, const Value& v, BinOp op) {
    for (auto& item : s) item = op(item, v);
}
```

- User controlled version

```
template <typename T> struct ByRef { using type = const T&; };
template <typename T> struct ByVal { using type = T; };

template <typename Value, typename BinOp, typename PassBy = ByRef<Value>>
void user_apply(std::span<Value> s, const Value& v_ref, BinOp op, PassBy = {}) {
    typename PassBy::type v{v_ref};
    for (auto& item : s) item = op(item, v);
}
```


https://godbolt.org/z/T15odT8Ks

Defensive Code

- Safe version

```
template <typename Value, typename BinOp>
void safe_apply(std::span<Value> s, const Value& v, BinOp op) {
    if (!s.empty() && std::less_equal{}(&s.front(), &v) &&
        std::less_equal{}(&v, &s.back()))
    {
        user_apply(s, v, op, ByVal<Value>{});
        return;
    }
    user_apply(s, v, op, ByRef<Value>{});
}
```

- Sometimes bounds/alias checking isn't as easy
- Questions and comments are welcome…

Proposals on Aliasing in the C++ Standard

- The `restrict` keyword signal to users and compiler that aliasing is UB
  - Many compilers have some support for it, but standardization isn't likely
- [[alias_set]] (2014) - annotate the relationship between variables
  - has some similarities with Rust lifetime annotations
- span<T, std::restrict_access> (2018) - property-based 'qualifier' for added semantics
- std::disjoint (2018) - meant for *contracts* to convey aliasing consistently
- Lifetime safety (2019) - Core guidelines and static analysis which "default to banning passing non-owning Pointers that alias".

Tricking the Compiler ?

- **`union`** is a mechanism for several object types to reside in the same address.
- At any time one type is *active* and accessing a different type is *typically* UB
  - **`variant`** is a type safe STL class that enforces correct access
- C++ does allow some accesses to non-active types - and aliasing
  - Types need to be *StandardLayoutType* and accessed members need to be in their common prefix. **`std::is_corresponding_member`** checks for this condition.
- This implies that “strict aliasing” has limits
  - I might be wrong, or this might be a bug in the standard/compilers

Aliasing of Standard Layout Types

- Accessing aliased union members is sometimes allowed:

  "In a standard-layout union with an active member of struct type `T1`, it is permitted to read a non-static data member `m` of another union member of struct type `T2` provided `m` is part of the common initial sequence of `T1` and `T2`; the behavior is as if the corresponding member of `T1` were nominated." [class.mem.general]

- Example from the standard:

```
struct T1 { int a, b; };
struct T2 { int c; double d; };
union U { T1 t1; T2 t2; };
int f() {
  U u = { { 1, 2 } };    // active member is t1
  return u.t2.c;         // OK, as if u.t1.a were nominated
}
```

Motivating Example

- Implement a C++ Conference:

```
struct CppPerson {
    std::string name; double expertise;
    //... more fields, methods
};
//Precondition - teacher and students
can't alias
void teach(span<CppPerson*> students,
           const CppPerson& teacher) {
    for (auto pStudent : students)
      pStudent->expertise +=
       std::max(teacher.expertise,100.0);
};
```

- Can we use unions to express non-aliasing to the compiler ?

- Suggested approach:

```
struct Student : CppPerson {};
struct Teacher : CppPerson {};
static_assert(std::is_layout_compatible_v<
Student, Teacher>);
union Attendee {
    Student student;
    Teacher teacher;
};
void teach(span<Student*> students,
           const Teacher& teacher) {
    for (auto pStudent : students)
      pStudent->expertise +=
       std::max(teacher.expertise,100.0);
};
```

- Is this UB ??

https://godbolt.org/z/5coxPxxdf

Strict Aliasing and `union`

- Recall strong typedefs:

```
struct A { int i; };
struct B { int i; };

int mayAlias(auto& a, const auto& b)
{
    a.i += b.i;
    if (b.i == 2) return 0;
    return 1;
}
```

- Let's add unions:

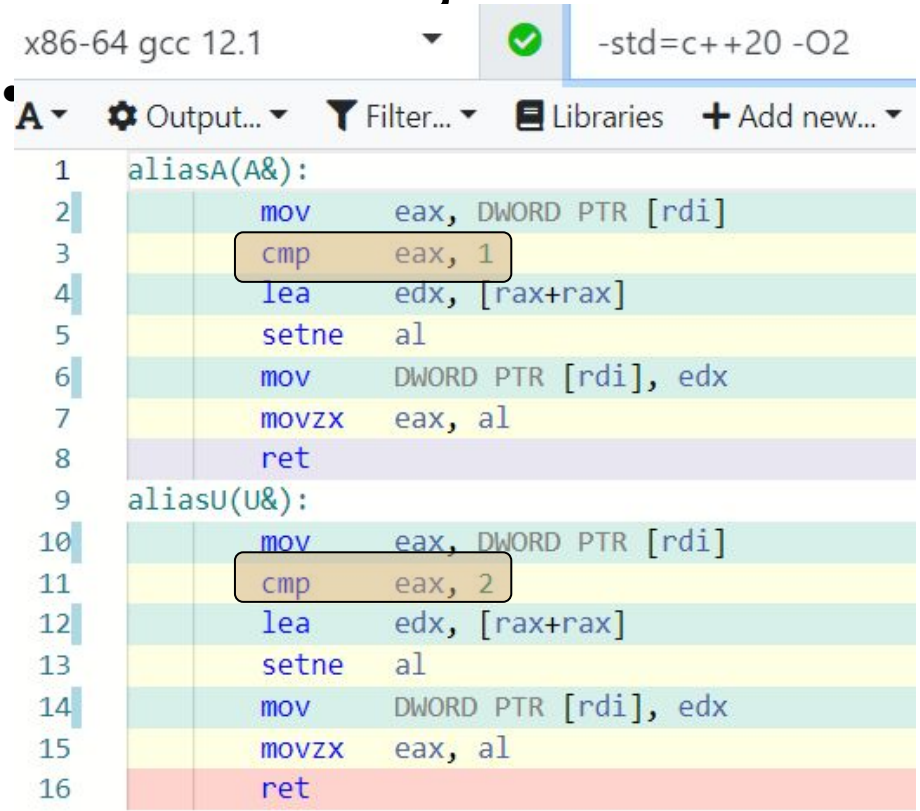
```
union U {
    A a;
    B b;
};

int aliasA(A& a) {
    return mayAlias(a, a);
};

int aliasU(U& u) {
    return mayAlias(u.a, u.b);
};
```


https://godbolt.org/z/d71Mjaxz7

Strict Aliasing and `union`



- Let's add unions:

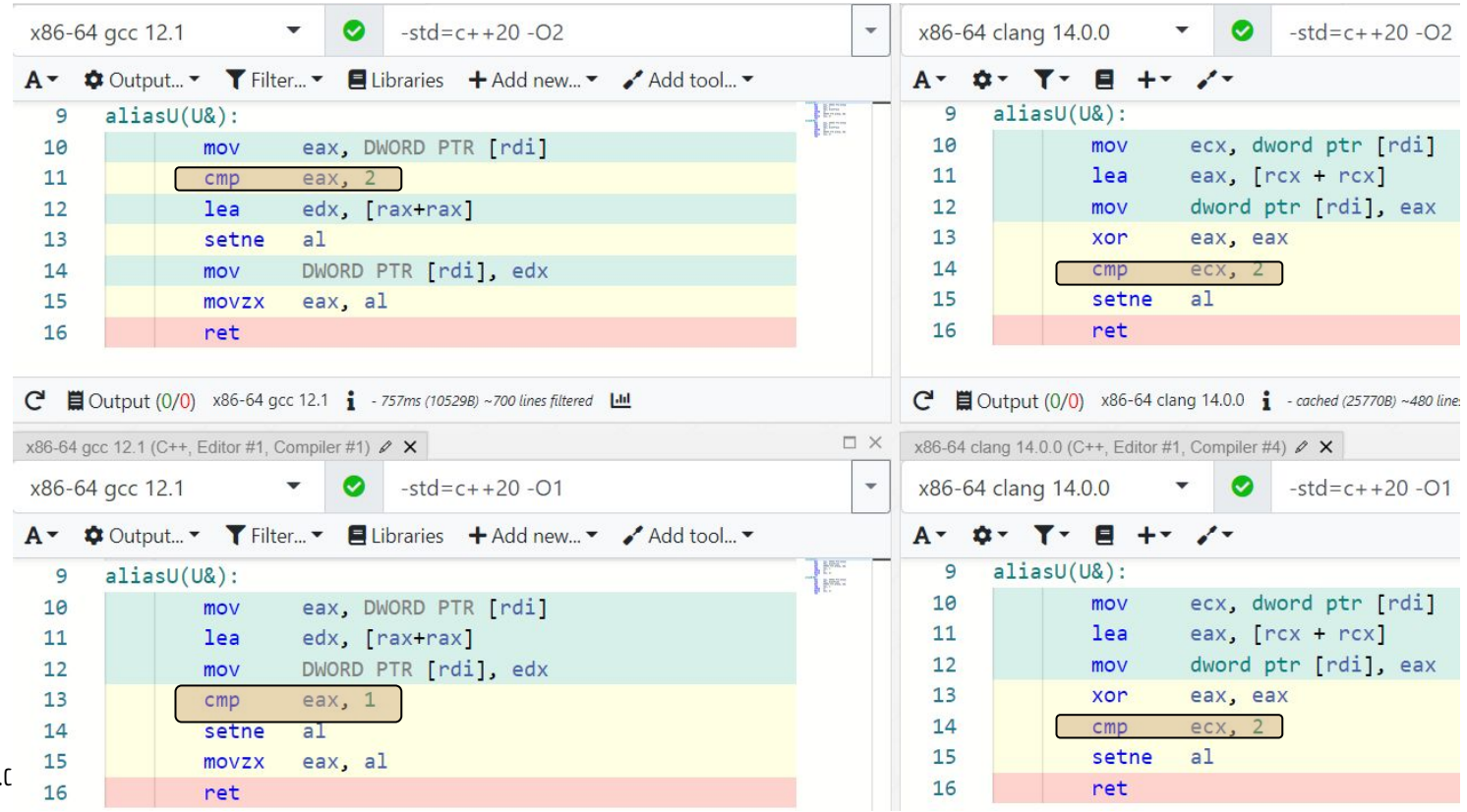
```
union U {
    A a;
    B b;
};

int aliasA(A& a) {
    return mayAlias(a, a);
};

int aliasU(U& u) {
    return mayAlias(u.a, u.b);
};
```

https://godbolt.org/z/d71Mjaxz7

Different Optimizers, Different Worlds



`variant` State Machines

- State machine is a typical case for using `variant`
  - At any point only one state is valid
- Changing the state to `T` is done via `operator=(T&&)` or `emplace<T>()`
- Different states commonly share information
  - `variant<WorkingPerson, RestingPerson>` - both states typically have a name, might inherit from `Person`.
  - Semantic strong typedefs might be identical in structure, e.g. `variant<Cat, HappyCat>`
- Sadly, state changing functions aren't allowed (UB) to read the previous state (especially relevant for `emplace<T>()`)
  - Previous state gets destructed before the new state constructor is invoked
  - STL chose performance over safety (unlike most containers).

`variant` State Changes

- Undefined/unexpected behavior:

```
variant<filesystem::path, string> v{"some_long_file_name"s};
v = std::move(v); //Bad on non-variants as well
v.emplace<filesystem::path>(std::move(get<string>(v)));
v.emplace<filesystem::path>(get<string>(v));
```

- The proper (no copy) way is to use temporaries, and rely on move semantics:

```
v.emplace<filesystem::path>(
                        string{std::move(get<string>(v))})
```

Summary

- Aliasing is tricky - people assume independence
- Value semantics makes life simpler
- Strong typedefs can assist
- Implement and document your code with care
- Smart people in the committee are working on improvements
- Know how to communicate with others and the compiler

Thank You !!

- Happy coding !
- Questions/comments are welcome

roi.barkan@gmail.com

References / Acknowledgements

- OptView2 - https://youtu.be/nVc439dnMTk
- [[alias_set]] - https://wg21.link/n3988
- span<T, std::restrict_access> - https://wg21.link/p0856
- std::disjoint - https://wg21.link/p1296
- Lifetime safety - https://wg21.link/p1179

Thank You !!