# Policy-Based Class Design

## Policies and policy classes

- help in implementing safe, efficient, and highly customizable design elements.

- A policy defines a class interface or a class template interface

- new

template <class T>

struct OpNewCreator

{

static T\* Create()

{

return new T;

}

};

- malloc

template <class T>

struct MallocCreator

{

static T\* Create()

{

void\* buf = std::malloc(sizeof(T));

if (!buf) return 0;

return new(buf) T;

}

};

- clone

template <class T>

struct PrototypeCreator

{

PrototypeCreator(T\* pObj = 0)

:pPrototype\_(pObj)

{}

T\* Create()

{

return pPrototype\_ ? pPrototype\_->Clone() : 0;

}

T\* GetPrototype() { return pPrototype\_; }

void SetPrototype(T\* pObj) { pPrototype\_ = pObj; }

private:

T\* pPrototype\_;

};

- class that exploits the Creator policy

// Library code

template <class CreationPolicy>

class WidgetManager : public CreationPolicy

{

};

## Implementing Policy Classes with Template Template Parameters

// Library code

template <class CreationPolicy>

class WidgetManager : public CreationPolicy

{

};

// Library code

template <template <class> class CreationPolicy>

class WidgetManager : public CreationPolicy<Widget>

{

...

void DoSomething()

{

Gadget\* pW = CreationPolicy<Gadget>().Create();

...

}

};

## with Template Member Functions

struct OpNewCreator

{

template <class T>

static T\* Create()

{

return new T;

}

};

## Destructors of Policy Classes

template <class T>

struct OpNewCreator

{

static T\* Create()

{

return new T;

10}

protected:

~OpNewCreator() {}

};

destructor is protected:

- only derived classes can destroy policy objects → it's impossible for outsiders to apply delete to a pointer to a policy class.

- The destructor is not virtual → there is no size or speed overhead.

## Combining Policy Classes

template

<

class T,

template <class> class CheckingPolicy,

template <class> class ThreadingModel

>

class SmartPtr :

public CheckingPolicy<T>,

public ThreadingModel<SmartPtr>

{

...

T\* operator->()

{

typename ThreadingModel<SmartPtr>::Lock guard(\*this);

CheckingPolicy<T>::Check(pointee\_);

return pointee\_;

}

private:

T\* pointee\_;

};

# Techniques

## Compile-Time Assertions

developing a function for safe casting: larger types must not be cast to smaller types

template <class To, class From>

To safe\_reinterpret\_cast(From from)

{

assert(sizeof(From) <= sizeof(To));

return reinterpret\_cast<To>(from);

}

int i = ...;

char\* p = safe\_reinterpret\_cast<char\*>(i);

template<bool> struct CompileTimeChecker

{

CompileTimeChecker(...);

};

template<> struct CompileTimeChecker<false> { };

#define STATIC\_CHECK(expr, msg) \

{\

class ERROR\_##msg {}; \

(void)sizeof(CompileTimeChecker<(expr) != 0>((ERROR\_##msg())));\

}

## Partial Template Specialization

class template Widget ,

template <class Window, class Controller>

class Widget

{

... generic implementation ...

};

explicitly specialize the Widget class

template <>

class Widget<ModalDialog, MyController>

{

... specialized implementation ...

};

ModalDialog and MyController are classes defined by your application.

## Local Classes

define classes right inside functions

void Fun()

{

class Local

{

... member variables ...

... member function definitions ...

};

... code using Local ...

}

- local classes cannot define static member variables and cannot access nonstatic local variables

- use them in template functions: MakeAdapter in the following code adapts one interface to another.

class Interface

{

public:

virtual void *Fun*() = 0;

...

};

template <class T, class P>

Interface\* MakeAdapter(const T& obj, const P& arg)

{

class Local : public Interface

{

public:

Local(const T& obj, const P& arg)

: obj\_(obj), arg\_(arg) {}

virtual void *Fun*()

{

obj\_.Call(arg\_);

}

private:

T obj\_;

P arg\_;

};

return new Local(obj, arg);

}

## Mapping Integral Constants to Types

Int2Type generates a distinct type for each distinct constant integral value passed

template <int v>

struct Int2Type

{

enum { value = v };

};

use Int2Type whenever you need to "typify" an integral constant quickly.

- You need to call one of several different functions, depending on a compile-time constant.

- You need to do this dispatch at compile time

design a generic container NiftyContainer

template <class T> class NiftyContainer

{

...

};

NiftyContainer contains pointers to objects of type T . To duplicate an object contained in NiftyContainer , you want to call either its copy constructor (for nonpolymorphic types) or a Clone() virtual function (for polymorphic types).

template <typename T, bool isPolymorphic>

class NiftyContainer

{

...

void DoSomething()

{

T\* pSomeObj = ...;

if (isPolymorphic)

{

T\* pNewObj = pSomeObj->Clone();

...

polymorphic algorithm ...

}

else

{

T\* pNewObj = new T(\*pSomeObj);

... nonpolymorphic algorithm ...

}

}

};

problem is that the compiler won't let you get away with this code. For example, because the polymorphic algorithm uses pObj->Clone() , NiftyContainer::DoSomething does not compile for any type that doesn't define a member function Clone().

template <typename T, bool isPolymorphic>

class NiftyContainer

{

private:

void DoSomething(T\* pObj, Int2Type<true>)

{

T\* pNewObj = pObj->Clone();

... polymorphic algorithm ...

}

void DoSomething(T\* pObj, Int2Type<false>)

{

T\* pNewObj = new T(\*pObj);

... nonpolymorphic algorithm ...

}

public:

void DoSomething(T\* pObj)

{

DoSomething(pObj, Int2Type<isPolymorphic>());

}

};

Int2Type comes in very handy as a means to translate a value into a type. You then pass a temporary

variable of that type to an overloaded function. The overloads implement the two needed algorithms.

## Type-to-Type Mapping

template <class T, class U>

T\* Create(const U& arg)

{

return new T(arg);

}

Create makes a new object, passing an argument to its constructor.

Objects of type Widget are untouchable legacy code and must take two arguments upon construction

→ overloading

template <class T, class U>

T\* Create(const U& arg, T /\* dummy \*/)

{

return new T(arg);

}

template <class U>

Widget\* Create(const U& arg, Widget /\* dummy \*/)

{

return new Widget(arg, -1);

}

→Type2Type

template <typename T>

struct Type2Type

{

typedef T OriginalType;

};

Factory

creating families of related or dependent objects without specifying their concrete classes

Use

- a system should be independent of how its products are created, composed, and represented.

- a system should be configured with one of multiple families of products.

- a family of related product objects is designed to be used together, and you need to enforce this constraint.

- you want to provide a class library of products, and you want to reveal just their interfaces, not their implementations.