```cpp
#ifndef __LinkList__Implementation__        // For those using Visual Studio, lets try to avoid LNK2004 errors...
#define __LinkList__Implementation__

#define PlaneSize 80        // Small commercial plane [Boeing?]

#pragma region Inclusions
#include <iostream>         // Input and Output
#include <cstring>          // Used for the 'NULL' keyword, despite that it
                            //  is '0' or 'OL', but keep this for convention
                            //  sakes.
#include <cstdlib>
#include "LinkedList.h"     // Implementation for Link List.
#pragma endregion



// Instructions
// ===============================================
// Documentation:
//  Provide instructions to the user as to what this program is doing and how to use it.
// ===============================================
void Reservation::Instructions()
{
    std::cout << "WELCOME TO BLUE-SKY AIRLINES!" << std::endl
        << "Where we are dedicated to suckering you out of your money!" << std::endl
        << "---------------------------------------------" << std::endl
        << "This is a simulation program that provides some flexibility with" << std::endl
        << "adjusting the link list and minor management tools.  With this program," << std::endl
        << "it is possible to generate a reasonable size customer list and perform" << std::endl
        << "maintenance as needed.  Such maintenance could be removing a customer," << std::endl
        << "updating the customer's information, update reservations, change seats," << std::endl
        << "and even revise customer's in-flight meals!  Use the menu to navigate" << std::endl
        << "through this program and try to crash it!  I dare you!" << std::endl
        << "Looks like I picked the wrong week to quit sniffing glue. -Steve McCroskey"  // reference: https://youtu.be/VmW-ScmGRMA
        << std::endl << std::endl;
} // Instructions()



// Main Menu
// ===============================================
// Documentation:
```

```
//  This function will provide a list of functionality that is available to the end-user.
// ===============================================
void Reservation::MainMenu()
{
    // The border for the main menu screen
    std::cout << "Select an option:" << std::endl
        << "====================" << std::endl;

    // Main menu options
    std::cout << "[1] - Automatically Generate a Customer List" << std::endl
        << "[2] - Manually add a new customer" << std::endl
        << "[3] - Print all customer information" << std::endl
        << "[4] - Search for a passenger" << std::endl
        << "[5] - Update passenger information" << std::endl
        << "[6] - Remove passenger from list" << std::endl
        << "[7] - Check in passenger" << std::endl
        << "[8] - Print check in report" << std::endl
        << "[9] - Print menu report" << std::endl
        << "[0] - Sort passenger list" << std::endl
        << "[X] - Exit" << std::endl << std::endl;
} // MainMenu()



// Evaluate and Run
// ===============================================
// Documentation:
//  This will allow the user to perform the requested action, if STDIN is legal.
// -----------------------------------------------
// Parameters:
//  STDIN [char]
//      User's requested action
//  head [Node**]
//      Primary list that contains user info
// ===============================================
void Reservation::EvaluateAndRun(char STDIN)
{
    Node* nullityNode = NULL;           // If the user chooses option '3', then
                                        // this pointer will be available as
                                        // the Print_Passenger_List() function requires
                                        // a node datatype, we will send
                                        // an empty node but will not be
                                        // used during the execution.
    switch (STDIN)
    {
    case '1':   // Automatically generate customer list
        Autofill_List();
        break;
    case '2':   // Manually input customer
        ManualCustomerAdd();
```

```cpp
        break;
    case '3':   // Print all customers [primary pointer]
        Print_Passenger_List(nullityNode, false);
        break;
    case '4':   // Search for passenger
        FindPrintPassenger();
        break;
    case '5':   // Update passenger information
        UpdatePassengerInformation();
        break;
    case '6':   // Thrash passenger node
        delete_node();
        break;
    case '7':   // Passenger Checking
        CheckInPassenger();
        break;
    case '8':
        Print_CheckIn_List();   //print check in report
        break;
    case '9':
        Print_Meal_List();      //print meal choice report
        break;
    case '0':
        Sort();     //sort list
        break;
    case 'X':   // Quietly pass through; exit
        break;
    case 'x':   // Quietly pass through; exit
        break;
    default:    // Bad Input
        std::cout << "Incorrect option!" << std::endl << std::endl;
        break;
    } // switch

    ClearBuffer();  // Provide spacing after evaluation
} // EvaluateAndRun()



// Prompt User [Main Menu]
// ===============================================
// Documentation:
//  Capture user input and return it as an integer.  By convention, use a python'ish input prompt.
//  From personal experience, this is more clear that the program is 'wanting' something from the user.
// -----------------------------------------------
// Output:
//  STDIN [Char]
//      Return the STDIN from the end-user.
// ===============================================
char Reservation::PromptUser_MainMenu()
```

```cpp
{
    char inputCapture = '-';        // If in case - to avoid bugs, use a default value.
    std::cout << ">>>>> ";          // The python'ish prompt
    std::cin >> inputCapture;       // Capture the input

    return inputCapture;            // Return the value.
} // PromptUser_MainMenu()



// Clear Buffer
// ===============================================
// This function, despite minimal, will try to make the buffer easier to read for the end user.
// ===============================================
void Reservation::ClearBuffer()
{
    std::cout << std::endl
        << std::endl
        << std::endl
        << std::endl
        << std::endl
        << std::endl;
} // ClearBuffer()



// Reservation Constructor
// ===============================================
// Documentation:
//  This function will initialize the head pointer to NULL.
// ===============================================
Reservation::Reservation()
{
    head = NULL;
};



// Print Passenger List
// ===============================================
// Documentation:
//  This function will output all of the information within the link list.
// -----------------------------------------------
// Parameters:
//  listIndex [Node*]
//      This will take any valid link list, but this is only
//      usable for single indexes, not a list!  With that,
//      this parameter works with the 'runOnce' parameter and must be
//      set to 'true' instead of false.  IIF false, then listIndex
//      ignored.  Otherwise, when true, listIndex will be printed.
//  runOnce [bool]
//      When true, this will allow 'listIndex' to be printed on the screen.
//      This parameter adjusts this function to only print ONE node on the
//      terminal buffer and no more.  If this variable is 'false', then
//      the standard list takes precedence.
// ===============================================
void Reservation::Print_Passenger_List(Node* listIndex, bool runOnce = false)
{
    // IIF runOnce is true then use the listIndex, otherwise use head.
    Node *temp = runOnce ? listIndex : head;

    // If head is empty, present an error message and leave this function.
    if (temp == NULL)
    {
        std::cout << "<!> ERROR <!>: No entries within the list are present!" << std::endl
            << "Nothing to print nor report!  Please generate or create a new list." << std::endl;
        return;
    } // if

    int indexCounter = 1; // This will be helpful to know what index the client is located within the list.

    // Output the available list with the information required.
    //  IIF runOnce is true, then we will only output one index.
    //  else, when runOnce is false, we will output the entire list available.
    while (temp != NULL && ((runOnce && indexCounter < 2) || !runOnce))
    {
        std::cout << "Index number: " << indexCounter << std::endl
            << "Passenger ID: " << temp->passengerID << std::endl
            << "Passenger Name: " << temp->nameLast << ", " << temp->nameFirst << std::endl
            << "Telephone Number: " << temp->telephoneNum << std::endl
            << "Reservation Number: " << temp->reservationNum << std::endl
            << "Seat on the plane: " << temp->seatNum << std::endl
            << "Preferred Meal Plan: " << temp->mealType << std::endl << std::endl;

        temp = temp->next;  // Move to the next node
        indexCounter++;
    } // while
} // Print_Passenger_List()



// Insert New Node Entry
// ===============================================
// Documentation:
//  This function will forward the new node (or data) onto
//  the primary link list (or pointer)
// -----------------------------------------------
// Parameters:
//  head [Node]
//      The primary list that is to be updated.
//  NewEntry [Node]
//      New information that is to be merged or imported into the 'head'.
// ===============================================
void Reservation::InsertNode(Node* newEntry)
{
    // Disallow any new entries if the plane is full.
    if (PlaneSize <= ListSize())
    {
        std::cout << "<!> ERROR <!>: Unable to add new entry!" << std::endl
            << "The plane is currently full and we have yet to add seats on the wings." << std::endl
            << "Please consider deleting a passenger with no refunds." << std::endl;
            return;
    } // if

    if (head == NULL)           // if the current list contains no entries
        head = newEntry;        //  then immediately add the temp entry to the list.
    else
    {
        newEntry->next = head;  // Import the primary list to the temp entry.
        head = newEntry;        // Export all of the entries from the temp
                                // list to the primary list.
    }
} // InsertNode()



// Create a New Node
// ===============================================
// Documentation:
//  Generate a new entry to be imported into the primary list.
//  This function will merely make sure that all information is
//   present before allowing the new node to be official in the
//   primary list.
// -----------------------------------------------
// Parameters:
//  head [Node]
//      The primary list that will soon include the new node entry.
//  nameFirst [string]
//      Client's first name
//  nameLast [string]
//      Client's last name
//  passengerID [int]
//      Client's Passenger ID
//  reservationNum [int]
//      Client's reservation ID
//  telephoneNum [int]
//      Client's telephone number
//  seatNum [int]
//      Client's seat number
//  mealType [string]
//      Client's requested meal.
// ===============================================
void Reservation::CreateNewNode(std::string nameFirst, std::string nameLast, int passengerID, int reservationNum, int telephoneNum, int seatNum, std::string mealType)
{
    // Create a new node to store the new information
    Node* newEntry = new Node;


    // Generate the incoming data into the newly created node
    newEntry->nameFirst = nameFirst;
    newEntry->nameLast = nameLast;
    newEntry->passengerID = passengerID;
    newEntry->reservationNum = reservationNum;
    newEntry->telephoneNum = telephoneNum;
    newEntry->seatNum = seatNum;
    newEntry->mealType = mealType;
    newEntry->checkedIn = false;

    newEntry->next = NULL;
    // Import the data into the primary list
    InsertNode(newEntry);
} // CreateNewNode()



// Autofill List [Numbers - Dependency]
// ===============================================
// Documentation:
//  This function will provide an easier way to manage
//   randomized numbers for populating the passenger
//   information.
// -----------------------------------------------
// Parameters:
//  key [int]
//      This will allow us to provide a random number
//      based on what attribute we are requesting.
//      Acceptable Key Values:
//      0 = Passenger ID
//      1 = Reservation Number
//      2 = Telephone Number
// -----------------------------------------------
// Output:
//  Returns a specific, though randomized, integer based on the
//   key used.
// ===============================================
int Reservation::Autofill_List_Numbers(int key)
```

```cpp
{
    // Initializations
    // -------------------------
    // We will be using these as a way to assure that all values are unique - dynamically at runtime.
    // Best approach?  No.  Best on performance?  Absolutely not - at least if worst case occurs.
    int newRandomValue;         // This will be used to inspect the rand() value is unique
    bool uniqueFound;           // Will be used in cooperation with the loops.
    Node* temp = head;          // The list that we will be searching
    Node* nullityNode = NULL;   // A temporary list that wont be used, but required for the search() function.
    // -------------------------

    // Determine the requested type to return
    switch (key)
    {
    case 0:         // Passenger ID
        do          // Keep scanning until we find unique key
        {           // It is possible for processing lag phenomenon
            newRandomValue = rand() % 9999 + 1;
            uniqueFound = Search(&temp, &nullityNode, 2, " ", newRandomValue) ? false : true;
        } while (!uniqueFound);
        return newRandomValue;  // Return the unique key
        break;
    case 1:         // Reservation Number
        do          // Keep scanning until we find unique key
        {           // It is possible for processing lag phenomenon
            newRandomValue = rand() % 999 + 100;
            uniqueFound = Search(&temp, &nullityNode, 3, " ", newRandomValue) ? false : true;
        } while (!uniqueFound);
        return newRandomValue;  // Return the unique key
        break;
    case 2:         // Telephone Numbers
        do          // Keep scanning until we find unique key
        {           // It is possible for processing lag phenomenon
            newRandomValue = rand() % 8999999999 + 1000000000;
            uniqueFound = Search(&temp, &nullityNode, 4, " ", newRandomValue) ? false : true;
        } while (!uniqueFound);
        return newRandomValue;  // Return the unique key
        break;
    default:
        // Error; access violation occurred.
        return -255;
        break;
    } // switch
} // Autofill_List_Numbers()

```

```cpp


// Autofill List [Meal Choice - Dependency]
// ===============================================
// Documentation:
//  This function will provide a randomized choice of
//  the <del>horrible</del> best foods available
//  in Blue-Sky Air Lines!
//
//  The list is inspired by Indiana Jones Temple of Doom
// -----------------------------------------------
// Output:
//  string
//      A randomized string output of the desired
//      food or meal that the passenger is willing
//      order.
// ===============================================
std::string Reservation::Autofill_List_MealChoice()
{
    // Randomly pick a number that will
    // allow us to choose which meal the
    // passenger is going to eat.
    int choice = rand() % 4;

    // Evaluate the choice and return the appropriate value.
    switch (choice)
    {
    case 0:
        return "Monkey Brains";
        break;
    case 1:
        return "Tuna Eyeballs";
        break;
    case 2:
        return "Raw Octopus";
        break;
    case 3:
        return "Fish"; // https://youtu.be/rQbj9uvYL8I
        break;
    default:
        return "Expired Peanuts";
        break;
    } // switch
} // Autofill_List_MealChoice()



// Find Available Seat
// ===============================================
// Documentation:
//  This function will merely provide the next available seat.
// -----------------------------------------------
// Output:
//  Returns the seat ID that is available.
//  if the seat is not available or is not unique,
//  then a value of -255 is returned, signaling an error.
//   however, if the seat is available - then the
//   seat requested will be returned to confirm.
// ===============================================
int Reservation::GetSeatAvailable(int requestKey = -255)
{
    // Initializations
    // -------------------------
    // We will be using these as a way to assure that all values are unique - dynamically at runtime.
    // Best approach?  No.  Best on performance?  Absolutely not - at least if worst case occurs.
    int newRandomValue;         // This will be used to inspect the rand() value is unique
    bool uniqueFound;           // Will be used in cooperation with the loops.
    Node* temp = head;          // The list that we will be searching
    Node* nullityNode = NULL;   // A temporary list that wont be used, but required for the search() function.
    // -------------------------

    // Check to see if the user is requesting a new seat
    if (requestKey != -255)
    {   // Check if the seat is available; end-user manual request
        uniqueFound = Search(&temp, &nullityNode, 5, " ", requestKey) ? false : true;
        newRandomValue = uniqueFound ? requestKey : -255;
        return newRandomValue;
    }
    else            // This will be used for auto fill functionality
        do          // Keep scanning until we find unique key
        {           // It is possible for processing lag phenomenon
            newRandomValue = rand() % 100 + 1;
            uniqueFound = Search(&temp, &nullityNode, 5, " ", newRandomValue) ? false : true;
        } while (!uniqueFound);
        return newRandomValue;  // Return the unique key
} // GetSeatAvailable()



// Autofill List
// ===============================================
// Documentation:
//  This function will automatically populate and generate a reasonably
//   sized list.
// Notes:
//   I feel like a tool for using 'babynames.com' to populate
//   with random names....
// -----------------------------------------------
// Parameters:
//  head [Node*]
//      This will take any valid link list.
// ===============================================
void Reservation::Autofill_List()
{
    // __HARD_CODED__
    // Update this algorithm with caution!
    //---
    // To auto-generate, we are going to throw hard-coded values
    // to the link list.  This will greatly allow us to debug or
    // generally work with the list.
    for (int i = 0; i < 20; i++)
    {
        switch (i)
        {
        case 0:
            CreateNewNode(                      // primary list
                        "Fabian",                   // First Name
                        "Nadie",                    // Last Name
                        Autofill_List_Numbers(0),   // Passenger ID
                        Autofill_List_Numbers(1),   // Reservation Num
                        Autofill_List_Numbers(2),   // Telephone Num
                        GetSeatAvailable(),         // Seat Num
                        Autofill_List_MealChoice());// Preferred Meal
            break;
        case 1:
            CreateNewNode(                      // primary list
                        "Ganit",                    // First Name
                        "Ume",                      // Last Name
                        Autofill_List_Numbers(0),   // Passenger ID
                        Autofill_List_Numbers(1),   // Reservation Num
                        Autofill_List_Numbers(2),   // Telephone Num
                        GetSeatAvailable(),         // Seat Num
                        Autofill_List_MealChoice());// Preferred Meal
            break;
        case 2:
            CreateNewNode(                      // primary list
                        "Dan",                      // First Name
                        "Randi",                    // Last Name
                        Autofill_List_Numbers(0),   // Passenger ID
                        Autofill_List_Numbers(1),   // Reservation Num
                        Autofill_List_Numbers(2),   // Telephone Num
                        GetSeatAvailable(),         // Seat Num
                        Autofill_List_MealChoice());// Preferred Meal
            break;
        case 3:
            CreateNewNode(                  // primary list
                        "Reese",                    // First Name
                        "Nafisa",                   // Last Name
                        Autofill_List_Numbers(0),   // Passenger ID
                        Autofill_List_Numbers(1),   // Reservation Num
                        Autofill_List_Numbers(2),   // Telephone Num
                        GetSeatAvailable(),         // Seat Num
                        Autofill_List_MealChoice());// Preferred Meal
            break;
        case 4:
            CreateNewNode(                      // primary list
                        "Nina",                     // First Name
                        "Albany",                   // Last Name
                        Autofill_List_Numbers(0),   // Passenger ID
                        Autofill_List_Numbers(1),   // Reservation Num
                        Autofill_List_Numbers(2),   // Telephone Num
                        GetSeatAvailable(),         // Seat Num
                        Autofill_List_MealChoice());// Preferred Meal
            break;
        case 5:
            CreateNewNode(                      // primary list
                        "Alexis",                   // First Name
                        "Wayne",                    // Last Name
                        Autofill_List_Numbers(0),   // Passenger ID
                        Autofill_List_Numbers(1),   // Reservation Num
                        Autofill_List_Numbers(2),   // Telephone Num
                        GetSeatAvailable(),         // Seat Num
                        Autofill_List_MealChoice());// Preferred Meal
            break;
        case 6:
            CreateNewNode(                      // primary list
                        "Rani",                     // First Name
                        "Falcon",                   // Last Name
                        Autofill_List_Numbers(0),   // Passenger ID
                        Autofill_List_Numbers(1),   // Reservation Num
                        Autofill_List_Numbers(2),   // Telephone Num
                        GetSeatAvailable(),         // Seat Num
                        Autofill_List_MealChoice());// Preferred Meal
            break;
        case 7:
            CreateNewNode(                      // primary list
                        "Yasmine",                  // First Name
                        "Benicia",                  // Last Name
                        Autofill_List_Numbers(0),   // Passenger ID
                        Autofill_List_Numbers(1),   // Reservation Num
                        Autofill_List_Numbers(2),   // Telephone Num
                        GetSeatAvailable(),         // Seat Num
                        Autofill_List_MealChoice());// Preferred Meal
            break;
        case 8:
            CreateNewNode(                      // primary list
                        "Al",                       // First Name
                        "Bundy",                    // Last Name
                        Autofill_List_Numbers(0),   // Passenger ID
                        Autofill_List_Numbers(1),   // Reservation Num
                        Autofill_List_Numbers(2),   // Telephone Num
                        GetSeatAvailable(),         // Seat Num
                        Autofill_List_MealChoice());// Preferred Meal
            break;
        case 9:
            CreateNewNode(                      // primary list
                        "Janeeva",                  // First Name
                        "Zaina",                    // Last Name
                        Autofill_List_Numbers(0),   // Passenger ID
                        Autofill_List_Numbers(1),   // Reservation Num
                        Autofill_List_Numbers(2),   // Telephone Num
                        GetSeatAvailable(),         // Seat Num
                        Autofill_List_MealChoice());// Preferred Meal
            break;
        case 10:
            CreateNewNode(                      // primary list
                        "Ofra",                     // First Name
                        "Sable",                    // Last Name
                        Autofill_List_Numbers(0),   // Passenger ID
                        Autofill_List_Numbers(1),   // Reservation Num
                        Autofill_List_Numbers(2),   // Telephone Num
                        GetSeatAvailable(),         // Seat Num
                        Autofill_List_MealChoice());// Preferred Meal
            break;
        case 11:
            CreateNewNode(                      // primary list
                        "Nadalia",                  // First Name
                        "Hao",                      // Last Name
                        Autofill_List_Numbers(0),   // Passenger ID
                        Autofill_List_Numbers(1),   // Reservation Num
                        Autofill_List_Numbers(2),   // Telephone Num
                        GetSeatAvailable(),         // Seat Num
                        Autofill_List_MealChoice());// Preferred Meal
            break;
        case 12:
            CreateNewNode(                      // primary list
                        "Hana",                     // First Name
                        "Starr",                    // Last Name
                        Autofill_List_Numbers(0),   // Passenger ID
                        Autofill_List_Numbers(1),   // Reservation Num
                        Autofill_List_Numbers(2),   // Telephone Num
                        GetSeatAvailable(),         // Seat Num
                        Autofill_List_MealChoice());// Preferred Meal
            break;
        case 13:
            CreateNewNode(                  // primary list
                        "Ashia",                    // First Name
                        "Baeddan",                  // Last Name
                        Autofill_List_Numbers(0),   // Passenger ID
                        Autofill_List_Numbers(1),   // Reservation Num
                        Autofill_List_Numbers(2),   // Telephone Num
                        GetSeatAvailable(),         // Seat Num
                        Autofill_List_MealChoice());// Preferred Meal
            break;
        case 14:
            CreateNewNode(                      // primary list
                        "Qi",                       // First Name
                        "Wahponjea",                // Last Name
                        Autofill_List_Numbers(0),   // Passenger ID
                        Autofill_List_Numbers(1),   // Reservation Num
                        Autofill_List_Numbers(2),   // Telephone Num
                        GetSeatAvailable(),         // Seat Num
                        Autofill_List_MealChoice());// Preferred Meal
            break;
        case 15:
            CreateNewNode(                      // primary list
                        "Hali",                     // First Name
                        "Eamon",                    // Last Name
                        Autofill_List_Numbers(0),   // Passenger ID
                        Autofill_List_Numbers(1),   // Reservation Num
                        Autofill_List_Numbers(2),   // Telephone Num
                        GetSeatAvailable(),         // Seat Num
                        Autofill_List_MealChoice());// Preferred Meal
            break;
        case 16:
            CreateNewNode(                      // primary list
                        "Tai Yang",                 // First Name
                        "Taipa",                    // Last Name
                        Autofill_List_Numbers(0),   // Passenger ID
                        Autofill_List_Numbers(1),   // Reservation Num
                        Autofill_List_Numbers(2),   // Telephone Num
                        GetSeatAvailable(),         // Seat Num
                        Autofill_List_MealChoice());// Preferred Meal
            break;
        case 17:
            CreateNewNode(                      // primary list
                        "Achava",                   // First Name
                        "Nili",                     // Last Name
                        Autofill_List_Numbers(0),   // Passenger ID
                        Autofill_List_Numbers(1),   // Reservation Num
                        Autofill_List_Numbers(2),   // Telephone Num
                        GetSeatAvailable(),         // Seat Num
                        Autofill_List_MealChoice());// Preferred Meal
            break;
        case 18:
            CreateNewNode(                      // primary list
                        "John",                     // First Name
                        "Hancock",                  // Last Name
                        Autofill_List_Numbers(0),   // Passenger ID
                        Autofill_List_Numbers(1),   // Reservation Num
                        Autofill_List_Numbers(2),   // Telephone Num
                        GetSeatAvailable(),         // Seat Num
                        Autofill_List_MealChoice());// Preferred Meal
            break;
        case 19:
            CreateNewNode(                      // primary list
                        "Theodore",                 // First Name
                        "Roosevelt",                // Last Name
                        Autofill_List_Numbers(0),   // Passenger ID
                        Autofill_List_Numbers(1),   // Reservation Num
                        Autofill_List_Numbers(2),   // Telephone Num
                        GetSeatAvailable(),         // Seat Num
                        Autofill_List_MealChoice());// Preferred Meal
            break;
        default: // Easter Egg!
            CreateNewNode(                      // primary list
                        "Jenny",                    // First Name
                        "Tommy Tutone",             // Last Name
                        -919,                       // Passenger ID [919 area code ;)]
                        -919,                       // Reservation Num
                        8675309,                    // Telephone Num [reference: https://youtu.be/8ou6DDG5e7I ]
                        919,                        // Seat Num
                        "MRE");                     // Preferred Meal - Military acronym for 'Meal Ready to Eat', its horrible.
            break;
        } // switch
    } // for
} // Autofill_List()



// User Input [String]
// ===============================================
// Documentation:
//  This function will allow the user to input a specific string into the program.
// -----------------------------------------------
// Parameters:
//  UsePrompt [bool]
//      If true, this will provide the python'ish prompt.
// -----------------------------------------------
// Output:
//  string
//      Returns a string captured from STDIN.
// ===============================================
std::string Reservation::UserInput_String(bool UsePrompt = true)
{
    std::string userInput;      // Use this to capture the STDIN

    if (UsePrompt)
        std::cout << ">>>>> ";  // The python'ish prompt

    std::cin >> userInput;      // Capture the input

    return userInput;           // Return the value.
} // UserInput_String()
```

```cpp



// User Input [Number]
// ===============================================
// Documentation:
//  This function will allow the user to input a specific number into the program.
// -----------------------------------------------
// Parameters:
//  UsePrompt [bool]
//      If true, this will provide the python'ish prompt.
// -----------------------------------------------
// Output:
//  int
//      Returns an int captured from STDIN.
// ===============================================
int Reservation::UserInput_Number(bool UsePrompt = true)
{
    int userInput;              // Use this to capture the STDIN

    if (UsePrompt)
        std::cout << ">>>>> ";  // The python'ish prompt

    std::cin >> userInput;      // Capture the input

    while(!std::cin)        //ensures user inputed number is an integer
    {
        std::cin.clear();
        std::cin.ignore(INT_MAX, '\n');
        std::cout << "bad input, please enter again." << std::flush << std::endl;

        std::cout << ">>>>> ";  // The python'ish prompt

        std::cin >> userInput;

    }

    std::cin.clear();                //clears cin buffer
    std::cin.ignore(INT_MAX, '\n');
        return userInput;            // Return the value.

} // UserInput_Number()



// User Input [Bool]
// ===============================================
// Documentation:
//  This function will allow the user to input a yes or no into the program.
// -----------------------------------------------
// Parameters:
//  UsePrompt [bool]
//      If true, this will provide the python'ish prompt.
// -----------------------------------------------
// Output:
//  bool
//      Returns a bool captured from STDIN.
// ===============================================
bool Reservation::UserInput_Bool(bool UsePrompt = true)
{
    char userInput;             // Use this to capture the STDIN

    if (UsePrompt)
        std::cout << ">>>>> ";  // The python'ish prompt

    std::cin >> userInput;      // Capture the input


    if (tolower(userInput) == 'y') // See if 'Yes' was selected
        return true;        // Yes
    else
        return false;       // No
} // UserInput_Bool()



// Manual Customer Add [Meal Choice]
// ===============================================
// Documentation:
//  I have decided to separate this chunk of code into its own function
//  this way its a bit easier and the parent function is not so bloated.
// -----------------------------------------------
// Output:
//  string
//      Returns the preferred meal choice
// ===============================================
std::string Reservation::ManualCustomerAdd_MealChoice()
{
    // We will use this to store the user's choice and then process it later.
    int userChoice;

    // Provide the in-flight meal list:
    std::cout << "Select a number: " << std::endl
        << " 1) Monkey Brains" << std::endl
        << " 2) Tuna Eyeballs" << std::endl
        << " 3) Raw Octopus" << std::endl
        << " 4) Fish" << std::endl
        << " 5) Expired Peanuts" << std::endl << std::endl;

    // Prevent bad input; run away protection
    bool badInputCatch;
    do
    {
        // Get the customer's request
        userChoice = UserInput_Number();

        // Process the user's request
        switch (userChoice)
        {
        case 1:                 // Monkey Brains
            return "Monkey Brains";
            break;
        case 2:                 // Tuna Eyeballs
            return "Tuna Eyeballs";
            break;
        case 3:                 // Raw Octopus
            return "Raw Octopus";
            break;
        case 4:                 // Fish [Seriously, don't go for the fish! https://youtu.be/rQbj9uvYL8I ]
            return "Fish";
            break;
        case 5:                 // Expired Peanuts
            return "Expired Peanuts";
            break;
        default:                // Bad Input
            std::cout << "Incorrect option!" << std::endl;
            badInputCatch = true;
            break;
        } // switch
    } while (badInputCatch);
} // ManualCustomerAdd_MealChoice()



// Manual Customer add
// ===============================================
// Documentation:
//  This will allow the end-user to manually create a new entry within the list.
//   Capture all fields possible and then run through the importing algorithm.
// -----------------------------------------------
// Parameters:
//  head [Node**]
//      The list in which is to be appended
// ===============================================
void Reservation::ManualCustomerAdd()
{
    // Declarations
    // ------------------
    std::string stdinNameFirst;
    std::string stdinNameLast;
    int stdinPhone;
    std::string stdinMealChoice;
    int stdinSeat;
    // ----
    // working variables
    bool cacheBit;
    // ------------------


    // Provide instructions to the user
    std::cout << "Please provide the following information:" << std::endl << std::endl;

    // ----

    // Capture first name:
    std::cout << "First name: ";
    stdinNameFirst = UserInput_String(false);
    std::cout << std::endl;

    // Capture last name:
    std::cout << "Last name: ";
    stdinNameLast = UserInput_String(false);
    std::cout << std::endl;

    // Capture telephone number:
    std::cout << "Telephone number: ";
    stdinPhone = UserInput_Number(false);
    std::cout << std::endl;

    // Capture preferred meal:
    std::cout << "In-flight meal choice: " << std::endl;
    stdinMealChoice = ManualCustomerAdd_MealChoice();

    // Ask the user about preferred seating
    int seatCheck;      // Used for inspecting if seat is available
    bool seatConfirmed; // Used for confirming if the seat is available.

    do  // To avoid errors or frustrating the end-user, put this
    {   // question in a loop and easily escapable via auto-seat-placement.
        // [optional] Seat
        std::cout << "Preferred seating arrangement? [Y] = Yes | [N] = No" << std::endl;

        // Capture user input for seating preference
        cacheBit = UserInput_Bool();

        if (cacheBit)
        {                   // The customer has seating arrangements
            std::cout << "Preferred seating: ";
            stdinSeat = UserInput_Number(false);

            // Check to make sure that the seat is available
            if (GetSeatAvailable(stdinSeat) != -255)
                seatConfirmed = true;   // Seat is available!
            else
            {
                seatConfirmed = false;  // Seat is not available
                std::cout << std::endl << "This seat is presently not available!" << std::endl;
            } // else
        } // if
        else        // Automatically find the next available seat
        {
            stdinSeat = GetSeatAvailable(); // Automatically find available seat
            seatConfirmed = true;           // Auto-confirm the seat being available.
        } // else
    } while (!seatConfirmed);


    std::cout << std::endl;

    // ----

    // Now that we have the information we need, now lets try to add this new entry into the list!
    CreateNewNode(              // primary list
        stdinNameFirst,             // First Name
        stdinNameLast,              // Last Name
        Autofill_List_Numbers(0),   // Passenger ID [919 area code ;)]
        Autofill_List_Numbers(1),   // Reservation Num
        stdinPhone,                 // Telephone Num [reference: https://youtu.be/8ou6DDG5e7I ]
        stdinSeat,                  // Seat Num
        stdinMealChoice);           // Preferred Meal -  Military acronym for 'Meal Ready to Eat', its horrible.
} // ManualCustomerAdd()



// Search Node Entries
// ===============================================
// Documentation:
//  This is a very important function that will promptly scan the entire list
//   for a specific key.
//  This function will scan all nodes available, but it will stop at the
//   first available hit from the scan.  With that, once the scan finds the
//   key in one of the nodes, this function will NOT continue scanning afterwards.
// -----------------------------------------------
// Methodology:
//  Using both Node types, cur and pre, we will perform the scan as follows:
// | ====== |     | ====== |     | ====== |     | ====== |     | ====== |
// | NODE 1 |     | NODE 2 |     | NODE 3 |     | NODE N |     | NODE N+1
// | ~~~~~~ |     | ~~~~~~ |     | ~~~~~~ |     | ~~~~~~ |     | ~~~~~~ |
// |  DATA  |     |  DATA  |     |  DATA  |     |  DATA  |     |  DATA  |
```

```
// | ------ |     | ------ |     | ------ |     | ------ |     | ------ |
// |  NEXT  ----> |  NEXT  ----> |  NEXT  ----> |  NEXT  ----> |  NEXT  ---->
// | ====== |     | ====== |     | ====== |     | ====== |     | ====== |
//    ---            pre             cur             <NOT YET SCANNED>
// Simple Logic:
// Fist scan:
// cur = node 1 || pre = NULL
// Second scan:
// cur = node 2 || pre = node 1
// Third scan:
// cur = node 3 || pre = node 2
// N scan: <general form>
// cur = node N || pre = node N-1
// -----------------------------------------------
// Parameters:
//  cur [Node** - Alterable]
//      Used for processing; holds the search point
//  pre [Node** - Alterable]
//      Used for processing before 'cur'; holds the node right before 'cur' pointer.
//  searchMode [int]
//      Determines how the lists will be scanned within this function.
//      0 = Scan Last name [string]
//      1 = Scan first name [string]
//      2 = Scan passenger ID [int]
//      3 = Scan reservation number [int]
//      4 = Scan telephone number [int]
//      5 = Scan seat number [int]
//  searchKeyString [string]
//      A specific key in a string form used for scanning each node.
//  searchKeyInt [int]
//      A specific key in an int form used for scanning each node.
// -----------------------------------------------
// Output:
//  bool
//      Reports the status if the operation was successful or failed.
//      true = an error occurred
//      false = operation successful
// ===============================================
bool Reservation::Search(Node** cur, Node** pre, int searchMode, std::string searchKeyString = "NA", int searchKeyInt = -255)
{
    // If the cur pointer points to NULL, then there is nothing to scan.
    if (*cur == NULL)
        return false;

    // Scan the node
    while (*cur != NULL)
    {
        // Besides using a nesting conditional statement, we are going to use
        //  a switch statement for simplicity.
        switch (searchMode)
        {
        case 0:     // Scan last name
            if (searchKeyString.compare((*cur)->nameLast) == 0)
                                    // Equality between the two strings
                                    // [ http://www.cplusplus.com/reference/string/string/compare/ ]
                return true;
            break;
        case 1:     // Scan first name
            if (searchKeyString.compare((*cur)->nameFirst) == 0)
                                    // Equality between the two strings
                                    // [ http://www.cplusplus.com/reference/string/string/compare/ ]
                return true;
            break;
        case 2:     // Scan passenger ID
            if ((*cur)->passengerID == searchKeyInt)
                return true;
            break;
        case 3:     // Scan reservation number
            if ((*cur)->reservationNum == searchKeyInt)
                return true;
            break;
        case 4:     // Scan telephone number
            if ((*cur)->telephoneNum == searchKeyInt)
                return true;
            break;
        case 5:     // Scan seat number
            if ((*cur)->seatNum == searchKeyInt)
                return true;
            break;
        } // switch

        // Update the node positions
        *pre = (*cur);              // Update pre to cur's current position.
        *cur = (*cur)->next;        // Shift cur to next position.
    } // while

    // Unable to find a node with that specific key and data.
    return false;
} // Search()



// List Size
// ===============================================
// Documentation:
//  This function will scan through the entire list and evaluate its
//  total size, thus providing with the entire list size allocated.
// -----------------------------------------------
// Output:
//  List Size
```

```cpp
//      Return the list size
// ===============================================
int Reservation::ListSize()
{
    Node* temp = head;      // Directly copy the address of the 'head' in which
                            // it will be evaluated.  We will not do this directly
                            // with the primary list.
    int counter = 0;        // This will retain a count of how many nodes exists within
                            // the list.

    for (; temp != NULL; temp = temp->next)
        counter++;

    return counter;         // Return the size to the calling function
} // ListSize()



// Find and Print Passenger
// ===============================================
// Documentation:
//  This function will allow the end-user to search for a specific passenger
//  and output the passenger's information on the terminal.
//
//  This function will depend on several functions:
//   Search() && Print_Passenger_List() && UserInput_String()
//   UserInput_Number()
// ===============================================
void Reservation::FindPrintPassenger()
{
    // Present the user with a menu in which to search by
    std::cout << "Search for Passenger:" << std::endl
        << "-----------------------" << std::endl
        << " 1) Last name" << std::endl
        << " 2) Telephone" << std::endl
        << " 3) Reservation ID" << std::endl
        << " 4) Passenger ID" << std::endl
        << " 5) Seat Number" << std::endl
        << " 0) Exit" << std::endl
        << std::endl;

    bool badInput;  // Make sure that the input provided from the end-user
                    // is valid; run-away protection.
    std::string captureString;
    int captureInt;

    Node* nodeIndex = head;
    Node* nullityNode = NULL;

    do {
        // Capture the user's request and process the request
        switch (UserInput_Number())
        {
        case 1:                     // Last name
            // Capture the last name from the end-user
            std::cout << "Enter passenger's last name: ";
            captureString = UserInput_String(false);
            std::cout << std::endl;

            if (Search(&nodeIndex,  // Our list to be scanned and processed.
                &nullityNode,       // Required for the function, but not used.
                0,                  // Search by last name
                captureString))     // String to search
                Print_Passenger_List(nodeIndex, true);// Output the results
            else
                std::cout << "Unable to find passenger: " << captureString
                << std::endl;

            badInput = false;
            break;
        case 2:                     // Telephone
            std::cout << "Enter passenger's telephone number: ";
            captureInt >> UserInput_Number(false);
            std::cout << std::endl;

            if (Search(&nodeIndex,  // Our list to be scanned and processed.
                &nullityNode,       // Required for the function, but not used.
                4,                  // Search by telephone number
                "NA",               // Default to 'NA' due to standard; unused.
                captureInt))        // integer to search
                Print_Passenger_List(nodeIndex, true);// Output the results
            else
                std::cout << "Unable to find passenger with the telephone number: " << captureInt
                << std::endl;

            badInput = false;
            break;
        case 3:                     // Reservation ID
            std::cout << "Enter passenger's Reservation number: ";
            captureInt = UserInput_Number(false);
            std::cout << std::endl;

            if (Search(&nodeIndex,  // Our list to be scanned and processed.
                &nullityNode,       // Required for the function, but not used.
                3,                  // Search by reservation number
                "NA",               // Default to 'NA' due to standard; unused.
                captureInt))        // integer to search
                Print_Passenger_List(nodeIndex, true);// Output the results
            else
                std::cout << "Unable to find passenger with the reservation number: " << captureInt
```

```cpp
                << std::endl;

            badInput = false;
            break;
        case 4:                     // Passenger ID
            std::cout << "Enter passenger's Passenger number: ";
            captureInt = UserInput_Number(false);
            std::cout << std::endl;

            if (Search(&nodeIndex,  // Our list to be scanned and processed.
                &nullityNode,       // Required for the function, but not used.
                2,                  // Search by passenger number
                "NA",               // Default to 'NA' due to standard; unused.
                captureInt))        // integer to search
                Print_Passenger_List(nodeIndex, true);// Output the results
            else
                std::cout << "Unable to find passenger with the passenger number: " << captureInt
                << std::endl;

            badInput = false;
            break;
        case 5:                     // Seat Number [Occupied]
            std::cout << "Enter passenger's Seat number: ";
            captureInt = UserInput_Number(false);
            std::cout << std::endl;

            if (Search(&nodeIndex,  // Our list to be scanned and processed.
                &nullityNode,       // Required for the function, but not used.
                5,                  // Search by seat number
                "NA",               // Default to 'NA' due to standard; unused.
                captureInt))        // integer to search
                Print_Passenger_List(nodeIndex, true);// Output the results
            else
                std::cout << "Unable to a find a passenger in seat number: " << captureInt
                << std::endl;

            badInput = false;
            break;
        case 0:                     // Exit; silently leave this function
            badInput = false;
            break;
        default:                    // Bad input
            std::cout << "Incorrect option!" << std::endl;
            badInput = true;
            break;
        } // switch
    } while (badInput);
} // FindPrintPassenger()


```

```cpp

// Cancel Reservation
// =====================================================================================================
// Documentation:
// This code allows the user to search for a passenger in the list then remove them from said list.
// Logic:
// This code is a modified version of the Search function that only searches for passenger Last names.
// When it finds the Last Name it will delete the Node the Last Name is connected to.
// ========================================================================================================
bool Reservation::delete_node()
{
        Node* pre = head;
        Node* temp = head;
        std::string captureString;


        if (temp == NULL){   //makes sure there is something in head
                std::cout << "The list is empty." <<std::endl;
                return false;
        }
        std::cout << "Which passenger would you like to delete? Enter their last name:  ";

        captureString = UserInput_String(false);    //captures user inputed last name

        if (Search(&temp,       // Our list to be scanned and processed.
                &pre,       // the node before the desired node will be stored here
                0,                  // Search by last name
                captureString)) {   // String to search

            if(head == temp){
                head = temp->next;
            }else{
                pre->next = temp->next; // sets the prev node in the list to point to the node after the node
            }                           // that will be deleted
            delete temp; // deletes the node if the node is the node that needs to be deleted

            std::cout << "The passenger was removed from the list" << std::endl;
            return true;
        }else{
            std::cout <<"The passenger was not found" <<std::endl;
            return false;
        }

}



// Update Passenger Information
// ===============================================
// Documentation:
//  This function will provide a front-end to the end-user to identify what
//  passenger's information is to be updated.
//  This function will first ask the end-user to input the current information
//   about the passenger, then with this information - we scan the list for the node.
//   IIF if the node exists with the passenger's information, then will provide the
//   end-user with update options for that passenger.
//
// NOTES: This function was coded badly and this should be better optimized, but
//          as we work at a coding sweat shop, there is no better way!
//          The pay grade at this code sweat shop only offers 27 cents per hour :(
//          Please help us!
//
//  This function depends on the Search() and the input (number\int) function.
// ===============================================
void Reservation::UpdatePassengerInformation()
{
    // Present the user with a menu in which to search by
    std::cout << "Search for Passenger:" << std::endl
        << "------------------------" << std::endl
        << " 1) Last name" << std::endl
        << " 2) Telephone" << std::endl
        << " 3) Reservation number" << std::endl
        << " 4) Passenger number" << std::endl
        << " 5) Seat Number" << std::endl
        << " 0) Exit" << std::endl
        << std::endl;

    bool badInput;  // Make sure that the input provided from the end-user
                    // is valid; run-away protection.
    std::string captureString;
    int captureInt;

    Node* nullityNode = NULL;   // Create a list placeholder; we will not need it in this function, but
                                // it is required when calling the Search() function.

    bool targetFound = false;   // We will use this to determine if the passenger was
```

```
                                // detect during the search.  This will become true
                                // IIF the passenger was found, otherwise - its false.
                                // IIF this is true, then we will allow the user to update
                                //  that node index.

    Node* temp = head;          //used so the head pointer is not wrongly updated

    do {
        // Capture the user's request and process the request
        switch (UserInput_Number())
        {
        case 1:                     // Last name
                                    // Capture the last name from the end-user
            std::cout << "Enter passenger's last name: ";
            captureString = UserInput_String(false);
            std::cout << std::endl;

            if (Search(&temp,       // Our list to be scanned and processed.
                &nullityNode,       // Required for the function, but not used.
                0,                  // Search by last name
                captureString))     // String to search
                targetFound = true;
            else
                std::cout << "Unable to find passenger: " << captureString
                << std::endl;

            badInput = false;
            break;
        case 2:                     // Telephone
            std::cout << "Enter passenger's telephone number: ";
            captureInt >> UserInput_Number(false);
            std::cout << std::endl;

            if (Search(&temp,       // Our list to be scanned and processed.
                &nullityNode,       // Required for the function, but not used.
                4,                  // Search by telephone number
                "NA",               // Default to 'NA' due to standard; unused.
                captureInt))        // integer to search
                targetFound = true;
            else
                std::cout << "Unable to find passenger with the telephone number: " << captureInt
                << std::endl;

            badInput = false;
            break;
        case 3:                     // Reservation ID
            std::cout << "Enter passenger's Reservation number: ";
            captureInt = UserInput_Number(false);
            std::cout << std::endl;

            if (Search(&temp,       // Our list to be scanned and processed.
                &nullityNode,       // Required for the function, but not used.
                3,                  // Search by reservation number
                "NA",               // Default to 'NA' due to standard; unused.
                captureInt))        // integer to search
                targetFound = true;
            else
                std::cout << "Unable to find passenger with the reservation number: " << captureInt
                << std::endl;

            badInput = false;
            break;
        case 4:                     // Passenger ID
            std::cout << "Enter passenger's Passenger number: ";
            captureInt = UserInput_Number(false);
            std::cout << std::endl;

            if (Search(&temp,       // Our list to be scanned and processed.
                &nullityNode,       // Required for the function, but not used.
                2,                  // Search by passenger number
                "NA",               // Default to 'NA' due to standard; unused.
                captureInt))        // integer to search
                targetFound = true;
            else
                std::cout << "Unable to find passenger with the passenger number: " << captureInt
                << std::endl;

            badInput = false;
            break;
        case 5:                     // Seat Number [Occupied]
            std::cout << "Enter passenger's Seat number: ";
            captureInt = UserInput_Number(false);
            std::cout << std::endl;

            if (Search(&temp,       // Our list to be scanned and processed.
                &nullityNode,       // Required for the function, but not used.
                5,                  // Search by seat number
                "NA",               // Default to 'NA' due to standard; unused.
                captureInt))        // integer to search
                targetFound = true;
            else
                std::cout << "Unable to a find a passenger in seat number: " << captureInt
                << std::endl;

            badInput = false;
            break;
        case 0:                     // Exit; silently leave this function
            badInput = false;
            break;
        default:                    // Bad input
            std::cout << "Incorrect option!" << std::endl;
            badInput = true;
            break;
        } // switch
    } while (badInput);

    // If the passenger was not found, leave this function.
    if (!targetFound)
    {
        // Allow the end-user to view the message that the passenger
        // was not found during the scan.
        // ----
        std::cout << "Press the enter or return key to continue. . ." << std::endl;      // This is alternative version of 'system("PAUSE")'
        std::cin.ignore();          // and will work outside of Windows.
        std::cin.ignore();  // avoid input ghosting?
        // ----

        return;
    }


    // =========================================================================
    // -------------------------------------------------------------------------
    // =========================================================================
    // TARGET FOUND

    do
    {

    // What does the end-user want to update?
    std::cout << "Information to Update:" << std::endl
        << "-----------------------" << std::endl
        << " 1) Telephone" << std::endl
        << " 2) Reservation number" << std::endl
        << " 3) Passenger number" << std::endl
        << " 4) Seat Number" << std::endl
        << " 0) Exit" << std::endl
        << std::endl;


    // Get the user's input and evaluate it

        switch (UserInput_Number())
        {
        case 1:             // Update telephone
            // Let the user know of the current value
```

```cpp
            std::cout << "Current telephone number: " << temp->telephoneNum << std::endl;
            // Allow the user to update that specific field:
            std::cout << "Enter a new value: ";
            temp->telephoneNum = UserInput_Number(false);

            badInput = false;
            break;
        case 2:             // Update reservation ID
            // Let the user know of the current value
            std::cout << "Current reservation number: " << temp->reservationNum << std::endl;
            // Allow the user to update that specific field:
            std::cout << "Enter a new value: ";
            temp->reservationNum = UserInput_Number(false);

            badInput = false;
            break;
        case 3:             // Update passenger ID
            // Let the user know of the current value
            std::cout << "Current passenger number: " << temp->passengerID << std::endl;
            // Allow the user to update that specific field:
            std::cout << "Enter a new value: ";
            temp->passengerID = UserInput_Number(false);

            badInput = false;
            break;
        case 4:             // Seat Number
                int newSeatNum;
                // Let the user know of the current value
                std::cout << "Current seat number: " << temp->seatNum << std::endl;
                // Allow the user to update that specific field:
                std::cout << "Enter a new value: ";

                newSeatNum = UserInput_Number(false);       //gathers user input

                if(GetSeatAvailable(newSeatNum) != -255)    //checks to make sure seat is not taken
                {
                    temp->seatNum = newSeatNum;     //changes seat number if it is not taken
                        badInput = false;
                }else{
                    std::cout<< std::endl << "Seat already taken!" << std::endl;
                    badInput = true;
                }
            break;
        case 0:             // Exit; silently leave this function
            badInput = false;
            break;
        default:            // Bad input
            std::cout << "Incorrect option!" << std::endl;
            badInput = true;
            break;
        }

        std::cout << std::endl;
    } while (badInput);
} // UpdatePassengerInformation()





// Check In Passenger
// ===============================================
// Documentation:
//  This function will search for a user inputed passenger in a list
//  and change the passengers status to checked in if he/she is found.
// -----------------------------------------------
// Output:
//  changes passenger's status to checked in or says passenger not found.
// ===============================================
void Reservation::CheckInPassenger()
{
        Node* nullityNode = NULL;
        Node* temp = head;
        std::string captureString;


        if (temp == NULL){   //makes sure there is something in head
                std::cout << "The list is empty.  There are no passengers to check in" <<std::endl;
        }else{
            std::cout << "Which passenger would you like to Check in?  Enter their last name:";

            captureString = UserInput_String(false);    //captures user inputted last name

            if (Search(&temp,       // Our list to be scanned and processed.
                    &nullityNode,       // the node before the desired node will be stored here
                    0,                  // Search by last name
                    captureString)) {   // String to search

                temp->checkedIn = true;
                std::cout << temp->nameFirst << " " << temp->nameLast << " has been checked in." << std::endl;
            }else{
                std::cout << "This passenger is not in the list." <<std::endl;
            }
        }
}//CheckInPassenger




// Print Check In List
// ===============================================
// Documentation:
//  This function will produce the check in report
// -----------------------------------------------
// Output:
//  displays a list of all passengers in the list.  Will display
//  their first and last names and whether they are checked in or not.
//  it will also display the total amount of checked in passengers and
//  the total amount of passengers not checked in
// ===============================================
void Reservation::Print_CheckIn_List()
{
    int CheckedInCnt = 0;   //counts the number of passengers checked in
    int NotCheckedInCnt = 0;    //counts the number of passengers not checked in
    Node* temp = head;      //starts at the beginning of the list

    while(temp != NULL){

        //prints each passenger's name and a checked in prompt
        std::cout << temp->nameFirst <<" " <<temp->nameLast << std::endl
                  << "Checked In?:  ";

        //checks to see if each passenger is checked in
        if(temp->checkedIn){
            std::cout << "Yes" << std::endl << std::endl;
            CheckedInCnt++;
        }else{
            std::cout << "No" << std::endl << std::endl;
            NotCheckedInCnt++;
        }
        temp = temp->next;
    }

    std::cout<< std::endl << "There are " << CheckedInCnt << " Passengers checked in." <<std::endl;
    std::cout<< std::endl << "There are " << NotCheckedInCnt << " Passengers not checked in." <<std::endl;
}   //Print Check In List




// Print meal list
// ===============================================
// Documentation:
```

```
//  This function will display the passenger meal list report
// -----------------------------------------------
// Output:
//  Outputs a list of all passengers.  Will display their meal choices and
//  their first and last names.  It will also display the totals for each meal
//  choice that is available.
// ===============================================
void Reservation::Print_Meal_List()
{

    Node* temp = head;

    if(temp != NULL){
        //these counters will count the amount of each meal that was chosen
        //for all passengers in the list
        int meal1Cnt = 0;
        int meal2Cnt = 0;
        int meal3Cnt = 0;
        int meal4Cnt = 0;
        int meal5Cnt = 0;



        while(temp != NULL){    //increments until the end of the list is reached

                                //displays the passengers name and meal choice
            std::cout << temp->nameFirst << " " << temp->nameLast << std::endl
                      << "Meal Choice:  " << temp->mealType << std::endl << std::endl;

                      //increments counter for whichever meal the passenger chose.
            if(temp->mealType.compare("Monkey Brains") == 0){
                meal1Cnt++;
            }else if(temp->mealType.compare("Tuna Eyeballs") == 0){
                meal2Cnt++;
            }else if(temp->mealType.compare("Raw Octopus") == 0){
                meal3Cnt++;
            }else if(temp->mealType.compare("Fish") == 0){
                meal4Cnt++;
            }else if(temp->mealType.compare("Expired Peanuts") == 0){
                meal5Cnt++;
            }


            temp = temp->next;  //moves to next passenger in the list
        }//while

        //displays meal choice totals
        std::cout << "There are " << meal1Cnt << " orders for Monkey Brains." << std::endl
```

```cpp
                  << "There are " << meal2Cnt << " orders for Tuna Eyeballs." << std::endl
                  << "There are " << meal3Cnt << " orders for Raw Octopus." << std::endl
                  << "There are " << meal4Cnt << " orders for Fish." << std::endl
                  << "There are " << meal5Cnt << " orders for Expired Peanuts." << std::endl << std::endl;

    }else{      //if for no list

        std::cout << "There are currently no passengers in the list." << std::endl;
    }

}//print meal list


// Sort list
// ===============================================
// Documentation:
//  This function will put a passenger list in alphabetical order by last name
//
//  it does this by first checking each node against the first node.  if a node that
//  comes alphabetically before the first node is found, the function places that
//  node just before the first node.  Then the process begins again with the new first
//  node.  When no nodes are found that come alphabetically before the first node,
//  the node that other nodes are compared to is moved to the second node, and so on
//  until the node other nodes are compared to is the last node in the list.
// -----------------------------------------------
// Output:
//  Outputs a list sorted by last name.
// ===============================================
void Reservation::Sort()
{
    if(head != NULL)        //to make sure list is not empty
    {
            Node* current = head;       //this pointer contains the node being challenged
            Node* challenger = head;    //this pointer contains the node challenging the
                                        //current pointer for alphabetically first position

            Node* pre = head;           //this pointer is used to place the challenger node before
                                        //the current node

            Node* challengePre = head;  //this pointer is used to remove the challenger pointer from its old spot

            bool swap = false;          //this variable is used to indicate that a swap was made\

            std::string currentName;    //holds the current node's last name

            std::string challengerName; //holds the challenger node's last name

            while(current != NULL)      //iterates through until it reaches the end of the list, or
                                        //until a node that comes alphabetically before it is found
            {
                while(challenger->next != NULL)     //iterates through all nodes after the current node
                                                    //stops when it reaches the end of the list, or when
                                                    //it finds a node that comes alphabetically before the
                                                    //current node
                {
                    challengePre = challenger;
                    challenger = challenger->next;  //incrementing the challenger node to the next node
                                                    //in the list to be checked

                    currentName = current->nameLast;            //setting the last name strings to be checked
                    challengerName = challenger->nameLast;

                    if(Alphabetize(currentName, challengerName)){       // checking to see which node comes
                                                                        // alphabetically first by last name
                                                                        //if the challenger comes first,
                                                                        //it is placed just before the current
                                                                        //node in the list, and the process is restarted
                                                                        //for the new current node
                        challengePre->next = challenger->next;
                        if(current == head){        //for if the current node is at the head of the list
                            challenger->next = current;
                            head = challenger;
                        }else{
                            pre->next = challenger;
                            challenger->next = current;
                        }
                        current = challenger;
                        swap = true;        //indicates a swap was made
                        break;
                    }
                }//inner while

                if(!swap)       //if no swaps were made, the current node is moved to the next node
                                //in the list.
                {
                    pre = current;
                    current = current->next;
                    challenger = current;
                }
                swap = false;
            }//outer while

    }else{  //outer if
        std::cout << "The list is empty." << std::endl;
    }
}//sort




//Alphabetize
// ===============================================
// Documentation:
//  This function will decide which of two strings comes
//  alphabetically first
// -----------------------------------------------
// Output:
//  outputs true if the challenger string comes alphabetically
//  before the current string
//------------------------------------------------
//  Parameters:
//  current and challenger hold strings that will be compared
//  to determine which one comes alphabetically first
// ===============================================
bool Reservation::Alphabetize(std::string current, std::string challenger)
{
    if(current.length() <= challenger.length())     //checks to see which string is longer
                                                    //uses for loop to the shorter strings length
                                                    //this allows the function to show that the shorter
                                                    //word comes alphabetically first (i.e. Al, vs Ale)
    {
```

```cpp
        for(int i = 0; i <current.length(); i++)
        {
            if(int(tolower(current.at(i))) < int(tolower(challenger.at(i))))        //this statement analyzes
            {                           //the current character and challenger character at the current
                                        //iteration of the for loop.  It converts the letters
                                        //to their ANSII equivalent values and analyzes
                                        //those to see which one is larger.  capitol letters will be
                                        //converted to lowercase letters.  The other if statements basically
                                        //follow the same concept as this one

                return false;   //if the current character is alphabetically before the challenger character,
                                //the function returns false to note that the current string is alphabetically
                                //first
            }else if(int(tolower(current.at(i))) > int(tolower(challenger.at(i)))){


                return true;     //if the current character being analyzed is alphabetically after the
                                 //challenger character being analyzed, the function returns true to note that
                                 //the current string is alphabetically after the challenger string
            }
        }
        return false;       //returns false if all letters analyzed where the same.  This means that either
                            //the challenger string is longer than the current string (i.e al vs ale), or
                            //both strings are the same in which case no action is taken.


    }else{      //for if the challenger string is shorter than the current string.  same concepts as above,
                //accept for when current string is the same as challenger string up to the last characters
                //analyzed.  In this case the challenger string is alphabetically first since it is shorter
                //(i.e. ale vs al).
        for(int i = 0; i < challenger.length(); i++)
        {
            if(int(tolower(current.at(i))) < int(tolower(challenger.at(i))))
            {
                return false;
            }else if(int(tolower(current.at(i))) > int(tolower(challenger.at(i)))){
                return true;
            }
        }

        return true;
    }//outer else

}//Alphabetize
#endif // !__LinkList__Implementation__

```