Universidade Federal do Rio Grande do Norte
Instituto Metrópole Digital

Programming Language I • IMD0030
◁ Implementing the List Abstract Data Type (ADT) ▷
26 de abril de 2016

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 three different forms of organizing the data inside a list, using dynamic array, singly linked list, and doubly linked list.

We begin by introducing basic definition of terms, properties and operations. Next we provide details on the three types of underlying data structure we may use to implement a list ADT.

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 strutural 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 $A[k-1]$ preceeds $A[k]$.

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 or 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 spaece for the new element.

In this document we discuss how to implement a list ADT based on three types of underlying data structures: *dynamic arrays*, *singly linked lists*, and *doubly linked list*. These three versions of a list are equivalent, respectivelly, to the `std::vector`, the `std::forward_list`, and the `std::list` classes of the STL library.

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 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_back( const T & x )`: adds `x` to the end of the list.
- `void pop_back()`: removes the object at the end 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( const T & x )`: replaces the content of the list with copies of value `x`.

Notice that all references to either a return type or variable declation related to the size of a list are defined as `size_type`. This is basically an alias to some 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 long`.

2.2 Operations exclusive to linked list implementation

Both singly and doubly linked lists support efficient (constant time) changes at the front of the list. Therefore, these methods below are available only in those classes[1]

- `void push_front( const T & x )`: adds `x` to the front of the list.
- `void pop_front()`: removes the object at the front of the list.

2.3 Operations exclusive to dynamic array implementation

Because array implementations allows for efficient indexing, there are some operations that are exclusive to them[2].

- `T & operator[]( size_type idx)`: returns the object at the index `idx` in the array, with no bounds-checking.
- `Object & at ( size_type idx )()`: returns the object at the index `idx` in the array, with bounds-checking.
- `size_type capacity() const`: return the internal storage capacity of the array.
- `void reserve( size_type new_capacity )`: sets the new storage capacity of the array, in case the `new_capacity` is greater than the current capacity. This function also keeps the data already stored in the list.

2.4 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 informmaly 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.

[1] Of course, these methods might be implemented in an array-based list, but they would not be as efficient as the same methods present in a linked list version.

[2] Again, these methods might also be implemented in a linked list, however they would be considered very inneficient if compared with their conterpart in a array-based implementarion of a list.

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 itetaror varible `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 achive 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 addres 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.

2.4.1 Getting an iterator

- `iterator begin()`: returns an iterator pointing to the first item in the list (see Figure 1).
- `const_iterator begin() 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 end() 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.

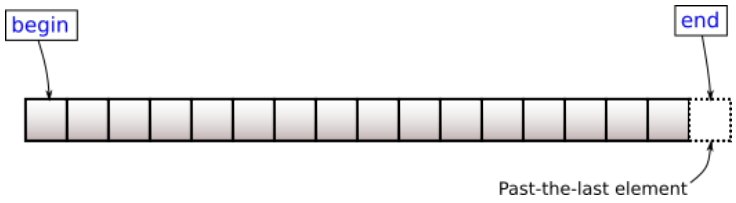


Figura 1: Visual interpretation of iterators in a container.
Source: http://upload.cppreference.com/mwiki/images/1/1b/range-begin-end.svg

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

2.4.2 Iterator operations

- `operator++()`: advances iterator to the next location within the list. We should provide both prefix and posfix form, or `++it` and `it++`.
- `operator*()` as in `*it`: return a reference to the objected located at the position pointed by the iterator. The referance may or may not be modifiable.
- `operator==()` as in `it1 == it2`: returns `true` if both iterators refer to the same location within the list, and `false` otherwise.
- `operator!=()` as in `it1 != it2`: returns `true` if both iterators refer to a different location within the list, and `false` otherwise.

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

2.4.3 List container operations that require iterators

- `iterator insert( iterator pos, const T & x )`: adds `x` into the list *before* the position given by the iterator `pos`. The method returns an interator 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 inser 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());`.
- `template < typename InItr>`
  `void assign( InItr first, InItr last )`: replaces the contenst 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` before `pos`. We may call, for instance, `myList.assign( {1, 2, 3, 4} )`, to replace the first four elements of the list with the elements 1, 2, 3, and 4, assuming that `myList` is a list of `int`.

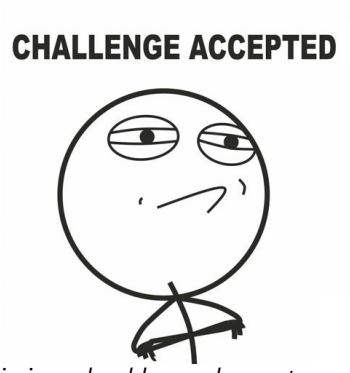
For each operation described abover, you must provide a const version, which means replacing `iterator` by `const_iterator`. For the particular case of list ADT being implemented with array,

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.

3 Implementation

Your mission, should you choose to accept it, is to implement the list ADT as a series of three classes, mimicking the behavior of similar classes in the STL library. They are described next.

3.1 Description of `Vector` class



"— Your mission, should you choose to accept it, is to implement the list ADT."

You should implement a class called `Vector` (with uppercase 'V') 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 behaviour 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 pratical 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) tha basically is friend of the `Vector` class and holds a private regular pointer. The class `iterator` should extend the class `const_iterator` through public inheritance.

Every time the current storage capacity 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.

In all operations that remove an element of the list (e.g. `pop_back`, `erase, clear`) remenber to call the `T`'s destructor, as in `(&a)-> T()`, where `a` is the object being removed, so that any resources `a` might have are correcly released.

For debugging purposes, you may peek inside the class during your tests using the method `T* data()`, 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.

3.2 Description of `Forward_List` class

You should implement a class called `Forward_list` (with uppercase 'F') that follows the list ADT design and stores its elements in a *singly linked list*. You are encouraged to create `head` and `tail` extra *nodes* (not pointers!) to facilitate some operations on the list.

Your class should mimic the behaviour of the class `std::list_forward`. Try to look the documentation of that class to see the example code, which might clear up some issued on how to use certain list methods.

Notice, however, that the methods `erase` and `insert` have their name and behaviour changed, respectivelly, to `erase_after` and `insert_after`, as the following:

- `iterator insert_after( const_iterator pos, ... )`: inserts values(s) after the elemented pointed by `pos`.
- `iterator erase_after( const_iterator pos )`: removes the element following `pos`. The method returns an iterator to the element that follows `pos` before the call.
- `iterator erase_after( iterator first, iterator last )`: removes elements in the range `(first; last)`.

To support these modified versions of the tradicional insert/erase operations, this class should also provide extra function that returns iterators. They are:

- `before_begin()`: returns an interator to the element before the first element of the list. This element act as a placeholder. Trying to use this element in any way results in undefined behaviour. The only purpose of this special iterator is to provide support to methods such as `insert_after` and `erase_after`. For instance, if we neede to remove the first element of the list `myList`, we would call `myList.erase_after( myList.before_begin() )`.
- `cbefore_begin()`: same function as the method above, the only difference is that it returns a `const_iterator`.

3.3 Description of `List` class

You should implement a class called `List` (with uppercase 'L') that follows the list ADT design and stores its elements in a *doubly linked list*. You are encouraged to create `head` and `tail` extra *nodes* (not pointers!) to facilitate some operations on the list.

Your class should mimic the behaviour of the class `std::list`. Try to look the documentation of that class to see the example code, which might clear up some issued on how to use certain list methods.

3.4 Driver code

Your last task is to design a driver program to thorougly test each method for all three classes you developed. Try to be creaive and compreheensive in your tests, relying on `assert()` calls to make sure everything goes smoothly. Look for corner cases and identify them in your tests.

4 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` (25 credits);
2. Correct implementation of `Forward_list` (25 credits);
3. Correct implementation of `List` (25 credits);
4. A driver program that compreensivelly tests the three classes you should implement (25 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 game, 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`), `bin` (for `.o` and executable files) and `data` (for storing input files).

5 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.

6 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 ▶