CSCI 104

Rafael Ferreira da Silva

rafsilva@isi.edu

Slides adapted from: Mark Redekopp

STATIC MEMBERS

One For All

- As students are created we want them to have unique IDs
- How can we accomplish this?

```
class USCStudent {
 public:
   USCStudent(string n) : name(n)
   {  id = _________ ; // ????
   }

 private:
   string name;
   int id;
}

int main()
{
  // should each have unique IDs
  USCStudent s1("Tommy");
  USCStudent s2("Jill");




}
```

One For All

- Can we just make a counter data member of the USCStudent class?
- What's wrong with this?

```
class USCStudent {
 public:
   USCStudent(string n) : name(n)
   {  id = id_cntr++; }

 private:
   int id_cntr;
   string name;
   int id;
}


int main()
{
  USCStudent s1("Tommy"); // id = 1
  USCStudent s2("Jill");  // id = 2



}
```

One For All

- It's not something that we can do from w/in an instance
  - A student doesn't assign themselves an ID, they are told their ID
- Sometimes there are functions or data members that make sense to be part of a class but are shared amongst all instances
  - The variable or function doesn't depend on the instance of the object, but just the object in general
  - We can make these 'static' members which means one definition shared by all instances

```
class USCStudent {
 public:
   USCStudent(string n) : name(n)
   {  id = id_cntr++; }

 private:
   static int id_cntr;
   string name;
   int id;
}

// initialization of static member
int USCStudent::id_cntr = 1;

int main()
{
  USCStudent s1("Tommy"); // id = 1
  USCStudent s2("Jill");  // id = 2



}
```

Static Data Members

- A 'static' data member is a single variable that all instances of the class share
- Can think of it as belonging to the class and not each instance
- Declare with keyword 'static'
- Initialize outside the class in a .cpp (can't be in a header)
  - Precede name with className::

```
class USCStudent {
 public:
  static int id_cntr;
  USCStudent(string n) : name(n)
   {  id = id_cntr++; }

 private:
  string name;
  int id;
}

// initialization of static member
int USCStudent::id_cntr = 1;

int main()
{
  USCStudent s1("Tommy"); // id = 1
  USCStudent s2("Jill");  // id = 2



}
```

Another Example

- All US Citizens share the same president, though it changes over time
- Rather than wasting memory for each citizen to store a pointer to the president, we can make it static
- However, private static members can't be accessed from outside functions
- For this we can use a static member functions

```
class USCitizen{
 public:
   USCitizen();




 private:
   static President* pres;
   string name;
   int ssn;
}

int main()
{
  USCitizen c1;
  USCitizen c2;
  President* curr = new President;

  // won't compile..pres is private
  USCitizen::pres = curr;




}
```

Static Member Functions

- Static member functions do tasks at a class level and can't access data members (since they don't belong to an instance)
- Call them by preceding with 'className::'
- Use them to do common tasks for the class that don't require access to an instance's data members
  - Static functions could really just be globally scoped functions but if they are really serving a class' needs it makes sense to group them with the class

```
class USCitizen{
 public:
   USCitizen();
   static void setPresident(President* p)
    {  pres = p; }

 private:
   static President* pres;
   string name;
   int ssn;
}

int main()
{
  USCitizen c1;
  USCitizen c2;
  President* curr = new President;
  USCitizen::setPresident(curr);
  ...
  President* next = new President;
  USCitizen::setPresident(next);

}
```

It's an object, it's a function…it's both rolled into one!

DESIGN PATTERNS AND PRINCIPLES

Coupling

- Coupling refers to how much components depend on each other's implementation details (i.e. how much work it is to remove one component and drop in a new implementation of it)
  - Placing a new battery in your car vs. a new engine
  - Adding a USB device vs. a new processor to your laptop
- OO Design seeks to reduce coupling (i.e. **loose** coupling) as much as possible
  - If you need to know or depend on the specific implementation of another class to write your current code, you are **tightly** coupled…BAD!!!!
  - Code should be designed so modification of one component/class does not require modification and unit-testing of other components
    - Just unit-test the new code and test the overall system

Design Principles

- Let the design dictate the details as much as possible rather than the details dictate the design
  - Top-down design
  - A car designer shouldn't say, "It would be a lot easier to make anti-lock brakes if the driver would just pulse the brake pedal 30 times a second"
- Open-Close Principle
  - Classes should be **open** to extension but **closed** to modification (After initial design and testing that is)
    - To alter behavior and functionality, inheritance should be used
    - Base classes should be designed with that in mind (i.e. extensible)
  - Extend and change behavior by allocating different (derived) objects at creation and passing them in (via the abstract base class pointer) to an object
    - Did you use this idea during the semester?
  - The client has programmed to an interface and thus doesn't need to change (is decoupled)

Re-Factoring

- f(x) = axy + bxy + cy
  - How would you factor this?
  - f(x) = y*(x*(a+b)+c)
  - We pull or **lift** the common term out leaving just what is unique to each term
- During design implementation we often need to refactor our code which may include
  - Extracting a common sequence of code into a function
  - Extracting a base class when you see many classes with a common interface
  - Replacing if..else statements based on the "type" of thing with polymorphic classes
  - …and many more
  - http://sourcemaking.com/

How to design effective class hierarchies with low coupling

SPECIFIC DESIGN PATTERNS

Design Patterns

- Common software practices to create modular code
  - Often using inheritance and polymorphism
- Researches studied software development processes and actual code to see if there were common patterns that were often used
  - Most well-known study resulted in a book by four authors affectionately known as the "Gang of Four" (or GoF)
    - Design Patterns: Elements of Reusable Object-Oriented Software by Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides
- Creational Patterns
  - Singleton, Factory Method, Abstract Factory, Builder, Prototype
- Structural Patterns
  - Adapter, Façade, Decorator, Bridge, Composite, Flyweight, Proxy
- Behavioral Patterns
  - Iterator, Mediator, Chain of Responsibility, Command, State, Memento, Observer, Template Method, Strategy, Visitor, Interpreter

Understanding UML Relationships

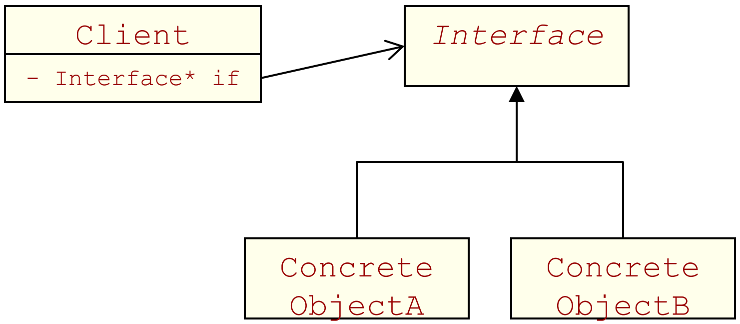
- UML Relationships
  - http://wiki.msvincognito.nl/Study/Bachelor/Year_2/Object_Oriented_Modelling/Summary/Object-Oriented_Design_Process
  - http://www.cs.sjsu.edu/~drobot/cs146/UMLDiagrams.htm
- Design Patterns
  - Strategy
  - Factory Method
  - Template Method
  - Observer

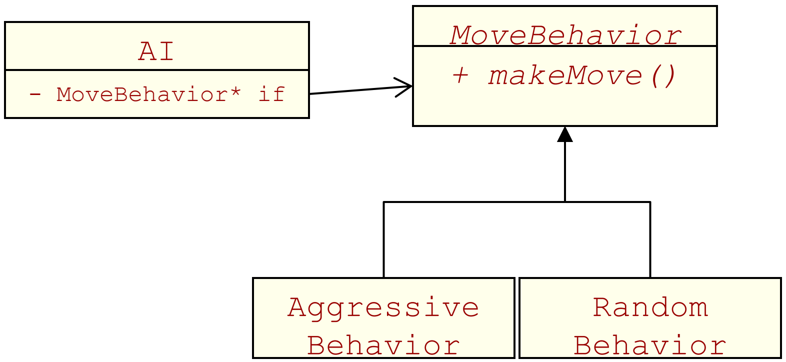
Iterator

- Decouples organization of data in a collection from the client who wants to iterate over the data
  - Data could be in a BST, linked list, or array
  - Client just needs to…
    - Allocate an iterator [it = collection.begin()]
    - Dereferences the iterator to access data [*it]
    - Increment/decrement the iterator [++it]

Strategy

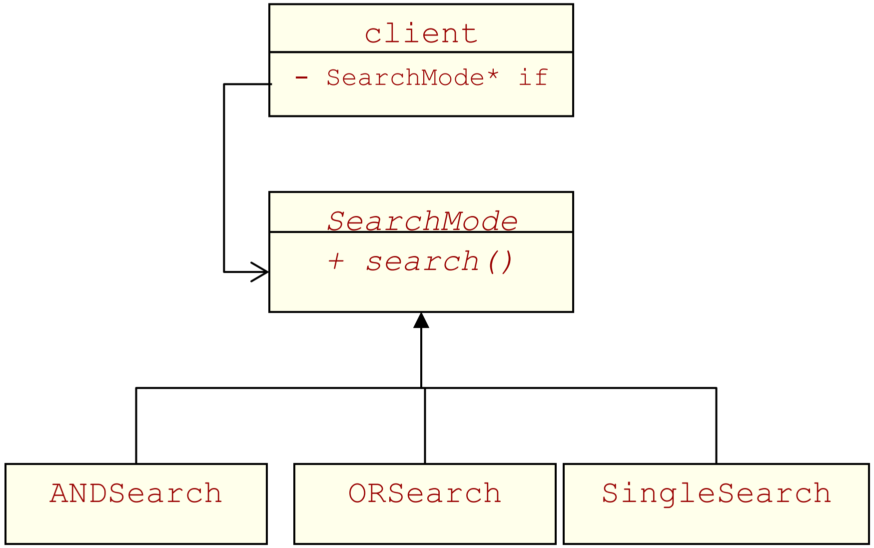
- Abstracting interface to allow alternative approaches
- Fairly classic polymorphism idea
- In a video game the AI may take different strategies
  - Decouples AI logic from how moves are chosen and provides for alternative approaches to determine what move to make
- Recall "Shapes" exercise in class
  - Program that dealt with abstract shape class rather than concrete rectangles, circles, etc.
  - The program could now deal with any new shape provided it fit the interface





Your Search Engine

- Think about your class project and where you might be able to use the strategy pattern
- AND, OR, Normal Search

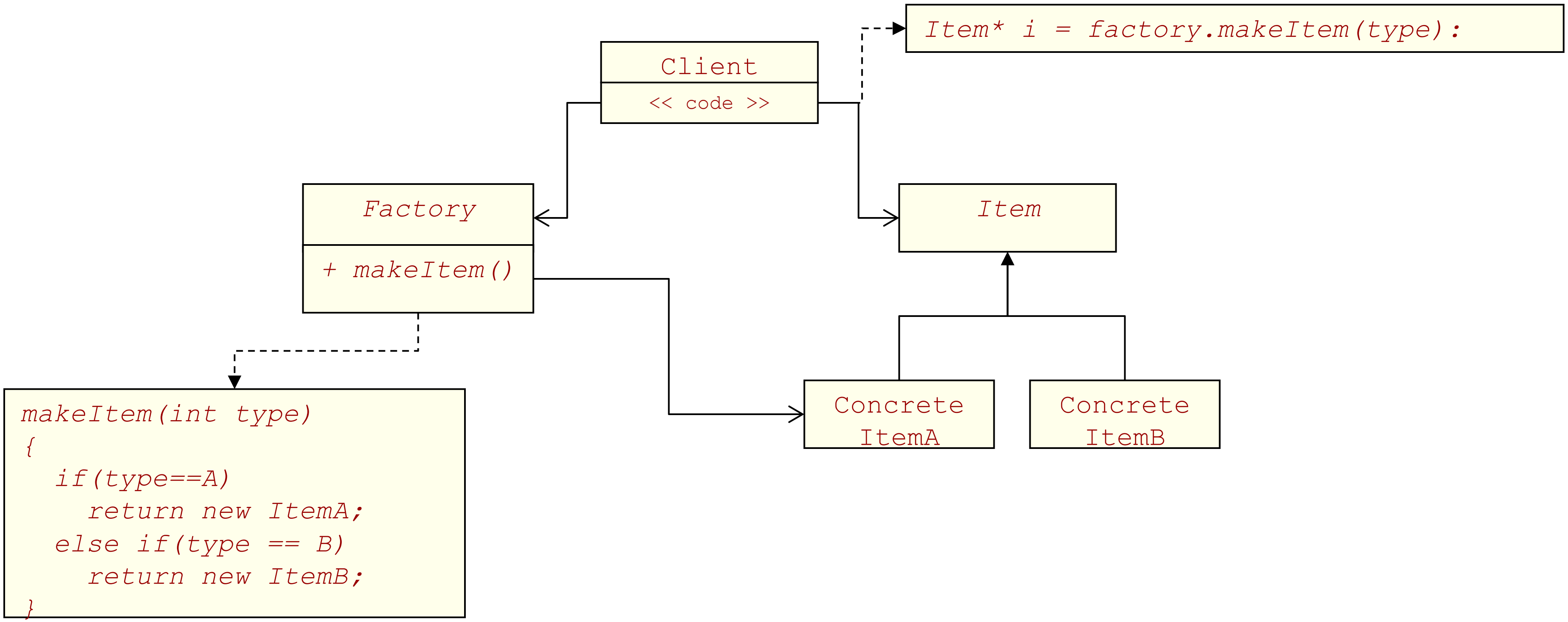



```
string searchType;
string searchWords;

cin >> sType;
SearchMode* s;
if(sType == "AND"){
  s = new ANDSearch;
}
else if(sType == "OR")
{
  s = new ORSearch;
}
else {
  s = new SingleSearch;
}

getline(cin, searchWords);
s->search(searchWords);
```

Client

Factory Pattern

- A function, class, or static function of a class used to abstract creation
- Rather than making your client construct objects (via 'new', etc.), abstract that functionality so that it can be easily extended without affecting the client



Factory Example

- We can pair up our search strategy objects with a factory to allow for easy creation of new approaches

Factory

```
class SearchFactory{
 public:
   static SearchMode* create(string type)
   {
     if(type == "AND")
       return new ANDSearch;
     else if(searchType == "OR")
       return new ORSearch;
     else
       return new SingleSearch;
   }
};
```

Client

```
string searchType;
string searchWords;

cin >> sType;
SearchMode* s = SearchFactory::create(sType);

getline(cin, searchWords);
s->search(searchWords);
```

Search Interface

```
class SearchMode {
  public:
   virtual search(set<string> searchWords) = 0;
...
};
```

Concrete Search

```
class AndSearchMode : public SearchMode
{
  public:
   search(set<string> searchWords){
     // perform AND search approach
   }
   ...
};
```

Factory Example

- The benefit is now I can add new search modes without the client changing or even recompiling

```
class SearchFactory{
 public:
   static SearchMode* create(string type)
   {
     if(type == "AND")
       return new ANDSearch;
     else if(searchType == "OR")
       return new ORSearch;
     else if(searchType == "XOR")
       return new XORSearch;
     else
       return new SingleSearch;
   }
};
```

```
string searchType;
string searchWords;

cin >> sType;
SearchMode* s = SearchFactory::create(sType);

getline(cin, searchWords);
s->search(searchWords);
```

```
class XORSearchMode : public SearchMode
{
  public:
   search(set<string> searchWords);
   ...
};
```

On Your Own

- Design Patterns
  - Observer
  - Proxy
  - Template Method
  - Adapter
- Questions to try to answer
  - How does it make the design more modular (loosely coupled)
  - When/why would you use the pattern
- Resources
  - http://sourcemaking.com/
  - http://www.vincehuston.org/dp/
  - http://www.oodesign.com/

Templates vs. Inheritance

- Inheritance and dynamic-binding provide run-time polymorphism
  - Example:
    - Strategy *s; …; s->search(words);
- C++ templates provide compile-time inheritance

```
class ANDSearch {
  public:
   set<WebPage*> search(vector<string>& words);
};
class ORSearch {
  ...
};

template <typename S>
set<WebPage*> doSearch(S* search_mode,
                       vector<string>& words)
{
  return search_mode->search(words);
}

...
ANDSearch mode;
Set<WebPage*> results = doSearch(mode, ...);
```

Templates vs. Inheritance

- Benefit of inheritance and dynamic-binding is its ability to store different-type but related objects in a single container
  - Example:
    - forEach shape s in Shapes { s->getArea(); }
  - Benefit: Different objects in one collection
- Benefit of templates is less run-time overhead (faster) due to compiler ability to optimize since it knows the specific type of object used