Communication Among Objects

• For a function to call member function of an object, the function must have access to that object
• To be accessible, object may be
  – Parameter to function
  – Part of class
  – Pointed to by field of class

Composition

• Aggregation - object is composed of sub-objects
• Association - one object uses services of another

Parameters versus Composition

• Parameters
  – Allow member function to use other object’s services
  – Short-term relationship
• Composition
  – Allow object (all member functions) to use other object’s services
  – Longer-term relationship

Example: Library System

• Reconsider library system discussed earlier
• Specifically two classes:
  - Catalog - holds information about books in library
  - Circulation - holds information about books that are circulating (due dates, etc.)
• And, one function Circulation::checkout
Checkout with Parameters

- Circulation::checkout needs to be able to access the catalog for the library
- Can do this by giving the Catalog object as a parameter
  void checkout(const Catalog&, const Book&, const Patron&);
- Must pass in catalog object each time
- Could pass in different catalogs each time

Checkout with Parameters

void Circulation::checkout(const Catalog& cat,
const Book& bk, const Patron& user) {
if (cat.bookExists(bk))
add circulation record to list
}

Checkout with Association

- Provide access to Catalog by association
- Circulation has (static) association to Catalog object
- Checkout function uses association pointer to access Catalog object
  void checkout(const Book&, const Patron&);
- Relationship lasts longer than one function call
Checkout with Association

```cpp
class Circulation {
public:
    // other constructors not shown
    Circulation(Catalog* cp) : catptr(cp) {} 
    // other functions not shown
    void checkout(const Book&, const Patron&); 
    private:
    Catalog* catptr; // associated catalog object 
    // data for Circulation database 
};
```

Checkout with Association

```cpp
void Circulation::checkout(const Book& bk, const Patron& user) { 
    if (catptr->bookExists(bk)) 
        add circulation record to list 
}
```

Parameters vs. Composition

- Both allow the functions of an object to “use” another object
- Use parameters if object only used for immediate computation
- Use composition if relationship will last beyond one computation
- Ex. Library system - use composition for circulation - catalog relationship, because lasts for life of objects

Aggregation vs. Association

- Aggregation - one object made of others, the sub-objects are part of the whole
- Association - one object uses another, but the two objects exist independently, and can be accessed independently

Checkout with Aggregation

```cpp
class Circulation {
public:
    // default and copy constructors not shown
    Circulation(const Catalog&); // encapsulate copy
    // other functions not shown
    bool search(const Book&); // is book checked out?
    bool searchCatalog(const Book&); // is book in catalog?
    void checkout(const Book&, const Patron&); 
    private:
    Catalog cat; // aggregated Catalog 
    // data for Circulation database 
};
```
Checkout with Aggregation

```cpp
void Circulation::
checkout(const Book& bk, const Patron& user) {
    if (cat.bookExists(bk))
        add circulation record to list
}
```

Aggregation vs. Association

- Two concepts completely different
- Use aggregation to define or construct objects from others
- Use association when one class needs to use other class, but objects need to be independent

Types of Aggregation

- Assembly-parts - whole is made of discrete parts
- Material-object - whole is "mixture of" materials
- Portion-object - whole is made of of portions
- Place-area - place occurs in area
- Collection-members
- Container-content
- Membership-partnership

Implementing Aggregation

- Static aggregation - fixed number of objects
- Dynamic aggregation - changing number
- In both cases, if dynamically allocate subobjects, must define constructors, assignment and destructor
- Copying must be deep

Implementing Association

- Static association - use constructor only
- Dynamic association - use member function
- Must use pointer to associated object

Implementing Associations

- **One-way**
  - ![Diagram](image1)
- **Two-way**
  - ![Diagram](image2)