Winter 2020 CS 32

Project 3 FAQ

1. **How can I get started writing code?**

   Start with this task: Re-read the spec, but when it mentions game objects, pay attention only to things relating to Socrates and dirt. Implement a Socrates class whose *doSomething()* method does nothing. Have StudentWorld's *init()* create a new Socrates, tell him to do something, and have *cleanUp()* delete him. This can be done in about five lines of StudentWorld code and five lines of Actor code. Once you do this, you'll be surprised at what a psychological boost it is to see the graphics on the screen. (You can type `q` to quit, but will have a memory leak if the StudentWorld destructor doesn't call *cleanUp()*.)

   With a few more lines of code, you can create dirt at a fixed location. Then work on getting Socrates to move in response to player input.

2. **How do I generate random numbers?**

   GameConstants.h has a function `int randInt(int min, int max)` that returns a random int in the range [min, max].

3. **May we use `protected` members?**

   You must not use protected *data* members. You may use protected member *functions* if you wish.

4. **May we use `friend`?**

   No.

5. **Suppose that while calling *doSomething()* for each actor during a given tick, we create a new actor. Must we call *doSomething()* for the new actor on the current tick?**

   It's you choice whether or not you do. Of course, on the next tick, *doSomething()* will be called on all actors including that new one.

6. **Suppose there's a requirement about an interaction between an object of type A (e.g., a Flame) and an object of type B (e.g., a Regular Salmonella). If the spec says that the A object has to manage that interaction or that both the A and the B objects have to manage it, but I find it more convenient to have the only the B object manage it, is it acceptable to have the only the B object manage it if the result is essentially the same?**

   Yes.

7. **Why is a program compiled with g32 giving me a runtime error like this?**

   ```
   Actor.cpp:lineNum:…: runtime error: member call on address …
   which does not point to an object of type 'someType'
   …: note: object has invalid vptr
   ```

   A derived class constructor (at the indicated line number) is calling a function implemented in the derived or base class before the base class constructor has even been called. As far as the compiler knows, this could be dangerous, since the base object and the data members of the derived object have not yet been initialized — what if the function depends on their values?

8. **Could you please tell me a story about type casting?**

   Sure. Suppose we have the following class hierarchy:

   ```
   class Shape { ... };  // no fill function
   class LineSegment : public Shape { ... } // no fill function
   class Arc : public Shape { ... } // no fill function
   ```

```
class ClosedFigure : public Shape
{...
    virtual void fill(Color c) = 0;
};
class Circle : public ClosedFigure
{...
    virtual void fill(Color c) { ... }
};
class Rectangle : public ClosedFigure
{...
    virtual void fill(Color c) { ... }
};

Shape* f() { ... }
```

Only shapes that enclose an area can be filled with a color; you can't fill the nonexistent interior of a line segment or an arc, but you can fill a closed figure like a circle or a rectangle.

Suppose we call f, and the circumstances of the call are such that we *know* with certainty that for this particular call of f, the Shape* that it returns is a pointer to the Shape part of a Circle. We'd like to fill that circle. Here's what *won't* work:

```
Shape* sp = f();  // Suppose f() returns a pointer to the Shape part of a Circle
sp->fill(RED);  // Error!  Shape did not declare fill.

Circle* cp = f();  // Error!  No automatic conversion from Shape* to Circle*
cp->fill(RED);

ClosedFigure* cfp = f();  // Error!  No automatic conversion from Shape* to ClosedFigure*
cfp->fill(RED);
```

Although the conversion from a Derived* to a Base* (an *upcast*) is automatic, we have to explicitly ask for a conversion from a Base* to a Derived* (a *downcast*). Using `static_cast`, we had better be right: The behavior is undefined if the Base* doesn't actually point to a Derived object of the that type.

```
Shape* sp = f();  // Suppose f() returns a pointer to the Shape part of a Circle
Circle* cp = static_cast<Circle*>(sp);
cp->fill(RED);

ClosedFigure* cfp = static_cast<ClosedFigure*>(sp);
cfp->fill(RED);

static_cast<Circle*>(sp)->fill(RED);
static_cast<ClosedFigure*>(sp)->fill(RED);

Rectangle* rp = static_cast<Rectangle*>(sp);  // Undefined behavior!
rp->fill(RED);
```

For most compilers, the undefined behavior shows up as the call to fill doing something weird or crashing. Code compiled with g32 will report the bad downcast itself as a runtime error.

9. **As part of writing *a high-level description of each of your public member functions* (spec p. 55), do we have to write pseudocode?**

   Only for the more complex functions, of which there are probably comparatively few.