```
// Tim Lum
// twhlum@gmail.com
// https://uwb-hibernaculum.slack.com/
// 2017.10.01
// For the University of Washington Bothell CSS 501A
// Autumn 2017, Graduate Certificate in Software Design & Development (GCSDD)
//
// File Description:
// Design document for Assignment 01 - Punchcards.
//
// Required files:
// None
//
// Acknowledgements:
// Source material from:
// University of Washington Bothell
// CSS 501A Data Structures And Object-Oriented Programming I
// "Design and Coding Standards"
// Michael Stiber
//
// Template author:
// Tim Lum (twhlum@gmail.com)
//
// Source file originally at:
// https://canvas.uw.edu/courses/1116064/pages/css-501-design-and-coding-standards
// (Deprecated by end of quarter; source closed. Content included as 'APPENDIX_02')
//
// License:
// This software is published under the GNU general license which guarantees end users
// the freedom to run, study, share and modify the software.
// https://www.gnu.org/licenses/gpl.html

// ---- BEGIN PRINTABLE CONTENT ----
```

Assignment 01 - Reading a Virtual Punchcard

--- **STEP 00** --- IS DESIGN DOCUMENTATION REQUIRED? ---

If 'yes' to any of the following, documentation is required:

I. Will more than one person work on or examine this code?
Y / **N**

II. Is the problem too large to mentally comprehend in its entirety?
Y / **N**

III. Will the code ever be revisited at a future date?
Y / N

IV. Will questions by other users ever arise regarding the code?
Y / N

--- **STEP 01** --- PROBLEM SPECIFICATION ---

Specification - A precise and detailed problem statement. Divided into five sections:

I. **Section 1/5 - Problem Statement:**
   A. Problem Description - A textual explanation of the problem to be solved:
      1. Provided a single or set of virtual punchcards in Extended Binary Coded Decimal (EBCD) format and a partial encoding cipher, process the cards as "cin" input and send the resultant character sequences to the "cout" output stream.
      2. The EBCD card format in this exercise utilizes an 80-wide by 12-high format, whereby each character is represented by a single column, with the first column position always representing a whitespace character, whether by being completely "punched" or completely blank.
      3. The solution should be coded in the C++ language and compiled using G++ in the Linux Operating System (OS).
   B. Assumptions being made:
      1. All inputs will be valid.
         a) There shall be no erroneous characters
         b) Punchcard data will be completely filled
      2. The final card in the series will have its first (left-most) column fully punched.
      3. The rows shall be labelled (in sequence): Y, X, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.
      4. The columns shall be referenced by index from 0 to 79.

5. The cipher shall correspond to the following punched values:
    a)

	-	1	2	3	4	5	6	7	8	9
-		1	2	3	4	5	6	7	8	9
Y	&	A	B	C	D	E	F	G	H	I
X	-	J	K	L	M	N	O	P	Q	R
0	0	/	S	T	U	V	W	X	Y	Z

6. While other characters exist in the expanded EBCD format, they are ignored for the purposes of this assignment.
7. Only three features will be present in the input stream:
    a) Blanks (a.k.a. Whitespace, ' ')
    b) Punches ('X')
    c) Line feeds/carriage return

C. Clarifications needed:
1. What character should represent a fully punched row, given that it does not appear on the EBCD sample cipher?
    a) A whitespace (' ')
2. How many blank lines will exist between sequential punchcards?
    a) A single line will be placed between punchcards.
3. Can linebreaks exist prior to the punchcards?
    a) Potentially.
4. What character or characters will the linebreak use?
    a) Indeterminate, depending upon the operating system environment.
    b) Experimentation or robustness will be required to solve for this.
5. In what orientation will the inputs be fed to "cin"?
    a) Cards are in landscape format, read from left to right, top to bottom.
6. Will the 'X's be capital or lowercase?
    a) Lowercase.
7. Should the entire contents of the last card be outputed?
    a) Yes.
8. Should the output of each card be on its own line?
    a) Yes. Place a linebreak between each card's output.

D. Assumptions that can be made:
1. "Zone punches" and "other punches" may be interpreted as a binary sequence.
    a) The encoding table may be organized as a binary tree.
2. Output shall be in ASCII encoding.
3. Every card shall begin with a whitespace character, followed by 79 encoded values.
4. If the first character of a new card is a punch, that is the last card of the sequence.
5. The first whitespace or 'X' encountered is the start of the first card.
    a) Counting the first whitespace encountered, the first 80 characters make up the first row.
    b) The second line may be identified by the first whitespace or 'X' encountered after the end of the first line.

        c) Sequential lines down to the 12th may be identified in a self-same fashion.
        d) The end of the 12th row is the end of the card.
        e) The first whitespace or ‘X’ found after the end of a card is the start of the next card.

E. Corner cases:
    1. No cards are provided by “cin”
    2. Endless cards provided by “cin”
    3. Cards provided by “cin” after the final marked card
    4. Blank cards
    5. Erroneous card data (void, see assumptions)

II. **Section 2/5 - Input Data**

A. Description - What is the data?
    1. A series of whitespace, ‘X’s, and line feeds which describe the contents of EBCD encoded punchcards.

B. Source - From where is the data received?
    1. Unknown initial source, but for this assignment, we may assume that data will be loaded to “cin”, the standard input stream.

C. Format - In what format will the data arrive?
    1. In a linear, streamed sequence of characters.

D. Invalid data
    1. For this assignment, we may assume that no invalid data can be provided.

III. **Section 3/5 - Output Data:**

A. Description - What is the data?
    1. A series of ASCII characters ranging from ‘0 - 9’, ‘A - Z’, ‘&’, ‘-’, ‘/’, and ‘ ‘.

B. Destination - To where is the data sent?
    1. Output data will be streamed to “cout”, the standard output stream.

C. Format - In what format will the data be sent?
    1. As a linear, streamed sequence of characters.

IV. **Section 4/5 - Error Handling:**

A. No data or connection has timed out
    1. Trigger: No content appears from “cin” after a set time.
    2. Message: “No data found from “cin”. Please check your connection and try again.”
    3. Handle: Program terminate.

B. Erroneous data (void, see assumptions)
    1. Trigger: Byte string cannot be interpreted as a valid character.
    2. Message: “Error: Input cannot be parsed at card <CardNumber>, column <ColumnNumber>”
    3. Handle: Append error to ErrorString, display output stream as normal, substituting ‘!’ for erroneous entry, display error messages after.

C. Endless data
    1. Trigger: Number of punchcards fed from cin exceeds a designed value
    2. Message: “Error: Input exceeds maximum number of cards. Halting.”

3. Handle: Output stream as normal up to top limit. Halt. Display error message. Program terminate.

V. **Section 5/5 - Test Plan** - A set of sequential tests of the correct operation of the program.
   A. Case 00: Set up IDE and compile a .cpp file using G++.
      1. Install Visual Studio, Bash, and G++.
      2. Set a simple, “Hello World” style behavior to a .cpp file.
      3. Attempt to compile and invoke “Hello World” at console.
      4. If “Hello World” can be displayed to console, this step is successful.
   B. Case 01: main() can redirect data from a test file, to cin, to a char variable.
      1. Generate a test punchcard holding a single value.
      2. Attempt to store this value to a variable.
      3. Print the contents of the variable to console.
      4. If the character matches that in the test file, the test file is connected to cin.
   C. Case 02: main() invokes the PunchCard.cpp class
      1. Set a simple, “Hello World” style behavior to PunchCard.cpp
      2. Adjust main() to call the PunchCard class and attempt to trigger its internal behavior.
      3. If “Hello World” is displayed to console, this step is successful.
   D. Case 03: PunchCard.cpp class can accept data from test file to cin, output to cout.
      1. Using code developed in Case 01, move from main() to PunchCard.cpp
      2. If “Hello World” is still displayed to console, this step is successful.
   E. Case 04: PunchCard.cpp can run a simple if-else test to modify the stream output.
      1. Set up if-else behaviors. If whitespace received, output “blank” to cout.
      2. If “blank” appears in console, this step is successful.
   F. Case 05: PunchCard.cpp can send someData to LetterTable.cpp and receive back someAsciiChar.
      1. LetterTable.cpp receives 1 or 0, sends back ‘L’ or ‘R’, respectively
      2. If ‘L’s and ‘R’s are printed to console, this step is successful.
   G. Case 06: LetterTable.cpp can generate binary tree to store table contents
      1. Key the value “Nought” to the data field of a node positioned to left, left, left of the root node.
      2. Send the sequence “000” to LetterTable.cpp
      3. If return result is “Nought”, the tree is working.
   H. Case 07: PunchCard.cpp can send data sequence to LetterTable.cpp
      1. Use PunchCard.cpp to send the sequence in TestFile.txt (“000”) to LetterTable.cpp
      2. If “Nought” is displayed to console, the pipeline is functional.
   I. Case 08: LetterTable.cpp is populated
      1. Generate the remaining nodes of the cipher table.
      2. Input the expanded sequences provided in the assignment description.
      3. If the console result is “ 0123456789 ABCDEFGHIJKLMNOPQRSTUVWXYZ &1/ “, the pipeline is functional.
   J. Case 09: PunchCard.cpp can halt after time limit if no input is provided.
      1. Develop a timer, attempt to halt and exit if no data is received from cin.
      2. If the program closes after the set time, this feature is functional.

K. Case 10: PunchCard.cpp can differentiate between cards.
   1. Alter the test file to contain 2 cards.
   2. Increment a counter on each 80-wide line which counts to 12 and prints out “Line Break” at that point.
   3. If “Line Break” appears twice in the output string to console, this feature is operational.
L. Case 11: PunchCard.cpp can halt after a set number of cards.
   1. Set a constant, private variable of 5.
   2. Alter the test file to contain a total of 6 cards.
   3. Increment a counter and halt the program when the counter exceeds the halting variable.
   4. If the program halts, this feature is operational.
M. Case 12: PunchCard.cpp can load chars to an array.
   1. Modify test card to have a single column reading “012345678901”
   2. Attempt to load an 80-wide array with values.
   3. If a query to the first index of this array returns “012345678901”, then this feature is operational.
N. Case 13: Program is functional.
   1. Modify test card to reflect possible corner cases and typical values
   2. Execute for anticipated behaviors.

--- **STEP 02** --- STRUCTURE CHARTS ---

I. See 'APPENDIX_00' - Displays the hierarchical algorithmic structure of a system.
  A. Designate 'Entire Problem'
    0. Name and description
    1. Box at top, center of diagram
  B. Designate sub-problems
    0. Name and description
    2. Draw below relevant problem box, forming a logical hierarchy
  C. Use annotated lines to indicate calls between functions
    0. Use arrowheads on lines to indicate direction of function calls
    1. Parameters - One-way information passed, but not modified.
      a. Draw with dot and arrow.
      b. Label with the name used by the calling procedure.
    2. Return values - Value sent back by the called function
      a. If return value is not used by calling function, then omit.
      b. Otherwise, draw with dot and arrow.
      c. And label with the name used by the called procedure.
    3. Modified parameters - Parameters passed via reference or to which pointers are passed.
      a. Draw with dot and two arrows.
      b. Arrows point to called and calling procedure.
  D. Continue designating subproblems until you arrive at problems which can be solved
    0. When all subproblem branches have solutions at all their leaves, the diagram is complete

--- STEP 03 --- CLASS DIAGRAMS ---

I. See 'APPENDIX_01' - Defines critical data for each defined class within the program

II. Comprised of a rectangle, subdivided into three sections:

A. The class name
  0. Located at the top
  1. CamelCased

B. Attributes / Fields
  0. Listed in the middle (in alphabetical order)
  1. Prefixed by visibility (Optional)
    a. '+' == public
    b. '#' == protected
    c. '-' == private
  2. The attribute name (Required)
    a. Lower-cased first word, CamelCased thereafter (e.g. "variableName")
    b. End with colon character (':')
  3. Data type (optional)
  4. Initial assignment value (optional)
    a. Preceded by 'gets' or assignment operator ("=")
  5. Example: "+ someVariable : boolean = true"
    a. Public boolean attribute named "someVariable", assigned the initial value of 'true'.

C. Operations
  0. Listed at the bottom (in alphabetical order)
  1. Prefixed by visibility (Optional)
    a. '+' == public
    b. '#' == protected
    c. '-' == private
  2. The attribute name (Required)
    a. CamelCased (e.g. "OperationName")
    b. End with colon character (':')
  3. The return value (Required if applicable)
    a. Lower-cased first word, CamelCased thereafter (e.g. "returnValue")
    b. End with colon character (':')
  4. Example: "+ SomeOperation(someArgument) : someReturn"
    a. Public operation named "SomeOperation", receiving "someArgument", generates "someReturn"

--- STEP 04 --- DESIGN STANDARDS ---

I. Variable scope (in order of preference)
A. const - Constant, the value of the variable cannot be changed. Should be performed as often as possible
B. auto - Private, allocated and deallocated at block entry and exit, respectively
C. static - Global, allocated at program execution, deallocated at program termination (?CONFIRM?)
D. extern - ??? (?CONFIRM?)

II. Functions
A. Should perform only one, simply defined operation
B. This operation may be the merging of two other functions

III. Parameters and Return Values
A. Descriptively named
B. Minimal in number per function

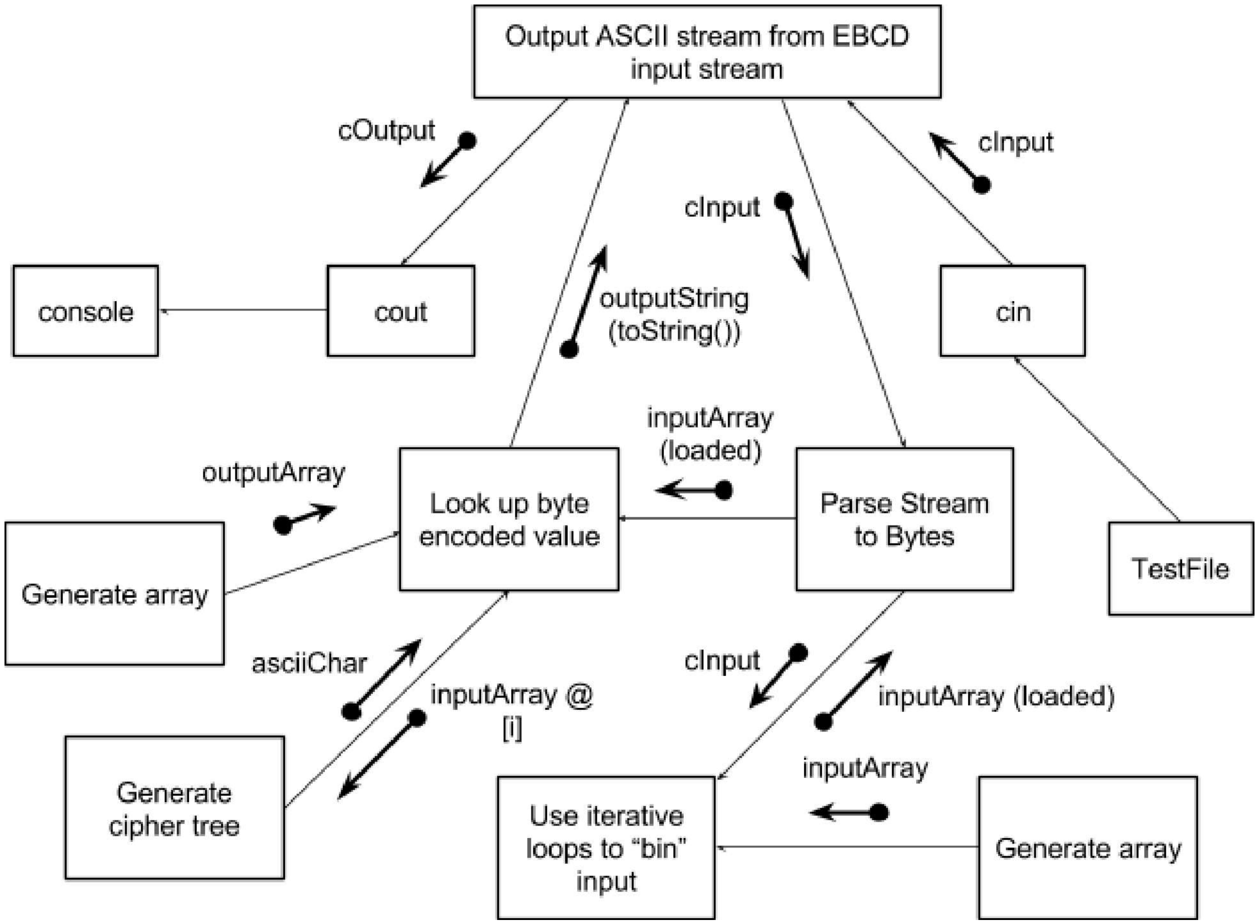
IV. Methods
A. Descriptively named
B. Private by default - Properly encapsulated
C. const - Constant, by default unless method is intended to modify the object's state.

V. Interfaces
A. The interface of an Abstract Data Type (ADT, also called a "class" or "object type") are the outward facing values.
B. Differentiated from internal implementation details.
C. Internal implementation and external interfaces should be kept separate.

VI. Input-Output (I/O) and User-Interfacing (UI)
A. ADTs should not include any user interface operations
0. UI functionality should be a separate problem from ADT design
B. You *may* use generic stream I/O operators in ADT design
0. friend operator<< - ??? (?CONFIRM?)
1. friend operator>> - ??? (?CONFIRM?)

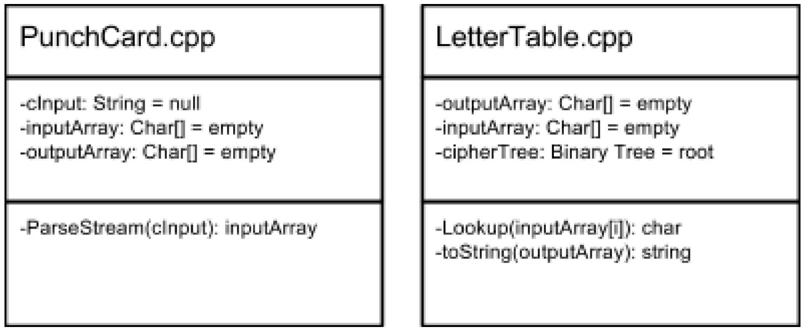
--- **APPENDIX_00** ---

Program UML Chart:



--- **APPENDIX_01** ---

Class Diagrams:



--- APPENDIX_02 ---

"CSS 501 Design and Coding Standards" text:

<Omitted pending author approval (Michael Stiber)>

// ---- END PRINTABLE CONTENT ----