Universidade Federal do Rio Grande do Norte
Instituto Metrópole Digital

Basic Data Structure I • DIM0119

◁ Programming Project #1: Implementing the List Abstract Data Type ▷

2 de maio de 2019

Overview

In this document we describe the *list Abstract Data Type* (ADT). We focus on two aspects of the list: the core operations that should be supported, and the two different forms of organizing the data inside a list, using arrays(static or dynamic) or linked list (singly or doubly linked).

In this initial programming project, we begin by introducing basic definition of terms, properties and operations. Next we provide details on implementing a list ADT with dynamic array.

Sumário

1 Definition of a List

We define a *general linear list* as the set of $n \geq 0$ elements $A[0], A[1], A[2], \ldots, A[n-1]$. We say that the size of the list is n and we call a list with size $n = 0$ an **empty list**. The structural properties of a list comes, exclusively, from the relative position of its elements:-

- if $n > 0$, $A[0]$ is the first element,
- for $0 < k \leq n$, the element $A[k-1]$ precedes $A[k]$.

Therefore, the first element of a list is $A[0]$ and the last element is $A[n-1]$. The **position** of element $A[i]$ in a list is i.

In addition to the structural properties just described, a list is also defined by the set of operations it supports. Typical list operations are to print the elements in the list; to make it empty; to access one element at a specific position within a list; to insert a new element at one of the list's ends, or at a specific location within the list; to remove one element at a given location within a list, or a range of elements; to inquire whether a list is empty or not; to get the size of the list, and so on.

Depending on the **implementation** of a list ADT, we may need to support other operations or suppress some of them. The basic factor that determines which operations we may support in our implementation is their performance, expressed in terms of time complexity. For example, the time complexity of inserting elements at the beginning of a list ADT implemented with array is $O(n)$, with the undesired side effect of having to shift all the elements already stored in the list to make space for the new element.

In this document we discuss how to implement a list ADT based on a *dynamic arrays* data structure. This versions of a list is equivalent to the `std::vector`.

Later on, after learning how to implement *linked lists*, we will revisit the list ADT to explore alternative ways of implementing this ADT.

2 The List ADT

In this section we present the core set of operations a list ADT should support, regardless of the underlying data structure one may choose to implement a list with.

Most of the operations presented here and in the next sections follow the naming convention and behavior adopted by the STL containers.

2.1 Constructors, Destructors, and Assignment

Usually a class provides more than one type of constructor. Next you find a list of constructors that should be supported by our `vector` class.

Notice that the name of the class we want to implement is also `vector`. Therefore, to distinguish your `vector` from the STL's, we shall define our class within the namespace `sc` (a short for *sequence container*), which means the full name of the class shall be `sc::vector`.

In the following specifications, consider that `T` represents the template type.

Notice that all references to either a return type or variable declaration related to the size of a list are defined as `size_type`. This is basically an alias to some unsigned integral type, such as `long int`, `size_t`, for example. The use of an alias such as this is a good programming practice that enables better code maintenance. Typically we are going to define an alias at the beginning of our class definition with `typedef` or `using` keywords, as for instance in `using size_type = unsigned long`.

`vector( );`	(1)
`explicit vector( size_type count );`	(2)
`template< typename InputIt >` `vector( InputIt first, InputIt last );`	(3)
`vector( const vector& other );`	~~(4)~~
`vector( std::initializer_list<T> ilist );`	~~(5)~~
`~vector( );`	~~(6)~~
`vector& operator=( const vector& other );`	(7)
`vector& operator=( std::initializer_list<T> ilist );`	(8)

(1) Default constructor that creates an empty list.
(2) Constructs the list with `count` default-inserted instances of `T`.
(3) Constructs the list with the contents of the range `[first, last)`.
(4) Copy constructor. Constructs the list with the *deep* copy of the contents of `other`.
(5) Constructs the list with the contents of the initializer list `init`.
(6) Destructs the list. The destructors of the elements are called and the used storage is deallocated. Note, that if the elements are pointers, the pointed-to objects are not destroyed.
(7) Copy assignment operator. Replaces the contents with a copy of the contents of other. (i.e. the data in `other` is moved from `other` into this container). `other` is in a valid but unspecified state afterwards.
(8) Replaces the contents with those identified by initializer list `ilist`.

The two `operator=()` (overloaded) assign methods return `*this` at the end, so we may have multiple assignments in a single command line, such as `a = b = c = d;`.

Parameters

`count` - the size of the list.
`value` - the value to initialize the list with.
`fist, last` - the range to copy the elements from.
`other` - another list to be used as source to initialize the elements of the list with.
`ilist` - initializer list to initialize the elements of the list with.

2.2 Common operations to all list implementations

- `size_type size() const`: return the number of elements in the container.
- `void clear()`: remove (either logically or physically) all elements from the container.
- `bool empty()`: returns `true` if the container contains no elements, and `false` otherwise.
- `void push_front( const T & value )`: adds `value` to the front of the list.
- `void push_back( const T & value )`: adds `value` to the end of the list.
- `void pop_back()`: removes the object at the end of the list.
- `void pop_front()`: removes the object at the front of the list.
- `const T & back() const`: returns the object at the end of the list.
- `const T & front() const`: returns the object at the beginning of the list.
- `void assign( sizet_type count, const T & value )`: replaces the content of the list with `count` copies of `value`.

2.3 Operations exclusive to dynamic array implementation

Because array implementations allows for efficient indexing, there are some operations that are exclusive to them[1].

- `T & operator[]( size_type pos)`: returns the object at the index `pos` in the array, with no bounds-checking.
- `T & at ( size_type pos )()`: returns the object at the index `pos` in the array, with bounds-checking. If `pos` is not within the range of the list, an exception of type `std::out_of_range` is thrown.
- `size_type capacity() const`: return the internal storage capacity of the array.
- `void reserve( size_type new_cap )`: increase the storage capacity of the array to a value that's is greater or equal to `new_cap`. If `new_cap` is greater than the current `capacity()`, new storage is allocated, otherwise the method does nothing. If `new_cap` is greater than the current capacity, all iterators and references, including the past-the-end iterator/index, are invalidated. Otherwise, no iterators or references are invalidated. This function also preserve the data elements already stored in the list, as well as their original order.
- `shrink_to_fit()`: Requests the removal of unused capacity. It is a non-binding request to reduce `capacity()` to `size()`. It depends on the implementation if the request is fulfilled.
  All iterators, including the past-the-end iterator, are potentially invalidated.

2.4 Operator overloading — non-member functions

Lastly, we need to provide a couple of binary operator on lists. They are:

[1] Again, these methods might also be implemented in a linked list, however they would be considered very inefficient if compared with their counterpart in a array-based implementation of a list.

- `bool operator==( const vector& lhs, const vector& rhs );`: Checks if the contents of `lhs` and `rhs` are equal, that is, whether `lhs.size() == rhs.size()` and each element in `lhs` compares equal with the element in `rhs` at the same position.
- `bool operator!=( const vector& lhs, const vector& rhs ;`: Similar to the previous operator, but the opposite result.

In both cases, the type `T` stored in the list must be `EqualyComparable`, in other words, it must support the `operator==()`.

Note that these **are not methods nor a friend function**, but some plain-old regular functions. Therefore, the implementation of these functions must rely only on public methods provided by each class. This is an alternative way of providing operator overloading for classes. The `lhs` and `rhs` represent, respectively, the *left-hand-side* and the *right-hand-side* elements of a comparison operation.

3 Iterators

There are other operations common to all implementations of a list. These operations require the ability to insert/remove elements in the middle of the list. For that, we require the notion of *position*, which is implemented in STL as a nested type `iterator`.

Iterators may be defined informally as a class that encapsulate a pointer to some element within the list. This is an object oriented way of providing some degree of access to the list without exposing the internal components of the class.

Before delving into more details on how to implement iterators, let us consider a simple example of iterators in use. Suppose we declared a doubly linked list of integers and wish to create an iterator variable `it` that points to the beginning of our list. Next, we wish to display in the terminal the content of the list separated by a blank space, and delete the first three elements of the list, after that. The code to achieve this would be something like:

```
1  std::list<int> myList;  // List declaration.
2  // Here goes some code to fill in the list with elements.
3  ...
4  // Let us instantiate an iterator that points to the beginning of the list.
5  std::list<int>::iterator it = myList.begin();
6  // Run through the list, accessing each element.
7  for ( ; it != myList.end(); ++it )
8      std::cout << *it << " ";
9  // Removing the first 3 elements from the list.
10 myList.erase( myList.begin(), myList.begin()+3 );
```

By examining this simple example we identify three issues we need to address if we wish to implement our own iterator object: how one gets an iterator; what operations the iterators should support, and; which list methods require iterators as parameters. These issues are discussed next and their corresponding methods depend upon the definition of the type `iterator` and `const_iterator`,

which are friend classes declared inside each list class, as demonstrated in Code 1 for the `Forward_list` class.

3.1 Getting an iterator

- `iterator begin()`: returns an iterator pointing to the first item in the list (see Figure 1).
- `const_iterator cbegin() const`: returns a constant iterator pointing to the first item in the list.
- `iterator end()`: returns an iterator pointing to the end mark in the list, i.e. the position just after the last element of the list.
- `const_iterator cend() const`: returns a constant iterator pointing to the end mark in the list, i.e. the position just after the last element of the list.

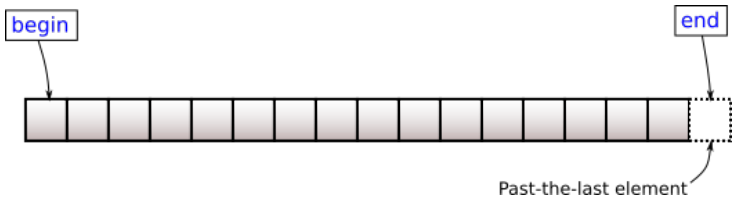

Figura 1: Visual interpretation of iterators in a container.
Source: http://upload.cppreference.com/mwiki/images/1/1b/range-begin-end.svg

The constant versions of the iterator are necessary whenever we need to use an iterator inside a `const` method, for instance. The `end()` method may seem a bit unusual, since it returns a pointer "out of bounds". However, this approach supports typical programming idiom to iterate along a container, as seen in the previous code example.

3.2 Iterator operations

- `operator++()`: advances iterator to the next location within the vector. We should provide both prefix and posfix form, or `++it` and `it++`.
- `operator*()` as in `*it`: return a reference to the object located at the position pointed by the iterator. The reference may or may not be modifiable.
- `operator-()` as in `it1-it2`: return the difference between two iterators.
- `friend operator+(int n, iterator it)` as in `2+it`: return a iterator pointing to the `n`-th successor in the vector from `it`.
- `friend operator+(iterator it, int n)` as in `it+2`: return a iterator pointing to the `n`-th successor in the vector from `it`.
- `friend operator-(int n, iterator it)` as in `2-it`: return a iterator pointing to the `n`-th predecessor in the vector from `it`.
- `friend operator-(iterator it, int n)` as in `it-2`: return a iterator pointing to the `n`-th predecessor in the vector from `it`.

- `operator->()` as in `it->field`: return a pointer to the location in the vector the `it` points to.
- `operator==()` as in `it1 == it2`: returns `true` if both iterators refer to the same location within the vector, and `false` otherwise.
- `operator!=()` as in `it1 != it2`: returns `true` if both iterators refer to a different location within the vector, and `false` otherwise.

Notice how theses operations involving iterators are (intentionally) very similar to the way we manipulate regular pointers to access, say, elements in an array.

3.3 List container operations that require iterators

- `iterator insert( iterator pos, const T & value )`: adds `value` into the list *before* the position given by the iterator `pos`. The method returns an iterator to the position of the inserted item.
- `template < typename InItr>`
  `iterator insert( iterator pos, InItr first, InItr last )`: inserts elements from the range `[first; last)` before `pos`.
- `iterator insert( const_iterator pos, std::initializer_list<T> ilist )`: inserts elements from the initializer list `ilist` before `pos`. Initializer list supports the user of insert as in `myList.insert( pos, {1, 2, 3, 4} )`, which would insert the elements 1, 2, 3, and 4 in the list before `pos`, assuming that `myList` is a list of `int`.
- `iterator erase( iterator pos )`: removes the object at position `pos`. The method returns an iterator to the element that follows `pos` before the call. This operation *invalidates* `pos`, since the item it pointed to was removed from the list.
- `iterator erase( iterator first, iterator last )`: removes elements in the range `[first; last)`. The entire list may be erased by calling `a.erase(a.begin(), a.end());`
- `void assign( size_type count, const T& value );`: Replaces the contents with `count` copies of value `value`.
- `template < typename InItr>`
  `void assign( InItr first, InItr last )`: replaces the contents of the list with copies of the elements in the range `[first; last)`.
- `void assign( std::initializer_list<T> ilist )`: replaces the contents of the list with the elements from the initializer list `ilist`.
  We may call, for instance, `myList.assign( {1, 2, 3, 4} )`, to replace the elements of the list with the elements 1, 2, 3, and 4, assuming that `myList` is a list of `int`.

For each operation described above, you must provide a const version, which means replacing `iterator` by `const_iterator`. For the particular case of list ADT being implemented with array, you should notice that insertion operations may (1) cause reallocation if the new `size()` is greater than the `capacity()`, and (2) make all previous iterators and references invalid.

4 Implementation

Your mission, should you choose to accept it, is to implement a class called `vector` that follows the list ADT design described previously, and store its elements as a *dynamic array*.

Try to use `std::unique_ptr` so that the release of resources is automatically taken care of. This behavior is called *Resource Acquisition Is Initialization* (RAII), which basically binds the life cycle of a resources (in the `vector`'s case, the dynamic array) to the lifetime of an object (in the case, the `vector` itself).

In practical terms, this means that when a the life cycle of a `vector` object ends, the internal dynamic array is automatically deleted, without the need to put the `delete []` command in the destructor `~vector()`.

You should declare a nested `const_iterator` class (so it has access to the dynamic array) that basically is friend of the `vector` class and holds a private regular pointer. The same goes to the class `iterator`, which means it is also a nested class.

Every time the current storage capacity limit of the array is reached because of an insertion operation, the class should **double** its storage capacity, requesting a new dynamic memory block, and copying the original values to the new block. This doubling-memory-and-copying-data action should go completely unnoticed to the client code.

For debugging purposes, you may peek inside the class during your tests using the method `const T* data() const`, which should return a raw pointer to the underlying array. Because you are using a `std::unique_ptr` it is safe to provide such a pointer because the client will not have permission to delete the data, if she wanted to, since the ownership of the array (i.e. the memory block) belongs to the object.

The Codes 1 to 3 present an overview of the declaration of all suggested classes and some of their corresponding methods. All classes may be declared in the same file, named `vector.h`.

Driver code

Your last task is to design a driver program to thoroughly test each method of the class you developed. Try to be creative and comprehensive in your tests, relying on `assert()` calls to make sure everything goes smoothly. Look for corner cases and identify them in your tests.

5 Project Evaluation

You should hand in a complete program, without compiling errors, tested and fully documented. The assignment will be credit according to the following criteria:-

1. Correct implementation of `vector`:
   (a) Special members (18 credits);
      i. Regular constructor (3 credits);
      ii. Destructor (3 credits);

iii. Copy constructor (3 credits);
iv. Constructor from range (3 credits);
v. Constructor from initialize list (3 credits);
vi. Assignment operator (3 credits);

(b) Iterator methods (4 credits);:
i. `begin()` (1 credits);
ii. `end()` (1 credits);
iii. `cbegin()` (1 credits);
iv. `cend()` (1 credits);

(c) Capacity methods (3 credits);
i. `empty()` (1 credits);
ii. `size()` (1 credits);
iii. `capacity()` (1 credits);

(d) Modifiers methods (47 credits);
i. `clear()` (1 credits);
ii. `push_front()` (2 credits);
iii. `push_back()` (2 credits);
iv. `pop_front()` (2 credits);
v. `pop_back()` (2 credits);
vi. `insert()` $\times 3$ (12 credits);
vii. `reserve()` (3 credits);
viii. `shrink_to_fit()` (3 credits);
ix. `assign()` $\times 3$ (12 credits);
x. `erase()` $\times 2$ (8 credits);

(e) Element access methods (6 credits); and,
i. `front()` (1 credits);
ii. `back()` (1 credits);
iii. `operator[]()` $\times 2$ (2 credits);
iv. `at()` $\times 2$ (2 credits);

(f) Operators (2 credits).
i. `operator==()` (1 credits);
ii. `operator!=()` (1 credits);

2. Iterator classes (20 credits);

The following situations may *take credits out* of your assignment, if they happen during the evaluation process:-

- Compiling and/or run time errors (up to -20 credits)
- Missing code documentation in Doxygen style (up to -10 credits)
- Memory leak (up to -10 credits)

- Missing README file (up to -20 credits).

The README file (Markdown file format recommended here) should contain a brief description of the project, and how to run it. It also should describe possible errors, limitations, or issues found. Do not forget to include the author(s) name(s)!

Good Programming Practices

During the development process of your assignment, it is strongly recommend to use the following tools:-

- Doxygen: professional code documentation;
- Git: version control system;
- Valgrind: tracks memory leaks, among other features;
- gdb: debugging tool, and;
- Makefile: helps building and managing your programming projects.

Try to organize you code in several folders, such as `src` (for `.cpp` files), `include` (for header files `.h`, and `.inl`), `bin` (for `.o` and executable files) and `data` (for storing input files).

6 Authorship and Collaboration Policy

This is a pair assignment. You may work in a pair or alone. If you work as a pair, comment both members' names atop every code file, and try to balance evenly the workload. Only one of you should submit the program via Sigaa.

Any team may be called for an interview. The purpose of the interview is twofold: to confirm the authorship of the assignment and to identify the workload assign to each member. During the interview, any team member should be capable of explaining any piece of code, even if he or she has not written that particular piece of code. After the interview, the assignment's credits may be re-distributed to better reflect the true contribution of each team member.

The cooperation among students is strongly encouraged. It is accepted the open discussion of ideas or development strategies. Notice, however, that this type of interaction should not be understood as a free permission to copy and use somebody else's code. This is may be interpreted as plagiarism.

Any two (or more) programs deemed as plagiarism will automatically receive **zero** credits, regardless of the real authorship of the programs involved in the case. If your project uses a (small) piece of code from someone else's, please provide proper acknowledgment in the README file.

7 Work Submission

Only one team member should submit a single zip file containing the entire project. This should be done only via the proper link in the Sigaa's virtual class.

◀ The End ▶

Code 1 Partial listing of the class `sc::vector`. This code has references to Code 3, which contains the declaration of the iterator class.

```
1  template <typename T>
2  class vector {
3    public:
4      using size_type = unsigned long; //!< The size type.
5      using value_type = T;            //!< The value type.
6      using pointer = value_type*;     //!< Pointer to a value stored in the container.
7      using reference = value_type&;   //!< Reference to a value stored in the container.
8      using const_reference = const value_type&; //!< Const reference to a value stored in the co
9      using iterator = MyIterator< T > // See Code 3
10     using const_iterator = MyIterator< const T > // See Code 3
11
12     //=== [I] SPECIAL MEMBERS
13     explicit vector( size_type = 0 );
14     virtual ~vector( void );
15     vector( const vector & );
16     vector( std::initializer_list<T> );
17     vector( vector && );
18     template < typename InputItr >
19     vector( InputItr, InputItr );
20     vector& operator=( const vector& );
21     vector& operator=( vector && );
22
23     //=== [II] ITERATORS
24     iterator begin( void );
25     iterator end( void );
26     const_iterator cbegin( void ) const;
27     const_iterator cend( void ) const;
28
29     // [III] Capacity
30     size_type size( void ) const;
31     size_type capacity( void ) const;
32     bool empty( void ) const;
33
34     // [IV] Modifiers
35     void clear( void );
36     void push_front( const_reference );
37     void push_back( const_reference );
38     void pop_back( void );
39     void pop_front( void );
40     iterator insert( iterator, const_reference );
41     template < typename InputItr >
42     iterator insert( iterator , InputItr, InputItr );
43     iterator insert( iterator , const std::initializer_list< value_type >& );
44     void reserve( size_type );
45     void shrink_to_fit( void );
```

Code 2 Partial listing of the class `sc::vector` (cont.). This code has references to Code 3, which contains the declaration of the iterator class.

```
46      void assign( size_type, const_reference );
47      void assign( const std::initializer_list<T>& );
48      template < typename InputItr >
49      void assign( InputItr, InputItr );
50
51      iterator erase( iterator, iterator );
52      iterator erase( iterator );
53
54      // [V] Element access
55      const_reference back( void ) const;
56      reference back( void );
57      const_reference front( void ) const;
58      reference front( void );
59      const_reference operator[]( size_type ) const;
60      reference operator[]( size_type );
61      const_reference at( size_type ) const;
62      reference at( size_type );
63      pointer data( void );
64      const_reference data( void ) const;
65
66      // [VII] Friend functions.
67      friend std::ostream& operator<<( std::ostream&, const vector<T>& );
68      friend void swap( vector<T>&, vector<T>& );
69
70  private:
71      size_type m_end;               //!< Current list size (or index past-last valid element).
72      size_type m_capacity;           //!< List's storage capacity.
73      //std::unique_ptr<T[]> m_storage; //!< Data storage area for the dynamic array.
74      T *m_storage; //!< Data storage area for the dynamic array.
75  };
```

Code 3 Partial listing of classes `MyIterator`.

```
template <typename T>
class MyIterator {
  public:
    // Below we have the iterator_traits common interface
    /// Difference type used to calculated distance between iterators.
    typedef std::ptrdiff_t difference_type;
    typedef T value_type;           //!< Value type the iterator points to.
    typedef T* pointer;             //!< Pointer to the value type.
    typedef T& reference;           //!< Reference to the value type.
    typedef std::bidirectional_iterator_tag iterator_category; //!< Iterator category.
    MyIterator( );
    MyIterator& operator=( const MyIterator& ) = default;
    MyIterator( const MyIterator& ) = default;
    reference operator* ( ) const;
    pointer operator->( void ) const {   assert( m_ptr != nullptr ); return m_ptr; }
    MyIterator& operator++ ( );   // ++it;
    MyIterator operator++ ( int ); // it++;
    MyIterator& operator-- ( );   // --it;
    MyIterator operator-- ( int ); // it--;
    friend MyIterator operator+( difference_type, MyIterator );
    friend MyIterator operator+( MyIterator, difference_type  );
    friend MyIterator operator-( difference_type, MyIterator  );
    friend MyIterator operator-( MyIterator, difference_type );
    bool operator== ( const MyIterator& ) const;
    bool operator!= ( const MyIterator& ) const;
  private:
    T *current;
};
```