```cpp
#include <cmath>       // abs()
#include <cstddef>     // size_t
#include <iomanip>     // setprecision(), setw()
#include <iostream>    // cerr, ,clog, fixed(), showpoint(), left(), right()
#include <map>
#include <queue>
#include <stack>
#include <stdexcept>   // invalid_argument, out_of_range
#include <string>      // stod()

#include "Book.hpp"
#include "BookDatabase.hpp"



namespace
{
  // Output some observed behavior.
  // Call this function from within the carefully_move_books functions, just before kicking off the recursion and then just after each move.
  void trace( const std::stack<Book> & sourceCart, const std::stack<Book> & destinationCart, const std::stack<Book> & spareCart, std::ostream & s = std::clog )
  {
    // Count and label the number of moves
    static std::size_t move_number = 0;

    // First time called will bind parameters to copies
    static std::map<const std::stack<Book> *, std::stack<Book>> bookCarts = { { &sourceCart, {}           }, { &destinationCart, {}            }, { &spareCart, {}          } };
    static std::map<const std::stack<Book> *, std::string>      colLabels = { { &sourceCart, "Broken Cart" }, { &destinationCart, "Working Cart" }, { &spareCart, "Spare Cart" } };

    // Interrogating the stacks is a destructive process, so local copies of the parameters are made to work with.  The
    // carefully_move_books algorithm will swap the order of the arguments passed to this functions, but they will always be the
    // same objects - just in different orders. When outputting the stack contents, keep the original order so we humans can trace
    // the movements easier.  A container (std::map) indexed by the object's identity (address) is created so the canonical order
    // remains the same from one invocation to the next.
    bookCarts[&sourceCart]      = sourceCart;
    bookCarts[&destinationCart] = destinationCart;
    bookCarts[&spareCart]       = spareCart;


    // Determine the height of the tallest stack
    std::size_t tallestStackSize = 0;
    for( auto & [index, cart] : bookCarts )   if( cart.size() > tallestStackSize )   tallestStackSize = cart.size();


    // Print the header
    s << "After " << std::setw( 3 ) << move_number++ << " moves:     " << std::left;      // print the move number
    for( auto & [index, label] : colLabels )    s << std::setw( 23 ) << label;           // print the column labels
    s << std::right << "\n                      " << std::string( 23*3, '-' ) << '\n';    // underline the labels


    // Print the stack's contents
    for( ; tallestStackSize > 0;  --tallestStackSize )                                   // for each book on a cart
    {
      s << std::string( 21, ' ' );

      for( auto & [key, cart] : bookCarts )                                              // for each book cart
      {
        if( cart.size() == tallestStackSize )
        {
          auto && title = cart.top().title();
          if( title.size() > 20 ) title[17] = title[18] = title[19] = '.';                // replace last few characters of long titles with "..."
          s << std::left << std::setw( 23 ) << title.substr( 0, 20 ) << std::right;
```

```
62            cart.pop();
63          }
64          else
65          {
66            s << std::string( 23, ' ' );
67          }
68        }
69        s << '\n';
70      }
71      s << "                      " << std::string( 69, '=' ) << "\n\n\n\n";
72    }
73
74
75    /*****************************************************************************************************************
76    ** A recursive algorithm to carefully move books from a broken cart to a working cart is given as follows:
77    ** START
78    ** Procedure carefully_move_books (number_of_books_to_be_moved, broken_cart, working_cart, spare_cart)
79    **
80    **    IF number_of_books_to_be_moved == 1, THEN
81    **       move top book from broken_cart to working_cart
82    **       trace the move
83    **
84    **    ELSE
85    **       carefully_move_books (number_of_books_to_be_moved-1, broken_cart, spare_cart, working_cart)
86    **       move top book from broken_cart to working_cart
87    **       trace the move
88    **       carefully_move_books (number_of_books_to_be_moved-1, spare_cart, working_cart, broken_cart)
89    **
90    **    END IF
91    **
92    ** END Procedure
93    ** STOP
94    *****************************************************************************************************************/
95    void carefully_move_books( std::size_t quantity, std::stack<Book> & sourceCart, std::stack<Book> & destinationCart, std::stack<Book> & spareCart )
96    {
97      #ifndef STUDENT_TO_DO_REGION
98        /// Implement the algorithm above.
99        if( quantity == 1 )
100       {
101         // move book from source cart to destination cart
102         destinationCart.push( sourceCart.top() );
103         sourceCart.pop();
104         trace( sourceCart, destinationCart, spareCart );
105       }
106       else
107       {
108         carefully_move_books( quantity - 1, sourceCart, spareCart, destinationCart );
109
110         // move book from source cart to destination cart
111         destinationCart.push( sourceCart.top() );
112         sourceCart.pop();
113         trace( sourceCart, destinationCart, spareCart );
114
115         carefully_move_books( quantity - 1, spareCart, destinationCart, sourceCart );
116       }
117     #endif
118   }
119
120
121
122
```

```
123   void carefully_move_books( std::stack<Book> & from, std::stack<Book> & to )
124   {
125     #ifndef STUDENT_TO_DO_REGION
126       /// Implement the starter function for the above algorithm.  If the "from" cart contains books, move those books to the "to"
127       /// cart while ensuring the breakable books are always on top of the nonbreakable books, just like they already are in the
128       /// "from" cart.  That is, call the above carefully_move_books function to start moving books recursively.  Call the above
129       /// trace function just before calling carefully_move_books to get a starting point reference in the movement report.
130       if( !from.empty() )
131       {
132         std::stack<Book> temp;
133
134         trace( from, to, temp );
135         carefully_move_books( from.size(), from, to, temp );
136       }
137     #endif
138   }
139 }    // namespace
140
141
142
143
144
145 int main( int argc, char * argv[] )
146 {
147   // Snag an empty cart as I enter the grocery store
148   #ifndef STUDENT_TO_DO_REGION
149     /// Create an empty book cart as a stack of books and call it myCart.
150     std::stack<Book> myCart;
151   #endif
152
153
154
155
156   // Shop for awhile placing books into my book cart
157   #ifndef STUDENT_TO_DO_REGION
158     /// Put the following books into your cart with the heaviest book on the bottom and the lightest book on the top
159     /// according to the ordering given in the table below
160     ///
161     ///      ISBN            Title              Author
162     ///      --------------  -------------      ---------------
163     ///      9780895656926   Like the Animals   any                    <=== lightest book, put this on the top so heavy books wont break them
164     ///      54782169785     131 Answer Key     any
165     ///      0140444300      Les Mis            any
166     ///      9780399576775   Eat pray love      Asher
167     ///      9780545310581   Hunger Games       any                    <===  heaviest book, put this on the bottom
168
169     myCart.push( {"Hunger Games",       "",          "9780545310581"} );
170     myCart.push( {"Eat pray love",      "Asher",     "9780399576775"} );
171     myCart.push( {"Les Mis",            "",          "0140444300"   } );
172     myCart.push( {"131 Answer Key",     "",          "54782169785"  } );
173     myCart.push( {"Like the Animals",   "",          "9780895656926"} );
174   #endif
175
176
177
178   // A wheel on my cart has just broken and I need to move books to a new cart that works
179   #ifndef STUDENT_TO_DO_REGION
180     /// Create an empty book cart as a stack of books and call it workingCart.  Then carefully move the books in your
181     /// broken cart to this working cart by calling the above carefully_move_books function with two arguments.
182     std::stack<Book> workingCart;
183
```

```
184     carefully_move_books( myCart, workingCart );
185   #endif
186
187
188
189
190   // Time to checkout and pay for all this stuff.  Find a checkout line and start placing books on the counter's conveyor belt
191   #ifndef STUDENT_TO_DO_REGION
192     /// Create an empty checkout counter as a queue of books and call it checkoutCounter.  Then remove the books
193     /// from your working cart and place them on the checkout counter, i.e., put them in this checkoutCounter queue.
194     std::queue<Book> checkoutCounter;
195     while( ! workingCart.empty() )
196     {
197       checkoutCounter.push( workingCart.top() );
198       workingCart.pop();
199     }
200   #endif
201
202
203
204
205   // Now add it all up and print a receipt
206   double amountDue = 0.0;
207   BookDatabase & storeDataBase = BookDatabase::instance();              // Get a reference to the store's book database.
208                                                                         // The database will contains a full description of the
209                                                                         // book and the book's price.
210   #ifndef STUDENT_TO_DO_REGION
211     /// For each book in the checkout counter queue, find the book by ISBN in the store's database.  If the book on the counter is
212     /// found in the database then accumulate the amount due and print the book's full description and price on the receipt (i.e.
213     /// write the book's full description and price to standard output).  Otherwise, print a message on the receipt that a
214     /// description and price for the book wasn't found and there will be no charge.
215     while( !checkoutCounter.empty() )
216     {
217       if( auto book = storeDataBase.find( checkoutCounter.front().isbn() ); book != nullptr )    // look up book in database
218       {
219         amountDue += book->price();
220         std::cout << *book << '\n';
221       }
222       else
223       {
224         std::cout << '"' << checkoutCounter.front().isbn() << "\" (" << checkoutCounter.front().title() << ") not found, book is free!\n";
225       }
226       checkoutCounter.pop();
227     }
228   #endif
229
230
231
232   // Now check the receipt - are you getting charged the correct amount?
233   // You can either pass the expected total when you run the program by supplying a parameter, like this:
234   //    program 35.89
235   // or if no expected results provided at the command line, then prompt for and obtain expected result from standard input
236   double expectedAmmountDue = 0.0;
237   if( argc >= 2 )
238     try
239     {
240       expectedAmmountDue = std::stod( argv[1] );
241     }
242     catch( std::invalid_argument & ) {}  // ignore anticipated bad command line argument
243     catch( std::range_error &      ) {}  // ignore anticipated bad command line argument
244   else
```

```
245   {
246     std::cout << "What is your expected amount due?  ";
247     std::cin  >> expectedAmmountDue;
248   }
249
250
251   std::cout << std::fixed << std::setprecision( 2 ) << std::showpoint
252             << std::string( 25, '-' )  << '\n'
253             << "Total  $" << amountDue << "\n\n\n";
254
255
256   if( std::abs(amountDue - expectedAmmountDue) < 1E-4 ) std::clog << "PASS - Amount due matches expected\n";
257   else                                                  std::clog << "FAIL - You're not paying the amount you should be paying\n";
258
259   return 0;
260 }
261
```