Design Patterns in C++

Metaprogramming applied

Giuseppe Lipari
`http://retis.sssup.it/~lipari`

Scuola Superiore Sant'Anna – Pisa

April 13, 2011

Outline

How to generate a class with typelists

- Suppose you want to generate a class with *N* fields, of different types
- This can be done by using template template parameters, and the typelist facility

```
template<class TList, template <class> class Unit>
class GenScatterHierarchy;

// empty specialization for NullType (end of list)
template<template <class> class Unit>
class GenScatterHierarchy<NullType, Unit> {};

// if an atomic type (not a list), pass it to Unit
template <class AtomicType, template <class> class Unit>
class GenScatterHierarchy : public Unit<AtomicType>
{
    typedef typename Unit<AtomicType> LeftBase;
};
```

GenScatterHierarchy continued

```
// recursively apply Unit to the TList elements
template <class Head, class Tail, template <class> class Unit>
class GenScatterHierarchy<TypeList<Head, Tail>, Unit> :
    public GenScatterHierarchy<Head, Unit>,
    public GenScatterHierarchy<Tail, Unit>
{
    public:
        typedef typename TypeList<Head, Tail> TList;
        typedef typename GenScatterHierarchy<Head, Unit> LeftBase;
        typedef typename GenScatterHierarchy<Tail, Unit> RightBase;

};
```

- `GenScatterHierarchy` derives from two classes:
  - the left base applies an atomic type (`Head`) to `Unit`
  - the right base is a GenScatterHierarchy applied on the Tail

GenScatterHierarchy at work

- The best way to understand what is going on, is to do an example with two types, `int` and `string`
- first, lets define a template class that will work as an `Unit`

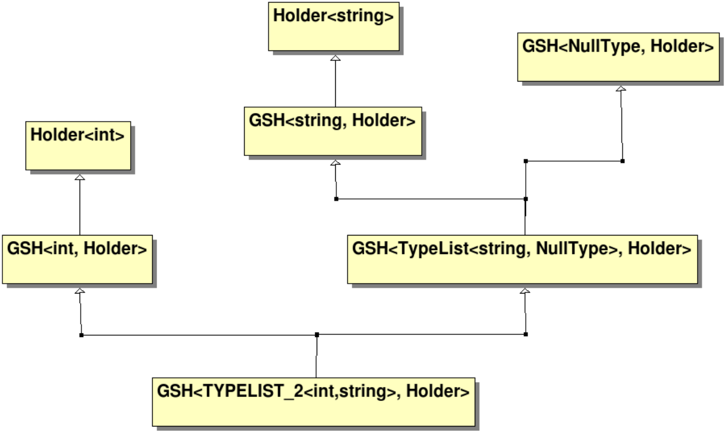
```
template <class T>
struct Holder {
    T value;
};
```

- And now let's apply it to GenScatterHierarchy:

```
typedef GenScatterHierarchy<TYPELIST_2(int, string), Holder> MyClass;
```

- That's it!
- What did we obtain?

The hierarchy



Outline

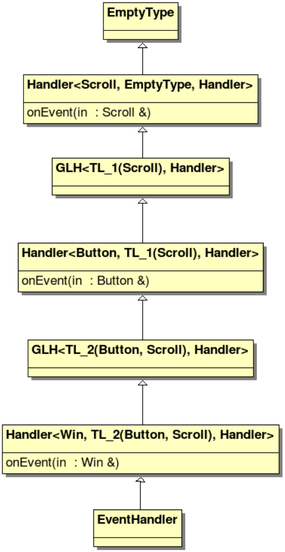
Linear

- Sometimes it is useful to have a linear inheritance hierarchy
- this can be done if we provide a Holder class with two template parameters

```
template<class TList,
   template<class Atomic, class Base> class Unit,
   class Root = EmptyType>
class GenLinearHierarchy;

// atomic type
template <class Atomic,
   template <class, class> class Unit,
   class Root>
class GenLinearHierarchy<TYPELIST_1(T), Unit, Root> :
   public Unit<T, Root> {};

// recursion
template <class Head, class Tail,
   template <class, class> class Unit,
   class Root>
class GenLinearHierarchy<TypeList<Head, Tail>, Unit, Root> :
   public Unit<Head, GenLinearHierarchy<Tail, Unit, Root> > {};
```

A linear hierarchy

```
template <class T, class Base>
class Handler : public Base {
public:
   virtual void onEvent(T& obj);
};

typedef GenLinearHierarchy
<
   TYPELIST_3(Win, Button, Scroll),
   Handler
>
EventHandler;
```



Outline

Functors

- Functors in C++ are simple classes that implement the Command pattern
  - they encapsulate a function call, with its state
  - they allow to defer call to a later times
  - they allow “call backs”, very useful for building libraries and frameworks
- a functor in C++

```
class Functor {
    ...
public:
    ReturnType operator()(ParameterType p);
};
...
Functor f;
...
f(p1); // call to operator()
```

Generalized Functors

- we want to design a *generalized* functor that forwards the call to another function
  - It can forward to a simple C-like function
  - it can forward to a member functions of some class
  - it can forward to another functor (to allow chaining)
- of course, we will use templates a lot

Functor basics

- Let's start easy (no parameters)

```
template<class ReturnType>
class Functor {
public:
    ResultType operator()();
private:
    // implementation
};
```

- Now, we want to add parameters to the function call, to allow calling any function, with any number of parameters
- of course, parameters can be of different types
- we need typelists here

Implementation

- Unfortunately, typelists cannot do the magic by themselves, we need to make some repetition.
- to simplify things, we are going to build an implementation class `FunctorImpl`, using the pimpl idiom

```
template <typename R, class TList>
class FunctorImpl;

template <typename R>
class FunctorImpl<R, NullType> {
public:
    virtual R operator()() = 0;
    virtual FunctorImpl *clone() const = 0;
    virtual ~FunctorImpl() {}
};

template <typename R, typename P1>
class FunctorImpl<R, TYPELIST_1(P1)> {
public:
    virtual R operator()(P1 p1) = 0;
    virtual FunctorImpl *clone() const = 0;
    virtual ~FunctorImpl() {}
};
...
```

Functor

```
template<typename R, class TList>
class Functor {
    typedef TList ParmList;
    typedef typename TypeAtNonStrict<TList, 0, EmptyType>::Result Parm1;
    typedef typename TypeAtNonStrict<TList, 1, EmptyType>::Result Parm2;
    ...

    R operator()() { return (*spImpl)(); }
    R operator()(Parm1 p1) { return (*spImpl)(p1); }
    R operator()(Parm1 p1, Parm2 p2) { return (*spImpl)(p1, p2); }
    ...
};
```

- Keep in mind that only the correct operator() will be compiled, the other ones will be ignored by the compiler
- also, if you try to invoke the wrong operator, you get an error (see code)

Construction

- How to construct functors

```
template <typename R, class TList>
class Functor {
    ...
public:
    template <typename Fun>
    Functor(const Fun& f);
};
```

- Where is the implementation?
  - We can define FunctorHandler to derive from FunctorImpl, implementing a simple forwarding to f

Handlers

```
template <class ParentFunctor, typename Fun>
class FunctorHandler :
    public FunctorImpl<
        typename ParentFunctor::ResultType,
        typename ParentFunctor::ParmList> {
public:
    typedef typename ParentFunctor::ResultType ResultType;

    FunctorHandler(const Fun& fun) : fun_(fun) {}
    FunctorHandler * clone() const {
        return new FunctorHandler(*this);
    }

    ResultType operator()() { return fun_(); }
    ResultType operator()(typename ParentFunctor::Parm1 p1) {
        return fun_(p1);
    }
    ...
private :
    Fun fun_;
};
```