Table 7.4. Parameter passing with `_var` types.

IDL Type	`in`	`inout/out`	Return
`string`	`const String_var &`	`String_var &`	`String_var`
`wstring`	`const WString_var &`	`WString_var &`	`WString_var`
`any`	`const Any_var &`	`Any_var &`	`Any_var`
`objref`	`const` *`objref`*`_var &`	*`objref`*`_var &`	*`objref`*`_var`
`sequence`	`const` *`sequence`*`_var &`	*`sequence`*`_var &`	*`sequence`*`_var`
`struct`	`const` *`struct`*`_var &`	*`struct`*`_var &`	*`struct`*`_var`
`union`	`const` *`union`*`_var &`	*`union`*`_var &`	*`union`*`_var`
`array`	`const` *`array`*`_var &`	*`array`*`_var &`	*`array`*`_var`

Simple types, enumerated types, and fixed-point types are necessarily absent from the table. `_var` types are not generated for these types because `_var` types are not needed (simple types are always fixed-length, caller-allocated, and passed by value).

Note that `_var` types are provided for `in` parameters even though no memory management issues arise here. This is both for consistency and to allow a `_var` type to be passed transparently when an operation expects the underlying type.

Following is an example that illustrates the advantages. The example uses a fixed-length and a variable-length `struct` passed as `out` parameters, and a string as the return value. Here is the IDL:

```
struct Fls {
    long    l_mem;
    double  d_mem;
};

struct Vls {
    double  d_mem;
    string  s_mem;
};

interface Foo {
    string  op(out Fls fstruct, out Vls vstruct);
};
```

If you use the low-level mapping and choose to manage memory yourself, you must write code such as the following:

```
Foo_var fv = ...;       // Get reference

Fls fstruct;            // Note _real_ struct
Vls * vstruct;          // Note _pointer_ to struct
char * ret_val;

ret_val = fv->op(fstruct, vstruct);

delete vstruct;
CORBA::string_free(ret_val);
```

This doesn't look very bad at first glance, but it contains its share of potential problems. You must remember to pass a structure as the first parameter and a *pointer* to a structure as the second parameter, and you also must remember that the variable-length structure and the returned string must be deallocated. Moreover, you must remember to use the correct deallocation function. If your code has any degree of complexity, throws exceptions, and possibly takes early returns out of functions, you can easily make a mistake that leads to a memory leak or, worse, causes memory corruption because you deallocated something twice.

The same code using `_var` types is much simpler:

```
Foo_var fv = ...;           // Get reference

Fls_var fstruct;            // Don't care if fixed or variable
Vls_var vstruct;            // Ditto
CORBA::String_var ret_val;  // To catch return value

ret_val = fv->op(fstruct, vstruct);

// Show some return values
cout << "fstruct.d: " << fstruct->d_mem << endl;
cout << "vstruct.d: " << vstruct->d_mem << endl;
cout << "ret_val:   " << ret_val << endl;

// Deallocation (if needed) is taken care of by _var types
```

The differences in parameter passing rules for the two structures are completely hidden here. To access the structure members, you use the overloaded indirection `->` operator whether the underlying structure is fixed-length or variable-length. When the three `_var` types go out of scope, `vstruct` calls

`delete`, `ret_val` calls `string_free`, and `fstruct` behaves like a stack-allocated structure.

Because `_var` types can also be passed as `in` and `inout` parameters, it is easy to receive a result from one operation and pass that result to another operation. Consider the following IDL:

```
interface Foo {
    string  get();
    void    modify(inout string s);
    void    put(in string s);
};
```

Assume that you are given stringified references to three of these objects and that you want to get a string from the first object, pass it to the second object for modification, and then pass the modified string to the third object. Using `_var` types, this is trivial:

```
{
    Foo_var fv1 = orb->string_to_object(argv[1]);
    Foo_var fv2 = orb->string_to_object(argv[2]);
    Foo_var fv3 = orb->string_to_object(argv[3]);

    // Test fv1, fv2, and fv3 with CORBA::is_nil() here...

    CORBA::String_var s;
    s = fv1->get();         // Get string
    fv2->modify(s);         // Change string
    fv3->put(s);            // Put string
}
// Everything is deallocated here
```

You can also use the explicit directional member functions to pass `_var` parameters, either to get around compiler bugs or to improve the readability of your code:

```
s = fv1->get();         // Get string
fv2->modify(s.inout()); // Change string
fv3->put(s.in());       // Put string
```

This code does the same thing as the previous example but makes it explicit in which direction the parameter is passed.

Note that `_var` types are useful mainly to ensure that `out` parameters and return values are deallocated correctly. There is no point in using a `_var` type purely as an `in` parameter, because this forces two unnecessary calls to the memory allocator. It is far better to instead use a stack-allocated variable. Here is an IDL operation that expects a variable-length `struct` as an `in` parameter: