Exercises on C++: Inheritance

Giuseppe Lipari
`http://retis.sssup.it`

Scuola Superiore Sant'Anna – Pisa

May 25, 2009

Outline

Exercises on Inheritance

- Virtual functions
  - See virtual2/.
  - What happens when you execute the program?
- Private and protected inheritance
  - Use private and protected inheritance to create two new classes from a base class.
  - Then attempt to upcast objects of the derived class to the base class.
  - Explain what happens.

Exercise on Constructors

- The Rock class
  - Create a class Rock with a default constructor, a copy-constructor, an assignment operator, and a destructor, all of which announce to cout that they've been called.
  - In main( ), create a vector$<$Rock$>$ (that is, hold Rock objects by value) and add some Rocks.
  - Run the program and explain the output you get.
  - Note whether the destructors are called for the Rock objects in the vector.
  - Now repeat the exercise with a vector<Rock*>.
  - Is it possible to create a vector$<$Rock&$>$?

Exercise on Proxy

- The proxy
  - This exercise creates the design pattern called proxy.
  - Start with a base class Subject and give it three functions: f( ), g( ), and h( ).
  - Now inherit a class Proxy and two classes Implementation1 and Implementation2 from Subject.
  - Proxy should contain a pointer to a Subject, and all the member functions for Proxy should just turn around and make the same calls through the Subject pointer.
  - The Proxy constructor takes a pointer to a Subject that is installed in the Proxy (usually by the constructor).
  - In main( ), create two different Proxy objects that use the two different implementations.
  - Now modify Proxy so that you can dynamically change implementations.

Exercises on templates

- Write a class template that uses a vector to implement a stack data structure
- Modify your solution to the previous exercise so that the type of the container used to implement the stack is a template template parameter.

Exercise on templates - II

- Smart Pointer
  - Create a template class SmartPointer that holds a pointer
  - Define constructor, copy constructor, and assignment operator
  - However, only one smart pointer object is the owner! (i.e. responsible for calling delete when the smart pointer goes out of scope)

Exercise on templates - II

- Smart Pointer
  - Create a template class SmartPointer that holds a pointer
  - Define constructor, copy constructor, and assignment operator
  - However, only one smart pointer object is the owner! (i.e. responsible for calling delete when the smart pointer goes out of scope)
  - this is similar to pointer in Java with automatic garbage collector when nobody points to an object;

Exercise on templates - II

- Smart Pointer
  - Create a template class SmartPointer that holds a pointer
  - Define constructor, copy constructor, and assignment operator
  - However, only one smart pointer object is the owner! (i.e. responsible for calling delete when the smart pointer goes out of scope)
  - this is similar to pointer in Java with automatic garbage collector when nobody points to an object;
  - Hint: use the reference-counter technique

Outline

The final project

- The final project will consist in a program to simulate discrete-time systems
- As an example of such kind of application, you can consider Matlab/Simulink, restricted to the discrete-time library of blocks
- A system is built as a collection of “blocks, connected by signals. Each block implements a sub-system with2 the following equation:

$$\begin{aligned} x(k+1) &= f(x(k), u(k)) \\ y(k+1) &= g(x(k+1)) \end{aligned}$$

- where x, y and u are vectors of reals, f and g are functions. x represents the internal state, u is the input signal, y is the output signal.

Blocks

- Some block can be stateless (i.e., $y(k) = g(u(k))$)
- A block may be implemented directly as an equation, or in terms of other blocks. The final system will be a superblock, with inputs and outputs, that may contain other blocks.
- It is possible to design systems with feedback. However, in every feedback loop, there must be at least one block with state (i.e. no fixed point loop computation), otherwise an exception is raised.
- The library should contain a collection of blocks, and signals.

User requirements

- The user of the library should be able to:
  1. create the appropriate blocks and superblocks (subsystems) hierarchically
  2. connect the blocks
  3. create source signals
  4. create scopes (to observe the evolution of the system)
  5. run a simulation, collect and show the results
  6. debug a system by running a step-by-step simulation

User interface

- The user interface is a configuration file that specifies the blocks and their connections, the scopes, the source signals
- The user runs the command-line program that
  - reads the config file
  - creates and connects all objects
  - runs the simulation producing the outputs

Assignment - I : Parser

- Write the parser component of the library
  - A set of object that are passed a configuration file
  - reads the file and separates them in tokens organized hierarchically
- Language specification
  - the file consists of a list of these elements
    - *block_name* = *block_type*(*parameters*);
    - *source_name* = *source_type*(*parameters*);
    - *sink_name* = *sink_type*(*parameters*);
    - connect(*block_name*.output[*i*], *block_name*.input[*j*]);

Language specification example

```
block_a = moving_average(10);       // moving average, 10 samples
block_b = filter_high(1000);        // high pass filter, 1Khz
source = sin(10, 2.5);              // 10 * sin(2.5 * t);
sink1 = simple_scope(out1.txt);     // output on out1.txt
sink2 = simple_scope(out2.txt);     // output on out2.txt
connect(source.output[0], block_a.input[0]);
connect(block_a.output[0], block_b.input[0]);
connect(block_a.output[0], sink1.input[0];
connect(block_b.output[0], sink2.output[0]);
```