Execute-Around Pointer [edit]

Intent [edit]

Provide a smart pointer object that transparently executes actions before and after each function call on an object, given that the actions performed are the same for all functions.[1] This can be regarded as a special form of aspect oriented programming(AOP).

Also Known As [edit]

Double application of smart pointer.

Motivation [edit]

Often times it is necessary to execute a functionality before and after every member function call of a class. For example, in a multi-threaded application it is necessary to lock before modifying the data structure and unlock it afterwards. In a data structure visualization application might be interested in the size of the data structure after every insert/delete operation.

```
using namespace std;
class Visualizer {
    std::vector <int> & vect;
  public:
    Visualizer (vector<int> &v) : vect(v) {}
    void data_changed () {
       std::cout << "Now size is: " << vect.size();
    }
};
int main () // A data visualization application.
{
  std::vector <int> vector;
  Visualizer visu (vector);
  //...
  vector.push_back (10);
  visu.data_changed ();
  vector.push_back (20);
  visu.data_changed ();
  // Many more insert/remove calls here
  // and corresponding calls to visualizer.
}
```

Such a repetition of function calls is error-prone and tedious. It would be ideal if calls to visualizer could be automated. Visualizer could be used for std::list <int> as well. Such funcitonality which is not a part of single class but rather cross cuts multiple classes is commonly known as aspects. This particular idiom is useful for designing and implementing simple aspects.

Solution and Sample Code [edit]

```
class VisualizableVector {
  public:
    class proxy {
      public:
        proxy (vector<int> *v) : vect (v) {
          std::cout << "Before size is: " << vect->size ();
        }
        vector<int> * operator -> () {
          return vect;
        }
        ~proxy () {
          std::cout << "After size is: " << vect->size ();
        }
      private:
        vector <int> * vect;
    };
    VisualizableVector (vector<int> *v) : vect(v) {}
    proxy operator -> () {
       return proxy (vect);
    }
  private:
    vector <int> * vect;
};
int main()
{
  VisualizableVector vecc (new vector<int>);
  //...
  vecc->push_back (10); // Note use of -> operator instead of . operator
  vecc->push_back (20);
}
```

Overloaded -> operator of visualizableVector creates a temporary proxy object and it is returned. Constructor of proxy object logs size of the vector. The overloaded -> operator of proxy is then called and it simply forwards the call to the underlying vector object by returning a raw pointer to it. After the real call to the vector finishes, destructor of proxy logs the size again. Thus the logging for visualization is transparent and the main function becomes free from clutter. This idiom is a special case of Execute Around Proxy, which is more general and powerful.

The real power of the idiom can be derived if we combine it judiciously with templates and chain the overloaded -> operators.

```
template <class NextAspect, class Para>
class Aspect
{
  protected:
    Aspect (Para p): para_(p) {}
    Para  para_;
  public:
    NextAspect operator -> ()
    {
      return NextAspect (para_);
    }
};

template <class NextAspect, class Para>
struct Visualizing : Aspect <NextAspect, Para>
{
  public:
    Visualizing (Para p)
       : Aspect <NextAspect, Para> (p)
    {
    std::cout << "Before Visualization aspect" << std::endl;
    }
    ~Visualizing ()
    {
```

```cpp
    std::cout << "After Visualization aspect" << std::endl;
    }
};
template <class NextAspect, class Para>
struct Locking : Aspect <NextAspect, Para>
{
  public:
    Locking (Para p)
       : Aspect <NextAspect, Para> (p)
    {
        std::cout << "Before Lock aspect" << std::endl;
    }
    ~Locking ()
    {
    std::cout << "After Lock aspect" << std::endl;
    }
};
template <class NextAspect, class Para>
struct Logging : Aspect <NextAspect, Para>
{
  public:
    Logging (Para p)
        : Aspect <NextAspect, Para> (p)
    {
        std::cout << "Before Log aspect" << std::endl;
    }
    ~Logging ()
    {
    std::cout << "After Log aspect" << std::endl;
    }
};
template <class Aspect, class Para>
class AspectWeaver
{
public:
    AspectWeaver (Para p) : para_(p) {}
    Aspect operator -> ()
    {
       return Aspect (para_);
    }
private:
    Para para_;
};

#define AW1(T,U) AspectWeaver <T <U, U>, U >
#define AW2(T,U,V) AspectWeaver <T < U <V, V> , V>, V >
#define AW3(T,U,V,X) AspectWeaver <T < U <V <X, X>, X> , X>, X >

int main()
{
  AW3(Visualizing, Locking, Logging, vector <int> *)
      X (new vector<int>);
  //...
  X->push_back (10); // Note use of -> operator instead of . operator
  X->push_back (20);
  return 0;
}
```

Known Uses [edit]

Related Idioms [edit]

Smart Pointer

References [edit]

1. ↑ Execute Around Sequences - Kevlin Henney

Category: More C++ Idioms