Objective Viewpoint

Getting the Handle of Handles

by G. Bowden Wise

As C++ programmers, we design classes to represent concepts from the problem domain in which our programs reside. We provide a public interface for the class so that users may interact with class objects. The underlying implementation details are hidden in the private part of the class, inaccessible to the class's users. This is the all familiar object-oriented concept of *encapsulation*.

Usually, a class is implemented by defining the data structures and algorithms necessary to support its operations (by defining private data and function members). However, sometimes it is undesirable to place all of the implementation details directly into a class. Instead, the implementation details are placed in a second class, called the *representation*, or simply, the *rep*, for short. The actual class contains a pointer to the rep.

Classes which have a pointer to another class which contains the actual implementation are called *handle* classes. When a handle object receives a message to perform one of its operations, the message is forwarded to the rep for actual processing. Coplien [2] calls this the *handle/body* idiom where the representation is considered the body, and the outer class the handle.

The handles technique is very useful and should be a part of every object-oriented programmers toolbox. Handles, and their variations, allow you to:

- hide implementation details;
- minimize the impact of changes during development;
- determine the type of an object from the context in which it is constructed;
- create objects when an object's size is not known;
- provide multiple representations of an object;
- delegate operations to another class for processing; and
- change the type (representation) of an object at run-time.

In this issue of OBJECTIVE VIEW POINT, we will look at some of these uses of handles in more detail.

Handles

The general form for a handle class is:

```
class Handle
{
public:
   Handle();
   ~Handle();
   void foo();
private:
   Rep* _theRep;
};
```

The class `Rep` contains the actual implementation details. When a user invokes `foo()` using an instance of the `Handle` class, the rep pointer is used to pass the request to the actual underlying representation:

```
void Handle::foo()
{
   // Pass operation to the rep
   _theRep->foo();
}
```

As we will see, the handle technique is very useful for allowing classes to interact with other classes in various ways. Handles do have a few disadvantages, however. They increase the size of the object by the addition of the rep pointer. Each time an operation is invoked by the handle class through the rep pointer, a little more time is spent performing the extra indirection (though this might be optimized away). Also, functions in the handle class that use any members of the rep class cannot be made inline, since the definition of the rep is not provided.

One strategy is to use handles during development to minimize the impact of changes on client code. Later, when it can be shown that the differences in size or speed is significant, the handle can be replaced by a more concrete class that does not use a handle.

Hiding Implementation Details

C++ supports encapsulation and data abstraction through the use of the `public` and `private` parts of a class definition. A class's public interface is placed in the public part of the class, while the hidden implementation details are placed in the private part.

Note, however, that C++ does not really hide the implementation details of a class from its users. After all, a user need only look at a class's header file, and examine the private part of the class definition to glean some insight into how the class has been implemented. C++ only prevents users from *accessing* the data and function members defined in the private part of the class.

Sometimes you may not want your users to see how you have implemented your class. You may be using a proprietary algorithm, and wish to shield the implementation details from prying eyes. By using a handle class, you can do just that.

Suppose you have developed a new way to encrypt data, but you do not want your users know the details of your algorithm. How do you provide your users with the ability to encrypt and decrypt data without giving them the details of your implementation? Simple, you implement two header files: a public one, which you give your users, and a private one for your internal use. Carolan [1] has appropriately dubbed this particular technique the *Cheshire cat*.

The public header file, which *is* distributed to your users, looks like this:

```
// File: cypher.h
// Forward class definitions
class TheCypher;

// The handle class
class DataCypher
{
public:
   DataCypher();
   ~DataCypher();
   bool encrypt(const char* buf,
                char*&      result);
   bool decrypt(const char* buf,
                char*&      result);
private:
   TheCypher* _theCypher;
};
```

The `encrypt()` and `decrypt()` routines simply forward their operations to the representation through the rep pointer, `_theCypher`, just as we saw in `Handle::foo()` above.

The private header file, which *is not* distributed to the class's users, contains the proprietary information about the hidden implementation:

```
// File: cypherP.h
// Include the public header
#include "cypher.h"

// The private rep class:
class TheCypher
{
private:
   friend class  DataCypher;
   TheCypher();
   bool encrypt(const char* buf,
                char*&      result);
   bool decrypt(const char* buf,
                char*&      result);
   // Details omitted
};
```

Note that all of the members of `TheCypher` have been made `private` and that only `DataCypher` may create objects of type `TheCypher` since it is a `friend`. The private header is used to compile the implementations of both `DataCypher` and `TheCypher`:

```
#include "cypherP.h"

TheCypher::TheCypher()
{
   // Details omitted
}

DataCypher::DataCypher()
: _theCypher (new TheCypher)
{
}
// Other details omitted
```

As long as the private header file is never distributed, users of `DataCypher` will never learn about the implementation details of the proprietary encryption algorithm.

Minimizing the Impact of Changes

During development many changes are made to a class's implementation. These changes will almost always propagate and require that any code that uses that class to also be recompiled. Modifications to a class, such as

- a change to an inline function definition;
- a change to the size of an object (e.g., by adding or deleting a data member, for example); or
- a change to the structure of an object.

all require a recompilation.

Using a handle to the representation can make recompiles of the user code unnecessary. A change to the underlying representation will require that the implementation be recompiled, but the user code using the handle class will only require a relink. For example, whenever `TheCypher` is changed, the handle class `DataCypher` will not have to be recompiled, only relinked.

Providing Multiple Representations

Handles can also be used to provide multiple representations for an object. The rep pointer can point to any of several representations as long as they all share the same interface. To do this, the rep pointer is made a pointer to an *abstract base class*. The multiple representations are then descendents of the abstract base class.

An abstract base class contains almost pure interface --- implementation details are usually non-existent. In practice, there may also be some function or data members provided in the base class; but these should be functionality that is common to all derived classes.

In C++, *pure virtual functions* are used to specify the interface to a member function without providing an implementation for the function. Descendents of the abstract base class become *concrete classes* by providing specific implementations for each of the virtual functions specified in the abstract base class. By defining the interface up front, we can change the rep to point to any of the classes derived from the abstract base class, thereby changing the representation of the object.

To illustrate, suppose we are designing an error reporting mechanism for a graphical user interface framework. We do not want to worry about how the error will be displayed at this point. We might provide a very simple error generator initially, but give our users the flexibility of adding new error generators without requiring any changes to existing code.

Our error generator will have an abstract base class, `ErrorGen`, which contains the interface that all error generator representations must provide:

```
class ErrorGen
{
public:
   ErrorGen() {}
```

```
      virtual ~ErrorGen() {}
      virtual void ReportError
              (const char* info) = 0;
   };
```

Different representations for the error generator can be provided by deriving them from `ErrorGen` and providing an implementation of the pure virtual function, `ReportError()`. In this way, several error generators can be provided: `FileErrorGen`, which writes the error to a file; `MBoxErrorGen`, which displays the message in a message box; and `SBarErrorGen`, which displays the message in a status bar.

The handle class looks like this:

```
   class ErrorReporter
   {
      public:
         ErrorReporter(ErrorGen* gen=0);
         ~ErrorReporter();
         void ReportError(const char* info);
         void SetGenerator(ErrorGen* gen=0);
      private:
         ErrorGen* _errorGen;
   };
```

As we have seen previously, the handle class forwards operations to its rep. In this case, `ReportError` forwards the request to the error generator representation:

```
   void
   ErrorReporter::ReportError
                  (const char* info)
   {
      _errorGen->ReportError(info);
   }
```

The handle class does not care about the underlying implementation, it simply forwards the request.

The only remaining issue that needs to be addressed is where to create the error reporter object. One strategy is to make it global so that it is available throughout the entire program. At the start of the program, the global error reporting object must be initialized with a particular error generator. There are several scenarios which may be used. Users may be required to specify the error generator when the global object is created:

```
   errorReporter = new ErrorReporter
                       (new MBoxErrorGen);
```

Another strategy would be to use a default error generator when the global object is first created, but then allow the user to change the representation at run-time by calling the `SetGenerator()` member function:

```
   errorReporter->SetGenerator
                     (new SBarErrorGen);
```

Then, whenever an error needs to be reported, `ReportError()` is called. For example, suppose a memory allocation failure has occurred:

```
   char errorMsg = "Memory failure";
   errorReporter->ReportError(errorMsg);
```

To build more sophistication into the error reporter, the type of representation could be determined automatically based on some setting such as an environment variable. We will see more about how to delay the construction an of object until run-time when virtual constructors are discussed in the next section.

Virtual Constructors

Sometimes we do not know what the type of an object will be until run-time. The actual construction of an object must be delayed until the type is known and then the object can be created. This is a problem for strongly-typed languages, like C++, which do not allow the type of an object to change at run-time. But, it is possible to change the representation of a handle at run-time. Changing the representation of a handle effectively changes the object's type.

Suppose you are developing software for a large bank. The bank has many different kinds of accounts, including checking accounts, savings accounts, and money market accounts. Furthermore, suppose that each account has a special prefix code that identifies the type of account: `` `CC' `` for checking accounts, `` `SS' `` for savings accounts, and `` `MM' `` for money market accounts. Your task is to develop a program that reads in account balances from a database in order to initialize the objects for the program.

All accounts have an account identifier and a total balance. In addition, each account may have credits or debits applied to the balance, end-of-day processing performed, and monthly charges deducted. With this description, we can define the abstract base class for the accounts:

```
class Account
{
```

```
public:
   Account (const char* acctID,
            const float initial)
   : balance(initial)
   { strcpy (id, acctID); }
   virtual ~Account() {}
   virtual const char* Id()
      { return id; }
   virtual float Balance()
      { return balance; }
   virtual float Credit(const float amt)
      { balance += amt; }
   virtual float Debit(const float amt)
      { balance -= amt; }
   virtual void  PostDay() = 0;
   virtual void  MonthlyCharge() = 0;
private:
   float balance;
   char id[8];
};
```

Note that `PostDay()` and `MonthlyCharge()` are pure virtual functions and the classes `CheckingAccount`, `SavingsAccount`, `MoneyMarketAccount`, all must provide versions for these functions specific to those types of accounts. Savings accounts and money market accounts will also have an additional data member to keep track of the daily interest rate so that interest can be accrued when end-of-day processing is performed on the account.

A handle class is defined which can represent any of these different types of accounts:

```
class BankAccount
{
public:
   BankAccount(const char* id,
               const float bal);
   virtual ~BankAccount()
      { delete _account;  }
   virtual const char* Id()
      { return _account->Id(); }
   virtual float Balance()
      { return _account->Balance(); }
   virtual float Credit(const float amt)
      { return _account->Credit (amt);  }
   virtual float Debit(const float amt)
      { return _account->Debit(amt);  }
   virtual void  PostDay()
      { _account->PostDay();  }
   virtual void  MonthlyCharge()
      { _account->MonthlyCharge();  }
private:
   Account* _acct;
};
```

Note that all of the operations are simply forwarded to the rep pointer, `_acct`, for processing. Class `BankAccount` has a *virtual constructor* which examines the account identifier to determine what type of account is to be constructed. Here is the constructor:

```
BankAccount::BankAccount (const char* id,
                          const float bal)
{
   if (id[0] == 'C'  && id[1] == 'C')
   {
      _acct=new CheckingAccount(id,bal);
   }
   else if (id[0] == 'S'  && id[1] == 'S')
   {
      _acct=new SavingsAccount(id,bal);
   }
   else if (id[0] == 'M'  && id[1] == 'M')
   {
      _acct=new MoneyMarketAccount(id,bal);
   }
}
```

Virtual constructors can be used whenever the creation of an object must be delayed until run-time when some other context information is available to determine the object's type.

To Probe Further

Handles are described in many different contexts by Murray [3]. The handle/body idiom, virtual constructors, and other useful C++ programming idioms are beautifully discussed by James Coplien in his book on advanced C++ idioms [2].

In a future installment of this column, we will look at some more advanced uses of handles, including delegation and more dynamic ways to change an object's type at run-time.

References

1
CAROLAN, J. Constructing bullet-proof classes. In *Proceedings of C++ At Work '89* (1989), SIGS Publications.

2
COPLIEN, J. O. *Advanced C++ Programming Styles and Idioms*. Addison-Wesley, Reading, Mass., 1992.

3
MURRAY, R. B. *C++ Strategies and Tactics*. Addison-Wesley, Reading, Mass., 1993.