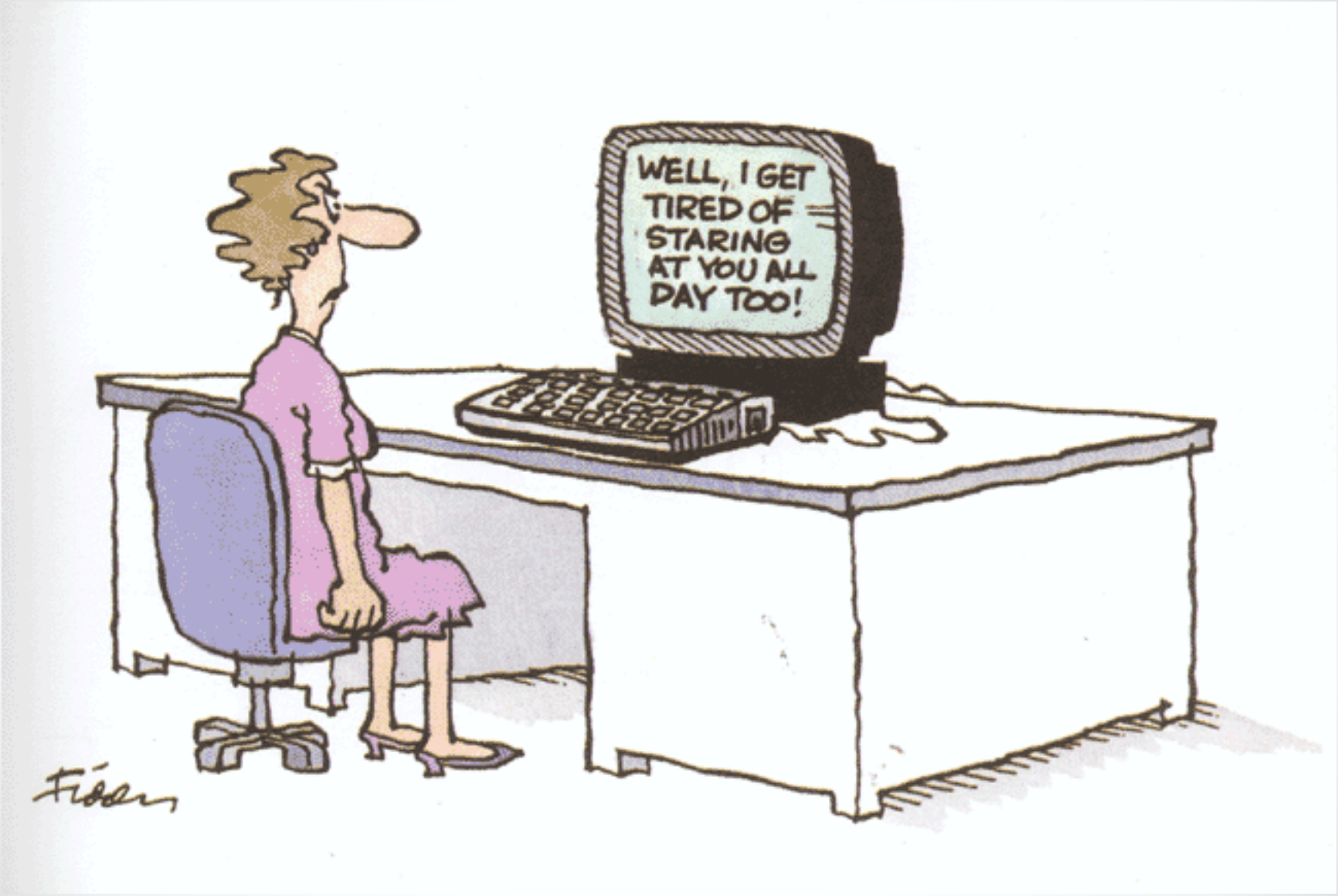
C++

SWE315

Lesson 13

Prof. Zachi Baharav
zbaharav@cogswell.edu


WELL, I GET TIRED OF STARING AT YOU ALL DAY TOO!

Lesson 8

- In this lesson:
  - Virtual functions (recap)
    - Static and Dynamic binding
  - Virtual destructors (use these whenever using inheritance!).
  - Pure Virtual functions.
  - Abstract classes (interface classes).
  - Lab: work. Mainly verifying projects status. (ideas, plan)
  - Good easy resource:
    - http://www.learncpp.com/cpp-tutorial/

Before we start

- Submitting non compiling (working) code.
  - Professionalism
  - Effort

- Final project:
  - Design big (OOP)
    - Board games: Variable size board (dynamic memory)
    - Various game pieces (classes, inheritance)
  - Implement 'small' (or at least in stages)
    - Make sure you progress along the path

Inheritance

- Base Class, Derived class
- Initialization list
- Protected (we saw in Animal example)
  - Like Private, but does allow derived classes to use.
- Example in class: Person

Base Class pointer

- A Base class pointer can point to Derived class object, and NOT vice versa
  - Array of pointers to Animal can point to Cat and Dog
  - Calling function 'report' with pointer to Base class, enables to use it for Cats and Dogs.

Virtual functions

- Animal example:
  - Speak() is virtual
  - When rAnimal.Speak() is evaluated, the program notes that it is a virtual function. In the case where rAnimal is pointing to the Animal portion of a Cat object, the program looks at all the classes between Animal and Cat to see if it can find a more derived function. In that case, it finds Cat::Speak(). In the case where rAnimal points to the Animal portion of a Dog object, the program resolves the function call to Dog::Speak().
- Static .vs. Dynamic binding:
  - Vtable(s)
  - Inefficiency of Virtual functions.

Virtual destructors

- Should ALWAYS use when dealing with inheritance.
  - Because of dynamic memory allocations in the derived classes!.
  - See code examples @ Learncpp

Pure virtual functions

- Setting the stage for the Derived classes to implement those.
- The derived classes HAVE to implement this.
- Cow example in class.

Interface classes

- An **interface class is** a class that :
  - No member variables,
  - All functions are pure virtual!

- In other words, the class is purely a definition, and has no actual implementation.

- Interfaces are useful when you want to define the functionality that derived classes must implement, but leave the details of how the derived class implements that functionality entirely up to the derived class.

- Taken (with minor modifications) from: http://www.learncpp.com/cpp-tutorial/126-pure-virtual-functions-abstract-base-classes-and-interface-classes/

Example: Error Log

```
class IErrorLog
{
    virtual bool OpenLog(const char *strFilename) = 0;
    virtual bool CloseLog() = 0;

    virtual bool WriteError(const char *strErrorMessage) = 0;
};
```

- Any class inheriting from IErrorLog must provide implementations for all three functions in order to be instantiated.
  - **File:** FileErrorLog, where OpenLog() opens a file on disk, CloseLog() closes it, and WriteError() writes the message to the file.
  - **Message box**: ScreenErrorLog, where OpenLog() and CloseLog() do nothing, and WriteError() prints the message in a pop-up message box on the screen.
  - **Graphics**, etc…

Example of using directly one of the derived classes

```
double MySqrt(double dValue, FileErrorLog &cLog)
{
    if (dValue < 0.0)
    {
        cLog.WriteError("Tried to take square root of value less than 0");
        return 0.0;
    }
    else
        return dValue;
}
```

- Limiting…

Example of using directly one of the derived classes

```
double MySqrt(double dValue, FileErrorLog &cLog)
{
    if (dValue < 0.0)
    {
        cLog.WriteError("Tried to take square root of value less than 0");
        return 0.0;
    }
    else
        return dValue;
}
```

```
double MySqrt(double dValue, IErrorLog &cLog)
{
    if (dValue < 0.0)
    {
        cLog.WriteError("Tried to take square root of value less than 0");
        return 0.0;
    }
    else
        return dValue;
}
```

Better!

Interface class….

- Interface classes have become extremely popular because they are easy to use, easy to extend, and easy to maintain.

- Java and C#, have added an "interface" keyword that allows programmers to directly define an interface class without having to explicitly mark all of the member functions as abstract.
  - Furthermore, although Java and C# will not let you use multiple inheritance on normal classes, they will let you multiply inherit as many interfaces as you like.
  - Because interfaces have no data and no function bodies, they avoid a lot of the traditional problems with multiple inheritance while still providing much of the flexibility.

END