CS2704

Topic: Aggregation
Outline

• Aggregation & Advantages
• Aside: Encapsulation
• Static Aggregation Examples
• Aside: Equality
• Dynamic Aggregation Examples
Aggregation

- Construction of an object from others
- Object may have sub-objects contained within it
Advantages

• Simple – deal with the containing object rather than all of them
• Safe – sub-objects encapsulated
• Specialized interface – general objects used together with interface specific to problem
• Structure indicates designers intention
• Can substitute implementations
Types of Aggregation

• Static – number of sub-objects cannot vary
  – A Rectangle has a Point and Dimension
  – A Customer has a Name and an Address

• Dynamic – number of sub-objects may vary
  – A catalog has many catalog items
  – A Host list has changing entries
Aggregation in Diagram

Static

Dynamic

4

*
Aside: Encapsulation

• Encapsulation – preventing access to internal data of object

• Always want to consider how much user of class needs to know about its internals
  – Less is better for design flexibility
  – More can be better for efficiency
  – Our focus always on “good” design
Ex. Static Aggregation (1)

Counter for bus passengers by payment method

Counter

void increment()

int getCount()

Passenger Counter

void incUniv()
void incMonthly()
void incCash()

3
Counter Class

class Counter {
        public:
        Counter() : cnt(0) {}  
        Counter(const Counter& c) : cnt(c.cnt) {}
        Counter(int c) : cnt(c) {}
        void increment() const { cnt++; }
        int getCount() const { return cnt; }

        private:
        int cnt;
};
Passenger Counter Class

class PassengerCounter {
public:
    PassengerCounter();
    PassengerCounter(const PassengerCounter&);
    void incUniv();
    void incMonthly();
    void incCash();
    int getUnivCount() const;
    int getMonthlyCount() const;
    int getCashCount() const;
private:
    Counter univ, monthly, cash;
};
PassengerCounter Constructors

PassengerCounter::PassengerCounter() :
    univ(), monthly(), cash() {}

PassengerCounter::
    PassengerCounter(const PassengerCounter& p) :
    univ(p.univ), monthly(p.monthly),
    cash(p.cash) {}
PassengerCounter Mutators

    void PassengerCounter::incUniv() {
        univ.increment();
    }
    void PassengerCounter::incMonthly() {
        monthly.increment();
    }
    void PassengerCounter::incCash() {
        cash.increment();
    }
PassengerCounter Accessors

```cpp
int PassengerCounter::getUnivCount() const {
    return univ.getCount();
}

int PassengerCounter::getMonthlyCount() const {
    return monthly.getCount();
}

int PassengerCounter::getCashCount() const {
    return cash.getCount();
}
```
Aside: Equality

• When are two Counters equal?
  – When are same object?
  – When have same value?

• Which is more appropriate depends on class

• Provide equality predicate or operator for class
Equality: Same Object

Check addresses of objects

```cpp
bool Counter::equalTo(const Counter& c) {
    return (this == &c);
}
```

Silly unless have pointers to Counter objects
Equality: Same Value

bool Counter::equalTo(const Counter& c) {
    return (cnt == c.cnt);
}

bool Counter::operator==(const Counter& c) {
    return (cnt == c.cnt);
}
Ex. Static Aggregation (2)

Diagram:

- Rectangle
  - Dimension
    - int width, height;
- Point
  - int x, y;
Point Class

class Point {
public:
    Point() : x(0), y(0) {}  
    Point(const Point& p) : x(p.x), y(p.y) {}  
    Point(int x_, int y_) : x(x_), y(y_) {}  
    int getX() const { return x; }  
    int getY() const { return y; }  
    Point translate(int x_delta, int y_delta) {
        return Point(x+x_delta, y+y_delta);  
    }
private:
    int x, y;
};
Dimension Class

class Dimension {
    public:
        Dimension() : width(0), height(0) {}  
        Dimension(const Dimension& d) :
            width(d.width), height(d.height) {} 
        Dimension(int w, int h) : width(w), height(h) {} 
        int getWidth() const { return width; } 
        int getHeight() const { return height; } 
        // possibly some other methods
    private:
        int width, height;
};
Rectangle Class

class Rectangle {

public:
    Rectangle (Point corner, Dimension shape);
    void MoveUp (int deltaY);
    void MoveDown (int deltaY);
    void MoveLeft (int deltaX);
    void MoveRight (int deltaX);
    void Draw (Canvas& canvas);
    void Clear (Canvas& canvas);
    ~Rectangle ();

private:
    Point upperLeft;
    Dimension area;

};
Rectangle Implementation (1)

Rectangle::Rectangle(Point corner, Dimension shape) :
    upperLeft(corner), area(shape) { }

void Rectangle::MoveUp(int deltaY) {
    upperLeft = upperLeft.translate(0, -deltaY);
}

// ... MoveDown, MoveLeft, MoveRight similar to MoveUp
Rectangle Implementation (2)

```cpp
void Rectangle::Draw(Canvas& canvas) {
    Point up_rgt = upperLeft.translate(area.getWidth(),0);
    Point lw_rgt = upperRight.translate(0,area.getHeight());
    Point lw_lft = upperLeft.translate(0,area.getHeight());
    canvas.DrawLine(upperLeft, up_rgt);
    canvas.DrawLine(up_rgt, lw_rgt);
    canvas.DrawLine(lw_rgt, lw_lft);
    canvas.DrawLine(lw_lft, upperLeft);
}
```
Rectangle Implementation (3)

```cpp
void Rectangle::Clear(Canvas& canvas) {
    canvas.Clear(upperLeft, area)
}

Rectangle::~Rectangle() {}
```
Implementing Static Aggregation

• Can manage sub-objects as either automatic or dynamically allocated objects

• Dynamically allocated data must be managed by
  – default constructor – create default sub-object
  – copy constructor, assignment – deep copy
  – destructor – delete sub-object

• Makes no sense for pointer to be null
Dynamic Aggregation

• Object with variable numbers of sub-objects
• Example: polygonal figure with arbitrary number of sides
PolyShape Aggregation

PolyShape → Point (*)
Detailed PolyShape Aggregation
PolyShape Structure

PolyShape

Object

tail

head

PointNode

next

Point

next

Point
PolyShape Class

class PolyShape {
public:
    PolyShape(int x, int y);
    void Add(const Point& newpoint);
    void Draw(Canvas& canvas);
    ~PolyShape();
private:
    PointNode *head;
    PointNode *tail;
    int length;
};
PolyShape Implementation

PolyShape::PolyShape (int x, int y) :
    head(0), tail(0), length(1)
{
    head=tail=new PointNode(Point(x,y));
}
Adding Point to PolyShape

//Add a point to polygon

void PolyShape::Add(const Point& newpoint) {
    PointNode *newNode =
        new PointNode(newpoint);
    tail->Next(newNode);
    tail = newNode;
    length = length + 1;
}
void PolyShape::Draw (Canvas& canvas) {
    if (length == 1) return;
    PointNode *node, *next;
    node = head;
    while (node != tail)
        next = node->Next();
        canvas.DrawLine(node->Contents(), next->Contents());
        node = next;
    }
    canvas.DrawLine(head->Contents(), tail->Contents());
}
PointNode Class

class PointNode {
    public:
        PointNode(const Point& loc);
        PointNode* Next();
        void Next(PointNode* nxt);
        Point& Contents();
        ~PointNode();
    private:
        PointNode *next;
        Point        *location;
};
PointNode Implementation

PointNode::PointNode(const Point& loc) {
    location = new Point(loc); //need location copy
    next = (PointNode*)0;
}

PointNode* PointNode::Next() { return next; }

void PointNode::Next(PointNode* nxt) { next = nxt; }

Point& PointNode::Contents() { return *location; }

PointNode::~PointNode() { delete location; }

PolyShape Destructor

PolyShape::~PolyShape() {
    PointNode *next = head;
    while (next) {
        PointNode *node = next->Next();
        delete next;
        next = node;
    }
}
Host Table Aggregation (2)
STL List class

- Standard Template Library (STL)
- Collection of template *container* classes
- Container class - contains other objects
- Type: `list<typename>`
- Functions to add objects to list, capability to iterate through list
Host Table Class

class HostTable {
public:
    HostTable() : entries() {} // build empty list
    HostTable(const HostTable& hl) : entries(hl.entries) {} // copy
    void insert(MachineInfo); // add entry
    MachineInfo find(const Name&) const; // find entry by name
    MachineInfo find(const IPAddress&) const; // find entry by number
    bool remove(const Name&); // remove entry
private:
    list<MachineInfo> entries;
};
Machine Information

class MachineInfo {
public:
    MachineInfo(): nme(), ipaddress() {} 
    MachineInfo(const MachineInfo& mi) : 
        nme(mi.nme), ipaddress(mi.ipaddress) {} 
    MachineInfo(const Name& n, const IPAddress& ip) : 
        nme(n), ipaddress(ip) {} 
    Name name() const { return nme; } 
    IPAddress address() const { return ipaddress; } 
private: 
    Name nme; 
    IPAddress ipaddress; 
};
class Name {
    public:
        Name();
        Name(const Name&);
        Name(const string& fst, const string& lst);
        void set(const string& fst, const list<string>& lst);
        string getHost() const { return host; }
        list<string> getDomain() const { return domain; }
        bool operator< (const Name&);
        bool operator== (const Name&);
    private:
        string host;
        list<string> domain;
};
Machine Address

//IP address is 4 numbers in range 0-255.
class IPAddress {
public:
    IPAddress() : address() {}
    IPAddress(const IPAddress& a) : address(a.address) {}
    IPAddress(const list<char>& a) : address(a) {}
    list<char> mask(const list<char>& a) {}
    list<char> mask(const list<char>& a);
    bool operator==(const IPAddress&);
    bool operator<(const IPAddress&);
private:
    list<char> address;
};
Inserting into Host Table

- Adds name to end of list

  ```cpp
  void HostTable::insert(const MachineInfo& machine) {
    entries.push_back(machine);
  }
  ```

- `push_back` is function of STL list class to add to end
- To add info in sorted position we need more STL
List Iterators

- Design concept: iterators
- Object that allows sequential access to data inside a container
- Kind of like a pointer
- Type can be iterator or const_iterator depending on whether container can be changed
Using List Iterators

• Iterator declaration:

  ```cpp
  list<MachineInfo>::iterator e_i;
  ```

• Set to front of list: `e_i = entries.begin();`

• Test if past last element: `e_i != entries.end()`

• Accessing object: `(*e_i).name()`

• Moving to next element: `e_i++`
Inserting Again

• Algorithm: look for location, then insert

    //inserting in sorted order
    void HostTable::insert(const MachineInfo& machine) {
        list<MachineInfo>::iterator e_i = entries.begin();
        while (e_i != entries.end() && machine.name() < (*e_i).name())
            e_i++;
        entries.insert(e_i, machine);
    }

• Requires that we overload Name::operator<
Aside: Comparison Operators

- Uses comparison operators from string and list classes

```cpp
bool Name::operator< (const Name& nme) {
    return (host < nme.host) && (domain < nme.domain);
}
```

- List comparison compares first entry, then second, ...
- Can also write as nonmember function
Find by Name

// similar to insert, but with const object.
// requires equality operator for Name class
MachineInfo HostTable::find(const Name& mname) const {
    list<MachineInfo>::const_iterator e_i = entries.begin();
    while (e_i != entries.end() && (*e_i).name() != mname)
        e_i ++;
    if (e_i != entries.end())
        return (*e_i);
    else
        return MachineInfo(); // can we handle this case better?
}
Remove by Name

```cpp
bool HostTable::remove(const Name& machine) {
    list<MachineInfo>::iterator e_i = entries.begin();
    while (e_i != entries.end() && (*e_i).name() != machine)
        e_i++;
    if (e_i != entries.end()) { //found equal
        entries.erase(e_i); //erase value at iterator
        return true;
    }
    else
        return false;
}
```
Implementing Dynamic Aggregation

• Sub-object must be stored in dynamic data structure

• Options:
  – Use existing container classes (e.g., from STL)
  – Create new data structure that must be managed
    • default constructor – create empty data structure
    • copy constructor, assignment – deep copy
    • destructor – must completely destroy data structure