

# Composition I

- *composition* is a way to combine or compose multiple classes together to create new class
- composition  $\equiv$  “has-a” relationship
  - a car “has-a” gear-box
  - a graduate student “has-a” course list
- if XX “has-a” YY, ZZ, ... then we could define a class with
  - XX being the *composite* class
  - YY, ZZ, ... being the *component* classes

## Composition II

- suppose a generic MCMC Sampler class looks like:

```
class Sampler
{
private:
    MonteCarloSpecs mcs;
    Draw current_draw;
    Draw proposal_draw;

public:
    Sampler (MonteCarloSpecs mcs,
            Draw draw);
    void do_sampling (void);
    void report_summary (void);
};
```

- so here a Sampler object has a MonteCarloSpecs object and two Draw objects

## Composition III

- construction of a composed class (Sampler) from the component classes (MonteCarloSpecs and Draw):

```
Sampler::Sampler (MonteCarloSpecs &mcs,  
                  Draw &draw)  
    : mcs(mcs),  
      current_draw(draw),  
      proposal_draw(draw)  
{  
    // empty body  
}
```

- that was easy, but wait, is `current_draw(draw)` a familiar construct in a initializer list?
- NO, its a little deep, its called the use of a “copy-constructor”

## Composition IV

- for any class the compiler gives you a “copy-constructor” e.g. compiler would generate code for `Draw (Draw const &draw);`
  - this constructs a new object of class `Draw`, say, `draw_new` by copying the members of the argument object of class `draw` `Draw const &draw`
  - if all the data-members of the class `Draw` are non-pointers, this automatic copying is fine, but what happens if you have pointer member(s), say, `double *var_comp_vals`
  - then after member copying `draw_new.var_comp_vals` and `draw.var_comp_vals` will point to the same memory location, wouldn't they?
  - so when the destructor `~Draw( )` is called once for `draw` and once for `draw_new`, guess what, you will get a nice error message saying “trying to free an memory location twice!”, why?

# Composition V

- so don't rely on the compiler, write your own copy-constructor if your class has at least one pointer data member:

```
class Draw {
private:
    int const var_comp_dim;
    double *var_comp_vals;
    double log_density;
    int check_var_comp_dim (int var_comp_dim);
    void set_log_density (double val);

public:
    Draw (int var_comp_dim,
          double const *var_comp_vals);
    Draw (Draw const &draw);
    ~Draw( );
    int const get_var_comp_dim (void) const;
    double get_log_density (void) const;
    double const * const get_var_comp_vals (void) const;
    double compute_log_density (void);
};
```

- note the (usual) constructor and the copy constructor have the same name (but they take different arguments), this is called function overloading

# Composition VI

- compare the implementations of the two types of constructors:

```
Draw::Draw (int var_comp_dim,  
           double const *var_comp_vals)  
  : var_comp_dim(check_var_comp_dim(var_comp_dim))  
{  
    assert(var_comp_vals);  
    cout << "Creating a Draw object" << endl;  
    this->var_comp_vals = new double[var_comp_dim];  
    for (int ii = 0; ii < var_comp_dim; ++ii)  
        this->var_comp_vals[ii] = var_comp_vals[ii];  
}  
Draw::Draw (Draw const &draw)  
  : var_comp_dim(draw.var_comp_dim),  
    log_density(draw.log_density)  
{  
    cout << "Creating a Draw object" << endl;  
    var_comp_vals = new double[var_comp_dim];  
    for (int ii = 0; ii < var_comp_dim; ++ii)  
        var_comp_vals[ii] = draw.var_comp_vals[ii];  
}
```

- no error checking necessary in the copy-constructor, why?

## Aside I

- one could have declared a constructor for `Sampler` like:

```
Sampler (int n_iters,  
         float time_in_secs,  
         float prop_burn_in,  
         string const name_of_algo,  
         int &debug_level,  
         int var_comp_dim,  
         double const *var_comp_vals);
```

- note this takes all the arguments necessary for passing to the constructors of its component classes: `MonteCarloSpecs` and `Draw`
- its a cumbersome and error-prone approach, so instead use copy-constrcutors for the component classes for constructing a complicated composed class

## Composition Use I

- the use of composed classes is same as of a regular class (only possible difference is in their construction where the copy-constructors of the component classes are being used)

```
int
main (int argc, char **argv)
{
    int debug_level = 1;
    MonteCarloSpecs mcs(10, 10, 0.1, "test", debug_level);
    double vals[] = { 1.0, 100.0 };
    Draw draw(2, vals);
    Sampler sampler(mcs, draw);

    sampler.do_sampling( );
    sampler.report_summary( );
    return 0;
}
```

## Code Files

```
prog7.H  
prog7.C  
prog7Makefile
```