Composition of Classes

Composition: an organized collection of components interacting to achieve a coherent, common behavior.

Why compose classes?

Permits a “lego block” approach to design and implementation:

- Each object captures one reusable concept.
- Composition conveys design intent clearly.

Improves readability of code.

Promotes reuse of existing implementation components.

Simplifies propagation of change throughout a design or an implementation.

Composition by Association

Association (acquaintance)

Example: a database object may be associated with a file stream object.

The database object is “acquainted” with the file stream and may use its public interface to accomplish certain tasks.

Acquaintance may be one-way or two-way.

Association is managed by having a “handle” on the other object.

Associated objects have independent existence (as opposed to one being a sub-part of the other).

Association is generally established dynamically (at run-time), although the design of one of the classes must make a provision for creating and maintaining the association.

Sometimes referred to as the “knows-a” relationship.

A Simple Association

```
class DisplayableNumber {
private:
   int      Count;
   ostream* Out;
public:
   DisplayableNumber(int InitCount = 0, ostream& Where = cout);
   void ShowIn(ostream& setOut);
   void Show() const;
   void Reset(int newValue);
   int Value() const;
};
```

```
void DisplayableNumber::ShowIn(ostream& setOut) {
   Out = &setOut;
}

void DisplayableNumber::Show() const {
   *Out << Count << endl;
}
```

Composition by Aggregation

Aggregation (containment)

Example: a LinkList object contains a Head pointer to the first element of a linked list of Node objects, which are only created and used within the context of a LinkList object.

The objects do not have independent existence; one object is a component or sub-part of the other object.

Aggregation is generally established within the class definition. However, the connection may be established by pointers whose values are not determined until run-time.

Sometimes referred to as the “has-a” relationship.

A Simple Aggregation

```
class Array {        // static-sized array encapsulation
private:
   int   Capacity;  // maximum number of elements list can hold
   int   Usage;     // number of elements list currently holds
   Item* List;      // the list

   void ShiftTailUp(int Start);
   void ShiftTailDown(int Start);
   void Swap(Item& First, Item& Second);

public:
   Array();                              // empty list of size zero
   Array(int initCapacity);              // empty list of size initCapacity
   Array(int initCapacity, Item Value);  // list of size initCapacity,
                                         //    each cell stores Value
   Array(const Array& oldArray);         // copy constructor

   int  getCapacity() const;               // retrieve Capacity
   int  getUsage() const;                  //          Usage
   bool isFull() const;                    // ask if List is full
   bool isEmpty() const;                   //             or empty
// . . . continues . . .
```

A Simple Aggregation

```
// . . . continued
   bool InsertAtTail(Item newValue);      // insert newValue at tail of list
   bool InsertAtIndex(Item newValue, int Idx);// insert newValue at specified
                                              //    position in List

   bool DeleteAtIndex(int Idx);       // delete element at given index
   bool DeleteValue(Item Value);      // delete all copies of Value in list

   Item Retrieve(int Idx) const;      // retrieve value at given index
   int  FindValue(Item Value) const;  // find index of first occurrence of
                                      //    given value

   void Clear();                      // clear list to be empty, size zero
   void Reverse();                    // reverse order of list elements

   ~Array();                          // destroy list (deallocate memory)
};
```