M. D. Wilson, "Veneers – A Definition"

The concept of veneers is exceedingly simple. A veneer is a template class with the following characteristics:

1) It derives from its primary parameterising type, usually publicly
2) It does not define any virtual methods
3) It does not define any non-static member variables, including not defining a virtual destructor
4) Further to 2) and 3), it does not increase the memory footprint of the parameterised composite type over that of the parameterising type.

Veneers usually modify the behaviour – supplementing existing functionality with additional functionality obtained from the secondary parameterising types – or the type of the parameterising type.

Consider a situation where you wish to inject functionality into an already rich hierarchy. An example of this could be when using the ATL [1] class, `CWindow`. The class does not have a member function to get the text length of a dialog item. Making use of the Win32 API function `GetWindowTextLength()`, and the member function `GetDlgItem()`, we can create the veneer class `parent_window_veneer` [SY-ATL], defined as follows:

```
template <typename T>
class parent_window_veneer
    : public T
{
public:
    typedef T                       ParentClass;
    typedef parent_window_veneer<T> Class;

// Construction
public:
    ...

// Operations
public:
    int GetDlgItemTextLength(UINT id)
    {
        return ::GetWindowTextLength(GetDlgItem(id));
    }
};
```

Wherever one might use the `CWindow` class, `parent_window_veneer<CWindow>` (or a typedef specifying such) can be used instead, and the method `GetDlgItemTextLength()` is accessible on instances of that type, or any derived types. Of course, this could be achieved with conventional non-template derivation. However, if one later had reason to want to see this function in another ATL windowing class, say `CContainedWindow`, then another derived class would be needed, and so on. The veneer can be easily used with other types of windowing classes which have a

compatible `GetDlgItem()` member function, and all that is needed is to change the declaration, usually amounting to a single change to a typedef .

Veneers also can provide a destructor, although one needs to be careful and respect that if the parameterising type is not a polymorphically derivable type (i.e. if it does not have a virtual destructor) then it should not be used polymorphically.

Finally, because veneers are the same size as, and publicly inherit from, their parameterising types, they can be used to substitute for them in situations requiring arrays.

The advantages of veneers are, then:

- They facilitate the injection of code into an inheritance hierarchy without duplication of code
- They can be used to add the automatic cleaning up of resources (via the Resource Acquisition Is Initialisation idiom [3]) into existing classes. (The STLSoft libraries' [4] container veneers provide this functionality for container classes in the C++ standard library)
- The size restrictions means that the parameterised veneer type can be used in array form without experiencing the problems [5] associated with polymorphic arrays

Some real examples of veneers are available from the STLSoft libraries [4], and include `pod_veneer`, `conversion_veneer`, `sequence_container_veneer` and `associative_container_veneer`. For more detail on the concept of veneers, please consult the documentation for these classes.

Notes and References

[1] The ActiveX Template Library. This ships with Microsoft's Visual C++ compiler (http://www.microsoft.com), and Digital Mars C++ compiler (http://digitalmars.com)
[2] The Synesis Software Public-domain Source Code Library contains a fully-fledged version of this veneer class, available online at http://synesis.com.au/software
[3] Bjarne Stroustrup, "The C++ Programming Language", Third Edition, Addison-Wesley, 1997
[4] The STLSoft libraries are available online at http://stlsoft.org
[5] Scott Meyers, "More Effective STL", Addison-Wesley, 1997. Item #3