ISEL | DEETC

Programação de Sistemas Computacionais

3ª Série de Exercícios

Semestre de Inverno de 2009/2010

Autores:
30896 – Ricardo Canto
31401 – Nuno Cancelo
33595 – Nuno Sousa

Indicie

Enunciado

Construa os programas indicados usando a linguagem C. Entregue o código desenvolvido, devidamente indentado e comentado, e um relatório com a descrição das soluções. Inclua na entrega o makefile que permita gerar os ficheiros executáveis a partir do código fonte. O relatório deverá ser um guia para a compreensão do código desenvolvido e não uma mera tradução deste para língua natural.

Contacte o docente se tiver dúvidas.
Encoraja-se a discussão de problemas e soluções com colegas de outros grupos, mas recorda-se que a partilha directa de soluções leva, no mínimo, à anulação das entregas de todos os envolvidos.

Os formatos de imagem PBM, PGM e PPM [http://ozviz.wasp.uwa.edu.au/~pbourke/dataformats/ppm/] permitem o armazenamento de imagens a preto e branco, escala de cinzentos e a cores, respectivamente. Em anexo é fornecido um conjunto de imagens para usar nos exercícios propostos. A este conjunto pode juntar outras imagens nestes formatos.

1. Escreva um programa (imgInfo) que apresente no ecrã a dimensão (largura x altura), o tipo de representação (P1, P2,…,P6) e o valor máximo admissível para cada campo de informação sobre os pixels da imagem armazenada num ficheiro PBM, PGM ou PPM. Indique também qual o ponto no ficheiro em que começa a informação sobre os pixels da imagem. Apresente no relatório o resultado deste programa em todas as imagens do conjunto.

2. A biblioteca zlib [http://www.zlib.net/] fornece funções que realizam a compactação (deflate) e descompactação (inflate) de dados baseados numa variante do algoritmo LZ77. Utilizando a informação disponível no documento online "zlib Usage Example", gere o compactador/descompactador (zpipe) e compacte as imagens PPM do conjunto, apresentando no relatório as dimensões (em bytes) dos ficheiros originais e as do resultado da compactação.

O formato de imagem PNG [http://www.libpng.org/pub/png/] utiliza uma técnica de compressão, sem perdas (compactação), composta por dois passos: primeiro aplica um filtro de transformação sobre os bytes que descrevem os pixels da imagem original; o resultado da transformação é, em seguida, assado à zlib para compactação. Na especificação PNG são indicados 5 filtros diferentes (None, Sub, Up, Average e Paeth). No filtro Sub, por exemplo, cada byte de um pixel é transformado na diferença para o byte correspondente do pixel à sua esquerda.

3. Inspirando-se na técnica de compactação do formato PNG, construa um compactador (ppmZip) de imagens PPM P6 apenas com campos até 8 bits. Os ficheiros gerados devem ter extensão PPZ. Compacte as imagens do conjunto, apresentando no relatório as dimensões (em bytes) dos ficheiros originais e as do resultado da compactação. Apresente os resultados em conjunto com os obtidos no exercício 2.

4. Construa um descompactador (ppmUnzip) para reconstituir as imagens PPM P6 originais a partir dos ficheiros produzidos no exercício anterior. Reconstitua as imagens do conjunto a partir dos ficheiros PPZ gerados pelo compactador. Verifique que os ficheiros reconstituídos e os originais são iguais.

Exercício 1

Na análise deste exercício, concluímos que seria aconselhável que o nosso código fosse realizado por módulos, realizando uma solução para cada problema que fosse encontrado.

Tendo este principio em mente, procedemos à implementação dos seguintes ficheiros fonte:

ficheiro: header.c

```
#include <stdio.h>

#include "imageFormat.h"

#include "myLib.h"

/*

 * Processa múltiplos ficheiros de entrada. Necessita de pelo menos 1.

 * */

int main(int argc, char *argv[]) {

    header fhp;

    int nbrChars=0;

    if (argc == 1) {

        puts("Need at least one argument to be processed!");

        return UNSUCCESS;

    }

    while ((argc-1) > 0) {

        argc--;

                nbrChars=processFileHeader(argv[argc],&fhp);

                listfileHeader(argv[argc],&fhp,nbrChars);

    }


return SUCCESS;

}
```

Como se pode verificar, este código fonte somente têm o método main que se comporta como um programa de teste, sendo independente da implementação do código que faz o processamento do ficheiro de imagens.
Verifica-se que são incluídas algumas bibliotecas que implementámos, com o intuito de manter as definições de funções e estruturas independentes das funcionalidades implementadas (caso do mylib.h) e de uma biblioteca com as definições das funções que realmente irão processar o ficheiro para retornar o pretendido pelo enunciado.

ficheiro: mylib.h

```
#ifndef MYLIB_H
#define MYLIB_H

#define SUCCESS 0;
#define UNSUCCESS 1;
enum boolean {false,true};
typedef unsigned char byte;

#endif
```

ficheiro: imageFormat.h

```
#ifndef IMAGEFORMAT_H
#define IMAGEFORMAT_H

typedef struct FileHeader{
        int magicValue;
        int imageWidth;
        int imageHeight;
        int maxGrey;
        int isFilled;
} header;

void initFileHeader(struct FileHeader *fh);

void listfileHeader(char* filename,struct FileHeader *fh,int nbrChars);

void fillHeader(struct FileHeader *fh, int value);

int processFileHeader(char *filename, struct FileHeader *fhp);

int processFileHeader(char *filename, struct FileHeader *fhp);

#endif
```

ficheiro: imageFormat.c

```
#include <stdio.h>

#include "imageFormat.h"

#include "mylib.h"


/*
 * void initFileHeader(struct FileHeader *fh);
 * Função que vai inicializar os valores do objecto FileHeader
 */
void initFileHeader(struct FileHeader *fh){
        fh->magicValue=0;
        fh->imageWidth=0;
        fh->imageHeight=0;
        fh->maxGrey=0;
        fh->isFilled=false;
}
/*
 * void listfileHeader(char* filename,struct FileHeader *fh,int nbrChars)
 * Lista os conteudos do FileHeader.
 */
void listfileHeader(char* filename,struct FileHeader *fh,int nbrChars){
        puts("----------------------------------------");
        printf("File: %s \nHeader File Information\n",filename);
        puts("----------------------------------------");
        printf("File magicValue: P%i\n",fh->magicValue);
        printf("Image Width: %i\n",fh->imageWidth);
        printf("Image Height: %i\n",fh->imageHeight);
        if (fh->magicValue > 1){
                printf("Max Gray: %i\n",fh->maxGrey);
        }
        printf("Nbr of chars read: %i\n",nbrChars);
        puts("----------------------------------------");
}
/*
 * void fillHeader(struct FileHeader *fh, int value)
 * Função que preenche o header.
 * Dadas as caracteristicas do ficheiro, os dados vão ser preenchidos
 * pela ordem de chegada.
 * Existe uma atenção especial ao MagicValue para o preenchimento do valor
 * do MaxGray.
```

```
 */
void fillHeader(struct FileHeader *fh, int value){
        if (fh->magicValue == 0){
                fh->magicValue = value;
        }else if(fh->imageWidth == 0){
                fh->imageWidth = value;
        }else if(fh->imageHeight == 0){
                fh->imageHeight = value;
                fh->isFilled = (fh->magicValue == 1)?1:0;
        }else if(fh->maxGrey == 0){
                fh->maxGrey = value;
                fh->isFilled = (fh->magicValue > 1)?1:0;
        }
}



/*
 * int processFileHeader(char *filename, struct FileHeader *fhp)
 * Processa a informação constante no bloco de dados referente à descri-
 * ção do ficheiro preenchendo a estrutura do Header do Ficheiro.
 * Retorna o numero de caracteres lidos, até chegar ao bloco do Bitmap.
 */
int processFileHeader(char *filename, struct FileHeader *fhp) {
        int chars=0;
        int ignore=false;
        int parsedInt=0;
        int isHeaderBlockFinished=false;
        int c=0;
        FILE *fp=fopen(filename,"r");
        initFileHeader(fhp);
        while((isHeaderBlockFinished==false) && (c=fgetc(fp))!= EOF  ){
                switch(c){
                        case '#':
                        case '\t':
                        case ' ' :
                        case '\r':
                        case '\n':
                                if ((ignore == false) && parsedInt > 0){fillHeader(fhp,parsedInt);}
                                ignore = (c == '\n' || c == '\t')?false:(c == '#')?true:ignore;
                                parsedInt=0;
                                break;
                        default:
                                if ((ignore == false)){
                                        if (fhp->isFilled == true) {
                                                isHeaderBlockFinished=true;
                                                chars--;
                                                break;
                                        }
                                        if( c > 47 && c < 58){parsedInt = parsedInt*10+(c-48);}
                                }
                        break;
                }
                chars++;
        }
        fclose(fp);
```

```
 return chars;

}
```

Exercício 2

Para este exercício a principal dificuldade foi a análise dos argumentos passados na linha de comandos uma vez que, após definirmos as operações que se pretendem e após estudo da biblioteca zlib, basta chamar os as funções da biblioteca com os argumentos correctos para que seja efectuada a compressão/descompressão do ficheiro.

Tendo em consideração a necessidade de utilização destas funções nos exercícios seguintes criámos o seguinte código:

ficheiro: z-Library.h

```
#ifndef ZLIBRARY_H
#define ZLIBRARY_H
/* *
 * Definição de Bibliotecas Necessárias
 * */
#ifndef STDIO_DEF
#define STDIO_DEF
#include <stdio.h>
#endif

#include <string.h>
#include <assert.h>
#include "mylib.h"
#include "zlib.h"


/* *
 * Definição de um Work-Around para os ambientes Windows
 * */

#if defined(MSDOS) || defined(OS2) || defined(WIN32) || defined(__CYGWIN__)
#  include <fcntl.h>
#  include <io.h>
#  define SET_BINARY_MODE(file) setmode(fileno(file), O_BINARY)
#else
#  define SET_BINARY_MODE(file)
#endif

/* *
 * Definição do tamanho do CHUNK
 * */

#define KBYTES 1024
#define CHUNK 128*KBYTES

int my_compress(FILE *source, FILE *dest, int level);

int my_decompress(FILE *source, FILE *dest);

void my_errors(int ret);

void progUsage(char *program);

typedef unsigned char boolean;

enum action {compress_action,decompress_action};

typedef struct cmdLnArgs{
                  int action;
                  char * source;
                  char * destination;
                  int compressLevel;
                  int filter;
                  boolean destParsed;
}myArgs;
#endif
```

ficheiro: z-Library.c

```
#ifndef STDIO_DEF
#define STDIO_DEF
#include <stdio.h>
#endif
#include "z-Library.h"
#include <stdlib.h>
#define DEFAULT_EXT_NAME ".z"
enum CMD_ERRORS{CMD_INVALID_EXT,CMD_INVALID_OPTION,CMD_INVALID_FILES};

int my_compress(FILE *source, FILE *dest, int level)
{
    int ret, flush;
    unsigned have;
```

```
    z_stream strm;
    unsigned char in[CHUNK];
    unsigned char out[CHUNK];

    /* allocate deflate state */
    strm.zalloc = Z_NULL;
    strm.zfree = Z_NULL;
    strm.opaque = Z_NULL;
    ret = deflateInit(&strm, level);
    if (ret != Z_OK)
        return ret;

    /* compress until end of file */
    do {
        strm.avail_in = fread(in, 1, CHUNK, source);
        if (ferror(source)) {
            (void)deflateEnd(&strm);
            return Z_ERRNO;
        }
        flush = feof(source) ? Z_FINISH : Z_NO_FLUSH;
        strm.next_in = in;

        /* run deflate() on input until output buffer not full, finish
           compression if all of source has been read in */
        do {
            strm.avail_out = CHUNK;
            strm.next_out = out;
            ret = deflate(&strm, flush);    /* no bad return value */
            assert(ret != Z_STREAM_ERROR);  /* state not clobbered */
            have = CHUNK - strm.avail_out;
            if (fwrite(out, 1, have, dest) != have || ferror(dest)) {
                (void)deflateEnd(&strm);
                return Z_ERRNO;
            }
        } while (strm.avail_out == 0);
        assert(strm.avail_in == 0);     /* all input will be used */

        /* done when last data in file processed */
    } while (flush != Z_FINISH);
    assert(ret == Z_STREAM_END);        /* stream will be complete */

    /* clean up and return */
    (void)deflateEnd(&strm);
    return Z_OK;
}


int my_decompress(FILE *source, FILE *dest)
{
    int ret;
    unsigned have;
    z_stream strm;
    unsigned char in[CHUNK];
    unsigned char out[CHUNK];

    /* allocate inflate state */
    strm.zalloc = Z_NULL;
    strm.zfree = Z_NULL;
    strm.opaque = Z_NULL;
    strm.avail_in = 0;
    strm.next_in = Z_NULL;
    ret = inflateInit(&strm);
    if (ret != Z_OK)
        return ret;

    /* decompress until deflate stream ends or end of file */
    do {
        strm.avail_in = fread(in, 1, CHUNK, source);
        if (ferror(source)) {
            (void)inflateEnd(&strm);
            return Z_ERRNO;
        }
        if (strm.avail_in == 0)
            break;
        strm.next_in = in;

        /* run inflate() on input until output buffer not full */
        do {
            strm.avail_out = CHUNK;
            strm.next_out = out;
            ret = inflate(&strm, Z_NO_FLUSH);
            assert(ret != Z_STREAM_ERROR);  /* state not clobbered */
            switch (ret) {
            case Z_NEED_DICT:
                ret = Z_DATA_ERROR;     /* and fall through */
            case Z_DATA_ERROR:
            case Z_MEM_ERROR:
                (void)inflateEnd(&strm);
                return ret;
            }
            have = CHUNK - strm.avail_out;
            if (fwrite(out, 1, have, dest) != have || ferror(dest)) {
                (void)inflateEnd(&strm);
                return Z_ERRNO;
            }
        } while (strm.avail_out == 0);

        /* done when inflate() says it's done */
    } while (ret != Z_STREAM_END);

    /* clean up and return */
    (void)inflateEnd(&strm);
```

```
    return ret == Z_STREAM_END ? Z_OK : Z_DATA_ERROR;
}
/* report a zlib or i/o error */
void my_errors(int ret)
{
    fputs("zpipe: ", stderr);
    switch (ret) {
    case Z_ERRNO:
        if (ferror(stdin))
            fputs("error reading stdin\n", stderr);
        if (ferror(stdout))
            fputs("error writing stdout\n", stderr);
        break;
    case Z_STREAM_ERROR:
        fputs("invalid compression level\n", stderr);
        break;
    case Z_DATA_ERROR:
        fputs("invalid or incomplete deflate data\n", stderr);
        break;
    case Z_MEM_ERROR:
        fputs("out of memory\n", stderr);
        break;
    case Z_VERSION_ERROR:
        fputs("zlib version mismatch!\n", stderr);
    }
}

void my_cmdErrors(int ret){
         fputs("Comand Line Errors: ", stderr);
         switch(ret){
                    case CMD_INVALID_EXT:
                              fputs("Not Know Extention\n", stderr);
                    case CMD_INVALID_FILES:
                              fputs("Invalid source/destination files\n", stderr);
                              break;
                    case CMD_INVALID_OPTION:
                              fputs("Invalid Option\n", stderr);
                              break;
         }
}
void progUsage(char *program){
         puts("-------------------------------------------------------------------------------");
         printf("Usage: \n%s -[c,d] -[l] [0-9] -[f] [0-5] source [dest]\n",program);
         puts("-------------------------------------------------------------------------------");
         puts("-c : compress");
         puts("-d : decompress");
         puts("-l : compress level");
         puts("\t[0-9] : 0- arquive, 9- maximum compress.");
         puts("-f : Filter type");
         puts("\t[0-9] : 0- None, 1- Sub, 2- Up, 3- Average, 4- Paeth, 5-Best.");
         puts("source: source file");
         puts("destination: destination file. It's optional. Case ignored destination file:");
         puts("\twhile compress: end with the extention source.z");
         puts("\twhile decompress: end with the extention source.dz");
         puts("-------------------------------------------------------------------------------");
}

void init_Args(struct cmdLnArgs *ma){
         /*Setting defaults values*/
         ma->action = compress_action;
         ma->compressLevel = Z_DEFAULT_COMPRESSION;
         ma->filter = 0;
         ma->destParsed = false;
}

int parseArgs(struct cmdLnArgs *ma, int argc, char **argv){
         int idx=1;
    while (idx < argc){

                  if ((char)argv[idx][0] == '-'){
                            switch ((char)argv[idx][1]){
                                      case 'c':
                                                ma->action=compress_action;
                                                break;
                                      case 'd':
                                                ma->action=decompress_action;
                                                break;
                                      case 'l':
                                                idx++;
                                                ma->compressLevel=*argv[idx] - '0';
                                                break;
                                      case 'f':
                                                idx++;
                                                ma->filter=*argv[idx] - '0';
                                                break;
                                      default:
                                                printf("%s \n",argv[idx]);
                                                my_cmdErrors(CMD_INVALID_OPTION);
                                                progUsage(argv[0]);
                                                return false;
                                                break;
                            }
                  }else{
                            switch (((int)(argc -  idx))){
                                      case 1:
                                                ma->source=argv[idx];
                                                ma->destination=0;
                                                return true;
                                                break;
                                      case 2:
                                                ma->source=argv[idx++];
                                                ma->destination=argv[idx];
```

```
                                                return true;
                                                break;
                                        default:
                                                my_cmdErrors(CMD_INVALID_FILES);
                                                progUsage(argv[0]);
                                                return false;
                                                break;
                                }
                        }
                        idx++;
                }
                return false;

}

void printArgs(struct cmdLnArgs *arg){
                        puts("------------------------------------------------------------------------------");
                        printf("[Action]:%i\n",arg->action);
                        printf("[Filter]:%i\n",arg->filter);
                        printf("[Compress Level]:%i\n",arg->compressLevel);
                        printf("[Source]:%s\n",arg->source);
                        printf("[Destination]:%s\n",arg->destination);
                        printf("[DestParsed]:%i\n",arg->destParsed);
                        puts("------------------------------------------------------------------------------");
}
void cleanNewFileName(char * destination){
                        free(destination);
}
/**
 * Na ausência do ficheiro de destino, gera o seu nome como:
 * :: Na realização da compressão
 * ficheiro_origem.DEFAULT_EXT_NAME (.z)
 * :: Na realização da descompressão
 * ficheiro_origem.SEM DEFAULT_EXT_NAME (sem .z)
 *
 * No Caso da extensão não ser conhecida não realiza a operação.
 * */
unsigned int  newOutputFile(struct cmdLnArgs *myargs, char *outputfile){
                int sizeOfExt=0;
                int equal=0;
                        sizeOfExt=(myargs->action == compress_action)?strlen(myargs->source)
+strlen(DEFAULT_EXT_NAME):strlen(myargs->source)-strlen(DEFAULT_EXT_NAME);

                        outputfile=(char *)malloc(sizeof(char)*(sizeOfExt));
                        if (outputfile == NULL){
                                my_errors(Z_MEM_ERROR);
                                return false;
                        }
                        if (myargs->action == compress_action){
                                myargs->destination=strcpy(outputfile,myargs->source);
                                myargs->destination=strcat(outputfile,DEFAULT_EXT_NAME);
                        }else{
                                equal=strcmp(&(myargs->source[sizeOfExt]),DEFAULT_EXT_NAME);
                                if(equal){
                                                my_cmdErrors(CMD_INVALID_EXT);
                                                return false;
                                }

                                myargs->destination=strncpy(outputfile,myargs->source,sizeOfExt);
                        }
                        return true;
}

int main(int argc, char **argv)
{
    int ret=0;
    myArgs myargs;
    char * outputfile=0;
    FILE *sourceFile;
    FILE *destinationFile;


    if(argc == 1){
                progUsage(argv[0]);
                return UNSUCCESS;
        }

        init_Args(&myargs);
        ret=parseArgs(&myargs,argc,argv);
    if (ret == false) {
                progUsage(argv[0]);
                return UNSUCCESS;
        }


        if (myargs.destination == NULL){
                ret=newOutputFile(&myargs,outputfile);
                if (ret == false){
                        return UNSUCCESS;
                }
                myargs.destParsed=true;
        }



        /**
         * Processamento do ficheiro
         **/
        sourceFile=fopen(myargs.source,"rb");
        destinationFile=fopen(myargs.destination,"wr");
        SET_BINARY_MODE(sourceFile);
        SET_BINARY_MODE(destinationFile);
```

```
        printf("Processing file: %s into %s ...", myargs.source,myargs.destination);
        if (myargs.action == compress_action){
                ret = my_compress(sourceFile,destinationFile, myargs.compressLevel);
        }else{
                ret = my_decompress(sourceFile, destinationFile);
        }
        if (ret != Z_OK){
                        printf("--- ::: ERROR. File not processed!\n");
                        my_errors(ret);
                        if(myargs.destParsed){
                                cleanNewFileName(outputfile);
                        }
                        return UNSUCCESS;
        }
        printf(" ::: DONE File successufly processed!\n");


        if(myargs.destParsed){
                cleanNewFileName(outputfile);
        }
        return ret;
}
```

No quadro seguinte podemos ver o tamanho ocupado pelo ficheiro original e pelas versões compactadas do mesmo com nível de compressão 0 (sem compactação), 5 e 9.

Ficheiro	Tipo img.	Tamanho inicial	Tamanho (nível 0)	Tamanho (nível 5)	%	Tamanho (nível 9)	%
blackbuck.ppm	P3	1.989.962	1.990.268	356.377	18%	332.687	17%
boxes_1.ppm	P6	11.921	11.932	2.219	19%	2.189	18%
feep.pgm	P3	571	582	145	25%	147	26%
feep.ppm	P2	182	193	60	33%	60	33%
house_1.ppm	P6	43.971	43.987	37.064	84%	37.058	84%
J_letter.pbm	P1	175	186	79	45%	79	45%
moreboxes_1.ppm	P6	11.920	11.931	2.218	19%	2.188	18%
sign_1.ppm	P6	29.416	29.427	27.431	93%	27.431	93%
snail.ppm	P3	760.829	760950	52.686	7%	47844	6%
stop_1.ppm	P6	29.416	29.427	27.529	94%	27.529	94%
synth_1.ppm	P6	30.060	30.071	208	1%	196	1%
tree_1.ppm	P6	53.082	53.098	47.537	90%	47.537	90%
west_1.ppm	P6	237.183	237.224	209.585	88%	209.585	88%

Podemos verificar que o nível de compactação é bastante alto em quase todos os ficheiros embora nos mais pequenos e que estão guardados por bytes em vez de caracteres a redução não é tão grande.

Exercício 3 e 4

Os exercício 3 e 4 foram efectuados em conjunto uma vez que tratam de filtrar e "desfiltrar" o conteúdo dos ficheiros de acordo com os filtros usados nas imagens PNG.

Analisando os filtros utilizados facilmente se percebe que são usados 5 filtros diferentes (sendo que um deles é não ser aplicado qualquer filtro). Criámos um módulo *filters* para lidar com os mesmos sem nos preocuparmos com o resto do tratamento do ficheiro.

Na operação de filtragem optámos por tirar partido do processamento do ficheiro pela zlib para passar a linha a ser filtrada (e a anterior, caso exista) e recebemos de volta a linha após a filtragem para ser compactada.

Na operação de desfiltragem, o ficheiro tem de ser descompactado primeiro antes de termos acesso ao header do mesmo para podermos passar os dados necessários à desfiltragem do mesmo. Esta questão levantou um problema uma vez que não podemos proceder à filtragem e

desfiltragem do ficheiro do mesmo modo.

Assim para efectuarmos a desfiltragem temos de "correr" o ficheiro novamente efectuarmos a desfiltragem linha a linha como foi com a filtragem.

Na descompressão foi onde encontramos mais um problema uma vez que a descompressão está a dar erro e não conseguimos descobrir onde se encontra o problema uma vez que este está a acontecer no processo de compressão e não de filtragem. O mesmo se passa com a descompressão.

Adicionalmente, no processo de filtragem/desfiltragem for a musados *cases* que pretendiamos substituir mais tarde por uma estrutura de ponteiros para função mas devido aos problemas encontrados não nos foi possível.