Advanced Programming in C++

Exercise – Policy-based design for Smart Pointers

In this exercise you shall, given a very simple definition of a smart pointer type, make the smart pointer type more flexible by allowing for variation of a number of design aspects. Each such design aspect is called a *policy* and takes care of one behavioural or structural aspect of the smart pointer type. You will design a number of such policies and assemble the smart pointer class out of these policies. For each policy there will be a number of variations and, since these can be mixed, the behaviour of the smart pointer type can be varied i a number of ways.

If you have done the separate smart pointer exercise, you may add on the policies for smart pointers described below to the smart pointer class achieved there. It may, though, be substantially simpler to first do this exercise and then add on policies to that far more advanced smart pointer class.

About policies

Policies can define as classes, even if it in some cases could be possible to implemented a policy with simpler means. A policy class establishes an interface to the specific issue, and typically consist of inner type definitions, member functions and data members. If a policy class is to be parameterized on a type, one can either refine the policy as a template class, or a class with template member functions.

An example of a policy is how objects of some type `T` is to be created. Such a *creator policy* should be a class parameterized on some type `T`, and have a member function `create()` to return a pointer to a new object of type `T`. One way to create new objects is to use the **new** operator, another way is to use `malloc` to allocate memory and then invoke the placement **new** operator to construct the object in that memory, and if the type `T` have a `clone()` member function a third way could be to call `clone()`. These three variations of the creator policy could be implemented as:

```
template<typename T> struct new_creator
{
  static T* create() { return new T; }
};

template<typename T> struct malloc_creator
{
  static T* create()
  {
    void* buffer{ std::malloc(sizeof(T)) };
    return buf ? new(buf) T : nullptr;
  }
};

template<typename T> class clone_creator
{
  public:
    clone_creator(T* p = nullptr) : prototype_(p) {}
    T* create() { return prototype_ ? prototype_->clone() : nullptr; }

    T*   get_prototype()     { return prototype_; }
    void set_prototype(T* p) { prototype_ = p; }

  private:
    T* prototype_;
};
```

All three policy classes above define the create() member function to conform to the creator policy, but as also shown policy classes for a given policy may have different implementations and also different interfaces. The clone_creator class have a data member prototype_ which the other two do not have, but more notable clone_creator have two member functions, get_prototype() and set_prototype(), that are not required by the creator policy.

A class that exploits a policy is called a *host class*. A policy is declared as a template parameter of the host class and the host may then either contain the policy class:

```
template<typename T, class <typename> class creator_policy = new_creator>
class host
{
     …
  private:
     creator_policy<T> policy_;
};
```

or inherit the policy class, with suitable access specification:

```
template<typename T, class <typename> class creator_policy = new_creator>
class host : public creator_policy<T>
{
  …
};
```

or, if the policy members are **static**, use the template parameter directly in expressions.

Whenever an object of the host class needs to create a T object it invokes create() for its creator_policy<T> member:

```
T* p{ policy_.create() };
```

or creator_policy<T> subobject (the class prefix is not needed, unless another create() member is present):

```
T* p{ creator_policy<T>::create() };
```

If a host class exploits several policies it is desired that the policies are *orthogonal*. This means that it should be possible to combine them without any undesired effects for certain combinations or that certain combinations are not allowed.

Note: It's often better to declare a policy class as a non-template, and instead the policy member functions as member templates.

Smart pointers and policies

For a smart pointer type there are a number of variation points, each of which can translate into a *policy*:

- A *storage policy* concerns variation of the stored type. By default the stored type is a raw pointer to the referred type T, that is T*, but there are other possibilities.
- An *ownership policy* concerns ownership, for which there are a number of variations, such as *deep copy*, *destructive copy*, *no copy*, *reference counting* and *reference linking*.
- A *conversion policy* is about whether conversion to the raw pointer type is to be allowed or not.
- A *checking policy* decides if, and in such case how, to check for valid initializers and if to check if a smart pointer is valid for dereferencing.

Below are a number of exercises in which you are to successively modify the given smart pointer class to support these policies.

Given smart pointer class

The following simple definition of a smart pointer class implements *deep copy*:

```
template<typename T>
class smart_pointer
{
public:
  smart_pointer() = default;

  explicit smart_pointer(T* p) : ptr_{ p } {}

  smart_pointer(const smart_pointer& rhs) { ptr_ = copy(rhs); }

  ~smart_pointer() { delete ptr_; }

  smart_pointer& operator=(const smart_pointer& rhs)
  {
    if (this != &rhs)
    {
      delete ptr_;
      ptr_ = copy(rhs);
    }
    return *this;
  }

  T& operator*() const { return *ptr_; }
  T& operator*() { return *ptr_; }

  T* operator->() const { return ptr_; }
  T* operator->() { return ptr_; }

private:
  T* ptr_;

  T* copy(const smart_pointer& sp)
  {
    return sp.ptr_ ? new T{ *sp.ptr_ } : nullptr;
  }
};
```

Move constructor, move assignment operator, and swap functions could be added.

Some techniques

We start this exercise by looking at some useful techniques, *compile-time checks* and *type selection*. Both are based on templates and demonstrates the power of templates.

Compile-time checks

Sometimes we want to generate compile-time errors, based on some predicate.

One example of such a case is if we have a number of related classes, for example policy classes, implementing a common interface as a set of member functions. Suppose, that for one of the classes one of the member functions is not allowed to invoke, because that variation of the policy does not allow for that service. The simple way to solve this would be to just remove that member function from that class, but the compiler message due to the missing function might not give us any hint of the real nature of the error.

```
template <typename T> class X
{
public:
   void memfun(const T&)
   {
      // Something that would generate a, hopefully somewhat informative compile-time error
   }
   …
};
```

The standard assert macro can not be used, since such assertions are performed during runtime. Instead we would like to have some kind of compile-time assertion, one that preferably also makes the compiler say something informative about the nature of the error.

One solution is to declare a template with a informative name, for example compile_time_error, which hopefully the compiler will mention in its error message. The template shall have one parameter of type **bool**. In this case we get what we need by declaring the primary, and only supply a specialization for **true**. If an attempt is made to instantiate for **false**, this will give a compile error, since there is no definition.

```
template <bool> struct compile_time_error;
template <>     struct compile_time_error<true> {};
```

To simplify the use of compile_time_error a function macro as the following can be declared:

```
#define STATIC_CHECK(expr) { compile_time_error<((expr) != 0)> ERROR; }
```

There is one improvement that can be done, and that is to supply an error message. This is solved by using the preprocessor ## operator. If ## is used in a macro definition and immediately followed by a parameter, the parameter is replaced by the corresponding argument (to put it simple):

```
#define STATIC_CHECK(expr, msg) \
{ compile_time_error<((expr) != 0)> ERROR_##msg; }
```

The argument must be an identifier and we therefore use underscores instead of spaces to compose a message:

```
STATIC_CHECK(false, This_policy_do_not_allow_copy);
```

will result in an error message hopefully containing the string:

```
ERROR_This_policy_does_not_allow_copy
```

Type selection

Sometimes we need to select one of two types, depending on a Boolean constant. Some examples:

- Select the type of a function parameter. For a class X, the copy constructor normally have a parameter of type *reference-to-const*-X:

  ```
  X(const X& rhs);
  ```

  but in some circumstances we may instead need to have a parameter of type *reference-to*-X:

  ```
  X(X& rhs);
  ```

- Select the return type of a function. We want to have either have a return type that is the same as (or at least compatible with) the type of the return expression:

  ```
  int* fun(int n) { return new int{ n }; }
  ```

  or a type incompatible with the return expression, to causes a compile-time error:

  ```
  incompatible fun(int n) { return new int{ n }; }
  ```

  The type incompatible could for example be defined as:

  ```
  class incompatible {};
  ```

The type selection must be static, that is, only compile-time constructs can be used. This can be achieved by defining a template struct (no need for privacy) with three template parameters, a **bool** parameter and two type parameters, T1 and T2, and an inner type definition. In the case that the **bool** argument is **true**, we want the inner type definition to define T1, and in the case that the **bool** parameter is **false**, we want the inner type definition to define T2. To make this happen we must define a specialization of the template for one case, for example **false**, and let the **true** case be automatically instantiated from the non-specialized version.

Define a template struct named select_type, with three parameters, one **bool** parameter and two type parameters T1 and T2, that is a template with the following prototype:

```
select_type<bool selector, typename T1, typename T2>
```

The struct shall define an inner type result, which in this non-specialized version defines T1.

Define a specialization of select_type for **false**, where the inner type definition defines T2.

Use select_type in the following way, where flag is a **bool**:

```
select_type<flag, int*, incompatible>::result
```

Depending in flag, result will be either **int*** or incompatible.

Implement and test.

1. Conversion policy

The conversion policy specifies whether conversion to the raw pointer type is to be allowed or not.

```
smart_pointer<X> sp{ new X{ 1 } };
X* p;
p = sp;   // Calls for conversion from smart_pointer<X> to X* – allowed or not?
```

If allowed, a type conversion operator defined as below should exist:

```
operator T*() const { return ptr_; }
```

If not allowed a compile-time error should instead be issued, whenever a smart pointer object is used in a way that calls for an implicit conversion to the raw pointer type.

Type conversion is either to be *allowed* or *disallowed*, so a straightforward solution would be to use a **bool** parameter for the conversion policy. Declaring smart pointer objects could then look like:

```
smart_pointer<Y, true>  sp1;
smart_pointer<Y, false> sp2;
```

One negative side of such a simple solution is that the instantiation arguments **true** and **false** gives no hint what it is about. One of the benefits of defining policies as classes, and of course naming the classes appropriately, is readability.

Define one struct allow_conversion and another disallow_conversion. Each shall have a **bool** member named allow, which states whether type conversion is allowed or not.

Add a parameter named conversion_policy to smart_pointer (with, for example, disallow_conversion as default argument).

Now to the more tricky part, that is how to allow and disallow for type conversion? When type conversion is allowed, a type converting operator, as the following, should exist:

```
operator T*() const { return ptr_; }
```

A way to disallow type conversion is to let the type converting operator return some other type than T*, say *disallowed*, for which there is no conversion from T*, that is ptr_ can not be converted to type *disallowed*:

```
operator disallowed() const { return ptr_; }
```

The compiler will then report something like “Can not convert from T* to *disallowed*”. Se **Type selection** above for how to solve this.

Test the conversion policy.

2. Checking policy

There are several variations for checking a pointer on initialization and assignment, and before dereferencing or indirecting. Here are some possibilities:

- No checks. This means that initialization and assignment to null is allowed and also that no check for null is done before dereferencing or indirecting. Dereferencing or indirecting a null pointers will cause a program to terminate abnormally.
- Allow for a smart pointer to be initialized or assigned to null, but check the pointer before dereferencing and indirecting and throw an exception if the pointer is null.
- Throw an exception if a smart pointer is initialized or assigned to null, and also if a null pointer is dereferenced or indirected.
- Allow for null pointers when initializing and assigning, but use assert for checking before dereferencing and indirecting. The program will terminate with an assert message, if an attempt is made to dereference or indirect a null pointer.
- Use assert for checking both when initializing, assigning, dereferencing and indirecting. An attempt to initialize or assign a smart pointer to null, or to dereference or indirect a null pointer will terminate the program with an assert message.

Design a policy class named no_checks for the case *no checks*. The class shall be parameterized on the *pointer type* and have (at least) three member functions, one for checking on default initialization, one for checking on explicit initialization and assigning, and one for checking when dereferencing or indirecting. All should have a parameter of the pointer type but when called not perform any kind of operation on their argument. You have two choices regarding parameterized on the pointer type, either you could make the policy class a template, or you could let the member functions be templates.

Add a template parameter to smart_pointer for the checking policy, and let no_check be default policy.

Insert a call to the appropriate member function of the checking policy class into each function of smart_pointer which shall do checking.

Compile and run the test program to verify the design of the policy class and its use.

Define an exception, null_pointer_exception, to be thrown in cases when an exception is to be thrown on null pointer detection, and implement the exception-throwing variants of the checking policy.

Implement the assert-using variants of the checking policy.

Test the different variations of the checking policy.

3. Storage policy

A storage policy allow for variants of the stored type. Usually the stored type is the pointer type T*, if the pointee type is T:

```
template <typename T>
smart_pointer
{
  …
private:
  T* ptr_;
}:
```

Other variants may concern the stored type as such but also other aspects, such as locking an object for multiple access.

To be able to vary the stored type, it is moved from the smart pointer class to the storage policy classes. The storage policy classes must then define an interface, so the smart pointer member functions can operate on the implementation without knowing the exact nature of the stored type.

The interface shall give the impression that the implementation is a pointer type. For example shall smart_pointer<T>::**operator**-> return a T*. To get that pointer it calls a member function in the policy class, and therefore the policy class member function must return a T*.

Destruction, default value for the pointer type, and cloning of the pointee must be part of the interface.

Define a policy class default_storage, which uses T* as the stored type. It is basically a straight forward exercise.

Add a template parameter named storage_policy to the smart pointer class, with default_storage as default argument, and let smart_pointer derive from storage_policy.

Change the smart_pointer member functions to operate via the member functions of thew policy class.

Test your design.

Hints

It should now have become obvious that using class prefixes such as storage_policy<T>:: makes names very long and the code hard to read. It is therefore a good idea to typedef shorthands for such things, e.g. SP for storage_policy<T>, and use such shorthands.

The storage policy classes should also typedef names for the pointer type (T*) and the reference type (T&), say pointer_type and reference_type, to be used by member functions in the storage policy classes and also in smart_pointer.

After ownership policy is implemented, you can try to define a policy class array_storage to handle pointers to arrays. The primary issue is to call **new**[] instead of **new** to allocate memory for an array object, and **delete**[] instead **delete** to delete an array object. A problem with arrays is that their size is not known within functions.

4. Ownership policy

Ownership is about what shall happen when smart pointers objects are initialized, copied and destroyed. There are a number of well known ownership policies, for example:

- *Deep copy* basically means that in an initialization or an assignment, a full copy of the object of the source smart pointer is created and a pointer to that copy is used to initialize or assign to the destination smart pointer. Each smart pointer have its own instance of an object and is therefore responsible for deleting the object appropriately.
- *Destructive copy* means that whenever a smart pointer is used to initialize och assign to another smart pointer, the object is transferred to the destination smart pointer and the source smart pointer is set to null. Therefore, an object always belongs to a certain smart pointer, and that smart pointer is responsible for the deletion of the object.
- *No copy* do not allow any copying, that is, the copy constructor and copy assignment is not allowed.
- *Reference counting* allow smart pointers to share an object and uses a reference count to keep track of how many smart pointers that are referring to an object. When a smart pointer is the last referrer to an object, it is also responsible to delete the object. There are several alternatives for implementing reference-linked smart pointers. This is the most used ownership strategy.
- *Reference linking* links together all smart pointers referring a certain object in a double linked list, an ownership list. When the list is to become empty, the object is deleted.

Reference counting is a bit difficult to implement, while the other strategies are easy. It will turn out that ownership and storage policies are not orthogonal when it comes to more advanced ownership policies as reference counting, and some changes to the storage policy because of this is probably needed.

Destructive copy is a bit tricky to mix with the other strategies. The parameter of the smart pointer copy constructor and copy assignment operator can not be *reference-to-const* for destructive copy, since the source smart pointer must be set to null. But, fortunately, we already know how to select a type.

There are two operations basically associated with the ownership policy, *clone* and *release* of objects.

- clone() shall take a pointer to the object to be cloned as argument, clone the object, and return a pointer to the cloned object.
- release() may needs some hints to get right. It shall take a pointer to the object to be released, possibly perform some actions associated with the release, but *not* actually release the object. Instead release() shall return **true** or **false**, and if **true** the caller of release() shall delete the object. For simple strategies, release() shall normally only return **true**. In case of *reference counting*, it is release() that shall be responsible for decrement the reference count and return **true** if it reaches 0, and possibly do something more, depending on the exact implementation of the reference counting.

Each policy also have to declare if it is *destructive copy*, or not, to make it possible to choose the parameter type of the copy constructor and copy assignment operator.

Define policy classes for *deep copy* (the given strategy), *destructive copy* and *no copy*.

One difficulty is how to select the parameter type for the copy constructor and copy assignment operator. It shall be *reference-to-const* for all strategies except destructive copy, for which it shall be *reference-to*. The trick to be used is described in the conversion policy exercise.

Another difficulty is how to disallow copy initialization and assignment for *no copy*. A readable compile-time error would be desired. Se **Compile-time checks** above for a way to solve this.

Define a policy class for reference counting. Allocate the count dynamically, as an **int**, and let all smart pointers sharing an object all point to that dynamic **int**.