M. D. Wilson, "Shims – A Definition"

Shims are lightweight components – comprising free-functions and smart-pointer classes – which are used in the manipulation and conversion of types. They facilitate generalised manipulation of sets of types that are related conceptually but not by type, e.g. raw and smart pointers, C-style strings and `std::basic_string<>`.

A shim is an unbound set of functions sharing four characteristics:

- **Name**. The name of the shim corresponds to the name of the shim functions, e.g. the functions of the **c_str_ptr** shim are all `c_str_ptr()`.
- **Namespace**. All functions for a given shim reside in a common namespace
- **Category**. One of the four main shim categories – *Attribute Shim*, *Control Shim*, *Conversion Shim*, *Logical Shim* – or a composite category (comprising the features of two or more fundamental categories).
- **Result Type**. The result type(s) of the functions of a given shim are depending on the shim category. In the case of *Conversion Shims*, all shim functions return a common result type `R`, or an instance of a type that is implicitly convertible to `R`.

The following fundamental shim categories are defined:

Fundamental Shim Categories

Attribute Shims

Attribute shims access attributes, or state, of the (instances of the) types for which types they are defined.

Attribute shims are named according to the `get_xxx()` form, e.g. `get_ptr()`.

Example:

```
template <typename T>
inline T *get_ptr(T *p)
{
    return p;
}

template <typename T>
inline T *get_ptr(std::auto_ptr<T> &p)
{
    return p.get();
}
```

One can now write a template (or open-source code where the types are selected by the pre-processor) that will manipulate pointers in a generalised way, by accessing them through `get_ptr()`.

Attribute shims return the type being accessed.

Attribute shims may throw exceptions, though this is not usual.

Control Shims

Control shims define operations that are applied to the instances of the types for which they are defined.

Control shims have a less strict naming convention than the three other fundamental shim concepts, because the operations they perform can be quite diverse. All the names take a verb form, and refer to the action performed, e.g. `make_empty()`, `remove_all()`, etc.

Control shims usually return void, but may return a boolean indication of success, or a count representing the number of items affected or operations carried out.

Control shims may throw exceptions.

Conversion Shims

Conversion shims operate by converting one type to another. For instance, a `to_int()` conversion shim would operate on instances of whatever type, such that it returned an int.

Example:

```
inline int to_int(int i)
{
    return i;
}

inline int to_int(char const *s)
{
    return atoi(s);
}
```

Conversion shims are named according to the `to_xxx()` form, e.g. `to_double()`.

Where the return value is to a complex or pointer type, the shim may be implemented by returning a temporary variable. Conversion shim return values **must**, therefore, be used within their defining expression, rather than assigned to a variable and used at a later stage (which would result in undefined behaviour – a crash in other words).

Consider a conversion shim that returns a pointer to a C-style string from a Win32 window handle, by creating an intermediary smart-pointer proxy instance.

```
class c_str_ptr_HWND_proxy
{
    ...

    operator char const *() const
    {
        ...
    }
};

c_str_ptr_HWND_proxy  c_str_ptr(HWND hwnd);
```

The following code is ok, because the conversion is done and used within the same expression, so the pointer used while the temporary lives.

```
puts(c_str_ptr(hwnd));
```

The following code is not, because the returned value is held in s after the temporary has been destroyed, and will point to something that is now undefined.

```
char const *s = c_str_ptr(hwnd);

puts(s); // Error! s points to who know's what
```

Conversion shims may throw exceptions.

Logical Shims

Logical shims, like Attribute shims, in that they report on the state of an instance to which they are applied. They differ in that they pertain to logical concepts only, e.g. `is_open()`, `is_empty()`.

Example:

```
template <typename C>
inline bool is_empty(std::basic_string<C> const &s)
{
    return s.empty();
}

inline bool is_empty(char const *s)
{
    return *s == '\0';
}
```

Logical shims return a boolean value.

Logical shims do not throw exceptions.

Composite Shim Concepts

Composite shim concepts are those composed of two or more of the fundamental concepts. At this time one composite shim concept is defined, Access Shims

Access Shims

Access shims are notionally similar to attribute shims, but they operate on a wider variety of types. The STLSoft conversion library's [1] `c_str_ptr()` access shim is a good example. For C-style strings, and most string classes (including `std::basic_string<>` and MFC's `CString`) the shim functions follow the attribute shim concept. For more complex string types (e.g. Win32's `LSA_UNICODE_STRING`, which is not null-terminated) and non-string types (e.g. window handles, COM's `VARIANT` type, etc.) the shim functions follow the conversion shim concept. Hence the limitation of conversion shims applies to all access shims.

(An article entitled "Generalised String Manipulation: Access Shims and Type Tunnelling", which goes into greater detail on the concept of access shims, was published in the August 2003 issue of C/C++ User's Journal.)

Access shims do not always follow a strict naming convention. The `c_str_ptr()` shim is a good example: it is named after the standard library's `basic_string`'s `c_str()` method.

Access shims return the type being accessed, or a smart-pointer intermediary that can be converted to that type.

Access shims may throw exceptions.

Notes and References

[1] The STLSoft libraries are available online at http://stlsoft.org.
[2] Shims are covered in Chapter 20 of *Imperfect C++*, Matthew Wilson, Addison-Wesley 2004.