9.3. Auxiliary Iterator Functions

The C++ standard library provides some auxiliary functions for dealing with iterators: `advance()`, `next()`, `prev()`, `distance()`, and `iter_swap()`. The first four give all iterators some abilities usually provided only for random-access iterators: to step more than one element forward (or backward) and to process the difference between iterators. The last auxiliary function allows you to swap the values of two iterators.

9.3.1. advance()

The function `advance()` increments the position of an iterator passed as the argument. Thus, the function lets the iterator step forward (or backward) more than one element:

```
#include <iterator>
void advance (InputIterator& pos, Dist n)
```

- Lets the input iterator *pos* step *n* elements forward (or backward).
- For bidirectional and random-access iterators, *n* may be negative to step backward.
- `Dist` is a template type. Normally, it must be an integral type because operations such as `<`, `++`, `--`, and comparisons with `0` are called.
- Note that `advance()` does *not* check whether it crosses the `end()` of a sequence (it can't check because iterators in general do not know the containers on which they operate). Thus, calling this function might result in undefined behavior because calling operator `++` for the end of a sequence is not defined.

Due to the use of iterator traits (see Section 9.5, page 466), the function always uses the best implementation, depending on the iterator category. For random-access iterators, it simply calls *pos*+=*n*. Thus, for such iterators, `advance()` has constant complexity. For all other iterators, it calls `++` *pos n* times (or `--` *pos* if *n* is negative). Thus, for all other iterator categories, `advance()` has linear complexity.

To be able to change container and iterator types, you should use `advance()` rather than operator `+=`. In doing so, however, be aware that you risk unintended worse performance. The reason is that you don't recognize that the performance is worsening when you use other containers that don't provide random-access iterators (bad runtime is the reason why operator `+=` is provided only for random-access iterators). Note also that `advance()` does not return anything. Operator `+=` returns the new position, so it might be part of a larger expression. Here is an example of the use of `advance()`:

```
// iter/advance1.cpp

#include <iterator>
#include <iostream>
#include <list>
#include <algorithm>
using namespace std;

int main()
{
    list<int> coll;

    // insert elements from 1 to 9
    for (int i=1; i<=9; ++i) {
        coll.push_back(i);
    }

    list<int>::iterator pos = coll.begin();

    // print actual element
    cout << *pos << endl;

    // step three elements forward
    advance (pos, 3);

    // print actual element
    cout << *pos << endl;

    // step one element backward
    advance (pos, -1);
```

```
        // print actual element
        cout << *pos << endl;
    }
```

In this program, `advance()` lets the iterator `pos` step three elements forward and one element backward. Thus, the output is as follows:

```
    1
    4
    3
```

Another way to use `advance()` is to ignore some input for iterators that read from an input stream. See the example in Section 9.4.3, page 465.

9.3.2. `next()` and `prev()`

Since C++11, two additional helper functions allow you to move to following or previous iterator positions.

```
    #include <iterator>
    ForwardIterator next (ForwardIterator pos)
    ForwardIterator next (ForwardIterator pos, Dist n)
```

- Yields the position the forward iterator *pos* would have if moved forward 1 or *n* positions.
- For bidirectional and random-access iterators, *n* may be negative to yield previous positions.
- `Dist` is type `std::iterator_traits<ForwardIterator>::difference_type` .
- Calls `advance(` *pos* `,` *n* `)` for an internal temporary object.
- Note that `next()` does *not* check whether it crosses the `end()` of a sequence. Thus, it is up to the caller to ensure that the result is valid.

Click here to view code image

```
    #include <iterator>
    BidirectionalIterator prev (BidirectionalIterator pos)
    BidirectionalIterator prev (BidirectionalIterator pos, Dist n)
```

- Yields the position the bidirectional iterator *pos* would have if moved backward 1 or *n* positions.
- *n* may be negative to yield following positions.
- `Dist` is type. `std::iterator_traits<BidirectionalIterator>::difference_type` .
- Calls `advance(` *pos*,-*n* `)` for an internal temporary object.
- Note that `prev()` does *not* check whether it crosses the `begin()` of a sequence. Thus, it is up to the caller to ensure that the result is valid.

This allows, for example, running over a collection while checking values of the next element:

Click here to view code image

```
    auto pos = coll.begin();
    while (pos != coll.end() && std::next(pos) != coll.end()) {
        ...
        ++pos;
    }
```

Doing so especially helps because forward and bidirectional iterators do not provide operators `+` and `-` . Otherwise, you always need a temporary:

```
    auto pos = coll.begin();
    auto nextPos = pos;
    ++nextPos;
    while (pos != coll.end() && nextPos != coll.end()) {
         ...
         ++pos;
         ++nextPos;
    }
```

or have to restrict code to random-access iterators only:

```
    auto pos = coll.begin();
    while (pos != coll.end() && pos+1 != coll.end()) {
         ...
         ++pos;
```

```
}
```

Don't forget to ensure that there is a valid position before you use it (for this reason, we first check whether `pos` is equal to `coll.end()` before we check the next position).

Another application of `next()` and `prev()` is to avoid expressions, such as `++coll.begin()`, to deal with the second element of a collection. The problem is that using `++coll.begin()` instead of `std::next(coll.begin())` might not compile (see Section 9.2.6, page 440, for details).

A third application of `next()` is to work with `forward_list`s and `before_begin()` (see Section 7.6.2, page 307, for an example).

9.3.3. distance()

The `distance()` function is provided to process the difference between two iterators:

```
#include <iterator>
Dist distance (InputIterator pos1, InputIterator pos2)
```

- Returns the distance between the input iterators *pos1* and *pos2*.
- Both iterators have to refer to elements of the same container.
- If the iterators are not random-access iterators, *pos2* must be reachable from *pos1*; that is, it must have the same position or a later position.
- The return type, *Dist*, is the difference type according to the iterator type:

  ```
  iterator_traits<InputIterator>::difference_type
  ```

  See Section 9.5, page 466, for details.

By using iterator tags, this function uses the best implementation according to the iterator category. For random-access iterators, this function simply returns *pos2-pos1*. Thus, for such iterators, `distance()` has constant complexity. For all other iterator categories, *pos1* is incremented until it reaches *pos2* and the number of increments is returned. Thus, for all other iterator categories, `distance()` has linear complexity. Therefore, `distance()` has bad performance for other than random-access iterators. You should consider avoiding it.

The implementation of `distance()` is described in Section 9.5.1, page 470. The following example demonstrates its use:

```
// iter/distance1.cpp

#include <iterator>
#include <iostream>
#include <list>
#include <algorithm>
using namespace std;

int main()
{
    list<int> coll;

    // insert elements from -3 to 9
    for (int i=-3; i<=9; ++i) {
        coll.push_back(i);
    }

    // search element with value 5
    list<int>::iterator pos;
    pos = find (coll.begin(), coll.end(),     // range
                5);                           // value

    if (pos != coll.end()) {
        // process and print difference from the beginning
        cout << "difference between beginning and 5: "
             << distance(coll.begin(),pos) << endl;
    }
    else {
        cout << "5 not found" << endl;
    }
}
```

After `find()` assigns the position of the element with value `5` to `pos`, `distance()` uses this position to process the difference between this position and the beginning. The output of the program is as follows:

```
difference between beginning and 5: 8
```

To be able to change iterator and container types, you should use `distance()` instead of operator `-`. However, if you use `distance()`, you might not recognize that the performance gets worse when you switch from random-access iterators to other iterators.

To compute the difference between two iterators that are not random-access iterators, you must be careful. The first iterator must refer to an element that is not after the element of the second iterator. Otherwise, the behavior is undefined. If you don't know which iterator position comes first, you have to obtain the distance between both iterators to the beginning of the container and compute the difference of these distances. However, you must then know to which container the iterators refer. Otherwise, you have no chance of processing the difference of the two iterators without running into undefined behavior. See the remarks about subranges in Section 6.4.1, page 205, for additional aspects of this problem.

9.3.4. iter_swap()

This simple auxiliary function is provided to swap the values to which two iterators refer:

```
#include <algorithm>
void iter_swap (ForwardIterator1 pos1, ForwardIterator2 pos2)
```

- Swaps the values to which iterators *pos1* and *pos2* refer.
- The iterators don't need to have the same type. However, the values must be assignable.

Here is a simple example (function `PRINT_ELEMENTS()` is introduced in Section 6.6, page 216):

```
// iter/iterswap1.cpp

#include <iostream>
#include <list>
#include <algorithm>
#include <iterator>

#include "print.hpp"
using namespace std;

int main()
{
    list<int> coll;

    // insert elements from 1 to 9
    for (int i=1; i<=9; ++i) {
        coll.push_back(i);
    }

    PRINT_ELEMENTS(coll);

    // swap first and second value
    iter_swap (coll.begin(), next(coll.begin()));

    PRINT_ELEMENTS(coll);

    // swap first and last value
    iter_swap (coll.begin(), prev(coll.end()));

    PRINT_ELEMENTS(coll);
}
```

The output of the program is as follows:

```
1 2 3 4 5 6 7 8 9
2 1 3 4 5 6 7 8 9
9 1 3 4 5 6 7 8 2
```

Note that `next()` and `prev()` are provided since C++11, and that using operators `++` and `--` instead might not compile for each container:

Click here to view code image

```
vector<int> coll;
...
iter_swap (coll.begin(), ++coll.begin());    // ERROR: might not compile
...
iter_swap (coll.begin(), --coll.end());      // ERROR: might not compile
```

See Section 9.2.6, page 440, for details about this problem.