CPSC 427: Object-Oriented Programming

Michael J. Fischer

Lecture 14
October 22, 2018

Handling Circularly Dependent Classes

Modeling the Think-A-Dot Machine

Handling Circularly Dependent Classes

Tightly coupled classes

Class `B` *depends on* class `A` if `B` refers to elements declared within class `A` or to `A` itself.

The class `B` definition must be read by the compiler after reading `A`.

This is often ensured by putting `#include "A.hpp"` at the top of file `B.hpp`.

A pair of classes `A` and `B` are *tightly coupled* if each depends on the other.

It is not possible to have each read after the other.

Whichever the compiler reads first will cause the compiler to complain about undefined symbols from the other class.

Example: `List` and `Cell`

Suppose we want to extend a cell to have a pointer to a sublist.

```
class Cell {
  int data;
  List* sublist;
  Cell* next;
  ...
};
class List {
  Cell* head;
  ...
};
```

This won't compile, because `List` is used (in `class Cell`) before it is defined. But putting the two class definitions in the opposite order also doesn't work since then `Cell` would be used (in `class List`) before it is defined.

Circularity with #include

Circularity is less apparent when definitions are in separate files.

File list.hpp:

```
#pragma once
#include "cell.hpp"
class List { ... };
```

File cell.hpp:

```
#pragma once
#include "list.hpp"
class Cell { ... };
```

File main.cpp:

```
#include "list.hpp"
#include "cell.hpp"
int main() { ... }
```

What happens?

In this example, it appears that `class List` will get read before `class Cell` since `main.cpp` includes `list.hpp` before `cell.hpp`.

Actually, the opposite occurs. The compiler starts reading `list.hpp` but then jumps to `cell.hpp` when it sees the `#include "cell.hpp"` line.

It jumps again to `list.hpp` when it sees the `#include "list.hpp"` line in `cell.hpp`, but this is the second attempt to load `list.hpp`, so it only gets as far as `#pragma once`. It then resumes reading `cell.hpp` and processes `class Cell`.

When done with `cell.hpp`, it resumes reading `list.hpp` and processes `class List`.

Resolving circular dependencies

Several tricks can be used to allow tightly coupled classes to compile. Assume `A.hpp` is to be read first.

1. Suppose the only reference to `B` in `A` is to declare a pointer. Then it works to put a "forward" declaration of `B` at the top of `A.hpp`, for example:

```
class B;
class A { B* bp; ... };
```

2. If a function defined in `A` references symbols of `B`, then the *definition* of the function must be moved outside the class and placed where it will be read after `B` has been read in, e.g., in the `A.cpp` file.
3. If the function needs to be inline, this is still possible, but it's much trickier getting the inline function definition in the right place.

Modeling the Think-A-Dot Machine

Modeling Think-A-Dot

The ThinkADot class in PS3 illustrates the issues in keeping a clean separation between the external model and the internal implementation of the functionality of the model.

The external description of the machine identified the three holes (A, B, C) into which a marble could be dropped, and the two output holes (P, Q) from which the marble would come out of the machine. It also included the 8 colored dots visible on the front of the machine and their geometric relationships to the holes and the dots.

The external actions that a user can perform on a real Think-A-Dot machine are to drop a ball in one of the input holes and observe how the dots change and where the ball comes out, and to tip the machine to one side or the other.

ThinkADot.hpp

The public interface in the ThinkADot class is intended to model the externally visible parts of a ThinkADot.

The private part of the interface is to allow the faithful implementation of the public functions. The flip-flop gates are modeled as private data members.

Bridging the Gap

The place where the interface becomes tricky is in passing parameters to the public functions. For example, public function `play(h)` should be callable from outside of the class. It's purpose is to simulate the drop of a ball into one of the three starting input holes `h`.

What should the type of `h` be? It's natural to make it the number of the gate that the ball first encounters. But the external user doesn't know how the gates are numbered. She only knows the identities of the input holes.

A Solution

Class `ThinkADot` should have a public enum type `InHole`.
The parameter to `play()` should be an object of type `InHole`.

The first thing `play()` should do is to translate the parameter into the private internal gate name that corresponds to the input hole.

The internal names are given by the private type

```
enum Place { T0, T1, T2, M0, M1, B0, B1, B2, LEXIT, REXIT };
```

Then `InHole A` corresponds to `Place T0`, `B` to `T1`, and `C` to `T2`.

Implementing the translation

A simple switch statement is sufficient to carry out the translation:

```
play(InHole h) {
    Place pl;
    switch (h) {
    case A:
        pl = T0;
        break;
    case B:
        pl = T1;
        break;
    case C:
        pl = T2;
        break;
    }
    ...
```

Output translation

Similar remarks apply to the translation from the internal representation of the two output channels `LEXIT` and `REXIT` to the external representation given by the public enum type

```
enum OutHole { P, Q };
```

`play()` returns a value of type `OutHole`, so its complete prototype is `OutHole play( InHole h );`