1. Define an abstract class GeoObject with the geographic coordinates of its main point (longitude, latitude), a label describing it, proper setters and getters for the attributes, the virtual destructor and a pure virtual method printInfo. Try to define the coordinates as a separate class of Point.
   Define the following classes inherited from the GeoObject:
   - Marker extending the GeoObject with a size and a colour,
   - Polygon extending the GeoObject with a fill colour, a border colour and a dynamic array/list of the polygon's coordinates.

   Override the virtual method for the classes printing all the features of the particular object. Implement the necessary constructors, destructors, getters, setters, other methods and exceptions.
   Define the class Layer with a dynamic array/list of pointers to the geographic objects. Implement public methods of the class enabling to add a new object of an arbitrary type to the layer and to remove all the objects from the layer. Overload the indexing operator ([]) for the layer to have a direct access to the particular object in the layer (throwing a range check exception if necessary). Add other members which are necessary to implement the class.
   Write a program which tests all the class capabilities, in particular the following code should be enabled:

```
Layer layer("Layer 1");

try
{

      layer.addObject(new Marker(20.42, 50.12, "Point", 10, 0xff0000));
      layer[0].setLabel("Point 1");

      layer.addObject(new Marker(22, 49, "Point 2", 16, 0x00ff00));
      layer[1].setCoordinates(22.56, 49.04);

      Polygon *poly = new Polygon(20, 50, 0xffffff, 0x00ff00, "Poly 1");
      poly->addPoint(20, 50);
      poly->addPoint(21, 50);
      poly->addPoint(21, 49);
      poly->addPoint(20, 49);
      layer.addObject(poly);

      for(int i = 0; i < layer.size(); i++)
            layer[i].printInfo();

}
catch(Layer::RangeError &e)
{

      cout << e.what() << " Max=" << e.maxIndex(); //"Index out of range. Max=2"

}

layer.clear();
```

2. Define an abstract class Process with the unique read-only identifier (PID) and a name, proper setters and getters for the attributes, the virtual destructor and a pure virtual methods run, stop and getStatus (*stopped* or *running* with additional data). Define the following classes inherited from the Process:
   - Service extending the Process with a description and a starting mode (automatic, manual),
   - abstract class Application extending the Process with an owner name,
   - ConsoleApplication extending the Application with the text encoding,
   - WindowApplication extending the Application with the dimensions of the window.

   Override the virtual methods for the non-abstract classes. Implement the necessary constructors, destructors, getters, setters, other methods and exceptions. Write a program which tests all the classes, in particular the following code should be enabled:

```
queue<Process*> tasks;

Service *p1 = new Service("logger", "System logger", "automatic");
p1->setMode("manual");
tasks.push(p1);

ConsoleApplication *p2 = new ConsoleApplication("cmd", "user1", "windows-1250");
p2->setEncoding("UTF-8");
tasks.push(p2);

WindowApplication *p3 = new WindowApplication("browser", "user2", 800, 600);
p3->setWindow(1366, 768);
tasks.push(p3);

try
{

            Process *p = tasks.front();
            p->run();
            cout << p->getPID() << ":" << p->getStatus() << endl; //1:running
            p->stop();
            cout << p->getPID() << ":" << p->getStatus() << endl; //1:stopped
            tasks.pop();

            p = tasks.front();
            cout << p->getPID() << ":" << p->getStatus() << endl; //2:stopped
            p->stop();

}
catch(Process::RunError &e)
{

      cout << e.what(); //"Process <PID, name> has been already started."

}
catch(Process::StopError &e)
{

      cout << e.what(); //"Process <PID, name> is not running."

}
```