Inheritance

- Building block of Object Oriented Programming
- Captures the notion of the “is a” relationship between objects of classes
- Sub-classing or sub-typing
  - Terminology: base and derived classes or super and sub classes
- Fundamental idea of extension – the derived class “inherits” all of the properties of its base class, and adds data and methods
  - All of the methods of the base class are methods of the derived class
  - All of the data of the base class is data of the derived class
- For practical code design, allows factoring of maximum commonality
  - Promotes code reuse without duplication
- C++ allows single inheritance (like Java), but also multiple inheritance
- Examples:
  - Organizational hierarchy – Employees, Managers, Officers
    - A Manager “is a” Employee
    - An Officer “is a” Employee
  - Shapes – Shape, Circle, Rectangle, Square
    - A Circle is a shape
    - A Rectangle is a shape
    - A Square is a Rectangle

Syntax of C++ Inheritance

- Base class is named after derived class name in class definition, separated by colon (i.e., colon is used in C++ where keyword “extends” is used in Java)
- Base class is qualified by access – public is real inheritance (more about private later)
- For example:
  ```cpp
  class Derived : public Base {
  ...
  }
  ```
- Think of this as “including” the class Base in the definition of Derived
  - As if the code of Base is duplicated in Derived
- Derived may override any data or methods it inherits from Base, re-defining the inherited definitions
  - To override, signatures of methods must match exactly
- Data or methods that are private in Base are hidden from Derived
  - I.e., deriving does not give any special access
  - Otherwise, entire access protection scheme would break down
  - This is a visibility issue – the private data and methods exist in the derived class, but just cannot be accessed by the code of the derived class
  - Some special access with protected – more later
- No limit to inheritance chain (assume all public):
  ```cpp
  class A { void foo(); }
  class B : public A { void bar(); }
  class C : public B { int x; void bar(); }
  class D : public C { void foo(); }
  ```
  - D has foo, bar, and x. It has re-defined foo, but inherits the rest. C has foo, bar, and x. It defines x and bar, but inherits foo. Etc.
Note that inherited methods or data may not come from immediate ancestor
There is no common base class implied for all classes like the Object class in Java

Constructors and Destructors

Constructors and destructors are not explicitly inherited, but they are implicitly used for construction and destruction of base class
  - For any Derived object, a Base constructor is called before the constructor of Derived
  - For any Derived object, the Base destructor is called after the destructor of Derived
  - If you think of a Derived as being built upon a Base, then of course the Base must be built first, and likewise the Derived must be torn down first

Since the base class constructors may require various arguments, we need a way to pass these values
  - The point at which the Base constructor is called is before beginning the Derived constructor, so we have Base initialization syntax:
    ```
    class A {
      A();
      A(int, double);
    };
    class B : public A {
      B() { }
      B(double d) : A(5, d) { }
    };
    ```
  - This is just like member initialization syntax
  - No such syntax is needed for destructors since no arguments are passed

Private “Inheritance”

If the keyword private is used to modify the base class in the derived class definition, then the inheritance relationship is not part of the interface

This is not real inheritance since the “is a” relationship is known only to the derived class itself, so is just an implementation detail

This is useful for some implementations
  - E.g., a Stack may be privately derived from a base of Vector
  - Convenient for the implementation, but probably don’t want all of the Vector interface available to Stack users

Protected access

Keyword protected gives special access for classes in an inheritance hierarchy
For classes not derived from the base, protected is equivalent to private
For classes derived from the base, protected is equivalent to public
So protected reveals parts of the class to any class that may derive
Reduces encapsulation and data hiding, but better than completely public
- Should be treated like public with regard to design since access is granted to
  unknown extent (*any* class that derives)
- Same rules apply when used on the derivation itself

**Pointers and References**
- The B “is a” A relationship means that a B object can be used wherever an A
  object is expected
- This holds for pointers and references
  ```
  class A { }
  class B : public A { }
  A a; B b;
  A *pa = & a; // Legal – always has been
  A *pa = & b; // Also legal – a B is an A
  B *pb = & a; // Illegal – we don’t have a whole B
  void foo(A *); foo(&a); foo(&b); // Legal
  void bar(A &); bar(a); bar(b); // Legal
  ```
- Allowing addresses of derived objects to be treated as addresses of base objects
  appears to lose information
- Okay if all we need to know is the base characteristics of the object, but what if
  the derived object has overridden function definitions?
  - Could result in surprises – if compiler sees whole object definition, it
    “knows” what the object is – static binding
  - But if it sees only the address, maybe the object there is really a derived
    type and then we would be missing information
  - Solution is to bind member functions dynamically

**Virtual Functions**
- Dynamically bound functions are called *virtual* functions
- Default behavior is static binding, to get dynamic binding you must have the
  `virtual` keyword on the function declaration in base class
  - Only required on earliest base class where we want dynamic binding
- This is the way all methods are resolved in Java (dynamic binding is default,
  keyword *final* is used for static binding in Java)
- Cost of dynamic binding is fairly trivial, but still too expensive if not needed (e.g.,
  an inline’d length function)
- Dynamic binding means the compiler arranges that the “right thing” is done at
  runtime, even if static analysis doesn’t show true type of object
- Allows for polymorphic treatment
  - A manager class can maintain a list of pointers, where pointer type is
    pointer to base.
  - Pointers may really be addresses of various derived objects
  - When methods are called, the correct method for the actual object is
    invoked if it is a virtual method
Example: window manager has Window pointers or references. Calls draw() for each one, and the correct draw() method is invoked for each object
- Replaces need for carrying around type fields and using switches on type
  - Compiler does this for us, and efficiently
- Typical implementation uses a virtual function table
  - Table has slots for each function declared virtual
  - Populated with addresses of the right functions
  - Only need one table for each class, not for each object of the class
  - Object only needs to associate with correct table
  - Overhead is one table per class, one pointer per object, extra level of indirection to call a function

Virtual Destructors
- Destructors can be virtual, too
- Always a good idea if any virtual functions (or if polymorphic treatment with pointers or references is expected)
- Remember that destructor calls begin with derived, and work back to base destructor (like unwinding of stack)
- If destructor is not virtual, and compiler only has a pointer or reference, it can only start this chain at the destructor for the type that it knows
  - This skips the calling of any destructors further down subclass chain
  - …which could be catastrophic
- A best practice: always define empty virtual destructor in base class

Pure Virtual Functions
- A member function may be declared pure virtual
- This means that no definition can or will be given – the function is a declaration only, so is a placeholder (like an abstract method in Java)
- Rather than a keyword, the syntax is to “initialize” the function with the null pointer, i.e., the value zero
  
  ```
  class A {
    virtual void foo() = 0;
  }
  ```
  - A class containing (or inheriting without redefining) a pure virtual is called an abstract class
  - No object of an abstract class can be instantiated
    - It’s abstract, and we can’t have a concrete instance
    - If we could, and attempted to call the pure virtual function…
  - A pure virtual holds a place in the virtual function table
  - Abstract characteristic of class continues down inheritance until all pure virtuals are defined
  - Useful for factoring out common abstractions, where objects are too incomplete to function on their own
  - Useful for abstraction of objects kept by container classes
  - Similar to Java interface
Copy Constructors and Assignment
- Copy constructors and assignment not inherited
- However, default copy constructor and assignment do **member-wise** initialization and assignment
  - This applies to the base class as well
  - So derived class only has to take care of itself (the part not in the base)
- Consistent with object oriented philosophy of self containment

Multiple Inheritance
- A class can be derived from multiple base classes
- Expresses several “is a” relationships
- Syntax is to list base classes, separated by commas (and each qualified with `public`, `private`)
  - Similar to Java multiple interfaces
- Difficulty comes if two of the bases are in turn derived from a common base
  - Do we want multiple copies of the common base or do we want one copy of the common base?
  - If only one copy desired, base inheritance should be declared virtual
  - For example,
    ```
    class B : virtual public A { };
    class C : virtual public A { };
    class D : public B, public C;
    ```
    Then objects of type D have a single A part
- Multiple inheritance results in a lattice diagram of inheritance
- Ambiguities may result
  - Can usually be resolved with scoping operator
  - Sometimes we want redefinition to invoke overridden functions from each super class (sum of parts)
- Can be argued that multiple inheritance is unnecessary
  - In any case, should be used sparingly as it complicates design