6-ranges

June 3, 2024

1 C++20 functional pipes with Ranges

> The issue of unnecessary memory allocation arises from the composability problem of STL algorithms. This problem has been known for some time, and a few libraries have been created to fix it. (…) Ranges are currently published as a Technical Specification and are planned for inclusion in C++20. Until ranges become part of the STL, they're available as a third-party library that can be used with most C++11-compatible compilers. Ranges let you have your cake and eat it too, by creating composable smaller functions without any performance penalties. *Ivan Cukic*

1.1 Motivation : composability problem of STL algorithms

1.1.1 Benefits of the STL design

Because the deep copy of a collection is expensive, the design favors **cheap in-place modification of collections**.

Thanks to the use iterators, **algorithms can work on a collection subset**. Some algorithms, such as `std::partition`, are meant to be used repeatedly on different subsets of a given collection.

1.1.2 Yet an impediment to composability

When trying to apply a sequence of transformations on a collection, the STL design leads to store intermediate results in **temporary collections**, with **useless deep copies**, and **numerous iterators**.

[5]:
```
%%file tmp.ranges.h

#include <vector>
#include <algorithm>

bool even( int value ) { return 0==value%2 ; }
int square( int value ) { return value*value ; }

std::vector<int> process( std::vector<int> const & input )
 {
  std::vector<int> intermediary ;
  std::copy_if(input.cbegin(), input.cend(), std::back_inserter(intermediary),␣
 ↪even) ;
```

```
 std::vector<int> output ;
 std::transform(intermediary.cbegin(), intermediary.cend(), std::
↪back_inserter(output), square) ;

 return output ;
}
```

```
Overwriting tmp.ranges.h
```

[6]:
```
%%file tmp.ranges.cpp

#include "tmp.ranges.h"
#include <iostream>

int main()
 {
  for ( auto data : process({ 0, 1, 2, 3, 4, 5 }) )
   { std::cout << data << ' ' ; }
 }
```

```
Overwriting tmp.ranges.cpp
```

[7]:
```
!rm -f tmp.ranges.exe && g++ -std=c++20 tmp.ranges.cpp -o tmp.ranges.exe
```

[8]:
```
!./tmp.ranges.exe
```

```
0 4 16
```

1.2 Introduction about ranges

A *range* basically behave like a pair of begin/end iterators. The usual algorithms have been adapted so that they can receive a range as input, produce a range as output, and can be chained with the operator |.

[9]:
```
%%file tmp.ranges.h

#include <ranges>

int even( int value ) { return 0==value%2 ; }
int square( int value ) { return value*value ; }

auto process( std::initializer_list<int> const & input )
 { return ( input | std::views::filter(even) | std::views::transform(square)  )␣
↪; }
```

```
Overwriting tmp.ranges.h
```

[10]:
```
!rm -f tmp.ranges.exe && g++ -std=c++20 tmp.ranges.cpp -o tmp.ranges.exe
```

[11]:
```
!./tmp.ranges.exe
```

```
0 4 16
```

1.3 `std::views` : lazy algorithms & smart iterators

When they do not need to modify the input collection, the ranges algorithms return some special iterators which are using **lazy evaluation** : the transformation on the values are not made immediatly by the algorithm, but hidden in the iterator and made on the fly.

1.3.1 std::view::filter

This algorithm returns a range whose iterators have a smart increment operator, which is ignoring the elements which do not match the predicate. The input collection is not modified or duplicated in any way. The iterator is filtering the element on the fly.

1.3.2 std::view::transform

Similarly, this algorithm returns a range whose iterators have a smart dereferencing operator, which is applying the transformation when one ask for the pointed value. Again, the input collection is not modified or duplicated in any way. The iterator is transforming the element on the fly.

1.4 `ranges::actions` : active algorithms

Sometimes you cannot avoid modifying the original collection. For this, the range-v3 library provides some other algorithms, called **actions**. Those actions are not lazy, and can be applied to left-value collection with the operator `|=`. They may prove more efficient when we know that a given transformation will be reused many times.

WARNING: apparently, the C++ committee has not kept the idea of some `std::actions` namespace in C++20, and has not kept the operator `|=`.

1.5 Special ranges

- One can customize what is the end of a range. instead of an *end iterator*, we will rather call this a *sentinel*. In a range-based for, the use of sentinels is allowed since C++17.
- A **delimited range** has a fixed, but unknown, size. It can be the values of an `istream`. A predicate is used so to define the end of the range.
- An **infinite range** has no end. This may prive useful when you combine several collections with a `zip`.

In the example below, which also demonstrates the *lazy* nature of those views, we are adding: - `iota(0)` : which is the infinite list of all positive integers, - `take(3)` : which is taking the first three elements of its input range.

[12]:
```
%%file tmp.ranges.cpp

#include <vector>
#include <ranges>
#include <iostream>
```

```
int main()
 {
  for( auto data :
    std::views::iota(0) |
    std::views::filter([]( int value ){ return 0==value%2 ; }) |
    std::views::transform([]( int value ) { return value*value ; }) |
    std::views::take(3) )
   { std::cout << data << ' ' ; }
 }
```

```
Overwriting tmp.ranges.cpp
```

[13]:
```
!rm -f tmp.ranges.exe && g++ -std=c++20 tmp.ranges.cpp -o tmp.ranges.exe
```

[14]:
```
!./tmp.ranges.exe
```

```
0 4 16
```

1.6 Availability

GCC 10 offers the best support today (except the *actions*, apparently not adopted in the standard). MSVC 19.28 has partial support. Two other external implementations are available : * The reference implementation which has inspired C++20 : **Range-v3**, by Eric Niebler. * The older mature library, compatible with older C++ : **Boost.Range**.

2 Questions ?

3 Sources

- Cpp Reference
- Marius Bancila's Blog

© CNRS 2024
Assembled and written by David Chamont, reviewed and extended by Hadrien Grasland, this work is made available according to the terms of the
Creative Commons License - Attribution - NonCommercial - ShareAlike 4.0 International