Lab: Linked Lists as Containers
CSC 161, “Imperative Problem Solving and Data Structures”
Department of Computer Science · Grinnell College
April 15, 2016

Background
From the “Scheme-Like Lists” lab, we have learned that C makes it easy to use
constructions that change the internal structure of linked lists, sometimes leading to side
effects that are surprising and confusing to programmers accustomed to thinking of lists
as values, as Scheme encourages them to do. In C, it is much easier and more common
to think of a list as a container, with an identity that is separate from and independent
of the elements of the list — an identity that survives even as the internet structure of the
list changes through internal insertions and deletions of elements or possibly in even more
radical ways.

In this lab, we’ll look at a way to express this idea more clearly in C and to combine
it with a convention for storage allocation and deallocation that will block the confusing
side effects and make it easier to use linked-list structures in the C style.

The key idea is to enclose the list in a structure, as its only field:

typedef struct node {
    int datum;
    struct node *next;
} *list;

typedef struct {
    list front;
} enclosed_list;

The enclosed_list structure constitutes the “identity” of the list. It persists over
time and survives any changes in the content or structure of the list it contains. The front
field of that structure holds the underlying Scheme-like list. (It will be NULL if that list is
empty; otherwise front will point to the struct node that contains the first element of
the list.)

The convention about storage allocation and deallocation is that each enclosed_list
will have exclusive control over the storage for the list it contains. We’ll promise never
to use any of the struct node blocks from that list inside any other enclosed_list, and
if we do any destructive operations on the list in the front field of an enclosed_list,
we’ll make sure that the result is still placed inside that same enclosed_list. Whatever
happens in an enclosed_list stays in that enclosed_list and cannot affect any other
enclosed_list.
Ella Nicolson will take the place of anyone who is absent.

Exercises

First, let’s figure out how to build up a list of values inside an enclosed list. In general, we’ll want the parameters of the functions that we define to operate on enclosed lists to be pointers to the actual enclosed list structures (typically, the addresses of those structures) because we don’t want the function-calling mechanism to waste time copying the structures onto the run-time stack, and because we want any side effects that we perform inside those functions to occur inside the original structure that the caller provides rather than in a local copy.

Exercise 0000: Write a C program in which you declare a variable sample of type enclosed_list and initialize the front field of sample to be an empty list.

Exercise 0001: Define a C function that takes a pointer to an enclosed_list as its argument and writes out the contents of the list that is currently stored in its front field of the enclosed_list at the other end of that pointer. The display can use Scheme-like syntax, as in the display function that you wrote for exercise 011 in the “Introduction to Lists” lab.

Exercise 0010: Define and test a C function prepend that takes two arguments, an int and a pointer to an enclosed_list structure, and replaces the list in the front field of that enclosed_list structure with the result of adding a new node containing the given int at the beginning of the list that was formerly in that field. (You can use the cons function from the “Introduction to Lists” lab to do the actual storage allocation, but make sure that you assign the result of the call to cons to the front field of the enclosed_list structure before you finish.) The prepend function should return void. It is called only for its side effect on the front field of the enclosed_list structure.

Since we’re forbidding ourselves to use the same nodes inside different enclosed_list structures, it will often be convenient to build a duplicate of a given list, allocating completely separate storage blocks for all of the components of the copy.

Exercise 0011: Write and test a C function list_copy that takes a list as argument and returns a list containing the same int values, but in completely separate storage blocks obtained through new calls to malloc or calloc. None of the pointers in any of the components of the copy should be equal to any of the pointers in components of the original (except for the NULL pointer that terminates both).

Exercise 0100: Write and test a C function enclosed_list_copy that takes as arguments two pointers (target and source) to enclosed_list structures, makes a copy of the list in the front field of the enclosed_list to which source points, and places the copy in
the `front` field of the `enclosed_list` to which `target` points. Again, `enclosed_list_copy` should return `void`—it’s called only for its side effect. You may assume, as a precondition of `enclosed_list_copy`, that `target` initially points to an empty `enclosed_list` structure—that is, to a structure whose `front` field is `NULL`.

Exercise 0101: Write and test a C function `enclosed_list_free` that takes a pointer to an `enclosed_list` structure as argument and deallocates the storage for all of the nodes in the list stored in the `front` field of that `enclosed_list` structure. After freeing all of those nodes, your function should assign `NULL` to the `front` field, so that the `enclosed_list` will be recognized as empty and does not have any “dangling pointers” to storage locations that have been freed (and possibly recycled). As in the preceding exercises, `enclosed_list_free` should return `void`.

Exercise 0110: Write and test a C function `enclosed_list_reverse` that takes a pointer to an `enclosed_list` structure as argument and destructively reverses the list in the `front` field of that `enclosed_list` structure, replacing the old value of `front` with a pointer to the first node in the reversed list. (You can use `d_reverse_list` to rearrange the pointers in the nodes of the list.) `enclosed_list_reverse` is invoked only for the side effect that it has on the `enclosed_list` and should return `void` once that side effect has been accomplished.

Exercise 0111: Write and test a C function `enclosed_list_concatenate` that takes as arguments three pointers (`target`, `left`, and `right`) to `enclosed_list` structures and stores in the `front` field of the structure to which `target` points a list comprising all of the elements of the list in the `front` field of the structure to which `left` points followed by all of the elements of the list in the `front` field of the structure to which `right` points. You may assume, as a precondition, that the structure to which `target` points is empty.

In your solution, you should observe the convention that none of the storage allocated within either of the source structures can be used in the construction of the list that will be stored inside the structure to which `target` points. This means that you can’t begin by writing ‘`target->front = right->front`’ and then adding the elements of the left list one at a time. You need freshly allocated nodes for all of the elements of both source lists. (Hint: You can, however, start with an appropriate call to `enclosed_list_copy`.)

Exercise 1000: Write and test a C function `postpend` that takes two arguments, an `int` and a pointer to an `enclosed_list` structure, and replaces the list in the `front` field of that `enclosed_list` structure with the result of destructively adding a new node containing the given `int` at the end of the list that was formerly in that field. (Hint: The value of `front`, which is a pointer, won’t actually change unless it was `NULL` to begin with. If the list was initially non-empty, the change in the list structure will occur in the next field of the list’s last node—a pointer to the node containing the new value will overwrite the `NULL` that formerly terminated the list.)

Exercise 1001: Write and test a C function `find_node` that takes two arguments, an `int` and a pointer to an `enclosed_list` structure, that traverses the list stored in the `front` field of the structure to which the given pointer points, looking for a node containing the given `int` in its `datum` field and returning a pointer to the first such node it finds, or a `NULL` pointer if it reaches the end of the list without finding such node.

Exercise 1010: Write and test a C function `enclosed_list_ref` that takes two arguments, a pointer to an `enclosed_list` structure and a value of type `size_t` (as defined in the standard `stdint.h` header file), and returns the `int` value from the (zero-based) position specified by the second argument in the list stored in the `front` field of the `enclosed_list` structure to which the first argument points. It is a precondition of this
function that the second argument is strictly less than the length of that list (i.e., the number of nodes it contains).

**Exercise 1011:** Write and test a C function `enclosed_list_delete` that takes two arguments, a pointer to an `enclosed_list` structure and a value of type `size_t` (as defined in the standard `stdint.h` header file), and removes the node at the (zero-based) position specified by the second argument in the list stored in the `front` field of the `enclosed_list` structure to which the first argument points from the list, freeing the storage block that the node occupies. Do not leave any dangling pointers to this block anywhere in the structure.

Again, it is a precondition of this function that the second argument is strictly less than the length of that list (before the deletion). Note that this implies that the `enclosed_list_delete` function cannot be called at all if the first argument is a pointer to an empty `enclosed_list` structure, since the `size_t` type provides no position numbers less than 0. You can’t delete something that isn’t there!

**Exercise 1100:** Write and test a C function `enclosed_list_insert` that takes three arguments, a pointer to an `enclosed_list` structure, a value of type `size_t` (as defined in the standard `stdint.h` header file), and an `int`, and allocates and inserts a new node containing the specified `int` in its `datum` field at the (zero-based) position specified by the second argument in the list stored in the `front` field of the `enclosed_list` structure to which the first argument points.

For this function, the second argument must be less than or equal to the length of the list (before the insertion). If it’s equal to the length of the list, the new node must be attached at the end of the list.