Universidade Federal do Rio Grande do Norte
Instituto Metrópole Digital

Programming Language I • DIM0120
◁ Exercise on Generic Pointer and Function Pointer ▷
September 24, 2018

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 generic pointer manipulation. 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>
2 #include <vector>
3 using namespace std;
4 int main() {
```

```
5     vector<int> vect;                // Creating an vector of integers.
6     for (auto i(0) ; i < 6 ; ++nCount)
7         vect.push_back(i);           // Inserting some elements into the vector.
8
9     vector<int>::const_iterator it; // Declare a read-only iterator
10    it = vect.begin();               // Assign it to the start of the vector
11    while (it != vect.end()) {       // While it hasn't reach the end
12        cout << *it << " ";          // print the value of the element it points to
13        ++it;                        // and iterate to the next element
14    }
15
16    cout << endl;
17 }
```

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!

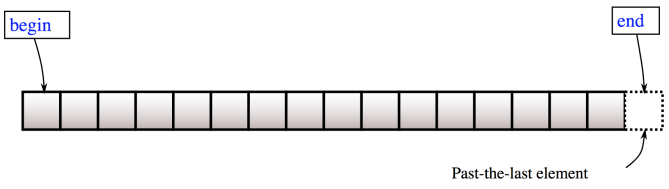

Figure 1: Visual interpretation of the `begin`, and `end` iterators in a container.
Source: http://upload.cppreference.com/mwiki/images/1/1b/range-begin-end.svg

In this exercise we want to mimic this iterator behavior while working with regular generic pointers representing **arrays**, rather than the STL containers. So, every time we pass a generic array 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 in Section 5) 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.

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,...)`.

3 Memory Handling Functions

Besides the iterators (or pointers) defining a range of action, any functions that has to operate on generic arrays should ask the client code to provide the size, in bytes, of single element of the array we want to work on.

This piece of information is necessary in order to call any one of the three functions designed to manage chunks of bytes on memory. These functions, defined in `<cstring>`, are:

1. `std::memcpy`: copies a certain amount of bytes from one location (source) to another (destination).
2. `std::memmove`: moves a certain amount of bytes from one location (source) to another (destination). The source and destination may overlap.
3. `std::memset` copies a certain value (interpreted as `unsigned char`) into each of the first `count` characters of the objected pointed by a generic pointer (destination). If the `count` is greater than the destination region, the behavior is undefined.

The locations mentioned above are given by generic pointers. In the context of this exercise, these locations would most likely be iterators (or pointers) to certain parts of the array.

4 Pointer Arithmetic

Finally, before invoking any of the three memory handling function mentioned in Section 3, you should recall that it is necessary to convert the generic pointer to a specific C++ type, otherwise the compiler will display an error message stating that it's not possible to operate on `void *`.

Consequently, we nee to cast the generic pointer into a type that allows us to manipulate multiples of a single byte. The basic C++ type suggested by the functions explained in Section 3 is `unsigned char`. Hence, all pointer cast should convert `void *` to `unsigned char *`. To make code writing easier, we suggest to define a short alias for this type by declaring

```
using byte = unsigned char;
```

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

after we have done that, we may use `byte` as a regular casting type.

5 Exercises

1. Implement a function called `min` that finds and returns the first occurrence of the smallest element in a range [`first`, `last`) defined over a generic array. Assume that the input range is valid. One possible prototype for this function is:

```
const void * min( const void *first, const void *last, size_t size, Compare cmp );
```

   - ⋆ `first`, `last` - the range of elementos to examine.
   - ⋆ `size` - size of each element in the array in bytes.
   - ⋆ `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 void *, const void * );
```

     We may use an alias to this function pointer by declaring:

```
using Compare = bool (*)(const void *, const void *);
```

   Return value
   Pointer to the smallest element of the range.
   Complexity
   Linear or, exactly `first` - `last` -1 comparisons.

2. Implement a function called `reverse` that reverses the order of the elements located in a range defined over a generic array. One possible prototype for this function is:

```
void reverse( void * first, void * last, size_t size );
```

   Parameters
   - ⋆ `first`, `last` - the range of elementos to examine.
   - ⋆ `size` - size of each element in the array in bytes.

3. Implement a function called `copy` that receives a range [`first`; `last`) defined over an array 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. One possible prototype for this function is:

```
void * copy( const void * first, const void * last, const void * d_first, size_t size );
```

   Parameters
   - ⋆ `first`, `last` - the range of elementos to examine.
   - ⋆ `size` - size of each element in the range in bytes.

Return value

Pointer to the memory area that contains the copy of the original range.

4. Implement a function called `clone` that receives a range defined over an array and returns a pointer to a new array containing a copy of the original range. The function **must allocate the memory necessary to hold the data**. If the range passed to the function corresponds to the entire array, the function return a "clone" of the array. Notice, though, that the function performs a so called "shallow copy" of the original (range) array. One possible prototype for this function is:

```
void * clone( const void * first, const void * last, size_t size );
```

Parameters

- ⋆ `first`, `last` - the range of elementos to examine.
- ⋆ `size` - size of each element in the range in bytes.

Return value

Pointer to the memory area that contains the copy of the original range.

5. 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. One possible prototype for this function is:

```
const void *
find_if( const void * first, const void * last, size_t size, Predicate p )
```

Parameters

- ⋆ `first`, `last` - the range of elementos to examine.
- ⋆ `size` - size of each element in the range in bytes.
- ⋆ `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 void * );
```

  Again, we may use an alias to this function pointer by declaring:

```
using Predicate = bool (*)(const void *);
```

  or

```
using Predicate = std::function< bool( const void * ) >;
```

Return value

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

6. Implement a function called `find` that receives a range [`first`;`last`) over an array, 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 that returns `true` if two elements passed as arguments are equal.

   One possible prototype for this function is:

   ```
   const void *
   find( const void * first, const void * last, size_t size,
         const void * value, Equal eq )
   ```

   Parameters

   - ⋆ `first`, `last` - the range of elementos to examine.
   - ⋆ `size` - size of each element in the range in bytes.
   - ⋆ `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 void *, const void * );
     ```

     Again, we may use an alias to this function pointer by declaring:

     ```
     using Equal = bool (*)(const void *, const void *);
     ```

     or

     ```
     using Equal = std::function< bool( const void *, const void * ) >;
     ```

   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()`.

7. Implement three related functions, `all_of`, `any_of`, and `none_of`. All three functions receive a generic 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 might be:

   ```
   bool all_of( const void * first, const void * last, size_t size, Predicate p );
   bool any_of( const void * first, const void * last, size_t size, Predicate p );
   bool none_of( const void * first, const void * last, size_t size, Predicate p );
   ```

   Parameters

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

- ⋆ `size` - size of each element in the range in bytes.
- ⋆ `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 void * );
```

  We may use an alias to this function pointer by declaring:

```
using Predicate = bool (*)(const void *);
```

  or

```
using Predicate = std::function< bool( const void * ) >;
```

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.

8. 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.

   In both cases the equality should be tested with a function pointer that receives two generic elements and return `true` if they are equal (according to the client code definition).

9. 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 might be:

```
void * unique( void * first, void * last, size_t size, Equal eq );
```

Parameters

[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.

- ⋆ `first`, `last` - the range of elementos to examine.
- ⋆ `size` - size of each element in the range in bytes.
- ⋆ `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 void *, const void * );
```

  We may use an alias to this function pointer by declaring:

```
using Equal = bool (*)(const void *, const void *);
```

Return value

A pointer to the address just past the last element of the range with unique elements.

Complexity

Linear or `first` - `last` applications of the equal function.

10. 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:

```
void * partition( void * first, void * last, size_t size, Predicate p );
```

Parameters

- ⋆ `first`, `last` - the range of elementos to examine.
- ⋆ `size` - size of each element in the range in bytes.
- ⋆ `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 void * );
```

  We may use an alias to this function pointer by declaring:

```
using Predicate = bool (*)(const void *);
```

  or

```
using Predicate = std::function< bool( const void * ) >;
```

Return value

A pointer to the begining of the range for which predicate `p` return `false`.

Complexity

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

11. 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 generic pointers `a`, and `b`, and returns `true` if $a < b$, or `false` otherwise.

Hint: use a signature similar to `std::qsort`. But remember that this version is slightly different since the comparison function returns a boolean rather than an integer.

$\sim$ The End $\sim$