Supplementary Materials to Lecture 3

**Polymorphism**

- (Greek) "Occurring in many forms"
  - The ability of *one thing* to appear in many forms depending on context and also the ability of *different things* to appear the same in a certain context
  - Four different types of polymorphism [1]
Universal Polymorphism
- The purest form of polymorphism.
- The same code may be applied to an infinite number of types. A variable/value can have different type.

Parametric Polymorphism
- Same implementation can serve multiple types.
- E.g. a container can be parameterized with the element type.
- (Bounded polymorphism lets you constrain the type parameter to be of a special subtype.)
Polymorphism

Universal Polymorphism

Ad Hoc Polymorphism

Parametric

Inclusion

Overloading

Coercion

Parametric Polymorphism

- Type independent structures and operations

Example:
Homogeneous but polymorphic stacks.

Stack \([T]\), for any type \(T\)

```c
void push(T item) { ... }
T pop() { ... };
```

Parametric Polymorphism and Type Inference (example in SML)

```sml
map : forall type T and type S, (T → S) × List[T] → List[S]

/* map applies ‘f’ to every element of list l */
fun map (f, l) is
    if null (l) then
        nil
    else
        cons ( f (head l), map (f, tail l) )
end
```
Polymorphism

Universal Polymorphism

Ad Hoc Polymorphism

Parametric

Inclusion

Overloading

Coercion

Simulation in C++ Example (function)

class A {
    public: void f(){...
    ...
}
class B {
    public: void f(){...
    ...
}

template <class T>
void test(T& v){
    v.f();
}
A a; B b;
test(a);
test(b);

C++ Example (class)

template<class ElementType> class Archive {
    ... virtual void insert(const ElementType *newElement)....
};
class Employee {...
Archive <Employee> employeeArchive;
Polymorphism

Universal Polymorphism

Ad Hoc Polymorphism

Parametric
Inclusion
Overloading
Coercion

UML Example

Employee Archive

+insert(newElement : ElementType)

EmployeeArchive <<bind>> (Employee)

Polymorphism

Universal Polymorphism

Ad Hoc Polymorphism

Parametric
Inclusion
Overloading
Coercion

UML Example

Observer

Subject

Observer

PersonData

Subject

Observer

PersonForm

Observer

Parameterized collaboration

Applied parameterized collaboration
Polymorphism

Inclusion Polymorphism

- A variable of a base class (or interface) can refer to objects of the base class and of subtypes (inclusion of subtypes). The variable is said to be of polymorphic type.
- (UML [2]: "Indicates an operation whose implementation may be supplied by a descendant class.")

Java Example

```java
class A {
    public void f(){...}
    ...
}
class B extends A {
    public void f(){...} // overriding
    ...
}
class Tester{
    public void someF(A a){ a.f();}
    ...
}
```
Polymorphism

Universal Polymorphism

Ad Hoc Polymorphism

Parametric Inclusion Overloading Coercion

Ad Hoc Polymorphism

• This polymorphism is considered less pure or not real polymorphism.

Overloading

• Groups operations (operators) under a common name; Creating the illusion that there is one operation that can deal with parameters of many different kinds. In reality there is a set of monomorphic functions that is used!
Polymorphism

Universal Polymorphism

Ad Hoc Polymorphism

Parametric

Inclusion

Overloading

Coercion

C++ Example

class A {
    public: void f(){...}
    ...}
class B {
    public: void f(){...}
    ...}
void test(A& v){ v.f();}
void test(B& v){ v.f();}

Coercion

• Coercion is based on code conversion; it allows an argument to be converted to the type expected by a function or operator.
• E.g. $2+1.2$ here $2$ is automatically converted to a float $(2.0)$: $2.0 + 1.2$
**C++ Templates**

- **Classes**
  ```cpp
template < class T1, class T2>
class clase1 {
    T1 atr1;
    template < class T3 >
    T3 func ( T1 a, T2 b, int l);
};

clas1<int, float> C1;
```

- **Functions**
  ```cpp
template < class T >
T max (T a, T b) {
    return (a > b)? a: b;
}
```

```
floa a, b, c;
int d, e, f;
a = max ( b, c);
d = max ( e, f);
```

---

**A templated (generic) list class**

```cpp
template <class ListElementType>
class List {
public:
    List();
    void insert(const ListElementType & elem);
    bool first(ListElementType & elem);
    bool next(ListElementType & elem);
private:
    struct Node {
        ListElementType elem;
        Node* next;
    };
    Node* head;
    Node* current;
};
```
What happens when....

- When `double` is passed to the generic list class
  
  ```cpp
  List <double> doubleList;
  ```

- C++ conceptually replaces every occurrence of `ListElementType` with `double`:
  
  ```cpp
  // ...
  void insert(const double & elem);
  bool first(double & elem);
  bool next(double & elem);
  ```

  ```cpp
  private:
  struct Node {
    double elem;
  // ...
  ```

---

C++ Standard Template Library

- Goal: represent algorithms in as general form as possible without compromising efficiency

- Extensive use of templates

- Only uses static binding (an inlining)

- Use of *iterators* for decoupling algorithms from containers

- Iterator: abstraction of pointers
STL Organization

- Containers
  - vector, deque, list, set, map, ...
- Algorithms
  - for_each, find, transform, sort
- Iterators
  - forward_iterator, reverse_iterator, istream_iterator, ...
- Function Objects
  - plus, equal, logical_and, project1
- Allocators

Forward Iterator

```cpp
int main(int argc, char **argv) {
    vector<string> args(argv, argv + argc);

    vector<string>::iterator iter = args.begin();
    for ( ; iter != args.end(); ++iter )
        cout << *iter << " ",
    cout << endl;
}
Reverse Iterator

```cpp
vector<string>::reverse_iterator iter = args.rbegin();
for (; iter != args.rend(); ++iter )
    cout << *iter << " ";
    cout << endl;
```

How to design a generic “qsort”?

“qsort”, Quick Sort, takes an array of **integers** and sorts them into **ascending** order.

“qsort”, Quick Sort, takes a bunch of **objects** and sorts them into a specific order.
Example: Polymorphic Container

- Since *Object* is the superclass of all other, we can use it to implement polymorphic (heterogenous) containers.

```java
public class Stack {
    private Object[] data; private Object ptr = -1;
    static int default_size = 100;
    public Stack() { data = new Object[default_size]; }
    public void push(Object value) {
        ptr++; data[ptr] = value;
    }
    public Object pop() {
        ptr--; return (data[ptr+1]);
    }
}
```

```java
Stack s = new Stack();
Rectangle rect = new Rectangle();
s.push(rect); s.push(new Integer(5)); s.push("abc");
```

Polymorphic Container (Cont’d)

- But how about "popping" those objects out of the polymorphic stack?

```java
String str = s.pop(); // an error!
```

Must use

```java
Object o = s.pop();
String str = (String) o;
```

Or **Downcasting**

```java
String str = (String) s.pop(); // casting
```
Casting Between Classes

- Casting: conversion of a class A into another class B to force class B’s behavior on class A
- Two types: (Cannot cast between sibling classes)
  - **Widening**: from subclass to superclass, (Upcasting)
    No explicit cast is required
  - **Narrowing**: from superclass to subclass, (Downcasting)
    No guaranteed reliability.
    An explicit cast is required.
    \[(classNameOfObjectBeingCastTo)ObjectBeingCast\]

- Casts in Java are checked at runtime. Illegal casts raise an exception. (Unlike C/C++)

---

Cast and Instanceof Operators

- We need to perform a cast to recover the class a particular object refers to. In the previous example:
  \[\text{string} = (\text{String}) \text{s.pop}();\]
- If we don’t know the type of the object, we can use the `instanceof` operator to avoid runtime error from an illegal cast. *(runtime typechecking)*
  \[
  \text{Object object} = \text{s.pop}();
  \text{if (object instanceof String)} \{
  \text{string} = (\text{String}) \text{s.pop}();
  \}
  ...
  \]
Simulating Constrained Parametric Polymorphism in C

C Function Pointers

- **Defining function types**
  
  The following declare a variable `fn` to be a *pointer to a function* taking and returning a *double*
  
  ```c
  double (*fn)(double);
  ```
  
  Note that the parentheses are required here. The alternative
  
  ```c
  double *fn(double);
  ```
  
  would declare `fn` as a function returning a pointer to a *double*.

- In most cases, however, it makes sense to use `typedef` to name the class of functions
  
  ```c
  typedef double (*RealFn)(double);
  RealFn fn;
  ```
QuickSort Algorithm

Void qsort (int data[ ], int n) {
    int p_index, n1, n2;
    if (n > 1) {
        partition (data, n, p_index);
        n1 = p_index; n2 = n - n1 - 1;
        qsort(data, n1);
        qsort(data + p_index + 1, n2);
    }
}

qsort

• (close)Treat the array to sort as the pointer to the address of the first element, and pass the size of each element (elementSize) in addition to the number of elements in the array (n).
  void qsort(void *array, int n, int elementSize); /*close*/ <
• void qsort(element list[],...) typedef int element;
• But
  There is no way for a general qsort function to know how to compare two values. Only the client knows how to compare two values.
• So the type for generic qsort:
  void qsort(void *array, int n, int elementSize, CmpFn cmp);
    where
    typedef int (*CmpFn)(const void *e1, const void *e2);
Calling qsort

- To call \texttt{qsort} on an integer array of size \texttt{n}, we would then call

\begin{verbatim}
qsort(intarray, n, sizeof(int), IntPointerCompare);
\end{verbatim}

\begin{verbatim}
where \texttt{int iarray[100]; int n =100; int *intarray; intarray = &iarray[0];}
\end{verbatim}

\begin{verbatim}
int IntPointerCompare(const void *e1, const void *e2) {
    int n1, n2;
    n1 = *((int *) e1);
    n2 = *((int *) e2);
    if (n1 == n2) return (0);
    if (n1 < n2) return (-1);
    return (1);
}
\end{verbatim}
Iterator Design Pattern

Rich Klein
February 10, 2003

Preliminaries

• What is a container?
  – a class that organizes and stores data
  – examples: list, vector, tree, set, map
  – each container organizes data differently

• What is an iterator?
  – a pointer used to access the data elements in the container
Sample Navigation of A List

Sample Navigation Of A Tree
What Does An Iterator Do?

- An iterator object simplifies navigation. It can:
  - point to the first element
  - point to the current element
  - point to the next element
  - tell you when you have reached the end

- Example:

```java
for ( i.First(); !i.IsDone(); i.Next() ) {
  ...
}
```

Why Have An Iterator?

- Why not just build the navigation into the member functions of the container?
  - you may want different types of traversals to be available (e.g., backwards and forwards), which would clutter the container
  - the container builders may not know the different types of traversals clients may need in future
  - clients may need more than one traversal going at the same time
Why Have An Iterator (cont’d)

• You want to abstract navigation details from the client

• You may not even want client to know the type of container being used - you might want to change it someday

Iterator Design Pattern

• Key Idea
  – to take the responsibility for access and traversal out of the container object and put it into an iterator object

• Iterators must behave differently for different containers, and so…

• We’ll give you the right iterator when you need it, and you don’t have to worry how we store our data
C++: Class Container

template<class Item>
class Container
{
  public:
    virtual void addItem( const Item &item ) = 0;
    virtual void removeItem( const Item &item ) = 0;
    virtual Iterator<Item> *createIterator( ) = 0;
    . . . .
};
C++: Class Iterator

template<class Item>
class Iterator
{
    public:
        virtual void First( ) = 0;
        virtual void Next( ) = 0;
        virtual bool IsDone( ) = 0;
        virtual Item & CurrentItem( ) = 0;
};

C++: Class List

template<class Item>
class List : public Container<Item>
{
    public:
        Iterator<Item> *CreateIterator( )
        {
            return new ListIterator<Item>( );
        }
    (implement AddItem() and RemoveItem() and other behaviors)
    ...
};
C++: Class ListIterator

In similar fashion, ListIterator would implement the abstract member functions of Iterator to work properly on a list:

- First( )
- Next( )
- IsDone( )
- CurrentItem( )

C++: Function In ClientClass

When called by a list or tree object, this function will create the proper iterator (list iterator or tree iterator) and will print the elements of the list or tree without knowing whether it is a list or tree:

```cpp
void PrintElements( Container<Item> * c )
{
    Iterator<Item> *it = c -> CreateIterator( );
    it->First( );
    while ( ! (it->IsDone( ) ) )
    {
        cout << it->Next( );
    }
    delete it;
}
```
Summary

• Use Iterator pattern:
  – to access a data structure’s contents without exposing the structure’s representation

  – to support multiple traversals of a data structure

  – to provide a uniform interface for traversing different types of data structures (polymorphic iteration)

Coupling

Weak coupling means:
- few dependencies
- weak dependencies
Weak coupling

Aims of weak coupling:
• Need to understand only one module at a time
• Change in a module does not lead to change in many other modules
• Necessary changes are obvious

Coupling: an example

Scenario:
• button press on Floor
• Floor asks LiftController for an ArrayList() of Lifts
• Floor searches through ArrayList() to find Lift to send
• Floor tells Lift to come
Coupling: an example

In class Floor:

```java
void buttonPress()
{
    ArrayList lifts = controller.getLifts();
    // search through ArrayList to find idle lift
    ...
    // send Lift up or down to this floor
    lift.moveTo(this.floorNumber());
}
```

Coupling example: problems

What if...

- Implementation changes:
  - Controller wants to use LinkedList() instead of ArrayList()
  - Lift interface changes: moveTo needs another parameter
Coupling: an example

Scenario:
- button press on Floor
- Floor tells LiftController about button press
- LiftController finds Lift to send and sends it on its way

In class Floor:
```java
void buttonPress()
{
    controller.buttonPressed(this.floorNumber());
}
```

In class LiftController:
```java
void buttonPressed(int floorNumber)
{
    // search through LinkedList() to find idle lift
    ...
    // send Lift up or down to this floor
    lift.moveTo(floorNumber);
}
```
Object Composition: Inheritance and Composition

Aggregation

• A special relation used to indicate the idea of *owns* or *whole-part* or *has-a.*
  – Usually not named

• Two flavours:
  1. Weak ♦ an object may belong *weakly* to several containers
  2. Strong ♠ an object may *strongly* be a part of at most one container.
     – “When container dies, subparts die too.”
Composition

• Composition is a *strong* type of aggregation indicating that the part object only exist as a part of the assembly class. The part object of an aggregation will be deleted if the assembly object is deleted. An object may be part of only one composite at a time.

*Composition can be represented in to different ways:*

OO Reuse

• Inheritance
  – White-box
• Composition
  – Black-box
  – Via Delegation
Delegation

Delegation of client calls to B::m(int) to A::m(int)

```java
class A {
    ...
    int m(int a) { ... }
}
class B {
    private A delegate;
    ...
    public int m(int a) {
        return delegate.m(a);
    }
}
```

Analyzing Objects with Reflection

- **reflection**: mechanism for discovering information about, or manipulating, objects and classes at runtime

- possible uses of reflection:
  - print / display information on screen about an object
  - create an instance of a class whose name isn’t known until runtime
  - call a arbitrary method by stating its name
Reflection in Java: Class

Class

• An object of the java.lang.Class class is a representation of a Java type

• With a Class object, you can:
  – ask for information about the class
  – learn about the fields, methods of objects of that class
  – construct instances (objects) of the class
  – discover the superclasses, subclasses, implemented interfaces, etc. of the class

Useful Methods in Class

Class

• public static Class forName(String className)
  Returns a Class object that represents the class with the given name.

• public String getName()
  Returns the full name of this Class object, such as “java.awt.Rectangle”.

• public int getModifiers()
  Returns a set of flags with information about the class, such as whether it is abstract, whether it is an interface, et cetera.

• public Object newInstance()
  Returns a new instance of the type represented by this Class object. Assumes a no-argument constructor.
More Methods in Class `Class`

- `public Class[] getClasses()`
  Returns an array of all inner classes inside this class.

- `public Constructor getConstructor(Class[] params)`
- `public Constructor[] getConstructors()`
- `public Field getField(String name)`
- `public Field[] getFields()`
- `public Method getMethod(String name, Class[] params)`
- `public Method[] getMethods()`
  Return information about the constructors, fields (instance variables), and methods inside this class.

- `public Package getPackage()`
- `public Class getSuperClass()`
  Return information about the class’s package or parent.

Programming with `Class`

- **in a `toString` method:**
  ```java
  public String toString() {
      return "My type is " + getClass().getName();
  }
  ```

- **to print names of all methods in a class:**
  ```java
  public void printMethods() {
      Class claz = getClass();
      Method[] methods = claz.getMethods();
      for (int ii = 0; ii < methods.length; ii++)
          System.out.println(methods[ii]);
  }
  ```
Using Other Reflection Classes
(found in java.lang.reflect package)

• Field
  - public Object get(Object obj)
  - public <type> get<type>(Object obj)
  - public void set(Object obj, Object value)
  - public void set<type>(Object obj, <type> value)

• Constructor
  - public Object newInstance(Object[] args)

• Method
  - public Object invoke(Object obj, Object[] args)

Getting the Class You Want

• Every class has a corresponding Class object that you can get by:
  – writing class’s name followed by .class (e.g. Vector.class)
  – calling .getClass() on an instance of that type (e.g. new Vector().getClass())
  – calling Class.forName(className) with that type’s name as a String (e.g. Class.forName("java.util.Vector"))
  – load a class from an arbitrary class file by using a ClassLoader object:
    ClassLoader loader = ClassLoader.getSystemClassLoader();
    loader.
Reflection Usage Example

```java
Class cl = Class.forName("java.awt.Rectangle");
Class[] paramTypes = new Class[] {
    Integer.TYPE, Integer.TYPE};
Constructor ctor = cl.getConstructor(paramTypes);

Object[] ctArgs = new Object[] {
    new Integer(20), new Integer(40)};
Object rect = ctor.newInstance(ctArgs);

Method meth = cl.getMethod("getWidth", null);
Object width = meth.invoke(rect, null);

System.out.println("Object is "+ rect);
System.out.println("Width is "+ width);

OUTPUT:
Object is java.awt.Rectangle[x=0,y=0,width=20,height=40]
Width is 20.0
```

Java Core Collection Interfaces

Collection interfaces allow collections to be manipulated independently of the details of their representation