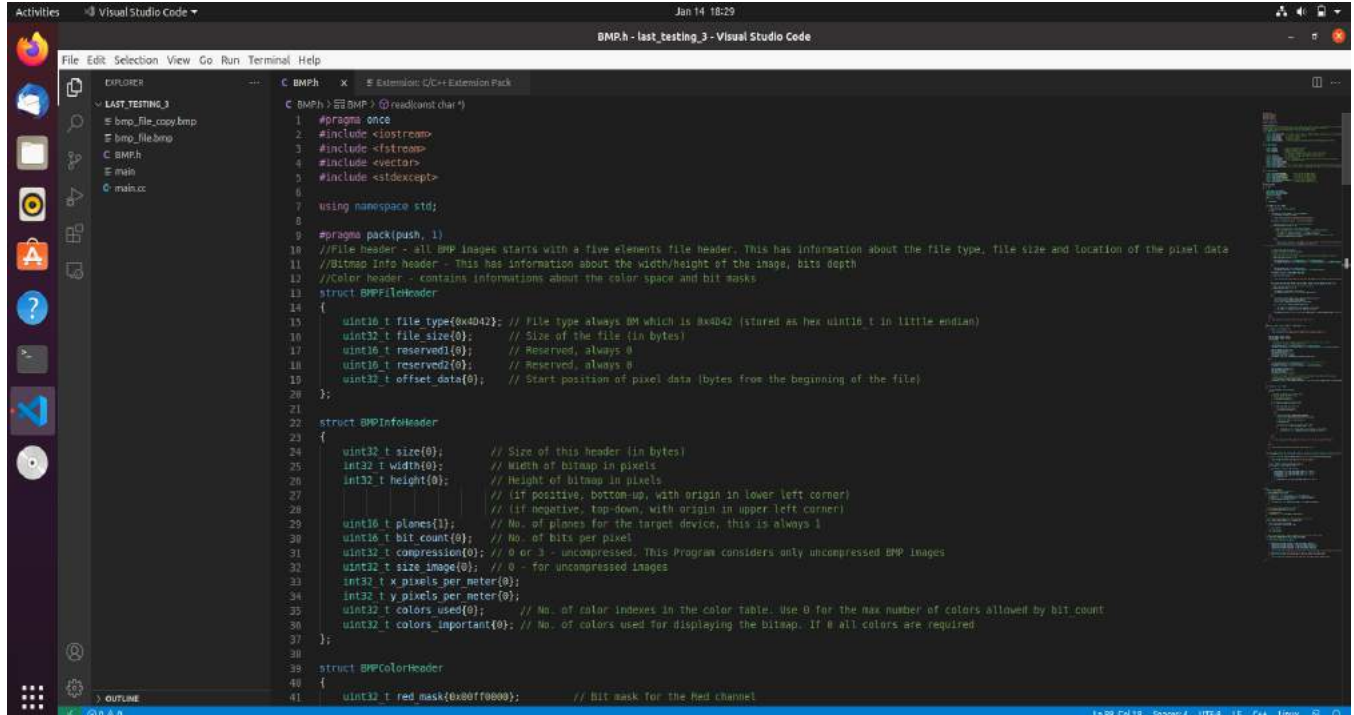
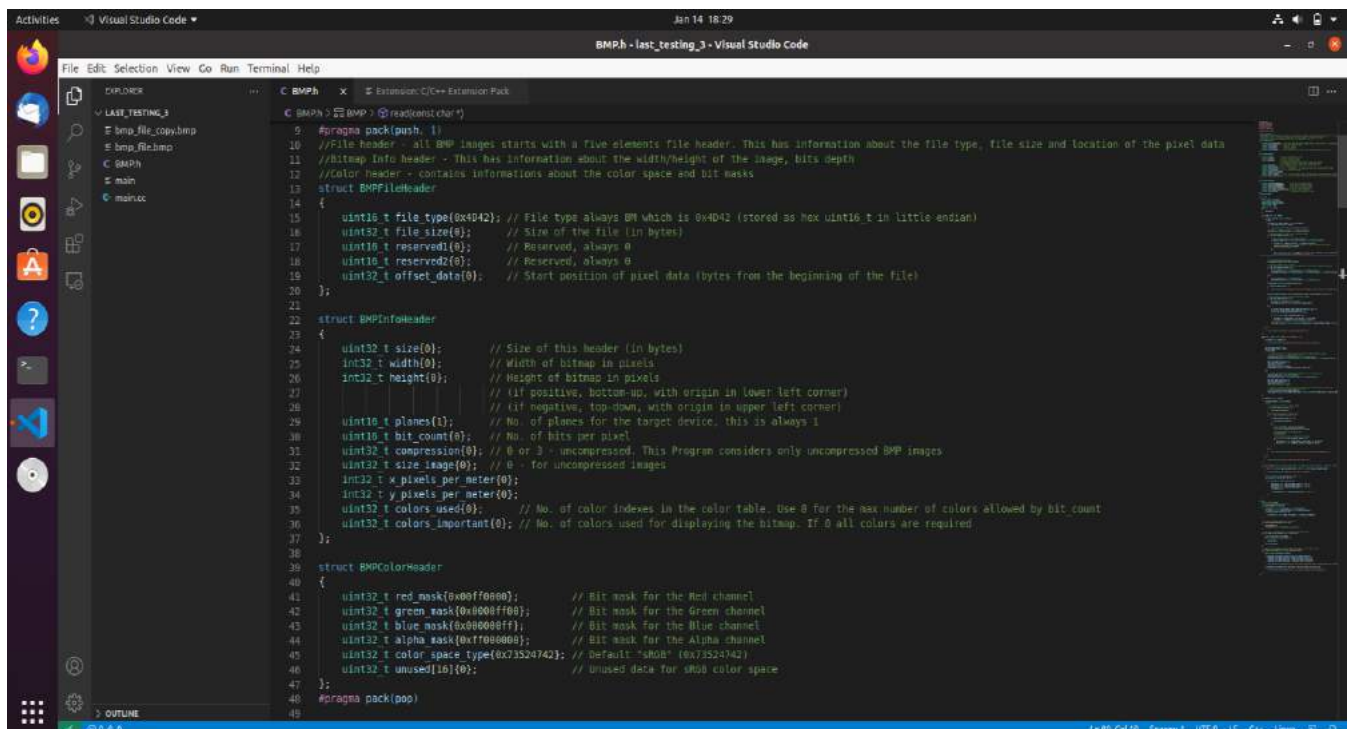


Task #3 Implement an Image File Reader and Writer

1. Structure of BMPFileHeader, BMPInfoHeader, BMPColorHeader.

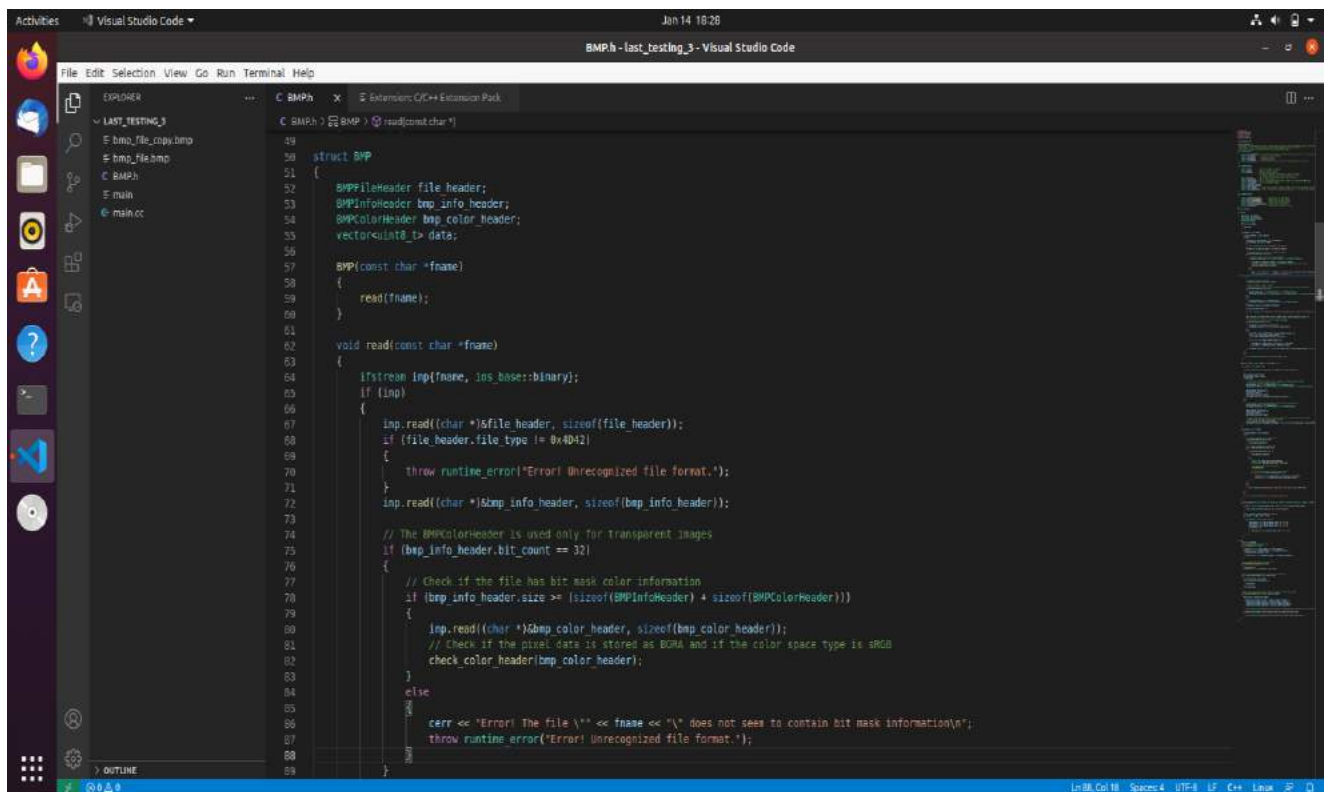


```
1 #pragma once
2 #include <iostream>
3 #include <fstream>
4 #include <vector>
5 #include <string>
6 using namespace std;
7
8 #pragma pack(push, 1)
9
10 //File header - all BMP images starts with a five elements file header. This has information about the file type, file size and location of the pixel data
11 //Bitmap Info header - This has information about the width/height of the image, bits depth
12 //Color header - contains informations about the color space and bit masks
13 struct BMPFileHeader
14 {
15     uint16_t file_type{0x4D52}; // File type always BM which is 0x4D52 (stored as hex uint16_t in little endian)
16     uint32_t file_size(0); // Size of the file (in bytes)
17     uint16_t reserved1(0); // Reserved, always 0
18     uint16_t reserved2(0); // Reserved, always 0
19     uint32_t offset_data(0); // Start position of pixel data (bytes from the beginning of the file)
20 };
21
22 struct BMPInfoHeader
23 {
24     uint32_t size(0); // Size of this header (in bytes)
25     int32_t width(0); // Width of bitmap in pixels
26     int32_t height(0); // Height of bitmap in pixels
27     // (if positive, bottom-up, with origin in lower left corner)
28     // (if negative, top-down, with origin in upper left corner)
29     uint16_t planes(1); // No. of planes for the target device, this is always 1
30     uint16_t bit_count(0); // No. of bits per pixel
31     uint32_t compression(0); // 0 or 3 - uncompressed. This Program considers only uncompressed BMP images
32     uint32_t size_image(0); // 0 - for uncompressed images
33     int32_t x_pixels_per_meter(0);
34     int32_t y_pixels_per_meter(0);
35     uint32_t colors_used(0); // No. of color indexes in the color table. Use 0 for the max number of colors allowed by bit count
36     uint32_t colors_important(0); // No. of colors used for displaying the bitmap. If 0 all colors are required
37 };
38
39 struct BMPColorHeader
40 {
41     uint32_t red_mask{0x00ff0000}; // Bit mask for the Red channel
```



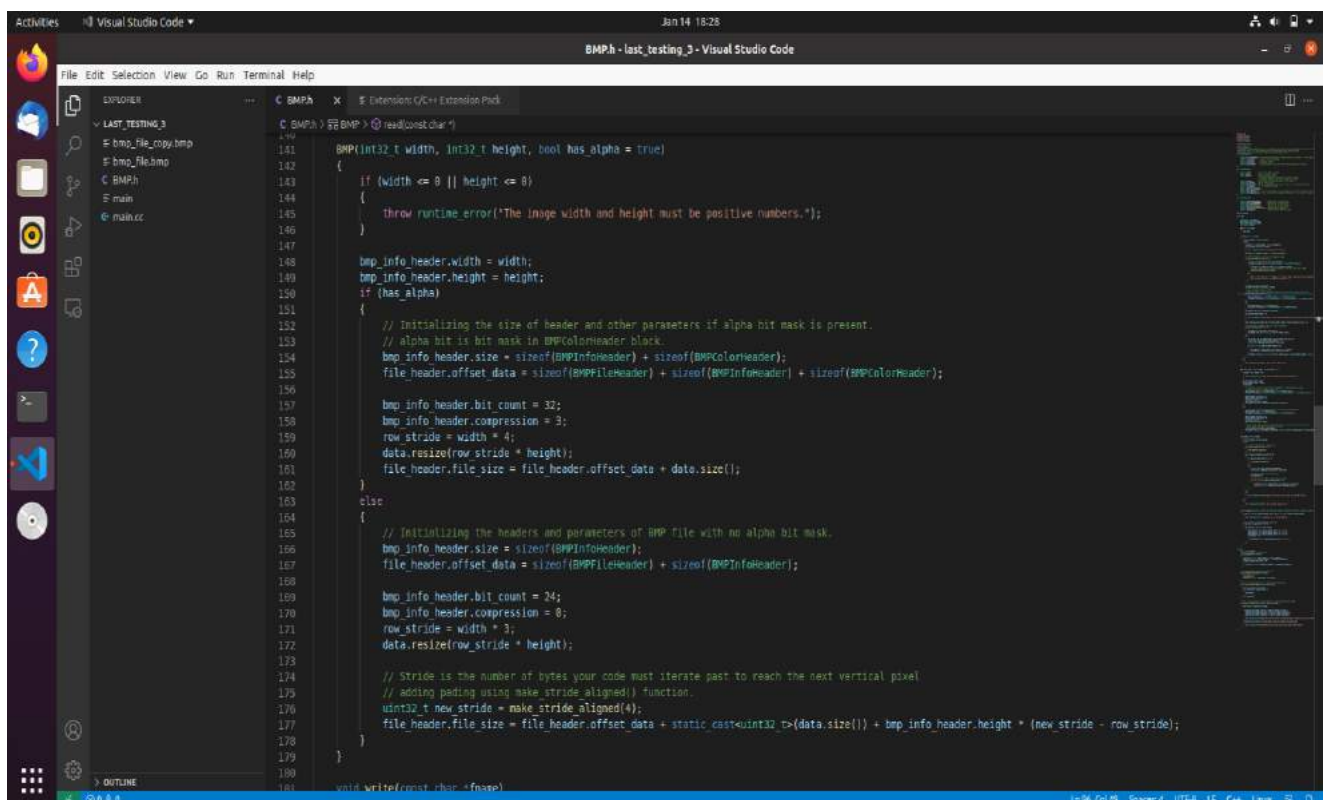
```
42     uint32_t green_mask{0x0000ff00}; // Bit mask for the Green channel
43     uint32_t blue_mask{0x000000ff}; // Bit mask for the Blue channel
44     uint32_t alpha_mask{0xff000000}; // Bit mask for the Alpha channel
45     uint32_t color_space_type{0x73524742}; // Default "sRGB" (0x73524742)
46     uint32_t unused16(0); // unused data for sRGB color space
47 };
48 #pragma pack(pop)
49
```

2. BMP constructor



The screenshot shows the Visual Studio Code editor with the file `BMP.h` open. The code defines a `BMP` struct and a `BMP` constructor. The struct contains a `BMPFileHeader`, a `BMPInfoHeader`, a `BMPColorHeader`, and a `vector<uint8_t>` for data. The constructor takes a filename and calls `read`. The `read` function opens the file in binary mode and checks for a valid BMP file format. It reads the file header and info header, and checks for a bit mask color header. If the file has a bit mask color header, it reads the color header and checks for a valid color space type. If the file does not have a bit mask color header, it throws a runtime error.

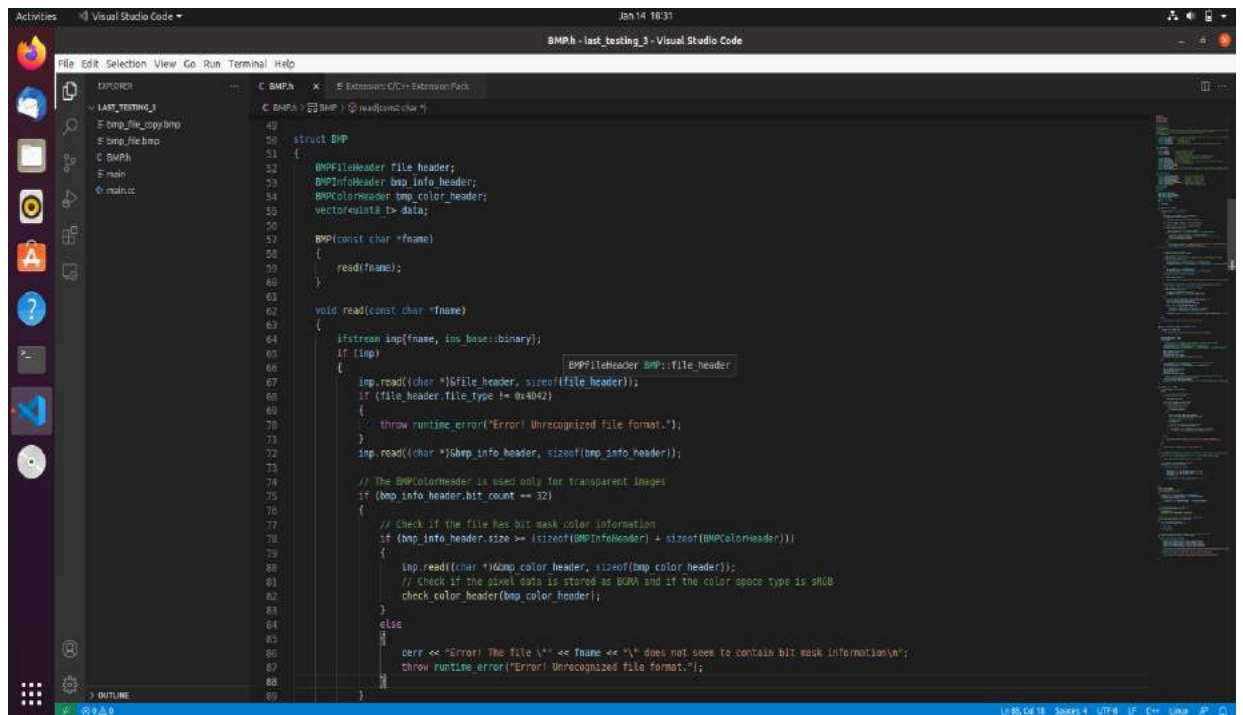
```
39 struct BMP
40 {
41     BMPFileHeader file_header;
42     BMPInfoHeader bmp_info_header;
43     BMPColorHeader bmp_color_header;
44     vector<uint8_t> data;
45
46     BMP(const char *fname)
47     {
48         read(fname);
49     }
50
51     void read(const char *fname)
52     {
53         ifstream inp(fname, ios_base::binary);
54         if (!inp)
55         {
56             throw runtime_error("Error! Unrecognized file format.");
57         }
58         inp.read((char *) &file_header, sizeof(file_header));
59         if (file_header.file_type != 0x4D52)
60         {
61             throw runtime_error("Error! Unrecognized file format.");
62         }
63         inp.read((char *) &bmp_info_header, sizeof(bmp_info_header));
64
65         // The BMPColorHeader is used only for transparent images
66         if (bmp_info_header.bit_count == 32)
67         {
68             // Check if the file has bit mask color information
69             if (bmp_info_header.size >= (sizeof(BMPInfoHeader) + sizeof(BMPColorHeader)))
70             {
71                 inp.read((char *) &bmp_color_header, sizeof(bmp_color_header));
72                 // Check if the pixel data is stored as BGRA and if the color space type is sRGB
73                 check_color_header(bmp_color_header);
74             }
75             else
76             {
77                 cerr << "Error! The file '" << fname << "' does not seem to contain bit mask information\n";
78                 throw runtime_error("Error! Unrecognized file format.");
79             }
80         }
81     }
82 }
```



The screenshot shows the continuation of the `BMP` constructor implementation in `BMP.h`. The `read` function continues to initialize the `BMP` struct. It checks for a valid image width and height, and initializes the `BMP` struct with the appropriate headers and data. It also checks for a valid color space type and initializes the `BMP` struct with the appropriate headers and data. The `read` function ends by writing the `BMP` struct to the file.

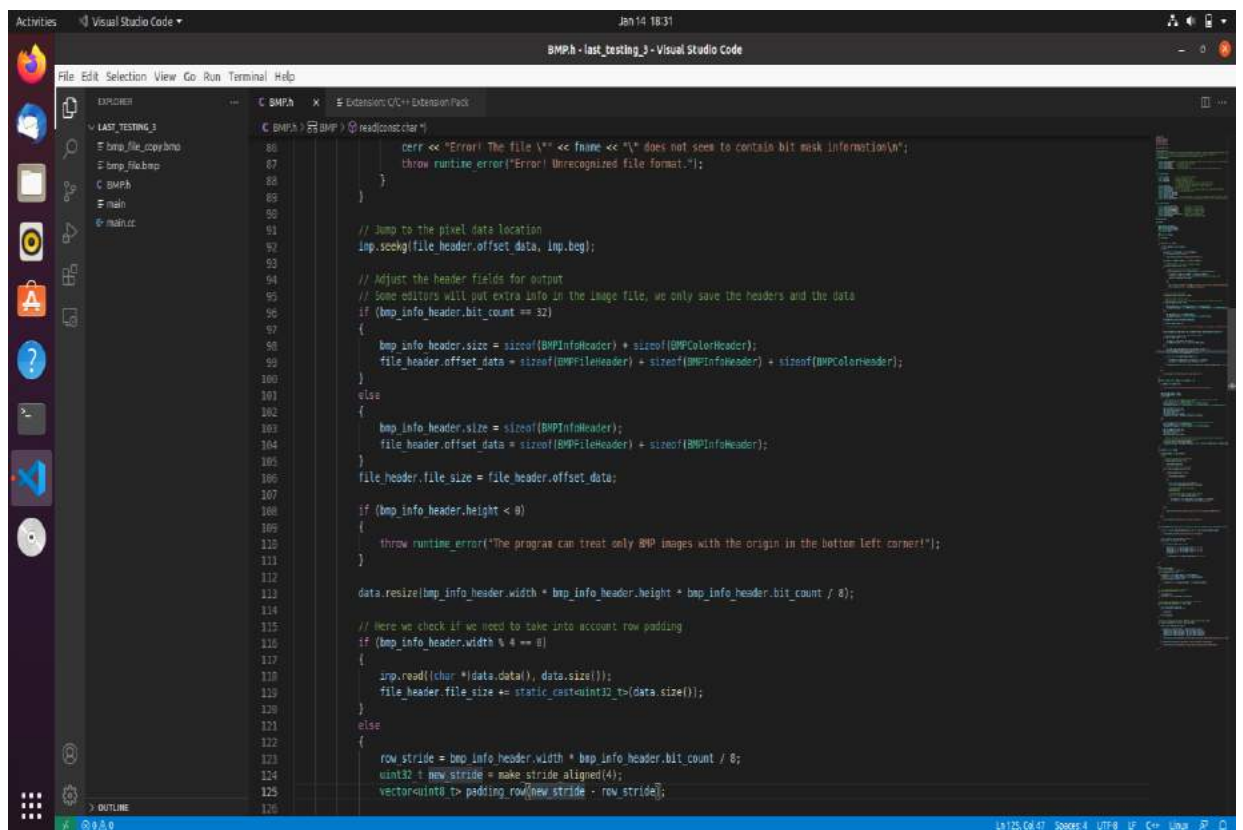
```
141 BMP(int32_t width, int32_t height, bool has_alpha = true)
142 {
143     if (width <= 0 || height <= 0)
144     {
145         throw runtime_error("The image width and height must be positive numbers.");
146     }
147
148     bmp_info_header.width = width;
149     bmp_info_header.height = height;
150     if (has_alpha)
151     {
152         // Initializing the size of header and other parameters if alpha bit mask is present.
153         // alpha bit is bit mask in BMPColorHeader block.
154         bmp_info_header.size = sizeof(BMPInfoHeader) + sizeof(BMPColorHeader);
155         file_header.offset_data = sizeof(BMPFileHeader) + sizeof(BMPInfoHeader) + sizeof(BMPColorHeader);
156
157         bmp_info_header.bit_count = 32;
158         bmp_info_header.compression = 0;
159         row_stride = width * 4;
160         data.resize(row_stride * height);
161         file_header.file_size = file_header.offset_data + data.size();
162     }
163     else
164     {
165         // Initializing the headers and parameters of BMP file with no alpha bit mask.
166         bmp_info_header.size = sizeof(BMPInfoHeader);
167         file_header.offset_data = sizeof(BMPFileHeader) + sizeof(BMPInfoHeader);
168
169         bmp_info_header.bit_count = 24;
170         bmp_info_header.compression = 0;
171         row_stride = width * 3;
172         data.resize(row_stride * height);
173
174         // Stride is the number of bytes your code must iterate past to reach the next vertical pixel
175         // adding padding using make_stride_aligned() function.
176         uint32_t new_stride = make_stride_aligned(4);
177         file_header.file_size = file_header.offset_data + static_cast<uint32_t>(data.size()) + bmp_info_header.height * (new_stride - row_stride);
178     }
179 }
180
181 void write(const char *fname)
```

3. Function to Read Bitmap Image



This screenshot shows the Visual Studio Code editor with the `BMP.h` file open. The `BMP` struct is defined with fields for `BMPFileHeader`, `BMPInfoHeader`, `BMPColorHeader`, and a `vector<uint8_t>` for `data`. The `read` function is implemented, which takes a filename and reads the file into memory. It checks for the file format, reads the headers, and checks for bit mask information. The function is currently incomplete, with a placeholder for the `data` vector.

```
49 struct BMP
50 {
51     BMPFileHeader file_header;
52     BMPInfoHeader bmp_info_header;
53     BMPColorHeader bmp_color_header;
54     vector<uint8_t> data;
55
56     BMP(const char *fname)
57     {
58         read(fname);
59     }
60
61 void read(const char *fname)
62 {
63     ifstream inp(fname, ios_base::binary);
64     if (!inp)
65     {
66         throw runtime_error("Error: File not found");
67     }
68     inp.read((char *) &file_header, sizeof(file_header));
69     if (file_header.file_type != 0x4D42)
70     {
71         throw runtime_error("Error: Unrecognized file format.");
72     }
73     inp.read((char *) &bmp_info_header, sizeof(bmp_info_header));
74     // The BMPColorHeader is used only for transparent images
75     if (bmp_info_header.bit_count == 32)
76     {
77         // Check if the file has bit mask color information
78         if (bmp_info_header.size == (sizeof(BMPInfoHeader) + sizeof(BMPColorHeader)))
79         {
80             inp.read((char *) &bmp_color_header, sizeof(bmp_color_header));
81             // Check if the pixel data is stored as BGR and if the color space type is sRGB
82             check_color_header(bmp_color_header);
83         }
84         else
85         {
86             cerr << "Error: The file '" << fname << "' does not seem to contain bit mask information\n";
87             throw runtime_error("Error: Unrecognized file format.");
88         }
89     }
90 }
```



This screenshot shows the Visual Studio Code editor with the `BMP.h` file open. The `BMP` struct is defined with fields for `BMPFileHeader`, `BMPInfoHeader`, `BMPColorHeader`, and a `vector<uint8_t>` for `data`. The `read` function is implemented, which takes a filename and reads the file into memory. It checks for the file format, reads the headers, and checks for bit mask information. The function is currently incomplete, with a placeholder for the `data` vector.

```
86     cerr << "Error: The file '" << fname << "' does not seem to contain bit mask information\n";
87     throw runtime_error("Error: Unrecognized file format.");
88 }
89
90 // Jump to the pixel data location
91 inp.seekg(file_header.offset_data, inp.beg);
92
93 // Adjust the header fields for output
94 // Some editors will put extra info in the image file, we only save the headers and the data
95 if (bmp_info_header.bit_count == 32)
96 {
97     bmp_info_header.size = sizeof(BMPInfoHeader) + sizeof(BMPColorHeader);
98     file_header.offset_data = sizeof(BMPFileHeader) + sizeof(BMPInfoHeader) + sizeof(BMPColorHeader);
99 }
100 else
101 {
102     bmp_info_header.size = sizeof(BMPInfoHeader);
103     file_header.offset_data = sizeof(BMPFileHeader) + sizeof(BMPInfoHeader);
104 }
105 file_header.file_size = file_header.offset_data;
106
107 if (bmp_info_header.height < 0)
108 {
109     throw runtime_error("The program can treat only BMP images with the origin in the bottom left corner!");
110 }
111
112 data.resize(bmp_info_header.width * bmp_info_header.height * bmp_info_header.bit_count / 8);
113
114 // Here we check if we need to take into account row padding
115 if (bmp_info_header.width % 4 != 0)
116 {
117     inp.read((char *) data.data(), data.size());
118     file_header.file_size += static_cast<int32_t>(data.size());
119 }
120 else
121 {
122     row_stride = bmp_info_header.width * bmp_info_header.bit_count / 8;
123     uint32_t new_stride = make_stride_aligned(4);
124     vector<uint8_t> padding_row(new_stride - row_stride);
125 }
```

```

198 }
199 else
200 {
201     bmp_info_header.size = sizeof(BMPInfoHeader);
202     file_header.offset_data = sizeof(BMPFileHeader) + sizeof(BMPInfoHeader);
203 }
204 file_header.file_size = file_header.offset_data;
205
206 if (bmp_info_header.height < 0)
207 {
208     throw runtime_error("The program can treat only BMP images with the origin in the bottom left corner!");
209 }
210
211 data.resize(bmp_info_header.width * bmp_info_header.height * bmp_info_header.bit_count / 8);
212
213 // Here we check if we need to take into account row padding
214 if (bmp_info_header.width % 4 != 0)
215 {
216     inp.read((char *)data.data(), data.size());
217     file_header.file_size += static_cast<uint32_t>(data.size());
218 }
219 else
220 {
221     row_stride = bmp_info_header.width * bmp_info_header.bit_count / 8;
222     uint32_t new_stride = make_stride_aligned(4);
223     vector<uint8_t> padding_row(new_stride - row_stride);
224
225     for (int y = 0; y < bmp_info_header.height; ++y)
226     {
227         inp.read((char *)data.data() + row_stride * y, row_stride);
228         inp.read((char *)padding_row.data(), padding_row.size());
229     }
230     file_header.file_size += static_cast<uint32_t>(data.size()) + bmp_info_header.height * static_cast<uint32_t>(padding_row.size());
231 }
232 }
233 else
234 {
235     throw runtime_error("Unable to open the input image file.");
236 }
237 }
238 }
239 }

```

4. Function to Write Bitmap Image

```

198 void write(const char *fname)
199 {
200     ofstream of(fname, ios_base::binary);
201     if (of)
202     {
203         // To check if the bmp file is 32-bit format.
204         if (bmp_info_header.bit_count == 32)
205         {
206             write_headers_and_data(of);
207         }
208         // To check if the bmp file is 24-bit count.
209         else if (bmp_info_header.bit_count == 24)
210         {
211             if (bmp_info_header.width % 4 != 0)
212             {
213                 write_headers_and_data(of);
214             }
215             else
216             {
217                 uint32_t new_stride = make_stride_aligned(4);
218                 vector<uint8_t> padding_row(new_stride - row_stride);
219
220                 // Write the headers in the new bmp file.
221                 write_headers(of);
222
223                 // Write data and stride in new bmp file till height.
224                 for (int y = 0; y < bmp_info_header.height; ++y)
225                 {
226                     of.write((const char *)data.data() + row_stride * y, row_stride);
227                     of.write((const char *)padding_row.data(), padding_row.size());
228                 }
229             }
230         }
231     }
232     else
233     {
234         // Error handling
235     }
236 }

```

tonystark:~/testing_tasks/last_testing_35

Downloading OpenDebugAD7 (Linux x64)


```
188 write_headers_and_data(of);
189
190 // To check if the bmp file is 24-bit count.
191 else if (bmp_info_header.bit_count == 24)
192 {
193     if (bmp_info_header.width % 4 != 0)
194     {
195         write_headers_and_data(of);
196     }
197     else
198     {
199         uint32_t new_stride = make_stride_aligned(4);
200         vector<uint8_t> padding_row(new_stride - row_stride);
201
202         // Write the headers in the new bmp file.
203         write_headers(of);
204
205         // Write data and stride in new bmp file till height.
206         for (int y = 0; y < bmp_info_header.height; ++y)
207         {
208             of.write((const char *)data.data() + row_stride * y, row_stride);
209             of.write((const char *)padding_row.data(), padding_row.size());
210         }
211     }
212 }
213
214 }
215
216 {
217     throw runtime_error("The program can treat only 24 or 32 bits per pixel BMP files");
218 }
219
220 }
221
222 {
223     throw runtime_error("Unable to open the output image file.");
224 }
225 }
```

tonystark:~/testing_tasks/last_testing_3\$

```
247 private:
248
249     uint32_t row_stride(0);
250     // To write the headers of new bmp file.
251     void write_headers(ofstream &of)
252     {
253         of.write((const char *)file_header, sizeof(file_header));
254         of.write((const char *)bmp_info_header, sizeof(bmp_info_header));
255         if (bmp_info_header.bit_count == 32)
256         {
257             of.write((const char *)bmp_color_header, sizeof(bmp_color_header));
258         }
259     }
260
261     // To write the headers and data in new bmp file.
262     void write_headers_and_data(ofstream &of)
263     {
264         write_headers(of);
265         of.write((const char *)data.data(), data.size());
266     }
267
268     // Add 1 to the row_stride until it is divisible with align_stride
269     uint32_t make_stride_aligned(uint32_t align_stride)
270     {
271         uint32_t new_stride = row_stride;
272         while (new_stride % align_stride != 0)
273         {
274             new_stride++;
275         }
276         return new_stride;
277     }
278
279     // Check if the pixel data is stored as RGBA and if the color space type is sRGB
280     void check_color_header(BMPColorHeader &bmp_color_header)
281     {
282         BMPColorHeader expected_color_header;
```

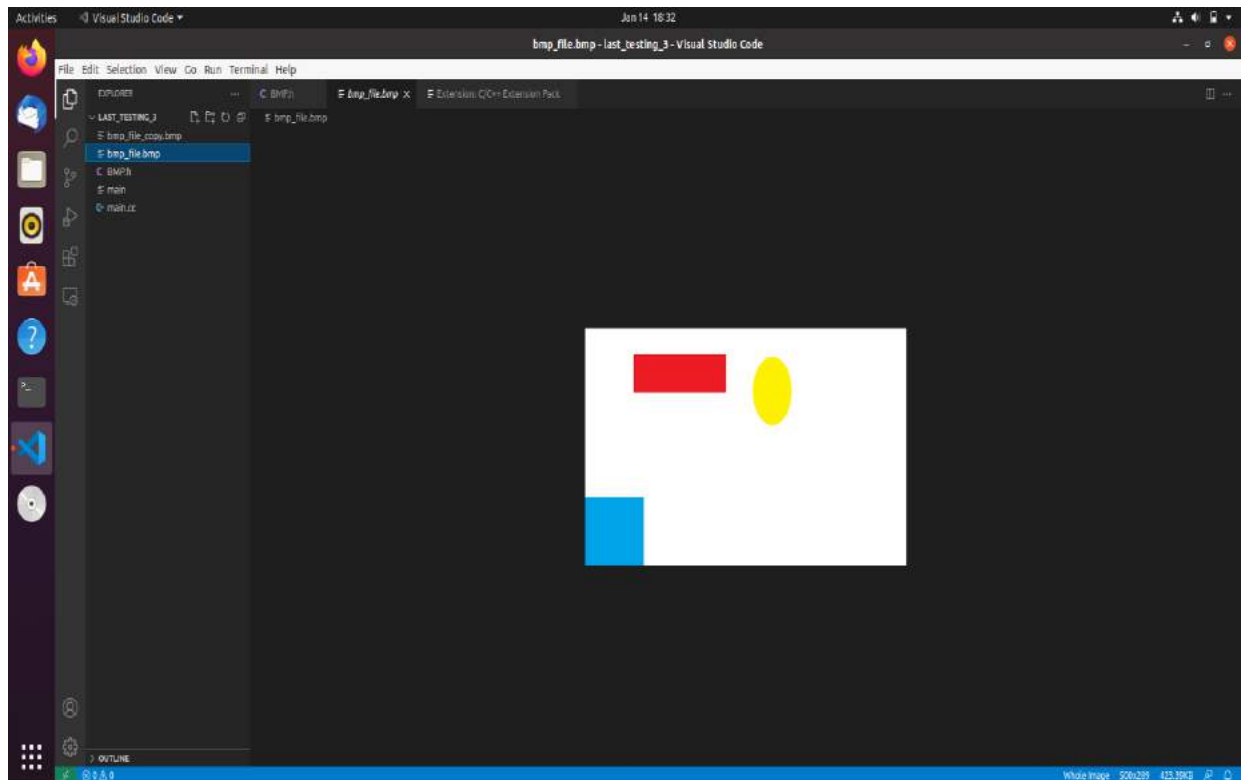
tonystark:~/testing_tasks/last_testing_3\$

```
263 {
264     write_headers(of);
265     of.write((const char *)data.data(), data.size());
266 }
267
268 // Add 1 to the row stride until it is divisible with align_stride
269 uint32_t make_stride_aligned(uint32_t align_stride)
270 {
271     uint32_t new_stride = row_stride;
272     while (new_stride % align_stride != 0)
273     {
274         new_stride++;
275     }
276     return new_stride;
277 }
278
279 // Check if the pixel data is stored as BGRA and if the color space type is sRGB
280 void check_color_header(BMPColorHeader &bmp_color_header)
281 {
282     BMPColorHeader expected_color_header;
283
284     if (expected_color_header.red_mask != bmp_color_header.red_mask ||
285         expected_color_header.blue_mask != bmp_color_header.blue_mask ||
286         expected_color_header.green_mask != bmp_color_header.green_mask ||
287         expected_color_header.alpha_mask != bmp_color_header.alpha_mask)
288     {
289         throw runtime_error("Unexpected color mask format! The program expects the pixel data to be in the BGRA format");
290     }
291     if (expected_color_header.color_space_type != bmp_color_header.color_space_type)
292     {
293         throw runtime_error("Unexpected color space type! The program expects sRGB values");
294     }
295 }
296 }
297
```

5. Main Function

```
1 #include "BMP.h"
2 #include <iostream>
3
4 int main()
5 {
6     // Read an image from disk, modify it and write it back
7     BMP bmp("bmp_file.bmp");
8     bmp.write("bmp_file_copy.bmp");
9     return 0;
10 }
11
```

6. Input (Read) Bitmap Image



7. Output (Write) Bitmap Image

