

Composition of Classes

Composition by Association

A Simple Association

Composition by Aggregation

A Simple Aggregation

A Simple Aggregation

Composition for Flexibility

Communicating Objects

Communication: inter-Object Communication

Communication: Passing an Object

Communication: Returning an Object

Composition Example: Simple Array of Objects

Aggregation Example: Fancy Array of Objects

Aggregation Example: Fancy Array of Objects

Aggregation and Constructor Sequencing

Aggregation and Destructor Sequencing

Anonymous Objects and Returned Objects

One Last Aggregation Puzzle

Typical Aggregation Sequencing

Controlling Aggregation Sequencing

Different Ways to Communicate

Characteristics of Communicated Objects

**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.

## 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;
}
```

## Aggregation (containment)

Example: a LinkedList 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 LinkedList 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.

```
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 . . .
};
```

```
// . . . 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

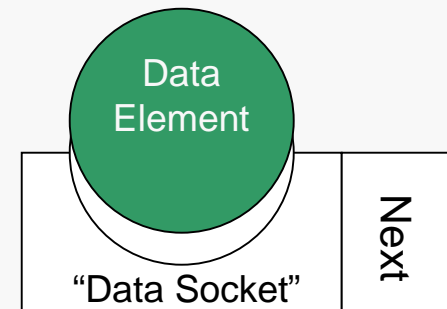
void Clear();                             // clear list to be empty, size zero
void Reverse();                           // reverse order of list elements

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

The use of composition not only promotes the reuse of existing implementations, but also provides for more flexible implementations and improved encapsulation:

Here we have a design for a list node object that:

- separates the structural components (list pointers) from the data values
- allows the list node to store ANY type of data element



```
class LinkNode {
private:
    ItemType Data;           // data "capsule"
    LinkNode* Next;         // pointer to next node

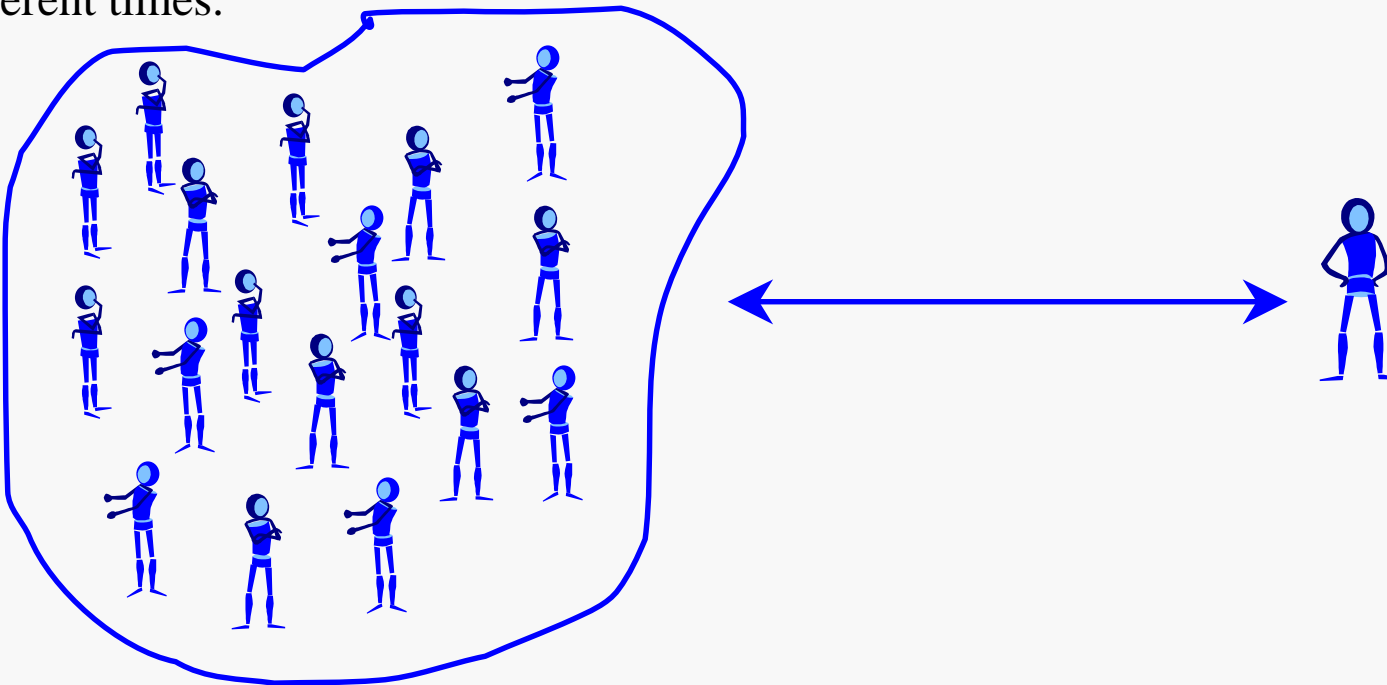
public:
    LinkNode();
    LinkNode(ItemType newData);
    void setData(ItemType newData);
    void setNext(LinkNode* newNext);
    ItemType getData() const;
    LinkNode* getNext() const;
};
```

Think of the “Borg” on Star Trek.

Borg crew member = 1 object

Borg Collective = composition of objects

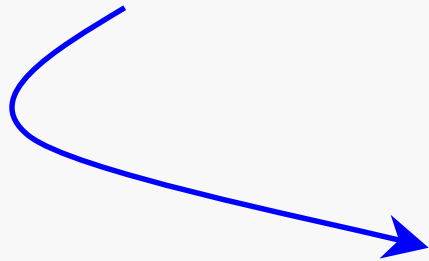
To achieve the purpose of the Collective, each Borg must continually communicate with other Borg. A Borg can be a “sender” or a “receiver”, and may play both roles at different times.



Similarly, objects can be senders or receivers.

By name: sender “knows the name” of the receiver and uses the name to access the public interface of the receiver.

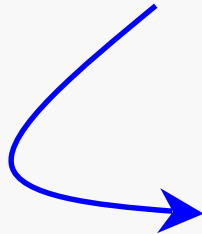
```
DisplayableNumber D(42, cout);  
  
D.Show(); // D accesses cout by its pointer member
```



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

An object may be passed as a function parameter:

```
DisplayableNumber D(42, cout);  
ofstream oFile("output.text");  
  
D.ShowIn(oFile); // D receives oFile as a parameter
```

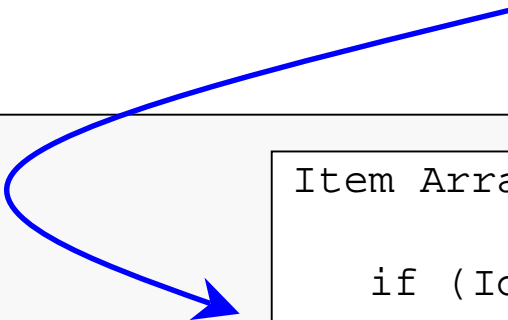


```
void DisplayableNumber::ShowIn(ostream& setOut) {  
    Out = &setOut; // store address of setOut  
}
```

As is always the case in C++, by default an object parameter is passed by value to the called function.

An object may be the return value from a function:

```
typedef DisplayableNumber Item;    // define an alias
const int Digits = 10;
Array LCD(Digits, Item(0, cout));  // array of DNs
. . .
DisplayableNumber Digit4 = LCD.Retrieve(4); // shallow copy
. . .
```



```
Item ArrayClass::Retrieve(Item Idx) const {
    if (Idx >= Usage)
        return -1;
    else
        return List[Idx];
}
```

Using an object as the return value provides a mechanism for encapsulating a body of related heterogeneous data (as does using a struct).

```
#include <iostream>
using namespace std;
#include "DisplayableNumber.h"
const int Digits = 5;

void main() {
    DisplayableNumber* LCD = new DisplayableNumber[Digits];

    for (int Idx = 0; Idx < Digits; Idx++) {
        LCD[Idx].Show();
    }
    delete [] LCD;
}
```

```
Constructing: DisplayableNumber
Constructing: DisplayableNumber
Constructing: DisplayableNumber
Constructing: DisplayableNumber
Constructing: DisplayableNumber
0
0
0
0
0
Destructing: DisplayableNumber
Destructing: DisplayableNumber
Destructing: DisplayableNumber
Destructing: DisplayableNumber
Destructing: DisplayableNumber
```

If the constructors and destructors are instrumented, this program produces the output:

# Aggregation Example: Fancy Array of Objects

```
#include <iostream>
using namespace std;

#include "Array.h"
const int Digits = 5;

void main() {
    Array LCD(Digits, Item(0, cout));

    for (int Idx = 0; Idx < Digits; Idx++) {
        LCD.Retrieve(Idx).Show();
    }
}
```

Note use of a nameless, or “anonymous” object.

Note use of member function of the returned DisplayableNumber object.

Array.h #includes Item.h, which contains:

```
#include "DisplayableNumber.h"
typedef DisplayableNumber Item;
```

# Aggregation Example: Fancy Array of Objects

```
#include <iostream>
using namespace std;

#include "Array.h"
const int Digits = 5;

void main() {
    Array LCD(Digits, Item(0, cout));

    for (int Idx = 0; Idx < Digits; Idx++) {
        LCD.Retrieve(Idx).Show();
    }
}
```



If the constructors and destructors are instrumented, this program produces the output shown.

There are a few subtleties illustrated here...

```
Constructing: DisplayableNumber
Constructing: Array
Constructing: DisplayableNumber
Constructing: DisplayableNumber
Constructing: DisplayableNumber
Constructing: DisplayableNumber
Constructing: DisplayableNumber
Destructing: DisplayableNumber
Destructing: DisplayableNumber
0
Destructing: DisplayableNumber
0
Destructing: DisplayableNumber
0
Destructing: DisplayableNumber
0
Destructing: DisplayableNumber
0
Destructing: DisplayableNumber
0
Destructing: Array
Destructing: DisplayableNumber
Destructing: DisplayableNumber
Destructing: DisplayableNumber
Destructing: DisplayableNumber
Destructing: DisplayableNumber
```

```
const int Digits = 5;  
...  
Array LCD(Digits, Item(0, cout));
```



```
Constructing: DisplayableNumber  
Constructing: Array  
Constructing: DisplayableNumber  
Constructing: DisplayableNumber  
Constructing: DisplayableNumber  
Constructing: DisplayableNumber  
Constructing: DisplayableNumber  
Constructing: DisplayableNumber
```

Here, the sub-objects are constructed AFTER the Array object is constructed. The reason is fairly clear if the Array constructor is examined:

```
Array::Array(int initCapacity, Item Value) {  
  
    Capacity = initCapacity;  
    Usage    = Current = 0;  
    List     = new Item[initCapacity];  
  
    for (int Idx = 0; Idx < Capacity; Idx++)  
        List[Idx] = Value;  
  
    Usage = Capacity;  
}
```

Here, the array elements don't exist until the Array constructor creates the array dynamically.

```
#include <iostream>
using namespace std;

#include "Array.h"
const int Digits = 5;

void main() {
    Array LCD(Digits, Item(0, cout));
    . . .
}
```

```
. . .
Destructing: DisplayableNumber
Destructing: DisplayableNumber
. . .
Destructing: Array
Destructing: DisplayableNumber
Destructing: DisplayableNumber
Destructing: DisplayableNumber
Destructing: DisplayableNumber
Destructing: DisplayableNumber
```

The anonymous `DisplayableNumber` object constructed in the `Array` constructor call is destructed when the call completes. Also, since it is passed by value, a copy is made within the constructor, and the copy is also destructed when the call completes.

When `main()` terminates, the `Array` object is destructed. The `Array` destructor deletes its array, and that causes the destruction of each array element.


```
. . .  
for (int Idx = 0; Idx < Digits; Idx++) {  
    LCD.Retrieve(Idx).Show();  
}  
. . .
```

On each pass through the for loop, an anonymous object is created and then destructed (when it goes out of scope at the end of the loop).

But, why are no constructor calls shown?

```
Item Array::Retrieve(int Idx) const {  
  
    if (Idx >= Usage)  
        return Item();  
    else  
        return List[Idx];  
}
```

Because the return value is created by a copy constructor (not instrumented).



```
. . .  
0  
Destructing: DisplayableNumber  
0  
Destructing: DisplayableNumber  
0  
Destructing: DisplayableNumber  
0  
Destructing: DisplayableNumber  
0  
Destructing: DisplayableNumber  
. . .
```

# One Last Aggregation Puzzler

```
class Foo {  
private:  
    DisplayableNumber DN;  
public:  
    Foo();  
    Foo(DisplayableNumber DN);  
    ~Foo();  
};
```

Now, the `DisplayableNumber` is a data member; it is created when a `Foo` object is created.

(TRUE aggregation)

```
DisplayableNumber myDN(17, cout);  
  
Foo urFoo(myDN);
```

```
Foo::Foo() {  
    cout << "Constructing: Foo" << endl;  
    DN = DisplayableNumber(0, cout);  
}  
  
Foo::Foo(DisplayableNumber DN) {  
    cout << "Constructing: Foo" << endl;  
    this->DN = DN;  
}  
  
Foo::~~Foo() {  
    cout << "Destructing: Foo" << endl;  
}
```

```
Constructing: DisplayableNumber  
. . .  
Constructing: DisplayableNumber  
Constructing: Foo  
Destructing: DisplayableNumber  
. . .  
Destructing: Foo  
Destructing: DisplayableNumber  
Destructing: DisplayableNumber
```

In a typical aggregation, where the sub-objects are data members (not allocated dynamically), the following rules hold for constructor and destructor sequencing:

**Construction:** the default constructor is invoked for each sub-object, then the constructor for the containing object is invoked.

So, aggregates are constructed from the inside-out.

**Destruction:** the destructor is invoked for the containing object first, and then the destructor for each sub-object is invoked.

So, aggregates are destructed from the outside-in.

It's also possible to invoke a non-default constructor for a sub-object. . .

Constructors for sub-objects may be explicitly invoked BEFORE the body of the containing object's constructor:



```
Foo::Foo() : DN(0, cout) {  
    cout << "Constructing: Foo" << endl;  
}
```

Here, a `DisplayableNumber` constructor is invoked and passed two parameters (which could have been parameters to the `Foo` constructor if that had been appropriate).

Is object communicated by:

- copying
- reference
- pointer

Can the receiver modify the object?

If the receiver does modify the object, does the sender see the changes?

What language syntax is used in receiver to access (. or ->)?

<u>Technique</u>	<u>Copied</u>	<u>Changeable</u>	<u>Visible</u>	<u>C++ Syntax</u>
by copy	yes	yes	no	.
by reference	no	yes	yes	.
by pointer	no	yes	yes	->
by const reference	no	no	no	.
by const pointer	no	no	no	->

## By Copy:

- ✓ Sender is “isolated” from changes by receiver
- ✗ No good if sender/receiver want to share object
- ✗ Bad if object is large (why?)

## By Identity (pointer or reference):

- ✗ No isolation
- ✓ Permits sharing of objects
- ✓ Improves memory cost for large objects