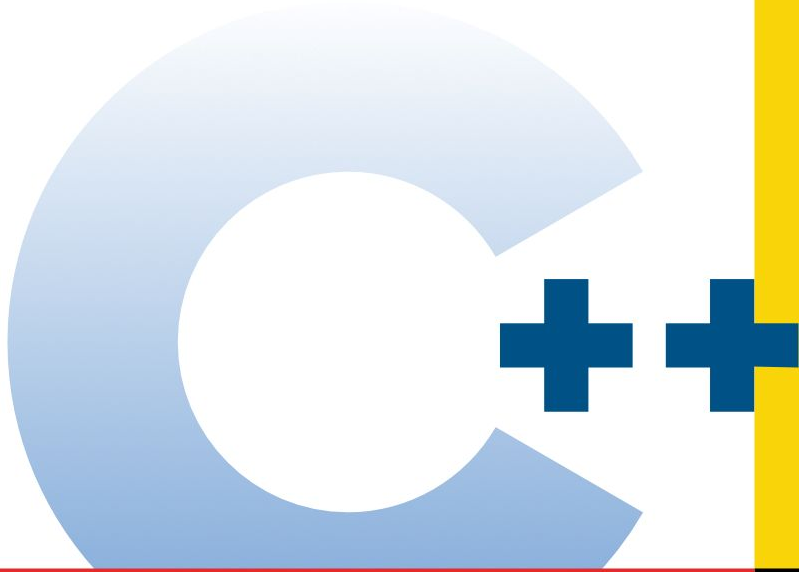
BeC++ SYMPOSIUM

Andre Kostur

There's a Hole in the C++ Type System



BECPP SYMPOSIUM 2026 | March 30th | Kortrijk

About Me

Agenda

- Spotting a problem
- What does the Standard say
- First proposed solution
- P3039
- Engaging the committee
- Future work

It Started With a Code Review

The Problematic Statement

```
auto bookmark = topObj()->attr()->addrIterator();
```

Our Framework

Return Types

- There are two classes of objects
  - Pointer
  - Value
- Accessing an “attribute” returns by value
  - A copy of the (refcounted) pointer

```
Ptr topObj() { return topObj_; }
```

  - A copy of the value

```
Val attr() { return attr_; }
```

- Notably there is no return by reference

Collections/Containers

- Collections/Containers are not first-class types

```
class AttrType {
    public:
        T addrIterator();
        T subnetIterator();
    private:
        AddrColl addr_;
        SubnetColl subnet_;
};
```

A Problem We Used to Have

- We accidentally would get an Iterator from a temporary object
  - Recall: attributes are returned by value

```
auto bookmark = attr().addrIterator();
bookmark.someFn();  // bookmark is dangling!
```

- We solved that by reference-qualifying the Iterator() member functions

```
T addrIterator() & { /* … */ }
T addrIterator() && = delete;
```

- Now attempting to get the iterator is a compile-time error

An Optional Thing

We Added Optional

- We added Optional to our type system
- Like the existing value-types, the Optional is also returned by value

```
std::optional<Val> attr() { return attr_; }
```

Back to the Code Review

Looking at the Review Again

```
auto bookmark = topObj()->attr()->addrIterator();
```

- `topObj()` returns a temporary smart pointer
- `->attr()` returns a temporary optional from the pointed-to object
- `->addrIterator()` calls that member function on the contained value
- But: `T addrIterator() && = delete;`
- This shouldn’t compile, but it does!

Diagnosing the Success

- There's two other ways to express that use of `->` on a `std::optional`
- What if we expanded the use of `->` on the optional?
- `topObj()->attr().value().addrIterator();`
  - `.value()` on `std::optional` is reference-qualified
  - Correctly fails to compile
- `(*(topObj()->attr()).addrIterator();`
  - `.operator*()` on `std::optional` is reference-qualified
  - Correctly fails to compile
- Why is `->` special?

-› In The Standard

Postfix expressions : Class member access

Paragraph 2 of [expr.ref] ends with:

> The expression E1->E2 is converted to the equivalent form (*(E1)).E2; the remainder of [expr.ref] will address only the form using a dot.

But, this only applies if E1 is a pointer.

We need to look further in the Standard as to what happens if E1 isn't a pointer.

operator->() In The Standard

Overloaded operators : Class member access

[over.ref] says:

```
A class member access operator function is a function
named operator-> that is a non-static member function
taking no non-object parameters.  For an expression of
The form
    postfix-expression -> template id-expression
the operator function is selected by overload resolution,
And the expression is interpreted as
    (postfix-expression.operator->())->template id-expression
```

[over.ref] interpreted

This means that if you have `E1->E2`, then call `E1.operator->()`, and then keep calling `operator->()` on whatever is returned until the type of that which is returned is a pointer. Then apply [expr.ref]: dereference it and do `.E2` on that.

Lost In Translation

Losing rvalueness

- `topObj()->attr()` gets us a temporary `std::optional`
- What does calling `operator->()` return?

```
T * operator->();
```

- We have a temporary pointer, [expr.ref] applies, not [over.ref]
- So `E1->E2` gets rewritten to `(*(E1)).E2`
  - Where `E2` is `attrIterator()`
- `*(E1)` gets us an lvalue, and thus uses the lvalue-qualified version of `E2`
- We lost the rvalueness!

Mitigation

Mitigate first

- Making a change to the language takes time
- C++26 is around the corner, thus anything proposed now won't get in until C++29
  - Formally. Compilers might implement it sooner.
- If we can't get the compiler to refuse the code, can we at least detect it?
- Yes, by writing a clang-tidy plugin
  - Detect if we are applying operator-> to an rvalue and are invoking an rvalue-qualified deleted member function
  - Still a problem here: if we have a normal lvalue-qualified and rvalue-qualified member function, this won't be detected and will call the wrong one when compiled

First Proposed Solution

Avoiding the Problem

- We need to consider a number of things when coming up with a solution
  - How does the solution change existing code?
  - Is there an existing feature that one can pattern a proposed solution after?
    - Can you make the new solution consistent with existing language features?
  - What does migrating existing code to the new solution look like?
- We're having this problem because we are getting a raw pointer during the evaluation of the expression
- Using `operator*()` doesn't have the problem
  - `operator*()` returns a reference
  - If it is important to the type, it can reference-qualify the overloads and return the appropriate type of reference

Following An Example

- Can I push `operator->()` to use `operator*()` automatically?
- We have examples of other operators where we can say “do the natural thing for this operator”:
  - `bool operator==() = default;`
  - `auto operator<=>() = default;`
- How about the same syntax here:

```
auto operator->() = default;
```

- It would mean “use [expr.ref] here instead of [over.ref]”
  - Use `operator*()` instead, and use the dot operator on what that returns

Consult Other Experts

- I think I've got a good idea, but other folk may see issues that I hadn't considered
- There's a mailing list to discuss potential proposals: std-proposals@lists.isocpp.org
- I'd actually started the thread before I had a solution, I first wanted to confirm that what I'm seeing is actually a problem
- Discussion ensued, and that night I considered my =default solution
- Someone else pointed out that there exists a paper that would seem to be dealing with a similar solution

P3039

Near Solution

- P3039 - Automatically Generate operator->
- Published 2023-11-07
- Was discussed in an ISO meeting and there was consensus that the paper should continue
- This paper’s focus was about simplifying the language, less about solving a specific problem
- For whatever reason, the author hadn’t published a revision of the paper
- I contacted the author for permission for me to continue their paper

P3039 vs =default

- P3039 took a different approach
- If an `operator->()` is found, use that
  - Even if `=delete`, in which case fail to compile
- If not, do the `(*(E1)).E2` rewrite
- If that fails, then fail to compile
- Consider:
  - If the type already has an `operator->()`, then nothing changes.
  - If the type has an `operator*()`, then the class behaves like a pointer
    - And is easier to write a class to be “correct”, by saying nothing!
  - If the type doesn’t want `operator->()` to work, it can `=delete` it

Next Steps

- This shortened my work by a lot to have an existing paper that has already been seen by the committee
- More work was, and is, called for:
  - A new revision with proposed wording
  - An example implementation for people to explore with
- Note that the example implementation is not strictly necessary
  - It should not constrain compiler implementers to doing it that way, they can implement as they see fit
  - But it is nice that other committee members can play with the implementation and experiment with issues that they think might break the proposal

P3039R1

- I did publish a P3039R1 to both LEWG and EWG
- Expanded and changed the focus more on the rvalue problem than making the language simpler
- Got valuable feedback from a number of folk in both mailing lists
  - Further wording changes
  - One important impediment: `std::to_address`

std::to_address(p)

- Obtains the address represented by p without forming a reference to the object pointed-to by p
- Written in terms of either a traits overload, or calling `operator->()`
- Where this interferes with this proposal is that it calls for the `operator->()` to no longer be explicitly declared
- The “simple” answer would be to use `std::addressof(p.operator*())`
- This is a step backwards to why `std::to_address` isn’t already defined in that manner: it may not be valid to “dereference” p.

Outstanding Flaws

- If one were to omit declaring an `operator->()`, anything trying to directly invoke it would fail
- This includes `std::to_address`
- One can work around that issue by supplying a specialization of `std::pointer_traits` for the type
- That specialization can provide a static member function `std::pointer_traits::to_address` which can do whatever it needs to in order to avoid forming a reference to the “pointed-to” object

Croyden ISO Meeting

- WG21 met in person, and virtually, last week
- P3039R1+ was presented on Thursday to Evolution Working Group (EWG)
- There were a number of comments and questions raised
- The status of the paper can be seen at https://github.com/cplusplus/papers/issues/1693

Poll Results

Encourage more in the direction of D3039R2 (Automatically Generate `operator->`)

SF	F	N	A	SA
7	19	2	3	0

Result: consensus

Poll Results

D3039R2: Requiring specialization of `std::pointer_traits` is the right answer to `std::to_address`

SF	F	N	A	SA
1	2	5	3	4

Result: not consensus

Poll Results

D3039R2: Deferring the modification of other Standard classes is appropriate (specifically `std::optional`, `std::expected`, and others)

SF	F	N	A	SA
2	10	8	2	0

Result: weak consensus

More Work Needed

- More versions of the paper will be necessary
- For this to get into the Standard there's a few more levels to go through
  - EWG has to vote to move it to Core
  - Core will probably spend time with the actual wording
  - Core has to vote to move it to Plenary, which actually gets it into the draft Standard
  - Somewhere in 2029, the National Bodies vote on the draft to make it the next Standard
- Publishing a paper is kinda like peer review
- You will get critical feedback
- The other people will be aware of other corners of the Standard that your adjustments may impact

Q & A

Resources:

http://wg21.link/p3039

https://godbolt.org/z/Y6q4Gn5W6

std-proposals@lists.isocpp.org

Andre Kostur
andre@kostur.net
https://www.linkedin.com/in/akostur