Inheritance and classes

- sometimes we want to create a new class that is a specialized form of an existings class
- we'd like to avoid simply replicating the code for the original, would rather have some way of re-using it/incorporating it
- we can create new classes that ***inherit*** fields and methods from previously defined classes
- the original class is called the ***base*** class, the new class is called the ***derived*** class (sometimes called the **parent** and **child** classes)
- the derived class can use the inherited fields/methods, but can also override some of them with new versions

Conceptual example

- suppose we have a simulation with lots of types of land vehicles (cars, motorcycles, bicycles, unicycles, etc)
- we might create an initial **vehicle** class with fields and methods that would be needed by all vehicles
- we might derive a **motorized** class from **vehicle**, with fields and methods used by all powered vehicles
- we might derive an **automobile** class from **motorized**, with fields specific to automobiles
- at each level of inheritance we add just the extra functionality needed for the new specialization

Syntax for declaration

- in the derived class declaration we specify which class we're inheriting from
- class mynewclass: public thebaseclass { ... rest of class def }
  - (we'll talk about that public keyword later)
- the new class gets copies of all the fields and methods from the base class, and can declare its own additional fields and methods
- it can also override inherited methods, providing its own version of them

Today: simple static binding

- today we'll focus on simplest form of inheritance
- called static binding
- involves minimal use of keywords, but limits flexibility at runtime
- tomorrow we'll introduce the use of more keywords (virtual, override, final, etc) and discuss static vs dynamic binding

Inheritance example

```
class vehicle {
   private:
      float weight;
   public:
      vehicle();
      ~vehicle();
      void setwt(float w);
      float getwt();
      void print();
};
```

```
class motorized {
   private:
      float horsepower;
   public:
      motorized();
      ~motorized();
      void sethp(float hp);
      float gethp();
      void print();
};
// inherits methods
//   vehicle, ~vehicle
//   setwt, getwt, print
// inherits a weight field
//    but can't access it directly in motorized methods
//          would have to go through getwt/setwt
```

```
int main()
{
    motorized m;
    m.setwt(300);
    m.sethp(15);
    m.print(); // uses motorized print
}
```

Accessing overridden methods

- motorized inherited vehicle's print, but overrode it with it's own print
- can still access the original using vehicle::print() (classname::methodname)

```
int main()
{
    motorized m;
    m.setwt(300);
    m.sethp(15);
    m.print(); // uses motorized print
    m.vehicle::print(); // uses vehicle print, only displays the weight field
}
```

Accessing globals with ::

- sometimes within a class we'll use the same field/method name as an existing global constant/variable/function
  - e.g. a local field X and a global variable X
- to access the global from inside the class we can use :: and the global's name
  - e.g. to access the global X instead of the local X:
    - ::X = whatever;

private, public, and ***protected***

- private fields and methods do get inherited, but cannot be directly called/accessed inside the derived class methods
- public fields and methods are inherited and usable
- there is a third type: protected
  - these are not visible to outside functions/methods (e.g. main)
  - but they are accessible to derived classes
  - e.g. supposed vehicles has a protected field named topspeed
    - the derived motorized class can access the inherited field directly
    - functions/methods that aren't derived from vehicles cannot access topspeed directly

private,protected,public example

```
class parent {
   private:
      int x;
   protected:
      int y;
   public:
      int z;
};
```

```
class child: public parent {
   private:
      int A;
   protected:
      int B;
   public:
      int C;
};
```

```
class grandchild: public child {
   // inherits A,B,C from child
   // inherits x,y,z from parent through child
   // but can't access x or A fields directly
   //       since they were declared private
};
```

- chains of inheritance can go as deep as desired
- here grandchild is derived from child, which is derived from parent
- grandchild has everything from all its ancestors
- *(it can't directly access anything they made private, it would have to access those through the ancestors' protected/public methods)*

Order of constructors/destructors

- constructors run from earliest ancestor to latest descendant
- if we declared a grandchild object
  - parent constructor runs, initializing its fields
  - child constructor runs, initializing its fields (may adjust inherited fields)
  - grandchild constructor runs, initializing its fields (may adjust inherited fields)
  - makes sense if we think of the inheritance as marking specializations: each constructor initializes its associated fields, but the derived classes can then alter/customize
- destructors run in the opposite order (grandchild first, parent last)

Constructor/destructor order

```
class First {
   public:
      First() { cout << “cons 1st\n”; }
     ~First() { cout << “dest 1st\n”; }
};

class Second: public First {
      Second() { cout << “cons 2nd\n”; }
     ~Second() { cout << “dest 2nd\n”; }
};

class Third: public Second {
      Third() { cout << “cons 3rd\n”; }
     ~Third() { cout << “dest 3rd\n”; }
};
```

```
int main()
{
   Third x;
}
```

resulting output would be:

```
cons 1st
cons 2nd
cons 3rd
dest 3rd
dest 2nd
dest 1st
```

Inherit public, protected, private

- we showed definition of form
  - class child: public parent {
- can also use private or protected, e.g.
  - class child: protected parent {
  - class child: private parent {
- sets the minimum privacy settings for inherited fields/methods
  - public: inherited field has same setting as in the parent class
  - protected: inherited public fields become protected in derived class (protected/private stay the same)
  - private: everything inherited becomes private in derived class

Example: queue inheriting from list

- will treat queues as specialization of a list

- list class (for a list of string)
  - a variety of typical methods:
    - insert at front
    - insert at back
    - remove from front
    - remove from back
    - print list

- queue class (inherits from list)
  - typical queue methods:
    - insert at back
    - remove from front
    - print list

list and queue class definitions

```
class list {
   private:
      // details don't matter here
   public:
      list();
     ~list();
      bool insertFront(string s);
      bool insertBack(string s);
      bool removeFront(string& s);
      bool removeBack(string& s;
      void print();
};
```

```
// inherit privately so people can't use our
// queue as if it was a list, denies them direct
// access to the list methods
class queue: private list {
   public:
      queue() {  } // uses list constructor
     ~queue() { } // uses list destructor

      bool enqueue(string s)
         { return insertBack(s); }
      bool dequeue(string &s)
         { return removeFront(s); }

      void print()
         { list::print(); } // redirects to list's print
};
```