```
#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:
```

```
43  //  This function will provide a list of functionality that is available to the
      end-user.
44  // ============================================
45  void Reservation::MainMenu()
46  {
47      // The border for the main menu screen
48      std::cout << "Select an option:" << std::endl
49          << "====================" << std::endl;
50
51      // Main menu options
52      std::cout << "[1] - Automatically Generate a Customer List" << std::endl
53          << "[2] - Manually add a new customer" << std::endl
54          << "[3] - Print all customer information" << std::endl
55          << "[4] - Search for a passenger" << std::endl
56          << "[5] - Update passenger information" << std::endl
57          << "[6] - Remove passenger from list" << std::endl
58          << "[7] - Check in passenger" << std::endl
59          << "[8] - Print check in report" << std::endl
60          << "[9] - Print menu report" << std::endl
61          << "[0] - Sort passenger list" << std::endl
62          << "[X] - Exit" << std::endl << std::endl;
63  } // MainMenu()
64
65
66
67  // Evaluate and Run
68  // ============================================
69  // Documentation:
70  //  This will allow the user to perform the requested action, if STDIN is legal.
71  // --------------------------------------------
72  // Parameters:
73  //  STDIN [char]
74  //      User's requested action
75  //  head [Node**]
76  //      Primary list that contains user info
77  // ============================================
78  void Reservation::EvaluateAndRun(char STDIN)
79  {
80      Node* nullityNode = NULL;           // If the user chooses option '3', then
81                                          // this pointer will be available as
82                                          // the Print_Passenger_List() function
                        requires
83                                          // a node datatype, we will send
84                                          // an empty node but will not be
85                                          // used during the execution.
86      switch (STDIN)
87      {
88      case '1':   // Automatically generate customer list
89          Autofill_List();
90          break;
91      case '2':   // Manually input customer
92          ManualCustomerAdd();
```

```
 93          break;
 94      case '3':   // Print all customers [primary pointer]
 95          Print_Passenger_List(nullityNode, false);
 96          break;
 97      case '4':   // Search for passenger
 98          FindPrintPassenger();
 99          break;
100      case '5':   // Update passenger information
101          UpdatePassengerInformation();
102          break;
103      case '6':   // Thrash passenger node
104          delete_node();
105          break;
106      case '7':   // Passenger Checking
107          CheckInPassenger();
108          break;
109      case '8':
110          Print_CheckIn_List();   //print check in report
111          break;
112      case '9':
113          Print_Meal_List();      //print meal choice report
114          break;
115      case '0':
116          Sort();     //sort list
117          break;
118      case 'X':   // Quietly pass through; exit
119          break;
120      case 'x':   // Quietly pass through; exit
121          break;
122      default:    // Bad Input
123          std::cout << "Incorrect option!" << std::endl << std::endl;
124          break;
125      } // switch
126
127      ClearBuffer();  // Provide spacing after evaluation
128  } // EvaluateAndRun()
129
130
131
132  // Prompt User [Main Menu]
133  // ===============================================
134  // Documentation:
135  //  Capture user input and return it as an integer.  By convention, use a
       python'ish input prompt.
136  //  From personal experience, this is more clear that the program is 'wanting'
       something from the user.
137  // -----------------------------------------------
138  // Output:
139  //  STDIN [Char]
140  //      Return the STDIN from the end-user.
141  // ===============================================
142  char Reservation::PromptUser_MainMenu()
```

```
143 {
144     char inputCapture = '-';        // If in case - to avoid bugs, use a default ↵
        value.
145     std::cout << ">>>>> ";          // The python'ish prompt
146     std::cin >> inputCapture;       // Capture the input
147
148     return inputCapture;            // Return the value.
149 } // PromptUser_MainMenu()
150
151
152
153 // Clear Buffer
154 // ================================================
155 // This function, despite minimal, will try to make the buffer easier to read   ↵
      for the end user.
156 // ================================================
157 void Reservation::ClearBuffer()
158 {
159     std::cout << std::endl
160         << std::endl
161         << std::endl
162         << std::endl
163         << std::endl
164         << std::endl;
165 } // ClearBuffer()
166
167
168
169 // Reservation Constructor
170 // ================================================
171 // Documentation:
172 //  This function will initialize the head pointer to NULL.
173 // ================================================
174 Reservation::Reservation()
175 {
176     head = NULL;
177 };
178
179
180
181 // Print Passenger List
182 // ================================================
183 // Documentation:
184 //  This function will output all of the information within the link list.
185 // ------------------------------------------------
186 // Parameters:
187 //  listIndex [Node*]
188 //      This will take any valid link list, but this is only
189 //      usable for single indexes, not a list!  With that,
190 //      this parameter works with the 'runOnce' parameter and must be
191 //      set to 'true' instead of false.  IIF false, then listIndex
192 //      ignored.  Otherwise, when true, listIndex will be printed.
```

```
193 //  runOnce [bool]
194 //      When true, this will allow 'listIndex' to be printed on the screen.
195 //      This parameter adjusts this function to only print ONE node on the
196 //      terminal buffer and no more.  If this variable is 'false', then
197 //      the standard list takes precedence.
198 // ============================================
199 void Reservation::Print_Passenger_List(Node* listIndex, bool runOnce = false)
200 {
201     // IIF runOnce is true then use the listIndex, otherwise use head.
202     Node *temp = runOnce ? listIndex : head;
203 
204     // If head is empty, present an error message and leave this function.
205     if (temp == NULL)
206     {
207         std::cout << "<!> ERROR <!>: No entries within the list are present!" << ↵
               std::endl
208             << "Nothing to print nor report!  Please generate or create a new    ↵
                list." << std::endl;
209         return;
210     } // if
211 
212     int indexCounter = 1; // This will be helpful to know what index the client  ↵
          is located within the list.
213 
214     // Output the available list with the information required.
215     //  IIF runOnce is true, then we will only output one index.
216     //  else, when runOnce is false, we will output the entire list available.
217     while (temp != NULL && ((runOnce && indexCounter < 2) || !runOnce))
218     {
219         std::cout << "Index number: " << indexCounter << std::endl
220             << "Passenger ID: " << temp->passengerID << std::endl
221             << "Passenger Name: " << temp->nameLast << ", " << temp->nameFirst   ↵
                << std::endl
222             << "Telephone Number: " << temp->telephoneNum << std::endl
223             << "Reservation Number: " << temp->reservationNum << std::endl
224             << "Seat on the plane: " << temp->seatNum << std::endl
225             << "Preferred Meal Plan: " << temp->mealType << std::endl <<         ↵
                std::endl;
226 
227         temp = temp->next;  // Move to the next node
228         indexCounter++;
229     } // while
230 } // Print_Passenger_List()
231 
232 
233 
234 // Insert New Node Entry
235 // ============================================
236 // Documentation:
237 //  This function will forward the new node (or data) onto
238 //  the primary link list (or pointer)
239 // --------------------------------------------
```

```
240 // Parameters:
241 //  head [Node]
242 //      The primary list that is to be updated.
243 //  NewEntry [Node]
244 //      New information that is to be merged or imported into the 'head'.
245 // ============================================
246 void Reservation::InsertNode(Node* newEntry)
247 {
248     // Disallow any new entries if the plane is full.
249     if (PlaneSize <= ListSize())
250     {
251         std::cout << "<!> ERROR <!>: Unable to add new entry!" << std::endl
252             << "The plane is currently full and we have yet to add seats on the wings." << std::endl
253             << "Please consider deleting a passenger with no refunds." << std::endl;
254             return;
255     } // if
256
257     if (head == NULL)           // if the current list contains no entries
258         head = newEntry;        //  then immediately add the temp entry to the list.
259     else
260     {
261         newEntry->next = head;  // Import the primary list to the temp entry.
262         head = newEntry;        // Export all of the entries from the temp
263                                 // list to the primary list.
264     }
265 } // InsertNode()
266
267
268
269 // Create a New Node
270 // ============================================
271 // Documentation:
272 //  Generate a new entry to be imported into the primary list.
273 //  This function will merely make sure that all information is
274 //   present before allowing the new node to be official in the
275 //   primary list.
276 // --------------------------------------------
277 // Parameters:
278 //  head [Node]
279 //      The primary list that will soon include the new node entry.
280 //  nameFirst [string]
281 //      Client's first name
282 //  nameLast [string]
283 //      Client's last name
284 //  passengerID [int]
285 //      Client's Passenger ID
286 //  reservationNum [int]
287 //      Client's reservation ID
288 //  telephoneNum [int]
```

```
289 //      Client's telephone number
290 //  seatNum [int]
291 //      Client's seat number
292 //  mealType [string]
293 //      Client's requested meal.
294 // ================================================
295 void Reservation::CreateNewNode(std::string nameFirst, std::string nameLast, int ↵
        passengerID, int reservationNum, int telephoneNum, int seatNum, std::string  ↵
      mealType)
296 {
297     // Create a new node to store the new information
298     Node* newEntry = new Node;
299 
300 
301     // Generate the incoming data into the newly created node
302     newEntry->nameFirst = nameFirst;
303     newEntry->nameLast = nameLast;
304     newEntry->passengerID = passengerID;
305     newEntry->reservationNum = reservationNum;
306     newEntry->telephoneNum = telephoneNum;
307     newEntry->seatNum = seatNum;
308     newEntry->mealType = mealType;
309     newEntry->checkedIn = false;
310 
311     newEntry->next = NULL;
312     // Import the data into the primary list
313     InsertNode(newEntry);
314 } // CreateNewNode()
315 
316 
317 
318 // Autofill List [Numbers - Dependency]
319 // ================================================
320 // Documentation:
321 //  This function will provide an easier way to manage
322 //   randomized numbers for populating the passenger
323 //   information.
324 // ------------------------------------------------
325 // Parameters:
326 //  key [int]
327 //      This will allow us to provide a random number
328 //      based on what attribute we are requesting.
329 //      Acceptable Key Values:
330 //      0 = Passenger ID
331 //      1 = Reservation Number
332 //      2 = Telephone Number
333 // ------------------------------------------------
334 // Output:
335 //  Returns a specific, though randomized, integer based on the
336 //   key used.
337 // ================================================
338 int Reservation::Autofill_List_Numbers(int key)
```

```
339 {
340     // Initializations
341     // -------------------------
342     // We will be using these as a way to assure that all values are unique -
          dynamically at runtime.
343     // Best approach?  No.  Best on performance?  Absolutely not - at least if
          worst case occurs.
344     int newRandomValue;         // This will be used to inspect the rand() value
           is unique
345     bool uniqueFound;           // Will be used in cooperation with the loops.
346     Node* temp = head;          // The list that we will be searching
347     Node* nullityNode = NULL;   // A temporary list that wont be used, but
          required for the search() function.
348     // -------------------------
349
350     // Determine the requested type to return
351     switch (key)
352     {
353     case 0:         // Passenger ID
354         do          // Keep scanning until we find unique key
355         {           // It is possible for processing lag phenomenon
356             newRandomValue = rand() % 9999 + 1;
357             uniqueFound = Search(&temp, &nullityNode, 2, " ", newRandomValue) ?
                false : true;
358         } while (!uniqueFound);
359         return newRandomValue;  // Return the unique key
360         break;
361     case 1:         // Reservation Number
362         do          // Keep scanning until we find unique key
363         {           // It is possible for processing lag phenomenon
364             newRandomValue = rand() % 999 + 100;
365             uniqueFound = Search(&temp, &nullityNode, 3, " ", newRandomValue) ?
                false : true;
366         } while (!uniqueFound);
367         return newRandomValue;  // Return the unique key
368         break;
369     case 2:         // Telephone Numbers
370         do          // Keep scanning until we find unique key
371         {           // It is possible for processing lag phenomenon
372             newRandomValue = rand() % 8999999999 + 1000000000;
373             uniqueFound = Search(&temp, &nullityNode, 4, " ", newRandomValue) ?
                false : true;
374         } while (!uniqueFound);
375         return newRandomValue;  // Return the unique key
376         break;
377     default:
378         // Error; access violation occurred.
379         return -255;
380         break;
381     } // switch
382 } // Autofill_List_Numbers()
383
```

```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.
// ------------------------------------------------
```

```
436 // Output:
437 //  Returns the seat ID that is available.
438 //  if the seat is not available or is not unique,
439 //  then a value of -255 is returned, signaling an error.
440 //   however, if the seat is available - then the
441 //   seat requested will be returned to confirm.
442 // ============================================
443 int Reservation::GetSeatAvailable(int requestKey = -255)
444 {
445     // Initializations
446     // -------------------------
447     // We will be using these as a way to assure that all values are unique -
        dynamically at runtime.
448     // Best approach?  No.  Best on performance?  Absolutely not - at least if
        worst case occurs.
449     int newRandomValue;         // This will be used to inspect the rand() value
        is unique
450     bool uniqueFound;           // Will be used in cooperation with the loops.
451     Node* temp = head;          // The list that we will be searching
452     Node* nullityNode = NULL;   // A temporary list that wont be used, but
        required for the search() function.
453     // -------------------------
454
455     // Check to see if the user is requesting a new seat
456     if (requestKey != -255)
457     {   // Check if the seat is available; end-user manual request
458         uniqueFound = Search(&temp, &nullityNode, 5, " ", requestKey) ? false :
          true;
459         newRandomValue = uniqueFound ? requestKey : -255;
460         return newRandomValue;
461     }
462     else            // This will be used for auto fill functionality
463         do          // Keep scanning until we find unique key
464         {           // It is possible for processing lag phenomenon
465             newRandomValue = rand() % 100 + 1;
466             uniqueFound = Search(&temp, &nullityNode, 5, " ", newRandomValue) ?
              false : true;
467         } while (!uniqueFound);
468         return newRandomValue;  // Return the unique key
469 } // GetSeatAvailable()
470
471
472
473 // Autofill List
474 // ============================================
475 // Documentation:
476 //  This function will automatically populate and generate a reasonably
477 //   sized list.
478 // Notes:
479 //   I feel like a tool for using 'babynames.com' to populate
480 //   with random names....
481 // --------------------------------------------
```

```
482 // Parameters:
483 //  head [Node*]
484 //      This will take any valid link list.
485 // ============================================
486 void Reservation::Autofill_List()
487 {
488     // __HARD_CODED__
489     // Update this algorithm with caution!
490     //---
491     // To auto-generate, we are going to throw hard-coded values
492     // to the link list.  This will greatly allow us to debug or
493     // generally work with the list.
494     for (int i = 0; i < 20; i++)
495     {
496         switch (i)
497         {
498         case 0:
499             CreateNewNode(                      // primary list
500                         "Fabian",                   // First Name
501                         "Nadie",                    // Last Name
502                         Autofill_List_Numbers(0),   // Passenger ID
503                         Autofill_List_Numbers(1),   // Reservation Num
504                         Autofill_List_Numbers(2),   // Telephone Num
505                         GetSeatAvailable(),         // Seat Num
506                         Autofill_List_MealChoice());// Preferred Meal
507             break;
508         case 1:
509             CreateNewNode(                      // primary list
510                         "Ganit",                    // First Name
511                         "Ume",                      // Last Name
512                         Autofill_List_Numbers(0),   // Passenger ID
513                         Autofill_List_Numbers(1),   // Reservation Num
514                         Autofill_List_Numbers(2),   // Telephone Num
515                         GetSeatAvailable(),         // Seat Num
516                         Autofill_List_MealChoice());// Preferred Meal
517             break;
518         case 2:
519             CreateNewNode(                      // primary list
520                         "Dan",                      // First Name
521                         "Randi",                    // Last Name
522                         Autofill_List_Numbers(0),   // Passenger ID
523                         Autofill_List_Numbers(1),   // Reservation Num
524                         Autofill_List_Numbers(2),   // Telephone Num
525                         GetSeatAvailable(),         // Seat Num
526                         Autofill_List_MealChoice());// Preferred Meal
527             break;
528         case 3:
529             CreateNewNode(                  // primary list
530                         "Reese",                    // First Name
531                         "Nafisa",                   // Last Name
532                         Autofill_List_Numbers(0),   // Passenger ID
533                         Autofill_List_Numbers(1),   // Reservation Num
```

```
534                         Autofill_List_Numbers(2),   // Telephone Num
535                         GetSeatAvailable(),         // Seat Num
536                         Autofill_List_MealChoice());// Preferred Meal
537             break;
538         case 4:
539             CreateNewNode(                      // primary list
540                         "Nina",                     // First Name
541                         "Albany",                   // Last Name
542                         Autofill_List_Numbers(0),   // Passenger ID
543                         Autofill_List_Numbers(1),   // Reservation Num
544                         Autofill_List_Numbers(2),   // Telephone Num
545                         GetSeatAvailable(),         // Seat Num
546                         Autofill_List_MealChoice());// Preferred Meal
547             break;
548         case 5:
549             CreateNewNode(                      // primary list
550                         "Alexis",                   // First Name
551                         "Wayne",                    // Last Name
552                         Autofill_List_Numbers(0),   // Passenger ID
553                         Autofill_List_Numbers(1),   // Reservation Num
554                         Autofill_List_Numbers(2),   // Telephone Num
555                         GetSeatAvailable(),         // Seat Num
556                         Autofill_List_MealChoice());// Preferred Meal
557             break;
558         case 6:
559             CreateNewNode(                      // primary list
560                         "Rani",                     // First Name
561                         "Falcon",                   // Last Name
562                         Autofill_List_Numbers(0),   // Passenger ID
563                         Autofill_List_Numbers(1),   // Reservation Num
564                         Autofill_List_Numbers(2),   // Telephone Num
565                         GetSeatAvailable(),         // Seat Num
566                         Autofill_List_MealChoice());// Preferred Meal
567             break;
568         case 7:
569             CreateNewNode(                      // primary list
570                         "Yasmine",                  // First Name
571                         "Benicia",                  // Last Name
572                         Autofill_List_Numbers(0),   // Passenger ID
573                         Autofill_List_Numbers(1),   // Reservation Num
574                         Autofill_List_Numbers(2),   // Telephone Num
575                         GetSeatAvailable(),         // Seat Num
576                         Autofill_List_MealChoice());// Preferred Meal
577             break;
578         case 8:
579             CreateNewNode(                      // primary list
580                         "Al",                       // First Name
581                         "Bundy",                    // Last Name
582                         Autofill_List_Numbers(0),   // Passenger ID
583                         Autofill_List_Numbers(1),   // Reservation Num
584                         Autofill_List_Numbers(2),   // Telephone Num
585                         GetSeatAvailable(),         // Seat Num
```

```
586                         Autofill_List_MealChoice());// Preferred Meal
587             break;
588         case 9:
589             CreateNewNode(                      // primary list
590                         "Janeeva",                  // First Name
591                         "Zaina",                    // Last Name
592                         Autofill_List_Numbers(0),   // Passenger ID
593                         Autofill_List_Numbers(1),   // Reservation Num
594                         Autofill_List_Numbers(2),   // Telephone Num
595                         GetSeatAvailable(),         // Seat Num
596                         Autofill_List_MealChoice());// Preferred Meal
597             break;
598         case 10:
599             CreateNewNode(                      // primary list
600                         "Ofra",                     // First Name
601                         "Sable",                    // Last Name
602                         Autofill_List_Numbers(0),   // Passenger ID
603                         Autofill_List_Numbers(1),   // Reservation Num
604                         Autofill_List_Numbers(2),   // Telephone Num
605                         GetSeatAvailable(),         // Seat Num
606                         Autofill_List_MealChoice());// Preferred Meal
607             break;
608         case 11:
609             CreateNewNode(                      // primary list
610                         "Nadalia",                  // First Name
611                         "Hao",                      // Last Name
612                         Autofill_List_Numbers(0),   // Passenger ID
613                         Autofill_List_Numbers(1),   // Reservation Num
614                         Autofill_List_Numbers(2),   // Telephone Num
615                         GetSeatAvailable(),         // Seat Num
616                         Autofill_List_MealChoice());// Preferred Meal
617             break;
618         case 12:
619             CreateNewNode(                      // primary list
620                         "Hana",                     // First Name
621                         "Starr",                    // Last Name
622                         Autofill_List_Numbers(0),   // Passenger ID
623                         Autofill_List_Numbers(1),   // Reservation Num
624                         Autofill_List_Numbers(2),   // Telephone Num
625                         GetSeatAvailable(),         // Seat Num
626                         Autofill_List_MealChoice());// Preferred Meal
627             break;
628         case 13:
629             CreateNewNode(                  // primary list
630                         "Ashia",                    // First Name
631                         "Baeddan",                  // Last Name
632                         Autofill_List_Numbers(0),   // Passenger ID
633                         Autofill_List_Numbers(1),   // Reservation Num
634                         Autofill_List_Numbers(2),   // Telephone Num
635                         GetSeatAvailable(),         // Seat Num
636                         Autofill_List_MealChoice());// Preferred Meal
637             break;
```

```
638         case 14:
639             CreateNewNode(                      // primary list
640                         "Qi",                       // First Name
641                         "Wahponjea",                // Last Name
642                         Autofill_List_Numbers(0),   // Passenger ID
643                         Autofill_List_Numbers(1),   // Reservation Num
644                         Autofill_List_Numbers(2),   // Telephone Num
645                         GetSeatAvailable(),         // Seat Num
646                         Autofill_List_MealChoice());// Preferred Meal
647             break;
648         case 15:
649             CreateNewNode(                      // primary list
650                         "Hali",                     // First Name
651                         "Eamon",                    // Last Name
652                         Autofill_List_Numbers(0),   // Passenger ID
653                         Autofill_List_Numbers(1),   // Reservation Num
654                         Autofill_List_Numbers(2),   // Telephone Num
655                         GetSeatAvailable(),         // Seat Num
656                         Autofill_List_MealChoice());// Preferred Meal
657             break;
658         case 16:
659             CreateNewNode(                      // primary list
660                         "Tai Yang",                 // First Name
661                         "Taipa",                    // Last Name
662                         Autofill_List_Numbers(0),   // Passenger ID
663                         Autofill_List_Numbers(1),   // Reservation Num
664                         Autofill_List_Numbers(2),   // Telephone Num
665                         GetSeatAvailable(),         // Seat Num
666                         Autofill_List_MealChoice());// Preferred Meal
667             break;
668         case 17:
669             CreateNewNode(                      // primary list
670                         "Achava",                   // First Name
671                         "Nili",                     // Last Name
672                         Autofill_List_Numbers(0),   // Passenger ID
673                         Autofill_List_Numbers(1),   // Reservation Num
674                         Autofill_List_Numbers(2),   // Telephone Num
675                         GetSeatAvailable(),         // Seat Num
676                         Autofill_List_MealChoice());// Preferred Meal
677             break;
678         case 18:
679             CreateNewNode(                      // primary list
680                         "John",                     // First Name
681                         "Hancock",                  // Last Name
682                         Autofill_List_Numbers(0),   // Passenger ID
683                         Autofill_List_Numbers(1),   // Reservation Num
684                         Autofill_List_Numbers(2),   // Telephone Num
685                         GetSeatAvailable(),         // Seat Num
686                         Autofill_List_MealChoice());// Preferred Meal
687             break;
688         case 19:
689             CreateNewNode(                      // primary list
```

```
690                         "Theodore",                 // First Name
691                         "Roosevelt",                // Last Name
692                         Autofill_List_Numbers(0),   // Passenger ID
693                         Autofill_List_Numbers(1),   // Reservation Num
694                         Autofill_List_Numbers(2),   // Telephone Num
695                         GetSeatAvailable(),         // Seat Num
696                         Autofill_List_MealChoice());// Preferred Meal
697             break;
698         default: // Easter Egg!
699             CreateNewNode(                      // primary list
700                         "Jenny",                    // First Name
701                         "Tommy Tutone",             // Last Name
702                         -919,                       // Passenger ID [919 area code ;)]
703                         -919,                       // Reservation Num
704                         8675309,                    // Telephone Num [reference: https://youtu.be/8ou6DDG5e7I ]
705                         919,                        // Seat Num
706                         "MRE");                     // Preferred Meal - Military acronym for 'Meal Ready to Eat', its horrible.
707             break;
708         } // switch
709     } // for
710 } // Autofill_List()
711
712
713
714 // User Input [String]
715 // ===============================================
716 // Documentation:
717 //  This function will allow the user to input a specific string into the program.
718 // -----------------------------------------------
719 // Parameters:
720 //  UsePrompt [bool]
721 //      If true, this will provide the python'ish prompt.
722 // -----------------------------------------------
723 // Output:
724 //  string
725 //      Returns a string captured from STDIN.
726 // ===============================================
727 std::string Reservation::UserInput_String(bool UsePrompt = true)
728 {
729     std::string userInput;      // Use this to capture the STDIN
730
731     if (UsePrompt)
732         std::cout << ">>>>> ";  // The python'ish prompt
733
734     std::cin >> userInput;      // Capture the input
735
736     return userInput;           // Return the value.
737 } // 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.
// ------------------------------------------------
```

```
788 // Parameters:
789 //  UsePrompt [bool]
790 //      If true, this will provide the python'ish prompt.
791 // ------------------------------------------------
792 // Output:
793 //  bool
794 //      Returns a bool captured from STDIN.
795 // ================================================
796 bool Reservation::UserInput_Bool(bool UsePrompt = true)
797 {
798     char userInput;             // Use this to capture the STDIN
799 
800     if (UsePrompt)
801         std::cout << ">>>>> ";  // The python'ish prompt
802 
803     std::cin >> userInput;      // Capture the input
804 
805 
806     if (tolower(userInput) == 'y') // See if 'Yes' was selected
807         return true;        // Yes
808     else
809         return false;       // No
810 } // UserInput_Bool()
811 
812 
813 
814 // Manual Customer Add [Meal Choice]
815 // ================================================
816 // Documentation:
817 //  I have decided to separate this chunk of code into its own function
818 //  this way its a bit easier and the parent function is not so bloated.
819 // ------------------------------------------------
820 // Output:
821 //  string
822 //      Returns the preferred meal choice
823 // ================================================
824 std::string Reservation::ManualCustomerAdd_MealChoice()
825 {
826     // We will use this to store the user's choice and then process it later.
827     int userChoice;
828 
829     // Provide the in-flight meal list:
830     std::cout << "Select a number: " << std::endl
831         << " 1) Monkey Brains" << std::endl
832         << " 2) Tuna Eyeballs" << std::endl
833         << " 3) Raw Octopus" << std::endl
834         << " 4) Fish" << std::endl
835         << " 5) Expired Peanuts" << std::endl << std::endl;
836 
837     // Prevent bad input; run away protection
838     bool badInputCatch;
839     do
```

```
840     {
841         // Get the customer's request
842         userChoice = UserInput_Number();
843 
844         // Process the user's request
845         switch (userChoice)
846         {
847         case 1:                 // Monkey Brains
848             return "Monkey Brains";
849             break;
850         case 2:                 // Tuna Eyeballs
851             return "Tuna Eyeballs";
852             break;
853         case 3:                 // Raw Octopus
854             return "Raw Octopus";
855             break;
856         case 4:                 // Fish [Seriously, don't go for the fish!
              https://youtu.be/rQbj9uvYL8I ]
857             return "Fish";
858             break;
859         case 5:                 // Expired Peanuts
860             return "Expired Peanuts";
861             break;
862         default:                // Bad Input
863             std::cout << "Incorrect option!" << std::endl;
864             badInputCatch = true;
865             break;
866         } // switch
867     } while (badInputCatch);
868 } // ManualCustomerAdd_MealChoice()
869 
870 
871 
872 // Manual Customer add
873 // ===============================================
874 // Documentation:
875 //  This will allow the end-user to manually create a new entry within the list.
876 //   Capture all fields possible and then run through the importing algorithm.
877 // -----------------------------------------------
878 // Parameters:
879 //  head [Node**]
880 //      The list in which is to be appended
881 // ===============================================
882 void Reservation::ManualCustomerAdd()
883 {
884     // Declarations
885     // ------------------
886     std::string stdinNameFirst;
887     std::string stdinNameLast;
888     int stdinPhone;
889     std::string stdinMealChoice;
890     int stdinSeat;
```

```
891     // ----
892     // working variables
893     bool cacheBit;
894     // ------------------
895
896
897     // Provide instructions to the user
898     std::cout << "Please provide the following information:" << std::endl <<
          std::endl;
899
900     // ----
901
902     // Capture first name:
903     std::cout << "First name: ";
904     stdinNameFirst = UserInput_String(false);
905     std::cout << std::endl;
906
907     // Capture last name:
908     std::cout << "Last name: ";
909     stdinNameLast = UserInput_String(false);
910     std::cout << std::endl;
911
912     // Capture telephone number:
913     std::cout << "Telephone number: ";
914     stdinPhone = UserInput_Number(false);
915     std::cout << std::endl;
916
917     // Capture preferred meal:
918     std::cout << "In-flight meal choice: " << std::endl;
919     stdinMealChoice = ManualCustomerAdd_MealChoice();
920
921     // Ask the user about preferred seating
922     int seatCheck;      // Used for inspecting if seat is available
923     bool seatConfirmed; // Used for confirming if the seat is available.
924
925     do  // To avoid errors or frustrating the end-user, put this
926     {   // question in a loop and easily escapable via auto-seat-placement.
927         // [optional] Seat
928         std::cout << "Preferred seating arrangement? [Y] = Yes | [N] = No" <<
              std::endl;
929
930         // Capture user input for seating preference
931         cacheBit = UserInput_Bool();
932
933         if (cacheBit)
934         {                   // The customer has seating arrangements
935             std::cout << "Preferred seating: ";
936             stdinSeat = UserInput_Number(false);
937
938             // Check to make sure that the seat is available
939             if (GetSeatAvailable(stdinSeat) != -255)
940                 seatConfirmed = true;   // Seat is available!
```

```
941             else
942             {
943                 seatConfirmed = false;  // Seat is not available
944                 std::cout << std::endl << "This seat is presently not
                      available!" << std::endl;
945             } // else
946         } // if
947         else        // Automatically find the next available seat
948         {
949             stdinSeat = GetSeatAvailable(); // Automatically find available seat
950             seatConfirmed = true;           // Auto-confirm the seat being
                  available.
951         } // else
952     } while (!seatConfirmed);
953
954
955     std::cout << std::endl;
956
957     // ----
958
959     // Now that we have the information we need, now lets try to add this new
          entry into the list!
960     CreateNewNode(              // primary list
961         stdinNameFirst,             // First Name
962         stdinNameLast,              // Last Name
963         Autofill_List_Numbers(0),   // Passenger ID [919 area code ;)]
964         Autofill_List_Numbers(1),   // Reservation Num
965         stdinPhone,                 // Telephone Num [reference: https://
              youtu.be/8ou6DDG5e7I ]
966         stdinSeat,                  // Seat Num
967         stdinMealChoice);           // Preferred Meal -  Military acronym for
              'Meal Ready to Eat', its horrible.
968 } // ManualCustomerAdd()
969
970
971
972 // Search Node Entries
973 // ===========================================
974 // Documentation:
975 //  This is a very important function that will promptly scan the entire list
976 //   for a specific key.
977 //  This function will scan all nodes available, but it will stop at the
978 //   first available hit from the scan.  With that, once the scan finds the
979 //   key in one of the nodes, this function will NOT continue scanning
        afterwards.
980 // -------------------------------------------
981 // Methodology:
982 //  Using both Node types, cur and pre, we will perform the scan as follows:
983 // | ====== |     | ====== |     | ====== |     | ====== |     | ====== |
984 // | NODE 1 |     | NODE 2 |     | NODE 3 |     | NODE N |     | NODE N+1
985 // | ~~~~~~ |     | ~~~~~~ |     | ~~~~~~ |     | ~~~~~~ |     | ~~~~~~ |
986 // |  DATA  |     |  DATA  |     |  DATA  |     |  DATA  |     |  DATA  |
```

```
987  // | ------ |     | ------ |     | ------ |     | ------ |     | ------ |
988  // |  NEXT  ----> |  NEXT  ----> |  NEXT  ----> |  NEXT  ----> |  NEXT  ---->
989  // | ====== |     | ====== |     | ====== |     | ====== |     | ====== |
990  //    ---            pre            cur            <NOT YET SCANNED>
991  // Simple Logic:
992  // Fist scan:
993  // cur = node 1 || pre = NULL
994  // Second scan:
995  // cur = node 2 || pre = node 1
996  // Third scan:
997  // cur = node 3 || pre = node 2
998  // N scan: <general form>
999  // cur = node N || pre = node N-1
1000 // ------------------------------------------------
1001 // Parameters:
1002 //  cur [Node** - Alterable]
1003 //      Used for processing; holds the search point
1004 //  pre [Node** - Alterable]
1005 //      Used for processing before 'cur'; holds the node right before 'cur'
       pointer.
1006 //  searchMode [int]
1007 //      Determines how the lists will be scanned within this function.
1008 //      0 = Scan Last name [string]
1009 //      1 = Scan first name [string]
1010 //      2 = Scan passenger ID [int]
1011 //      3 = Scan reservation number [int]
1012 //      4 = Scan telephone number [int]
1013 //      5 = Scan seat number [int]
1014 //  searchKeyString [string]
1015 //      A specific key in a string form used for scanning each node.
1016 //  searchKeyInt [int]
1017 //      A specific key in an int form used for scanning each node.
1018 // ------------------------------------------------
1019 // Output:
1020 //  bool
1021 //      Reports the status if the operation was successful or failed.
1022 //      true = an error occurred
1023 //      false = operation successful
1024 // ================================================
1025 bool Reservation::Search(Node** cur, Node** pre, int searchMode, std::string
       searchKeyString = "NA", int searchKeyInt = -255)
1026 {
1027     // If the cur pointer points to NULL, then there is nothing to scan.
1028     if (*cur == NULL)
1029         return false;
1030
1031     // Scan the node
1032     while (*cur != NULL)
1033     {
1034         // Besides using a nesting conditional statement, we are going to use
1035         //  a switch statement for simplicity.
1036         switch (searchMode)
```

```
1037         {
1038         case 0:     // Scan last name
1039             if (searchKeyString.compare((*cur)->nameLast) == 0)
1040                                 // Equality between the two strings
1041                                 // [ http://www.cplusplus.com/reference/
                         string/string/compare/ ]
1042                 return true;
1043             break;
1044         case 1:     // Scan first name
1045             if (searchKeyString.compare((*cur)->nameFirst) == 0)
1046                                 // Equality between the two strings
1047                                 // [ http://www.cplusplus.com/reference/
                         string/string/compare/ ]
1048                 return true;
1049             break;
1050         case 2:     // Scan passenger ID
1051             if ((*cur)->passengerID == searchKeyInt)
1052                 return true;
1053             break;
1054         case 3:     // Scan reservation number
1055             if ((*cur)->reservationNum == searchKeyInt)
1056                 return true;
1057             break;
1058         case 4:     // Scan telephone number
1059             if ((*cur)->telephoneNum == searchKeyInt)
1060                 return true;
1061             break;
1062         case 5:     // Scan seat number
1063             if ((*cur)->seatNum == searchKeyInt)
1064                 return true;
1065             break;
1066         } // switch
1067
1068         // Update the node positions
1069         *pre = (*cur);              // Update pre to cur's current position.
1070         *cur = (*cur)->next;        // Shift cur to next position.
1071     } // while
1072
1073     // Unable to find a node with that specific key and data.
1074     return false;
1075 } // Search()
1076
1077
1078
1079 // List Size
1080 // ============================================
1081 // Documentation:
1082 //  This function will scan through the entire list and evaluate its
1083 //  total size, thus providing with the entire list size allocated.
1084 // --------------------------------------------
1085 // Output:
1086 //  List Size
```

```
1087 //      Return the list size
1088 // ============================================
1089 int Reservation::ListSize()
1090 {
1091     Node* temp = head;      // Directly copy the address of the 'head' in which
1092                             // it will be evaluated.  We will not do this       ↵
                        directly
1093                             // with the primary list.
1094     int counter = 0;        // This will retain a count of how many nodes exists ↵
           within
1095                             // the list.
1096
1097     for (; temp != NULL; temp = temp->next)
1098         counter++;
1099
1100     return counter;         // Return the size to the calling function
1101 } // ListSize()
1102
1103
1104
1105 // Find and Print Passenger
1106 // ============================================
1107 // Documentation:
1108 //  This function will allow the end-user to search for a specific passenger
1109 //  and output the passenger's information on the terminal.
1110 //
1111 //  This function will depend on several functions:
1112 //   Search() && Print_Passenger_List() && UserInput_String()
1113 //   UserInput_Number()
1114 // ============================================
1115 void Reservation::FindPrintPassenger()
1116 {
1117     // Present the user with a menu in which to search by
1118     std::cout << "Search for Passenger:" << std::endl
1119         << "---------------------" << std::endl
1120         << " 1) Last name" << std::endl
1121         << " 2) Telephone" << std::endl
1122         << " 3) Reservation ID" << std::endl
1123         << " 4) Passenger ID" << std::endl
1124         << " 5) Seat Number" << std::endl
1125         << " 0) Exit" << std::endl
1126         << std::endl;
1127
1128     bool badInput;  // Make sure that the input provided from the end-user
1129                     // is valid; run-away protection.
1130     std::string captureString;
1131     int captureInt;
1132
1133     Node* nodeIndex = head;
1134     Node* nullityNode = NULL;
1135
1136     do {
```

```
1137          // Capture the user's request and process the request
1138          switch (UserInput_Number())
1139          {
1140          case 1:                     // Last name
1141              // Capture the last name from the end-user
1142              std::cout << "Enter passenger's last name: ";
1143              captureString = UserInput_String(false);
1144              std::cout << std::endl;
1145
1146              if (Search(&nodeIndex,  // Our list to be scanned and processed.
1147                  &nullityNode,       // Required for the function, but not used.
1148                  0,                  // Search by last name
1149                  captureString))     // String to search
1150                  Print_Passenger_List(nodeIndex, true);// Output the results
1151              else
1152                  std::cout << "Unable to find passenger: " << captureString
1153                  << std::endl;
1154
1155              badInput = false;
1156              break;
1157          case 2:                     // Telephone
1158              std::cout << "Enter passenger's telephone number: ";
1159              captureInt >> UserInput_Number(false);
1160              std::cout << std::endl;
1161
1162              if (Search(&nodeIndex,  // Our list to be scanned and processed.
1163                  &nullityNode,       // Required for the function, but not used.
1164                  4,                  // Search by telephone number
1165                  "NA",               // Default to 'NA' due to standard; unused.
1166                  captureInt))        // integer to search
1167                  Print_Passenger_List(nodeIndex, true);// Output the results
1168              else
1169                  std::cout << "Unable to find passenger with the telephone
                      number: " << captureInt
1170                  << std::endl;
1171
1172              badInput = false;
1173              break;
1174          case 3:                     // Reservation ID
1175              std::cout << "Enter passenger's Reservation number: ";
1176              captureInt = UserInput_Number(false);
1177              std::cout << std::endl;
1178
1179              if (Search(&nodeIndex,  // Our list to be scanned and processed.
1180                  &nullityNode,       // Required for the function, but not used.
1181                  3,                  // Search by reservation number
1182                  "NA",               // Default to 'NA' due to standard; unused.
1183                  captureInt))        // integer to search
1184                  Print_Passenger_List(nodeIndex, true);// Output the results
1185              else
1186                  std::cout << "Unable to find passenger with the reservation
                      number: " << captureInt
```

```
1187                 << std::endl;
1188
1189             badInput = false;
1190             break;
1191         case 4:                     // Passenger ID
1192             std::cout << "Enter passenger's Passenger number: ";
1193             captureInt = UserInput_Number(false);
1194             std::cout << std::endl;
1195
1196             if (Search(&nodeIndex,  // Our list to be scanned and processed.
1197                 &nullityNode,       // Required for the function, but not used.
1198                 2,                  // Search by passenger number
1199                 "NA",               // Default to 'NA' due to standard; unused.
1200                 captureInt))        // integer to search
1201                 Print_Passenger_List(nodeIndex, true);// Output the results
1202             else
1203                 std::cout << "Unable to find passenger with the passenger
                      number: " << captureInt
1204                 << std::endl;
1205
1206             badInput = false;
1207             break;
1208         case 5:                     // Seat Number [Occupied]
1209             std::cout << "Enter passenger's Seat number: ";
1210             captureInt = UserInput_Number(false);
1211             std::cout << std::endl;
1212
1213             if (Search(&nodeIndex,  // Our list to be scanned and processed.
1214                 &nullityNode,       // Required for the function, but not used.
1215                 5,                  // Search by seat number
1216                 "NA",               // Default to 'NA' due to standard; unused.
1217                 captureInt))        // integer to search
1218                 Print_Passenger_List(nodeIndex, true);// Output the results
1219             else
1220                 std::cout << "Unable to a find a passenger in seat number: " <<
                      captureInt
1221                 << std::endl;
1222
1223             badInput = false;
1224             break;
1225         case 0:                     // Exit; silently leave this function
1226             badInput = false;
1227             break;
1228         default:                    // Bad input
1229             std::cout << "Incorrect option!" << std::endl;
1230             badInput = true;
1231             break;
1232         } // switch
1233     } while (badInput);
1234 } // FindPrintPassenger()
1235
1236
```

```
1237
1238  // Cancel Reservation
1239  //                                                                              ↵
        ============================================================================ ↵
        ========================
1240  // Documentation:
1241  // This code allows the user to search for a passenger in the list then remove  ↵
        them from said list.
1242  // Logic:
1243  // This code is a modified version of the Search function that only searches for ↵
         passenger Last names.
1244  // When it finds the Last Name it will delete the Node the Last Name is         ↵
        connected to.
1245  //                                                                              ↵
        ============================================================================ ↵
        ==========================
1246  bool Reservation::delete_node()
1247  {
1248          Node* pre = head;
1249          Node* temp = head;
1250          std::string captureString;
1251
1252
1253          if (temp == NULL){   //makes sure there is something in head
1254                  std::cout << "The list is empty." <<std::endl;
1255                  return false;
1256          }
1257          std::cout << "Which passenger would you like to delete? Enter their last ↵
                name:  ";
1258
1259          captureString = UserInput_String(false);    //captures user inputed last ↵
                name
1260
1261          if (Search(&temp,       // Our list to be scanned and processed.
1262                  &pre,       // the node before the desired node will be stored  ↵
                    here
1263                  0,                  // Search by last name
1264                  captureString)) {   // String to search
1265
1266              if(head == temp){
1267                  head = temp->next;
1268              }else{
1269                  pre->next = temp->next; // sets the prev node in the list to    ↵
                      point to the node after the node
1270              }                           // that will be deleted
1271              delete temp; // deletes the node if the node is the node that needs ↵
                  to be deleted
1272
1273              std::cout << "The passenger was removed from the list" << std::endl;
1274              return true;
1275          }else{
1276              std::cout <<"The passenger was not found" <<std::endl;
```

```
1277                return false;
1278            }
1279
1280    }
1281
1282
1283
1284    // Update Passenger Information
1285    // ============================================
1286    // Documentation:
1287    //  This function will provide a front-end to the end-user to identify what
1288    //  passenger's information is to be updated.
1289    //  This function will first ask the end-user to input the current information
1290    //   about the passenger, then with this information - we scan the list for the
          node.
1291    //   IIF if the node exists with the passenger's information, then will provide
          the
1292    //   end-user with update options for that passenger.
1293    //
1294    // NOTES: This function was coded badly and this should be better optimized, but
1295    //          as we work at a coding sweat shop, there is no better way!
1296    //          The pay grade at this code sweat shop only offers 27 cents per
          hour :(
1297    //          Please help us!
1298    //
1299    //  This function depends on the Search() and the input (number\int) function.
1300    // ============================================
1301    void Reservation::UpdatePassengerInformation()
1302    {
1303        // Present the user with a menu in which to search by
1304        std::cout << "Search for Passenger:" << std::endl
1305            << "----------------------" << std::endl
1306            << " 1) Last name" << std::endl
1307            << " 2) Telephone" << std::endl
1308            << " 3) Reservation number" << std::endl
1309            << " 4) Passenger number" << std::endl
1310            << " 5) Seat Number" << std::endl
1311            << " 0) Exit" << std::endl
1312            << std::endl;
1313
1314        bool badInput;  // Make sure that the input provided from the end-user
1315                        // is valid; run-away protection.
1316        std::string captureString;
1317        int captureInt;
1318
1319        Node* nullityNode = NULL;   // Create a list placeholder; we will not need
            it in this function, but
1320                                    // it is required when calling the Search()
                        function.
1321
1322        bool targetFound = false;   // We will use this to determine if the
            passenger was
```

```
1323                                    // detect during the search.  This will become
                        true
1324                                    // IIF the passenger was found, otherwise - its
                        false.
1325                                    // IIF this is true, then we will allow the user
                         to update
1326                                    //  that node index.
1327
1328        Node* temp = head;          //used so the head pointer is not wrongly
              updated
1329
1330        do {
1331            // Capture the user's request and process the request
1332            switch (UserInput_Number())
1333            {
1334            case 1:                     // Last name
1335                                        // Capture the last name from the end-user
1336                std::cout << "Enter passenger's last name: ";
1337                captureString = UserInput_String(false);
1338                std::cout << std::endl;
1339
1340                if (Search(&temp,       // Our list to be scanned and processed.
1341                    &nullityNode,       // Required for the function, but not used.
1342                    0,                  // Search by last name
1343                    captureString))     // String to search
1344                    targetFound = true;
1345                else
1346                    std::cout << "Unable to find passenger: " << captureString
1347                    << std::endl;
1348
1349                badInput = false;
1350                break;
1351            case 2:                     // Telephone
1352                std::cout << "Enter passenger's telephone number: ";
1353                captureInt >> UserInput_Number(false);
1354                std::cout << std::endl;
1355
1356                if (Search(&temp,       // Our list to be scanned and processed.
1357                    &nullityNode,       // Required for the function, but not used.
1358                    4,                  // Search by telephone number
1359                    "NA",               // Default to 'NA' due to standard; unused.
1360                    captureInt))        // integer to search
1361                    targetFound = true;
1362                else
1363                    std::cout << "Unable to find passenger with the telephone
                          number: " << captureInt
1364                    << std::endl;
1365
1366                badInput = false;
1367                break;
1368            case 3:                     // Reservation ID
1369                std::cout << "Enter passenger's Reservation number: ";
```

```
1370            captureInt = UserInput_Number(false);
1371            std::cout << std::endl;
1372
1373            if (Search(&temp,       // Our list to be scanned and processed.
1374                &nullityNode,       // Required for the function, but not used.
1375                3,                  // Search by reservation number
1376                "NA",               // Default to 'NA' due to standard; unused.
1377                captureInt))        // integer to search
1378                targetFound = true;
1379            else
1380                std::cout << "Unable to find passenger with the reservation
                      number: " << captureInt
1381                << std::endl;
1382
1383            badInput = false;
1384            break;
1385        case 4:                     // Passenger ID
1386            std::cout << "Enter passenger's Passenger number: ";
1387            captureInt = UserInput_Number(false);
1388            std::cout << std::endl;
1389
1390            if (Search(&temp,       // Our list to be scanned and processed.
1391                &nullityNode,       // Required for the function, but not used.
1392                2,                  // Search by passenger number
1393                "NA",               // Default to 'NA' due to standard; unused.
1394                captureInt))        // integer to search
1395                targetFound = true;
1396            else
1397                std::cout << "Unable to find passenger with the passenger
                      number: " << captureInt
1398                << std::endl;
1399
1400            badInput = false;
1401            break;
1402        case 5:                     // Seat Number [Occupied]
1403            std::cout << "Enter passenger's Seat number: ";
1404            captureInt = UserInput_Number(false);
1405            std::cout << std::endl;
1406
1407            if (Search(&temp,       // Our list to be scanned and processed.
1408                &nullityNode,       // Required for the function, but not used.
1409                5,                  // Search by seat number
1410                "NA",               // Default to 'NA' due to standard; unused.
1411                captureInt))        // integer to search
1412                targetFound = true;
1413            else
1414                std::cout << "Unable to a find a passenger in seat number: " <<
                      captureInt
1415                << std::endl;
1416
1417            badInput = false;
1418            break;
```

```
1419          case 0:                     // Exit; silently leave this function
1420              badInput = false;
1421              break;
1422          default:                    // Bad input
1423              std::cout << "Incorrect option!" << std::endl;
1424              badInput = true;
1425              break;
1426          } // switch
1427      } while (badInput);
1428
1429      // If the passenger was not found, leave this function.
1430      if (!targetFound)
1431      {
1432          // Allow the end-user to view the message that the passenger
1433          // was not found during the scan.
1434          // ----
1435          std::cout << "Press the enter or return key to continue. . ." <<
                std::endl;      // This is alternative version of 'system("PAUSE")'
1436          std::cin.ignore();
                    // and will work outside of Windows.
1437          std::cin.ignore();  // avoid input ghosting?
1438          // ----
1439
1440          return;
1441      }
1442
1443
1444      // ============================================================================
1445      // ----------------------------------------------------------------------------
1446      // ============================================================================
1447      // TARGET FOUND
1448
1449      do
1450      {
1451
1452      // What does the end-user want to update?
1453      std::cout << "Information to Update:" << std::endl
1454          << "----------------------" << std::endl
1455          << " 1) Telephone" << std::endl
1456          << " 2) Reservation number" << std::endl
1457          << " 3) Passenger number" << std::endl
1458          << " 4) Seat Number" << std::endl
1459          << " 0) Exit" << std::endl
1460          << std::endl;
1461
1462
1463      // Get the user's input and evaluate it
1464
1465          switch (UserInput_Number())
1466          {
1467          case 1:             // Update telephone
1468              // Let the user know of the current value
```

```
1469            std::cout << "Current telephone number: " << temp->telephoneNum <<
                  std::endl;
1470            // Allow the user to update that specific field:
1471            std::cout << "Enter a new value: ";
1472            temp->telephoneNum = UserInput_Number(false);
1473
1474            badInput = false;
1475            break;
1476        case 2:             // Update reservation ID
1477            // Let the user know of the current value
1478            std::cout << "Current reservation number: " << temp->reservationNum
                  << std::endl;
1479            // Allow the user to update that specific field:
1480            std::cout << "Enter a new value: ";
1481            temp->reservationNum = UserInput_Number(false);
1482
1483            badInput = false;
1484            break;
1485        case 3:             // Update passenger ID
1486            // Let the user know of the current value
1487            std::cout << "Current passenger number: " << temp->passengerID <<
                  std::endl;
1488            // Allow the user to update that specific field:
1489            std::cout << "Enter a new value: ";
1490            temp->passengerID = UserInput_Number(false);
1491
1492            badInput = false;
1493            break;
1494        case 4:             // Seat Number
1495                int newSeatNum;
1496                // Let the user know of the current value
1497                std::cout << "Current seat number: " << temp->seatNum <<
                      std::endl;
1498                // Allow the user to update that specific field:
1499                std::cout << "Enter a new value: ";
1500
1501                newSeatNum = UserInput_Number(false);       //gathers user input
1502
1503                if(GetSeatAvailable(newSeatNum) != -255)    //checks to make
                      sure seat is not taken
1504                {
1505                    temp->seatNum = newSeatNum;     //changes seat number if it
                        is not taken
1506                        badInput = false;
1507                }else{
1508                    std::cout<< std::endl << "Seat already taken!" << std::endl;
1509                    badInput = true;
1510                }
1511            break;
1512        case 0:             // Exit; silently leave this function
1513            badInput = false;
1514            break;
```

```
1515          default:            // Bad input
1516              std::cout << "Incorrect option!" << std::endl;
1517              badInput = true;
1518              break;
1519          }
1520
1521          std::cout << std::endl;
1522      } while (badInput);
1523  } // UpdatePassengerInformation()
1524
1525
1526
1527
1528
1529  // Check In Passenger
1530  // ================================================
1531  // Documentation:
1532  //  This function will search for a user inputed passenger in a list
1533  //  and change the passengers status to checked in if he/she is found.
1534  // ------------------------------------------------
1535  // Output:
1536  //  changes passenger's status to checked in or says passenger not found.
1537  // ================================================
1538  void Reservation::CheckInPassenger()
1539  {
1540          Node* nullityNode = NULL;
1541          Node* temp = head;
1542          std::string captureString;
1543
1544
1545          if (temp == NULL){   //makes sure there is something in head
1546                  std::cout << "The list is empty.  There are no passengers to
                        check in" <<std::endl;
1547          }else{
1548              std::cout << "Which passenger would you like to Check in?  Enter
                      their last name:";
1549
1550              captureString = UserInput_String(false);    //captures user inputted
                     last name
1551
1552              if (Search(&temp,       // Our list to be scanned and processed.
1553                      &nullityNode,       // the node before the desired node will
                          be stored here
1554                      0,                  // Search by last name
1555                      captureString)) {   // String to search
1556
1557                  temp->checkedIn = true;
1558                  std::cout << temp->nameFirst << " " << temp->nameLast << " has
                        been checked in." << std::endl;
1559              }else{
1560                  std::cout << "This passenger is not in the list." <<std::endl;
1561              }
```

```
1562          }
1563  }//CheckInPassenger
1564
1565
1566
1567
1568  // Print Check In List
1569  // ============================================
1570  // Documentation:
1571  //  This function will produce the check in report
1572  // --------------------------------------------
1573  // Output:
1574  //  displays a list of all passengers in the list.  Will display
1575  //  their first and last names and whether they are checked in or not.
1576  //  it will also display the total amount of checked in passengers and
1577  //  the total amount of passengers not checked in
1578  // ============================================
1579  void Reservation::Print_CheckIn_List()
1580  {
1581      int CheckedInCnt = 0;   //counts the number of passengers checked in
1582      int NotCheckedInCnt = 0;    //counts the number of passengers not checked in
1583      Node* temp = head;      //starts at the beginning of the list
1584
1585      while(temp != NULL){
1586
1587          //prints each passenger's name and a checked in prompt
1588          std::cout << temp->nameFirst <<" " <<temp->nameLast << std::endl
1589                    << "Checked In?:  ";
1590
1591          //checks to see if each passenger is checked in
1592          if(temp->checkedIn){
1593              std::cout << "Yes" << std::endl << std::endl;
1594              CheckedInCnt++;
1595          }else{
1596              std::cout << "No" << std::endl << std::endl;
1597              NotCheckedInCnt++;
1598          }
1599          temp = temp->next;
1600      }
1601
1602      std::cout<< std::endl << "There are " << CheckedInCnt << " Passengers
        checked in." <<std::endl;
1603      std::cout<< std::endl << "There are " << NotCheckedInCnt << " Passengers not
        checked in." <<std::endl;
1604  }   //Print Check In List
1605
1606
1607
1608
1609  // Print meal list
1610  // ============================================
1611  // Documentation:
```

```
1612 //  This function will display the passenger meal list report
1613 // ------------------------------------------------
1614 // Output:
1615 //  Outputs a list of all passengers.  Will display their meal choices and
1616 //  their first and last names.  It will also display the totals for each meal
1617 //  choice that is available.
1618 // ================================================
1619 void Reservation::Print_Meal_List()
1620 {
1621
1622     Node* temp = head;
1623
1624     if(temp != NULL){
1625         //these counters will count the amount of each meal that was chosen
1626         //for all passengers in the list
1627         int meal1Cnt = 0;
1628         int meal2Cnt = 0;
1629         int meal3Cnt = 0;
1630         int meal4Cnt = 0;
1631         int meal5Cnt = 0;
1632
1633
1634
1635         while(temp != NULL){    //increments until the end of the list is
           reached
1636
1637                                 //displays the passengers name and meal choice
1638             std::cout << temp->nameFirst << " " << temp->nameLast << std::endl
1639                       << "Meal Choice:  " << temp->mealType << std::endl <<
                     std::endl;
1640
1641                       //increments counter for whichever meal the passenger
                     chose.
1642             if(temp->mealType.compare("Monkey Brains") == 0){
1643                 meal1Cnt++;
1644             }else if(temp->mealType.compare("Tuna Eyeballs") == 0){
1645                 meal2Cnt++;
1646             }else if(temp->mealType.compare("Raw Octopus") == 0){
1647                 meal3Cnt++;
1648             }else if(temp->mealType.compare("Fish") == 0){
1649                 meal4Cnt++;
1650             }else if(temp->mealType.compare("Expired Peanuts") == 0){
1651                 meal5Cnt++;
1652             }
1653
1654
1655             temp = temp->next;  //moves to next passenger in the list
1656         }//while
1657
1658         //displays meal choice totals
1659         std::cout << "There are " << meal1Cnt << " orders for Monkey Brains." <<
              std::endl
```

```
1660                    << "There are " << meal2Cnt << " orders for Tuna Eyeballs." << ↵
                           std::endl
1661                    << "There are " << meal3Cnt << " orders for Raw Octopus." <<   ↵
                          std::endl
1662                    << "There are " << meal4Cnt << " orders for Fish." <<         ↵
                          std::endl
1663                    << "There are " << meal5Cnt << " orders for Expired Peanuts." ↵
                          << std::endl << std::endl;
1664
1665      }else{      //if for no list
1666
1667          std::cout << "There are currently no passengers in the list." <<        ↵
              std::endl;
1668      }
1669
1670  }//print meal list
1671
1672
1673  // Sort list
1674  // ============================================
1675  // Documentation:
1676  //  This function will put a passenger list in alphabetical order by last name
1677  //
1678  //  it does this by first checking each node against the first node.  if a node ↵
        that
1679  //  comes alphabetically before the first node is found, the function places    ↵
        that
1680  //  node just before the first node.  Then the process begins again with the new↵
         first
1681  //  node.  When no nodes are found that come alphabetically before the first    ↵
        node,
1682  //  the node that other nodes are compared to is moved to the second node, and  ↵
        so on
1683  //  until the node other nodes are compared to is the last node in the list.
1684  // --------------------------------------------
1685  // Output:
1686  //  Outputs a list sorted by last name.
1687  // ============================================
1688  void Reservation::Sort()
1689  {
1690      if(head != NULL)        //to make sure list is not empty
1691      {
1692              Node* current = head;       //this pointer contains the node being  ↵
                  challenged
1693              Node* challenger = head;    //this pointer contains the node        ↵
                  challenging the
1694                                          //current pointer for alphabetically    ↵
                        first position
1695
1696              Node* pre = head;           //this pointer is used to place the     ↵
                  challenger node before
1697                                          //the current node
```

```
1698
1699            Node* challengePre = head;  //this pointer is used to remove the challenger pointer from its old spot
1700
1701            bool swap = false;          //this variable is used to indicate that a swap was made\
1702
1703            std::string currentName;    //holds the current node's last name
1704
1705            std::string challengerName; //holds the challenger node's last name
1706
1707            while(current != NULL)      //iterates through until it reaches the end of the list, or
1708                                        //until a node that comes alphabetically before it is found
1709            {
1710                while(challenger->next != NULL)     //iterates through all nodes after the current node
1711                                                    //stops when it reaches the end of the list, or when
1712                                                    //it finds a node that comes alphabetically before the
1713                                                    //current node
1714                {
1715                    challengePre = challenger;
1716                    challenger = challenger->next;  //incrementing the challenger node to the next node
1717                                                    //in the list to be checked
1718
1719                    currentName = current->nameLast;            //setting the last name strings to be checked
1720                    challengerName = challenger->nameLast;
1721
1722                    if(Alphabetize(currentName, challengerName)){       //checking to see which node comes
1723                                                                        //alphabetically first by last name
1724                                                                        //if the challenger comes first,
1725                                                                        //it is placed just before the current
1726                                                                        //node in the list, and the process is restarted
1727                                                                        //for the new current node
1728                        challengePre->next = challenger->next;
1729                        if(current == head){        //for if the current node is at the head of the list
1730                            challenger->next = current;
1731                            head = challenger;
1732                        }else{
1733                            pre->next = challenger;
```

```
1734                            challenger->next = current;
1735                        }
1736                        current = challenger;
1737                        swap = true;        //indicates a swap was made
1738                        break;
1739                    }
1740                }//inner while
1741
1742                if(!swap)       //if no swaps were made, the current node is moved to the next node
1743                                //in the list.
1744                {
1745                    pre = current;
1746                    current = current->next;
1747                    challenger = current;
1748                }
1749                swap = false;
1750            }//outer while
1751
1752    }else{  //outer if
1753        std::cout << "The list is empty." << std::endl;
1754    }
1755 }//sort
1756
1757
1758
1759
1760 //Alphabetize
1761 // ================================================
1762 // Documentation:
1763 //  This function will decide which of two strings comes
1764 //  alphabetically first
1765 // ------------------------------------------------
1766 // Output:
1767 //  outputs true if the challenger string comes alphabetically
1768 //  before the current string
1769 //-------------------------------------------------
1770 //  Parameters:
1771 //  current and challenger hold strings that will be compared
1772 //  to determine which one comes alphabetically first
1773 // ================================================
1774 bool Reservation::Alphabetize(std::string current, std::string challenger)
1775 {
1776     if(current.length() <= challenger.length())     //checks to see which string is longer
1777                                                     //uses for loop to the shorter strings length
1778                                                     //this allows the function to show that the shorter
1779                                                     //word comes alphabetically first (i.e. Al, vs Ale)
1780     {
```

```
1781          for(int i = 0; i <current.length(); i++)
1782          {
1783              if(int(tolower(current.at(i))) < int(tolower(challenger.at(i))))
                      //this statement analyzes
1784              {                           //the current character and challenger
                    character at the current
1785                                        //iteration of the for loop.  It
                        converts the letters
1786                                        //to their ANSII equivalent values and
                        analyzes
1787                                        //those to see which one is larger.
                        capitol letters will be
1788                                        //converted to lowercase letters.  The
                        other if statements basically
1789                                        //follow the same concept as this one
1790
1791                  return false;   //if the current character is alphabetically
                      before the challenger character,
1792                                  //the function returns false to note that the
                        current string is alphabetically
1793                                  //first
1794              }else if(int(tolower(current.at(i))) > int(tolower(challenger.at
                    (i)))){
1795
1796                  return true;     //if the current character being analyzed is
                      alphabetically after the
1797                                  //challenger character being analyzed, the
                        function returns true to note that
1798                                  //the current string is alphabetically after
                        the challenger string
1799              }
1800          }
1801          return false;       //returns false if all letters analyzed where the
               same.  This means that either
1802                              //the challenger string is longer than the current
                        string (i.e al vs ale), or
1803                              //both strings are the same in which case no action
                        is taken.
1804
1805
1806      }else{      //for if the challenger string is shorter than the current
            string.  same concepts as above,
1807                  //accept for when current string is the same as challenger
                      string up to the last characters
1808                  //analyzed.  In this case the challenger string is
                      alphabetically first since it is shorter
1809                  //(i.e. ale vs al).
1810          for(int i = 0; i < challenger.length(); i++)
1811          {
1812              if(int(tolower(current.at(i))) < int(tolower(challenger.at(i))))
1813              {
```

```
1814                 return false;
1815             }else if(int(tolower(current.at(i))) > int(tolower(challenger.at
                   (i)))){
1816                 return true;
1817             }
1818         }
1819 
1820         return true;
1821     }//outer else
1822 
1823 }//Alphabetize
1824 #endif // !__LinkList__Implementation__
1825 
```