Chapter 46 – Advanced Project Examples

I. Airport: An Example Problem and Solution in C++ with User Defined Classes

Problem Definition

Aircraft regularly takeoff and land on the only runway of a small airport. Of course, only one aircraft at a time may use the runway. If more than one aircraft requests the runway at the same time, all but one aircraft must wait. Aircraft wishing to takeoff wait in the taxi queue. Aircraft wishing to land wait in the landing pattern. The fuel onboard limits the amount of flight-time of an aircraft. As time elapses, the remaining fuel of the aircraft in the landing pattern is consumed. Aircraft in the landing pattern with low fuel can be moved to the front of the landing pattern so that they may land first.

Write a program using C++ that simulates the airport so that the airport flight controller can add aircraft to the taxi queue and landing pattern, and allow aircraft to takeoff and land, and move aircraft with low fuel to the front of the landing pattern. When the simulation finishes the program should report the average time spent by the aircraft waiting in the taxi queue and the landing pattern.

Outline of Problem Solution

1. Create a class aircraft to model the properties of the aircraft needed for the simulation.
2. Test and debug the class aircraft.
3. Create a class aircraftqueue to model the properties of the taxi queue and the landing pattern. Use a dynamically linked lists of objects of class aircraft to store the information in a manner appropriate to both the taxi queue and the landing pattern.
4. Test and debug the class aircraftqueue.
5. Create a function main that contains an object of class aircraftqueue to represent the taxi queue and an object of class aircraftqueue to represent the landing pattern. Function main will contain a loop that causes time to pass for the aircraft in the queue and allows the flight controller to add aircraft to the two queues, allow aircraft at the front of the taxi queue to takeoff, allow aircraft at the front of the landing pattern queue to land, move craft with low fuel to move to the front of the landing pattern queue, and quit the simulation by exiting the loop. When the loop halts, function main then reports on the average time spent in the taxi queue that have taken off and the time spent in the landing queue by aircraft that have landed.
Creating the Class Aircraft in File AIRCRAFT.H

1) Objects of class aircraft must retain certain information in order to be used in the simulation. Each aircraft must record fuel levels and the time spent in the queue. Though not explicitly stated, each aircraft must have an identifier that will make it unique to the airports traffic controller. In addition, since objects of class aircraft will be linked together in a dynamical list to simulate the taxi and landing queues, an aircraft object must be able to store the address of the next aircraft object. The following public and private variables make this possible:

```cpp
#include <iostream>
using namespace std;

class aircraft {
public:
    aircraft * next;   // new
private:
    int id;     // new
    int fuel;    // new
    int time_in_q;   // new
};
```

2) This class will require a minimum of two constructors. The required constructor (constructor with no arguments) is necessary because of the rules of C++. The second constructor will exist to create an aircraft object with a valid identification and fuel level.

```cpp
#include <iostream>
using namespace std;

class aircraft {
public:
    aircraft();    // new
    aircraft(int initid, int initfuel); // new
    aircraft * next;
private:
    int id;
    int fuel;
    int time_in_q;
};
```
3) Other member functions must
   a) return the aircraft identification
   b) reduce the amount of fuel to reflect fuel used in the landing queue
   c) return the amount of fuel available
   d) add to the time spend in a queue to reflect the passage of time
   e) return the current time spent in a queue

   #include <iostream>
   using namespace std;

class aircraft {
   public:
      aircraft();
      aircraft(int initid, int initfuel);
      int returnid();      // new
      void consumefuel();  // new
      int returnfuel();    // new
      void marktime();     // new
      int returntime();    // new
      aircraft * next;
   private:
      int id;
      int fuel;
      int time_in_q;
};

4) One additional function that might be of use is a function to display the status the aircraft.

   #include <iostream>
   using namespace std;

class aircraft {
   public:
      aircraft();
      aircraft(int initid, int initfuel);
      int returnid();
      void consumefuel();
      int returnfuel();
      void marktime();
      int returntime();
      void showcraft();       // new
      aircraft * next;
   private:
      int id;
      int fuel;
      int time_in_q;
};
5) Implementing the constructors: All member variables (public and private) should be initialized in the constructor. The default constructor should initialize these variables because a programmer might use this constructor even though it is not intended to be used.

```cpp
aircraft::aircraft() {
    id = -1;
    fuel = -1;
    time_in_q = -1;
    next = NULL;
}
```

```cpp
aircraft::aircraft(int initid, int initfuel) {
    id = initid;
    fuel = initfuel;
    time_in_q = 0;
    next = NULL;
}
```

6) Implementing the returnid() function:

```cpp
int aircraft::returnid() {
    return id;
}
```

7) Implementing the fuel functions: The consumefuel() function should subtract from the fuel amount only if the amount of fuel remaining is above 0.

```cpp
void aircraft::consumefuel() {
    if (fuel > 0) fuel--;
}
```

```cpp
int aircraft::returnfuel() {
    return fuel;
}
```
8) Implementing the time functions: Incrementing time spent in the queue only if the
time starts at zero eliminates producing invalid time numbers. (See the default
constructor.)

```cpp
void aircraft::marktime() {
    if (time_in_q >= 0) time_in_q++;
}

int aircraft::returntime() {
    return time_in_q;
}
```

9) Implementing the showcraft() function:

```cpp
void aircraft::showcraft() {
    cout << "Aircraft ID: " << id
    << ", Remaining Fuel: " << fuel
    << ", Time in Queue: " << time_in_q
    << endl;
}
```

Testing the Aircraft Class

To test the class aircraft, write a driver that uses all member functions. Here is an
example of such a driver.

```cpp
#include <iostream.h>
#include "aircraft.h"

void main() {
    aircraft jet1;
    aircraft jet2(35, 1000);

    jet1.showcraft();
    jet2.showcraft();
    jet2.marktime();
    jet2.consumefuel();
    cout << "\n\nID: " << jet2.returnid() << endl
    << "Fuel: " << jet2.returnfuel() << endl
    << "Time in Q: " << jet2.returntime() << endl;
}
```
Creating the Class Aircraftqueue

1) At a minimum, aircraftqueue objects will need to know the type of queue that it is representing and be uniquely identifiable. In addition, since a dynamic list of class aircraft will be used as the representation of the queue, the aircraftqueue objects will need to have front and back pointers to potentially point to objects of class aircraft.

```cpp
#include <iostream>
#include "aircraft.h"
using namespace std;

class aircraftqueue {
public:
private:
    aircraft *front, *back;
    char qtype;
};
```

2) The constructors: Aircraftqueue will need a constructor that will initialize the queue and set the queue identifier or type. A default constructor is also required, but is not used. Class aircraftqueue also requires a destructor as the data structure used is a linked list.

```cpp
#include <iostream>
#include "aircraft.h"
using namespace std;

class aircraftqueue {
public:
    aircraftqueue(); //new
    aircraftqueue(char initqtype); //new
    ~aircraftqueue(); //new
private:
    aircraft *front, *back;
    char qtype;
};
```
3) The most minimal set of member functions for class aircraft would allow craft to be added to the queue, craft removed from the queue, time in the queue to be incremented, fuel used to be decremented, and the status of the craft in queue to be reported.

```cpp
#include <iostream>
#include "aircraft.h"
using namespace std;

class aircraftqueue {
public:
    aircraftqueue();
    aircraftqueue(char initqtype);
    ~aircraftqueue();
    void addcraft(int id, int fuel); //new
    void removecraft(); //new
    void reducefuel_addtime(); //new
    void show(); //new
private:
    aircraft *front, *back;
    char qtype;
};
```
4) A function to report on the average time craft spend in the queue would allow the air controller to determine how many plane can use the airport safely at any given time. To report on average time in the queue, the class aircraftqueue must include member variables for the total time of all craft spent and the number of craft. In addition, means to allow the air controller to move a craft low on fuel to the front of the queue would be a valuable safety aid.

```cpp
#include <iostream>
#include "aircraft.h"
using namespace std;

class aircraftqueue {
public:
    aircraftqueue();
    aircraftqueue(char initqtype);
    ~aircraftqueue();
    void addcraft(int id, int fuel);
    void removecraft();
    void reducefuel_addtime();
    void show();
    void timereport();   //new
    void lowfuel();   //new
private:
    aircraft *front, *back;
    char qtype;
    int totaltime, totalcraft;  //new
};
```

5) Implementing the Constructors: *Note that a queue type of ‘T’ will indicate the taxi queue. All other types will be landing patterns.*

```cpp
aircraftqueue::aircraftqueue() {
    front = NULL;
    back = NULL;
    qtype = 'T';
    totaltime = 0;
    totalcraft = 0;
}

aircraftqueue::aircraftqueue(char initqtype) {
    front = NULL;
    back = NULL;
    qtype = initqtype;
    totaltime = 0;
    totalcraft = 0;
}```
6) Implementing the distructor:

```cpp
aircraftqueue::~aircraftqueue() {
    aircraft *temp;
    while (front != NULL) {
        temp = front;
        front = front->next;
        delete temp;
    }
}
```

7) Implementing adding craft to the queue back and removing craft from the queue front:

```cpp
void aircraftqueue::addcraft(int id, int fuel) {
    aircraft *temp;
    temp = new aircraft(id, fuel);
    if (front == NULL)  front = temp;
    else back->next = temp;
    back = temp;
}

void aircraftqueue::removecraft() {
    if (front != NULL) {
        totaltime += front->returntime();
        totalcraft++;
        aircraft *temp;
        temp = front;
        front = front->next;
        delete temp;
    }
}
```
8) Implementing the reporting function:

```c++
void aircraftqueue::show() {
    cout << "\n======== ";
    if (qtype == 'T') cout << "Taxi Queue ";
    else cout << "Landing Pattern " << qtype;
    cout << " =========\n";
    aircraft *current;
    current = front;
    while (current != NULL) {
        current->showcraft();
        current = current->next;
    }
}
```

9) Implementing the function to reduce fuel and increment time to reflect the time in the queue. *Note that queues with a type of T will not have fuel reduced.*

```c++
void aircraftqueue::reducefuel_addtime() {
    aircraft *current;
    current = front;
    while (current != NULL) {
        current->marktime();
        if (qtype != 'T') current->consumefuel();
        current = current->next;
    }
}
```

10) Implement the function to report average time spent in the queue:

```c++
void aircraftqueue::timereport() {
    cout << "Average time spent in ";
    if (qtype != 'T') cout << "Landing " << qtype << " ";
    else cout << "Taxi ";
    cout << "queue: ";
    if (totalcraft == 0) cout << 0 << endl;
    else cout << totaltime / totalcraft << endl;
}
```
11) Implement the function to move craft with low fuel to the front of the queue if the queue is a landing queue: *Note that the function will put the craft with the lowest fuel levels (10 or under) in the front of the queue in order with the craft with the lowest fuel first. To do this, the queue is treated as a simple linked list with pointer front serving as the pointer first. The one concession to the data structures existence as a queue is the updating of pointer back when the last aircraft in the queue is one with lowest fuel.*

```c
void aircraftqueue::lowfuel() {
    aircraft *current, *previous, *min, *previoustomin;
    if (qtype != 'T' && front != NULL && front->next != NULL) {
        previous = front;
        current = front->next;
        previoustomin = NULL;
        min = front;
        while (current != NULL) {
            if (min->returnfuel() > current->returnfuel()) {
                previoustomin = previous;
                min = current;
            }
            previous = current;
            current = current->next;
        }
        if (min != front) {
            previoustomin->next = min->next;
            min->next = front;
            front = min;
            if (min == back) back = previoustomin;
        }
    }
}
```
Testing the Class Aircraftqueue

To test the class aircraftqueue, write a driver that uses all member functions. Here is an example of such a driver.

```cpp
#include <iostream>
#include "aircraftqueue.h"
using namespace std;

void main() {
    aircraftqueue landing('L');
    aircraftqueue taxi("T");

    landing.addcraft(1,100);
    landing.addcraft(2,12);
    landing.addcraft(3,300);
    landing.addcraft(8,11);

    taxi.addcraft(4,100);
    taxi.addcraft(5,9);
    taxi.addcraft(6,98);
    taxi.addcraft(7,97);

    landing.show();
    taxi.show();

    landing.reducefuel_addtime();
    landing.reducefuel_addtime();
    landing.reducefuel_addtime();
    landing.reducefuel_addtime();
    taxi.reducefuel_addtime();

    landing.lowfuel();
    landing.lowfuel();
    taxi.lowfuel();

    landing.show();
    taxi.show();

    landing.removecraft();
    landing.removecraft();
    landing.removecraft();
    landing.timereport();
    taxi.timereport();
}
```
The Final Program

Here is an example of the final function main of the airport program.

```cpp
#include <iostream>
#include "aircraftqueue.h"
using namespace std;

char userchoice() {
    char choice;
    do {
        cout << "(1+taxi, 2+landing, 3-takeoff, 4-landing, "
             << " 5:low fuel, q-quit) ";
        cin >> choice;
    } while (choice < '1' || (choice > '5' && choice != 'q'));
    return choice;
}

void clrscr() {
    for (int x=0; x<25; x++) cout << endl;
}

void main() {
    aircraftqueue landing('1');
    aircraftqueue taxi("T");
    int id, fuel;

    clrscr();
    taxi.show();
    landing.show();

    char choice = userchoice();
    while (choice != 'q') {
        switch(choice) {
            case '1' :
                cout << "\nEnter aircraft id: ";
                cin >> id;
                taxi.addcraft(id,0);
                break;
```
```
case '2' :
        cout.flush();
        cout << "\nEnter aircraft id: ";
        cin >> id;
        cout << "Enter aircraft fuel: ";
        cin >> fuel;
        landing.addcraft(id,fuel);
        break;

    case '3' :
        taxi.removecraft();
        break;

    case '4' :
        landing.removecraft();
        break;

    case '5' :
        landing.lowfuel();
    }

    taxi.reducefuel_addtime();
    landing.reducefuel_addtime();

    clrscr();
    taxi.show();
    landing.show();
    choice = userchoice();

    cout << endl;
    taxi.timereport();
    landing.timereport();
}
II. Phone Book: An Example Problem and Solution in C++ with User Defined Classes

As part of the process of learning about files and arrays, students are often assigned to create a phone book. This example problem is the second of two such phone books. This first phone book is dynamic in size and as such requires more complex destructors and constructors as part of its development because the data structure used. (The first phone book was limited in size and did require complex destructors and constructors to handle the static data structure used.)

Problem Definition

An automated phone book program loads all previous entries from a file upon startup, allows the user to display these entries, add entries, and remove entries. Upon closing, the program writes the changes back to the file.

Write a C++ program, with user defined classes, that performs all functions of the automated phone book described above.

Outline of Problem Solution

6. Create a class line to model a single line of free text in the notes section of an entry in a phone book.

7. Test and debug the class line.

8. Create a class card to model a single entry in a phone book. Class line will be used to create the free text notes section of the entry.

9. Test and debug the class card.

10. Create a class phonelist that uses the class card to model a dynamic phone book.

11. Test and debug the class phonelist.

12. Create a function main that contains an object of class phonelist. Use the member functions of the class phonelist object to allow the user to select all of the operations of the phone book until the user decides to stop the program.
**Creating the Class line in the file line.h**

1. Objects of class line must be able to store a line of free text in the notes section of an entry in a phone book. The line of text will be a dynamic character variable.

```cpp
#include <iostream>
#include <string.h>
using namespace std;

class line {
    public:
        private:
            char *l; // new

};
```

2. Since the data structure used is dynamic, objects of class line will need both a constructor and a destructor.

```cpp
class line {
    public:
        line(); // new
        ~line(); // new
    private:
        char *l;

};
```

3. The member functions of class line must be able to accept the line from the keyboard, as a character string, and as an object of class line. In addition, a member function is needed to return the contents of the line as a character string.

```cpp
class line {
    public:
        line();
        ~line();
        void cinline(); // new
        void putline(char initline[ ]); // new
        void putline(line initline); // new
        char *returnline(); // new
    private:
        char *l;

};
```
4. Implementation of the constructor and destructor: *(Note that the size of 50 characters in the line is an arbitrary choice and could be substituted for any constant or named constant.)*

```cpp
    line::line() {
        l = new char[50];
        strcpy(l, "                                                  ");
    }

    line::~line() {
        delete [] l;
    }
```

5. Implementation of member function cinline: The istream function getline is used to allow the user to enter a complete line of text.

```cpp
    void line::cinline() {
        cin.getline(l,50,\n);  
    }
```

6. Implementation of the overloaded member functions putline:

```cpp
    void line::putline(char initline[]) {
        strcpy(l, initline);
    }

    void line::putline(line initline) {
        char *temp;
        temp = initline.returnline();
        int i = (int)(strlen(temp));
        for (int x=0; x<i; x++) {
            l[x] = temp[x];
        }
        if (i < 50) l[i] = '\0';  // '\0' is the line character
    }
```

7. Implementation of the member function returnline:

```cpp
    char *line::returnline() {
        return l;
    }
```
Testing the Class line

```cpp
#include <iostream>
#include <fstream>
#include <string.h>
#include "line.h"
using namespace std;

void main() {
    line * ln, * aln;
    ln = new line[3];
    aln = new line();

    aln->putline("birds gotta fly,");

    ln[0].putline("fish gotta swim.");
    ln[1].putline(aln->returnline());
    ln[2].cinline();

    for (int x=0; x<3; x++) cout << ln[x].returnline() << endl;
}
```

Creating the Class card in the file card.h

1. The class card uses the class line. In addition, each object of class card must store a name, phone number, and several lines of notes. Since objects of card will be used in a dynamic data structure and the information is stored in strings, all data variables in card will be pointers. The named constant LINEMAX will serve as the control of the number of objects of class line.

```cpp
#include <iostream> // may need to be written <string.h>
#include "line.h" // new
using namespace std;

const int LINEMAX = 5; // new

class card {
    public:
    private:
        char *fname; // new
        char *lname; // new
        char *phone; // new
        line *note; // new
};
```
2. Class card requires constructors that initialize card objects to an empty state and initialize card objects to a state with data such as is required by a copy operation. Since the data objects are dynamic, a destructor is required.

```cpp
class card {
public:
    card(); // new
    card(char fn[], char ln[], char ph[], line nt[]); // new
    ~card(); // new
private:
    char *fname;
    char *lname;
    char *phone;
    line *note;
};
```

3. The member functions of class card must collect user input and be able to return all items of information. The ubiquitous and handy display function is a nice addition to the class. Also, a function that displays name and phone information only would make an object of class card even more flexible.

```cpp
class card {
public:
    card();
    card(char fn[], char ln[], char ph[], line nt[]);
    ~card();
    void input(); // new
    void displayphone(); // new
    void displayall(); // new
    char * returnlename(); // new
    char * returnfname(); // new
    char * returnphone(); // new
    char * returnnote(int index); // new
private:
    char *fname;
    char *lname;
    char *phone;
    line *note;
};
```
4. Implementing the constructors and destructor:

```cpp
card::card() {
    fname = new char[15];
    lname = new char[15];
    phone = new char[15];
    note = new line[LINEMAX];
    strcpy(fname, "               ");
    strcpy(lname, "               ");
    strcpy(phone, "               ");
}

card::card(char fn[], char ln[], char ph[], line nt[]) {
    fname = new char[15];
    lname = new char[15];
    phone = new char[15];
    note = new line[LINEMAX];
    strcpy(fname, fn);
    strcpy(lname, ln);
    strcpy(phone, ph);
    for (int x=0; x< LINEMAX; x++) note[x].putline(nt[x]);
}

card::~card() {
    delete []& fname;
    delete []& lname;
    delete []& phone;
    delete []& note;
}
```
5. Implementing the input member function:

```cpp
void card::input() {
    cout << "Enter the last name: ";
    cin.getline(lname, 15, '\n');
    cout << "Enter the first name: ";
    cin.getline(fname, 15, '\n');
    cout << "Enter the phone number: ";
    cin.getline(phone, 15, '\n');
    cout << endl << "Enter up to five lines of notes. "
    << "A period on a line by itself stops." << endl;
    cout << "=" << endl;
    for (int y = 0; y < LINEMAX; y++) {
        cout << "line " << y + 1 << ": " << note[y].cinline();
        if (strcmp(note[y].returnline(), ".") == 0) break;
    }
}
```

6. Implementing the display member functions:

```cpp
void card::displayphone() {
    cout << lname << ", " << fname << ": " << phone << endl;
}

void card::displayall() {
    displayphone();
    cout << "=" << endl;
    for (int x = 0; x < LINEMAX; x++) {
        if (strcmp(note[x].returnline(), ".") != 0)
            cout << ": " << note[x].returnline() << endl;
        else break;
    }
```
7. Implementing the return functions:  *(Note that the returnnote function accepts an index and returns one line of the notes. The programmer-card user must be aware of and use the limit named constant LINEMAX.)*

```cpp
char * card::returnfname() {
    return fname;
}
char * card::returnphone() {
    return phone;
}
char * card::returnnote(int index) {
    return note[index].returnline();
}
```

**Testing the Class card**

Notice the card object acard is declared as a pointer to an object of class card. This is necessary for the destructors and to work correctly in class line and, hence, class card.

```cpp
#include <iostream>
#include "card.h"
using namespace std;

void main() {
    card * acard;
    acard = new card();
    acard->input();
    acard->displayphone();
    acard->displayall();
    cout << acard->returnfname() << "  " << acard->returnphone() << endl;
    for (int x=0; x<5; x++) cout  << acard->returnnote(x) << endl;
}
```
Creating the Class phonelist in the file phonelist.h

1. Class phonelist will work with a dynamic array. To do this, a type of pointer to card objects, called cardpointer, is created. This will allow a dynamic array of pointers to objects of class card to be created. As with all dynamic arrays, variables to track the current size of the array and the number of elements of the array in use are required.

```cpp
#include <iostream>
#include <fstream>
#include <string.h>
#include "card.h"
using namespace std;

typedef card * cardpointer; // new

class phonelist {
public:
    phonelist(); // new
    ~phonelist(); // new
private:
    cardpointer * list; // new
    int max; // new
    int length; // new
};
```

2. The default constructor will load stored information from a file into the array. The destructor will return the information to the file at the end of the program or when the phonelist object passes from scope.

```cpp
typedef card * cardpointer;

class phonelist {
public:
    phonelist(); // new
    ~phonelist(); // new
private:
    cardpointer * list;
    int max;
    int length;
};
```
3. The private functions expand and shrink are required for the management of a
dynamic array. The private function find will be used to locate a specific
element card and return its index. Function find is a private function because
the programmer-user of class phonelist should not have direct access to the
elements of the data structure.

```cpp
class phonelist {
public:
    phonelist();
    ~phonelist();
private:
    void expand();  // new
    void shrink();  // new
    int find(char lnametofind[], char fnametofind[]); // new
    cardpointer * list;
    int max;
    int length;
};
```

4. The member functions handle the process of adding new entries to the phone book,
removing entries from the phone book, and displaying the information contained in the
phone book. While only one add function is needed, two remove functions are
needed. One removes by name, the other by index. There are three display functions.
One displays the list without the note information, the second displays one card given
a name with the note information, and the third displays one card given its index with
the note information.

```cpp
class phonelist {
public:
    phonelist();
    ~phonelist();
    void add();  // new
    void removebyindex(); // new
    void removebyname(); // new
    void displaylist(); // new
    void displayonebyindex(); // new
    void displayonebyname(); // new
private:
    void expand();
    void shrink();
    int find(char lnametofind[], char fnametofind[]);
    cardpointer * list;
    int max;
    int length;
};
```
5. Implementing the constructor and destructor:

```cpp
phonelist::phonelist() {
    ifstream infile;
    infile.open("list.dat", ios::in);
    max = 2;
    length = 0;
    list = new cardpointer[max];
    char fn[15];
    char ln[15];
    char ph[15];
    line *nt;
    char temp[50];
    infile.getline(ln,15,'\n');
    while (!infile.eof() && strlen(ln) > 0) {
        nt = new line[5];
        infile.getline(fn,15,'\n');
        infile.getline(ph,15,'\n');
        for (int x=0; x<5; x++) {
            infile.getline(temp,50,'\n');
            nt[x].putline(temp);
        }
        list[length] = new card(fn, ln, ph, nt);
        length++;
        if (length >= max) expand();
    }
    infile.getline(ln,15,'\n');
}

infile.close();
}
```

```cpp
phonelist::~phonelist() {
    ofstream outfile;
    outfile.open("list.dat", ios::out);
    char *l;
    for (int x=0; x<length; x++) {
        outfile << list[x]->return lname() << endl
               << list[x]->returnfname() << endl
               << list[x]->return phone() << endl;
        for (int y=0; y<5; y++) {
            l = list[x]->returnnote(y);
            outfile << l << endl;
        }
    }
    outfile.close();
    delete [ ] list;
}
```
6. Implementing the add member function:

```c++
void phonelist::add() {
    list[length] = new card();
    list[length]->input();
    length++;
    if (length >= max) expand();
}
```

7. Implementing the removebyindex member function:

```c++
void phonelist::removebyindex() {
    cout << "Enter the index number of the card to remove: ";
    int i;
    cin >> i;
    i--;
    if (i > -1 && i < length) {
        for (int x=i; x<length-1; x++)
            list[x]=list[x+1];
        length--;
        if (max-length > 2) shrink();
    }
}
```

8. Implementing the removebyname member function:

```c++
void phonelist::removebyname() {
    char fn[15];
    char ln[15];
    cout << "Enter last name: ";
    cin >> ln;
    cout << "Enter first name: ";
    cin >> fn;
    int location = find(ln,fn);
    if (location > -1) {
        for (int x=location; x<length-1; x++)
            list[x]=list[x+1];
        length--;
        if (max-length > 2) shrink();
    } else cout << "*** Not Found ***" << endl;
}
```
9. Implementing the display member functions:

```cpp
    void phonelist::displaylist() {
        cout << endl;
        for (int x=0; x<length; x++) {
            cout << x+1 << " ": ";
            list[x]->displayphone();
        }
        cout << endl;
    }
```

```cpp
    void phonelist::displayonebyname() {
        char fn[15];
        char ln[15];
        cout << "Enter last name: ";
        cin >> ln;
        cout << "Enter first name: ";
        cin >> fn;
        int location = find(ln, fn);
        if (location > -1) list[location]->displayall();
        else cout << "*** Not Found ***" << endl;
    }
```

```cpp
    void phonelist::displayonebyindex() {
        int i;
        cout << "Enter index: ";
        cin >> i;
        i--;
        if (i > -1 && i < length) list[i]->displayall();
        else cout << "*** Index Not Found ***" << endl;
    }
```

10. Implementing the expand member function:

```cpp
    void phonelist::expand() {
        cardpointer *temp;
        max += 2;
        temp = new cardpointer[max];
        for (int x=0; x<length; x++) {
            temp[x] = new card();
            temp[x] = list[x];
        }
        delete [] list;
        list = temp;
    }
```
11. Implementing the shrink member function:

```cpp
void phonelist::shrink() {
    cardpointer *temp;
    if (max > 2) {
        max -= 2;
        temp = new cardpointer[max];
        for (int x=0; x<length; x++) {
            temp[x] = new card();
            temp[x] = list[x];
        }
        delete [] list;
        list = temp;
    }
}
```

12. Implementing the find member function:

```cpp
int phonelist::find(char lnametofind[ ], char fnametofind[ ]) {
    for (int x=0; x<length; x++)
        if (strcmp(list[x]->returnlname(), lnametofind) == 0 &&
            strcmp(list[x]->returnfname(), fnametofind) == 0) break;
    if (x >= length) return -1;
    else return x;
    return 0;
}
```
Testing the Class phonelist

This program must be run twice to test all code.

```cpp
#include "phonelist.h"
using namespace std;

void main() {
    phonelist thelist;
    for (int x=0; x<5; x++) {
        thelist.add();
        thelist.displaylist();
    }
    thelist.displaylist();
    thelist.displayonebyname();
    thelist.displayonebyindex();
    thelist.removebyname();
    thelist.displaylist();
    thelist.removebyindex();
    thelist.removebyname();
    thelist.displaylist();
}
An Example Program using Class phonelist

#include <iostream>
#include "phonelist.h"
using namespace std;

void main() {
    phonelist pb;
    char *choice;
    choice = new char[15];

    cout << "\n1-Add 2-DisplayList 3-DisplayByIndex 4-DisplayByName"
    << endl << "5-RemoveByIndex 6-RemoveByName Q-Quit: ";
    cin.getline(choice,15,\"\n\")
    while (choice[0] != 'q' && choice[0] != 'Q') {
        switch(choice[0]) {
            case '1': pb.add();
                break;
            case '2': pb.displaylist();
                break;
            case '3': pb.displayonebyindex();
                break;
            case '4': pb.displayonebyname();
                break;
            case '5': pb.removebyindex();
                break;
            case '6': pb.removebyname();
                break;
            case 'qq':
            case 'Q': break;
            default:
                cout << "************ Not a valid entry! **************" << endl;
                break;
        }
    }
    cout << "\n1-Add 2-DisplayList 3-DisplayByIndex "
    << "4-DisplayByName" << endl
    << "5-RemoveByIndex 6-RemoveByName Q-Quit: ";
    cin.getline(choice,15,\"\n\")
}