Universidade Federal do Rio Grande do Norte
Instituto Metrópole Digital

Programming Language I • DIM0120
◁ Exercises on Template Functions ▷
September 26, 2019

Contents

1 Introduction

In this exercise you should develop a series of functions that represent typical algorithms for array manipulation. Because we are still following an *imperative programming paradigm*, all functions should receive the range we wish to operate on as an argument.

The primary goal of this exercise is to practice a higher level of *abstract programming*, where our code does not need to know in advance which type of data we are operating on. To do that our code requires the use of **function template** in association with **function pointers**. The former supports the passage of *generic arguments* to functions, whereas the latter helps us to defer some decisions to the client by allowing them to provide the *code to operate on data*.

The secondary objective is to acquire programming experience by building a library of typical algorithms on arrays, called `graal` —*GeneRic Array Algorithms Library*. By building this generic library we want to demonstrate the importance of programming abstraction and code reuse while developing an application in the next exercise.

2 The Iterator Programming Pattern

An **iterator** is a *programming pattern* that usually is represented by an object (in the context of Object Oriented Programming) that can traverse (or iterate over) a **container** object without having to know how the container works internally. In the STL library, this is the primary method for accessing elements in lists and associative classes.

For instance, if we wish to iterate over all the elements in, say, a `std::vector` of integers, to print its content we would probably write a code like this:

```
1 #include <iostream>
```

```cpp
#include <vector>
using namespace std;
int main() {
    vector<int> vect;                // Creating an vector of integers.
    for (auto i(0) ; i < 6 ; ++nCount)
        vect.push_back(i);           // Inserting some elements into the vector.

    vector<int>::const_iterator it; // Declare a read-only iterator
    it = vect.begin();               // Assign it to the start of the vector
    while (it != vect.end()) {       // While it hasn't reach the end
        cout << *it << " ";          // print the value of the element it points to
        ++it;                        // and iterate to the next element
    }

    cout << endl;
}
```

Notice how similar the use of iterator is to a regular **pointer**. In fact, we may informally say that a iterator represents a pointer assigned to an element inside a container class.

Later, when we begin our study on the STL library, you will see that all container classes include four basic member functions to help us navigate them:

- ⋆ `begin()` returns an iterator pointing to the address of the first element in the container.
- ⋆ `end()` returns an iterator pointing to the address **just past the last element** of the container.
- ⋆ `cbegin()` returns a constant (read-only) iterator pointing to the address of the first element in the container.
- ⋆ `cend()` returns a constant (read-only) iterator pointing to the address **just past the last element** of the container.

The important fact is that both `end()` and `cend()` iterators always point to an address past the last element of the container (see Figure 1). So, if we wish to define a valid range of elements in any container we would do so by defining a range [`begin()` ; `end()`)—notice that we are defining a closed-open interval!

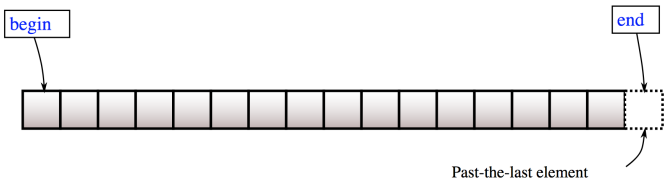

Figure 1: Visual interpretation of the `begin`, and `end` iterators in a container.
Source: http://upload.cppreference.com/mwiki/images/1/1b/range-begin-end.svg

3 Exercises

In the following exercises we want to support this iterator behavior by accepting template parameters that might either become a regular pointer or an STL's iterator, both defined over a sequential chunk of memory (a **static array**, or a **STL containers**). So, every time we pass data to a function, we need to inform also the range over which the function's operation should take place. For instance, if the client code wishes to **reverse** the elements of **an entire array**, s/he should pass to the function `reverse` (Exercise 2) two pointers: one pointing to the beginning of the range, and other pointing to the location just past the last element, hence defining the range to be reversed. Similarly, if the client wants to reverse the elements of a, say, `std::vector`, s/he may use the same `reverse` function you coded, but this time s/he must pass in the `vector`'s iterators `begin()` and `end()`. In terms of programming, it does not matter at all, because you are using **templates**. Remember that this programming feature "generates" code on demand, for each of these two possibilities (raw pointer or iterators) during the compiling step.

This is accomplished by calling `reverse( std::begin(A), std::end(A), ...)`, where `std::begin()` and `std::end()` are functions defined in `<iterator>` that return regular pointers, respectively, to the beginning of and to past the last element of a container or regular static array[1]. On the other hand, if the client wants to reverse only, say, the first 4 elements of an array, s/he should call `reverse( std::begin(A), std::begin(A)+4,...)`.

1. Implement a function called `minmax` that finds and returns the first occurrence of the smallest and the greatest elements in a range [`first`, `last`). Assume that the input range is valid. One possible prototype for this function is:

```
template< typename Itr, typename Compare >
std::pair<Itr,Itr> minmax( Itr first, Itr last, Compare cmp );
```

Parameters

- ⋆ `first`, `last` - the range of elements to examine.
- ⋆ `cmp` - binary function that returns `true` if the first element is *less* than the second element, or `false` otherwise.

  The signature of the compare function should be equivalent to the following:

```
bool cmp( const Type &a, const Type &b);
```

  The type `Type` must be such that an object of the type `Itr` can be dereferenced and then implicitly converted to `Type`, as in `cmp(*first, *last)`, for instance.

Return value

A pair in which the first component points to the smallest element, and the second component points to the greatest element. Returns `std::make_pair(first, first)` if the range is empty. If we have several elements with the same value as the smallest, for instance, you should return the iteartor to *the first occurance* of such element in

[1] This does not work on dynamic allocated arrays.

the range. On the other hand, if we have several elements equal to the greatest, your function should return the iterator to the *last occurance* of such element in the range.

Complexity

Linear or, exactly `first` - `last` -1 comparisons.

2. Implement a function called `reverse` that reverses the order of the elements in a range. The prototype for this function is:

```
template < typename Itr >
void reverse( Itr first, Itr last );
```

Parameters

⋆ `first`, `last` - the range of elements to examine.

3. Implement a function called `copy` that receives a range [`first`; `last`) and copies the range values into a new range beginning at `d_first`. Assume the destination range has enough memory space to receive the copied elements. Therefore, the function **must not allocate any memory**. The function shall return an iterator pointing to the address **just past the last element** of the destination range, i.e. the range at the receiving end of the copy operation. The prototype for this function is:

```
template < typename Itr >
Itr copy( Itr first, Itr last, Itr d_first );
```

Parameters

⋆ `first`, `last` - the range of elements to examine.

⋆ `d_first` - the beginning of the range the elements are copied to.

Return value

Iterator to the position just past the last element copied.

4. Implement a function called `find_if` that receives a range [`first`; `last`) over an array, and returns a pointer (iterator) to the first element in the range for which a given predicate `p` returns `true`. If the predicate is false for **all** elements, the function should return a pointer to `last`.

The function must perform a linear search. The predicate is passed to the function as a function pointer. The prototype for this function is:

```
template < typename Itr, typename Predicate >
Itr find_if( Itr first, Itr last, Predicate p )
```

Parameters

⋆ `first`, `last` - the range of elements to examine.

⋆ `p` - unary predicate which return `true` for the required element.
The signature of the predicate function should be equivalent to the following:

```
bool p( const Type& a );
```

The type `Type` must be such that an object of the type `Itr` can be dereferenced and then implicitly converted to `Type`, as in `p(*first)`, for instance.

Return value

Pointer to the first element satisfying the condition or last if no such element is found.

5. Implement a function called `find` that receives a range [`first`;`last`), a target value, and returns a pointer (iterator) to the first element in the range that is equal to the value passed to the function. If the value is not found in the range, the function returns `last`.

   The equality test is done with a function pointer provided by the client code, which returns `true` if two elements passed as arguments are equal.

   The prototype for this function is:

```
template < typename Itr, typename T, typename Equal >
Itr find( Itr first, Itr last, const T & value, Equal eq )
```

   Parameters

   - `first`, `last` - the range of elements to examine.
   - `value` - value to compare the elements to.
   - `eq` - binary function that returns `true` if the elements are equal, or `false` otherwise.

     The signature of the equal to function should be equivalent to the following:

```
bool eq( const Type &a, const Type &b);
```

     The type `Type` must be such that an object of the type `Itr` can be dereferenced and then implicitly converted to `Type`, as in `eq(*first, *last)`, for instance.

   Return value

   Pointer to the first element equal to `value` or last if no such element is found.

   Complexity

   Linear or, at most `first` - `last` applications of `eq()`.

6. Implement three related functions, `all_of`, `any_of`, and `none_of`. All three functions receive a range [`first`,`last`), and a predicate `p`. The `all_of` returns `true` if the predicate returns `true` for **all elements** in the range. The `any_of` returns `true` if the predicate returns `true` for **at least one element** in the range. The `none_of` returns `true` if the predicate returns `true` for **none of elements** in the range.

   The signature for these functions are:

```
template <typename Itr, typename Predicate>
bool all_of( Itr first, Itr last, Predicate p );
template <typename Itr, typename Predicate>
bool any_of( Itr first, Itr last, Predicate p );
template <typename Itr, typename Predicate>
bool none_of( Itr first, Itr last, Predicate p );
```

   Parameters

   - `first`, `last` - the range of elements to examine.

⋆ `p` - unary predicate which return `true` for the required element.
The signature of the predicate function should be equivalent to the following:

```
bool p( const Type& a );
```

The type `Type` must be such that an object of the type `Itr` can be dereferenced and then implicitly converted to `Type`, as in `p(*first)`, for instance.

Return value

- `all_of`: `true` if unary predicate returns `true` for all elements in the range, `false` otherwise. Returns `true` if the range is empty.
- `any_of`: `true` if unary predicate returns `true` for at least one element in the range, `false` otherwise. Returns `true` if the range is empty.
- `none_of`: `true` if unary predicate returns `true` for no elements in the range, `false` otherwise. Returns `true` if the range is empty.

Complexity

Linear or, at most `first` - `last` applications of the predicate.

7. Implement two **overloaded**[2] functions called `equal`. The first version should receive a range [`first1`; `last1`), and a pointer to the beginning of another range `first2`; and return `true` if the first range if **equal** to the range [`first2`; `first2` + (`last1` - `first1`)), and `false` otherwise. The second version should receive two ranges, [`first1`; `last1`), and [`first2`; `last2`); and return `true` if the elements in the ranges are equal, and `false` otherwise. The prototype for these functions are:

```
template < typename Itr, typename Equal >
bool equal( Itr first, Itr last, Itr first2, Equal eq )
template < typename Itr, typename Equal >
bool equal( Itr first, Itr last, Itr first2, Itr last2, Equal eq )
```

Parameters

⋆ `first`, `last` - the range of elements to examine.
⋆ `first2`, `last2` - the range of elements to compare to.
⋆ `eq` - binary function that returns `true` if the elements are equal, or `false` otherwise.
The signature of the equal to function should be equivalent to the following:

```
bool eq( const Type &a, const Type &b);
```

The type `Type` must be such that an object of the type `Itr` can be dereferenced and then implicitly converted to `Type`, as in `eq(*first, *last)`, for instance.

Return value

`true` if all elements in both ranges are equal; `false` otherwise.

[2] **Function overloading** is a feature of C++ that allows us to create multiple functions with the *same name*, so long as they have different parameters.

8. Implement a function called `unique` that receives a a range [`first`;`last`) over an array, reorders the elements in the range [`first`, `last`) in such a way that all unique elements should appear at the beginning of the range. Relative order of the original elements **must be preserved**. The function should return a pointer to the address the address **just past the last element** of the range with unique elements. The signature for this function is:

```
template < typename Itr, typename Equal >
Itr unique( Itr first, Itr last, Equal eq );
```

Parameters

- ⋆ `first`, `last` - the range of elements to examine.
- ⋆ `eq` - binary function that returns `true` if the elements are equal, or `false` otherwise.
  The signature of the equal to function should be equivalent to the following:

  ```
  bool eq( const Type &a, const Type &b);
  ```

  The type `Type` must be such that an object of the type `Itr` can be dereferenced and then implicitly converted to `Type`, as in `eq(*first, *last)`, for instance.

Return value

An iteratorto the address just past the last element of the range with unique elements.

9. Implement a function called `partition` that receives a a range [`first`;`last`) over an array, reorders the elements in the range [`first`, `last`) in such a way that all elements for which a given predicate `p` returns `true` precede the elements for which predicate `p` returns `false`. The function should return a pointer to the begining of the range for which predicate `p` return `false`, i.e. the second rante. Relative order of the elements may not be preserved. The signature for this function might be:

```
template < typename Itr, typename Predicate >
Itr partition( Itr first, Itr last, Predicate p );
```

Parameters

- ⋆ `first`, `last` - the range of elements to examine.
- ⋆ `p` - unary predicate which return `true` for the required element.
  The signature of the predicate function should be equivalent to the following:

  ```
  bool p( const Type& a );
  ```

  The type `Type` must be such that an object of the type `Itr` can be dereferenced and then implicitly converted to `Type`, as in `p(*first)`, for instance.

Return value

An iterator to the begining of the range for which predicate `p` return `false`.

Complexity

Linear or `first` - `last` applications of the predicate.

10. Implement a function called `sort` that receives and sorts the elements in ascending order. The order of equal elements is not guaranteed to be preserved. The function should also receive a function pointer to a comparison function that receives two arguments `a`, and `b`, and returns `true` if $a < b$, or `false` otherwise.

    The signature for this function might be:

    ```
    template < typename Itr, typename Comparison >
    Itr sort( Itr first, Itr last, Comparison comp );
    ```

    Parameters

    - ⋆ `first`, `last` - the range of elements to examine.
    - ⋆ `comp` - a comparison function object which returns `true` if the first argument is less than (i.e. is ordered before) the second. The signature of the comparison function should be equivalent to the following:

      ```
      bool comp( const Type& a, const Type& b );
      ```

      The type `Type` must be such that an object of the type `Itr` can be dereferenced and then implicitly converted to `Type`, as in `cmp(*first, *(first+1))`, for instance.

11. Implement a function called `rotate` that receives a range [`first`; `last`) and an iterator `n_first` in that range and does a left rotation on the elements in [`first`; `last`) in such a way that `n_first` becomes the first element of the rotated range and `n_first-1` becomes the last element. A precondition of this function is that [`first`; `n_first`) [`n_first`; `last`) are valid ranges. The function should return an iterator to the new location of the element, pointed by `first`. The signature for this function might be:

    ```
    template < typename Itr >
    Itr rotate( Itr first, Itr n_first, Itr last );
    ```

    Parameters

    - ⋆ `first`, `last` - the range of elements to rotate.
    - ⋆ `n_first` - the element that should appear at the beginning of the rotated range.

    Return value

    An iterator to the new location of the element pointed by the original `first`.

    Complexity

    Linear in the distance `first` and `last`.

∼ The End ∼