Assignment #0

Introduction to object-oriented programming with C++

Modeling and Simulation of Appearance

Master on Robotics, Graphics and Computer Vision

About

The goal of this un-graded assignment is to provide a brief overview of the basics of object-oriented programming in C++. While simple, this lab session will cover several concepts that you will be working with throughout all the lab sessions of the course. Note that this is a very small subset of all C++ functionalities. Please refer to the C++ tutorial by W3Schools[1] for a comprehensive overview on C++ and object-oriented programming, as well as the online C++ reference website[2]. In particular, we recommend you to learn about the following concepts in C++ not fully covered in this assignment:

- Syntax of control flow statements (if/else, for and while loops, switch, etc.), operators, conditions, and data types.
- Class constructors and destructors.
- Reference vs. pointer variables, constant variables, static variables, static classes.
- Pass by value, pass by reference, pass by pointer, in function parameters.
- Importing libraries. Declaration and implementation through `*.hpp` and `*.cpp` files.
- I/O operations on files.

1 Compiling and running C++ code

For this assignment you will only need a C++ compiler. We recommend you to install g++, but others such as `gcc` can be used as well. To compile and run a simple C++ program written in a cpp file `<your_file>.cpp`, you just have to run the following commands:

```
$ g++ <your_file>.cpp -o your_file
$ ./your_file
```

The first command takes `<your_file>.cpp` as an input for g++, and creates an executable file `your_file` in the current folder. You can use this command throughout this assignment

[1] https://www.w3schools.com/cpp/default.asp
[2] https://en.cppreference.com/w/

to compile the associated source code files. Please refer to C++ documentation for more complex methods to compile a project.

2 Reading and writing variables from the command line

The assignment files include a file named `p0_0.cpp`. This file contains a basic implementation of a program that reads two real values from the command line, stores them in two float-point variables, and writes their values afterwards. Understand, compile, and run the program.

```
$ g++ p0_0.cpp -o p0_0
$ ./p0_0
$ Introduce x and y coordinates (sep. by space): 4.5 5.5
$ You have introduced 4.5 and 5.5
```

3 Operations with variables

Besides reading and writing variables, you may want to do some calculations with them. We provide a file named `p0_1.cpp`, where the user is asked to introduce the coordinates of two 2D points (i.e., two pairs of (x,y) coordinates). The program has to calculate the Euclidean distance between those two points. For that purpose, you should write the implementation of the function `euclideanDistance(x1,y1,x2,y2)`.

This is a graphics course, and as such, you may have to operate with geometries. We include a file named `p0_2.cpp`, where the user is asked to introduce the three vertices of a triangle (where each vertex is a two-dimensional point). The program has to calculate the total area of the triangle formed by those three vertices. For that purpose, you should complete the function `triangleAreaFromVertices(x1,y1,x2,y2,x3,y3)`.

4 Arrays

In some cases, declaring too many variables of the same type can be cumbersome and not optimal, and can hinder the understanding and modification of the code. For such cases, using arrays (i.e., ordered structures of the same data type) can be useful.

We provide a similar file to the previous one, named `p0_3.cpp`. While the program does the same calculations as in `p0_2.cpp`, this time it uses an array of float variables (instead of passing every single float variable one by one). You will need to implement the same function `triangleAreaFromVertices(v)` as in the previous example, but with the vertex coordinates given as an array of float values.

C++ arrays allow the user to declare ordered sets of variables in a practical way, provided they are all the same type (in this case, float type). Try to create a new float array including both the vertices of a triangle (as in the previous example) and its area (using the aforementioned function). If you try to assign variables from some data types to position of an array of some other data type, the compilation will typically fail—unless a type conversion is implicitly performed—, and you will not be able to run your program. To test this, create a new array of nine floats `float a[9]`, and assign to its different positions the six coordinates of the triangle vertices, the triangle area, the number of vertices of the triangle, and a string with the triangle name (e.g., "Triangle"). Your program will fail in compiling on some of those cases, while in some others may perform a type conversion.

5 Structs

In order to store different types of data in a single variable, more advanced data structures can be used. For this purpose, structs are a composite data structure that defines an aggregation of variables of arbitrary types that are typically related in some way. Each variable within a struct is known as *member*. In C++, we can declare struct-type variables in two ways: by defining first a struct data type and declaring a variable of such data type, or by directly defining an unnamed struct data type with the variable name at the end. The following code blocks illustrate these:

```
// Data type Vertex2
struct Vertex2 {
  float x;
  float y;
};
// Define the variable
Vertex2 my_vertex;
my_vertex.x = 0.23;
my_vertex.y = -1.67;
```

```
// Directly define the variable
struct {
  float x;
  float y;
} my_vertex;

my_vertex.x = 0.23;
my_vertex.y = -1.67;
```

When declaring a variable as a struct type, any piece of code in our program with scope for that variable will be able to evaluate and modify the members of the variable. We provide a program `p0_4.cpp` where some structs are already implemented. See how to declare, read, and write struct members, and complete the function `triangleAreaFromVertices(v1, v2, v3)`.

6 Classes

Beyond structs, C++ provides more complex ways of aggregating code elements, such as classes. Classes are a fundamental building block in object-oriented programming that allows to aggregate variables (typically referred to as *attributes*) and functions, under a

predefined code implementation. The motivation of aggregating attributes (e.g. position, radius) and functions associated with such properties (e.g. `area(...)`, `volume(...)`, `translate(...)`) is to represent the behavior of some higher-level element (e.g. a sphere) under a single block of code named class (e.g. `class sphere {...}`). Classes allow you to create instances of variables that have the same attribute structure and the same set of predefined functions with specific behavior. The variables of a class declared within a program are known as *objects*, which become a copy of that class (attributes and functions) that can be accessed and run across our program. While objects share the same structure of attributes and functions of their class type, they typically have different attribute values, which may yield different function outputs for the same input parameters. Class definition and usage is the basis of object-oriented program design and coding. Access to object's attributes and functions is analogous to access of structs members, as `myobject.attribute1` (for an attribute), or `myobject.function1(param1, param2)` (for function evaluation). When a variable is declared as a pointer to a class, the access operator for class attributes and functions is `"->"` instead of `"."`. This syntax is analogous for variales declared as pointers to structs. Please refer to the documentation online for more information about pointers and memory allocation and management.

As an example of simple class usage, the file `p0_5.cpp` implements a class `Triangle` that should include data and functionalities associated with a triangle shape. You do not have to implement anything in this class, but learn how to declare variables inside a class, and how to access each of them. This first program, while fully functional, defines all the class attributes and functions under the keyword `public:`. Attributes and functions of an object declared as public can be evaluated and modified *publicly* from any part of our program, either inside its own functions, from inside other objects' functions or other execution blocks in our program, as long as that object is within the scope of that piece of code. Note a class that only contains public attributes in general has an equivalent behavior to a struct data type with the same members, as displayed in the following example.

```
struct Vertex2 {
  float x;
  float y;
};

// my_vertex is a struct of
// type Vertex2
Vertex2 my_vertex;
my_vertex.x = 0.23;
my_vertex.y = -1.67;
```

```
class Vertex2 {
  public:
    float x,
    float y;
};
// my_vertex is an object
// of class Vertex2
Vertex2 my_vertex;
my_vertex.x = 0.23;
my_vertex.y = -1.67;
```

However, usually we may want to constrain or control the access to some object's attribute or function within the scope of its own object, for example to prevent other parts of our program from accessing or modifying the object's attributes, or to deliver or modify the objects attributes only under certain conditions. For this purpose, C++ allows you to define some attributes and functions as `private` under the keyword `private:`. Any private

attribute or function within a class will only be visible within that class implementation. Any attempt to evaluate or manipulate a private element from outside that class will result in a compilation error.

A practical example: a class definition can ensure that some of their attributes are read-only to other parts of our program by declaring those attributes as private, and providing *public* functions within the class that return each attribute's value. A class may also define functions that, for example, only allow to modify some of the attributes simultaneously, or that verify that the input values are correct within the context of our object's expected behavior (e.g. not setting a negative value for an attribute that represents area or length).

There is a particular method: the constructor, which has the exact same name than the class, that can be used to initialize all the attributes of the class, and is invoked when any object of that class is created. The following code block illustrates an example:

```
class Vertex2 {
  private:
    // These attributes can be manipulated only within this class
    // "this->x" is equivalent to "x".
    float x,
    float y;
  public:
    //This is the constructor with parameters,
    // used to initialize the class members
    Vertex2(float _x, float _y) : x(_x), y(_y) {}
    // These functions can be called from any part of the code
    float get_x(){
      return x;
    }
    float get_y(){
      return y;
    }
    void modify_xy(float x0, float y0){
      this->x = x0;
      this->y = y0;
    }
};
// my_vertex is an object of class Vertex2
// initialized for particular values
Vertex2 my_vertex(0.1,0.3);
my_vertex.x = 0.23; // compilation error!
my_vertex.y = 0.67; // compilation error!
my_vertex.modify_xy(0.23, 0.67); // this works
```

We provide p0_6.cpp which requires no additional implementation, but serves as an example of how to work with class attributes and functions. You will find a new keyword `this` used within the functions of that class, which is a pointer that refers to the current instance of

a class (i.e. a pointer to the object that was created in runtime). Each function within the class definition can use `this->` to access the objects attributes or functions. Compile, run, and understand the code in `p0_6.cpp`.

7 Inheritance

Class inheritance is a mechanism supported by some object-oriented programming languages (such as C++) to re-use, re-define, and specify the implementation of another existing class. Under this principle, a class can be defined to inherit the structure from another class, usually called the *parent* class. The class that inherits from the parent class is called the *child* class or *subclass*, and it may or may not retain part of the behavior of the *parent* class. A child class can re-define (or define for the first time) the implementation of existing functions of the parent class. It can also add new functions and attributes. In general, a child class cannot re-define or override attributes of the parent class.

In some cases, the parent class may not define the implementation of some of its functions itself, but only declare the signature of its functions, which corresponds to their names, the types and order of their parameters, and the type of the returned value. In those cases, the parent class is referred to as *abstract class*, which cannot be used to create an object. The goal of an abstract class is usually to serve as an interface for child classes that constraints the signature of a set of functions.

Additionally, in order to let a child class manipulate some attributes of the parent class, we can declare some attributes as `protected:` instead of private. This will allow the child class to read and modify those parameters, but they will remain private for other non-child classes or parts of our program.

The following code implements an example of a parent class:

```
class ColorFunctionReal {
public:
    virtual RGBf color_at(float x, float y) const = 0;
};
```

This parent class serves as an abstraction of a function that yields a color from two real numbers. This parent class is used from two different functions:

```
void fill(Image& image, const ColorFunctionReal& f, int spp) {
```

```
void fill_parallel(Image& image, const ColorFunctionReal& f, int spp) {
```

Both of them fill an image by taking values from that color function. Note that these functions can be created even if there is no child class yet, and use the corresponding abstract functions form the parent class even without it being implemented.

In the provided code, you can find two files `rhombus_aa.cc` and `julia.cc` that make use of this features by defining specific child classes and invoking a function to fill an image, such as:

```
class Julia : public ColorFunctionReal {
    RGBf dark,middle,bright;
    std::complex<float> domain;
    unsigned int range;
public:
    Julia(const RGBf& d, const RGBf& m, const RGBf& b,
      std::complex<float> c, unsigned int r) :
          dark(d),middle(m),bright(b),domain(c),range(r) {}
    RGBf color_at(float x, float y) const override {
        std::complex<float> z0(x,y);
        std::complex<float> result;
        unsigned int exit = range;
        for (unsigned int i = 0; i<range; ++i) {
            z0 = z0*z0 + domain;
            if (std::abs(z0) >= 4) { exit=i; break; }
        }
        if (exit < (range/2))
      return dark*(float(range/2 - exit)/float(range/2))
            + middle*(float(exit)/float(range/2));
        else
      return middle*(float(range - exit)/float(range/2))
            + bright*(float(exit - range/2)/float(range/2));
    }
};
```

As a final task for this assignment, play with inheritance for a while:

- Play with the Julia set in `julia.cc` by changing the parameters of the constructor of `Julia` and generate different fractal images.
- In the same file, change the call `fill_parallel` with a `fill` call. The non-parallel implementation should work noticeably slower. Notice that you didn't need to change anything about any class. Then, bring back the `fill_parallel` invocation. It works faster.
- Copy and rename `julia.cc` (or `rhombus_aa.cc`) and define your own image function by inheriting from `ColorFunctionReal` and overriding the `color_at` method with your own implementation (not a rhombus nor a Julia set). You can get as creative as you want here and obtain a beautiful image.

8 What to submit?

This assignment is **not evaluable**, and therefore you do not need to submit anything. However, we highly recommend you to understand all the concepts and finish all the programming-related tasks stated throughout this document.