Modernize code base with range-v3 library #3394

VRichardJP started this conversation in **Design**

VRichardJP on Apr 4, 2023 Collaborator edited

C++20 and C++23 adds a lot of life changing features. Among others: views and ranges.

As every experienced C++ developer knows, the C way of iterating over a container is verbose, ugly and error-prone:

```
for (int i = 0; i < v.size(); ++i) {
  auto &x = v[i];
  // do something with x
}
```

C++ once introduced iterator, but it did not make the code much better:

```
for (auto it = v.begin(); v != v.end(); ++it) {
  auto &x = *it;
  // do something with x
}
```

Hopefully, the problem was solved long ago with C++11 range-based for loop:

```
for (auto& x : v) {
  // do something with x
}
```

Unfortunatelly, this construct only works for simple iteration. For example in Autoware planning module, it is very common to access points by pair (the current point and the next). In such case we are back to the ugly and error-prone:

```
for (int i = 0; i <= v.size()-1; ++i) {
  auto& curr = v[i];
  auto& next = v[i+1];
  // do something with curr and next
}
```

(Note: did you see the loop above reads past the last element?)

With C++23 (!!), such pattern would become very simple to implement:

Category

Design

Labels

None yet

2 participants

```
for (auto &p : v | views::slide(2)) {
  // do something with p[0] and p[1]
}
```

Obviously, I don't think Autoware can switch to C++23 anytime soon. So is Autoware cursed using meaningless indices everywhere for another X years? Actually not, since this C++23 is heavily based on the range-v3 library. With the range-v3 library, the previous loop would look like:

```
for (auto &p : v | views::sliding(2)) {
  // do something with p[0] and p[1]
}
```

This is just a simple example, there are a lot of common patterns that could be greatly simplified with just a few `v | views::XXX | actions::YYY` (a few examples).

I see the range-v3 library a great way to modernize the code base, improve readability, avoid/fix many hard-to-detect out-of-bound access bugs and prepare a future transition to C++23. As far as I know, Autoware code base does not use range-v3, but it has dependencies on it, so the library is already available. Wherever you find yourself using indices, iterators or performing simple operation such as filtering, mapping and such, there is most likely a simpler range-based construct that can do that for you (and bug-free, and optimization friendly).

I know introducing a new library is always a big thing, but even if it is not to rewrite every for loop, I think Autoware would benefit greatly from using it at least for new code. (I am not sure whether there are any policy regarding library usage)

What do you think?

↑ 4 👍 6 ❤️ 3 🚀 1

2 comments

Oldest | Newest | Top

kenji-miyake on Apr 4, 2023

@VRichardJP Thank you for your proposal! FYI, I've discussed a similar proposal in the previous project Autoware.Auto: https://gitlab.com/autowarefoundation/autoware.auto/AutowareAuto/-/issues/1154

Since it's obvious that readability and quality will improve, I'm up for your suggestion!

Also, it's ideal if there is a way to detect and warn old styles automatically. I guess ClangTidy might support it in the future as `modernize-*` rules. https://clang.llvm.org/extra/clang-tidy/checks/list.html

↑ 4 0 replies

VRichardJP on Apr 15, 2023 Collaborator Author edited

To be fair, although I think ranges and views functional-style constructs are far superior to their usual imperative equivalent (improved readability, less error-prone, etc), using `ranges-v3` would come with a cost:

- Many users have reported ranges-v3 increase compilation time a lot. This is partly due to the library emulating C++20 concepts with the good old SFINAE when C++14 or C++17 is used.
- People have reported the STL C++20 ranges are slow to compile aswell. In this case, compilation time depends a lot on the compiler version, most recent versions being faster and faster.
- Finally, the runtime performance could be impacted. Although there is no reason for ranges to be slower than their imperative counterpart, the truth is that compilers have been optimizing imperative-style code for decades. Once again, the most recent compiler version is likely to give the best result.

I don't think any of these points is necessarily deal-breaking. Personally, I would always choose +1 minute compilation time over spending 3 days debugging a buffer overflow. Still, I think these are important points to keep in mind and monitor if we start to use ranges-v3.

For instance, if someone makes a PR with ranges, we could ask to compare:

- module compilation time before and after
- runtime performance before and after (if code logic is unchanged)

↑ 1 👍 2 0 replies