POLYTECH
NICE-SOPHIA

Lab Mandelbrot Set

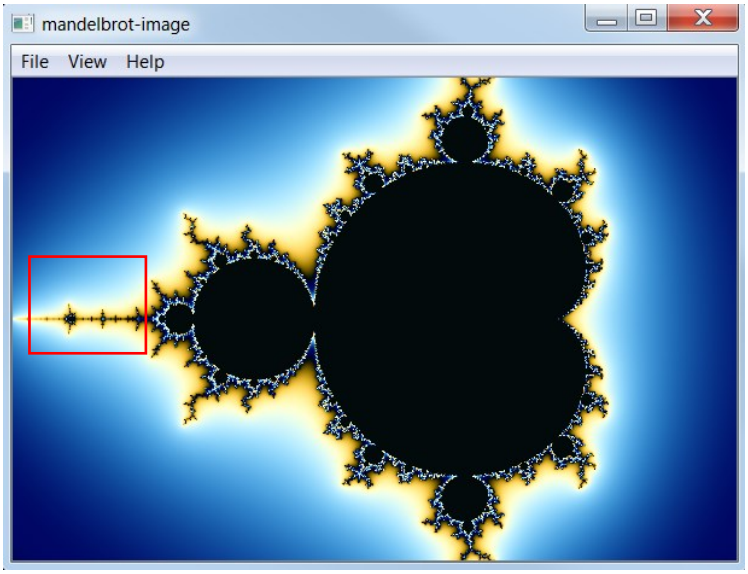
The goal of this lab is to draw an image of the Mandelbrot set and, in the process, learn new C++ language features such as friend functions, STL complex numbers and multi-threaded programming.

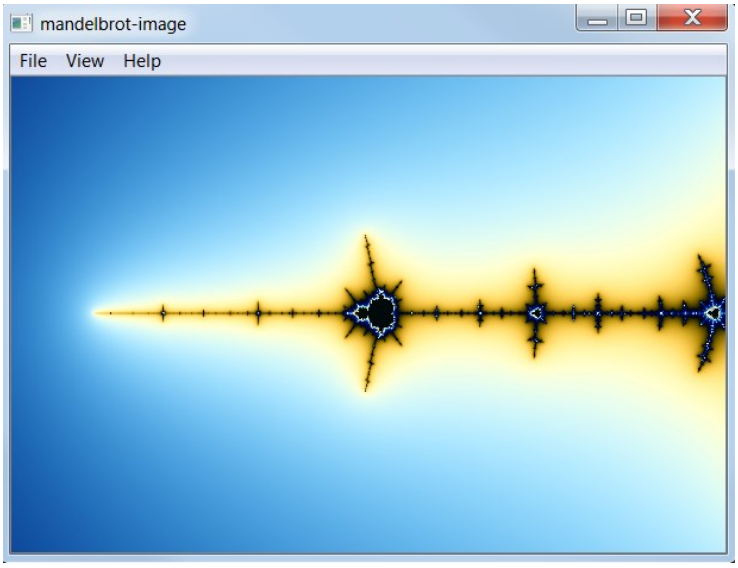
1 - Mandelbrot Set

1.1 Introduction

To read: http://en.m.wikipedia.org/wiki/Mandelbrot_set
To see: http://vimeo.com/12185093

Your goal is to draw an image which looks like these 2 examples below:



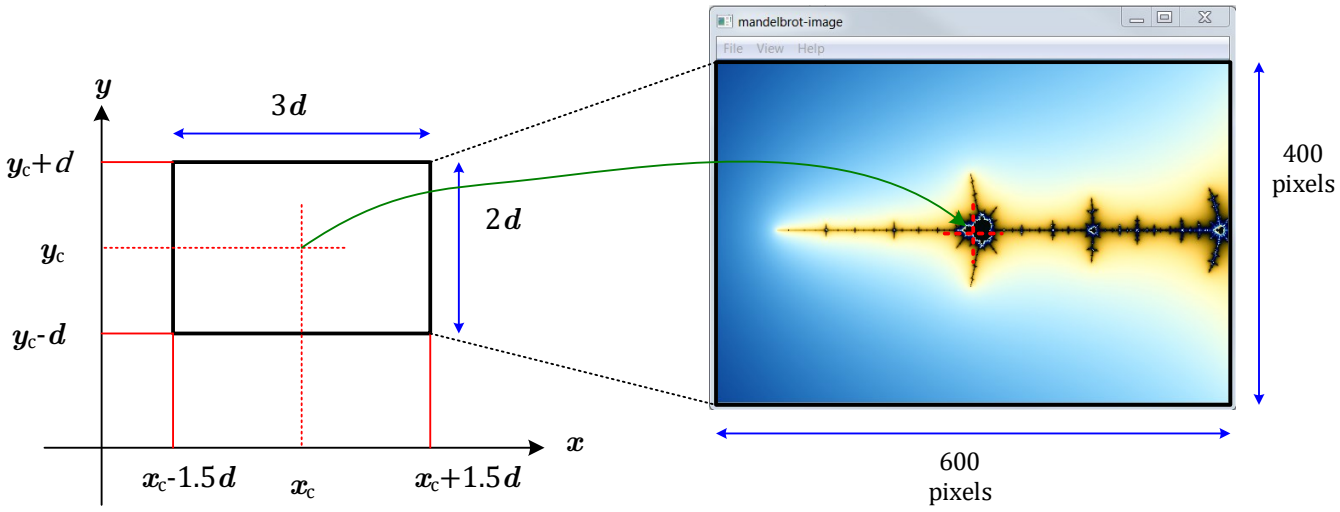


The bottom image is a zoom of the top image focusing on the red rectangle.

Several variables control this drawing:

- x_c, y_c the center of the rectangle,
- d the half-height of the rectangle,
- The rectangle half-length is always $1.5d$,
- The width and height of the window measured in pixels: w and h.

The diagram below can help you understand better the relationship between real coordinates and image coordinates.



Real World Image World

Simple linear interpolations can be used to translate coordinates from "Real World" domain to "Image World" domain and vice-versa. You will have to code these linear transforms.

To draw the Mandelbrot set, you will have loop over each pixel of the image world and obtain its coordinate in the real world. We assume that the coordinate of a given pixel is represented by the following values in the real world: (x_0, y_0).

It is always true that:

$$\begin{cases} x_c - 1.5d \leq x_0 \leq x_c + 1.5d \\ y_c - d \leq y_0 \leq y_c + d \end{cases}$$

We introduce the complex number $c_0 = x_0 + iy_0$, and the complex series: $\{z_n\}$ such that

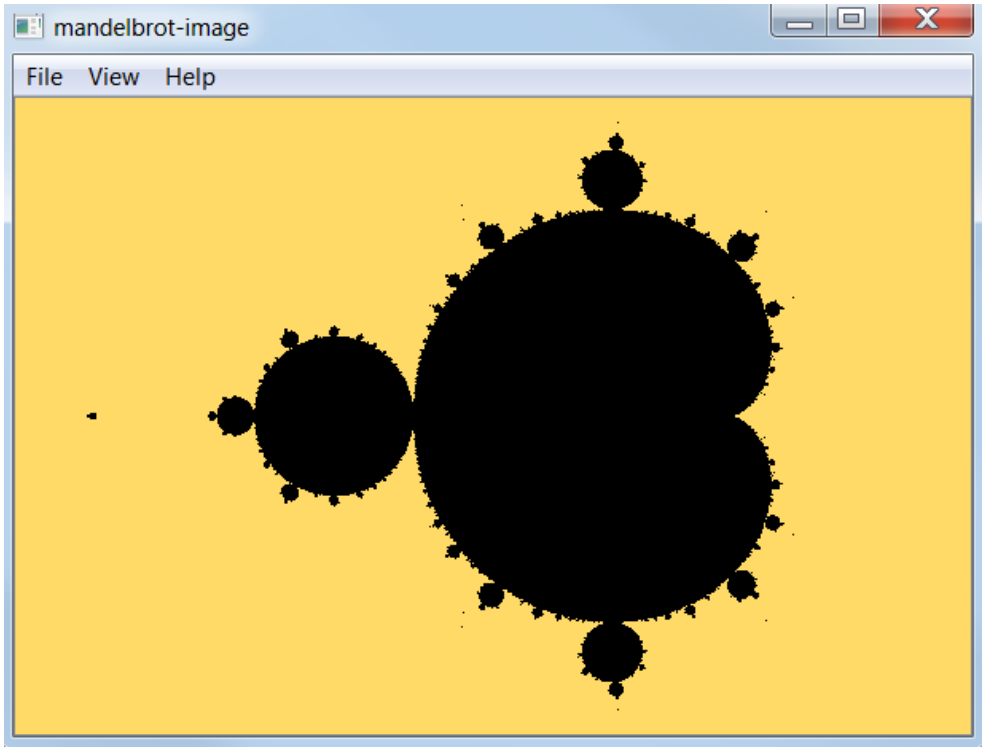
$z_{n+1} = z_n^2 + c_0$, with $z_0 = 0$.

When $|z_n| \geq 2$, the point c_0 is outside the Mandelbrot set and when $|z_n| < 2$ for all values of n, the point c_0 belongs to the Mandelbrot set.
To limit the compute time, we define a maximum number of iterations n_{MAX}, so if $|z_{n_{MAX}}| < 2$, we consider that the point c_0 is within the set.

For this lab, you shall use $n_{\text{MAX}} = 512$.

1.2 First Step: A Two-Color Image

For this first step, your goal is to obtain a two-color image as shown below within a window of size 600x400.



To get started, reuse your code from the triangle image, and add a new class `MandelbrotImage` derived from `QImage`. Check the appendix (Appendix: Creating a Derived Class) for the step-by-step method.

Use the complex number library from the STL `std::complex<double>`, check the link below for more information.
http://en.cppreference.com/w/cpp/numeric/complex.

For this image, the parameters are:

$$\begin{cases} x_c = -1/2 \\ y_c = 0 \\ d = 1 \end{cases}$$

The points within the set are colored in black, the points outside the set are colored in yellow RGB = (255, 218, 103).

To help you, I have pasted the following from my code base:

```
...
// iterate on each point of the image
for (int yp = 0; yp < height; ++yp) {
  // convert vertical pixel domain into real y domain
  double y0 = v_pixel2rect(yp);
  for (int xp = 0; xp < width; ++xp) {
    // convert horizontal pixel domain into real x domain
    double x0 = h_pixel2rect(xp);
    std::complex<double> c0(x0, y0);
    ...
```

The purple functions are function objects described in Appendix: Function Object.

POLYTECH
NICE-SOPHIA

2 - Colored Mandelbrot Set

To read information on how points outside the Mandelbrot set can be assigned to a given color:
https://en.wikipedia.org/wiki/Plotting_algorithms_for_the_Mandelbrot_set#Continuous_(smooth)_coloring

2.1 Color Palette

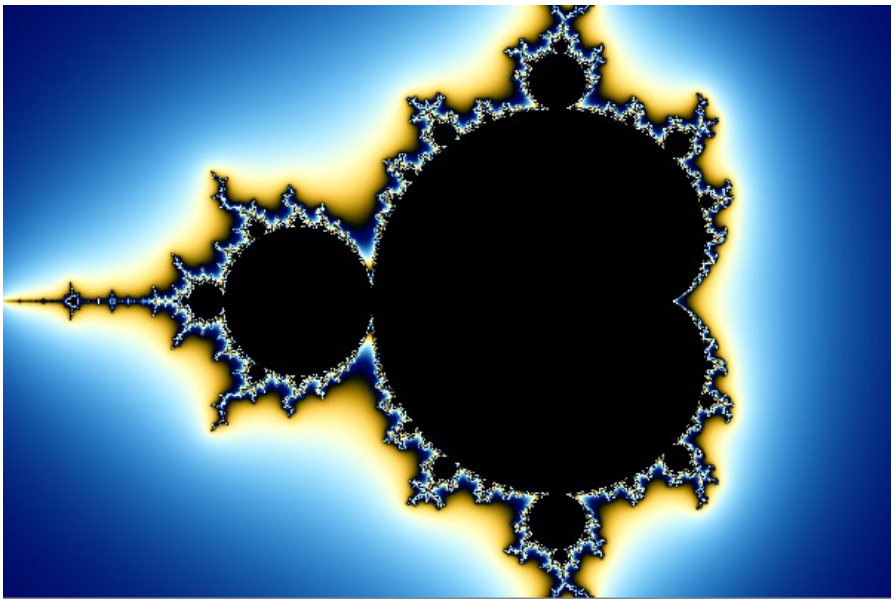
Reusing the code of Lab2, you will have to build a vector of 2048 colors. The details on how to build this color palette are found in the appendix (Appendix: RGB Colors).

2.2 Link between Points and Color

I used a simpler version than the Wikipedia algorithm to compute the color of a point.
Let n, be the number of iterations to have $|z_n| \geq 256$, exiting the loop if $n \geq 512$.

$$v = \log_2(\log_2(|z_n|^2))$$
$$i = 1024 \cdot \sqrt{(n + 5 - v)}$$

The value i is your color index, a modulo 2048 operation is needed to ensure that $0 \leq i < 2048$,
After updating your code base, you will obtain an image like this one:



Note the elapsed time to compute this image (see below) on how to

2.3 Measuring Time

We need to measure the time taken by your program to draw an image. You will have to use the STL library which offers several classes and methods.
Read:
http://en.cppreference.com/w/cpp/chrono

You can use the example given in the link and make the necessary updates to report a time duration in microseconds instead of seconds, so don't use `std::chrono::system_clock`.

As an example, I obtained the following results to compute the first image on page 1:

```
INFO: image calculed in 57166us
```

2.4 Friend functions

To facilitate reading of the duration, we propose to add coma separators for thousands, the displayed result looks like:

```
INFO: image calculed in 57,166us
```

You have many possible ways to solve this little problem.
The following line is found in my code:

```
std::cout << "Info: image calculated in " << ELEC4::Commify(my_value) << "us";
```

Here `Commify()` is not a function but a constructor which returns a `Commify` object. This object is then printed using the `operator<<`.

To help you further, I have given you a few lines of my `Commify` class:

```
class Commify {
private:
  ...
public:
  explicit Commify(int64_t value) {
    ...
  }
  friend std::ostream& operator<<(std::ostream &os, const Commify &c) {
    ...
    return os;
  }
};
```

You must complete this class.
Hint: http://en.cppreference.com/w/cpp/io/basic_ostringstream/str

3 - Improving Performances

3.1 Know Your Hardware

You have to find details on your hardware, CPU, number of core and supported instruction sets, tools such as CPUZ can help you:
Example with my laptop:

Processor Name	*Intel Core i7 4800MQ*
Number of Cores	*4*
Number of Threads	*8*
Base Frequency	*2.7GHz*
Extra SIMM Instructions	*SSE 4.1 / 4.2, AVX 2.0*

The number of threads is a key information, we shall use it to build a parallel version of the image rendering to improve performance.

3.2 Using « *threads* » for Parallel Computation

To read: https://solarianprogrammer.com/2011/12/16/cpp-11-thread-tutorial.

You can waste lot of time and energy to optimize your code, so you have to ensure that you are optimizing the right part of your code. Gathering statistics is an important first step to narrow down the high contributing functions to the total run time. See Amdahl law for details on the subject.

Here, we are taking the hypothesis that the double loop which computes the color of pixel is indeed the highest time contributor:

```
        // iterate on each point of the image
        for (int yp = 0; yp < height; ++yp) {
          // convert vertical pixel domain into real y domain
          double y0 = v_pixel2rect(yp);
          for (int xp = 0; xp < width; ++xp) {
            // convert horizontal pixel domain into real x domain
            double x0 = h_pixel2rect(xp);
            ...
          }
        }
```

Q The double loop we have done so far have always been with the outer loop on each line and the inner loop on each pixel of the line, but the result would be equivalent if you try to swap these two loops. Can you justify why our first choice is far better?

You will have to create a new method:

```
process_sub_image(int current_thread, int max_thread)
```

which will compute only a partial image, with `yp` iterating from row `yi` to row `yj`, details on how to compute `yi` and `yp` from `current_thread` and `max_thread` is up to you.

```
...
for(int i= 0; i < max_threads; i++) {
   process_sub_image(i, max_threads);
}
```

Once you have done this simple transformation, your code must continue to execute normally. Try changing the `max_threads` variable from 1 to 16 for example and verify that you code returns the same image and that the performance hasn't changed.

3.2.1 Adding Threads

You are now ready to activate the multi-threaded version. The sequence used so far is important: (a) we have analyzed the highest time contributor, (b) found a way to parallelize the algorithm, (c) updated the code in mono-thread to allow the parallelism in multi-thread. (d) we are ready to activate the multi-thread version. Such sequence will allow to revert to single thread when you need to debug part of the program.

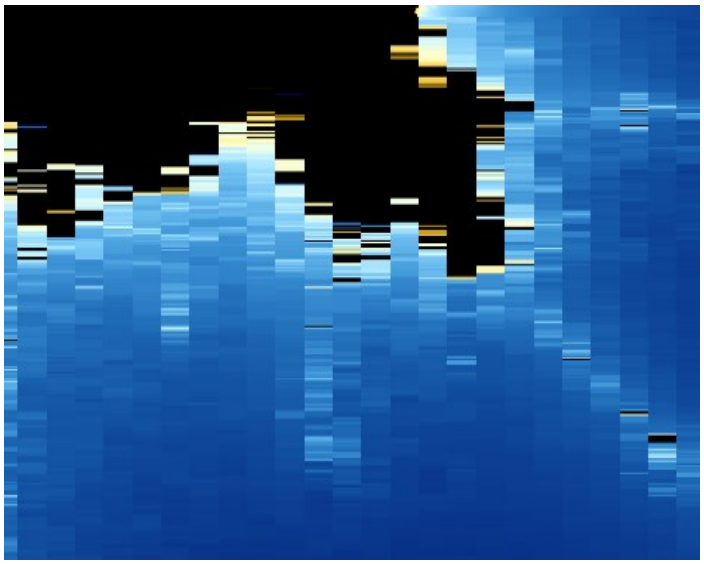
Since C++11, multi-threads programs are very easy to write, you can follow my example:

```
std::vector<std::thread> threads;
int max_threads = 4;
for(...) {
  threads.emplace_back([=]() {
    process_sub_image(i, max_threads);
  });
}
...
for (auto &thread_elem : threads) {
  thread_elem.join();
}
```

Plot a simple graph of the elapsed time as a function of the number of threads, comment your results.

3.3 Image Quality

When the zoom factor (d) is very small (10^{-15}), you can observe an image like the one below, with very poor quality:



Q Explain why the image quality is so low?

POLYTECH
NICE-SOPHIA

3.4 Lab Report

This lab will be graded, so prepare the following

An archive created with `tar` linux utility containing the following files:

```
you Qt project

report.pdf
```

The archive name must be `lab5_NAME1_NAME2_NAME3.tar.gz`
Name1, Name2 and Name3 are the names of the people in the group.
To create the archive:
`tar czf lab5_NAME1_NAME2_NAME3.tar.gz qt_folder report.pdf`

The `report.pdf` document must

- Demonstrate execution of your program (with screen capture)
- Provide details on how you have coded section 2 and section 3
- Answer all the questions found in this lab document
- Show the curve elapsed time vs thread

I will un-archive the deliverable and compile your files using qt Creator

- No credit will be given if the archive is not created with tar
- No credit will be given if the report is not a pdf file
- No credit will be given if your files cannot be compiled by qt Creator

4 - Extra Credit

4.1 User Interactions

With some minor modification in your code, you can make sure that image parameters:

$$\begin{cases} x_c \\ y_c \\ d \end{cases}$$

are passed as parameters to the image constructor.

In `MainWindow`, you can add a virtual `keyPressEvent` method (check here for a good example). This method is called when a key of the keyboard is pressed, you can detect

- A move up, down, left or right to implement a "pan" function on the image.
- A "+" or "-" to implement a zoom in or zoom out feature on the image.

You simply need to update x_c, y_c, d based on the key pressed and display the updated image.

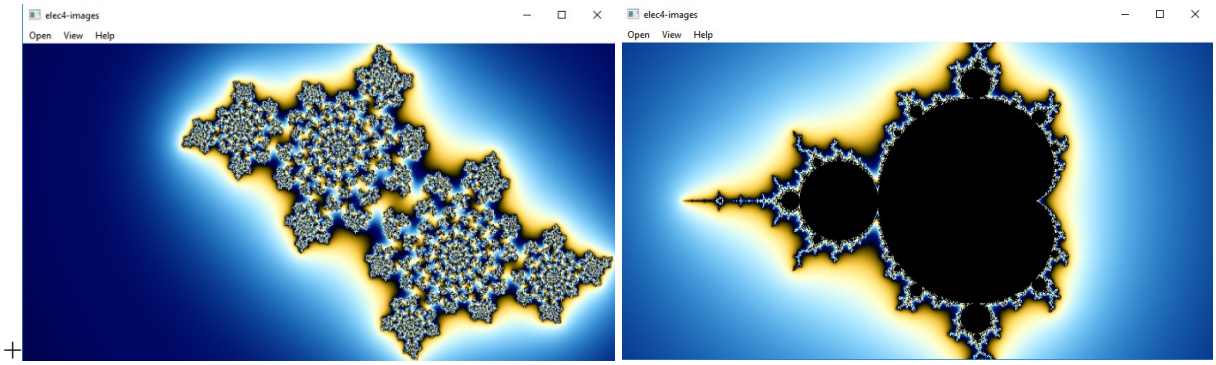
4.2 Julia Set

Add another parameter to the image constructor to change the fractal type and draw a Julia set based on the equation below. Add a case statement in the `keyPress` method to associate with a key "t" (toggle), to switch from Mandelbrot set to Julia set.

Julia Set Equation

$z_{n+1} = z_n^2 + c_0$, with $z_0 = (x + iy),\ c0 = -0.4 + 0.6i$

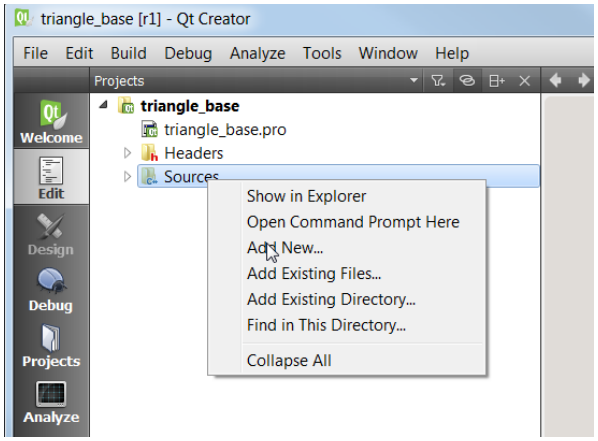
Julia and Mandelbrot sets side-by-side:



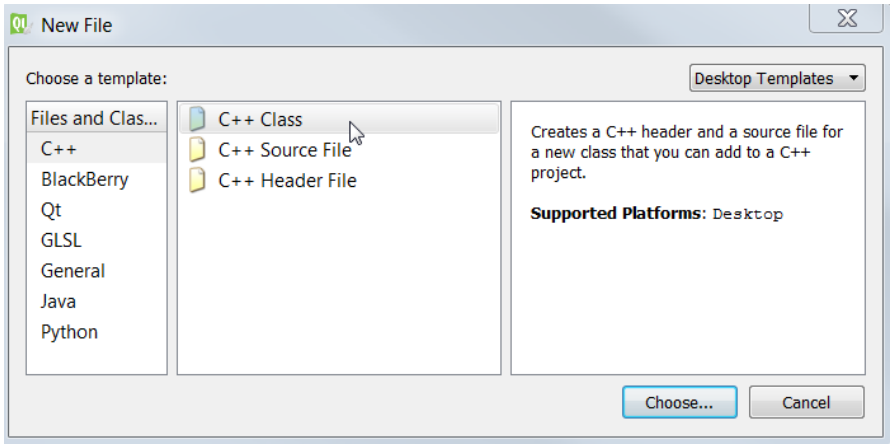
5 - Appendix: Creating a Derived Class

5.1 .h and .cpp File Creation

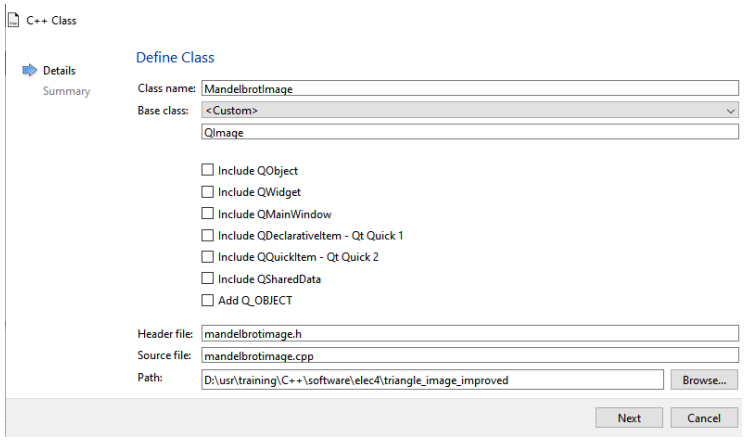
Follow these steps:
1) Click on the right button on the “Sources” menu, then select “Add New...”



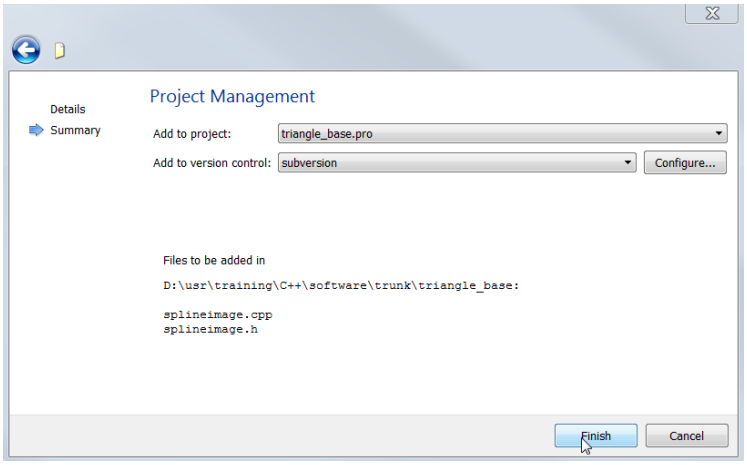
2) Select “C++ Class” and click on “Choose...”



3) Fill in the fields “Class Name” and “Base Class” then click on “Next”.



4) The last panel appears, click on “Finish”.



5.2 Update of MainWindow

For the original triangle source code, try refactoring the `slot_load_triangle_image()` method to call the constructor of Mandelbrot image.

```
void MainWindow::slot_load_mandelbrot_image() {

  MandelbrotImage mandelbrot_image(mandelbrot_width_, mandelbrot_height_);

  image_widget_->setPixmap(QPixmap::fromImage(mandelbrot_image));
  image_widget_->setFixedSize(mandelbrot_width_, mandelbrot_height_);
  adjustSize();
}
```

POLYTECH
NICE-SOPHIA

6 - Appendix: Function Object

Reading: https://en.wikipedia.org/wiki/Function_object#In_C_and_C++

You have certainly written small function for the linear interpolation to convert the real world values to pixel coordinates.

Using function object, I was able to write code like this:

```
double x_min = -0.05;
double x_max =  1.05;
ELEC4::LinearInterpolate h_interpolator(0.0, static_cast<double>(width - 1),
                                        x_min, x_max);

for (int xp = 0; xp < width; ++xp) {
  double  x = h_interpolator(xp);
  …
```

Try to do the same.
To help you, in my class `LinearInterpolate`,
function object as shown:

```
// example of a function object class
class LinearInterpolate {
private:
  double a_;
  double b_;...

public:
  LinearInterpolate(double x0, double x1, double y0, double y1) ...
{
...
}
  double operator()(double x) {
    ...
    return a_ * x + b_;
  }
};
```

A class with only one method "`operator()`" is a "function object", also known as "Functor"

POLYTECH
NICE-SOPHIA

7 - Appendix: RGB Colors

To compute the 2048 colors, you will have to re-use the code of Lab2 either with spline or linear interpolation.

The color points are given by:

```
vector<double> xs{   0.,   0.16,   0.42,   0.6425,  0.8575};

vector<double> yr{   0.,  32.  , 237.  , 215.    ,  0.    };
vector<double> yg{   7., 107.  , 255.  , 170.    , 10.    };
vector<double> yb{ 100., 183.  , 235.  ,  40.    , 15.    };
```

You will need to iterate over a vector of 2048 RGB value.

For the color at index i, the value x is given by:

$x = i/2048$

Then compute the R, G, and B values using `get_value(x);`

Beware that value returned by `get_value(x)` is a floating-point value of type `double`, which must be converted to integer (please use `static_cast`).
Also remember that each color value must be withing the range {0, ..., 255}, so you will have to clamp if the calculated value is outside the range.