Universidade Federal do Rio Grande do Norte

Instituto Metrópole Digital

Basic Data Structure I • DIM0119

◁ Implementing the List Abstract Data Type ▷

10 de maio de 2018

## Overview

This document describes the implementation of the *list Abstract Data Type* (ADT) based on a doubly linked list.

We begin by introducing basic definition of terms, properties and operations. Next we provide details on the underlying data structure you should 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 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 *doubly linked list*. This version of a list is equivalent to 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    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 your `list` class. In these specifications consider that the `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 program-ming 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`.

| | |
|---|---|
| `list( );` | (1) |
| `explicit list( size_type count );` | (2) |
| `template< typename InputIt >`<br>`list( InputIt first, InputIt last );` | (3) |
| `list( const list& other );` | (4) |
| `list( std::initializer_list<T> ilist );` | (5) |
| `~list( );` | (6) |
| `list& operator=( const list& other );` | (7) |
| `list& 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 de-allocated. 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.

(8) Replaces the contents with those identified by initializer list `ilist`.

All three `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( const T & value )` : replaces the content of the list with copies of value `value` .

## 2.3   Operator overloading — non-member functions

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

- `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.
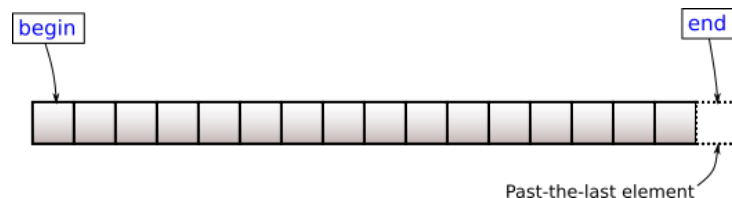
# 3   Iterators

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. In this section we describe methods that require an iteration to work.

## 3.1   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, 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 list. 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` : 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 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` .

# 4    Implementation

You should implement a class called `ls::list` 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 allow for constant time complexity of some operations done on either end of the list.
See Figure 2 for a abstract representation of the list with its two head and tail nodes (doubly linked
version).
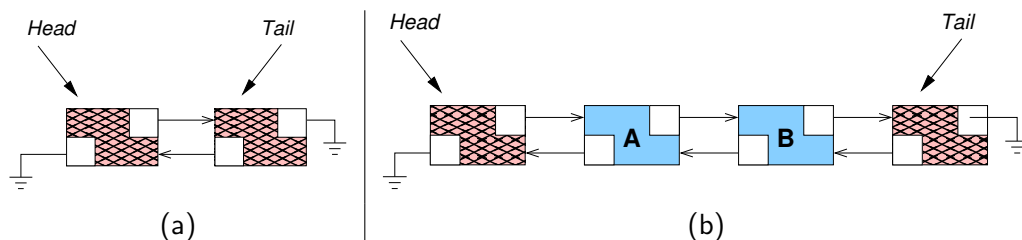


**Figura 2:** Abstract representation of a doubly linked list with a *head*, and *tail* nodes in (a) empty state, and (b)
with two elements stored.

One possible implementation for the `list` may comprehend the following four classes:

- The main `list` class, which should create the tail and head nodes, keep track of the number of elements in the list, create/delete nodes as the list receives/looses elements, and provide a series of public methods the client code should invoke to operate on or access the list.

- The structure `Node`, which is a nested structure that describes the memory layout of a linked list node; basically it is a struct with two fields, the data content and a pointer to the next node in the list. Because the `Node` is a nested structure inside `list`, its fields may be accessed directly from within every `list`'s methods by using the '`.`' operator.

- The `const_iterator` class, which is class that represents a const iterator that may point to whichever element in the list. The class `const_iterator` stores a raw pointer to the "current" node, and provide the basic iterator operations, such as `=`, `==`, `!=` e `++`.

- The `iterator` class, which is class similar to the `const_iterator` class. This class provide the same functionality as the `const_iterator`, with the exception of `operator*()`, which returns a *regular reference* to a list item, rather than a *constant reference*. Notice that an `iterator` may be used in any method that requires a `const_iterator`, but the opposite is not true. In other words, the `iterator` *IS-A*[1] `const_iterator`.

The Codes 1 to 3 present an overview of the declaration of all suggested classes and some of their corresponding methods. Note that these code listing are only suggestions and they may be changed/adapted as long as all the requested methods' signature are not altered. All classes may be declared in the same file, named `list.h`.

## 4.1 Driver code

Your last task is to design a driver program to thoroughly test each method for all three classes 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 special members (Section 2.1)(16 credits);

   (a) Regular constructor (2 credits);
   (b) Constructor (size) (2 credits);
   (c) Destructor (2 credits);
   (d) Copy constructor (2 credits);
   (e) Constructor from range (2 credits);
   (f) Constructor from initialize list (2 credits);
   (g) Assignment operator ×2 (4 credits);

---

[1]Object-oriented programming terminology for the concept of *inheritance*.

2. Correct implementation of the get iterators (Section 3.1) (4 credits);

    (a) `begin()` (1 credits);
    (b) `end()` (1 credits);
    (c) `cbegin()` (1 credits);
    (d) `cend()` (1 credits);

3. Correct implementation of common operations (Section 2.2) (20 credits);

    (a) `empty()` (1 credits);
    (b) `size()` (1 credits);
    (c) `clear()` (2 credits);
    (d) `front()` (2 credits);
    (e) `back()` (2 credits);
    (f) `push_front()` (2 credits);
    (g) `push_back()` (2 credits);
    (h) `pop_front()` (2 credits);
    (i) `pop_back()` (2 credits);
    (j) `assign()` ×2 (4 credits);

4. Correct implementation of operator overloading (Section 2.3) (2 credits);

    (a) `operator==()` (1 credits);
    (b) `operator!=()` (1 credits);

5. Correct implementation of methods that require iterator (Section 3.3) (21 credits);

    (a) `insert()` ×3 (9 credits);
    (b) `find()` ×2 (6 credits);
    (c) `erase()` ×2 (6 credits);

6. Correct implementation of the iterators (Section 3) (15 credits);

7. A driver program that comprehensively tests the three classes you should implement (22 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 `ls::list`. This code has references to Codes 2, and 3, which contains the declaration of related classes.

```cpp
1  template <typename T>
2  class list {
3    private:
4      struct Node { See Code 2 };
5
6    public:
7      class const_iterator { See Code 3 };
8      class iterator : public const_iterator { See Code 3 };
9
10     // [I] SPECIAL MEMBERS
11     list( );
12     ~list( );
13     list( const list & );
14     list & operator= ( const list & );
15
16     // [II] ITERATORS
17     iterator begin( );
18     const_iterator cbegin( ) const;
19     iterator end( );
20     const_iterator cend( ) const;
21     // [III] Capacity
22     int size( ) const;
23     bool empty( ) const;
24     // [IV] Modifiers
25     void clear( );
26     T & front( );
27     const T & front( ) const;
28     T & back( );
29     const T & back( ) const;
30     void push_front( const T & value );
31     void push_back( const T & value );
32     void pop_front( );
33     void pop_back( );
34     void assign(const T& value );
35     // [IV-a] Modifiers with iterators
36     template < class InItr >
37     void assign( InItr first, InItr last );
38     void assign( std::initializer_list<T> ilist );
39     iterator insert( const_iterator itr, const T & value );
40     iterator insert( const_iterator pos, std::initializer_list<T> ilist );
41     iterator erase( const_iterator itr );
42     iterator erase( const_iterator first, const_iterator last );
43     const_iterator find( const T & value ) const;
44
45   private:
46     int    m_size;
47     Node *m_head;
48     Node *m_tail;
49 };
```

**Code 2** Partial listing of the structure `Node`, which is part of the class `Forward_list` (see Code 1).

```
1  struct Node {
2      T   data;      //<! Data field
3      Node  *prev;  //<! Pointer to the previous node in the list.
4      Node  *next;  //<! Pointer to the next node in the list.
5      //<! Basic constructor.
6      Node( const T & d = T( ), Node * p = nullptr, Node * n = nullptr )
7        : data( d ), prev(p), next( n ) { /* Empty */ }
8  };
```

**Code 3** Partial listing of classes `const_iterator`, and `iterator`, part of the class `Forward_list` (see Code 1).

```
1  class const_iterator {
2    public:
3      const_iterator( );
4      const Object & operator* ( ) const;
5      const_iterator & operator++ ( );   // ++it;
6      const_iterator operator++ ( int ); // it++;
7      const_iterator & operator-- ( );   // --it;
8      const_iterator operator-- ( int ); // it--;
9      bool operator== ( const const_iterator & rhs ) const;
10     bool operator!= ( const const_iterator & rhs ) const;
11
12   protected:
13     Node *current;
14     const_iterator( Node * p ) : current( p );
15     friend class List<Object>;
16 };
17 class iterator : public const_iterator {
18   public:
19     iterator( ) : const_iterator() { /* Empty */ }
20     const Object & operator* ( ) const;
21     Object & operator* ( );
22
23     iterator & operator++ ( );
24     iterator operator++ ( int );
25     iterator & operator-- ( );
26     iterator operator-- ( int );
27
28   protected:
29     iterator( Node *p ) : const_iterator( p );
30     friend class List<Object>;
31 };
```