CPSC 5011: Object-Oriented Concepts

Lecture 8: Inheritance and Polymorphism

Inheritance

- Of Interface
  - Type extension (PUBLIC)
  - Child type IS-A (extension of) parent type
  - Child type retains, supports and, possibly extends parent interface
- Of Implementation
  - Code reuse (PRIVATE)
    - *Already defined, debugged, tested*
  - Child class inherits and uses parent code

Accessibility

- Child class has access to
  - All public data and functionality of parent
  - All protected data and functionality of parent
- Application programmer has access to
  - Parent class object
    - All public data and functionality of parent
  - Child class object
    - All public data and functionality of parent
    - All public data and functionality of child

Java : Inheritance Example

```
class JParent {
   public      boolean  decideForAll() {…}
   protected   boolean  decideForDescendants() {…}
   private     boolean  decideForClassOnly() {…}
   protected   boolean  forDescendants;
   private     boolean  forClassOnly;
      // not accessible in Child class
   …
}
class JChild extends JParent {
// only public extensibility for Java inheritance
   public JChild(int x)  { super(x); }
   public void Childfn() { … }
   public boolean  decideForAll() {
      if (super.forDescendants)
        return super.decideForAll();
      if (forDescendants)
        return decideForDescendants();
      if (forClassOnly)
        return false;
   }
   protected   boolean  forDescendants;
   private     boolean  forClassOnly;
   …
}
```

```
JParent baseObj = new JParent();
baseObj.decideForAll();

JChild derivedObj = new JChild();
derivedObj.decideForAll();

baseObj = derivedObj;
// SUBSTITUTABILITY

baseObj.decideForAll();

baseObj = new JChild(22);

// constructor calls explicit;
// base constructor fires first
```

C++ : Inheritance Example

```
class CParent {
public:         // accessible by all
   bool    decideForAll();
protected:      // accessible by Child class but not external world
   bool    forDescendants;
   bool    decideForDescendants();
private:        // not accessible in Child class
   bool    forClassOnly;
   bool    decideForClassOnly();
};

class CChild: public CParent {
// 3 types of extensibility for C++ inheritance
// inheritance access: private (default), protected, OR public (most common)
public:
   CChild(int  x): CParent(x) { … }
   bool   decideForAll() {
      if (CParent::forDescendants)   return CParent::decideForAll();
      if (forDescendants)            return decideForDescendants();
      if (forClassOnly)              return false;
   }
protected:
   bool    forDescendants;
private:
   bool    forClassOnly;
};
```

C++ : Private Inheritance

PRIVATE INHERITANCE

- Default mechanism but not commonly used
- Suppresses all of inherited interface
  - No is-a relation => DESIGN CAREFULLY
  - Impacts use by application programmer
- Cuts off inheritance hierarchy
  - Impacts use by child class designer
- Code reuse (INTERNAL UTILITY ONLY)

```
class Child: Parent{  …  };     // default private inheritance
void passValue(Parent);         // parameter is parent object
passValue(pObj);                // ok
passValue(cObj);
```

C++ : Protected Inheritance

PROTECTED INHERITANCE

- Must use protected qualifier in definition
- Public interface suppressed
  - Impacts use by application programmer
- Protected interface passed onto descendants

```
class Child: protected Parent { … };
class Grand: Child {   // private inheritance by default
   …
   public:
      void somefn() {  parentFn(); … }
};
```

~~cObj.parentFn();~~

C++ : Public Inheritance

PUBLIC INHERITANCE

- Must use public qualifier in definition
- Most commonly used means of inheritance
- Access as defined in parent class preserved
  - Public remains public
  - Protected remains protected
- Child class may redefine access for specified functions
  - May curtail inherited functionality for future descendants
    - override protected function and declare it private
  - May suppress inherited functionality
    - override public function and declare it protected or private

Java / C++ : Overloaded Constructors

```
// Java Example
class JParent {
   public  JParent() { … }
   public  JParent(int x) { … }
   private int old;
   …
}

class JChild extends JParent {
   public JChild() { … }
   public JChild(int x)
   { super(x);  this();  … }
   public JChild(int x, float y)
   { super(x);  data = y; … }
   private float data;
   …
}
```

```
// C++ Example
class CParent {
// default accessibility is private:
// not accessible in Child class
public:        // accessible by all
   CParent();
   CParent(int);
private:
   int old;
};

class CChild: public CParent {
public:
   // default base constructor
   CChild();

   // specify to compiler which parent
   // constructor to invoke
   CChild(int x): CParent(x) { … }

   CChild(int x, float y)
           : CParent(x), data(y) { … }
private:
   float data;
};
```

C++ : Is-A Relation

Every object of a publicly derived class is ALSO an object of the base class

- Utility restricted to base class interface
- C++ Child object assigned to parent: SLICED
- What about Java?
  - ASSIGNMENT COPIES REFERENCE => no slicing

```
Parent pObj;
Child  cObj;
pObj.parentFn();
cObj.parentFn();
cObj.childFn();
pObj = cObj;  // sliced: ONLY PARENT
              // INTERFACE
pObj.parentFn();
```

Substitutability

- Derived class objects stands in for base class object
- Not a symmetric relation
  - Parent cannot stand in for child. Why?

```
Parent * pPtr;
Child  * cPtr;
pPtr = new Parent;
cPtr = new Child;
pPtr->parentFn();
delete pPtr;
pPtr = cPtr;
pPtr->parentFn();
```

Inheritance : Language Differences

- Java supports only public inheritance
  - does not allow direct suppression of inherited functionality
- C++ offers public, protected, and private inheritance
  - only public inheritance is typically used
  - with protected inheritance, all inherited public functionality is demoted to protected accessibility
  - with private inheritance, all inherited public and protected functionality is demoted to private accessibility

  => application programmer has less accessibility via a derived object
- C++ allows class designers to directly suppress inherited functionality by changing the accessibility of inherited class methods on an individual basis

C++ : Direct Is-A Suppression

Purity of relation not guaranteed

- C++ may suppress directly by overriding with a private method

```
class Child: public Parent {
public:
   …
private:
   // private is default accessibility
   void parentFn() { // now private => suppressed  }
};

Parent pObj;
Child  cObj;

pObj.parentFn();
cObj.parentFn();
```

Inheritance : Advantages

- Code reuse
  - Reduce development cost (& time)
  - Usually better maintenance
    - Less cut & paste programming
  - Parent class presumed stable
    - Already designed, implemented, debugged and tested
- Type extension
  - Application Programmer familiar with parent
  - Substitutability
    - Polymorphism – run-time selection of functions
    - New type can be added without breaking application code

Inheritance : Disadvantages

- Increased Coupling
  - Child tightly coupled to parent
- Decreased Cohesion
  - Type definition spread across inheritance hierarchy
- Fixed Relationship
- Overhead
  - Child absorbs overhead of parent component even if not used
- Maintenance
  - Cost highly dependent on design

Has-A versus Is-A

- Has-a encapsulates and controls subObject(s)
  - Design variability in cardinality, association, lifetime and ownership
  - Interfaces may, but need not, be echoed
- Is-a implies a strong type dependency
  - Child object may stand in for parent object
  - Polymorphism, and heterogeneous collections, supported through inheritance and difficult to implement otherwise
- Is-a imperative to reuse functionality
  - Common interface
  - Extensibility promoted
  - Overhead is fixed as is cardinality, ownership, lifetime and association

Composite Principle

Use composition in preference to inheritance.

- Composite Principles states practitioners' preference for composition over inheritance – Why?
- Composition more flexible and offers more control over internal design than inheritance
- But remember, composition does NOT provide
  - Built-in subtype checking
  - Polymorphism
  - Support for heterogeneous collections
  - Type extensibility

Principle of Least Knowledge

Every object should assume the minimum possible about the structure and properties of other objects.

- Promotes low coupling
- When classes interact, in any relationship
  - Class design should not be dependent on private implementation details of any other class
- With clear documentation, deliberate design identifies relationships and their consequential effects

Open-Closed Principle (OCP)

A class should be open for extension and closed for modification.

- Inheritance is an attractive design option for
  - Class hierarchy that relies on implicit subtype selection to distinguish appropriate functionality
  - Substitutability
  - Heterogeneous collections
  - Type extensibility
- A good inheritance design adheres to OCP
  - Individual classes preserved
  - Type extensions are seamless
- OCP promotes software maintainability

Three Forms of Polymorphism

- Overloading aka Ad Hoc Polymorphism
  - Allows multiple function definitions with same name
  - Compiler uses parameter type(s) to resolve function calls
- Generics aka Parametric Polymorphism
  - Supports 'type-less' definition of a class or a function
  - Application programmer can later supply type
  - Compiler generates version of generic class (or function) with that type
- Subtyping aka Inclusion
  - Describes design of class hierarchy where descendant classes (re)define, augment, or modify inherited functionality
  - Descendant classes are dependent on the base class interface
  - Dynamic binding expected

C++ : Overloaded Functions

```
void reset() {
    for (int k = 0; k < size; k++) A[k] = 0.0;
}
void reset(double value) {
    for (int k = 0; k < size; k++) A[k] = value;
}
void reset(bool op, int factor) {
    if (op)
        for (int k = 0; k < size; k++) A[k] *= factor;
    else
        for (int k = 0; k < size; k++) A[k] += factor;
}
```

C++ : Generics

```
void swap(int &x, int &y) {
    int hold = x;
    x = y;
    y = hold;
}
void swap(float &x, float &y) {
    float hold = x;
    x = y;
    y = hold;
}
template <typename T>
void swap(T &x, T &y) {
    T hold = x;
    x = y;
    y = hold;
}
```

Third Form of Polymorphism

- Overloading
  - Allows multiple function definitions with same name
- Generics
  - ‘type-less’ definition of a class or a function
- **Subtyping depends on dynamic binding**
  - Associate function call resolution with subtype
  - POSTPONE function call resolution until run-time

Static versus Dynamic Binding

- Static binding
  - compiler resolves function calls
  - translates each function invocation into a direct jump
  - efficient but rigid
- Dynamic binding
  - compiler does not resolve a function call at compile-time.
  - extra instructions generated
  - at run-time, the appropriate function address extracted from a jump table
  - flexible but costly

Binding Design Choices

- Java
  - All functions dynamically bound (EXPENSIVE!)
- C++
  - Static binding by default (efficient!)
  - Dynamic binding with keyword ‘virtual’
- Static binding
  - Reasonable when function choice will not vary
- Dynamic binding
  - Reasonable when subtype affects function choice
  - Heterogeneous collections

C++ : Static Binding by Default

```
// How to select dynamic binding in C++?
//  cannot use objects directly:
//        memory allocated => (sub)type fixed
//    access objects indirectly: pointers!
//        (sub)type of object addressed may vary

Parent * pPtr;      // stack allocated pointer
Child  * cPtr;      // stack allocated pointer

pPtr = new Child;   // pointer holds address of
                    // heap-allocated Child object

pPtr->parentFn();   // parent class implementation
//  C++ -- static binding is default

// EXPLICIT DESIGN for C++ dynamic binding
//    #1 must define functions as virtual in base class
//    #2 object access must be indirect (via pointer)
```

C++ : Dynamic Binding by Choice

```
// C++ :  DYNAMIC BINDING by choice
// keyword virtual used to signal dynamic binding
class Parent
{   …
    public:
       virtual void parentFn();
};


class Child: public Parent
{   …
    public:
       void parentFn() override; // override inherited behavior
       void childFn();           // expanded interface
};

…
Parent * pPtr;   // stack allocated pointer
Child  * cPtr;   // stack allocated pointer

pPtr = new Child;     // pointer holds address of
                      // heap-allocated Child object
pPtr->parentFn();     // child class implementation
//  C++ -- dynamic binding with virtual functions AND access via pointers
```

Class design
- Add keyword ‘virtual’ to method in base class

Application code
- Use base class pointers

C++ : Dynamic Binding Issues

```
// Java only dynamic binding: consistent, readable

// consistency problem in C++
//    increased SOFTWARE COMPLEXITY  => confusion
//    both static & dynamic binding

Parent *  pPtr;   // stack allocated pointer
Child  *  cPtr;   // stack allocated pointer
…
pPtr = cPtr;  // okay to 'upcast'
              // parent pPtr holds address of Child object

// cannot easily discern effect of function invocation
pPtr-> parentFn();
// #1 parent function if static binding (non-virtual)
//      even if parentFn() overridden
// #2 child function if dynamic (virtual function)

// to get Parent behavior (even if virtual)
Parent    p = *pPtr; // object 'sliced'
p.parentFn();         // static binding
```

Static versus Dynamic Binding

- Static binding translates function calls directly into jump statements
  - ⇒ Function invoked at the point of call cannot vary
  - ⇒ No run-time overhead
  - ⇒ No run-time flexibility
- Dynamic binding postpones function call resolution until run-time
  - ⇒ Function invoke at the point of call can vary
  - ⇒ Run-time overhead
  - ⇒ Run-time flexibility
  - ⇒ Supports polymorphism and heterogeneous collections