Inheritance I

- *inheritance* is used to create new “child” classes from existing “parent” classes.
- *inheritance, subclass, derived class* all mean the same thing.
- inheritance $\equiv$ “is-a” relationship
  - a car “is-a” vehicle
  - a graduate student “is-a” student
- if XX “is-a” YY then we could form two classes called XX and YY with
  - XX being the *base class*
  - YY being the *derived class*
- here the derived class YY would have all the “functionalities” of the base class
Inheritance II

- why consider inheritance?
  - conceptually simpler to understand
  - saves lots of duplicate coding and hence time and error
Inheritance III

- Let's say we have the base class:

```cpp
#include <iostream>
#include <string>
using std::string;

class MonteCarloSpecs {
private:
    int n_iters;
    float time_in_secs;
    float prop_burn_in;
    double *log_density;
    string const name;

public:
    MonteCarloSpecs (int n_iters,
                     float time_in_secs,
                     float prop_burn_in = 0.05,
                     string const name = "Monte Carlo Specs");

    ~MonteCarloSpecs (void);
    int get_n_iters (void) const;
    void set_n_iters (int n_iters);
    float get_time_in_secs (void) const;
    void set_time_in_secs (float time_in_secs);
    float get_prop_burn_in (void) const;
    void set_prop_burn_in (float prop_burn_in);
    void print (void) const;
};
```
Inheritance IV

• declaration of a “derived” or “inherited” classes:

```cpp
class MHSpecs: public MonteCarloSpecs {
private:
    int sample_dim;

public:
    MHSpecs (int n_iters,
              float time_in_secs,
              float prop_burn_in,
              int sample_dim,
              string const name = "Metropolis Hastings Specs");
~MHSpecs (void);
    int get_sample_dim (void) const;
    void set_sample_dim (int sample_dim);
    void print (void) const;
};
```

• note here we do not re-declare the private and public members of the base class in the derived class declaration because “they are already there” due to inheritance
Inheritance V

• constructor:

```
MHSpecs::MHSpecs (int n_iters,
    float time_in_secs,
    float prop_burn_in,
    int sample_dim)
    : MonteCarloSpecs(n_iters, time_in_secs, prop_burn_in)
{
    set_sample_dim(sample_dim);
}
```

• using the base class constructor in the member intializer list is one of the better ways in this context
Inheritance VI

- destructor:

  ```
  MHSpecs::~MHSpecs (void)
  {
      cout << "destroying MHSpecs" << endl;
      // nothing to be done: empty body
  }
  ```

- note here nothing to be done because
  - inside `~MHSpecs`, `~MonteCarloSpecs` is called by the compiler
  - while creating a `MHSpecs` object we didn’t allocate any memory using `new`
Inheritance VII

- **overriding** a base class function:

  ```cpp
  void MHSpecs::print (void) const
  {
      MonteCarloSpecs::print( );
      cout << "sample_dim: " << get_sample_dim( ) << endl;
  }
  ```

- here we use the “scope resolution” operator :: to be able to reuse the code of the `print()` function for the class `MonteCarloSpecs`:

  ```cpp
  void MonteCarloSpecs::print (void) const
  {
      cout << "This " << name << " object:" << endl
           << "n_iters: " << get_n_iters( ) << endl
           << "time_in_secs: " << get_time_in_secs( ) << endl
           << "prop_burn_in: " << get_prop_burn_in( ) << endl
           << "log_density:" << endl;

      int ii;
      for (ii = 0; ii < n_iters - 1; ++ii)
          cout << log_density[ii] << ", ";
      cout << log_density[ii] << endl;
  }
  ```
Inheritance Use I

- the use of a derived class is same as any old class:

```c
int main (int argc, char **argv)
{
    MonteCarloSpecs mcs(20, 20, 0.2);
    MHSpecs MHs(10, 10, 0.1, 2);
    MonteCarloSpecs *mcsPtrArr[ ] = { &mcs, &MHs };

    mcs.print( );
    MHs.print( );
    cout << "================================" << endl
        << "MHs.n_iters: " << MHs.get_n_iters( ) << endl
        << "MHs.time_in_secs: " << MHs.get_time_in_secs( ) << endl
        << "MHs.prop_burn_in: " << MHs.get_prop_burn_in( ) << endl
        << "MHs.sample_dim: " << MHs.get_sample_dim( ) << endl;
    for (int ii = 0; ii < 2; ++ii)
        (mcsPtrArr[ii])->print( );
    return 0;
}
```
things to note:

- we can assign base class pointer point to a derived class object: note in the statement
  ```
  MonteCarloSpecs *mcsPtrArr[ ] = { &mcs, &MHs };
  mcsPtrArr[1] is &Mhs, i.e., it points to MHs
  ```

- the reverse is not allowed, why?

- note the use of the function `print`:
  ```
  * where what version of the function is called?
    mcs.print();
    MHs.print();
  
  * where what version of the function is called?
    for (int ii = 0; ii < 2; ++ii)
      (mcsPtrArr[ii])→print();
  ```

- note we get to use the functions `get_n_iters()`, `get_time_in_secs()` etc. on Mhs, i.e., on the derived class object because those are inherited
Runtime Polymorphism I

- in runtime polymorphism we can call the same code to produce “different results”:

- we noted that:
  - `mcs.print()` calls `MonteCarloSpecs::print()`
  - `MHs.print()` calls `MHSpecs::print()`

this is the “right thing”

- but:
  - `(mcsPtrArr[0])->print()` calls `MonteCarloSpecs::print()`
  - `(mcsPtrArr[1])->print()` calls `MonteCarloSpecs::print()`

is this the “right thing”? NO!

- can we make `(mcsPtrArr[1])->print()` call `MHSpecs::print()` instead which is the “right thing” to do?
Runtime Polymorphism II

- use of the virtual keyword:

```cpp
#include <string>
using std::string;
class MonteCarloSpecs {
private:
    int n_iters;
    float time_in_secs;
    float prop_burn_in;
    double *log_density;
    string const name;

public:
    MonteCarloSpecs (int n_iters,
                     float time_in_secs,
                     float prop_burn_in = 0.05,
                     string const name = "Monte Carlo Specs");

    virtual ~MonteCarloSpecs (void);
    int get_n_iters (void) const;
    void set_n_iters (int n_iters);
    float get_time_in_secs (void) const;
    void set_time_in_secs (float time_in_secs);
    float get_prop_burn_in (void) const;
    void set_prop_burn_in (float prop_burn_in);
    virtual void print (void) const;
};
```
Runtime Polymorphism III

- note we added the keyword `virtual` is added as a qualifier to the functions `print` and `MonteCarloSpecs`

- things to remember: (read and change and recompile code in `prog9.H` and `prog9.C` and verify the “truth” of the following)
  - the `virtual` keyword only applies to functions
  - a `virtual` function must be defined for the base class for which it was first declared
  - the derived classes may or may not implement the `virtual` function
  - if they don’t then the base class version of the `virtual` function would be used if called on an object of the derived class
  - any class with at least one `virtual` member function has to have a `virtual` destructor

- nothing else was changed in the implementation of the class `MonteCarloSpecs`, i.e., nowhere else the keyword `virtual` needs to be added
Runtime Polymorphism IV

- Let's look at the previous example to see the effect of the `virtual` keyword:

```c
int main (int argc, char **argv)
{
    MonteCarloSpecs mcs(20, 20, 0.2);
    MHSpecs MHs(10, 10, 0.1, 2);
    MonteCarloSpecs *mcsPtrArr[ ] = { &mcs, &MHs };

    mcs.print( );
    MHs.print( );
    for (int ii = 0; ii < 2; ++ii)
    {
        (mcsPtrArr[ii])->print( );
    }
    return 0;
}
```
Runtime Polymorphism V

- the `virtual` keyword:
  - it makes the compiler create a “virtual function table” or the “vtable” which points to the “right” implementation of a `virtual` function called on an object
  - `(mcsPtrArr[1])->print()` causes the compiler to realize that although `(mcsPtrArr[1])` is of type `MonteCarloSpecs` it is in fact a `MHSpecs` object and hence the “vtable” correctly makes `(mcsPtrArr[1])->print()` call the `MHSpecs::print()` function
  - this is how roughly how runtime polymorphism works
Code Files

prog3.H
prog3.C
prog3Makefile
prog4.H
prog4.C
prog4Makefile
prog9.H
prog9.C
prog9Makefile