```
#include <iostream>
#include <vector>
#include <exception>
#include <fstream>
#include <sstream>
#include "GrayLevelImage2D.hpp"
#include <algorithm>
#include <string>

// pas besoin d'ecrire les constructeurs par copie et d'affectation car ils sont generes automatiquement

// mise en place du type
typedef unsigned char GrayLevel; // le type pour les niveaux de gris.

// declare method of class GrayLevelImage2d.hpp
GrayLevelImage2D::GrayLevelImage2D()
{
    m_width = 0;
    m_height = 0;
}

GrayLevelImage2D::GrayLevelImage2D(int w, int h, GrayLevel g)
{
    m_width = w;
    m_height = h;
    // resize rempoli automatiquement de 0
```

```
    m_data.resize(w * h, g);
}

void GrayLevelImage2D::fill(GrayLevel g)
{
    m_data.assign(m_width * m_height, g);
}

int GrayLevelImage2D::w() const
{
    return m_width;
}

int GrayLevelImage2D::h() const
{
    return m_height;
}

GrayLevel GrayLevelImage2D::at(int i, int j) const
{
    return m_data[i + j * m_width];
}

GrayLevel &GrayLevelImage2D::at(int i, int j)
{
    return m_data[i + j * m_width];
```

```
}

// index function
int GrayLevelImage2D::index(int i, int j) const
{
    return i + j * m_width;
}

GrayLevelImage2D::Iterator::Iterator(GrayLevelImage2D &image, int x, int y)
    : Container::iterator(image.m_data.begin() + image.index(x, y))
{
}

GrayLevelImage2D::Iterator GrayLevelImage2D::begin()
{
    return Iterator(*this, 0, 0);
}

GrayLevelImage2D::Iterator GrayLevelImage2D::end()
{
    return Iterator(*this, 0, m_height);
    // ou return Iterator(*this, m_width, m_height-1);
}

GrayLevelImage2D::Iterator GrayLevelImage2D::start(int x, int y)
{
```

```
    return Iterator(*this, x, y);
}

std::pair<int, int> GrayLevelImage2D::position(Iterator it) const
{
    int x = it - m_data.begin();
    int y = x / m_width;
    x = x % m_width;
    return std::make_pair(x, y);
}

std::string readline(std::istream &input)
{
    std::string str;
    do
    {
        getline(input, str);
    } while (str != "" && str[0] == '#');
    return str;
}

bool GrayLevelImage2D::importPGM(std::istream &input)
{
    if (!input.good())
        return false;
    std::string format = readline(input);
```

```
std::string line = readline(input);
std::string delim = " ";
m_width = std::stoi(line.substr(0, line.find(delim)));
line.erase(0, line.find(delim) + delim.length());
m_height = std::stoi(line);
std::cout << m_width << " " << m_height << " " << format << std::endl;
std::cout << readline(input) << std::endl; // grayscale range
fill(0);
if (format == "P5")
{
    input >> std::noskipws;
    unsigned char v;
    for (Iterator it = begin(), itE = end(); it != itE; ++it)
    {
        input >> v;
        *it = v;
    }
}
else
{
    input >> std::skipws;
    int v;
    for (Iterator it = begin(), itE = end(); it != itE; ++it)
    {
        input >> v;
```

```
            *it = v;
        }
    }
    return true;
}

bool GrayLevelImage2D::exportPGM(std::ostream &output, bool ascii)
{
    using namespace std;
    output << "P5" << endl;
    output << m_width << " " << m_height << endl;
    output << "255" << endl;
    // write data
    if (ascii)
    {
        for (int j = 0; j < m_height; ++j)
        {
            for (int i = 0; i < m_width; ++i)
            {
                output << (int)at(i, j) << " ";
            }
            output << endl;
        }
    }
    else
    {
```

```
        for (Iterator it = begin(), itE = end(); it != itE; ++it)
        {
            output << *it;
        }
    }
    return true;
}

void GrayLevelImage2D::medianFilter(int k)
{
    GrayLevelImage2D copy(*this);
    for (int j = 0; j < m_height; ++j)
    {
        for (int i = 0; i < m_width; ++i)
        {
            std::vector<GrayLevel> values;
            for (int y = -k; y <= k; ++y)
            {
                for (int x = -k; x <= k; ++x)
                {
                    if (i + x >= 0 && i + x < m_width && j + y >= 0 && j + y < m_height)
                    {
                        values.push_back(copy.at(i + x, j + y));
                    }
                }
            }
```

```
            std::sort(values.begin(), values.end());
            at(i, j) = values[values.size() / 2];
        }
    }
}

void GrayLevelImage2D::convolation(double coefficient)
{
    GrayLevelImage2D copy(*this);
    for (int j = 0; j < m_width; ++j)
    {
        for (int i = 0; i < m_height; ++i)
        {

            // le if permettent de verifier si on est sur la bordure de l'image ou pas
            double newVal = at(i, j) * (1 + coefficient);
            if (i > 0)
                newVal -= at(i - 1, j) * (coefficient / 4);
            if (i < m_width - 1)
                newVal -= at(i + 1, j) * (coefficient / 4);
            if (j > 0)
                newVal -= at(i, j - 1) * (coefficient / 4);
            if (j < m_height - 1)
                newVal -= at(i, j + 1) * (coefficient / 4);
            copy.at(i, j) = newVal;
        }
```

```
    }

    *this = copy;

}
```

```
#ifndef _GRAYLEVELIMAGE2D_HPP_
#define _GRAYLEVELIMAGE2D_HPP_

#include <iostream>
#include <vector>
#include <exception>

class GrayLevelImage2D
{

public:
  typedef unsigned char GrayLevel;          // le type pour les niveaux de gris.
  typedef std::vector<GrayLevel> Container; // le type pour stocker les niveaux de gris de l'image.

  /**
     Représente un itérateur sur toutes les valeurs d'une image.

     Model of DefaultConstructible, CopyConstructible, Assignable,
     RandomAccessIterator. */
  struct Iterator : public Container::iterator
  {
    Iterator(GrayLevelImage2D &Image, int x, int y);
  };

public:
  GrayLevelImage2D();
```

```
GrayLevelImage2D(int w, int h, GrayLevel g = 0);
void fill(GrayLevel g);

//! [gli2d-sec3]
/// @return la largeur de l'image.
int w() const;
/// @return la hauteur de l'image.
int h() const;

/**
   Accesseur read-only à la valeur d'un pixel.
   @return la valeur du pixel(i,j)
*/
GrayLevel at(int i, int j) const;

/**
   Accesseur read-write à la valeur d'un pixel.
   @return une référence à la valeur du pixel(i,j)
*/
GrayLevel &at(int i, int j);
//! [gli2d-sec3]

Iterator begin();
Iterator end();
Iterator start(int x, int y);
```

```
  std::pair<int, int> position(Iterator it) const;


  bool importPGM(std::istream &input);
  bool exportPGM(std::ostream &output, bool ascii = true);
  bool exportPGM(std::ostream &output, bool ascii);
  void medianFilter(int k);
  void convolation(double coefficient);


private:
  // Calcule l'indice dans m_data du pixel (x,y).
  int index(int x, int y) const;
  // Le tableau contenant les valeurs des pixels.
  Container m_data;
  // la largeur
  int m_width;
  // la hauteur
  int m_height;
};


#endif // #ifndef _GRAYLEVELIMAGE2D_HPP_
```

```
#ifndef _COLOR_HPP_
#define _COLOR_HPP_

/**
   Représente une couleur avec un codage RGB. Ce codage utilise 3
   octets, le premier octet code l'intensité du rouge, le deuxième
   l'intensité du vert, le troisième l'intensité du bleu.
*/
struct Color
{
    typedef unsigned char Byte;

    /// Code les 3 canaux RGB sur 3 octets.
    Byte red, green, blue;

    Color() {}
    /// Crée la couleur spécifiée par (_red,_green,_blue).
    Color(Byte _red, Byte _green, Byte _blue)
        : red(_red), green(_green), blue(_blue) {}

    /// @return l'intensité de rouge (entre 0.0 et 1.0)
    float r() const { return ((float)red) / 255.0; }
    /// @return l'intensité de vert (entre 0.0 et 1.0)
    float g() const { return ((float)green) / 255.0; }
    /// @return l'intensité de bleu (entre 0.0 et 1.0)
    float b() const { return ((float)blue) / 255.0; }
```

```
/// Sert à désigner un canal.
enum Channel
{
    Red,
    Green,
    Blue
};
/// @return le canal le plus intense.
Channel argmax() const
{
    if (red >= green)
        return red >= blue ? Red : Blue;
    else
        return green >= blue ? Green : Blue;
}
/// @return l'intensité maximale des canaux
float max() const { return std::max(std::max(r(), g()), b()); }
/// @return l'intensité minimale des canaux
float min() const { return std::min(std::min(r(), g()), b()); }
/**
   Convertit la couleur RGB en le modèle HSV (TSV en français).
   @param h la teinte de la couleur (entre 0 et 359), hue en anglais.
   @param s la saturation de la couleur (entre 0.0 et 1.0)
   @param v la valeur ou brillance de la couleur (entre 0.0 et 1.0).
*/
```

```
void getHSV(int &h, float &s, float &v) const
{
    // Taking care of hue
    if (max() == min())
        h = 0;
    else
    {
        switch (argmax())
        {
        case Red:
            h = ((int)(60.0 * (g() - b()) / (max() - min()) + 360.0)) % 360;
            break;
        case Green:
            h = ((int)(60.0 * (b() - r()) / (max() - min()) + 120.0));
            break;
        case Blue:
            h = ((int)(60.0 * (r() - g()) / (max() - min()) + 240.0));
            break;
        }
    }
    // Taking care of saturation
    s = max() == 0.0 ? 0.0 : 1.0 - min() / max();
    // Taking care of value
    v = max();
}
/**
```

```
        TODO: Convertit la couleur donnée avec le modèle HSV (TSV en
        français) en une couleur RGB.
    */
    void setHSV(int h, float s, float v)
    {
    }
};

#endif //_COLOR_HPP_
```

```
// file Image2D.hpp

#ifndef _IMAGE2D_HPP_

#define _IMAGE2D_HPP_

#include <vector>


/// Classe générique pour représenter des images 2D.

template <typename TValue>

class Image2D

{

public:

  typedef Image2D<TValue> Self;         // le type de *this

  typedef TValue Value;                 // le type pour la valeur des pixels

  typedef std::vector<Value> Container; // le type pour stocker les valeurs des pixels de l'image.


  // Constructeur par défaut

  Image2D(){

    m_width = 0;

    m_height = 0;

  };

  // Constructeur avec taille w x h. Remplit tout avec la valeur g

  // (par défaut celle donnée par le constructeur par défaut).

  Image2D(int w, int h, Value g = Value())

  {

    m_width = w;

    m_height = h;

    // resize remplit automatiquement de 0
```

```
  m_data.resize(w * h, g);
}

// Remplit l'image avec la valeur \a g.
void fill(Value g)
{
  m_data.assign(m_width * m_height, g);
}

/// @return la largeur de l'image.
int w() const
{
  return m_width;
}
/// @return la hauteur de l'image.
int h() const
{
  return m_height;
}

/// Accesseur read-only à la valeur d'un pixel.
/// @return la valeur du pixel(i,j)
Value at(int i, int j) const
{
  return m_data[i + j * m_width];
}
```

```
/// Accesseur read-write à la valeur d'un pixel.
/// @return une référence à la valeur du pixel(i,j)
Value &at(int i, int j)
{
  return m_data[i + j * m_width];
}
/// Un itérateur (non-constant) simple sur l'image.
struct Iterator : public Container::iterator
{
  Iterator(Self &image, int x, int y)
      : Container::iterator(image.m_data.begin() + image.index(x, y))
  {
  }
};

/// @return un itérateur pointant sur le début de l'image
Iterator begin() { return start(0, 0); }
/// @return un itérateur pointant après la fin de l'image
Iterator end() { return start(0, h()); }
/// @return un itérateur pointant sur le pixel (x,y).
Iterator start(int x, int y) { return Iterator(*this, x, y); }

// ---------------------/
struct ConstIterator : public Container::iterator
{
```

```
  ConstIterator(Self image, int x, int y)
      : Container::iterator(image.m_data.begin() + image.index(x, y))
  {
  }
};

// create cbegin
ConstIterator cbegin() const { return cstart(0, 0); }
// create cend
ConstIterator cend() const { return cstart(0, h()); }
// create cstart
ConstIterator cstart(int x, int y) const { return ConstIterator(*this, x, y); }

template <typename TAccessor>
struct GenericConstIterator : public Container::const_iterator
{
  typedef TAccessor Accessor;
  typedef typename Accessor::Argument ImageValue; // Color ou unsigned char
  typedef typename Accessor::Value Value;         // unsigned char (pour ColorGreenAccessor)
  typedef typename Accessor::Reference Reference; // ColorGreenReference (pour ColorGreenAccessor)
  GenericConstIterator(const Image2D<ImageValue> &image, int x, int y) : Container::const_iterator(image.m_data
  {
  }

  Value operator*() const
  {
```

```
      return Accessor::access(Container::const_iterator::operator*());
    }
  };
  template <typename Accessor>
  GenericConstIterator<Accessor> start(int x = 0, int y = 0) const
  {
    return GenericConstIterator<Accessor>(*this, x, y);
  }
  template <typename Accessor>
  GenericConstIterator<Accessor> begin() const { return start<Accessor>(0, 0); }
  template <typename Accessor>
  GenericConstIterator<Accessor> end() const { return start<Accessor>(0, h()); }
  // Accès en lecture (rvalue)
  //< Appel de op* de l'térateur de vector

private:
  Container m_data; // mes données; évitera de faire les allocations dynamiques
  int m_width;      // ma largeur
  int m_height;     // ma hauteur

  /// @return l'index du pixel (x,y) dans le tableau \red m_data.
  int index(int i, int j) const
  {
    return i + j * m_width;
  }
};
```

```
#endif // _IMAGE2D_HPP_
```

```
#ifndef _IMAGE2DREADER_HPP_
#define _IMAGE2DREADER_HPP_

#include <iostream>
#include <fstream>
#include <ostream>
#include <string>
#include "Color.hpp"
#include "Image2D.hpp"
using namespace std;
template <typename TValue>
class Image2DReader
{
public:
    typedef TValue Value;
    typedef Image2D<Value> Image;

    static bool reader(Image &img, std::ostream &output, bool ascii)
    {
        std::cerr << "[Image2DReader<TValue>::write] NOT IMPLEMENTED." << std::endl;
        return false;
    }
};

/// Specialization for gray-level images.
template <>
```

```
class Image2DReader<unsigned char>
{
public:
    typedef unsigned char Value;
    typedef Image2D<Value> Image;

    static string readline(std::istream &input)
    {
        string str;
        do
        {
            getline(input, str);
        } while (str != "" && str[0] == '#');
        return str;
    }
    static bool reader(std::istream &input, Image &img, bool ascii)
    {
        // read header from PGM file
        if (!input.good())
            return false;
        std::string format = readline(input);

        std::string line = readline(input);
        std::string delim = " ";
        int m_width;
        int m_height;
```

```
m_width = std::stoi(line.substr(0, line.find(delim)));
line.erase(0, line.find(delim) + delim.length());
m_height = std::stoi(line);
std::cout << m_width << " " << m_height << " " << format << std::endl;
// read the line and put it in cout
std::cout << readline(input) << std::endl;
//  new image
img = Image(m_width, m_height, 0);
// img.fill(0);
if (format == "P5")
{
    input >> std::noskipws;
    unsigned char v;
    for (Image::Iterator it = img.begin(), itE = img.end(); it != itE; ++it)
    {
        input >> v;
        *it = v;
    }
}
else
{
    input >> std::skipws;
    int v;
    for (Image::Iterator it = img.begin(), itE = img.end(); it != itE; ++it)
    {
        input >> v;
```

```
                *it = v;
            }
        }
        return true;
    }
};
/// Specialization for color images.

template <>
class Image2DReader<Color>
{
public:
    typedef Color Value;
    typedef Image2D<Value> Image;

    static string readline(std::istream &input)
    {
        string str;
        do
        {
            getline(input, str);
        } while (str != "" && str[0] == '#');
        return str;
    }
    static bool reader(std::istream &input, Image &img, bool ascii)
    {
```

```
// import image
if (!input.good())
    return false;
std::string format = readline(input);

std::string line = readline(input);
std::string delim = " ";
int m_width;
int m_height;
m_width = std::stoi(line.substr(0, line.find(delim)));
line.erase(0, line.find(delim) + delim.length());
m_height = std::stoi(line);
std::cout << m_width << " " << m_height << " " << format << std::endl;
// read the line and put it in cout
std::cout << readline(input) << std::endl;

// new image
img = Image(m_width, m_height, Color(0, 0, 0));
// img.fill(0);
if (format == "P6")
{
    input >> std::noskipws;
    Color v;
    for (Image::Iterator it = img.begin(), itE = img.end(); it != itE; ++it)
    {
        input >> v.red >> v.green >> v.blue;
```

```
            *it = v;
        }
    }
    else
    {
        input >> std::skipws;
        int r, g, b;
        for (Image::Iterator it = img.begin(), itE = img.end(); it != itE; ++it)
        {

            input >> r >> g >> b;
            *it = Color(r, g, b);
        }
    }
    return true;
  }
};

#endif // _IMAGE2DREADER_HPP_
```

```
#ifndef _IMAGE2DWRITER_HPP_
#define _IMAGE2DWRITER_HPP_

#include <iostream>
#include <fstream>
#include <fstream>
#include <ostream>
#include <string>
#include "Color.hpp"
#include "Image2D.hpp"
using namespace std;

template <typename TValue>
class Image2DWriter
{
public:
    typedef TValue Value;
    typedef Image2D<Value> Image;

    static bool write(Image &img, std::ostream &output, bool ascii)
    {
        std::cerr << "[Image2DWriter<TValue>::write] NOT IMPLEMENTED." << std::endl;
        return false;
    }
};
```

```
/// Specialization for gray-level images.
template <>
class Image2DWriter<unsigned char>
{
public:
    typedef unsigned char Value;
    typedef Image2D<Value> Image;

    static bool write(Image &img, std::ostream &output, bool ascii)
    {
        output << (ascii ? "P2" : "P5" ) << endl;
        output << img.w() << " " << img.h() << endl;
        output << "255" << endl;
        // write data
        if (ascii)
        {
            for (int j = 0; j < img.h(); ++j)
            {
                for (int i = 0; i < img.w(); ++i)
                {
                    output << (int)img.at(i, j) << " ";
                }
                output << endl;
            }
        }
        else
```

```
        {
            for (Image::Iterator it = img.begin(), itE = img.end(); it != itE; ++it)
            {
                output << *it;
            }
        }
        return true;
    }
};

/// Specialization for color images.
template <>
class Image2DWriter<Color>
{
public:
    typedef Color Value;
    typedef Image2D<Value> Image;

    static bool write(Image &img, std::ostream &output, bool ascii)
    {

        // Reprenez la partie sauvegarde de l'exemple précédent testColorImage2D.cpp
        output << (ascii ? "P6":"P3") << std::endl; // PPM raw
        output << "# Generated by You !" << std::endl;
        output << img.w() << " " << img.h() << std::endl;
        output << "255" << std::endl;
```

```
        if (ascii)
        {

            for (int j = 0; j < img.h(); ++j)
            {
                for (int i = 0; i < img.w(); ++i)
                {
                    output << (int)img.at(i, j).red << " ";
                    output << (int)img.at(i, j).green << " ";
                    output << (int)img.at(i, j).blue << " ";
                }
                output << endl;
            }
        }
        else
        {
            for (Image::Iterator it = img.begin(), itE = img.end(); it != itE; ++it)
            {
                Color c = *it;
                output << c.red << c.green << c.blue;
            }
        }
        return true;
    }
};
```

```
#endif // _IMAGE2DWRITER_HPP_
```

```
#include <iostream>
#include <ostream>
#include <istream>
#include <fstream>
#include "Color.hpp"
#include "Image2D.hpp"
#include "Image2Dwriter.hpp"
#include "Image2Dreader.hpp"
using namespace std;

int testRedToBlue()
{
    typedef Image2D<Color> ColorImage2D;
    typedef ColorImage2D::Iterator Iterator;

    ColorImage2D img(256, 256, Color(255, 255, 0));
    Iterator it = img.begin();
    for (int y = 0; y < 256; ++y)
        for (int x = 0; x < 256; ++x)
        {
            *it++ = Color(y, x, (5 * x + 2 * y) % 256);
        }

    ifstream input("kowloon.ppm");
    bool ok2 = Image2DReader<Color>::reader(input, img, false);
    if (!ok2)
```

```
    {
        std::cerr << "Error writing reader file." << std::endl;
        return 1;
    }
    for (auto &c : img)
    {
        c = Color(c.blue,c.green,c.red);
    }

    ofstream output("kowloonInverted.ppm");
    bool ok3 = Image2DWriter<Color>::write(img, output, false);
    output.close();
    return 0;
}

int main()
{
    return testRedToBlue();
}
```

```cpp
// save-green-channel.cpp
#include <cmath>
#include <iostream>
#include <fstream>
#include "Image2D.hpp"
#include "Image2Dreader.hpp"
#include "Image2Dwriter.hpp"
#include "accessor.hpp"

int main(int argc, char **argv)
{
    typedef Image2D<Color> ColorImage2D;
    typedef Image2DReader<Color> ColorImage2DReader;
    typedef ColorImage2D::Iterator ColorIterator;
    if (argc < 3)
    {
        std::cerr << "Usage: save-green-channel <input.ppm> <output.pgm>" << std::endl;
        return 0;
    }
    ColorImage2D img;
    std::ifstream input(argv[1]); // récupère le 1er argument.
    bool ok = ColorImage2DReader::reader(input, img, false);
    if (!ok)
    {
        std::cerr << "Error reading input file." << std::endl;
        return 1;
```

```
}
input.close();
typedef Image2D<unsigned char> GrayLevelImage2D;
typedef Image2DWriter<unsigned char> GrayLevelImage2DWriter;
typedef GrayLevelImage2D::Iterator GrayLevelIterator;
GrayLevelImage2D img2(img.w(), img.h());

//-------------------------------------------------------------------------
// vvvvvvvvv Toute la transformation couleur -> canal vert est ici vvvvvvvvvvv
//
// Servira à parcourir la composante verte de l'image couleur.
typedef ColorImage2D::GenericConstIterator<ColorGreenAccessor> ColorGreenConstIterator;
// Notez comment on appelle la méthode \b générique `begin` de `Image2D`.
ColorGreenConstIterator itGreen = img.begin<ColorGreenAccessor>();
// On écrit la composante verte dans l'image en niveaux de gris.
for (GrayLevelIterator it = img2.begin(), itE = img2.end();
     it != itE; ++it)
{
    *it = *itGreen;
    ++itGreen;
    // NB: si on veut faire *itGreen++, il faut redéfinir GenericConstIterator<T>::operator++(int).
}
//-------------------------------------------------------------------------

std::ofstream output(argv[2]); // récupère le 2eme argument.
bool ok2 = GrayLevelImage2DWriter::write(img2, output, false);
```

```
    if (!ok2)
    {
        std::cerr << "Error writing output file." << std::endl;
        return 1;
    }
    output.close();
    return 0;
}
```

```
#ifndef _ACCESSOR_HPP_
#define _ACCESSOR_HPP_

#include "Color.hpp"
/// Accesseur trivial générique
template <typename TValue>
struct TrivialAccessor
{
    typedef TValue Value;
    typedef Value Argument;
    typedef Value &Reference;
    // Acces en lecture.
    static Value access(const Argument &arg)
    {
        return arg;
    }
    // Acces en écriture.
    static Reference access(Argument &arg)
    {
        return arg;
    }
};

// Accesseur trivial pour une image en niveaux de gris
typedef TrivialAccessor<unsigned char> GrayLevelTrivialAccessor;
// Accesseur trivial pour une image en couleur.
```

```
typedef TrivialAccessor<Color> ColorTrivialAccessor;


/// Accesseur à la composante verte.
struct ColorGreenAccessor
{
    typedef unsigned char Value;
    typedef Color Argument;

    /// Même astuce que pour les références à un bit dans un tableau de bool.
    struct ColorGreenReference
    {
        Argument &arg;
        ColorGreenReference(Argument &someArg) : arg(someArg) {}

        // Accesseur lvalue (écriture)
        // permet d'écrire *it = 120 pour changer l'intensité du vert
        ColorGreenReference &operator=(Value val)
        {
            arg.green = val;
            return *this;
        }

        // Accesseur rvalue (lecture)
        // permet d'écrire *it pour récupérer l'intensité du vert
        operator Value() const
        {
```

```
            return arg.green; // arg.green est de type Value.
        }
    };
    typedef ColorGreenReference Reference;

    // Acces en lecture.
    static Value access(const Argument &arg)
    {
        return arg.green;
    }
    // Acces en écriture.
    static Reference access(Argument &arg)
    {
        return ColorGreenReference(arg);
    }
};

struct ColorRedAccessor
{
    typedef unsigned char Value;
    typedef Color Argument;

    /// Même astuce que pour les références à un bit dans un tableau de bool.
    struct ColorRedReference
    {
        Argument &arg;
```

```
    ColorRedReference(Argument &someArg) : arg(someArg) {}

    // Accesseur lvalue (écriture)
    // permet d'écrire *it = 120 pour changer l'intensité du vert
    ColorRedReference &operator=(Value val)
    {
        arg.red = val;
        return *this;
    }

    // Accesseur rvalue (lecture)
    // permet d'écrire *it pour récupérer l'intensité du vert
    operator Value() const
    {
        return arg.red; // arg.green est de type Value.
    }
};
typedef ColorRedReference Reference;

// Acces en lecture.
static Value access(const Argument &arg)
{
    return arg.red;
}
// Acces en écriture.
static Reference access(Argument &arg)
```

```
    {
        return ColorRedReference(arg);
    }
};

struct ColorBlueAccessor
{
    typedef unsigned char Value;
    typedef Color Argument;

    /// Même astuce que pour les références à un bit dans un tableau de bool.
    struct ColorBlueReference
    {
        Argument &arg;
        ColorBlueReference(Argument &someArg) : arg(someArg) {}

        // Accesseur lvalue (écriture)
        // permet d'écrire *it = 120 pour changer l'intensité du vert
        ColorBlueReference &operator=(Value val)
        {
            arg.blue = val;
            return *this;
        }

        // Accesseur rvalue (lecture)
        // permet d'écrire *it pour récupérer l'intensité du vert
```

```
        operator Value() const
        {
            return arg.blue; // arg.green est de type Value.
        }
    };
    typedef ColorBlueReference Reference;

    // Acces en lecture.
    static Value access(const Argument &arg)
    {
        return arg.blue;
    }
    // Acces en écriture.
    static Reference access(Argument &arg)
    {
        return ColorBlueReference(arg);
    }
};

#endif // _ACCESSOR_HPP_
```

```
#include <iostream>
#include <ostream>
#include <istream>
#include <fstream>
#include "Color.hpp"
#include "Image2D.hpp"
#include "Image2Dwriter.hpp"
#include "Image2Dreader.hpp"
using namespace std;

//works
int testProfQ4()
{
    typedef Image2D<Color> ColorImage2D;
    typedef ColorImage2D::Iterator Iterator;
    ColorImage2D img(256, 256, Color(0, 0, 0));
    Iterator it = img.begin();
    for (int y = 0; y < 256; ++y)
        for (int x = 0; x < 256; ++x)
        {
            *it++ = Color(y, x, (2 * x + 2 * y) % 256);
        }
    std::ofstream output("colors.ppm", ios::binary); // ios::binary for Windows system
    output << "P6" << std::endl;                    // PPM raw
    output << "# Generated by You !" << std::endl;
    output << img.w() << " " << img.h() << std::endl;
```

```
    output << "255" << std::endl;
    for (Iterator it = img.begin(), itE = img.end(); it != itE; ++it) // (*)
    {
        Color c = *it;
        output << c.red << c.green << c.blue;
    }
    output.close();
    return 0;
}

//works
int constTestProfQ4()
{
    typedef Image2D<Color> ColorImage2D;
    typedef ColorImage2D::Iterator Iterator;
    typedef ColorImage2D::ConstIterator ConstIterator;

    ColorImage2D img(256, 256, Color(0, 0, 0));
    Iterator it = img.begin();
    for (int y = 0; y < 256; ++y)
        for (int x = 0; x < 256; ++x)
        {
            *it++ = Color(y, x, (2 * x + 2 * y) % 256);
        }
    std::ofstream output("colorsWithConst.ppm", ios::binary); // ios::binary for Windows system
    output << "P6" << std::endl;                              // PPM raw
```

```
    output << "# Generated by You !" << std::endl;
    output << img.w() << " " << img.h() << std::endl;
    output << "255" << std::endl;

    const ColorImage2D &cimg = img;                                    // Vue "constante" sur l'image img.
    for (ConstIterator it = cimg.cbegin(), itE = cimg.cend(); it != itE; ++it) // (*)
    {
        Color c = *it;
        output << c.red << c.green << c.blue;
    }
    output.close();
    return 0;
}

//works
void oldMain()
{
    typedef Image2D<Color> ColorImage2D;
    ColorImage2D img(8, 8, Color(255, 0, 255));
    for (int y = 0; y < img.h(); ++y)
    {
        for (int x = 0; x < img.w(); ++x)
            std::cout << " " << (int)img.at(x, y).red << "/" << (int)img.at(x, y).green << "/" << (int)img.at(x, y).blue;
        std::cout << std::endl;
    }
}
```

```
//works
int testProfWriterQ5()
{
    typedef Image2D<Color> ColorImage2D;
    typedef ColorImage2D::Iterator Iterator;

    ColorImage2D img(256, 256, Color(0, 0, 0));
    Iterator it = img.begin();
    for (int y = 0; y < 256; ++y)
        for (int x = 0; x < 256; ++x)
        {
            *it++ = Color(y, x, (2 * x + 2 * y) % 256);
        }
    ofstream output("colorsQ5Output.ppm");
    bool ok2 = Image2DWriter<Color>::write(img, output, false);
    if (!ok2)
    {
        std::cerr << "Error writing output file." << std::endl;
        return 1;
    }
    output.close();
    return 0;
}

//works
```

```
int testProfReaderQ5()
{
    typedef Image2D<Color> ColorImage2D;
    typedef ColorImage2D::Iterator Iterator;

    ColorImage2D img(256, 256, Color(255, 255, 0));
    Iterator it = img.begin();
    for (int y = 0; y < 256; ++y)
        for (int x = 0; x < 256; ++x)
        {
            *it++ = Color(y, x, (5 * x + 2 * y) % 256);
        }

    ifstream input("colorsQ5Output.ppm");
    bool ok2 = Image2DReader<Color>::reader(input, img, false);
    if (!ok2)
    {
        std::cerr << "Error writing reader file." << std::endl;
        return 1;
    }

    ofstream output("colorsQ5WRITERBIS.ppm");
    bool ok3 = Image2DWriter<Color>::write(img, output, false);
    output.close();
    return 0;
```

```
}


int main(int argc, char **argv)
{
    return 0;
}


// copilot peut-tu ecrire un poeme pour mon ami yuss le plus bel homme de la terre
// O Yuss, O Yuss, tu es le plus bel homme de la terre
// Car tu es le plus bel homme de la terre
// O Yuss, O Yuss, tu es le plus bel homme de la terre
// Avec tes yeux bleus, tes cheveux blonds, tes lèvres rouges


// co authored COPILOT LE S
```

```
#include <cmath>
#include <iostream>
#include <fstream>
#include "Image2D.hpp"
#include "Image2Dreader.hpp"
#include "Image2Dwriter.hpp"
#include "accessor.hpp"

using namespace std;

int main(int argc, char **argv)
{
    typedef Image2D<Color> ColorImage2D;
    typedef Image2DReader<Color> ColorImage2DReader;
    typedef ColorImage2D::Iterator ColorIterator;
    if (argc < 2)
    {
        std::cerr << "Usage: save-channel <input.ppm>" << std::endl;
        return 0;
    }

    ColorImage2D img;
    std::ifstream input(argv[1]); // récupère le 1er argument.
    bool ok = ColorImage2DReader::reader(input, img, false);
    if (!ok)
    {
```

```
    std::cerr << "Error reading input file." << std::endl;
    return 1;
}
input.close();
typedef Image2D<unsigned char> GrayLevelImage2D;
typedef Image2DWriter<unsigned char> GrayLevelImage2DWriter;
typedef GrayLevelImage2D::Iterator GrayLevelIterator;

GrayLevelImage2D img2(img.w(), img.h());

typedef ColorImage2D::GenericConstIterator<ColorGreenAccessor> ColorGreenConstIterator;
ColorGreenConstIterator itGreen = img.begin<ColorGreenAccessor>();

for (GrayLevelIterator it = img2.begin(), itE = img2.end();
     it != itE; ++it)
{
    *it = *itGreen;
    ++itGreen;
}

ofstream output("inputGreen.ppm"); // récupère le 2eme argument.
bool ok2 = GrayLevelImage2DWriter::write(img2, output, false);
if (!ok2)
{
    std::cerr << "Error writing output file." << std::endl;
    return 1;
```

```
}
output.close();

//---------------------------
GrayLevelImage2D img3(img.w(), img.h());

typedef ColorImage2D::GenericConstIterator<ColorBlueAccessor> ColorBlueConstIterator;
ColorBlueConstIterator itBlue = img.begin<ColorBlueAccessor>();

for (GrayLevelIterator it = img3.begin(), itE = img3.end();
     it != itE; ++it)
{
    *it = *itBlue;
    ++itBlue;
}

ofstream outputblue("inputBlue.ppm"); // récupère le 2eme argument.
bool ok3 = GrayLevelImage2DWriter::write(img3, outputblue, false);
if (!ok3)
{
    std::cerr << "Error writing output file." << std::endl;
    return 1;
}
outputblue.close();

//---------------------------
```

```
    GrayLevelImage2D img4(img.w(), img.h());


    typedef ColorImage2D::GenericConstIterator<ColorRedAccessor> ColorRedConstIterator;
    ColorRedConstIterator itRed = img.begin<ColorRedAccessor>();


    for (GrayLevelIterator it = img4.begin(), itE = img4.end();
         it != itE; ++it)
    {
        *it = *itRed;
        ++itRed;
    }


    ofstream outputred("inputRed.ppm"); // récupère le 2eme argument.
    bool ok4 = GrayLevelImage2DWriter::write(img4, outputred, false);
    if (!ok4)
    {
        std::cerr << "Error writing output file." << std::endl;
        return 1;
    }
    outputred.close();


    return 0;
}
```

```
#include <iostream>
#include <fstream>
#include "Image2D.hpp"

using namespace std;
// ça a make la
int main(int argc, char **argv)
{
  typedef unsigned char GrayLevel;
  typedef Image2D<GrayLevel> GrayLevelImage2D;
  GrayLevelImage2D img(8, 8, 5); // imagette 8x8 remplie de 5
  for (int y = 0; y < img.h(); ++y)
  {
    for (int x = 0; x < img.w(); ++x)
      std::cout << " " << (int)img.at(x, y); // la conversion permet de voir les caractères sous forme d'entiers.
    std::cout << std::endl;
  }
  return 0;
}
```