**Cpsc 1071 Lab 7: Object List Manager**

Chapter 8 of zyBooks discusses a simple linked list. A disadvantage of that linked list is that it can only be used to maintain lists of int values. A more useful system would allow us to maintain lists of any type of object (integers, floats, structures, etc.). In this lab we will use *void* pointers to implement a generic list that will allow to do just that.

**Goals**

* Develop a generalized list manager
* Understand the use of the *"void"* pointer.
* Understand how to develop modules that are general and can be applied to many different problem areas.
* Obtain additional experience with pointers.

**Background: void Pointers and type casting**

A *void* pointer can be used to hold the address of an arbitrary data type. For example, consider the following segment of code:

int x;

float y;

typedef struct s\_type

{

int f1;

int f2;

} stype;

stype s1, \*sp;

void \*p;

The *void* pointer *"p"* can be set to point to any of the objects above. e.g.

p = &x;   
p = &y;   
p = &s1;   
etc.

You can also write *"p = sp;"* or *"sp = p;"* without any problem (both *"p"* and *"sp"* are of type pointer -- they just happen to be pointing to different types of objects).

The problem occurs when you try to use *"p"* to access a field. For example, if you write *"x = \*p;"* you will get the rather obnoxious message

"...warning: dereferencing 'void \*' pointer" followed by "...error: void value not ignored as it ought to be".

There are two ways to avoid this problem:

1. **type casting:** You can "tell" the compiler the type of the data that *"p"* is pointing to, i.e.:

x= \*(int \*)p;

This is telling the compiler to treat "p" as a pointer to something of type "int", and then copy the integer contents it is pointing to to x.

1. **Define a separate integer pointer and copy "p" to it**: the other technique is to simply define another pointer, i.e.:

int \*q;   
...   
q = p;   
x = \*q;

Defining a few extra pointers of the appropriate type may result in more readable code and will likely be less error-prone (but either approach is okay).

Here is another example using the above structure. Assume we want to set the "f2" field in "s1". Assume "p" is currently pointing to "s1". We could:

1. write:

((stype \*)p)->f1 = 6;

1. or (noting that "sp" is defined above as pointing to something of type "stype"):

sp = p;   
sp->f1 = 6;

**List management functions -- the list\_t object**

In class, we have looked at a simple list structure. For this lab exercise, we will change the list structure so that:

1. objects of arbitrary type can be saved in a list,
2. objects are added to the FRONT of the list
3. an "iterator" object is implemented that can be used to process the nodes of the list.

We will also only add to the list -- we will not implement deletes. You are provided with the following list.h file:

/\*\* list.h -- lab 7 \*\*/

/\*\* List node \*\*/

typedef struct node\_type

{

void \*dataPtr; /\* Pointer to the associated \*/

struct node\_type \*next; /\* Pointer to next node \*/

} node\_t;

/\*\* List structure \*\*/

typedef struct list\_type

{

node\_t \*head; /\* Pointer to front of list \*/

} list\_t;

/\*\* Iterator \*\*/

typedef struct list\_iterator

{

list\_t \*list; /\* List iterator is associated with \*/

node\_t \*position; /\* Current position in list \*/

} iterator\_t;

/\*\* Function prototypes \*\*/

list\_t \*newList(); /\* Create and initialize list object \*/

void list\_add(list\_t \*list, void \*objPtr); /\* Insert object into list \*/

iterator\_t \*newIterator(list\_t \*list);

void list\_reset(iterator\_t \*iter); /\* Reset position to front of list \*/

void \*list\_next(iterator\_t \*iter); /\* Get object from list \*/

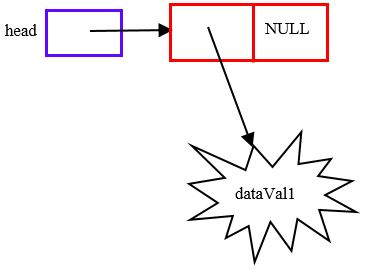
int l\_hasnext(iterator\_t \*iter); /\* Test if at end of list \*/

Note the use of the *void* pointer in the node structure. *object* will point to data of arbitrary type. *list\_add()* will add an object to the list, and *list\_next()* will return a void pointer to an object retrieved from the list.

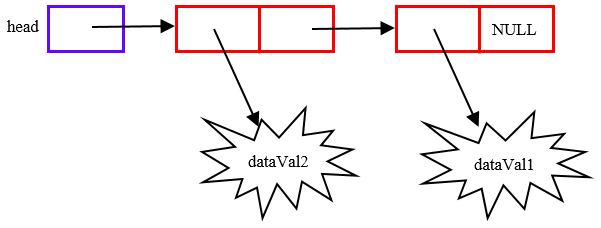
Next note the list structure. The *head* pointer points to the FRONT of the linked list. For this implementation, new entries will be added via *list\_add()* to the front of the list. Here are some examples. Initially the list is empty, in which case both head and tail are equal to NULL , i.e



After adding one entry in the list, the list would look like:



Then after adding a second entry:



**Processing a list -- the iterator\_t object**

The list\_t functions described above provide a means to add nodes to the list, but there are no functions to process the nodes of a list (i.e. retrieve the nodes). Both Java and C++ provide another class known as the iterator class that is associated with a list and can be used to sequentially retrieve elements of a list. We will develop our own version of an iterator. The structure of our iterator will be:

typedef struct list\_iterator

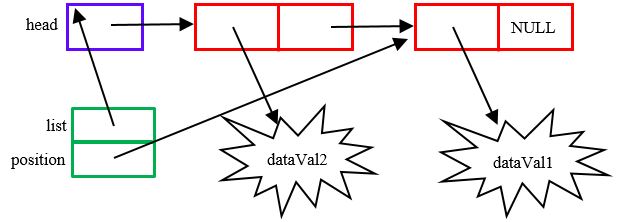
{

list\_t \*list; /\* List iterator is associated with \*/

node\_t \*position; /\* Current position in list \*/

} iterator\_t;

The *list* field points to the list\_t the iterator is associated with, and *position* points to the node that would be retrieved next. For example the following illustrates interator\_t *iter1* associated with list\_t *list1* The current position in the list is the node associated with *dataVal2* (as we will see this would be the next object retrieved).



**Tasks**

*Task 1:*   
Create a **"lab7"** directory and copy the lab files to the directory using:

lab1071copy 7

Next untar the file, i.e.:

**tar -xvf lab7.tar**

You should now have the files *listtest.c* *list.c*and *list.h*in your lab7 directory. Note: the provided *list.c* includes dummy *do-nothing* routines **but**it will save you from having to enter the function headers.

Also note that there is a *Makefile*. You can compile the code by simply typing *make*

*Task 2:* Create the module *list.c* and implement the following functions:

*list\_t functions:*

list\_t \*newList();

Create and initialize an instance of a list object. The *current* *head* and *tail* pointers should be initialized to NULL.

void list\_add(list\_t \*list, void \*object);

Add the object to the END of the linked list. Note: unlike the add function we looked at in class, DO NOT create a copy of the object Â– just add the pointer to the object as pr ovided by the caller. For this implementation it would be up to the caller to make sure that copies were made if needed.

*iterator\_t functions:*

iterator\_t \*newIterator(list\_t \*list);

Create and initialize an instance of an iterator\_t object. The *list* field of the iterator\_t should point to the list, and *position* should point to the current first node (head) of the list.

void list\_reset(iterator\_t \* iterator);

Reset the *position* pointer to the head of the list associated with the iterator.

void \*list\_next(iterator\_t \*iterator);

Return the object pointer in the node pointed to by *position* and advance *position* to the next node. If *position* is NULL, return NULL. Note that this function returns the pointer to the object, NOT THE NODE. In the illustration above this function wo uld return a pointer to *object2* and set the *position* field of the iterator to NULL (the *next* field).

int l\_hasnext(iterator\_t \*iterator);

Returns 0 if *position* is NULL (i.e. at end of list), 1 otherwise (i.e. not at end of list).

*Task 3:*

The test program currently only prints the contents of *list1* Add to the end of the program code that will print *list2* and *list3* using the iterators *iter2* and *iter3* in the format illustrated below:

List 1: 25, 20, 15, 10, 5,

Test list\_reset() function: First value = 25

List 2: banana, peach, orange, apple,

List 3: [34577 Toyota Prius 2013], [54387 Nissan Altima 2016], [34257 Chevrolet Impala 2017], [15677 Jeep Cherokee 2014], [32565 Toyota Camry 2016], [32168 Honda Accord 2015], [23456 Ford Mustang 2017],

Hint:

Note the special case when the first node is added to the list by list\_add(). In this one case both the *head* and *tail* pointers in the list\_t have to be set to point to the new node . When the list is NOT empty the tail pointer of the list\_t is updated to point to the new node, and the *next* pointer of node that was at the tail is updated to point to the new node.

**Submission**

Use [Handin](https://handin.cs.clemson.edu/) to submit your list.c and listtest.c modules.