image.h

```
#ifndef _BMP_H_
#define _BMP_H_
#include <stdint.h>
typedef enum
{
   SUCCESS,
   FILE_NOT_FOUND,
   Operation_Not_Supported,
   Cannot_Write_Image
}status_t;

typedef struct _image Image_t;
// reads a image
Image_t *proj_image_readFromFile(const char* );
// writes a image
status_t *proj_image_writeToFile( Image_t *, const char*);
status_t *proj_image_new_create(const char*,unsigned int,unsigned int);
// release the memory of a image structure
void Image_destroy(Image_t *img);
#endif
```

image.c

```
#include <stdio.h>
#include <stdlib.h>
#include<string.h>
#include "image.h"
// correct values for the header
#define MAGIC_VALUE 0X4D42
#define BITS_PER_PIXEL 24
#define NUM_PLANE 1
#define COMPRESSION 0
#define BITS_PER_BYTE 8
#pragma pack(1)
// A BMP file has a header (54 bytes) and data
typedef unsigned char uc;
typedef struct _BMP_Header_t
{
 uint16_t type; // Magic identifier
 uint32_t size; // File size in bytes
```

```
 uint16_t reserved1; // Not used
 uint16_t reserved2; // Not used
 uint32_t offset; //Offset where image starts in file
 uint32_t header_size; // Header size in bytes
 uint32_t width; // Width of the image
 uint32_t height; // Height of image
 uint16_t planes; // Number of color planes
 uint16_t bits; // Bits per pixel
 uint32_t compression; // Compression type
 uint32_t imagesize; // Image size in bytes
 uint32_t xresolution; // Pixels per meter
 uint32_t yresolution; // Pixels per meter
 uint32_t ncolours; // Number of colors
 uint32_t importantcolours; // Important colors
} BMP_Header;

typedef struct _image
{
 BMP_Header header;
 unsigned int data_size;
 unsigned int width;
 unsigned int height;
 unsigned int bytes_per_pixel;
 unsigned char * data;
} Image_t;

static int checkHeader(BMP_Header * hdr)
{
 if ((hdr -> type) != MAGIC_VALUE)
 {
 return 0;
 }
 if ((hdr -> bits) != BITS_PER_PIXEL)
 {
 return 0;
 }
 if ((hdr -> planes) != NUM_PLANE)
 {
 return 0;
 }
 if ((hdr -> compression) != COMPRESSION)
 {
```

```
 return 0;
 }
 return 1;
}
// close opened file and release memory
Image_t * cleanUp(FILE * fptr, Image_t * img)
{
 if (fptr != NULL)
 {
 fclose (fptr);
 }
 if (img != NULL)
 {
 if (img -> data != NULL)
{
 free (img -> data);
}
 free (img);
 }
 return NULL;
}
Image_t *proj_image_readFromFile(const char *str)
{
 FILE * fptr = NULL;
 Image_t *img = NULL;
 fptr = fopen(str, "rb");
 if (fptr == NULL)
 {
 return cleanUp(fptr, img);
 }
 img = malloc(sizeof(Image_t));
 if (img == NULL)
 {
 return cleanUp(fptr, img);
 }
 // read the header
 if (fread(& (img -> header), sizeof(BMP_Header), 1, fptr) != 1)
 {
 // fread fails
 return cleanUp(fptr, img);
 }
 if (checkHeader(& (img -> header)) == 0)
```

```
{
return cleanUp(fptr, img);
}
img -> data_size = (img -> header).size - sizeof(BMP_Header);
img -> width = (img -> header).width;
img -> height = (img -> header).height;
img -> bytes_per_pixel = (img -> header).bits / BITS_PER_BYTE;
img -> data =malloc(sizeof(unsigned char) * (img -> data_size));
if ((img -> data) == NULL)
{
// malloc fail
return cleanUp(fptr, img);
}
if (fread(img -> data, sizeof(char), img -> data_size,
fptr) != (img -> data_size))
{
// fread fails
return cleanUp(fptr, img);
}
char onebyte;
if (fread(& onebyte, sizeof(char), 1, fptr) != 0)
{
// not at the of the file but the file still has data
return cleanUp(fptr, img);
}
// everything successful
fclose (fptr);
return img;
}

status_t *proj_image_writeToFile( Image_t *img,const char* str)
{
FILE * fptr = NULL;
fptr = fopen(str, "wb"); //Performs Write Operation

if (fptr == NULL || img == NULL)
{
return (status_t*)FILE_NOT_FOUND;
}
// write the header first
else if (fwrite(& (img -> header), sizeof(BMP_Header), 1,fptr) != 1)
{
```

```
 // fwrite fails
 fclose (fptr);
 return (status_t*)Cannot_Write_Image;
 }
 else if(fwrite(img -> data, sizeof(char), img -> data_size, fptr) != (img -> data_size))
 {
 // fwrite fails
 fclose (fptr);
 return (status_t*)Cannot_Write_Image;
 }
 // everything successful
 else
{
 fclose (fptr);
 return (status_t*)SUCCESS;
}
}

status_t *proj_image_new_create( const char* str, unsigned int width , unsigned int height)
{

 FILE * fptr = NULL;
 fptr = fopen(str, "wb"); //Performs Write Operation
 // img -> width = width;
 // Image_t *img1= (Image_t *)img;
 Image_t *img1=NULL;
  img1 = malloc(sizeof(Image_t));

(img1 -> header).type=(int)19778; // Magic identifier
 // printf("\nHelloooooooooooooooooooooooooooo");
 (img1 -> header).reserved1=0; // Not used
 (img1 -> header).reserved2=0; // Not used
 // (img1 -> header).offset; //Offset where image starts in file
 (img1 -> header).header_size=40; // Header size in bytes
 (img1 -> header).width; // Width of the image
 (img1 -> header).height; // Height of image
 (img1 -> header).planes=1; // Number of color planes
 (img1 -> header).bits=24; // Bits per pixel
 (img1 -> header).compression=0; // Compression type
 // (img1 -> header).imagesize; // Image size in bytes
 (img1 -> header).xresolution=11808; // Pixels per meter
 (img1 -> header).yresolution=11808; // Pixels per meter
```

```
 (img1 -> header).ncolours=0; // Number of colors
 (img1 -> header).importantcolours=0;


 (img1 -> header).width=width;
 (img1 -> header).offset= sizeof(BMP_Header);
 // img -> height = height;
 (img1 -> header).height=height;
int r,g,b;
 int stride=((24*width)>>5)<<2;
(img1 -> header).size = (img1 -> header).offset+height*stride;
(img1 -> header).imagesize= stride*height;



img1 -> data_size=stride*height- sizeof(BMP_Header);
img1 -> width=width;
 img1 -> height=height;
img1 -> bytes_per_pixel=3;

 if (fptr == NULL || img1 == NULL)
 {
 return (status_t*)FILE_NOT_FOUND;
 }
 // write the header first
if (fwrite(& (img1 -> header), sizeof(BMP_Header), 1,fptr) != 1)
 {
 // fwrite fails
 fclose (fptr);
 return (status_t*)Cannot_Write_Image;
 }
 uc *array=(uc*)malloc(stride); //Allocate space
   for(int i=0;i<height;i++){
      for(int j=0;j<width;j++){
         r=44;
         g=177;
         b=160;
         array[3*j]=(uc)r;
         array[3*j+1]=(uc)g;
         array[3*j+2]=(uc)b;
      }
      fwrite(array,1,stride,fptr);
```

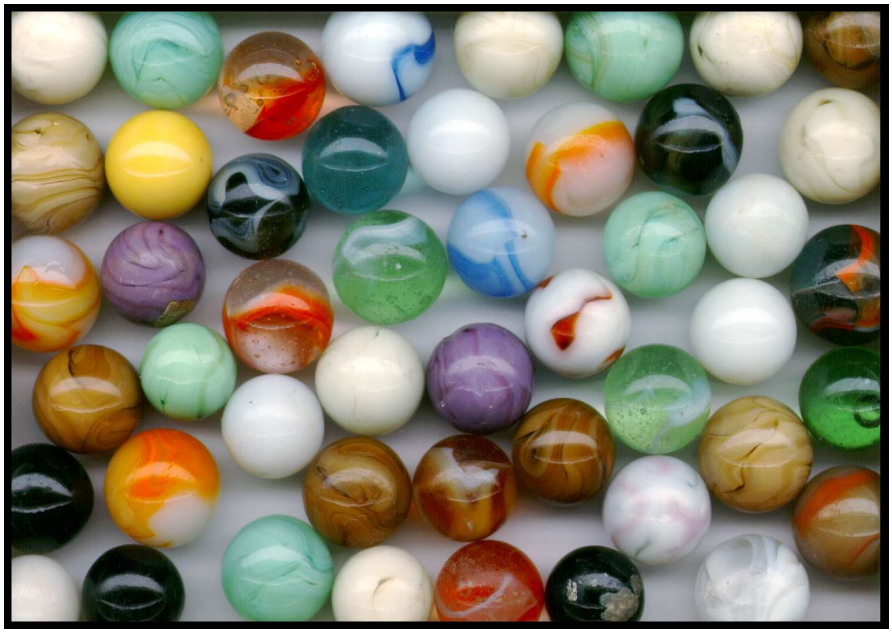
```
   }
 if(stride!=((24*width)>>5)<<2)
 {
 // fwrite fails
 fclose (fptr);
 free(array);
 array=NULL;
 return (status_t*)Cannot_Write_Image;
 }
 // everything successful
else
{
 fclose (fptr);
 return (status_t*)SUCCESS;
}
}

void Image_destroy(Image_t *img)
{
 free (img -> data);
 free (img);
}
```

test.c

```
#include <stdio.h>
#include <stdlib.h>
#include<string.h>
#include "image.h"
int main()
{
//FILE* fptr = fopen("pic.bmp", "rb");
Image_t *img = proj_image_readFromFile("pic.bmp");
if(img == NULL){
  printf("Cannot_Read_\n");
}else
{
   printf("Image read successfully\n");
}
// printf("%s %s",argv[1],argv[3]);
```

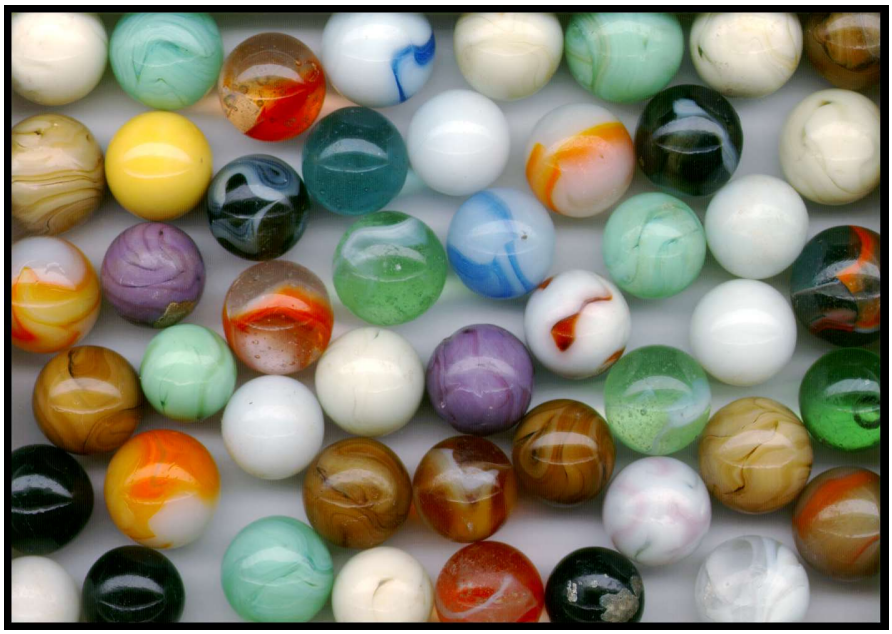
```
int a = (int)proj_image_writeToFile(img, "new.bmp");
// printf("%d\n",a);
if(a == SUCCESS){
  printf("Code %d : SUCCESS\n",a);
}
else if(a == Cannot_Write_Image){
  printf("Code %d : Cannot_Write_Image\n",a);
}
else if(a == FILE_NOT_FOUND){
  printf("Code %d : FILE_NOT_FOUND\n",a);
}
else if(a == Operation_Not_Supported){
  printf("Code %d : Operation_Not_Supported\n",a);
}else
{
    printf("Code %d : FAILURE\n",a);

}

a = (int)proj_image_new_create("new1.bmp",500,500);
if(a == SUCCESS){
  printf("Code %d : SUCCESS\n",a);
}
else if(a == Cannot_Write_Image){
  printf("Code %d : Cannot_Write_Image\n",a);
}
else if(a == FILE_NOT_FOUND){
  printf("Code %d : FILE_NOT_FOUND\n",a);
}
else if(a == Operation_Not_Supported){
  printf("Code %d : Operation_Not_Supported\n",a);
}else
{
    printf("Code %d : FAILURE\n",a);

}
 //Destroy the image
 Image_destroy(img);
}
```

Input image:



pic.bmp

Output image:



new.bmp
(write image)

new1.bmp
(500x500, 733KB)
(create image)

Output:

```
C:\Users\MANOJ\OneDrive\Desktop\amit project>a
Image read successfully
Code 0 : SUCCESS
Code 0 : SUCCESS
```

Output