Week 8 Lecture Notes

Linked Lists and Binary File I/O in C++

These material in these notes can also be found in this course’s textbook, Data Structures and Algorithms in C++ by Goodrich et al, Chapters 3.2.1-3.2.3, and 3.3.

Know definitions for all of the **bold, underlined** words that follow in these lecture notes.

Contents

[Linked Lists in C++ 1](#_Toc45789980)

[Node Class for a Linked List Class 3](#_Toc45789981)

[Linked List Class 4](#_Toc45789982)

[C++ Binary File I/O 7](#_Toc45789983)

# Linked Lists in C++

Let’s look at a class that dynamically allocates memory, a LinkedList class. As we know, a doubly-linked list has at least a head pointer which is a pointer to the first node in the list and a tail pointer that points to the last node in the list. In these lecture notes, we will make this linked list more complicated than just a list of integers. We will make a list of book objects. The list will have nodes in it, but each node will have a book object in it, not an integer. First we will create a class called Book that will have instance variables for an ISBN, title, author, publisher, year published, and edition.

**Before class:** Create a new workspace/project in the IDE called pp8a. MAKE SURE YOU BROWSE TO THE LOCATION WHERE YOU ARE SAVING WORK FOR THIS CLASS. Create a new project in this workspace called pp8a. Add two new files, book.h and book.cpp that implement a Book object with private data ISBN, title, author, publisher, year and edition. Write public set/get member functions for each private data member. Overload the extraction and insertion operators (>> and <<) and the == (comparison) operator. Write a default constructor. I suggest making the title, author and publisher defaults the empty string. DO NOT LEAVE A PRIVATE DATA MEMBER UNSET! Constructors should set ALL data members to something! You may use C++ string objects instead of C-strings since we’re coding in C++ now. Compile book.cpp and get rid of any compilation errors. This class, Book, is the data in our node. Below is the class code for book.h. Don’t forget the #ifndef and #include pre-processor directives. Also, recall that C++ getline functions read the delimiters (like the newline) from the input stream whereas the extraction operator, >>, does not. Sometimes you have to read the newline with an istream function like get when mixing and matching the use of getline and the extraction operator. When creating text input files for programs on the PC, don’t use Notepad. Notepad places 2 newlines at the end of every line although the file looks like it has only one newline. It is best to create your text input files with the editor in the IDE.

|  |
| --- |
| class Book {  private:     string isbn;     string title;     string author;     string publisher;     int year;     int edition;  public:     // constructors     Book();     Book( string, string, string, string, int, int );     // copy constructor     Book( const Book& );     // destructor     virtual ~Book(){}     // set/get for private data     void setIsbn( string );     string getIsbn() const { return isbn; }     void setTitle( string );     string getTitle() const { return title; }     void setAuthor( string );     string getAuthor() const { return author; }     void setPublisher( string );     string getPublisher() const { return publisher; }     void setYear( int );     int getYear() const { return year; }     void setEdition( int );     int getEdition() const { return edition; }     void setBook( const Book& );     // overloaded assignment operator     Book& operator=( const Book& );     // overloaded << and >>     friend ostream& operator<<( ostream&, const Book& );     friend istream& operator>>( istream&, Book& );     // overloaded == compares titles and isbns     friend bool operator==( const Book&, const Book& );  **// make Node class a friend**  **friend class Node;**  }; |

Note that the Book class makes the entire Node class a friend. This way code in the Node class can access the private data in the Book class. On the other hand, the Node class will NOT make the Book class a friend. The Node class has a book in it so it makes sense that the Node class has access to a book’s private data. This relationship is called **composition** – when one class has one or more objects of another class in it. Composition is a “has a” relationship. A node “has a’ book. The Book class does not have a Node in it, so it does not make sense that the Book class has access to a node’s private data. The Book class doesn’t know whether it’s in an array, a list, a vector, or even a tree It’s just a book. Note: You DO NOT need to #include “node.h” in book.h for the friend statement.

## Node Class for a Linked List Class

Now, let’s look at a Node class that has a private Book object and two private pointers to Node objects called nextPtr and prevPtr. This class also requires a public, default constructor. The default constructor should set both pointers to NULL. Since constructors MUST set ALL instance variables, the Node default constructor needs to set the node’s book object to some default book. In this case, the Book object’s default constructor will be called in the Node class’ default constructor without the need to write that code explicitly. In the case of composition, the C++ compiler inserts a call to the data member class, in this case Book, at the beginning of all the constructors of the class that contains the object, in this case Node, unless a Book constructor is called in the Node class initialization list. This is another reason to always write a default constructor.

The Node class default constructor does not dynamically allocate memory. Its algorithm is:

1. book = Book() (although this call happens automatically – don’t write code for this step.
2. nextPtr = prevPtr = NULL

Besides the default constructor, the Node class should have a constructor that takes a const Book& object and uses this parameter to initialize the data. Set the pointers to NULL in this constructor, also. Constructors must set ALL instance variables. Another useful constructor is a copy constructor for the Node class whose parameter would be const Node&. Recall copy constructors always have one parameter which is a constant reference to an object of the class.

Since the Book object in Node is private, the Node class should have set/get functions for it. The Node class should probably not have set/get functions for the next and previous pointers since it doesn’t make sense for other code in the program that is not in the Node or LinkedList class to have access to those pointers. Below is the code for node.h.

|  |
| --- |
| #ifndef NODE\_H  #define NODE\_H  #include "book.h"  class Node {  private:  Book book;  Node\* nextPtr;  Node\* prevPtr;  public:  Node();  Node( const Book& );  Node( const Node& );  virtual ~Node() {}  void setBook( const Book& );  Book getBook() const { return book; }  // LinkedList is a friend  friend class LinkedList;  };  #endif |

In order to allow the LinkedList class below to access the private data in the Node class, the Node class should make the LinkedList class a friend. Note: You DO NOT need to #include “linkedlist.h” in node.h for the friend statement.

**Pair Programming 8a:** In the IDE’s workspace/project, create two new files, node.h and node.cpp. Put the code above in node.h and add the appropriate member functions to node.cpp. Compile node.cpp and get rid of any syntax errors.

## Linked List Class

Now, let’s look at the LinkedList class. It has two private data members, headPtr and tailPtr, both pointers to Node objects. It needs a default constructor. What goes in this constructor? The same code that we wrote to create the linked list written in C earlier. We need to set headPtr and tailPtr to NULL. What other functionality does our LinkedList class need? Well, the same functionality we wrote before in C except this functionality will be implemented by class member functions. The basic algorithms for inserting, traversing, finding, etc. in a doubly-linked list haven’t changed just because we’re creating classes in C++.

In addition to the functions we wrote in C, we MUST write 1) a copy constructor, 2) an overloaded operator=, and 3) a destructor. Why? Because the LinkedList class dynamically allocates and deallocates memory. ***ALL CLASSES THAT DYNAMICALLY ALLOCATE MEMORY NEED TO HAVE THESE THREE MEMBER FUNCTIONS!***

**Pair Programming 8a:** Add linkedList.h and linkedList.cpp to your project. You might want to start member function code with your linked list C code then modify it. Below is a list of LinkedList member functions and their descriptions. The code for LinkedList.h follows. Read through it. Since most of the algorithms for these functions are identical to the C versions we’ve already written, the only algorithm described below is for operator= .

* LinkedList(): the default constructor. Recall, all constructors must set all instance variables.
* LinkedList( const LinkedList & ): the copy constructor which must make a deep copy
* void insertNode( Node\* ): inserts the Node\* parameter at the head of the list and increments the list’s count of the number of nodes it contains
* void traverseQueue() const: traverses and prints all book ISBN and titles in the list from tail to head.
* void traverseStack() const: traverses and prints all book ISBN and titles in the list from head to tail.
* Node\* findNode( const Book& ) const: finds the node with the matching book parameter in it. The function pre-condition is that the Book class has an overloaded operator== which compares title then ISBN numbers. The function returns a pointer to the node where it found the book or NULL if not found.
* void deleteNode( Node\* ): deletes the node parameter from the list. Pre-condition: the node parameter actually points to a node in the list. If it doesn’t, the function’s behavior is not defined and can be whatever you want – even causing a segmentation fault. This is analogous to passing strcpy an invalid character pointer which would cause a segmentation fault. The post-condition is that the node has been deallocated from memory. This function might be useful after a call to findNode.
* Node\* dequeueNode(): removes the node at the tail of the list and returns a pointer to it or NULL if the list is empty. Post-condition: the node still exists and has not been deallocated.
* Node\* popNode(): removes the node at the head of the list and returns a pointer to it or NULL if the list is empty. Post-condition: the node still exists and has not been deallocated.
* void deleteList(): deallocates all nodes in the list, sets the head and tail pointers to NULL, sets the count of the number of nodes to 0. Is called by the destructor.
* LinkedList& operator=( const LinkedList& ): deletes the invoking object’s list the does a deep copy of the parameter to the invoking object. For example,   
  list2 = list1;  
  which is really list2.operator=( list1 ) where list2 is the invoking object and list1 is the parameter.

|  |
| --- |
| #ifndef LINKEDLIST\_H  #define LINKEDLIST\_H  #include "book.h"  #include "node.h"  class LinkedList {  public:  LinkedList() : headPtr( NULL), tailPtr( NULL ), count( 0 ) {}  LinkedList( const LinkedList& );  void insertNode( Node\* ); // at head  void traverseQueue() const;  void traverseStack() const;  Node\* findNode( const Book& ) const;  void deleteNode( Node\* );  Node\* dequeueNode(); // from tail  Node\* popNode(); // from head  virtual ~LinkedList() { deleteList(); }  void deleteList();  bool isEmpty() const { return headPtr == NULL; }  LinkedList& operator=( const LinkedList& );  private:  Node\* headPtr;  Node\* tailPtr;  int count;  };  #endif |

Why must we create a copy constructor and an overloaded assignment operator for classes that dynamically allocate memory? If we don’t, we get what’s called a **shallow copy**. A shallow copy means that each object doesn’t have its own copy of the data that was dynamically allocated. For example, what happens if I have a linked list, list1, and I execute the following code without overloading the assignment operator, =

LinkedList list1, list2;

// then we insert a bunch of nodes into list1

list2 = list1;

Now, list2 has copied what data members? Well, there are only two data members in a LinkedList object, headPtr and tailPtr, so they are the only things that gets copied to list2. So, list2’s headPtr and tailPtr contains the same memory addresses as list1’s headPtr and tailPtr which means list2 and list1 are the same list. list2 isn’t a separate copy of list1. So, if some nodes are removed from list1, they are removed from list2 because they are the same list! We just did a shallow copy. We only copied the two pointers, headPtr and tailPtr, but not all of the dynamically allocated nodes. If we want to actually get a completely new list from the statement:

list2 = list1;

Then, we have to overload the assignment operator so that it allocates a completely new list and copies all of list1’s node data to the new list2’s node data: This is a **deep copy**.

Also, if we don’t write a destructor for classes that dynamically allocate memory, we will get **memory leaks**. A memory leak occurs when we no longer have a pointer to allocated memory, but it hasn’t been deallocated so the operating system keeps it for your program (although your program can’t use it) tying up memory resources that could be used for something else. Memory leaks can slow an entire system down and are really bad!

Below is the LinkedList operator= algorithm with one parameter (the list to copy) called l (“ell”). The function returns a reference to the new list so we could write:

list3 = list2 = list1;

1. delete this’ (the invoking object’s) current list so there’s no memory leak
2. if l’s headPtr isn’t NULL
   1. curPtr = l’s headPtr (to be used to traverse l’s list)
   2. this’ headPtr = new Node( l’s headPtr Node ) (get first node in this’ new list)
   3. thisPtr = headPtr // to be used to traverse this’ new list
   4. curPtr = curPtr->nextPtr
   5. while curPtr != NULL
      1. thisPtr ->nextPtr = new Node( the node that curPtr points to)
      2. thisPtr->nextPtr->prevPtr = thisPtr
      3. thisPtr = thisPtr->nextPtr (traverse this’ list)
      4. curPtr = curPtr->nextPtr (traverse l’s list)
3. tailPtr = thisPtr
4. count = l’s count
5. return this’ object

The algorithm for the copy constructor is identical to operator= with one exception: the pre-existing list does not need to be deleted because it does not already exist. Constructors are called when creating a brand new list after all.

When we start using pointers and dynamic memory allocation, we increase the opportunities for the program to access memory that doesn’t belong to it resulting in a segmentation fault and the program crashing when it executes. The first thing to do is determine where in the code the segmentation fault occurs. Using the debugger, put break points in the program to pause execution in order to narrow down the line number of the segmentation fault. We can also add cout statements to print information to narrow down the offending line number. **If you use cout statements, you must print endl also, which not only prints a new line but also flushes the buffer just like C’s fflush function.** The compiler may rearrange some the cout statements for optimization so they don’t actually execute where you place them without endl to flush the buffer.

**Pair Programming 8a:** Copy the text of file main8a.cpp from the pair programming assignment and paste it into the project’s main.cpp. Read through this file. Add a text file called b1.txt to the project and paste the contents of the pair programming assignment file b1.txt into it. Read through this file. It has data for four books in it. When you compile and execute the entire program. Where does the input file b1.txt go?

1. When simply executing, put the input file in the same folder as the workspace file (e.g., pp8.workspace)
2. When using the debugger, put the input file in the same directory as the header and cpp files (Why a different place? Ugh. I don't know.)

# C++ Binary File I/O

The steps for using any file are still the same:

1. Declare a variable to use as the file handle/file pointer/file object
2. Open the file (may have to specify mode)
3. Optional: check to see if the file exists or if it does, is empty
4. Access the file (read from it, write to it, seek to a different location in it, etc.)
5. Close the file as soon as you’re done with it

The code to do these steps in C++ for binary files follows.

1. Need either an ifstream or an ofstream variable, found in header file fstream (so, #include <fstream>). For example:

ofstream outBin;

ifstream inBin;

1. Open the file, in general:

*fileHandle.open( CStringFileName, mode );*

Some mode values are: binary, in, out, app (append output to the end of the file), trunc (delete contents of current file if there is one), all in the std::ios namespace. See <http://www.cplusplus.com/reference/fstream/fstream/open/> . Some, but not all, modes can be used together using the | (binary or) operator. For example:

outBin.open( “books.bin”, ios::out | ios::binary );

inBin.open( “books.bin”, ios::in | ios::binary );

1. Can still use the exists and/or fail member functions as shown below:

if ( inBin.fail() ) {

1. To write to a binary file, use the write ofstream member function, and to read, use the read ifstream member function:

char stuff[81] = “blah blah”;

outBin.write( stuff, 81 );

inBin.read( stuff, 81 );

Note the parameter types for these functions. The first is a pointer to a character, (char\*) – a memory address of a character. The second is the number of bytes to write/read. This has to be the exact number of bytes. In the example code this includes the terminating null character, \0. And the number of bytes has to match for the write and read. If the code wrote 81 bytes, it has to read exactly 81 bytes. The read function doesn’t automatically put the null character in the C-string parameter. It just reads bytes. So, you need to have written the null character. In the case above, the statement

outBin.write( stuff, 81 );

actually writes 81 characters even though there are 9 characters in the string plus \0 makes 10. So, the 10th character written was the null character. The 11th – 81st characters written are all just whatever is in memory at the time. Even though those characters aren’t part of the string that we care about, they still get written/read. The important thing about binary reading and writing is that everything is read/written in fixed-length pieces. So, if we write 81 characters then another 81 characters then another and so on in a loop, when we read exactly 81 characters every time in a loop we will get back exactly what was written. When we read with

inBin.read( stuff, 81 );

81 characters are read into the stuff array, including the 10th , the first null character, and all the remaining characters.

1. Close files as soon as you’re done with them:

outBin.close();

inBin.close();

What if you want to write (read) an integer? You will have to cast its address to a pointer to a character as shown here.

int i = 6;

outBin.write( (char\*)&i, sizeof( int ) );

What if you want to write (read) a double? You will have to cast its address to a pointer to a character as shown here.

double x = 12.345;

outBin.write( (char\*)&x, sizeof( double ) );

What if you want to write the entire contents of an array of doubles in a single statement? You will have to cast it (the array is already an address) to a pointer to a character as shown here. No loop needed!

double a[10]; // then a gets values in it somehow

outBin.write( (char\*)a, sizeof( double ) \* 10 );

What if you want to write a C++ string object? This is complicated by the fact that C++ string objects aren’t created with a particular size. If you look at sizeof( string ) you will see 8 (on PS11) because PS11 has 64-bit addresses. So, sizeof( string ) or even the size of a particular string variable, s1, sizeof( s1 ), is 8 no matter how many characters are in there. You could use the string class’ length member function, such as:

outBin.read( (char\*)&s1, s1.length() ); // problematic

but later, you would still have to know the length of the string you wrote, probably in some other function, even during the execution of some other program! The easiest way to solve this problem is DON’T WRITE C++ string OBJECTS TO BINARY FILES. Instead, copy a C++ string object to a temporary, fixed-length C-string and write that as shown here.

char temp[MAX\_LENGTH];

strcpy( temp, s1.c\_str() );

// where MAX\_LENGTH is a constant or #define

outBin.write( temp, MAX\_LENGTH );

We don’t need to cast temp to a pointer to a character because it already is one. We really need the variable temp because it has space for MAX\_LENGTH characters. If we write:

outBin.write( s1.c\_str(), MAX\_LENGTH );

and s1 doesn’t have MAX\_LENGTH characters in it, we could get a segmentation fault when our program basically goes out of bounds on the array of characters inside the C++ string object s1.

When reading the same data, read into a temporary C-string of the same length that was written before copying the C-string to the C++ string object with the overloaded = operator:

char temp[MAX\_LENGTH];

inBin.read( temp, MAX\_LENGTH );

s1 = temp;

Suppose we wanted to write a Book object to a binary file, how would we do it? A Book object should know how to binary write itself – encapsulation – so we should add a write member function for the Book class, such as the one shown below.

|  |
| --- |
| ostream& Book::write( ostream& outBin ) const {     char temp[128];     strcpy( temp, isbn.c\_str() );     outBin.write( temp, 18 ); //ISBN are 13 chars+4 dashes+\0     strcpy( temp, title.c\_str() );     outBin.write( temp, 128 ); // max title, etc. 127 chars     strcpy( temp, author.c\_str() );     outBin.write( temp, 128 );     strcpy( temp, publisher.c\_str() );     outBin.write( temp, 128 );     outBin.write( (char\*)&year, sizeof(int) );     outBin.write( (char\*)&edition, sizeof(int) );     return outBin;  } |

The write function above writes each Book data member separately. Alternatively, we could write a Book object in its entirety by using a temporary structure that uses C-strings instead of C++ string objects such as shown below. However we write the Book data, we have to read it **exactly** the same way!

|  |
| --- |
| ostream& Book::write( ostream& outBin ) const {  struct book\_t {  char isbn[128];     char title[128];     char author[128];     char publisher[128];     int year;     int edition;  } b;     strcpy( b.isbn, isbn.c\_str() );     strcpy( b.title, title.c\_str() );     strcpy( b.author, author.c\_str() );     strcpy( b.publisher, publisher.c\_str() );  b.year = year;  b.edition = edition;     outBin.write( (char\*)&b, sizeof(book\_t) );     return outBin;  } |

The following code creates a binary file called books.bin of book data from data entered at the console (or from redirected input using a file). In one or more labs, you will be required to create your own binary input file like this so read through the code carefully.

|  |
| --- |
| Book books[4];     ofstream binOut;     int i;     char junk;       // open book.bin for writing binary     binOut.open( "books.bin", ios::out | ios::binary );       // get  books from stdin and write them to a binary  // file called books.bin     for( i = 0; i < 4; i++ ) {        cin >> books[i]; // overloaded >> in Book class        cout << "in main: "  << books[i] << endl;        cin.get(junk);        books[i].write( binOut );     }     binOut.close(); |

**Pair Programming 8b:** Using the IDE, create another workspace/project called pp8b to test binary writing and reading. Add two new files to this new project called book.h and book.cpp. Copy the code from pp8a’s book.h and book.cpp files and paste it in these files. DO NOT ADD AN EXISTING FILE. Add two functions to the Book class:

ostream& write( ostream& ) const;

istream& read( istream& );

Copy main8b.cpp to this project and read through this file. The program creates a binary file called books.bin, reads from it, and prints the books read from the binary file to standard output to verify the binary writing and reading worked.

Regarding the location of the input file b1.txt:

1. When simply executing, put the input file in the same folder as the workspace file (e.g., pp8.workspace)
2. When using the debugger, put the input file in the same directory as the header and cpp files