Boost.Functional/Factory 1.0

Tobias Schwinger

Table of Contents

Brief Description

The template `boost::factory` lets you encapsulate a `new` expression as a function object, `boost::value_factory` encapsulates a constructor invocation without `new`.

```
boost::factory<T*>()(arg1,arg2,arg3)
// same as new T(arg1,arg2,arg3)

boost::value_factory<T>()(arg1,arg2,arg3)
// same as T(arg1,arg2,arg3)
```

For technical reasons the arguments to the function objects have to be LValues. A factory that also accepts RValues can be composed using the `boost::forward_adapter` or `boost::bind`.

Background

In traditional Object Oriented Programming a Factory is an object implementing an interface of one or more methods that construct objects conforming to known interfaces.

```
// assuming a_concrete_class and another_concrete_class are derived
// from an_abstract_class

class a_factory
{
  public:
    virtual an_abstract_class* create() const = 0;
    virtual ~a_factory() { }
};

class a_concrete_factory : public a_factory
{
  public:
    virtual an_abstract_class* create() const
    {
        return new a_concrete_class();
    }
};

class another_concrete_factory : public a_factory
{
  public:
    virtual an_abstract_class* create() const
    {
        return new another_concrete_class();
    }
};

// [...]

int main()
{
    __boost__ptr_map__<std::string,a_factory> factories;

    // [...]

    factories.insert("a_name",std::auto_ptr<a_factory>(
        new a_concrete_factory));
    factories.insert("another_name",std::auto_ptr<a_factory>(
        new another_concrete_factory));

    // [...]

    std::auto_ptr<an_abstract_factory> x = factories[some_name]->create();

    // [...]
}
```

This approach has several drawbacks. The most obvious one is that there is lots of boilerplate code. In other words there is too much code to express a rather simple intention. We could use templates to get rid of some of it but the approach remains inflexible:

```
o We may want a factory that takes some arguments that are forwarded to
  the constructor,
o we will probably want to use smart pointers,
o we may want several member functions to create different kinds of
  objects,
o we might not necessarily need a polymorphic base class for the objects,
o as we will see, we do not need a factory base class at all,
o we might want to just call the constructor - without `new` to create
  an object on the stack, and
o finally we might want to use customized memory management.
```

Experience has shown that using function objects and generic Boost components for their composition, Design Patterns that describe callback mechasisms (typically requiring a high percentage of boilerplate code with pure Object Oriented methodology) become implementable with just few code lines and without extra classes.

Factories are callback mechanisms for constructors, so we provide two class templates, `boost::value_factory` and `boost::factory`, that encasulate object construction via direct application of the constructor and the `new` operator, respectively.

We let the function objects forward their arguments to the construction expressions they encapsulate. Overthis `boost::factory` optionally allows the use of smart pointers and Allocators.

Compile-time polymorphism can be used where appropriate,

```
template< class T >
void do_something()
{
    // [...]
    T x = T(a,b);

    // for conceptually similar objects x we neither need virtual
    // functions nor a common base class in this context.
    // [...]
}
```

Now, to allow inhomogenous signaturs for the constructors of the types passed in for `T` we can use `value_factory` and `boost::bind` to normalize between them.

```
template< class ValueFactory >
void do_something(ValueFactory make_obj = ValueFactory())
{
    // [...]
    typename ValueFactory::result_type x = make_obj(a,b);

    // for conceptually similar objects x we neither need virtual
    // functions nor a common base class in this context.
    // [...]
}

int main()
{
    // [...]

    do_something(boost::value_factory<X>());
    do_something(boost::bind(boost::value_factory<Y>(),_1,5,_2));
    // construct X(a,b) and Y(a,5,b), respectively.

    // [...]
}
```

Maybe we want our objects to outlive the function's scope, in this case we have to use dynamic allocation;

```
template< class Factory >
whatever do_something(Factory new_obj = Factory())
{
    typename Factory::result_type ptr = new_obj(a,b);

    // again, no common base class or virtual functions needed,
    // we could enforce a polymorphic base by writing e.g.
    //    boost::shared_ptr<base>
    // instead of
    //    typename Factory::result_type
    // above.
    // Note that we are also free to have the type erasure happen
    // somewhere else (e.g. in the constructor of this function's
    // result type).

    // [...]
}

// [... call do_something like above but with __factory__ instead
// of __value_factory__]
```

Although we might have created polymorphic objects in the previous example, we have used compile time polymorphism for the factory. If we want to erase the type of the factory and thus allow polymorphism at run time, we can use Boost.Function to do so. The first example can be rewritten as follows.

```
typedef boost::function< an_abstract_class*() > a_factory;

// [...]

int main()
{
    std::map<std::string,a_factory> factories;

    // [...]

    factories["a_name"] = boost::factory<a_concrete_class*>();
    factories["another_name"] =
        boost::factory<another_concrete_class*>();

    // [...]
}
```

Of course we can just as easy create factories that take arguments and/or return Smart Pointers.

Reference

value_factory

Description

Function object template that invokes the constructor of the type `T`.

Header

```
#include <boost/functional/value_factory.hpp>
```

Synopsis

```
namespace boost
{
    template< typename T >
    class value_factory;
}
```

Notation

`T` an arbitrary type with at least one public constructor

`a0...aN` argument LValues to a constructor of `T`

`F` the type `value_factory<F>`

`f` an instance object of `F`

Expression Semantics

Expression	Semantics
`F()`	creates an object of type `F`.
`F(f)`	creates an object of type `F`.
`f(a0...aN)`	returns `T(a0...aN)`.
`F::result_type`	is the type `T`.

Limits

The macro BOOST_FUNCTIONAL_VALUE_FACTORY_MAX_ARITY can be defined to set the maximum arity. It defaults to 10.

factory

Description

Function object template that dynamically constructs a pointee object for the type of pointer given as template argument. Smart pointers may be used for the template argument, given that `boost::pointee<Pointer>::type` yields the pointee type.

If an __allocator__ is given, it is used for memory allocation and the placement form of the `new` operator is used to construct the object. A function object that calls the destructor and deallocates the memory with a copy of the Allocator is used for the second constructor argument of `Pointer` (thus it must be a __smart_pointer__ that provides a suitable constructor, such as `boost::shared_ptr`).

If a third template argument is `factory_passes_alloc_to_smart_pointer`, the allocator itself is used for the third constructor argument of `Pointer` (`boost::shared_ptr` then uses the allocator to manage the memory of its seperately allocated reference counter).

Header

```
#include <boost/functional/factory.hpp>
```

Synopsis

```
namespace boost
{
    enum factory_alloc_propagation
    {
        factory_alloc_for_pointee_and_deleter,
        factory_passes_alloc_to_smart_pointer
    };

    template< typename Pointer,
        class Allocator = boost::none_t,
        factory_alloc_propagation AllocProp =
            factory_alloc_for_pointee_and_deleter >
    class factory;
}
```

Notation

`T`	an arbitrary type with at least one public constructor
`P`	pointer or smart pointer to `T`
`a0...aN`	argument LValues to a constructor of `T`
`F`	the type `factory<P>`
`f`	an instance object of `F`

Expression Semantics

Expression	Semantics
`F()`	creates an object of type `F`.
`F(f)`	creates an object of type `F`.
`f(a0...aN)`	dynamically creates an object of type `T` using `a0...aN` as arguments for the constructor invocation.
`F::result_type`	is the type `P` with top-level cv-qualifiers removed.

Limits

The macro BOOST_FUNCTIONAL_FACTORY_MAX_ARITY can be defined to set the maximum arity. It defaults to 10.

Acknowledgements

Eric Niebler requested a function to invoke a type's constructor (with the arguments supplied as a Tuple) as a Fusion feature. These Factory utilities are a factored-out generalization of this idea.

Dave Abrahams suggested Smart Pointer support for exception safety, providing useful hints for the implementation.

Joel de Guzman's documentation style was copied from Fusion.

Further, I want to thank Peter Dimov for sharing his insights on language details and their evolution.

References

1. Design Patterns, Gamma et al. - Addison Wesley Publishing, 1995
2. Standard Template Library Programmer's Guide, Hewlett-Packard Company, 1994
3. Boost.Bind, Peter Dimov, 2001-2005
4. Boost.Function, Douglas Gregor, 2001-2004