CS3841 Lab 2
A Linked List in C

Dr. Walter Schilling

Due: September 20, 2012 23:59 CDT

1 Lab Objectives

1. Practice C development in a UNIX environment.
2. Use malloc and free to manage the allocation and deallocation of dynamic memory.
3. Implement a doubly linked list in the C programming language.
4. Understand the purpose for the void pointer in C.
5. Apply appropriate casts to correctly use a void pointer.
6. Implement and use C structures to solve a software problem.
7. Use test cases to verify the correct operation of a constructed source code module.

2 Recommended Reference Readings

Topic	Book	Pages
Structures in C	C: A Reference Manual Fifth Edition	148-154, 158-160
Dynamic Memory Allocation in C	C: A Reference Manual Fifth Edition	407-410

3 Introduction

In your freshman year, you took Data Structures. In this class, you learned about Data Structures and how you can use the Java Collections Framework to store data. In many sections, you constructed a Linked List class in Java as well. Throughout your freshman sequence, you also learned about how Java uses garbage collection to manage memory.

This is all very good experience. However, not all languages offer these features. In specific, the C language does not provide either one of these capabilities. You, as a developer, are responsible for managing your own memory, both allocating memory and freeing it when you no longer need it. Furthermore, you as a developer are responsible for developing your own data collections.

In this lab, you will be constructing a linked list library in C. This library can then be used on subsequent projects to store data. In doing so, you will also learn about the management of memory in C using malloc and free as well as practicing typecasting between pointer types.

In order to make it easier to develop this lab, you have been provided with a set of CPPUnit tests. These unit tests are similar in purpose to JUnit tests in that they will verify the correct operation of your developed library.

4 Specific Requirements

1. Your program shall be developed under UBUNTU Linux.
2. Your linked list implementation shall implement the functionality defined in the appendix of this document.

3. All code must be fully commented. At the top of each file shall contain a comment block with your name, your course, the assignment name, the date, and what the file is responsible for doing. Each function shall have a function block declaring the purpose for the function, the parameters accepted, and the return value (if one exists).

4. All functions which are externally visible shall be defined in an appropriate header file.

5. Header files must be protected against multiple inclusion using an appropriately constructed ifdef or ifndef construct.

6. When completed, your implementation shall pass all tests in the CPPUnit test suite for the linked list program.

5 Design

The design that you are to create has been provided to you in UML in Figure 1. This design shows the interface, the structures you are to use internally, and the functions that are to be implemented. The purpose for these functions is defined in the appendix to this document. The exact implementation is to be left up to you. However, the code must compile cleanly in a UBUNTU Linux environment and must pass all CPPUnit test cases.

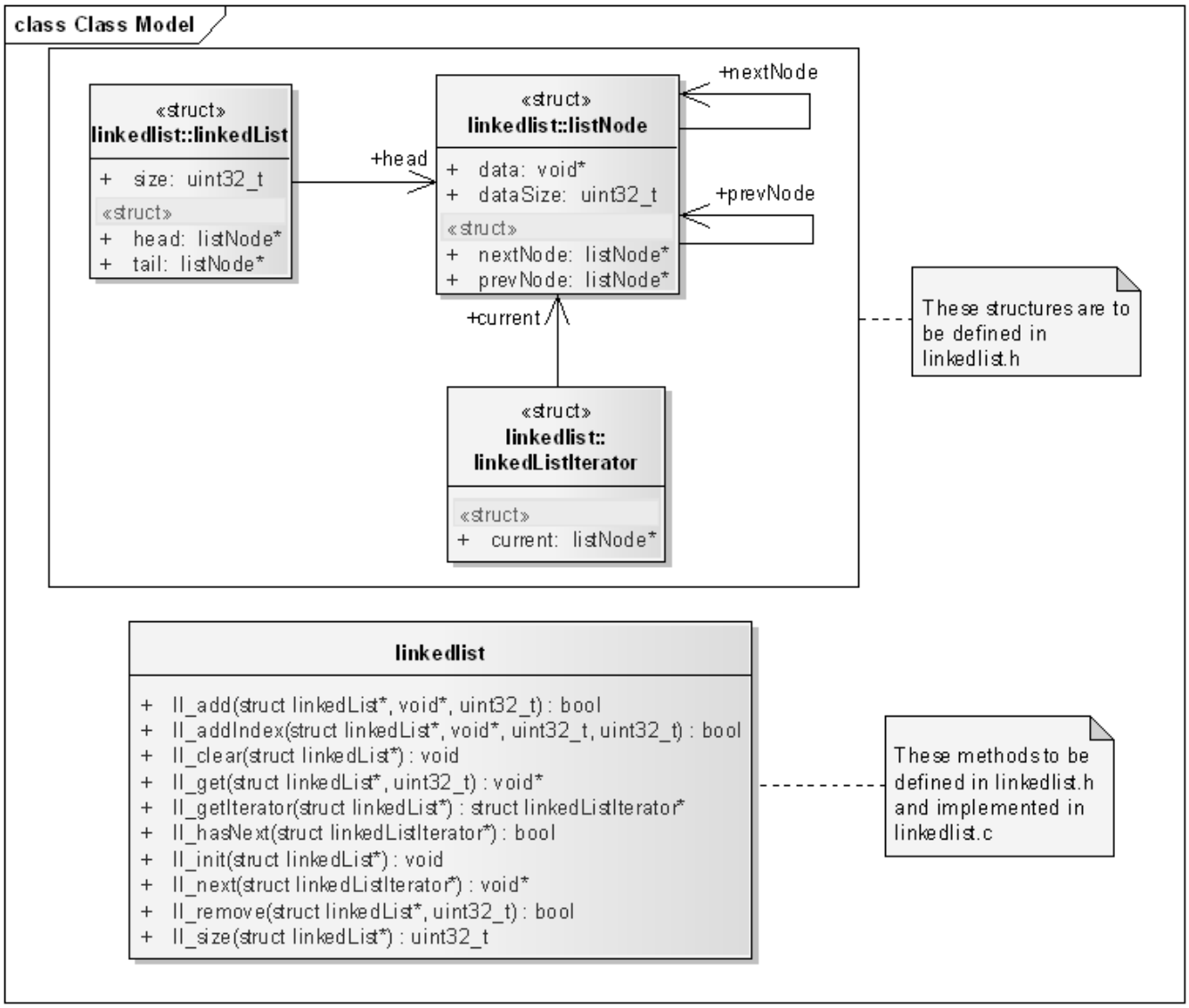

Figure 1: Sample UML diagram showing a potential design solution.

6 Development Process

The process you use to develop this package is entirely up to you. However, you will need to use the gnu debugger (gdb) to debug your software (unless your code is written perfectly the first time...). A complete manual on this package can be found at http://www.gnu.org/software/gdb/documentation/. To do this, you will need to make certain your makefile compiles your program with the -g option. There also is a video tutorial on the course website.

7 Deliverables

1. Tar file including all of your source code, makefile, etc. Your source code must compile without any compiler warnings or errors under 32 bit UBUNTU Linux.
2. Lab report, submitted in pdf format, with the following
   (a) Name, date, and lab title
   (b) What things went right and what things went wrong with this lab.
   (c) What did you learn from this lab.
   (d) Output captures of the CPPUnit test executing properly on your machine.
   (e) Conclusions
   (f) Source code as an appendix (In order that it can be commented). (Note: It is only required that you submit your source code in the report. Provided code does not need to be submitted.)

Deliverables are to be uploaded using the course website upload script.

8 API Specification

Operation Type	Function Prototype	Description and parameter
Create Operation	void ll_init(struct linkedList* list)	This method will initialize an instance of a linked list, passed as a parameter, to default values. This includes setting the head and tail to NULL and the size to 0. Parameters: struct linkedList* list : This is a pointer to the list that is initialized. The method must verify that this parameter is not NULL before dereferencing it. Preconditions: list is a valid, non-NULL pointer to a struct linkedList structure which has been declared external to the linked list library. Postconditions: The head and tail of the list are set to NULL. The size of the linked list is set to 0.
Destroy Operation	void ll_clear(struct linkedList* list)	This method will clear a linked list and restore it to its default, uninitialized state. All linked nodes will be returned to the heap using the free operation. Basic Algorithm: Starting with the head, iterate through the list. As each node is reached, free the data memory with a call to the free function. After the data memory has been freed, determine the next node before freeing the listNode structure. Parameters: struct linkedList* list : This is a pointer to the linked list that is to be cleared. The method must verify that this parameter is not NULL before dereferencing it. Preconditions: list must be a pointer to a struct linkedList which has previously been initialized with the ll_init function. Postconditions: All allocated memory has been returned to the heap. The head and tail of the list are set to NULL. The size of the linked list is set to 0.

Add Operation	bool ll_add(struct linkedList* list, const void* object, uint32_t size)	This function will add a the data pointed to by the object pointer to the end of the current linked list. This size will be adjusted accordingly. Basic algorithm: The function must first check the validity of all parameters. This list must not be NULL, the object must not be NULL, and the size must be greater than 0. If all parameter validations pass, then the function shall allocate memory using malloc. The first allocation is for a listNode instance. The second allocation is for the data. The data will be copied from the address pointed to by the object parameter to the memory allocated by the second call to malloc. This listNode instance data will be set to point to the memory allocated for the data by the second malloc call. Once this is finished, the listNode instance will be added to the tail of the list and the links within the list will be updated appropriately. The list size will also be incremented by 1. Parameters: struct linkedList* list : This is a pointer to the list that is to have data added to it. The function must verify that this parameter is not NULL before dereferencing it. const void* object : This is a pointer to the object that is to be added onto the linked list. Typically this will be a C struct, though it may also be an array or other c data type. The function must verify that this parameter is not NULL before dereferencing it. uint32_t size : This is the size of the object that is to be added onto the linked list. Typically, if a structure is being added to the list, this will be obtained through the sizeof() operator when the call to this function is made. The function must verify that this parameter is greater than 0. Return Value: This function will return true if it is able to successfully complete and the appropriate data object is added to the end of the linked list. It will return false if any of the parameters is NULL or otherwise out of range or if the memory allocation calls fail. Preconditions: list must be a pointer to a struct linkedList which has previously been initialized with the ll_init function. Postconditions: The data pointed to by the object parameter will be added to the linked list. The size will be incremented by 1. The tail of the list will point to the new node which has been added.
Add Operation	bool ll_addIndex(struct linkedList* list, const void* object, uint32_t size, uint32_t index)	This function will add a the data pointed to by the object pointer to the location defined by the parameter index. The size will be adjusted accordingly. Basic algorithm: The function must first check the validity of all parameters. This list must not be NULL, the object must not be NULL, and the size must be greater than 0. The index must fall within the range $0 \leq index \leq listSize - 1$. If all parameter validations pass, then the function shall allocate memory using malloc. The first allocation is for a listNode instance. The second allocation is for the data. The data will be copied from the address pointed to by the object parameter to the memory allocated by the second call to malloc. This listNode instance data will be set to point to the memory allocated for the data by the second malloc call. Once this is finished, the appropriate location shall be found within the list by iterating from the head to the index location. At this point, the links shall be updated appropriately. The list size will also be incremented by 1. Parameters: struct linkedList* list : This is a pointer to the list that is to have data added to it. The function must verify that this parameter is not NULL before dereferencing it. const void* object : This is a pointer to the object that is to be added onto the linked list. Typically this will be a C struct, though it may also be an array or other c data type. The function must verify that this parameter is not NULL before dereferencing it. uint32_t size : This is the size of the object that is to be added onto the linked list. Typically, if a structure is being added to the list, this will be obtained through the sizeof() operator when the call to this function is made. The function must verify that this parameter is greater than 0. uint32_t index : This is the index at which the specified data is to be added. This must be within the range $0/leqindex/leqlistSize - 1$. Return Value: This function will return true if it is able to successfully complete and the appropriate data object is added to the linked list. It will return false if any of the parameters is NULL or otherwise out of range or if the memory allocation calls fail. Preconditions: list must be a pointer to a struct linkedList which has previously been initialized with the ll_init function. Postconditions: The data pointed to by the object parameter will be added to the linked list. The size will be incremented by 1. The head and / or tail may be updated, depending on the index value passed in.

Retrieve Operation	void* ll_get(struct linkedList* list, uint32_t index)	This method will retrieve data from the linked list, placing the data in the data destination memory location. Basic algorithm: The function must first check the validity of all parameters. This list must not be NULL and the index must fall within the range $0 \leq index \leq listSize - 1$. If all parameter validations pass, then the iterate from the head until the appropriate index is reached. At this point, a pointer to the appropriate data is returned. Parameters: struct linkedList* list : This is a pointer to the linked list. The function must verify that this parameter is not NULL before dereferencing it. uint32_t index : This is the index for the data which is to be retrieved. This must be within the range $0 \leq index \leq listSize - 1$. Return Value: This function will return a valid pointer if it is able to successfully complete and the appropriate data object is obtained from the linked list. It will return NULL if any of the parameters is NULL or otherwise out of range. Preconditions: list must be a pointer to a struct linkedList which has previously been initialized with the ll_init function. Postconditions: A pointer to the desired memory will be returned.
Remove Operation	bool ll_remove(struct linkedList* list, uint32_t index)	This method will remove an item from the given linked list. When this method is completed, the list will be reduced in size by 1 and the allocated memory will be freed. Basic algorithm: The function must first check the validity of all parameters. This list must not be NULL and the index must fall within the range $0 \leq index \leq listSize - 1$. If all parameter validations pass, then the iterate from the head until the appropriate index is reached. At this point, adjust the links so that the node is no longer part of the list. Using the free function, free the data memory and then free the node of the linked list. Parameters: struct linkedList* list : This is a pointer to the linked list. The function must verify that this parameter is not NULL before dereferencing it. uint32_t index : This is the index for the data which is to be removed. This must be within the range $0 \leq index \leq listSize - 1$. Return Value: This function will return true if it is able to successfully complete and the appropriate data object is removed from the linked list. It will return false if any of the parameters is NULL or otherwise out of range. Preconditions: list must be a pointer to a struct linkedList which has previously been initialized with the ll_init function. Postconditions: The data stored in element index will be removed from the linked list. The size of the list will be decreased by 1. The head and / or tail of the list may be updated based on the removed element of the linked list.
Iterator	struct linkedListIterator* ll_getIterator(struct linkedList* list)	This function will return an iterator for the linked list. The iterator is contained within the linkedListIterator structure returned by this function. Basic algorithm: Verify that the list parameter is not NULL. Using malloc, allocate memory for a linkedListIterator structure. Set the current element of this allocated structure to point to the head of the list. Then return the allocated structure. Parameters: struct linkedList* list : This is a pointer to the linked list. The function must verify that this parameter is not NULL before dereferencing it. Return Value: This function will return a pointer to a linkedlistIterator structure. The return value will be NULL if there is any problem creating this structure. Preconditions: list must be a pointer to a struct linkedList which has previously been initialized with the ll_init function. Postconditions: A new linkedListIterator will be allocated on the heap, and the current element of this structure will be set to the head of the list.
Iterator	bool ll_hasNext(struct linkedListIterator* iter)	This method will determine if the given iterator has another element. If yes, a value of true will be obtained. If not a value of false will be returned. Parameters: struct linkedListIterator* iter : This is a pointer to the iterator. The function must verify that this parameter is not NULL before dereferencing it. Return Value: This function will return true if the given iterator has another element. It will return false if either the iterator is NULL or there are no additional items in the linked list. Preconditions: iter must be a pointer to a struct linkedListIterator which has previously been created by a call to the ll_getIterator function.

Iterator	void* ll_next(struct linkedListIterator* iter)	This function will obtain the next object from the linked list. A pointer to the next object will be returned. Basic algorithm: Verify that the iterator is not NULL. Determine the location for the given data and define a pointer to it. Update the current to point to the next node on the linked list. Return the pointer. Parameters: struct linkedListIterator* iter : This is a pointer to the iterator. The function must verify that this parameter is not NULL before dereferencing it. Return Value: The return value will be a pointer to the next object on the linked list. It if the iterator is invalid or there is no next object, the return value will be NULL. Preconditions: iter must be a pointer to a struct linkedListIterator which has previously been created by a call to the ll_getIterator function. Postconditions: The next element on the list will be returned. The iterator will be advanced to reference the next element on the linked list.
Size	uint32_t ll_size(struct linkedList* list)	This function will obtain the size of the linked list, namely the number of objects stored on the list. Basic algorithm: Verify that the list is not NULL. Return the size element of the linked list structure. Parameters: struct linkedList* list : This is a pointer to the linked list. The function must verify that this parameter is not NULL before dereferencing it. Return Value: The return value will be the size of the list. If the list is empty or the list parameter is NULL, a value of 0 will be returned. Preconditions: iter must be a pointer to a struct linkedList.