```
// FILENAME: macroblockManager.h

#ifndef JPEG_DCT_MACROBLOCKMANAGER_H
#define JPEG_DCT_MACROBLOCKMANAGER_H
#define BDIM 8

#include "block.h"
#include "macroblock.h"
#include "rawInput.h"


/*
 * This is the Driver class, creates and controls structure
 */
class macroblockManager {
public:
    macroblockManager();
    ~macroblockManager();
    void PGMtoDCT();                // Read pgmFileParser string and dumpToDCT it to output file
    macroblock **macroblocks;       //  Two dimensional array of macroblocks (the only thing here that needs to be deleted
    size_t macroBlocksX;            // Number of macro blocks in X dim
    size_t macroBlocksY;            // Number of macro blocks in Y dim
    size_t x;                       // Total X dim of the input
    size_t y;                       // Total Y dim of the input

    void transform();               // Perform DCT transformation
    double qscale;                  // qscale holder
    char *outDCT;                   // location of DCT encoded file (in pgm -> dct transformation)
    char * inDct;                   // location of DCT in file ( in dct -> pgm transformation)
    char * outPGM;                  // location of PGM out filr (in dct -> pgm transformation)
    unsigned char * pgmFormattedOutput; // pgm encoded string ( used to dump pgm in dct -> pgm transofrmation)
    char * quantFile;               // location of quantfile
    rawInput inputObject;           // DCT or PGM input object holder - depends on how it is invoked
                                    // it is ok to do since DCT and PGM header are close enough.


    int quantMatrix[BDIM][BDIM];    // quantmatrix - aquired by parsing quantfile

    void setScale(char *string);    // Setter function for qscale value
    void setOutFile(char *string);  // Setter function for outDCT
    void parseQuantMatrix(char *string); // Read quantfile and same it into quantMatrix

    void initPGM(char *inputfile, char *quantfile, char *outputfile, char *qscale); // Initialize all data required for
        PGM->DCT transformation

    void WriteDCTheaderTo(FILE *pFILE); // Dump DCT header into pFile
    void WritePGMheaderTo(FILE *pFILE); // Dump PGM header into pFile

    void gatherPGMResults();        // Collect PGM

    void initDct(char *inputImage, char *quantfile, char *outputfile); // Initialize data needed for DCT->PGM transformation

    void DCTtoPGM(); // Convert DCT to PGM

    void fillMacroblocksFromDCT();

    void initMacroBlocks(rawInput *test); //

    void createMacroBlock(unsigned char *dctString, size_t start, size_t anEnd); // Create macroblock from PGM

    void readLine(unsigned char **src, unsigned char **dst); // Just read line, will advance src to the end of the line

    void parseOffset(unsigned char *line, size_t *offset_x, size_t *offset_y); // Read offset from beginning of a block in
        DCT formatted string

    void inverseTransofrm();
};

#endif //JPEG_DCT_MACROBLOCKMANAGER_H
```

```
// ******* FILENAME macroblockManager.cpp ************************

#include <stddef.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <iostream>
#include "macroblockManager.h"


/*
 * MacroblockManager implementation
 */

/*
 * Default constructor
 */
macroblockManager::macroblockManager() {
    macroblocks = NULL;
    macroBlocksX = 0;
    macroBlocksY = 0;
    pgmFormattedOutput = NULL;
}


/*
 * Default destructor
 */
macroblockManager::~macroblockManager() {
    if (macroblocks != NULL) {
        for (size_t i=0; i< macroBlocksX;i++) {
            delete macroblocks[i];
        }
        delete macroblocks;
    }
    if (pgmFormattedOutput != NULL) { delete pgmFormattedOutput;}
}

/*
 * Read pgm and dumpToDCT DCT
 */
void macroblockManager::PGMtoDCT() {
    macroBlocksX = inputObject.macroblocksX;
    macroBlocksY = inputObject.macroblocksY;
    x = inputObject.xDim;
    y = inputObject.yDim;
    initMacroBlocks(&inputObject);
    for (size_t i =0; i < inputObject.macroblocksY; i++) {
        for (size_t j =0; j < inputObject.macroblocksX; j++) {
            macroblocks[j][i].parse(&inputObject, j,i, x); // Let each macroblock to parsePGM it's own part
        }

    }
    transform();
    return;
}

/*
 * Allocate macroblocks
 */

void macroblockManager::initMacroBlocks(rawInput *test) {
    this->macroblocks = new macroblock * [test->macroblocksX];
    for (size_t i = 0;i< test->macroblocksX;i++) {
        macroblocks[i] = new macroblock [test->macroblocksY];
    }
}

/*
 * Transform and dumpToDCT
 */
void macroblockManager::transform() {
    FILE * out = fopen(outDCT, "w"); // Open out file with write permissions (file will be overwritten)
    WriteDCTheaderTo(out);
```

```
143     if (outDCT == NULL) {
144         printf("Failed to open %s\n", outDCT);
145         exit(1);
146     }
147     for (size_t i = 0;i < macroBlocksY;i++) {
148         for (size_t j = 0;j < macroBlocksX;j++) {
149             macroblocks[j][i].transform(quantMatrix, qscale); // Make each macrobock to transform itself
150             macroblocks[j][i].dump(out); // Make each macrobock to dumpToDCT itself
151         }
152     }
153     fclose(out);
154 
155     return;
156 }
157 
158 /*
159  * Setters
160  */
161 void macroblockManager::setScale(char *string) { qscale =atof(string); }
162 void macroblockManager::setOutFile(char *string) { outDCT = string; }
163 
164 /*
165  * Parse quantfile
166  */
167 void macroblockManager::parseQuantMatrix(char *string) {
168     FILE * p = fopen(string, "r");
169     if (p == NULL) {
170         printf("Failed to open %s\n", string);
171         exit(1);
172     }
173 
174     size_t lineSize = 1000;
175     size_t charSize = 100;
176     char line [lineSize]; // Templine
177     char qs [charSize]; // Temp char (100 symbols is overkill but whatever)
178     memset(qs,0,charSize);
179 
180     int count = 0;
181     int row = 0;
182     int col = 0;
183 
184     // loop through each line in quantfile, skip spaces and save values
185     while(fgets(line, lineSize, p) != NULL) { //read
186         size_t i = 0;
187         for (i = 0; i< lineSize; i++ ) {
188             if (line[i] == 10) { // End of line - break;
189                 if (strlen(qs) != 0) {
190                     quantMatrix[row][col] = atoi(qs); // set entry in the matrix
191                 }
192                 memset(qs,0,charSize);
193                 count=0;
194                 col=0;
195                 break;
196             } else if (line[i] == 32) { // Search for space
197                 if (strlen(qs) == 0) {continue;}
198                 else {
199                     quantMatrix[row][col] = atoi(qs); // set entry in the matrix
200                     memset(qs,0,100);
201                     count=0;
202                     col++;
203                 }
204             } else {
205                 qs[count] = line[i];
206                 count++;
207             }
208         }
209         row++;
210     }
211     if (row >8 || col > 8) {
212         std::cout<<"Error: quantfile expected to have 8 columns and 8 rows\n";
213         exit(1);
214     }
215     fclose(p);
216 }
```

```
217
218 /*
219  * Initialize object
220  */
221 void macroblockManager::initPGM(char *inputfile, char *quantfile, char *outputfile, char *qscale) {
222     setScale(qscale);
223     parseQuantMatrix(quantfile);
224     setOutFile(outputfile);
225     inputObject.readInput(inputfile);
226 }
227
228 /*
229  * Dump dct header
230  */
231 void macroblockManager::WriteDCTheaderTo(FILE *pFILE) {
232     fprintf(pFILE, "%s\n", "MYDCT");
233     fprintf(pFILE, "%lu %lu\n", x, y);
234     fprintf(pFILE, "%f\n", qscale);
235 }
236
237 /*
238  * Dump pgm header
239  */
240 void macroblockManager::WritePGMheaderTo(FILE *pFILE) {
241     fprintf(pFILE, "%s\n", "P5");
242     fprintf(pFILE, "%lu %lu\n", x, y);
243     fprintf(pFILE, "%d\n", 255);
244 }
245
246 /*
247  * Save arguments withinthis object
248  */
249 void macroblockManager::initDct(char *inputImage, char *quantfile, char *outputfile) {
250     inDct = inputImage;
251     this->quantFile = quantfile;
252     this->outPGM = outputfile;
253 }
254
255
256 /*
257  * Convert DCT fromatted file back to PGM
258  */
259 void macroblockManager::DCTtoPGM() {
260     parseQuantMatrix(quantFile); // Parse quantfile
261     inputObject.readInput(inDct); // Read DCT file
262     macroBlocksX = inputObject.macroblocksX; // Save number of macroblocks
263     macroBlocksY = inputObject.macroblocksY;
264     x = inputObject.xDim; // Save total dimensions of the picture
265     y = inputObject.yDim;
266     qscale = atof(inputObject.formatString); // fetch qscale
267     fillMacroblocksFromDCT(); // Parse dct formatted string into macroblocks
268     inverseTransofrm(); // inverse transform each macroblock
269     return;
270 }
271
272 /*
273  * Fill each macroblock from dct formatted string (block by block)
274  *
275  * Find next block start position, and end position, and fill corresponding macroblock
276  */
277 void macroblockManager::fillMacroblocksFromDCT() {
278     unsigned char * dctString = inputObject.rawString;
279
280     size_t mBlock_start = 0;
281     size_t mBlock_end = 0;
282     initMacroBlocks(&inputObject);
283     pgmFormattedOutput = new unsigned char[macroBlocksX * macroBlocksY * 16 * 16 + 1];
284     memset(pgmFormattedOutput,0,macroBlocksX * macroBlocksY * 16 * 16 + 1);
285     size_t count = 0;
286
287     for (size_t pos = 0; pos < inputObject.rawStringSize; pos++) {
288         if (dctString[pos] == 0) { break;} // EOF
289         else if (dctString[pos] == 10) { // Breakline (can be inside a block, need to check
290             if (count == 8) { // Block end
```

```
291
292                 mBlock_end = pos; // remember data
293                 createMacroBlock(dctString, mBlock_start, mBlock_end);
294                 mBlock_start = pos + 1; // skip new line
295                 count = 0;
296             } else {
297                 count++;
298             }
299         }
300
301     }
302 }
303
304
305 /*
306  * Create (if needed) and add data to a macroblock
307  */
308 void macroblockManager::createMacroBlock(unsigned char *dctString, size_t start, size_t anEnd) {
309     unsigned char * cblock = dctString+start;
310     unsigned char * line = new unsigned char[500];
311     memset(line, 0, 500);
312     size_t offset_x = 0;
313     size_t offset_y = 0;
314     // Each block in dct file starts with block position - fetch that line
315     readLine(&cblock, &line);
316
317     // Get block position
318     parseOffset(line, &offset_x, &offset_y);
319
320     // Get macroblock positions (index)
321     size_t macroblock_offset_x = offset_x/16;
322     size_t macroblock_offset_y = offset_y/16;
323
324     // Add cblock to that macroblock
325     macroblocks[macroblock_offset_x][macroblock_offset_y].fill_blockFromDCT(cblock, offset_x, offset_y);
326     delete(line);
327 }
328
329 // Just read one line from a string
330 void macroblockManager::readLine(unsigned char **src, unsigned char **dst) {
331
332     int index = 0;
333     unsigned char t;
334     while((*src)[index] != '\n'){ //loop until new line
335         t = (*src)[index];
336         (*dst)[index] =t;
337         index ++;
338     }
339     *src = *src + index + 1; // advance string passed new line
340 }
341
342
343 // Fetch block offsets
344 void macroblockManager::parseOffset(unsigned char *line, size_t *offset_x, size_t *offset_y) {
345     size_t i = 0;
346     size_t j = 0;
347     char temp [100];
348     memset(temp,0,100);
349     while(line[i]==32) { // skip spaces
350         i++;
351     }
352     while(line[i]!=32) { // read offset x
353         temp[j] = line[i];
354         i++;
355         j++;
356     }
357     (*offset_x) = atoi(temp); // save offset x
358     j=0;
359     memset(temp,0,100); // reset temp storage
360     while(line[i]==32) { // sckip spaces
361         i++;
362     }
363     while(line[i]!=0) { // read offset y
364         temp[j] = line[i];
```

```
        i++;
        j++;
    }
    (*offset_y) = atoi(temp); // save offset y

}


/*
 * Transform each macroblock after it was filled
 */
void macroblockManager::inverseTransofrm() {
    // Loop through each macroblock
    for (size_t i = 0;i < macroBlocksY;i++) {
        for (size_t j = 0;j < macroBlocksX;j++) {
            macroblocks[j][i].inverse_transform(quantMatrix, qscale); // Make each macrobock to transform itself
        }
    }

    // Gather pgm data after each macraoblock was inverted
    gatherPGMResults();
    return;

}

/*
 * Save each macroblock into pgm encoded file
 */
void macroblockManager::gatherPGMResults() {

    // loop through each macroblock and gather data
    for (size_t i = 0;i < macroBlocksY;i++) {
        for (size_t j = 0;j < macroBlocksX;j++) {
            macroblocks[j][i].gatherPGMtoString(pgmFormattedOutput, x);
        }
    }

    // open file for binary write
    FILE * out = fopen(outPGM, "wb"); // Open out file with write permissions (file will be overwritten)
    // dump header
    WritePGMheaderTo(out);

    // dump pgmFormatted output
    fwrite(pgmFormattedOutput, macroBlocksX * macroBlocksY * 16 * 16,1,out);
    fclose(out);

    return;
}

```

```
// ******** FILENAME macroblock.h ******************

#ifndef JPEG_DCT_MACROBLOCK_H
#define JPEG_DCT_MACROBLOCK_H
#define BLOCKS_DIM 2
#define BLOCK_SIZE 8

#include <iostream>
#include "block.h"

/*
 * Representation of a single macroblock
 */
class macroblock {
public:
    block blocks [BLOCKS_DIM][BLOCKS_DIM]; // Each macroblock has fixed number of blocks
    void transform(int qmatrix [BLOCK_SIZE][BLOCK_SIZE], double); // transoform will apply rawInput, quantmatrix, and zigzag
    void dump(FILE *outfile); // dumpToDCT content to outDCT
    void parse(rawInput *pEncoded, size_t i, size_t i1, size_t max_x); // Parse corresponding pgm encoded string

    size_t offset_x; // Macroblock offset in X dim
    size_t offset_y; // Macroblock offset in Y dim

    void fill_blockFromDCT(unsigned char *block, size_t b_offset_x, size_t b_offset_y); // Fill on of the block from DCT
        encoded string

    void inverse_transform(int quantMarix[8][8], double qscale); // Inverse transform each block in this macroblock

    void gatherPGMtoString(unsigned char *pgmOutPutContainer, size_t totalX); // Gather pgm Encoded string from each block
};

#endif //JPEG_DCT_MACROBLOCK_H
```

```
// ******** FILENAME macroblock.cpp ******************

#include <string.h>
#include "macroblock.h"
#define MBLOCKSIZE 16


/*
 * Perform transofrmation on each block
 */
void macroblock::transform(int quantMatrix [BLOCK_SIZE][BLOCK_SIZE], double qscale) {
    for (int y = 0; y < BLOCKS_DIM; y++) {
        for (int x = 0; x < BLOCKS_DIM; x++) {
            blocks[x][y].dct();
            blocks[x][y].quantize(quantMatrix, qscale);
            blocks[x][y].zigzag(false);
        }
    }
    return;
}

/*
 * parsePGM pgm
 * mb_id_x - index x as index of double array of macroblocks in macroblocks manager
 * mb_id_y - index y as index of double array of macroblocks in macroblocks manager
 * max_x  - total x size of the pgm
 */
void macroblock::parse(rawInput *pEncoded, size_t mb_ind_x, size_t mb_ind_y, size_t max_x) {
    offset_x = mb_ind_x * MBLOCKSIZE; // Calculate real x offset
    offset_y = mb_ind_y * MBLOCKSIZE; // Calculate real y offset

    for (int y = 0; y < BLOCKS_DIM; y++) {
        for (int x = 0; x < BLOCKS_DIM; x++) {
            blocks[x][y].parsePGM(pEncoded, offset_x, offset_y, x, y, max_x); // Let each block parsePGM its part.
        }
    }
}

/*
 * Dump rawInput encoded input.
 */
void macroblock::dump(FILE *outfile) {
    for (int y = 0; y < BLOCKS_DIM; y++) {
        for (int x = 0; x < BLOCKS_DIM; x++) {
            blocks[x][y].dumpToDCT(outfile); // Let each block dumpToDCT its part.
        }
    }
}

/*
 * Fill block with b_offest_x, and b_offset_y
 */
void macroblock::fill_blockFromDCT(unsigned char *block, size_t b_offset_x, size_t b_offset_y) {
    /*
     * Need to figure out and save this macroblock offset
     * and corresponding block index, since block offset is passed as real offset
     */
    // 1. Find and save this macroblock offset
    size_t mb_ind_x = b_offset_x/16;
    size_t mb_ind_y = b_offset_y/16;
    size_t mb_offset_x =mb_ind_x * 16;
    size_t mb_offset_y =mb_ind_y * 16;
    this->offset_x = mb_offset_x;
    this->offset_y = mb_offset_y;
    // 2. Find corresponding block index and fill it
    size_t b_ind_x = (b_offset_x - mb_offset_x)/8;
    size_t b_ind_y = (b_offset_y - mb_offset_y)/8;
    blocks[b_ind_x][b_ind_y].fillFromDCT(block, b_offset_x, b_offset_y);
}


/*
 * Inverse transform each block in this macroblock
 */
```

```
520 void macroblock::inverse_transform(int quantMatrix[8][8], double qscale) {
521     // Just loop through each block and apply required steps - reversed zigzag, quantize, dct
522     bool inversed = true;
523     for (int y = 0; y < BLOCKS_DIM; y++) {
524         for (int x = 0; x < BLOCKS_DIM; x++) {
525             blocks[x][y].zigzag(inversed);
526             blocks[x][y].inverse_quantize(quantMatrix, qscale);
527             blocks[x][y].inverse_dct();
528         }
529     }
530     return;
531 
532 }
533 
534 
535 /*
536  * Gather pgm from each block in this macroblock
537  */
538 void macroblock::gatherPGMtoString(unsigned char *pgmOutPutContainer, size_t totalX) {
539     // loop through each block, and fetch pgm encoded data
540     for (int y = 0; y < BLOCKS_DIM; y++) {
541         for (int x = 0; x < BLOCKS_DIM; x++) {
542             blocks[x][y].gatherPGM(pgmOutPutContainer, totalX);
543         }
544     }
545     return;
546 }
547 
```

```
548 // ******** FILENAME block.h ******************
549
550 #ifndef JPEG_DCT_BLOCK_H
551 #define JPEG_DCT_BLOCK_H
552
553 #include <stddef.h>
554 #include <stdio.h>
555 #include "rawInput.h"
556
557
558 /*
559  * This class is representing 8x8 block
560  */
561 class block {
562 public:
563     block(); // Default constructor
564     void dct(); // Dct transform itself
565     void zigzag(bool inversed); // Zig zag transform itself
566     void setIndex(size_t x, size_t y); // Set offset
567     unsigned char items [8][8]; // Raw items in pgm
568     double transofrmed [8][8]; // DCT transformed items
569     int quantized [8][8];     // Quantized items
570     int reordered [8][8];     // Reordered items
571     //Could use only one array to save space -- easier this way, though less efficient.
572
573     size_t x; // Real offset in x
574     size_t y; // Real offset in y
575
576     void quantize(int qmatrix[8][8], double qscale); // Apply qmatrix and qscale to this block
577     void parsePGM(rawInput *pEncoded, size_t i, size_t i1, int i2, int i3, size_t total_x); // parse pgm formatted string
578     void dumpToDCT(FILE *outfile); // Dump this block to outfile in dct format
579
580     void fillFromDCT(unsigned char *block, size_t b_oofset_x, size_t b_offset_y); // parse dct formatted string
581
582     void inverse_quantize(int quantMatrix[8][8], double qscale); // Reverse quantization
583
584     void inverse_dct(); // Inverse dct
585
586     void gatherPGM(unsigned char *pgmContainer, size_t totalX);// Dump this block in pgm format into pgm formatted string
587 };
588
589 #endif //JPEG_DCT_BLOCK_H
590
```

```
591 // ******** FILENAME block.cpp ******************
592
593 #include "block.h"
594 #include <math.h>
595 #include <string.h>
596 #include <stdlib.h>
597
598 #define BDIM 8
599
600 /*
601  * Implementation of the block class
602  */
603
604 /*
605  * Default constructor - just set everything to 0
606  */
607 block::block() {
608     for (int i = 0; i < BDIM;i++) {
609         for (int j = 0;j< BDIM;j++) {
610             items[i][j]=0;
611             transofrmed[i][j]=0;
612             reordered[i][j]=0;
613         }
614     }
615     x = 0;
616     y = 0;
617 }
618
619 /*
620  * Offset setter
621  */
622 void block::setIndex(size_t x, size_t y) {
623     this->x = x;
624     this->y = y;
625 }
626
627
628 /*
629  * Dct transform (Just apply the given formula, without optimizations
630  */
631 void block::dct() {
632     double C_u = 0.0;
633     double C_v = 0.0;
634     double sqr = 1.0/sqrt(2.0);
635     double coef = 0.0;
636
637     for (int v = 0 ; v < BDIM; v++) {
638             if (v == 0) {
639                 C_v = sqr;
640             } else {
641                 C_v = 1.0;
642             }
643         for (int u = 0 ; u < BDIM; u++) {
644             if (u == 0) {
645                 C_u = sqr;
646             } else {
647                 C_u = 1.0;
648             }
649             coef = (C_u/2.0) * (C_v/2.0);
650             double sum = 0;
651             for (int y = 0; y < BDIM; y++) {
652                 for (int x = 0; x < BDIM; x++) {
653                     sum += (double)items[x][y]*cos(((double)(2* x +1)*(double) u *M_PI)/16.0)*cos(((double)(2* y +1)*(double)
                         v *M_PI)/16.0);
654                 }
655             }
656             transofrmed[u][v] = coef * sum;
657         }
658     }
659     return;
660 }
661
662 /*
663  * Zigzag reorder
```

```
 *  if inversed true - then inversed prcess applied
 */
void block::zigzag(bool inversed) {
    int m = 8;
    int y =0, x =0;
    int c =0, r =0; //row and column
    int res=0;
    int n = 0;
    for (int i = 0; i < m * 2; i++) {
        for (int j = (i < m) ? 0 : i - m + 1; j <= i && j < m; j++) {
            y = n / 8;
            x = n - y * 8;
            n++;
            res = (i & 1) ? j * (m - 1) + i : (i - j) * m + j;
            c = res / 8;
            r = res - c * 8;
            inversed ? quantized[r][c] = reordered[x][y] : reordered[x][y] = quantized[r][c];
        }
    }
}

/*
 * Quantize the transformed array
 */
void block::quantize(int qmatrix [8][8], double qscale) {
    // quantize each element of the DCT transformed array
    for (int x= 0;x<BDIM;x++) {
        for (int y= 0;y<BDIM;y++) {
            int val = (int)round( transofrmed[y][x] / ((double)qmatrix[x][y]*qscale)); // apply qmatrix and qscale and round
            if (val < -127) {
                val = -127;
            } else if (val > 128) {
                val = 128;
            }
            val +=127;
            quantized[y][x] = val; //save quantized value
        }
    }
}


/*
 * parsePGM pgm formatted string into double array, and further on transform to DCT
 */
void block::parsePGM(rawInput *pEncoded, size_t macroblock_offset_x, size_t macroblock_offset_y,
                     int block_offset_x, int block_offset_y, size_t total_x) {


    // Set current block real offset
    setIndex(macroblock_offset_x + block_offset_x*BDIM, macroblock_offset_y + block_offset_y*BDIM);

    size_t index=0;
    size_t loc_x = 0;
    size_t loc_y = 0;
    // Need to translate two dimensional indexes to flat index of pgm file
    for (size_t row = 0; row < BDIM; row++) {
        // y location
        loc_y = (y + row) * total_x;
        for (size_t column = 0; column < BDIM; column++) {
            // x location
            loc_x = x + column;
            // location in the PGM input
            index = loc_x + loc_y;
            items[column][row] = pEncoded->rawString[index];
        }
    }
}


/*
 * Save this block to provided DCT output file (used in pgm -> DCT)
 */
void block::dumpToDCT(FILE *outfile) {
    fprintf(outfile, "%lu %lu\n", x, y);
```

```
    for (int i = 0; i < 8; i++) {
        for (int j = 0; j < 8; j++) {
            fprintf(outfile, "%5d", reordered[j][i]);
        }
        fprintf(outfile, "\n");
    }
}

/*
 *
 * Use dct formatted string to fill current block
 */
void block::fillFromDCT(unsigned char *block, size_t b_oofset_x, size_t b_offset_y) {
    /*
     * Set current block offset
     * at this point block is current block, so just parse it
     */
    x = b_oofset_x;
    y = b_offset_y;

    size_t index = 0;
    size_t temp_index=0;
    unsigned char temp[100];
    memset(temp,0,100);
    for (int yl = 0; yl < BDIM; yl++) {
        for (int xl = 0; xl < BDIM; xl++) {
            while(block[index] != 10) {
                if (block[index] != 32) {
                    temp[temp_index] = block[index];
                    temp_index++;
                } else {
                    if (temp_index != 0) {
                        reordered[xl][yl] = atoi((char *)temp);
                        memset(temp,0,100);
                        temp_index = 0;
                        break;
                    }
                    temp_index = 0;
                }
                index++;
            }
            if (temp_index != 0) {
                reordered[xl][yl] = atoi((char *)temp);
                memset(temp,0,100);
                temp_index = 0;
            }
            index++;
        }
    }
}


/*
 * Apply inverse quantization
 */
void block::inverse_quantize(int (*quantMatrix)[8], double qscale) {
    for (int x= 0;x<BDIM;x++) {
        for (int y = 0; y < BDIM; y++) {
            double val = (double) (quantized[y][x] - 127);
            val = val * ((double) quantMatrix[x][y] * qscale);
            transofrmed[y][x] = val;
        }
    }

}


/*
 * Inverse DCT - by the book
 */
void block::inverse_dct() {
    double C_u = 0.0;
    double C_v = 0.0;
    double sqr = 1.0/sqrt(2.0);
```

```
812     double coef = 0.0;
813 
814     for (int y = 0 ; y < BDIM; y++) {
815         for (int x = 0 ; x < BDIM; x++) {
816             double sum = 0;
817             for (int u = 0; u< BDIM; u++) {
818                 if (u == 0) {
819                     C_u = sqr;
820                 } else {
821                     C_u = 1.0;
822                 }
823                 for (int v = 0; v< BDIM; v++) {
824                     if (v == 0) {
825                         C_v = sqr;
826                     } else {
827                         C_v = 1.0;
828                     }
829                     coef = C_u * C_v;
830                     sum += coef * transofrmed[v][u] *cos(((double)(2* y +1)*(double)u*M_PI)/16.0)*cos(((double)(2* x
                          +1)*(double)v*M_PI)/16.0);
831                 }
832 
833             }
834             sum = sum/4.0;
835             if (sum < 0) {
836                 sum = 0;
837             } else if (sum > 255) {
838                 sum = 255;
839             }
840             items[x][y] = (unsigned char)sum;
841         }
842     }
843     return;
844 }
845 
846 
847 /*
848  * Gather current block into pgm formatted string.
849  * String must be allocated and have correct size
850  * No bounds checking is preformed at this level
851  */
852 void block::gatherPGM(unsigned char *pgmContainer, size_t totalX) {
853     // Offset in the flat string
854     int realIndex = 0;
855     for (int yl = 0; yl <BDIM; yl++) {
856         for (int xl = 0; xl < BDIM; xl++) {
857             realIndex = xl + x + (y + yl) *totalX;
858             pgmContainer[realIndex] = items[xl][yl];
859         }
860     }
861 }
862 
```

```
863 // ******** FILENAME rawInput.h ******************
864
865 #ifndef DCT_DCTENCODED_H
866 #define DCT_DCTENCODED_H
867
868 #include <glob.h>
869
870 // This class represent rawInput encoded file.
871 class rawInput {
872 public:
873
874     rawInput();
875
876     ~rawInput();
877
878     void init(size_t rawSize);
879     void readInput(char *fname);
880
881     size_t xDim; // x dimension retrieved from header
882     size_t yDim; // y dimension retrieved from header
883     //
884     char header [20]; // Header
885     char formatString[20]; // qscale or 255 - depends on the file we are reading
886     unsigned char * rawString; // Encoded String - body of the file
887
888     size_t rawStringSize; // Encoded String size
889     size_t macroblocksX; // Number of macroblocks in y dimension
890     size_t macroblocksY; // Number of macroblocks in x dimension
891 };
892
893 #endif //DCT_DCTENCODED_H
894
```

```
// ******** FILENAME rawInput.cpp ******************

#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <iostream>
#include "rawInput.h"

/*
 * implementation of rawInput class
 */

/*
 * Default constructor
 * Set everything to 0
 */
rawInput::rawInput() {
    memset(this->formatString,0,20);
    memset(this->header,0,20);
    this->xDim = 0;
    this->yDim = 0;

    this->rawStringSize =0;

    this->macroblocksX = 0;
    this->macroblocksY = 0;
    this->rawString = NULL;
}

/*
 * Default destructor
 * rawString is the only thing needs to be cleaned up
 */
rawInput::~rawInput() {
    if (this->rawString != NULL) {
        free(this->rawString);
    }
}

/*
 * Allocate rawString
 */
void rawInput::init(size_t rawSize) {
    this->rawString = (unsigned char*) malloc(rawSize);
    memset(this->rawString, 0, rawSize);
}


/*
 * Read input file (can be either PGM or DCT)
 */
void rawInput::readInput(char *fname) {
    FILE * p = fopen(fname, "rb"); // Open file with reading permission
    if (p == NULL) {
        printf("Failed to open %s\n", fname);
        exit(1);
    }

    // Figure out total size of the file;
    fseek(p, 0L, SEEK_END);
    size_t totSize = ftell(p);
    fseek(p, 0L, SEEK_SET);

    init(totSize);

    char line [100]; // Temp string
    memset(line, 0, 100);
    fgets(this->header,20,p); // Read header
    fgets(line,100,p); // Read dimensions
    char dim[100];
    memset(dim, 0, 100);

    int dimInd = 0;
    int dimI = 0;
```

```
    while(line[dimInd] == 32) {
        dimInd++;
    }
    while(line[dimInd] != 32) {
        dim[dimI] = line[dimInd];
        dimI++;
        dimInd++;
    }
    dimI=0;
    this->xDim = atoi(dim);
    memset(dim, 0, 100);

    while(line[dimInd] == 32) {
        dimInd++;
    }
    while(line[dimInd] != 10) {
        dim[dimI] = line[dimInd];
        dimI++;
        dimInd++;
    }
    this->yDim = atoi(dim);

    if (xDim % 16 != 0 || yDim % 16 != 0) {
        std::cout<<"Error, input file dimensions expected to be divisible by 16\n";
        exit(1);
    }
    this->macroblocksX = this->xDim/16; // Calculate number of macroblocks
    this->macroblocksY = this->yDim/16;


    fgets(this->formatString,20,p); // Read 255 (if PGM), or qscale if (DCT)
    size_t sz = ftell(p);

    size_t encodedLineSize = totSize - sz;
    fread(this->rawString, encodedLineSize, 1, p); // Read encoded part in binary.
    this->rawStringSize = encodedLineSize;
    fclose(p); // Don't forget to close the input file.
}
```