Object-Oriented Programming in C++ (SS 2026) Exercise 1: April 2, 2026

Wolfgang Schreiner
Research Institute for Symbolic Computation (RISC)
Wolfgang.Schreiner@risc.jku.at

March 11, 2026

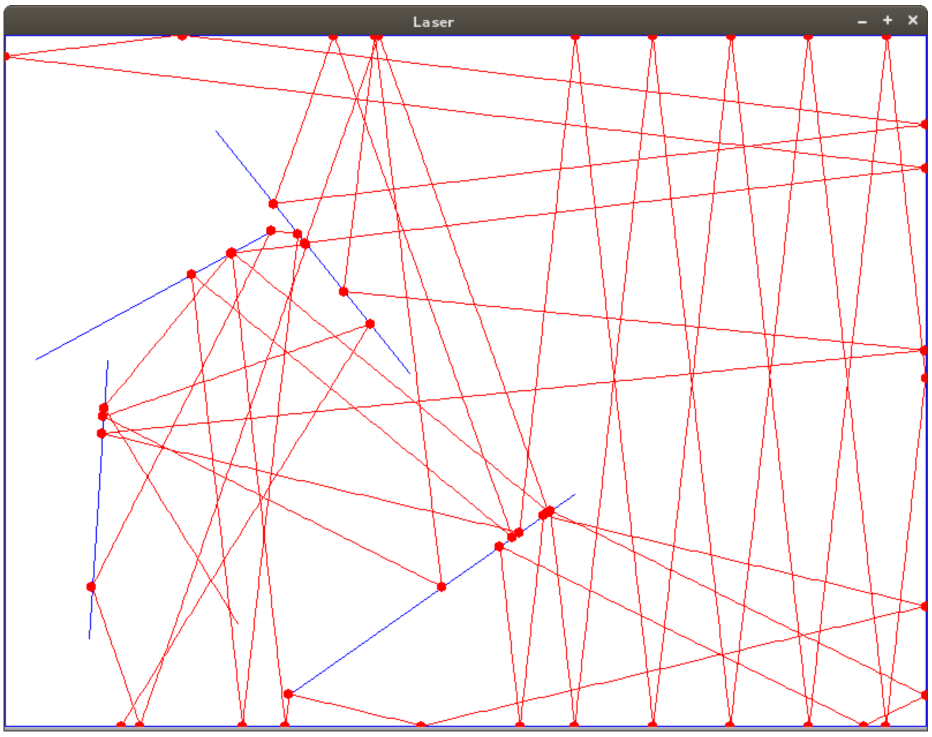
The exercise is to be submitted by the denoted deadline via the submission interface of the Moodle course as a single file in zip (`.zip`) or tarred gzip (`.tgz`) format which contains the following files:

- A PDF file `Exercise`*`Number-MatNr`*`.pdf` (where *Number* is the number of the exercise and *MatNr* is your "Matrikelnummer") which consists of the following parts:
  1. A decent cover page with the title of the course, the number of the exercise, and the author of the solution (identified by name, Matrikelnummer and email address).
  2. For every source file, a listing in a *fixed width font*, e.g. `Courier`, (such that indentations are appropriately preserved) and an appropriate *font size* such that source code lines do not break.
  3. A description of all tests performed (copies of program inputs and program outputs) explicitly highlighting, if some test produces an unexpected result.
  4. Any additional explanation you would like to give. In particular, if your solution has unwanted problems or bugs, please document these explicitly (you will get more credit for such solutions).
- Each source file of your solution (no object files or executables).

Please obey the coding style recommendations posted on the course site.

Exercise 1: A Laser in a Room of Mirrors

Write a program LaserRay that animates the path of a “laser ray” which bounces in a room of reflecting walls and other mirrors:



For this purpose, introduce a (structure/class) type Segment whose values represent line segments by the coordinates of their endpoints (as double values). The main program shall then execute the piece of code shown on the next page.

This program first creates a window of size $W \times H$ with white background (above screenshot is for $W = 800$ and $H = 600$, use these values also in your program[1]). It also initializes the ray which is represented by a line segment whose first point denotes the current position of the ray; the direction of the ray is represented by the vector from the first point to the second one (the length of this vector can be arbitrary). Finally the program determines the number n of mirrors that are represented by their end points and allocates and initializes a corresponding array of length n.

The program then updates in T iterations the state of the ray by setting its second point to the point where the ray hits some mirror; it also determines the new state of the ray after the reflection. The program then animates the movement of the ray from its first point to its second point, and then updates the state of the ray by the newly computed state. When all iterations have been performed, the allocated array is deallocated. When the user closes the window, the program terminates.

Your task is to implement the functions init, drawMirrors, drawRay and reflectRay[2]:

[1] Whenever numerical constants are needed, introduce named constants and assign them suitable values.
[2] Please note that init and reflectRay have output parameters that are to be represented as references.

```
#include <iostream>
#include <cstdlib>
#include <cmath>

#include "Drawing.h"

using namespace std;
using namespace compsys;

...

int main(int argc, const char* argv[])
{
  beginDrawing(W, H, "Laser", 0xFFFFFF);

  Segment ray; int n; Segment *mirrors;
  init(argc, argv, ray, n, mirrors);
  drawMirrors(n, mirrors);
  for (int i=0; i<T; i++)
  {
    Segment ray0;
    reflectRay(n, mirrors, ray, ray0);
    drawRay(ray);
    ray = ray0;
  }
  delete[] mirrors;

  cout << "Close window to exit..." << endl;
  endDrawing();
}
```

- If the program is called without command line arguments (i.e., `argc==1`), `init` creates an initial state of the ray with random end points. The program then creates $N + 4$ mirrors; the first four mirrors represent the walls of the room (using the coordinates 0, $W - 1$, $H - 1$); the remaining N mirrors are generated at random positions (mirrors may intersect each other). The end points of the ray and the mirrors must reside within the boundaries of the room; the length of the ray vector and the length of each mirror must be at least one.

  For this purpose, write an auxiliary function that creates a random non-negative integer within a given interval by declaring a C++ random number generator as follows:

  ```
  default_random_engine rng;
  ```

  If you initialize this variable in `init` as

  ```
  random_device rand_dev;
  rng.seed(rand_dev());
  ```

  then the random number generator is nondeterministically seeded such that each run of the program produces different values. You can use this generator to produce numbers of type T that are uniformly distributed in interval $[A, B]$, by globally declaring an object:

  ```
  uniform_int_distribution<> next_r(A,B);
  ```

  Then each call

  ```
  int r = next_r(rng);
  ```

  sets variable r to such a number. For each interval of random numbers needed by the program, you can declare a corresponding object.

- The program may also be called with one command line argument (i.e., `argc==2`). This argument `argv[1]` is the name of a text file that contains a sequence of lines

  ```
  x0 y0 x1 y1
  n
  x01 y01 x11 y11
  ...
  x0n y0n y1n y1n
  ```

  of integer numbers. The first line describes the initial state of the ray by position (x_0, y_0) and vector $(x_1 - x_0, y_1 - y_0)$. The second line describes the number of mirrors. Each line i of the subsequent n lines describes the position of mirror i with first point $(x_{0,i}, y_{0,i})$ and second point $(x_{1,i}, y_{1,i})$. The function `init` reads this file and initializes the program variables `ray`, `n`, and `mirror` appropriately. If the input file does not exist or is ill-formed, the program may abort with an error message. Please note that no additional mirrors representing the walls need to be created; if such walls exist, they are also described in the input file.

- However the program is called (with or without argument), the initial values of `ray`, `n` and `mirrors` n are printed to the standard output in the form shown above.

- The function `drawMirrors` first clears the screen by painting a filled rectangle of size $W \times H$ in white. It then draws each mirror as a blue line.

- The function `drawRay` animates the path of the ray from its start point to its end point as a red line; the animation consists of a sequence of r/D steps where r is the length of the ray from its first point to its second point and D is a user-defined constant that determines the speed of the animation; in each step the progress of the ray is displayed by drawing D additional pixels of the ray. After r/D iterations, the whole ray is displayed; the second point is then highlighted by a filled red circle of radius R.

- The function `reflectRay` updates the second point of `ray` to the reflection point and determines the new state `ray0` of the ray after the reflection: the first point of `ray0` is the second point of `ray`; its second point determines a vector of the same length as before the reflection but now pointing in a new direction.

  A first version of `reflectRay` only checks for reflections of the ray with horizontal or vertical mirrors (as are the walls of the room); the rays may pass through all other mirrors. Steps 1–3 below show how to compute the reflection point; the vector of the reflected ray differs from the vector of the original ray only in that the horizontal component is negated by reflection on a vertical wall and the vertical component is negated by reflection on a horizontal wall, respectively.

- In order to test floating point numbers d_1, d_2 for equality with a certain numerical tolerance, implement a function that uses two constants a and r denoting absolute and relative error. The values d_1 and d_2 are then "equal", if $|d_1 - d_2| < a$ or else if $|(d_1 - d_2)/d_2| < r$ (for the later test, take as d_2 the bigger of the two values).

  For inequality tests, always check first whether d_1 is "equal" to d_2 by applying above function before comparing further with the builtin operations $<$ or $>$.

Test your program once without argument and once with the file `input.txt` given on the course site. Give in each case as a deliverable the text output of your program (showing the initial values of the ray and the mirrors) and the screenshot of the final situation after $T = 50$ updates (for $D = 4$, the animation should run with reasonable speed).

After that, write a new version of `reflectRay` (keep the original one as `reflectRayOld`) to consider reflections on all kinds of mirrors (thus the special cases of vertical or horizontal mirrors need not be handled separately):

1. Determine for the ray and for each mirror the coefficients a, b, c of the equation $ax + by + c = 0$ that describes the line on which the ray respectively the mirror lies. From this determine the intersection point of both lines.
2. If there is no (respectively no unique) intersection point, the mirror is not a candidate for reflecting the ray. Likewise, if there is an intersection point p, but it is not between or on the end points a and b of the mirror (as can be determined by comparing the signs of vectors $\overrightarrow{ap}$ and $\overrightarrow{pb}$), the mirror is not a candidate for reflecting the ray. Furthermore, if p is not in the direction denoted by the ray vector (as can be determined by comparing the signs of vectors $\overrightarrow{ap}$ and $\overrightarrow{ab}$), the mirror is also not a candidate for reflection.
3. Among all candidates for reflections, the reflection point is that intersection point that is closest to the first point of the ray. If there is no candidate for reflection at all (i.e., the ray runs into infinity), you may abort the program with an error message.
4. After the reflecting mirror and the reflection point p have been determined, the direction of the ray after the reflection has to be computed. You can accomplish this as follows:

a) Determine the mirror vector and the original ray vector (by subtracting the second point from the first point).

b) Convert each vector from Cartesian form (x, y) into polar form (r, a) where r represents the length of the vector and a its angle with the horizontal axis (see below).

c) Subtract from the angle of the ray vector the angle of the mirror vector (i.e. rotate the coordinate system such that the mirror vector becomes horizontal).

d) Convert the resulting ray vector back into Cartesian coordinates (see below). Negate its vertical component (i.e. reflect the vector on the horizontal axis).

e) Convert the resulting ray vector back into polar form. Add to its angle the angle of the mirror vector (i.e., rotate the coordinate system back).

f) Convert the resulting ray vector back into Cartesian form. Add it to the first point of the ray to determine its second point.

In your program avoid code duplication (and recomputation of values) but make extensive use of auxiliary functions (and variables). In particular, the functions

```
void toPolar(double x, double y, double &r, double &a)
{
  a = atan2(y, x);
  r = sqrt(x*x+y*y);
}

void toCartesian(double r, double a, double &x, double &y)
{
  x = r*cos(a);
  y = r*sin(a);
}
```

can be used to perform the conversions of a vector from Cartesian form (x, y) into polar form (r, a) and back.

Test your updated programs in the same way as the original one and give the screenshots of the new final situations.

Deliver in your report the complete source code of the final version of the program with the new version of the `reflectRay` function and the old one renamed to `reflectOld`.