Abstract Classes in C++
• Classes with no instances
• Classes with at least one deferred method
  — Deferred method defines argument signature, but defers method definition to subclasses
• Purpose: to define a common protocol for a set of concrete subclasses
• Pure virtual function: Deferred method in C++

Syntax of Abstract Classes
• Pure virtual member function is declared, but not defined in abstract class
• Function prototype in definition of abstract class is followed by =0 syntax

class Collection {
  // An iterator for collections of integers
  // Note that iterator is a pointer to a function on integers
  virtual Collection& do(void (*iterator) (int))=0;
  ...}

More on Abstract Classes
• Any class containing a pure virtual function is abstract, as well as all subclasses that do not give a definition (executable body) for this function
  — First class that overrides method is concrete (unless it has other pure virtual functions)
• Attempts to create instances will always result in errors, whether by static, automatic or dynamic allocation
  — Value parameters and data members disallowed

More on Abstract Classes
• Constructors should be protected
  — Useful to initialize inherited portions of concrete subclasses (assuming abstract class has data members requiring initialization)
• Use abstract class only to create subclasses
  — Very useful in statically typed language to define common protocol for set of concrete subclasses
  — Dynamically bound message must be defined in common ancestor of subclasses receiving message

Example of an Abstract Class
class DisplayObject {
  public:
    void rotate(int degrees)=0;
    void resize(double percent)=0;
    void remove(...)=0;
    ...}
  protected:
    Window* pDisplayWindow;
  ...}

Multiple Inheritance in C++
• C++ supports full multiple inheritance scheme
• Syntax: Specify multiple base classes in derivation list of subclass
  — Each base preceded by access level specification, defaulting to private
class DisplaySquare: public DisplayObject, public Square {
  ...
  // now DisplaySquare has all the members of
  // DisplayObject and Square
  ...}
More on Multiple Inheritance

- Order of base classes in derivation list determines structure of derived instances; otherwise, irrelevant
  - Example of DisplaySquare instance:

    DisplaySquare instance:
    - DisplayObject members
    - Square members
    - Noninherited members

Issues in Multiple Inheritance

- C++ must address three usual problems of multiple inheritance (e.g., name ambiguity, method search, common ancestor) and some additional problems (e.g., conversions to base classes, polymorphic base classes)
  - Conversion of derived class to any of its base classes is well defined
    - DisplaySquare instance can be automatically converted to either DisplayObject or Square

Automatic Conversions

- Example of automatic conversion to Square:
  
  ```
  void foobar(Square sq);
  ...
  DisplaySquare aDispSq;
  foobar(aDispSq); // aDispSq converted to Square
  ```

Conversions Involving Pointers

- Problem: Start of embedded instance target of conversion may not coincide with start of derived instance
  
  ```
  void foobar(Square* pSquare);
  ...
  DisplaySquare aDispSq;
  foobar(&aDispSq); // &aDispSq converted to Square*
  // however, Square instance does not start where aDispSq starts
  // (preceded by DisplayObject instance)
  ```

Conversions Involving Pointers

- Problem is solved by compiler adding appropriate offset to address of derived class instance
  
  ```
  void foobar(Square* pSquare);
  ...
  DisplaySquare aDispSq;
  foobar(&aDispSq); // compiler adds size of DisplayObject to address of aDispSq upon call
  // resulting pointer has address of Square instance embedded in aDispSq
  ```

Inheriting Virtual Functions

- Inherited functions must work with base class instance embedded in receiver
  - Pass appropriate portion of receiver to function
  - Similar problem to previous conversion; this time solution is complicated by presence of vptr and vtbl
    - If multiple base classes are polymorphic, derived class has multiple vtbls, one for each base class
    - Each embedded base class instance has a vptr to corresponding vtbl
    - Store appropriate offset in each vtbl
Name Conflicts

- Suppose that multiple base classes define data members or member functions with the same name
  - A classical problem in multiple inheritance
  - Solution is different depending on whether data member or member function is involved

- Data members: Must use scope operator with member identifier to resolve ambiguity

- Member functions: Member functions defined in different base classes overload each other
  - Calls may be disambiguated by argument lists

Examples of Name Conflicts

```cpp
class A {
  int x;
  ...
  void foo(int);
  int bar(int);
};

class B {
  char* x;
  ...
  void foo();
  void bar(int);
};

class C : public A, public B {
  void mem1()
  {
    x++;          // Error: x reference ambiguous
    A::x++;       // OK
    foo(3);       // OK, A::foo() is invoked
    bar(3);       // Overloaded call is ambiguous
    B::bar(3);    // OK, B::bar() is invoked
  }
};
```