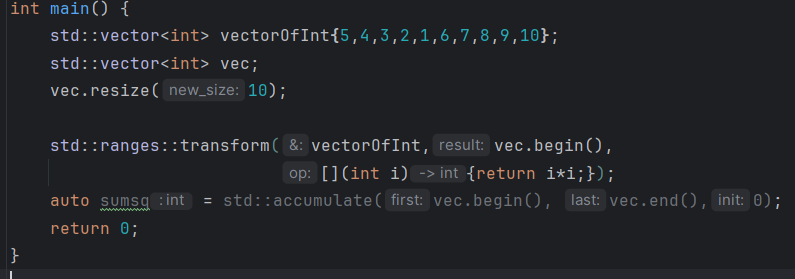
**C++ Advanced Feature: std::ranges**

By Mui Pham

Introduction

The Ranges Library was added to the standard template library in c++20. But before we get into the details of the library, we’d like to ask a few questions to underline the relevance of Ranges. Why should you care about Ranges? How does learning about Ranges further enrich your experience as a programmer? How does this connect to the broader picture?

So why should you care about Ranges? If you found this blog, then I am assuming you are a c++ programmer which likely means you want to run fast programs. If you want to run fast programs, you’ll want to take as many opportunities as you can to use the algorithms library to solve as many subproblems in your code. The algorithms are well vetted to run as efficiently as possible. It is unlikely that you are going to design an algorithm that is faster than the one provided by the standard template library. If you withhold that principle of efficiency, then by extension, you would hold the Ranges to the same standard. Here in the following code, Viewable Ranges saves us one iteration over a container by withholding/delaying the copy/move operation from the first algorithm and does the operation in the second algorithm when the values are needed to be accessed.

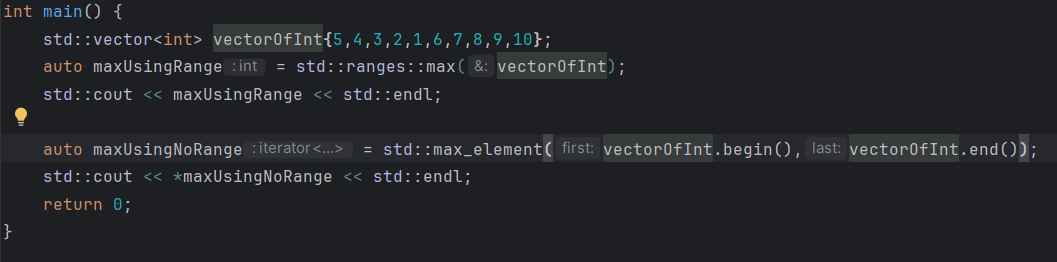


Credit to Tristan Brindle for the code block (cppnorth 2022).

Example of Efficiency

There might come a time where you take an input, copy its values to a container and then pass it to a standard templated algorithm. But if your code needs to modify those values, then you are in a situation where you could benefit from passing those values as an rvalue container. An rvalue container doesn’t copy those values to a variable, potentially saving you a lot of copying if the size of your input is large. The thing with passing an rvalue container to a STL algorithm is that certain algorithms return an iterator. This is VERY DANGEROUS. Rvalue containers die after the statement is run. The iterator pointing to that memory is no longer safe to dereference. In fact, dereferencing an iterator that points to unallocated memory is UNDEFINED BEHAVIOR!!!!! If we were to leverage the Range Library, we can pass rvalue containers safely because the algorithm would no longer return an iterator but another object that wraps around the iterator that doesn’t support the dereferencing operator (this is called a dangling iterator), which means our code will crash at compile time. This is very good because we can immediately see where the flaws are in our code instead of being huge blocks of code away from the infraction. So Ranges also makes our code a little bit safer.

Now, we’d like to address the second question: how does learning Ranges further enrich your experience as a programmer? As the designer of an interface, it makes your code much easier to write and read. Take a look at this example:



The top algorithm uses the Ranges Library to find the max element of the vector of ints. You write an “range::” but you no longer have to pass a beginning and ending iterator to the algorithm. We just pass the container as is. The c++17 way of doing things is written under the top algorithm. Notice how we no longer have to write “range::” but the algorithm for doing the same thing as the top algorithm is longer. The c++17 algorithm is written as “max\_element” which is about the same trade off as the “range::”. But the bigger disadvantage to the c++17 way of doing things is that we have to pass both the beginning iterator and ending iterator by calling the .begin() function and the .end() member function of the container. The line written in c++17 is much longer than the one in c++20 ways of doing things, yet they convey the same amount of information. Therefore c++20 ways of using algorithms is much more legible. Something else to note is that std::ranges::max is much more generalized to handle arguments. Whereas c++17 requires us to use the max\_element algorithm to go over a container. C++17 std::max requires us to pass an initializer list.

So we’ve seen that the Range Library makes copying and moving containers more efficient when possible. We’ve seen that the Range Library makes our code shorter and more readable. The shorter legible code is a small improvement on the experience of programming in c++. But it’s a bunch of these small quality of life improvements that add up to make a large impact. It gets us, c++ programmers, closer to the experience of a python developer.