CS 246 - Object Oriented Programming **Fall 2016**

Lecture 8: November 3rd, 2016

Lecturer: Brad Lushman *Notes By: Harsh Mistry*

8.1 Destructor Revisted

Consider :

```
class x {
    int * x;
  public :
    X (int n): x{new int {n}} {}
    ~X() {delete x}
};

class y : public X {
    int *y;
  public :
    Y(int m, int n) : X{n}, y{new int [m];}
    ~Y() {delete [] y;}
};

x *p = new Y{10,20};
delete p;
```

What Happens when object is destructed

- Destructor runs
- Fields destructed
- Base class destructor runs
- Space deallocated.

Given the example above, variable x does not need to be deleted from the class y and cannot be deleted as its private. Since the base class destructor is run after fields are destructed.

In addition, when P is deleted, memory will be leaked, as only the x constructor runs. To fix this we can set the destructor to be `Virtual` . It is recommend that destructors are always `virtual` in classes that are supposed to have subclasses, even if the destructor does not do anything because it helps ensure that subclass destructor is called.

If a class is not meant to have any subclasses declare it `final`.

```
Class y final : public x { ... };
```

8.2 Pure Virtual Methods and Abstract Classes

Consider :

```
Class Student {
    ...
  public :
    virtual int fees();
};

Class Regular : public Student {
    ...
  public :
    int fees() override;
};

Class Coop : public Student {
    ...
  public :
    int fees() override;
};
```

In the example above, there is nothing to put in `student::fees` as a student is either regular or coop. So to avoid having to define it, we can explicitly give `student::fees` no implementation by setting the function to equal 0, which means the method has no implementation.

```
class Student {
    ...
  public :
    virtual int fees() = 0;
};
```

This is refereed to as a **pure virtual method**. When a class has a pure virtual method, it cannot be instantiated. Such as class is called an **abstract class**.

Subclasses of an abstract class are also abstract unless you fully implement all the pure virtual methods. Additionally, non abstract classes are refereed to as **concrete**.

In UML : Abstract class names are italicized and anything static is underlined

8.3 Templates

Consider :

```
class List{
    struct node;
    ...
};

Struct List::node {
    int data;
    Node *next;
};
```

The above example allows you to create a list of integers, but if you want a list of any other type, you must redefine the class. Fortunately, C++ has a feature called templates.

C++ templates are classes parameterized by a type.

```
template <typename T> class Stack{
    int size; cap;
    T *contents;
  public :
    void push(T x) { ... }
    T top() {...}
    void pop() {...};
};

//Client Code
Stack <int> l1;
Stacl <string> l2;
```

8.3.1 The Standard Template Library (STL) and Vectors

A large number of templates are included within this library. A example of a useful template is `Vector` which allows for dynamic length arrays.

```
#include <vector>

std::vector<int> v {4,5}; // Vector with 4,5 as {} is type std::initializer_list <int>
std::vector<int> q {4,5}; // Vector with 5,5,5,5

v.emplace_back(8) //Adds 8 to the back

//Looping through vectors
for (int i =0; i < v.size(); ++i){
  cout << v[i] << endl;
}

//Iterator
for (vector<int>::iterator it=v.begin; it!=v.end(); ++i{
  cout << *it << endl;
}

for (auto n : v){
  cout << n << endl;
}

//Reverse Iterator
for (vector<int>::reverse.iterator it = v.begin (); it != v.rend(); ++it) {
  cout << *it << endl;
}

//Removing an item
v.pop.back() - removes last element

//Use Iterator to remove items from inside a vector
auto it = v.erase(v.begin()); // Erases item 0
it = v.erase(v.begin()+3); //Erases item 3
it = v.erase(it);
it - v.erase(v.end() - 1); // Erase last item

//Getting values
int i = v[j]; // Unchecked values
int u = v.at(j); //Checked for references outside bounds
```

8.4 Exceptions

Problem: Vector can detect errors, but doesn't know what to do with it. Client knows what to do, but can't detect it.

C solution : Functions return a status code or set the global variable error. But this leads to awkward programming and discourages error checking.

C ++ solution : Use a try catch block `try{...} catch {...}`

```
#include <stdexcept>

try{
  cout << v.at(3) << endl; // out of range
}
catch (out_of_range r) {
  cerr << "Bad range" << r.what() << endl;
  throw; //Calls error out_of_range because of inheritance
}

void f() { throw out-of-range {"f"};} // f is the what value
```

If you want to catch any generic error you can use "..." in place of the exception in catch.

```
try {}
catch (...) {}
```

In addition, in c++ you can throw anything and the actual item thrown, does not need to be an object

```
void f() {throw 5)
try {}
catch (int m ) {cout << m << endl;}
```

If you want to define your own exceptions, you can create an object (struct/class) that represents the exception and then proceed to throw that upon an exceptional situation.

```
class BadInput {};

try {
  ...
}
catch (BadInput &b){
}
```

The class is caught by reference in the catch block, to avoid slicing in cases of inheritance. So, its good practice to always catch by reference if you aren't throwing integers.

Example 8.1 `std::bad.alloc` *is raised when* `new` *fails*

8.5 Design Patterns Continued

Guiding Principal : Program to the interface, not the implementation.

- Abstract base classes define the interface
  - work with pointers to the abstract class and their methods
  - Concrete subclasses can be swapped in and out.

8.5.1 Observer Pattern

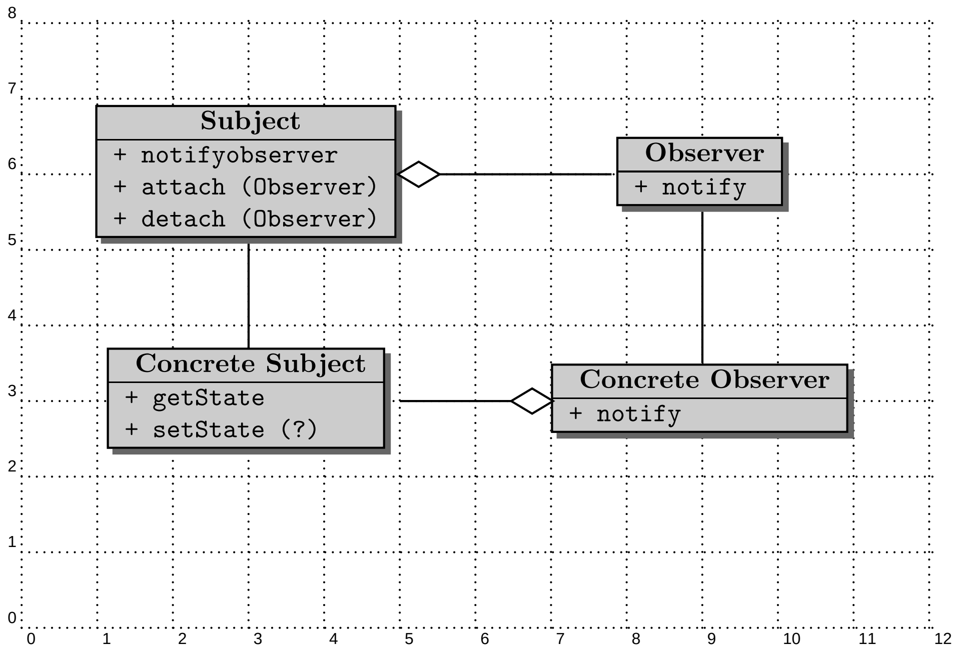
Publish-subscribe model

- One class:publisher/subject - generates data
- One or more subscribers/observers classes

Example 8.2 -

- *Publisher = Spreadsheet cells*
- *Observers = Graphs*
    - *When cells change, graphs update.*

The idea of the observer pattern is that that a subject should not need to know all the details.



Sequence of Method Calls

1. Subject state changes
2. Subject::notifyobservers() - calls each observer's notify
3. Each observer calls concrete ConcreteSubject::getState to query the state and react accordingly

Example 8.3 *Horse Race*

- *Subject : Publishes winners*
- *Observers : Bettors - declare victory if they win*

```
Class Subject{
    vector <observer *> observers;
  public :
    void attach (observer *o){ observer.emplace_back(0);}
    void detach (observer *o);
    void notify Observers (){
      for (auto &ob:observers) ob->notify();
    }
    virtual ~Subject() = 0;
};

Subject::~subject() {} // Must be defined, even though its pure virtual.
```