A caller may pass a reference to a pointer with a null value for **out** parameters, however, since the callee does not examine the value but rather just overwrites it. A callee may not return a null pointer under any of the following circumstances:

- **out** and return variable-length struct
- **out** and return variable-length union
- **out** and return string
- **out** and return sequence
- **out** and return variable-length array, return fixed-length array
- **out** and return any

11 Since OMG IDL has no concept of pointers in general or null pointers in particular, allowing the passage of null pointers to or from an operation would project C++ semantics onto OMG IDL operations.[13] A compliant implementation is allowed but not required to raise a **BAD_PARAM** exception if it detects such an error.

12 Table 24 on page 16-46 displays the mapping for the basic OMG IDL parameter passing modes and return type according to the type being passed or returned, while Table 25 on page 16-47 displays the same information for **T_var** types. Table 25 is merely for informational purposes; it is expected that operation signatures will be written in terms of the parameter passing modes shown in Table 24, and that **T_var** types will support the necessary conversion operators to allow them to be passed directly.

13 In Table 24, fixed-length arrays are the only case where the type of an **out** parameter differs from a return value, which is necessary because C++ does not allow a function to return an array. The mapping returns a pointer to a *slice* of the array, where a slice is an array with all the dimensions of the original specified except the first one.

TABLE 24. Basic Argument and Result Passing

Data Type	In	Inout	Out	Return
short	Short	Short&	Short&	Short
long	Long	Long&	Long&	Long
unsigned short	UShort	UShort&	UShort&	UShort
unsigned long	ULong	ULong&	ULong&	ULong
float	Float	Float&	Float&	Float
double	Double	Double&	Double&	Double
boolean	Boolean	Boolean&	Boolean&	Boolean
char	Char	Char&	Char&	Char
octet	Octet	Octet&	Octet&	Octet
enum	enum	enum&	enum&	enum
object reference ptr[1]	objref_ptr	objref_ptr&	objref_ptr&	objref_ptr
struct, fixed	const struct&	struct&	struct&	struct

13.When real C++ exceptions are not available, however, it is important that null pointers are returned whenever an **Environment** containing an exception is returned; see “Without Exception Handling” on page D-1 for more details.

TABLE 24. Basic Argument and Result Passing *(Continued)*

Data Type	In	Inout	Out	Return
struct, variable	const struct&	struct&	struct*&	struct*
union, fixed	const union&	union&	union&	union
union, variable	const union&	union&	union*&	union*
string	const char*	char*&	char*&	char*
sequence	const sequence&	sequence&	sequence*&	sequence*
array, fixed	const array	array	array	array slice*[2]
array, variable	const array	array	array slice*&[2]	array slice*[2]
any	const any&	any&	any*&	any*

1. Including pseudo-object references.

2. A slice is an array with all the dimensions of the original except the first one.

14 A caller is responsible for providing storage for all arguments passed as **in** arguments.

TABLE 25. T_var Argument and Result Passing

Data Type	In	Inout	Out	Return
object reference var[1]	const objref_var&	objref_var&	objref_var&	objref_var
struct_var	const struct_var&	struct_var&	struct_var&	struct_var
union_var	const union_var&	union_var&	union_var&	union_var
string_var	const string_var&	string_var&	string_var&	string_var
sequence_var	const sequence_var&	sequence_var&	sequence_var&	sequence_var
array_var	const array_var&	array_var&	array_var&	array_var
any_var	const any_var&	any_var&	any_var&	any_var

1. Including pseudo-object references.

15 Table 26 on page 16-45 and Table 27 on page 16-48 describe the caller's responsibility for storage associated with **inout** and **out** parameters and for return results.

TABLE 26. Caller Argument Storage Responsibilities

Type	Inout Param	Out Param	Return Result
short	1	1	1
long	1	1	1
unsigned short	1	1	1
unsigned long	1	1	1
float	1	1	1
double	1	1	1
boolean	1	1	1
char	1	1	1
octet	1	1	1
enum	1	1	1

TABLE 26. Caller Argument Storage Responsibilities ***(Continued)***

Type	Inout Param	Out Param	Return Result
object reference ptr	2	2	2
struct, fixed	1	1	1
struct, variable	1	3	3
union, fixed	1	1	1
union, variable	1	3	3
string	4	3	3
sequence	5	3	3
array, fixed	1	1	6
array, variable	1	6	6
any	5	3	3

TABLE 27. Argument Passing Cases

Case[1]	
1	Caller allocates all necessary storage, except that which may be encapsulated and managed within the parameter itself. For inout parameters, the caller provides the initial value, and the callee may change that value. For out parameters, the caller allocates the storage but need not initialize it, and the callee sets the value. Function returns are by value.
2	Caller allocates storage for the object reference. For inout parameters, the caller provides an initial value; if the callee wants to reassign the inout parameter, it will first call CORBA::release on the original input value. To continue to use an object reference passed in as an inout, the caller must first duplicate the reference. The caller is responsible for the release of all out and return object references. Release of all object references embedded in other structures is performed automatically by the structures themselves.
3	For out parameters, the caller allocates a pointer and passes it by reference to the callee. The callee sets the pointer to point to a valid instance of the parameter's type. For returns, the callee returns a similar pointer. The callee is not allowed to return a null pointer in either case. In both cases, the caller is responsible for releasing the returned storage. To maintain local/remote transparency, the caller must always release the returned storage, regardless of whether the callee is located in the same address space as the caller or is located in a different address space. Following the completion of a request, the caller is not allowed to modify any values in the returned storage—to do so, the caller must first copy the returned instance into a new instance, then modify the new instance.
4	For inout strings, the caller provides storage for both the input string and the char* pointing to it. Since the callee may deallocate the input string and reassign the char* to point to new storage to hold the output value, the caller should allocate the input string using string_alloc(). The size of the out string is therefore not limited by the size of the in string. The caller is responsible for deleting the storage for the out using string_free(). The callee is not allowed to return a null pointer for an inout, out, or return value.

TABLE 27. Argument Passing Cases ***(Continued)***

Case[1]	
5	For inout sequences and anys, assignment or modification of the sequence or any may cause deallocation of owned storage before any reallocation occurs, depending upon the state of the Boolean release parameter with which the sequence or any was constructed.
6	For out parameters, the caller allocates a pointer to an array slice, which has all the same dimensions of the original array except the first, and passes the pointer by reference to the callee. The callee sets the pointer to point to a valid instance of the array. For returns, the callee returns a similar pointer. The callee is not allowed to return a null pointer in either case. In both cases, the caller is responsible for releasing the returned storage. To maintain local/remote transparency, the caller must always release the returned storage, regardless of whether the callee is located in the same address space as the caller or is located in a different address space. Following completion of a request, the caller is not allowed to modify any values in the returned storage—to do so, the caller must first copy the returned array instance into a new array instance, then modify the new instance.

1. As listed in Table 26 on page 16-45