Phonebook Application Documentation

Authors: Alex Escobar, Aneesh Kumar, Daniel Larson, Lars Thorogood

Table Of Contents

Overview

This is a Phonebook application created for the CSC 340 semester final project. The application allows users to create and manage contacts in a phonebook through a command-line interface. Users may choose one of *seven* menu options:

(1) - Print phonebook
(2) - Search by name
(3) - Search by phone number
(4) - Add entry to phonebook
(5) - Remove entry from phonebook
(6) - Load new phonebook from file
(7) - Sort phonebook by name
(0) - Exit

The application streamlines contact management for any user, who may navigate and use the features of the app without any technical knowledge.

Usage

All usage of the Phonebook application is through a command-line interface after running *main.cpp*. Users may select menu options 1 through 7 to interact with the phonebook, choosing menu option 0 to exit.

Searching Entries

Users can search the phonebook for specific entries either by name or by phone number:

By Name: Choose the search by name option (2), then enter the full name of the person or business you are searching for.

By Phone Number: Choose the search by phone number option (3), then enter the phone number of the person or business you are searching for.

Adding Entries

Users can update the phonebook by manually adding in entries for either a business or a person.

Add a new entry: Choose the add entry option (4), then follow the prompts. Specify the type (Person or Business). For Person, provide name, number, and birthday. For Business, provide name, number, and address.

Deleting Entries

Users can update the phonebook by manually specifying an entry to delete for either a business or a person.

Delete an entry: Choose the remove entry option (5), then follow the prompts. Enter the name of the business/person you would like to remove, then the entry will be deleted.

Sorting Entries

Users can sort the entries in the phonebook alphabetically by name.

Sorting By Name: Choose the sort option (7), and the phonebook will be sorted by name. You may choose to use menu option (1), print, to view the sorted phonebook.

Loading Data

Users can update the phonebook by adding in entries from a file, rather than adding them one by one with menu option (4).

Loading Data: Choose the load new phonebook from file option (6). When prompted, provide the absolute file path to the phonebook you would like to import. Note: file must be a .txt file in the format:
type,name,number,birthday/address

Example phonebook.txt:

```
Person,John Doe,555-1234,1990-01-01
Business,Alpha Enterprises,555-5678,100 Alpha St
Person,Jane Smith,555-2345,1992-02-02
Business,Beta LLC,555-6789,200 Beta Lane
```

Example Usage

```
===============================================
 === Welcome to the phonebook application! ===
===============================================
Options
------------------
(1) - Print phonebook
(2) - Search by name
(3) - Search by phone number
(4) - Add entry to phonebook
(5) - Remove entry from phonebook
(6) - Load new phonebook from file
(7) - Sort phonebook by name
(0) - Exit
Select an option -->4
 Enter type (person/business):person
 Enter name:Jane Doe
 Enter phone number:510-366-4993
 Enter birthday:06-22-05
 Options
------------------
(1) - Print phonebook
(2) - Search by name
(3) - Search by phone number
(4) - Add entry to phonebook
(5) - Remove entry from phonebook
(6) - Load new phonebook from file
(7) - Sort phonebook by name
```

```
(0) - Exit
Select an option -->1
 ENTRY 1 || Person Name: Jane Doe | Phone Number: 510-366-4993 | Birthday:
06-22-05
Options
------------------
(1) - Print phonebook
(2) - Search by name
(3) - Search by phone number
(4) - Add entry to phonebook
(5) - Remove entry from phonebook
(6) - Load new phonebook from file
(7) - Sort phonebook by name
(0) - Exit
Select an option -->0
 Exiting Phonebook App.
```

Code Structure

User Class

The User class is an abstract base class designed to represent a generic user. This class contains fundamental attributes and methods of every single entry type in the phonebook, namely Persons and Businesses. All attributes/methods are public unless specified otherwise. List of attributes and methods:

Attributes:

`name (string)`: Stores the full name of the user. This attribute is protected, allowing access inside the User class and derived classes.

`phoneNumber (string)`: Stores the phone number of the user. This attribute is protected, allowing access inside the User class and derived classes.

Constructors:

`User(string name, string phoneNumber)`: This constructor is called by derived classes of the User class and sets the attributes `name` and `phoneNumber`.

Methods:

`virtual void display() const = 0`: Derived classes must override this method in order to be concrete. This method is intended to print out the information of the User object.

Getters and Setters:

```
string getName() const
string getPhoneNumber() const
void setName(string newName)
void setPhoneNumber(string newPhoneNumber)
```

Operator Overloading

`bool operator>(const User& other) const`: Overloads the greater than operator to compare to User objects based on their names. This is used in sorting operations.

Person & Business Classes

The Person and Business classes are derived classes of the User abstract base class. They retain attributes like name and phoneNumber, but introduce new attributes and methods for their respective types. All attributes/methods are public unless specified otherwise.

Attributes:

Person: `birthday (string)`: Stores the birthday of the Person. This attribute is private to the Person class.

Business: `address (string)`: Stores the address of the Business. This attribute is private to the Business class.

Constructors:

Person: `Person(string name, string phoneNumber, string birthday)`: The constructor instantiates a Person object with a name, phoneNumber, and birthday.

Business: `Business(string name, string phoneNumber, string address)`: The constructor instantiates a Business object with a name, phoneNumber, and birthday.

Methods:

Person & Business: `void display() const override`: This method overrides the display method in the User abstract base class. This displays all attributes of the object with labels.

Getters and Setters:

Person: `string getBirthday() const`
Person: `void setBirthday(string newBirthday)`
Business: `string getAddress() const`
Business: `void setAddress(string newAddress)`

Node Class

The Node class represents a node in the LinkedList. It is designed to hold individual entries (User objects) and links to the next and previous nodes in the list. All attributes/methods are public unless specified otherwise.

Attributes:

`user (User*)`: Stores a pointer to the User object. This attribute is private.

`next (Node*)`: Stores a pointer to the next node in the linked list. This attribute is private.

`prev (Node*)`: Stores a pointer to the previous node in the linked list. This attribute is private.

Constructors:

`Node(User *data, Node *nextNode = nullptr, Node *prevNode = nullptr)`: The constructor instantiates a node object. The User* pointer must be provided, while nextNode and prevNode will be set to nullptr if not provided.

Getters and Setters:

```
User* getUser() const
Node* getNext() const
Node* getPrev() const
void setUser(User* newUser)
void setNext(Node* newNext)
void setPrev(Node* newPrev)
```

LinkedList Class

The LinkedList class is a data structure to manage a collection of User objects. This is the primary structure used in the phonebook, which holds all of its data. It provides methods for adding/removing entries, searching, sorting, and displaying. All attributes/methods are public unless specified otherwise.

Attributes:

`head (Node*)`: Stores a pointer to the first node in the linked list. This attribute is private.

`tail (Node*)`: Stores a pointer to the last node in the linked list. This attribute is also private.

`size (int)`: Stores the number of elements in the linked list. This attribute is private.

Constructors:

`LinkedList()`: The constructor instantiates an empty linked list by setting the head and tail pointers to nullptr and size to 0.

Methods:

`void swapNodes(Node *first, Node *second)`: This method swaps two nodes in the LinkedList. This method is used internally by `sort()`. This method is private.

`void push_back(User* user)`: Appends a new User object to the end of the linked list.

`void push_back(Node* node)`: Appends a node to the end of the linked list.

`void display() const`: Iterates through the linked list and calls the display() method of each User object, specifying the entry # before.

`Node* linearSearchName(const string& name) const`: Uses a linear search algorithm to iterate through the linked list and find a user with a specific name.

`Node* linearSearchPhone(const string& phoneNumber) const`: Uses a linear search algorithm to iterate through the linked list and find a user with a specific phone number.

`void loadFromFile(const string& filename)`: Loads multiple entries from a file and appends them to the end of the linked list.

`void sort()`: Sorts the linked list by the name attribute of the User objects.
`void removeByName(const string& name)`: Removes a User from the linked list based on the name attribute.

Getters and Setters:

```
Node* getHead() const
Node* getTail() const
int getSize() const
void setHead(Node* node)
void setTail(Node* node)
void setSize(int newSize)
```

PhoneBook Class & main

The PhoneBook class uses a Bi-Directional Linked List of User* objects to provide a user interface for interacting with the phonebook. It implements functionalities such as loading data, adding/removing entries, and searching/sorting entries. main.cpp simply prints a welcome message then calls `runMenu()` in the PhoneBook class.

Attributes:

`users (LinkedList)`: Stores a LinkedList of Person/Business objects. This attribute is private.

Constructors:

`PhoneBook()`: The constructor instantiates a new PhoneBook by setting up an empty LinkedList of users.

Methods:

`void runMenu()`: Sets up the command-line interface so users may interact with the phonebook. Prints out menu options and handles user input, calling appropriate methods based on user input.

Known Issues/Bugs

In this section, current limitations and issues of the Phonebook application are discussed, leaving room for future development & troubleshooting.

No Way to Clear the Phonebook

Issue: The application currently lacks a feature to clear all entries from the phonebook at once. Users must remove each entry manually.

Impact: Increases the effort required to reset the phonebook and can lead to user frustration if they need to quickly clear all data.

Potential Solution: Implement a clear() method in the LinkedList class that resets the head, tail, and size to their initial values and deallocates all nodes to ensure no memory leaks.

Duplicates Are Not Ignored

Issue: The application does not prevent the addition of duplicate entries. As a result, the phonebook may contain multiple identical entries, which can clutter the phonebook and affect phonebook operations.

Impact: Affects the accuracy of the phonebook and can lead to confusion and difficulties in managing entries effectively.

Potential Solution: Modify the addEntry and loadFromFile methods to check for existing entries with the same name and phone number before adding a new entry.

Linear Search Algorithm

Issue: The linearSearchName and linearSearchPhone methods do not account for duplicates and return the first match found. This behavior might not be ideal if multiple entries with the same name or phone number exist.

Impact: Users cannot access subsequent duplicates through search, which limits the usability of the search function when duplicates are present.

Potential Solution: Modify the search functions to return a list of all matching entries, or provide navigation options to browse through all duplicates.

User Input Validation

Issue: User input is not checked for correct format. For example, when adding new users, the input for the type of user (Person or Business) is case-sensitive and does not trim whitespace, requiring exact matches to the strings "person" or "business".

Impact: This can lead to user errors and frustration, particularly if entries are not added because of minor mistakes in typing user input.

Potential Solution: Implement strict validation for all user inputs throughout the application. This includes but is not limited to: trimming whitespace, handling case-sensitivity, etc.