VISChainClientInterceptorFactory methods .................................. 10-6
VISChainServerInterceptorFactory ............................................. 10-7
Include file ................................................................. 10-7
VISChainServerInterceptorFactory destructor ................................. 10-7
VISChainServerInterceptorFactory methods .................................. 10-7
VISClientInterceptor .......................................................... 10-7
Include file ................................................................. 10-8
VISClientInterceptor constructors/ destructors ................................ 10-8
VISClientInterceptor methods ................................................ 10-8
VISClientInterceptorFactory .................................................. 10-11
Include file ................................................................. 10-11
VISClientInterceptorFactory constructor .................................... 10-11
VISClientInterceptorFactory methods ........................................ 10-11
VISClosure ................................................................. 10-12
Include file ................................................................. 10-12
VISClosure members .......................................................... 10-12
VISClosureData .............................................................. 10-12
VISClosureData methods ...................................................... 10-12
VISObjectWrapper::ChainUntypedObject-WrapperFactory ................. 10-13
Include file ................................................................. 10-13
ChainUntypedObjectWrapperFactory methods ................................ 10-13
VISObjectWrapper::UntypedObject-Wrapper .................................. 10-15
Include file ................................................................. 10-15
UntypedObjectWrapper methods .............................................. 10-15
VISObjectWrapper::UntypedObject-WrapperFactory ...................... 10-16
Include file ................................................................. 10-16
UntypedObjectWrapperFactory constructors ................................ 10-16
UntypedObjectWrapperFactory methods .................................... 10-17
VISServerInterceptor ........................................................ 10-17
Include file ................................................................. 10-17
VISServerInterceptor enumerations ......................................... 10-17
VISServerInterceptor constructors/ destructors .......................... 10-17
VISServerInterceptor methods ................................................ 10-18
VISServerInterceptorFactory .................................................. 10-22
Include file ................................................................. 10-22
VISServerInterceptorFactory constructor .................................... 10-22
VISServerInterceptorFactory methods ........................................ 10-23

Chapter 11
GIOP, IOP, and IIOP interfaces ........................................... 11-1
GIOP::MessageHeader ...................................................... 11-1
MessageHeader members ................................................... 11-1
GIOP::CancelRequestHeader ................................................ 11-2
CancelRequestHeader members ............................................. 11-2
GIOP::LocateReplyHeader .................................................. 11-3
LocateReplyHeader members ............................................... 11-3
GIOP::LocateRequestHeader ................................................ 11-3
LocateRequestHeader members ............................................. 11-3
GIOP::ReplyHeader ........................................................ 11-3
Include file ................................................................. 11-4
ReplyHeader members ...................................................... 11-4
GIOP::RequestHeader ........................................................ 11-4
Include file ................................................................. 11-4
RequestHeader members ................................................... 11-4
IIOP::ProfileBody .......................................................... 11-5
ProfileBody members ...................................................... 11-5
IOP::IOR ................................................................. 11-6
Include file ................................................................. 11-6
IOR members ............................................................... 11-6
IOP::TaggedProfile .......................................................... 11-6
TaggedProfile members ..................................................... 11-7

Chapter 12
Marshal buffer classes ....................................................... 12-1
CORBA::MarshalInBuffer .................................................... 12-1
Include file ................................................................. 12-2
CORBA::MarshalInBuffer constructors/ destructors ..................... 12-2
CORBA::MarshalInBuffer methods ......................................... 12-2
CORBA::MarshalInBuffer operators ....................................... 12-5
CORBA::MarshalOutBuffer .................................................. 12-5
Include file ................................................................. 12-6
CORBA::MarshalOutBuffer constructors/ destructors .................... 12-6
CORBA::MarshalOutBuffer methods ....................................... 12-6
CORBA::MarshalOutBuffer operators ..................................... 12-8
Tables

3.1 IDL primitive type mappings ............ 3-1
3.2 Summary of C++ mappings for complex data types ............ 3-7
3.3 Methods generated for the un_ex class ............ 3-9
3.4 Synopsis of the methods generated for the unbounded sequence in Code sample 3.28 ............ 3-12
15.1 Common attributes for TSingle, TPool, and TSession adapters ............ 15-2
15.2 Attributes for TPool adapters ............ 15-2
15.3 Attributes for TPool and TSession adapters ............ 15-3
A.1 BOA_init() options used by object implementations ............ A-2
A.2 ORB_init() options used by client programs ............ A-4
A.3 Location Service command-line options ............ A-6
Preface

The VisiBroker for C++ Reference provides concise descriptions of the classes, interfaces and methods available to programmers developing applications with VisiBroker for C++.

This Preface describes the manual’s contents, the typographic and syntax conventions used throughout the manual, and provides references for more information about VisiBroker and CORBA.

Organization of this manual

This manual includes the following sections:

- Chapter 2, “Programmer tools,” provides information about the programming tools used to compile C++ stubs and to populate the Interface Repository.
- Chapter 3, “IDL to C++ language mapping,” details the C++ to CORBA mapping specifications, including data types, strings, constant, type definitions, enumerations, and modules.
- Chapter 4, “Generated classes,” describes the classes generated by VisiBroker’s IDL compiler.
- Chapter 5, “Core interfaces and classes,” describes the VisiBroker for C++ core interfaces and classes.
- Chapter 6, “Dynamic interfaces and classes,” describes the Dynamic Invocation Interface used by clients, and the Dynamic Skeleton Interface used by object servers.
- Chapter 7, “Interface Repository classes and interfaces,” describes the classes and interfaces used to access the Interface Repository.
• Chapter 8, “Activation interfaces and classes,” describes the interfaces and
classes used to activate object implementations.
• Chapter 9, “Event Handler interfaces and classes,” describes the event
handler interfaces and classes.
• Chapter 10, “Interceptor and object wrapper interfaces and classes,”
describes the interfaces you use to create interceptors for client or server-side
message processing.
• Chapter 11, “GIOP, IOP, and IIOP interfaces,” describes the CORBA-defined
header and message formats.
• Chapter 12, “Marshal buffer classes,” describes the classes and methods for
creating and processing message buffers.
• Chapter 13, “Location service — interfaces and classes,” describes how to use
the location service to discover objects implemented on your network.
• Chapter 14, “Initialization interfaces and classes,” describes the interfaces
and methods for initializing interceptors and other services.
• Chapter 15, “ORB manager interfaces,” describes how client applications can
monitor and control server applications.
• Appendix A, “Using command-line options,” explains the ORB, BOA, and
location service options that can be passed as command-line arguments
when your application is started.

Typographic conventions

This manual uses the following conventions:

<table>
<thead>
<tr>
<th>Convention</th>
<th>Used for</th>
</tr>
</thead>
</table>
| boldface   | Bold type indicates that syntax should be typed exactly as shown. For
           | UNIX, used to indicate database names, filenames, and similar terms. |
| italics    | Italics indicates information that the user or application provides, such as
           | variables in syntax diagrams. Also used to introduce new terms. |
| computer   | Computer typeface is used for sample command lines and code. |
| UPPERCASE  | Uppercase letters indicate SQL statements and terms. For Windows, used |
           | to indicate database names, filenames, and similar terms. |
| []         | Brackets indicate optional items. |
| ...        | An ellipsis indicates that the previous argument can be repeated. |
| |         | A vertical bar separates two mutually exclusive choices. |
| ::         | A column of three dots indicates the continuation of previous lines of |
           | code. |
Platform conventions

This manual uses the following symbols to indicate that information is platform-specific:

- W All Windows platforms including Windows 3.1, Windows NT, and Windows 95
- NT Windows NT only
- 95 Windows 95 only
- U All UNIX platforms

Where to find additional information

For more information about VisiBroker for C++, refer to the following information sources:

- *VisiBroker for C++ Programmer’s Guide* provides information on developing distributed object-based applications in C++ for Windows and UNIX platforms.
- *VisiBroker for C++ Installation and Administration Guide* contains the instructions for installing VisiBroker for C++ on Windows and UNIX.
- *VisiBroker for C++ Release Notes* contain late-breaking information about the current release of VisiBroker for C++.

For more information about the CORBA specification, refer to the following sources:

- *IDL to C++ Language Mapping - 94-9-14.* This document is available from the Object Management Group and describes the Interface Definition Language mappings for C++.

Contacting Inprise Technical Support

Inprise offers a variety of support options to help you get the most from your Inprise products. For information about these options, see the “Services” section of Inprise’s web site at http://www.inprise.com, or contact our Sales Department at 1-800-632-2864.

When contacting VisiBroker Technical Support, be prepared to provide complete information about your environment, the version of the VisiBroker product you are using, and a detailed description of the problem.
Chapter 2

Programmer tools

This chapter describes the programmer tools offered by VisiBroker for C++. This chapter includes the following sections:

General information  page 2-1
idl2cpp  page 2-1
idl2ir  page 2-4

General information

The VisiBroker programming tools described in this chapter differ, depending on whether you have a Unix or a Windows environment. The Unix version of each tool is listed first followed by the Windows version.

**U** For UNIX users, to view options for a command, enter

**Syntax** command name -?  

**Example** idl2cpp -?

**W** For Windows users, to view options for a command, enter

**Syntax** command name -?

**Example** idl2cpp -?

idl2cpp

This command implements VisiBroker’s IDL to C++ compiler, which generates client stubs and server skeleton code from an IDL file.
**Syntax**

`idl2cpp [arguments] infile`

`idl2cpp` takes an IDL file as input and generates the corresponding C++ classes for the client and server side, client stubs, and server skeleton code.

The `infile` parameter represents the IDL file for which you wish C++ code to be generated and the arguments provide various controls over the resulting code.

**Example**

`idl2cpp -hdr_suffix hx -server_ext _serv -no_tie -no_excep-spec bank.idl`

When linking implementations based on the stubs and skeletons `idl2cpp` generates, use the `-DISTRICT` preprocessor option. Otherwise, the linker may display an error message suggesting that a constructor is missing from `orb.lib`.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-&lt;arg&gt;</td>
<td>Any other preprocessor argument, such as:</td>
</tr>
<tr>
<td></td>
<td><code>-Dx=y</code></td>
</tr>
<tr>
<td></td>
<td><code>-I../IDL</code></td>
</tr>
<tr>
<td>-client_ext &lt;file_extension&gt;</td>
<td>Specifies the file extension to be used for client files that are generated. The default extension is “.c”. To generate client files without an extension, specify <code>none</code> as the value for <code>&lt;file_extension&gt;</code>.</td>
</tr>
<tr>
<td></td>
<td><strong>Note:</strong> It is not useful to specify <code>none</code> as the <code>&lt;file_extension&gt;</code> for both <code>-client_ext</code> and <code>-server_ext</code>, as this will result in identical filenames for client and server files. The idl2cpp compiler will issue a warning, and then overwrite the client files with the server files.</td>
</tr>
<tr>
<td>-corba_inc &lt;filename&gt;</td>
<td>Causes the <code>#include &lt;filename&gt;</code> directive to be inserted in generated code instead of the usual <code>#include &lt;corba.h&gt;</code> directive. By default, <code>#include &lt;corba.h&gt;</code> is inserted into generated code.</td>
</tr>
<tr>
<td>-excep_spec</td>
<td>Generates exception specifications for methods. By default, exception specifications are not generated.</td>
</tr>
<tr>
<td>-export &lt;tag&gt;</td>
<td>Defines a tag name to be inserted into every client-side declaration (class, function, etc.) that is generated. Specifying <code>-export _MY_TAG</code> when invoking <code>idl2cpp</code> will result in a class definition like this:</td>
</tr>
<tr>
<td></td>
<td><code>class _MY_TAG Bank {...}</code></td>
</tr>
<tr>
<td></td>
<td>instead of</td>
</tr>
<tr>
<td></td>
<td><code>class Bank {...}</code></td>
</tr>
</tbody>
</table>

By default, no tag names for client-side declarations are generated.
Argument | Description
--- | ---
`--export_skel <tag>` | Defines a tag name to be inserted into just the server-side declarations that are generated. Specifying `--export_MY_TAG` when invoking idl2cpp will result in a class definition like this:
```
class _MY_TAG _sk_Bank{...}
```
instead of
```
class _sk_Bank {...}
```
By default, no tag names for server-side declarations are generated.

`-hdr_dir <path>` | Specifies the directory where the generated include files (_c.hh and _s.hh) are to be written. By default, the code is written to the current directory.

`-hdr_suffix <suffix>` | Specifies the file suffix to be used for include files that are generated. The default extension is “hh”.

`-incl_files_code` | Enables the generation of code for all IDL that is referenced in the IDL file being compiled by a #include statement. By default, files referenced by a #include statement are parsed, but no code is generated for the definitions they contain.

`-impl_base_object <object_name>` | Causes the classes in all generated code to be inherited from `object_name` instead of `CORBA::Object`. By default, all classes in generated code are inherited from `CORBA::Object`.

`-map_keyword <keywrd> <map>` | Adds `<keywrd>` as a keyword and associates with it the mapping indicated. Any IDL identifier that conflicts with `<keywrd>` will be mapped in C++ to `<map>`. This prevents clashes between keywords and names used in C++ code. All C++ keywords have default mappings—they do not need to be specified using this option.

`-no_exceptions` | Suppresses the generation of code that throws exceptions. By default, exceptions are generated.

`-no_tie` | Suppresses the generation of `_tie` template classes. By default, `_tie` classes are generated.

`-no_stdstream` | Suppresses the generation of class stream operators with standard iostream classes in their signature.

`-obj_wrapper` | Generates stubs and skeletons with object wrapper support. It also generates the base typed object wrapper from which all other object wrappers inherit, and a default object wrapper that performs the untyped object wrapper calls. When this option is not set, idl2cpp does not generate code for object wrappers.

`-ptie` | Enables the generation of `_ptie` template classes. By default, `_ptie` classes are not generated.

`-root_dir <path>` | Specifies the directory where the generated code is to be written; the same as setting `-hdr_dir` and `-src_dir` to `<path>`. By default, the code is written to the current directory.
This command allows you to populate an interface repository with objects defined in an Interface Definition Language source file.

**Syntax**

```
idl2ir [-ir <IR_name>] [-replace] {filename.idl}
```

**Example**

```
idl2ir -ir my_repository -replace bank/Bank.idl
```

**Description**

The `idl2ir` command takes an IDL file as input, binds itself to an interface repository server, and populates the repository with the IDL constructs contained in `infile`. If the repository already contains an item with the same name as an item in the IDL file, the old item will be replaced.
Note

The `idl2ir` command does not handle anonymous arrays or sequences properly. To work around this problem, `typedefs` must be used for all sequences and arrays.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-ir name</td>
<td>Specifies the instance name of the interface repository to which <code>idl2ir</code> will attempt to bind. If no name is specified, <code>idl2ir</code> will bind itself to the interface repository server found in the current domain. The current domain is defined by the <code>OSAGENT_PORT</code> environment variable.</td>
</tr>
<tr>
<td>-replace</td>
<td>Replaces definitions instead of updating them.</td>
</tr>
<tr>
<td>filename.idl</td>
<td>Specifies the IDL file to be used as input.</td>
</tr>
</tbody>
</table>
Chapter 3

IDL to C++ language mapping

This chapter discusses the IDL to C++ language mapping provided by the VisiBroker idl2cpp compiler, which strictly complies with the CORBA C++ language mapping specification. This chapter includes the following major sections:

- Primitive data types page 3-1
- Strings page 3-2
- Constants page 3-3
- Enumerations page 3-5
- Type definitions page 3-5
- Modules page 3-6
- Complex data types page 3-6

Primitive data types

The basic data types provided by the Interface Definition Language are summarized in Table 3.1. Due to hardware differences between platforms, some of the IDL primitive data types have a definition that is marked “platform dependent.” On a platform that has 64-bit integral representations, for example the g type, would still be only 32 bits. You should refer to the include file orbtypes.h for an exact mapping of these primitive data types for your particular platform.

Table 3.1 IDL primitive type mappings

<table>
<thead>
<tr>
<th>IDL type</th>
<th>VisiBroker type</th>
<th>C++ definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>short</td>
<td>CORBA::Short</td>
<td>short</td>
</tr>
<tr>
<td>long</td>
<td>CORBA::Long</td>
<td>platform dependent</td>
</tr>
</tbody>
</table>
Caution

The IDL boolean type is defined by the CORBA specification to have only one of two values: 1 or 0. Using other values for a boolean will result in undefined behavior.

Strings

String types in IDL may specify a length or may be unbounded, but both are mapped to the C++ type `char *`. You must use the functions shown in Code sample 3.1 for dynamically allocating strings to ensure that your applications and VisiBroker use the same memory management facilities. All CORBA string types are null-terminated.

Code sample 3.1

Methods for allocating and freeing memory for strings

```cpp
class CORBA
{
  ...
  static char *string_alloc(CORBA::ULong len);
  static void string_free(char *data);
  ...
};
```

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CORBA::string_alloc</td>
<td>Dynamically allocates a string and returns a pointer to the string. A NULL pointer is returned if the allocation fails. The length specified by the len parameter does not need to include the NULL terminator.</td>
</tr>
<tr>
<td>CORBA::string_free</td>
<td>Releases the memory associated with a string that was allocated with CORBA::string_alloc.</td>
</tr>
</tbody>
</table>

String_var Class

In addition to mapping an IDL `string` to a `char *`, the IDL compiler generates a `String_var` class that contains a pointer to the memory allocated to hold the string. When a `String_var` object is destroyed or goes out of scope, the memory allocated to the string is automatically freed. Code sample 3.2 shows the
String_var class and the methods it supports. For more information on the _var classes, see page 4-3.

**Code sample 3.2  String_var class**

```cpp
class CORBA {
    class String_var {
        protected:
            char*_p;
        ...
        public:
            String_var();
            String_var(char *p);
            ~String_var();
            String_var& operator=(const char *p);
            String_var& operator=(char *p);
            String_var& operator=(const String_var &s);
            operator const char *() const;
            operator char *();
            char &operator[](CORBA::ULong index);
            char operator[](CORBA::ULong index) const;
            friend ostream& operator<<(ostream&, const String_var&);
            inline friend Boolean operator==(const String_var &s1,
                const String_var &s2);
            ...
    };
    ...
};
```

## Constants

Code sample 3.3 shows how IDL constants defined outside of any interface specification will be mapped directly to a C++ constant declaration.

Code sample 3.4 shows how constants defined within an interface specification are declared in the include file and assigned a value in the source file.

**Code sample 3.3  Top-level definitions in IDL**

```idl
const string str_example = "this is an example";
const long long_example = 100;
const boolean bool_example = TRUE;
```

**Code sample 3.4  Resulting C++ code for constants**

```cpp
const char * str_example = "this is an example";
const CORBA::Long long_example = 100;
const CORBA::Boolean bool_example = 1;
```

**Code sample 3.5  IDL definitions from the example.idl file**

```idl
interface example {
    const string str_example = "this is an example";
    const long long_example = 100;
    const boolean bool_example = TRUE;
};
```
Special cases involving constants

**Code sample 3.6**  C++ code generated to the example_client.hh file

```cpp
class example : public virtual CORBA::Object
{
...
  static const char *str_example; /* this is an example */
  static const CORBA::Long long_example; /* 100 */
  static const CORBA::Boolean bool_example; /* 1 */
...
};
```

**Code sample 3.7**  C++ code generated to the example_client.cc file

```cpp
const char *example::str_example = "this is an example";
const CORBA::Long example::long_example = 100;
const CORBA::Boolean example::bool_example = 1;
```

Special cases involving constants

Under some circumstances, the IDL compiler must generate C++ code containing the value of an IDL constant rather than the name of the constant. Code sample 3.8 shows how the value of the constant `len` must be generated for the typedef `V` to allow the C++ code to compile properly.

**Code sample 3.8**  Definition of an IDL constant with a value

```idl
// IDL
interface foo {
    const long length = 10;
typedef long V[length];
};
```

**Code sample 3.9**  Generation of an IDL constant's value in C++

```cpp
class foo : public virtual CORBA::Object
{
  const CORBA::Long length;
typedef CORBA::Long V[10];
};
```

Enumerations

Code sample 3.11 shows how enumerations in IDL map directly to C++ enumerations.

**Code sample 3.10**  IDL definition of an enumeration

```idl
// IDL
enum enum_type {
    first,
    second,
    third
};
```
Code sample 3.11  Enumerations in IDL map directly to C++

```cpp
// C++ code
class enum_type {
    first,
    second,
    third
};
```

### Type definitions

Code sample 3.13 shows how type definitions in IDL are mapped directly to C++ type definitions. If the original IDL type definition maps to several C++ types, the IDL compiler generates the corresponding aliases for each type in C++. Code sample 3.15 and Code sample 3.17 show other type definition mapping examples.

**Code sample 3.12  Simple type definitions in IDL**

```idl
interface A1;
typedef A1 A2;
```

**Code sample 3.13  Mapping of simple type definitions from IDL to C++**

```cpp
// C++
typedef octet example_octet;
typedef enum enum_values {
    first,
    second,
    third
} enum_example;
```

**Code sample 3.14  IDL typedef of an interface**

```cpp
class A1;
typedef A1 *A1_ptr;
typedef A1_ptr A1Ref;
typedef A1 A1_var;
typedef A1 A2;
typedef A1_ptr A2_ptr;
typedef A1Ref A2Ref;
typedef A1_var A2_var;
```
Code sample 3.16 IDL typedef of a sequence

```idl
// IDL
typedef sequence<long> S1;
typedef S1 S2;
```

Code sample 3.17 Mapping the IDL sequence type definition to C++

```cpp
// C++
class S1;
typedef S1 *S1_ptr;
typedef S1_ptr S1Ref;
class S1_var;

typedef S1 S2;
typedef S1_ptr S2_ptr;
typedef S1Ref S2Ref;
typedef S1_var S2_var;
```

Modules

The OMG IDL to C++ language mapping specifies that an IDL module should be mapped to a C++ namespace with the same name. Since few compilers currently support the namespace, the C++ language mapping allows the use of class in its place. Code sample 3.19 shows how VisiBroker’s IDL compiler maps module to class.

Code sample 3.18 IDL module definition

```idl
// IDL
module ABC
{
    ...
};
```

Code sample 3.19 Mapping an IDL module to a C++ class

```cpp
// C++
class ABC
{
    ...
};
```

Complex data types

The C++ mappings for IDL structures, unions, sequences, and arrays depend on whether or not the data members they contain are of a fixed or variable length. These types are considered to have variable lengths. As a result, any complex data type that contains a structure, union, sequence, or array will also have a variable length. Complex data types include:

- The Any type.
- The string type, bounded or unbounded.
- The sequence type, bounded or unbounded.
• An object reference.
• Other structures or unions that contain a variable-length member.
• An array with variable-length elements.
• A typedef with variable-length elements.

Table 3.2 Summary of C++ mappings for complex data types

<table>
<thead>
<tr>
<th>IDL type</th>
<th>C++ mapping</th>
</tr>
</thead>
<tbody>
<tr>
<td>struct (fixed length)</td>
<td>struct and _var class</td>
</tr>
<tr>
<td>struct (variable length)</td>
<td>struct and _var class</td>
</tr>
<tr>
<td></td>
<td>(variable length members are declared with their respective T_var class)</td>
</tr>
<tr>
<td>union</td>
<td>class and _var class</td>
</tr>
<tr>
<td>sequence</td>
<td>class and _var class</td>
</tr>
<tr>
<td>array</td>
<td>array, array_slice, array_forany, and array_var</td>
</tr>
</tbody>
</table>

Fixed-length structures

Code sample 3.21 shows how fixed-length structures in IDL are mapped to C++ code. In addition to the structure, VisiBroker’s IDL compiler will also generate an example_var class for the structure. For more information on the _var class, see page 4-3.

Code sample 3.20 Fixed-length structure definition in IDL

// IDL
struct example {
    short a;
    long b;
};

Code sample 3.21 Mapping a fixed-length IDL structure to C++

// C++
struct example {
    CORBA::Short a;
    CORBA::Long b;
};

class example_var
{}
    ...;
    private:
        example *_ptr;
};

Using fixed length structures

Code sample 3.22 shows that to access the fields of the _var class ex2, the -> operator must always be used. When ex2 goes out of scope, the memory allocated to it will be freed automatically.
Variable length structures

Code sample 3.22 Use of the example structure and the example_var class

// Declare an example struct and initialize its fields.
example ex1 = { 2, 5 };

// Declare a _var class and assign it to a newly created example structure.
// The _var points to an allocated struct with un-initialized fields.
example_var ex2 = new example;

// Initialize the fields of ex2 from ex1
ex2->a = ex1.b;

Variable length structures

Code sample 3.24 shows how you could modify the example structure, replacing the long member with a string and adding an object reference, to change to a variable-length structure.

Code sample 3.23 Variable length structure definitions in IDL

// IDL
interface ABC {
    ...
};
struct vexample {
    short a;
    ABC c;
    string name;
};

Code sample 3.24 Mapping a variable-length structure to C++

// C++
struct vexample {
    CORBA::Short a;
    ABC_var c;
    CORBA::String_var name;
    vexample& operator=(const vexample& s);
};

class vexample_var {
    ...;
};

Notice how the ABC object reference is mapped to an ABC_var class. In a similar fashion, the string name is mapped to a CORBA::String_var class. In addition, an assignment operator is also generated for variable-length structures.

Memory management for structures

The use of _var classes in variable-length structures ensures that memory allocated to the variable-length members are managed transparently.
• If a structure goes out of scope, all memory associated with variable-length members is automatically freed.

• If a structure is initialized or assigned and then re-initialized or re-assigned, the memory associated with the original data is always freed.

• When a variable-length member is assigned to an object reference, a copy is always made of the object reference. If a variable-length member is assigned to a pointer, no copying takes place.

## Unions

Code sample 3.26 shows how an IDL union is mapped to a C++ class with methods for setting and retrieving the value of the data members. A data member, named `_d`, of the discriminant type is also defined. The value of this discriminant is not set when the union is first created, so an application must set it before using the union. Setting any data member using one of the provided methods automatically sets the discriminant. Table 3.3 describes some of the methods in the `un_ex` class.

### Table 3.3  Methods generated for the `un_ex` class

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>un_ex()</code></td>
<td>The default constructor sets the discriminant to zero but does not initialize any of the other data members.</td>
</tr>
<tr>
<td><code>un_ex(const un_ex&amp; obj)</code></td>
<td>The copy constructor performs a deep copy of the source object.</td>
</tr>
<tr>
<td><code>~un_ex()</code></td>
<td>The destructor frees all memory owned by the union.</td>
</tr>
<tr>
<td><code>operator=(const un_ex&amp; obj)</code></td>
<td>The assignment operator performs a deep copy, releasing old storage, if necessary.</td>
</tr>
</tbody>
</table>

VisiBroker’s IDL compiler may also generate hash and compare methods for unions, which you can generate using compiler options. See page 4-1 for more information on these compiler options.

### Code sample 3.25  IDL union containing a struct

```idl
// IDL
struct st_ex
{
    long abc;
};
union un_ex switch(long)
{
    case 1: long x; // a primitive data type
    case 2: string y; // a simple data type
    case 3: st_ex z; // a complex data type
};
```
Unions

Code sample 3.26  Mapping an IDL union to a C++ class

    // C++
    struct st_ex
    {
        CORBA::Long abc;
    };

class un_ex
{
    private:
        CORBA::Long _disc;
        CORBA::Long _x;
        CORBA::String_var _y;
        st_ex _z;
    public:
        un_ex();
        ~un_ex();
        un_ex(const un_ex& obj);
        un_ex& operator=(const un_ex& obj);
        void x(CORBA::Long val);
        CORBA::Long x() const;
        void y(char *val);
        void y(const char *val);
        void y(const CORBA::String_var& val);
        const char *y() const;
        const st_ex& z() const;
        st_ex& z();
        CORBA::Long _d();
        void _d(CORBA::Long);
    ...}

Managed types for unions
In addition to the un_ex class shown in Code sample 3.26, an un_ex_var class would also be generated. See page 4-3 for details on the _var classes.

Memory management for unions
Here are some important points to remember about memory management of complex data types within a union:

- When you use an accessor method to set the value of a data member, a deep copy is performed. You should pass parameters to accessor methods by value for smaller types, or by a constant reference for larger types.

- When you set a data member using an accessor method, any memory previously associated with that member is freed. If the member being assigned is an object reference, the reference count of that object will be incremented before the accessor method returns.

- A char * accessor method will free any storage before ownership of the passed pointer is assumed.
Sequences

IDL sequences, both bounded and unbounded, are mapped to a C++ class that has a current length and a maximum length. The maximum length of a bounded sequence is defined by the sequence’s type. Unbounded sequences can specify their maximum length when their C++ constructor is called. The current length can be modified programmatically. Code sample 3.28 shows how an IDL sequence is mapped to a C++ class with accessor methods.

Note When the length of an unbounded sequence exceeds the maximum length you specify, VisiBroker will transparently allocate a larger buffer, copy the old buffer to the new buffer, and free the memory allocated to the old buffer. No attempt will be made, however, to free any unused memory if the maximum length decreases.

Code sample 3.27 IDL unbounded sequence

```idl
typedef sequence<long> LongSeq;
```

Code sample 3.28 Mapping an IDL unbounded sequence to a C++ class

```cpp
// C++
class LongSeq
{
  public:
    LongSeq(CORBA::ULong max=0);
    LongSeq(CORBA::ULong max=0, CORBA::ULong length,
             CORBA::Long *data, CORBA::Boolean release = 0);
    LongSeq(const LongSeq&);
    ~LongSeq();
    LongSeq& operator=(const LongSeq&);
    CORBA::ULong maximum() const;
    void length(CORBA::ULong len);
    CORBA::ULong length() const;
    const CORBA::ULong& operator[](CORBA::ULong index) const;
    ...
    static LongSeq *_duplicate(LongSeq* ptr);
    static void _release(LongSeq *ptr);
    static CORBA::Long *allocbuf(CORBA::ULong nelems);
    static void freebuf(CORBA::Long *data);
  private:
    CORBA::Long * _contents;
    CORBA::ULong _count;
    CORBA::ULong _num_allocated;
    CORBA::Boolean _release_flag;
    CORBA::Long _ref_count;
};
```
**Sequences**

### Table 3.4  Synopsis of the methods generated for the unbounded sequence in Code sample 3.28

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LongSeq(CORBA::ULong max=0)</td>
<td>The constructor for an unbounded sequence takes a maximum length as an argument. Bounded sequences have a defined maximum length.</td>
</tr>
<tr>
<td>LongSeq(CORBA::ULong max=0, CORBA::ULong length, CORBA::Long *data, CORBA::Boolean release=0)</td>
<td>This constructor allows you to set the maximum length, the current length, a pointer to the data buffer associated and a release flag. If release is not zero, VisiBroker will free memory associated with the data buffer when increasing the size of the sequence. If release is zero, the old data buffer’s memory is not freed. Bounded sequences have all of these parameters except for max.</td>
</tr>
<tr>
<td>LongSeq(const LongSeq&amp;)</td>
<td>The copy constructor performs a deep copy of the source object.</td>
</tr>
<tr>
<td>~LongSeq();</td>
<td>The destructor frees all memory owned by the sequence only if the release flag had a non-zero value when constructed.</td>
</tr>
<tr>
<td>operator=(const LongSeq&amp;)</td>
<td>The assignment operator performs a deep copy, releasing old storage, if necessary.</td>
</tr>
<tr>
<td>maximum()</td>
<td>Returns the size of the sequence.</td>
</tr>
<tr>
<td>length()</td>
<td>Two methods are defined for setting and returning the length of the sequence.</td>
</tr>
<tr>
<td>operator<a href=""></a></td>
<td>Two indexing operators are provided for accessing an element within a sequence. One operator allows the element to be modified and one allows only read access to the element.</td>
</tr>
<tr>
<td>_release()</td>
<td>Releases the sequence. If the constructor’s release flag was non-zero when the object was created and the sequence element type is a string or object reference, each element will be released before the buffer is released.</td>
</tr>
<tr>
<td>allocbuf()</td>
<td>You should use these two static methods to allocate or free any memory used by a sequence.</td>
</tr>
<tr>
<td>freebuf()</td>
<td></td>
</tr>
</tbody>
</table>

### Managed types for sequences

In addition to the `LongSeq` class shown in Code sample 3.28, a class would also be generated. See page 4-3 for details on the classes. In addition to the usual methods, there are two indexing methods defined for sequences.

**Code sample 3.29**  Two indexing methods added for _var classes representing sequences

```cpp
CORBA::Long4 operator[](CORBA::ULong index);
const CORBA::Long4 operator[](CORBA::ULong index) const;
```

### Memory management for sequences

You should carefully consider the memory management issues listed below. Code sample 3.31 contains sample C++ code that illustrates these points.
Arrays

- If the release flag was set to a non-zero value when the sequence was created, the sequence will assume management of the user’s memory. When an element is assigned, the old memory is freed before ownership of the memory on the right-hand side of the expression is assumed.

- If the release flag was set to a non-zero value when a sequence containing strings or object references was created, each element will be released before the sequence’s contents buffer is released and the object is destroyed.

- Avoid assigning a sequence element using the [] operator unless the release flag was set to one, or memory management errors may occur.

- Sequences created with the release flag set to zero should not be used as input/output parameters because memory management errors in the object server may result.

- Always use allocbuf and freebuf to create and free storage used with sequences.

Code sample 3.30  IDL specification for an unbounded sequence

```idl
// IDL
typedef sequence<string, 3> String_seq;
```

Code sample 3.31  Example of memory management with two bounded sequences

```cpp
// C++
char *static_array[] = ("1", "2", "3");
char *dynamic_array = StringSeq::allocbuf(3);

// Create a sequence, release flag is set to FALSE by default
StringSeq static_seq(3, static_array);
// Create another sequence, release flag set to TRUE
StringSeq dynamic_seq(3, dynamic_array, 1);
static_seq[1] = "1"; // old memory not freed, no copying occurs

char *str = string_alloc(2);
dynamic_seq[1] = str; // old memory is freed, no copying occurs
```

Arrays

IDL arrays are mapped to C++ arrays, which can be statically initialized. If the array elements are strings or object references, the elements of the C++ array will be of the type _var. Code sample 3.33 shows three arrays with different element types.

Code sample 3.32  IDL array definitions

```idl
// IDL
interface Intf
{
    ...
};
typedef long L[10];
typedef string S[10];
typedef Intf A[10];
```
Arrays

Code sample 3.33  Mapping IDL arrays to C++ arrays

// C++
typedef CORBA::Long L[10];
typedef CORBA::String_var S[10];
typedef Intf_var A[10];

The use of the managed type _var for strings and object references allows memory to be managed transparently when array elements are assigned.

Array slices
The array_slice type is used when passing parameters for multi-dimensional arrays. VisiBroker’s IDL compiler also generates a _slice type for arrays that contains all but the first dimension of the array. The array _slice type provides a convenient way to pass and return parameters. Code sample 3.35 shows two examples of the _slice type.

Code sample 3.34  IDL definition of multi-dimensional arrays

// IDL
typedef long L[10];
typedef string str[1][2][3];

Code sample 3.35  Generation of the _slice type

// C++
typedef CORBA::Long L_slice[10];
typedef CORBA::String_var str_slice[2][3];
typedef str_slice *str_slice_ptr;

Managed types for arrays
In addition to generating a C++ array for IDL arrays, VisiBroker’s IDL compiler will also generate a _var class. This class offers some additional features for array.

• The operator[] is overloaded to provide intuitive access to array elements.
• A constructor and assignment operator are provided that take a pointer to an array _slice object as an argument.

Code sample 3.36  IDL definition of an array

// IDL
typedef long L[10];

Code sample 3.37  _var class generated for arrays

// C++
class L_var
{
public:
   L_var();
   L_var(L_slice *slice);
   L_var(const L_var& var);
   ~L_var();
   L_var& operators(L_slice *slice);
   L_var& operators(const L_var& var);
Arrays

IDL to C++ language mapping

```cpp
CORBA::Long& operator[](CORBA::ULong index);
operator L_slice *();
operator L &() const;
...
private:
    L_slice* _ptr;
};
```

**Type-safe arrays**

A special _forany class is generated to handle arrays with elements mapped to the type any. As with the _var class, the _forany class allows you to access the underlying array type. The _forany class does not release any memory upon destruction because the _any type maintains ownership of the memory. The _forany class is not implemented as a typedef because it must be distinguishable from other types for overloading to function properly.

**Code sample 3.38**  IDL array definition

```idl
// IDL
typedef long L[10];
```

**Code sample 3.39**  _forany class generated for an IDL array

```cpp
// C++
class L_forany
{
    public:
        L_forany();
        L_forany(L_slice* slice);
        -L_forany();
        CORBA::Long& operator[](CORBA::ULong index);
        const CORBA::Long& operator[](CORBA::ULong index) const;
        operator L_slice *();
        operator L &() const;
        operator const L &() const;
        operator const L&() const;
        L_forany& operator=(const L_forany obj);
        ...
    private:
        L_slice* _ptr;
};
```

**Memory management for arrays**

VisiBroker’s IDL compiler generates two functions for allocating and releasing the memory associated with arrays. These functions allow the ORB to manage memory without having to override the new and delete operators.

**Code sample 3.40**  IDL array definition

```idl
// IDL
typedef long L[10];
```
**Principal**

**Code sample 3.41**  Methods generated for allocating and releasing array memory

```cpp
// C++
inline L_slice *L_alloc(); // Dynamically allocates array. Returns
// NULL on failure.
inline void L_free(L_slice *data); // Releases array memory allocated with
// L_alloc.
```

**Principal**

A Principal represents information about client applications that are making operation requests on an object implementation. The IDL interface of Principal does not define any operations. The Principal is implemented as a sequence of octets. The Principal is set by the client application and checked by the ORB implementation. VisiBroker for C++ treats the Principal as an opaque type and its contents are never examined by the ORB.
This chapter describes classes generated by VisiBroker’s IDL compiler, their uses, and their features. This chapter includes the following sections:

- **Overview**
- `<Interface_name>`
- `<Interface_name>ObjectWrapper`
- `_sk_<class_name>`
- `_tie_<class_name>`
- `_ptie_<Class_Name>`
- `<class_name>_var`

**Overview**

VisiBroker’s IDL compiler can generate a variety of classes that makes it easier for you to develop client applications and object servers. Many of these generated classes are available for CORBA classes.

- Stub classes
- skeleton classes
- tie classes
- ptie classes
- var classes
class <interface_name>

The <interface_name> class is generated for a particular IDL interface and, is intended for use by client applications. This class provides all of the methods defined for a particular IDL interface. When a client uses an object reference to invoke methods on the object, the stub methods are actually invoked. The stub methods allow a client operation request to be packaged, sent to the object implementation, and the results to be reflected. This entire process is transparent to the client application.

**Note** You should never modify the contents of a stub class generated by the IDL compiler.

### <Interface_name>ObjectWrapper

This class is used to derive typed object wrappers and is generated for all your interfaces when you invoke the idl2cpp command with the -obj_wrapper option, as described on page 2-3. For complete details on using the object wrapper feature, see the *VisiBroker for C++ Programmer’s Guide.*

```cpp
static void add(CORBA::ORB_ptr orb, CORBA::ObjectFactory factory, VISObjectWrapper::Location loc);
```

Adds a typed object wrapper from a client application. If more than one typed object wrapper is installed, they will be invoked in the order in which they were registered.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>orb</td>
<td>The ORB the client wishes to use, returned by the ORB_init method.</td>
</tr>
<tr>
<td>factory</td>
<td>The factory method for the object wrapper class that you want to add.</td>
</tr>
<tr>
<td>loc</td>
<td>The location of the object wrapper being added, which should be one of the following values:</td>
</tr>
<tr>
<td></td>
<td>VISObjectWrapper::Client</td>
</tr>
<tr>
<td></td>
<td>VISObjectWrapper::Server</td>
</tr>
<tr>
<td></td>
<td>VISObjectWrapper::Both</td>
</tr>
</tbody>
</table>

```cpp
static void remove(CORBA::ORB_ptr orb, CORBA::ObjectFactory factory, VISObjectWrapper::Location loc);
```

Removes an un-typed object wrapper from a server application.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>orb</td>
<td>The ORB the client wishes to use, returned by the ORB_init method.</td>
</tr>
</tbody>
</table>
The __sk__<class_name> class is an abstract base class generated by the IDL compiler, which is used to derive an object implementation class. Object implementations are usually derived from a skeleton class, which provides the necessary methods for receiving and interpreting client operation requests.

The __tie__<class_name> class is generated by the IDL compiler to aid in the creation of delegation implementations. The tie class allows you to create an object implementation that delegates all operation requests to another object. This allows you to use existing objects that you do not wish to inherit from the CORBA::Object class.

The __ptie__<Class_Name> class is generated by the IDL compiler to aid in the creation of persistent object implementations. The ptie class allows you to create an object implementation that delegates all operation requests to a persistent object, using a persistent object database. For complete details on this class, see the VisiBroker for C++ Programmer’s Guide.

The __<class_name>_var class is generated for an IDL interface and provides simplified memory management semantics.
Chapter 5

Core interfaces and classes

This chapter describes the VisiBroker for C++ core interfaces and classes. This chapter includes the following sections:

- BindOptions page 5-1
- BOA page 5-2
- CompletionStatus page 5-9
- Context page 5-9
- Exception page 5-11
- Object page 5-12
- ORB page 5-17
- Principal page 5-27
- SystemException page 5-28
- UserException page 5-30

BindOptions

struct BindOptions

This structure is used to specify options to the _bind method, described in the section “Object” on page 5-12 of this manual. Each process has a global BindOptions structure that is used for all _bind invocations that do not specify bind options. You can modify the default bind options using the Object::_default_bind_options method.

Bind options may also be set for a particular object and will remain in effect for the lifetime of the connection to that object.
Include file

The corba.h file should be included when you use this structure.

BindOptions members

CORBA::Boolean defer_bind;

If set to TRUE, the establishment of the connection between client and the object implementation will be delayed until the first client operation is issued. If set to FALSE, the _bind method will establish the connection immediately.

CORBA::Boolean enable_rebind;

If set to TRUE and the connection is lost, due to a network failure or some other error, the ORB will attempt to re-establish a connection to a suitable object implementation. If set to FALSE, no attempt will be made to reconnect the client with the object implementation.

CORBA::Long max_bind_retries;

This member specifies the number of times to retry a bind request when the OAD is busy.

CORBA::ULong send_timeout;

This member specifies the maximum time in seconds that a client is to block waiting to send an operation request. If the request times out, CORBA::NO_RESPONSE exception will be raised and the connection to the server will be destroyed. The default value of 0 implies the client should block indefinitely.

CORBA::ULong receive_timeout;

This member specifies the maximum time in seconds that a client is to block waiting for a response to an operation request. If the request times out, CORBA::NO_RESPONSE exception will be raised and the connection to the server will be destroyed. The default value of 0 implies the client should block indefinitely.

CORBA::ULong connection_timeout;

This member specifies the maximum time in seconds that a client is to wait for a connection. If the time specified is exceeded, a CORBA::NO_IMPLEMENT exception is raised. The default value of 0 implies that the default system time-out for connections should be used.

BOA

class BOA
The BOA class represents the Basic Object Adaptor and provides methods for creating and manipulating objects and object references. Object servers use the BOA to activate and deactivate object implementations and to specify the thread policy they wish to use.

You do not instantiate a BOA object. Instead, you obtain a reference to a BOA object by invoking the ORB::BOA_init method, described on page 5-18.

VisiBroker provides extensions to the CORBA BOA specification which are covered in “VisiBroker extensions to CORBA::BOA” on page 5-7. These methods provide for the management of connections, threads, and the activation of services.

**Include file**

The corba.h file should be included when you use this class.

**CORBA::BOA methods**

```c
void change_implementation(CORBA::Object_ptr,
                           CORBA::ImplementationDef_ptr impl = NULL)
```

This method changes the implementation definition associated with the specified object. You should use this method with caution. The implementation name should not be changed and you must ensure that the new implementation definition specifies the same type of object as the original definition. If the `ImplementationDef_ptr` does not point to a `CreationImplDef` pointer, this method will fail.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object_ptr</td>
<td>A pointer to the object whose implementation is to be changed.</td>
</tr>
<tr>
<td>impl</td>
<td>A pointer to the new implementation definition for this object. This must actually be a <code>CreationImplDef</code> pointer to an <code>ImplementationDef_ptr</code>.</td>
</tr>
</tbody>
</table>

```c
CORBA::Object_ptr create(const CORBA::ReferenceData_ptr&,
                         CORBA::InterfaceDef_ptr,
                         CORBA::ImplementationDef_ptr)
```

This method registers the specified implementation with the OAD.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ReferenceData_ptr</td>
<td>This parameter is not used, but is provided for compliance with the CORBA specification.</td>
</tr>
<tr>
<td>InterfaceDef_ptr</td>
<td>This parameter is not used, but is provided for compliance with the CORBA specification.</td>
</tr>
<tr>
<td>ImplementationDef_ptr</td>
<td>This pointer’s true type is <code>CreationImplDef</code> and provides the interface name, object name, path name of the executable and the activation policy and other parameters. See page 8-4 for a complete discussion of the <code>CreationImplDef</code> class.</td>
</tr>
</tbody>
</table>
**CORBA::BOA methods**

```cpp
void deactivate_impl(CORBA::ImplementationDef_ptr)
```

This method deactivates the implementation specified by the `ImplementationDef_ptr`. After this method is called, no client requests will be delivered to the object within this implementation until the objects and implementation are activated using the `impl_is_ready` methods.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ImplementationDef_ptr</td>
<td>This pointer’s true type is <code>CreationImplDef</code> and provides the interface name, object name, path name of the executable and activation policy, along with other parameters. See page 8-4 for a complete discussion of the <code>CreationImplDef</code> class.</td>
</tr>
</tbody>
</table>

```cpp
void deactivate_obj(CORBA::Object_ptr)
```

This method notifies the BOA that the specified object is to be deactivated. After this method is invoked, the BOA will not deliver any requests to the object until `obj_is_ready` or `impl_is_ready` is invoked.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object_ptr</td>
<td>A pointer to the object to be deactivated.</td>
</tr>
</tbody>
</table>

```cpp
void dispose(CORBA::Object_ptr)
```

This method unregisters the implementation of the specified object from the Object Activation Daemon. After this method is invoked, all references to the specified object will be invalid and any connections to this object implementation will be broken. If the object is allocated, the application must delete the object.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object_ptr</td>
<td>Pointer to the object to be unregistered.</td>
</tr>
</tbody>
</table>

```cpp
virtual void exit_impl_ready()
```

This method is provided for backward compatibility with earlier releases of VisiBroker for C++. This method invokes BOA::shutdown, described on page 5-7, which will cause a previous invocation of the `impl_is_ready` method to return.

```cpp
CORBA::ReferenceData get_id(CORBA::Object_ptr)
```

This method returns the reference data for the specified object. The reference data is set by the object implementation at activation time and is guaranteed to remain constant throughout the life of the object.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>obj</td>
<td>A pointer to the object whose reference data is to be returned.</td>
</tr>
</tbody>
</table>
CORBA::Principal_ptr get_principal(CORBA::Object_ptr obj, 
CORBA::Environment_ptr env)

This method returns the Principal object associated with the specified object. 
This method may only be called by an object implementation during the 
processing of a client operation request.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>obj</td>
<td>A pointer to the object whose implementation is to be changed.</td>
</tr>
<tr>
<td>env</td>
<td>A pointer to the Environment object associated with this Principal.</td>
</tr>
</tbody>
</table>

void impl_is_ready(ImplementationDef_ptr impl_def=NULL)

This method notifies the BOA that one or more objects in the server are ready 
to receive service requests. This method will block the caller until the 
exit_impl_ready method is invoked. If all objects that the implementation is 
offering have been created through C++ instantiation and activated using the 
obj_is_ready method, the ImplementationDef_ptr should not be specified.

An object implementation may offer only one object and may want to defer 
the activation of that object until a client request is received. In these cases, 
the object implementation does not need to first invoke the obj_is_ready 
method. Instead, it may simply invoke this method, passing the 
ActivationImplDef pointer for its single object.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
</table>
| impl_def  | This pointer’s true type is ActivationImplDef and provides the 
interface name, object name, path name of the executable and 
activation policy, along with other parameters. See page 8-1 for a 
complete discussion of the ActivationImplDef class. |

void impl_is_ready(const char *service_name, 
CORBA::Activator_ptr activator, 
CORBA::Boolean block = 1)

This method notifies the BOA that the activation of the object 
implementation associated with the specified service_name is to be delayed 
until a client requests the service. Once a client requests the service, the 
specified Activator object is to be used to activate the object implementation. 
If block is set to 0, this method will block the caller until the exit_impl_ready 
method is invoked.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>service_name</td>
<td>The service name associated with the specified Activator object.</td>
</tr>
<tr>
<td>activator</td>
<td>The Activator to be used to activate the object implementation</td>
</tr>
</tbody>
</table>
| block      | If set to 1, indicates that this method should block the caller. If set 
to zero, the method will not block. The default behavior is to block. |
### CORBA::BOA methods

**static CORBA::BOA* instance(const char* oa_id)**

Returns a pointer to the BOA with the specified identifier.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
</table>
| oa_id     | The identifier of the BOA to be returned. This should be one of the following values:  
            TSingle—Single threaded BOA  
            TPool—Thread pool BOA  
            TPool_liop—Thread Pool BOA with local IPC  
            TSession—Thread-per-session BOA |

**void obj_is_ready(CORBA::Object_ptr obj,  
CORBA::ImplementationDef_ptr impl_ptr = NULL)**

This method notifies the BOA that the specified object is ready for use by clients. There are two different ways to use this method:

- Objects that have been created using C++ instantiation should only specify a pointer to the object and let the `ImplementationDef_ptr` default to `NULL`.

- Objects whose creation is to be deferred until the first client request is received should specify a `NULL Object_ptr` and provide a pointer to an `ActivationImplDef` object that has been initialized. For more information, see “ActivationImplDef” on page 8-1.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>obj</td>
<td>A pointer to the object to be activated.</td>
</tr>
<tr>
<td>impl_ptr</td>
<td>A optional pointer to an <code>ActivationImplDef</code> object.</td>
</tr>
</tbody>
</table>

**static void scope(RegistrationScope val)**

Sets the registration scope for an object.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
</table>
| val       | The scope for this BOA. Must be one of the following values:  
            LOCAL_SCOPE—For transient objects.  
            GLOBAL_SCOPE—For objects registered with the Smart Agent |

**static RegistrationScope scope()**

Returns this BOA’s scope.

**static void CORBA::release(CORBA::BOA_ptr boa)**

This static method releases the specified BOA pointer. Once the object’s reference count reaches zero, the object is automatically deleted.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>boa</td>
<td>A valid BOA pointer.</td>
</tr>
</tbody>
</table>
VisiBroker extensions to CORBA::BOA

static RegistrationScope scope()

This static method returns the registration scope of the BOA. The registration scope of an object can be `SCOPE_GLOBAL` or `SCOPE_LOCAL`. Only objects with a global scope are registered with the osagent.

static void scope(RegistrationScope val)

This static method changes the registration scope of the BOA to the specified value.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>val</td>
<td>The new registration scope for the BOA; either <code>SCOPE_LOCAL</code> or <code>SCOPE_GLOBAL</code>.</td>
</tr>
</tbody>
</table>

void shutdown()

This method will cause a previous invocation of the `impl_is_ready` method to return.

CORBA::Object_ptr string_to_object(const char *)

This method converts a stringified object reference, created with the `object_to_string` method described on page 5-24, back into an object reference that may be used to invoke methods on the object.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>const char *</td>
<td>The string to be converted back to an object reference.</td>
</tr>
</tbody>
</table>

static CORBA::BOA_ptr _duplicate(CORBA::BOA_ptr ptr)

This static method duplicates the BOA pointer that is passed in as a parameter.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ptr</td>
<td>A BOA pointer.</td>
</tr>
</tbody>
</table>

static CORBA::BOA_ptr _nil()

This static method returns a NULL BOA pointer that can be used for initialization purposes.

VisiBroker extensions to CORBA::BOA

This method is used by servers to set the maximum number of connection to be allowed. This property can also be set by using the command-line argument `-OAConnectionMax`, described in Appendix A, “Using command-line options.”

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>max_conn</td>
<td>The maximum number of connections to be allowed.</td>
</tr>
</tbody>
</table>
CORBA:ULong \texttt{connection\_max}()

This method returns the maximum number of connections that will be allowed.

\textbf{void} \texttt{delay\_registration}\texttt{(CORBA::Boolean flag)}

Enables or disables the delayed registration of objects, allowing you to register an object with more than one BOA. If you are using multiple BOA objects, you should enable delayed registration on all but one BOA. This ensures that only one BOA will register the objects offered by your server when the \texttt{obj\_is\_ready} method is invoked.

\begin{tabular}{|l|l|}
\hline
\textbf{Parameter} & \textbf{Description} \\
\hline
flag & Set to TRUE to enable delayed registration. Set to FALSE to disable delayed registration. \\
\hline
\end{tabular}

CORBA::Boolean \texttt{delay\_registration}()

Returns \texttt{TRUE} if delayed registration is enabled for this BOA, otherwise \texttt{FALSE} is returned.

CORBA:ULong \texttt{thread\_max}()

This method returns the maximum number of threads to be allowed if the TSession thread policy has been selected.

\textbf{void} \texttt{thread\_max}\texttt{(CORBA::ULong max)}

This method sets the maximum number of threads to be allowed when the TSession thread policy has been selected. If the current number of threads exceeds \texttt{size}, the necessary number of extra threads will be destroyed as soon as they are no longer in use.

\begin{tabular}{|l|l|}
\hline
\textbf{Parameter} & \textbf{Description} \\
\hline
max & The maximum number of threads to be allowed. \\
\hline
\end{tabular}

CORBA:ULong \texttt{thread\_stack\_size}()

This method returns the maximum number of threads to be allowed when the TPool thread policy has been selected.

\textbf{void} \texttt{thread\_stack\_size}\texttt{(CORBA::ULong size)}

This method sets the maximum number of threads to be allowed when the TPool thread policy has been selected. If the current number of threads exceeds \texttt{size}, the necessary number of extra threads will be destroyed as soon as they are no longer in use.

\begin{tabular}{|l|l|}
\hline
\textbf{Parameter} & \textbf{Description} \\
\hline
size & The new stack size to be set. \\
\hline
\end{tabular}
CompletionStatus

enum CompletionStatus
This enumeration represents how an operation request completed.

CompletionStatus members

COMPLETED_YES = 0 Indicates the operation request completed successfully.
COMPLETED_NO = 1 Indicates the operation request was not completed, due to some sort of exception or error.
COMPLETED_MAYBE = 2 Indicates that the operation request may have completed, in spite of an exception or error.

Context

class CORBA::Context
The Context class represents information about a client application’s environment that is passed to a server as an implicit parameter during static or dynamic method invocations. It can be used to communicate special information that needs to be associated with a request, but is not part of the method’s argument list.

The Context class consists of a list of properties, stored as name-value pairs, and provides methods for setting and manipulating those properties. A Context contains an NVList object and chains the name-value pairs together.

A Context_var class is also available and provides simpler memory management semantics.

See also ORB::get_default_context on page 5-23

Include file

The corba.h file should be included when you use this class.

Context methods

const char *context_name() const;
This method returns the name used to identify this context. If no name was provided when this object was created, an NULL value is returned.
Context methods

CORBA::Status create_child(const char *name, CORBA::Context_ptr&);

This method creates a child Context for this object.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>The name of the new Context object.</td>
</tr>
<tr>
<td>Context_ptr</td>
<td>A reference to newly created child Context.</td>
</tr>
</tbody>
</table>

CORBA::Status delete_values(const char *name);

This method deletes one or more properties from this object. The name may contain a trailing "*" wildcard character to delete all matching properties. A single asterisk can be specified to delete all properties.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>The name of the property, or properties, to be deleted.</td>
</tr>
</tbody>
</table>

CORBA::Status get_values(const char *start_scope,
CORBA::Flags,
const char *name,
CORBA::NVList_ptr&)
const;

This method searches the Context object hierarchy and retrieves one or more of the name/value pairs specified by the name parameter. It then creates an NVList object, places the name/value pairs in the NVList, and returns a reference to that object.

The start_scope parameter specifies the name of the context where the search is to begin. If the property is not found, the search continues up Context object hierarchy until a match is found or until there are no more Context objects to search.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>start_scope</td>
<td>The name of the Context object at which to start the search. If omitted, the search begins with this object. The search scope can be restricted to just this object by specifying CORBA::CTX_RESTRICT_SCOPE.</td>
</tr>
<tr>
<td>Flags</td>
<td>An exception is raised if no matching context name is found.</td>
</tr>
<tr>
<td>name</td>
<td>The property name to search for. A trailing &quot;*&quot; wildcard character may be used to retrieve all properties that match name.</td>
</tr>
<tr>
<td>NVList_ptr</td>
<td>A reference to the list of properties found.</td>
</tr>
</tbody>
</table>

CORBA::Context_ptr parent();

This method returns a pointer to the parent Context. If there is no parent Context, a NULL value is returned.
CORBA::Status set_one_value(const char *name, const CORBA::Any&);

This method adds a property to this object, using the specified name and value.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>The property’s name.</td>
</tr>
<tr>
<td>const Any&amp;</td>
<td>The property’s value.</td>
</tr>
</tbody>
</table>

CORBA::Status set_values(CORBA::NVList_ptr);

This method adds one or more properties to this object, using the name/value pairs specified in the NVList. When you create the NVList object to be used as an input parameter to this method, the Flags field must be set to zero and each Any object added to the NVList must set its TypeCode to TC_string. For more information on the NVList class, see page 6-20 in this guide.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NVList_ptr</td>
<td>A list of name/value pairs to be added to this object.</td>
</tr>
</tbody>
</table>

static CORBA::Context_ptr _duplicate(CORBA::Context_ptr ctx);

This method duplicates the specified object.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ctx</td>
<td>The object to be duplicated.</td>
</tr>
</tbody>
</table>

static CORBA::Context_ptr _nil();

This method returns a NULL Context_ptr suitable for initialization purposes.

static void _release(CORBA::Context_ptr ctx);

This static method releases the specified Context object. Once the object’s reference count reaches zero, the object is automatically deleted.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ctx</td>
<td>The object to be released.</td>
</tr>
</tbody>
</table>

### Exception

class CORBA::Exception

The Exception class is the base class of the system exception and user exception classes. For more information, see “SystemException” on page 5-28 in this guide.
Include file

You should include the corba.h file when using this class.

Object

class CORBA::Object

All ORB objects are derived from the Object class, which provides methods for binding clients to objects and manipulating object references as well as querying and setting an object’s state. The methods offered by the Object class are implemented by the ORB.

VisiBroker provides extensions to the CORBA Object specification. These are covered in “VisiBroker extensions to CORBA::Object” on page 5-15.

Include file

You should include the file corba.h when using this class.

CORBA::Object methods

CORBA::Status _create_request(CORBA::Context_ptr ctx,
    const char *operation,
    CORBA::NVList_ptr arg_list,
    CORBA::NamedValue_ptr result,
    CORBA::Request_ptr& request,
    Flags req_flags);

This method creates a Request for an object implementation that is suitable for invocation with the Dynamic Invocation Interface.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ctx</td>
<td>The Context associated with this request. For more information, see “CompletionStatus” on page 5-9.</td>
</tr>
<tr>
<td>operation</td>
<td>The name of the operation to be performed on the object implementation.</td>
</tr>
<tr>
<td>arg_list</td>
<td>A list of arguments to pass to the object implementation. See “NVList” on page 6-20 for more information.</td>
</tr>
<tr>
<td>result</td>
<td>The result of the operation. See “NamedValue” on page 6-19 for more information.</td>
</tr>
<tr>
<td>request</td>
<td>A pointer to the Request that is created. See “Request” on page 6-23 for more information.</td>
</tr>
<tr>
<td>req_flags</td>
<td>This flag must be set to OUT_LIST_MEMORY if one or more of the NamedValue items in arg_list is an output argument.</td>
</tr>
</tbody>
</table>
CORBA::Object methods

CORBA::Status _create_request(CORBA::Context_ptr ctx,
    const char *operation,
    CORBA::NVList_ptr arg_list,
    CORBA::NamedValue_ptr result,
    CORBA::ExceptionList_ptr eList,
    CORBA::ContextList_ptr ctxList,
    CORBA::Request_ptr& request,
    Flags req_flags);

This method creates a Request for an object implementation that is suitable for
invocation with the Dynamic Invocation Interface.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ctx</td>
<td>The Context associated with this request. For more information, see</td>
</tr>
<tr>
<td></td>
<td>“CompletionStatus” on page 5-9.</td>
</tr>
<tr>
<td>operation</td>
<td>The name of the operation to be performed on the object implementation.</td>
</tr>
<tr>
<td>arg_list</td>
<td>A list of arguments to pass to the object implementation. See</td>
</tr>
<tr>
<td></td>
<td>“NVList” on page 6-20 for more information.</td>
</tr>
<tr>
<td>result</td>
<td>The result of the operation. See “NamedValue” on page 6-19 for more</td>
</tr>
<tr>
<td></td>
<td>information.</td>
</tr>
<tr>
<td>eList</td>
<td>A list of exceptions for this request.</td>
</tr>
<tr>
<td>ctxList</td>
<td>A list of Context objects for this request.</td>
</tr>
<tr>
<td>request</td>
<td>A pointer to the Request that is created. See “Request” on page 6-23 for</td>
</tr>
<tr>
<td></td>
<td>more information.</td>
</tr>
<tr>
<td>req_flags</td>
<td>This flag must be set to OUT_LIST_MEMORY if one or more of the</td>
</tr>
<tr>
<td></td>
<td>NamedValue items in arg_list is an output argument.</td>
</tr>
</tbody>
</table>

static CORBA::Object_ptr _duplicate(CORBA::Object_ptr obj);

This static method duplicates the specified Object_ptr and returns a pointer to
the object. The object’s reference count is increased by one.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>obj</td>
<td>The object pointer to be duplicated.</td>
</tr>
</tbody>
</table>

CORBA::ImplementationDef_ptr _get_implementation();

This method returns a pointer to this object’s implementation definition. See
“ImplementationDef” on page 8-6 for more information.

CORBA::InterfaceDef_ptr _get_interface();

This method returns a pointer to this object’s interface definition. See
“InterfaceDef” on page 7-18 for more information.

CORBA::ULong _hash(CORBA::ULong maximum);

This method returns a hash value for this object. This value will not change
for the lifetime of this object, however the value is not necessarily unique. If
two objects return the different hash values, then they are not identical. The upper bound of the hash value may be specified. The lower bound is zero.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>maximum</td>
<td>The upper bound of the hash value returned.</td>
</tr>
</tbody>
</table>

`CORBA::Boolean _is_a(const char *logical_type_id);`

This method returns `TRUE` if this object implements the interface associated with the repository id. Otherwise, `FALSE` is returned.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>logical_type_id</td>
<td>The repository identifier to check.</td>
</tr>
</tbody>
</table>

`CORBA::Boolean _is_equivalent(CORBA::Object_ptr other_object);`

This method returns `TRUE` if the specified object pointer and this object point to the same object implementation. Otherwise, `FALSE` is returned.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>other_object</td>
<td>Pointer to an object that is to be compared to this object.</td>
</tr>
</tbody>
</table>

`static CORBA::Object_ptr _nil();`  

This static method returns a NULL pointer suitable for initialization purposes.

`CORBA::Boolean _non_existent();`  

This method returns `TRUE` if the object represented by this object reference no longer exists.

`CORBA::Object_ptr _resolve_reference(const char* id);`

Your client application can invoke this method on an object reference to resolve the server-side interface with the specified service identifier. This method causes the ORB::resolve_initial_references method, described on page 5-25, to be invoked on the server-side to resolve the specified service. An object reference is returned which your client can narrow to the appropriate server type.

This method is typically used by client applications that wish to use the ORB Management interfaces, described in Chapter 15 and in the *VisiBroker for C++ Programmer’s Guide*, to manage a server’s attributes.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>The name of the interface to be resolved on the server-side.</td>
</tr>
</tbody>
</table>
VisiBroker extensions to CORBA::Object

CORBA::Request_ptr _request(const char* operation);

This method creates a Request suitable for invoking methods on this object. A pointer to the Request object is returned. See “Request” on page 6-23 for more information.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>operation</td>
<td>The name of the object method to be invoked.</td>
</tr>
</tbody>
</table>

VisiBroker extensions to CORBA::Object

static const CORBA::BindOptions* _bind_options() const;

This method returns a pointer to the bind options that will be used for this object only. For more information, see “BindOptions” on page 5-1.

void _bind_options(const CORBA::BindOptions& opt);

This method sets the bind options for this object only. The options that are set will remain in effect for the lifetime of the proxy object. Any changes to timeout values will apply to all subsequent send and receive operations as well as any re-bind operations. For more information, see “BindOptions” on page 5-1.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>opt</td>
<td>The new bind options for this object.</td>
</tr>
</tbody>
</table>

static CORBA::Object_ptr _bind_to_object(const char *rep_id, const char *object_name=NULL, const char *host_name=NULL, const CORBA::BindOptions *options=NULL, CORBA::ORB_ptr orb=NULL);

This method attempts to bind to the object with the specified repository_id and object_name, on the specified host, using the specified BindOptions and ORB.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rep_id</td>
<td>The repository ID of the desired object.</td>
</tr>
<tr>
<td>object_name</td>
<td>The name of the desired object.</td>
</tr>
<tr>
<td>host_name</td>
<td>The name of the desired host where the object implementation is executing.</td>
</tr>
<tr>
<td>options</td>
<td>The bind options for this connection. See page 5-1 for more information.</td>
</tr>
<tr>
<td>orb</td>
<td>The ORB to use.</td>
</tr>
</tbody>
</table>

CORBA::BOA_ptr _boa() const;

This method returns a pointer to the Basic Object Adaptor with which this object is registered.
VisiBroker extensions to CORBA::Object

static CORBA::Object _clone(CORBA::Object_ptr obj);

This method clones the specified object reference.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>obj</td>
<td>The object reference to be cloned.</td>
</tr>
</tbody>
</table>

static const CORBA::BindOptions * _default_bind_options();

This method returns a pointer to the global, per client process BindOptions. For more information, see “BindOptions” on page 5-1.

static void _default_bind_options(const CORBA::BindOptions& opt);

This method sets the bind options that will be used by default for all _bind invocations that do not specify their own bind options. For more information, see “BindOptions” on page 5-1.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>opt</td>
<td>The new, default bind options to set.</td>
</tr>
</tbody>
</table>

static const CORBA::Principal_ptr _default_principal();

This method returns a pointer to the default Principal for this object.

static void _default_principal(const CORBA::Principal& principal);

This static method sets the global, default Principal for the specified object. For more information, see “Principal” on page 5-27.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>principal</td>
<td>The default, global Principal to use for this object.</td>
</tr>
</tbody>
</table>

static const CORBA::TypeInfo * _desc();

Returns type information for this object.

const char * _interface_name() const;

This method returns this object’s interface name.

CORBA::Boolean _is_bound() const;

This method returns TRUE if the client process has established a connection to an object implementation.

CORBA::Boolean _is_local() const;

This method returns TRUE if the object implementation resides within the same process or address space as the client application.

CORBA::Boolean _is_persistent() const;

This method returns TRUE if this object is a persistent object, and FALSE if it is transient.
CORBA::Boolean _is_remote() const;

This method returns TRUE if the object implementation resides in a different process or address space than the client application. The client and object implementation may or may not reside on the same host.

const char *_object_name() const;

This method returns the object name associated with this object.

const CORBA::Principal_ptr _principal() const;

This method returns a pointer to the Principal associated with this object. For more information, see “Principal” on page 5-27.

void _principal(const CORBA::Principal& principal);

This method sets the Principal for this object. For more information, see “Principal” on page 5-27.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>principal</td>
<td>The Principal to use for this object. If the Principal is passed by reference, a new copy is created and used. If the Principal is passed as a pointer, that object will be used.</td>
</tr>
</tbody>
</table>

CORBA::Long _ref_count() const;

Returns the reference count for this object.

void _release();

Decrement this object’s reference count and releases the object if the reference count has reached 0.

const char * _repository_id() const;

This method returns this object’s repository identifier.

class CORBA::ORB

The ORB class provides an interface to the Object Request Broker. It offers methods to the client object, independent of the particular Object or Object Adaptor.

VisiBroker provides extensions to the CORBA ORB that are covered in “VisiBroker extensions to CORBA::ORB” on page 5-26. These methods provide for the management of connections, threads, and the activation of services.

Include file

You should include the file corba.h when using this class.
CORBA::ORB methods

static CORBA::TypeCode_ptr create_alias_tc(const char *repository_id,
const char *type_name,
CORBA::TypeCode_ptr original_type);

This static method dynamically creates a TypeCode for the alias with the specified type and name.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>repository_id</td>
<td>The identifier generated by the IDL compiler or constructed dynamically.</td>
</tr>
<tr>
<td>type_name</td>
<td>The name of the alias’s type.</td>
</tr>
<tr>
<td>original_type</td>
<td>The type of the original for which this alias is being created.</td>
</tr>
</tbody>
</table>

static TypeCode_ptr create_array_tc(CORBA::Ulong bound,
TypeCode_ptr element_type);

This static method dynamically creates a TypeCode for an array.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bound</td>
<td>The maximum number of array elements.</td>
</tr>
<tr>
<td>element_type</td>
<td>The type of elements stored in this array.</td>
</tr>
</tbody>
</table>

CORBA::DynAny_ptr create_basic_dyn_any(CORBA::TypeCode_ptr type);

This method creates a DynAny object for a basic type, such as CORBA::Long and CORBA::String.

An InconsistentTypeCode exception will be raised if the type is invalid.

Note

The type of a DynAny cannot be changed during the lifetime of the object.
DynAny objects cannot be used as parameters on operation requests or DII requests nor can they be externalized using the ORB::object_to_string method. See “DynAny” on page 6-7 for complete details.

Parameter Description

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>The TypeCode of the values this object is to hold. See “TypeCode” on page 6-30 for a list of valid values.</td>
</tr>
</tbody>
</table>

CORBA::DynAny_ptr create_dyn_any(CORBA::Any& value)

Creates a DynAny object initializing it with the specified value.

Note

DynAny objects cannot be used as parameters on operation requests or DII requests nor can they be externalized using the ORB::object_to_string method. See “DynAny” on page 6-7 for complete details.

Parameter Description

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>An Any object used to initialize this object.</td>
</tr>
</tbody>
</table>
**CORBA::ORB methods**

**CORBA::DynArray_ptr create_dyn_array(CORBA::TypeCode_ptr type)**

Creates a `DynArray` object for holding an array of data types with the specified `TypeCode`. An `InconsistentTypeCode` exception will be raised if the `type` is invalid.

**Note**

The type of a `DynAny` cannot be changed during the lifetime of the object. `DynAny` objects cannot be used as parameters on operation requests or DII requests nor can they be externalized using the `ORB::object_to_string` method. See “DynAny” on page 6-7 for complete details.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>The <code>TypeCode</code> of the values this object is to hold. See “TypeCode” on page 6-30 for a list of valid values.</td>
</tr>
</tbody>
</table>

**CORBA::DynEnum_ptr create_dyn_enum(CORBA::TypeCode_ptr type)**

Creates a `DynEnum` object for the specified data type. `DynEnum` objects are used to dynamically create or interpret enumeration values that were not defined at run-time. An `InconsistentTypeCode` exception will be raised if the `type` is invalid.

**Note**

`DynEnum` objects cannot be used as parameters on operation requests or DII requests nor can they be externalized using the `ORB::object_to_string` method. See “DynEnum” on page 6-11 for complete details.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>The <code>TypeCode</code> of the value this object is to hold should have a <code>TCKind</code> of <code>tk_enum</code>.</td>
</tr>
</tbody>
</table>

**CORBA::DynSequence_ptr create_dyn_sequence(CORBA::TypeCode_ptr type)**

Creates a `DynSequence` object for holding an sequence of values with the specified type. An `InconsistentTypeCode` exception will be raised if the `type` is invalid.

**Note**

The type of a `DynSequence` cannot be changed during the lifetime of the object. `DynSequence` object cannot be used as parameters on operation requests or DII requests nor can they be externalized using the `ORB::object_to_string` method. See “DynSequence” on page 6-13 for complete details.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>The <code>TypeCode</code> of the values this object is to hold. See “TypeCode” on page 6-30 for a list of valid values.</td>
</tr>
</tbody>
</table>

**CORBA::DynStruct_ptr create_dyn_struct(CORBA::TypeCode_ptr type)**

Creates a `DynStruct` object with the specified type. `DynStruct` objects are used to dynamically create or interpret structures whose type was not necessarily defined at run-time.

**Core interfaces and classes** 5-19
An InconsistentTypeCode exception will be raised if the type is invalid.

**Note**

DynStruct object cannot be used as parameters on operation requests or DII requests nor can they be externalized using the ORB::object_to_string method. See “DynStruct” on page 6-13 for complete details.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>The TypeCode of the structure this object is to hold. See “TypeCode” on page 6-30 for a list of valid values.</td>
</tr>
</tbody>
</table>

**CORBA::DynUnion_ptr create_dyn_union(CORBA::TypeCode_ptr type)**

Creates a DynUnion object with the specified type. DynUnion objects are used to dynamically create or interpret unions that were not necessarily defined at run-time.

An InconsistentTypeCode exception will be raised if the type is invalid.

**Note**

DynUnion object cannot be used as parameters on operation requests or DII requests nor can they be externalized using the ORB::object_to_string method. See “DynUnion” on page 6-14 for complete details.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>The TypeCode of the value this object is to hold. See “TypeCode” on page 6-30 for a list of valid values.</td>
</tr>
</tbody>
</table>

**static CORBA::TypeCode_ptr create_enum_tc(**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>repository_id</td>
<td>The identifier generated by the IDL compiler or constructed dynamically.</td>
</tr>
<tr>
<td>type_name</td>
<td>The name of the enumeration’s type.</td>
</tr>
<tr>
<td>members</td>
<td>A list of values for the enumeration’s members.</td>
</tr>
</tbody>
</table>

**CORBA::Status create_environment(CORBA::Environment_ptr& env);**

This method creates an Environment object and returns a reference to the created object.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>env</td>
<td>The reference that will be set to point to the newly created Environment.</td>
</tr>
</tbody>
</table>
static CORBA::TypeCode_ptr create_exception_tc(const char *repository_id, const char *type_name, const CORBA::StructMemberSeq& members);

This static method dynamically creates a TypeCode for an exception with the specified type and members.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>repository_id</td>
<td>The identifier generated by the IDL compiler or constructed dynamically.</td>
</tr>
<tr>
<td>type_name</td>
<td>The name of the structure’s type.</td>
</tr>
<tr>
<td>members</td>
<td>A list of values for the structure members.</td>
</tr>
</tbody>
</table>

static CORBA::TypeCode_ptr create_interface_tc(const char *repository_id, const char *type_name);

This static method dynamically creates a TypeCode for the interface with the specified type.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>repository_id</td>
<td>The identifier generated by the IDL compiler or constructed dynamically.</td>
</tr>
<tr>
<td>type_name</td>
<td>The name of the interface’s type.</td>
</tr>
</tbody>
</table>

CORBA::Status create_named_value(CORBA::NamedValue_ptr& val);

This method creates a NamedValue object.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>val</td>
<td>Initialized to point to the newly created NamedValue object.</td>
</tr>
</tbody>
</table>

CORBA::Status create_list(CORBA::Long num, CORBA::NVList_ptr& nvlist);

This method creates an NVList with the specified number of elements and returns a reference to the list.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>num</td>
<td>The number of elements in the list.</td>
</tr>
<tr>
<td>nvlist</td>
<td>Initialized to point to the newly created list.</td>
</tr>
</tbody>
</table>
CORBA::ORB methods

CORBA::Status create_operation_list(CORBA::OperationDef_ptr op, CORBA::NVList& nvlist);

This method creates an argument list for the specified OperationDef object.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>Pointer to the operation definition whose argument list is to be created.</td>
</tr>
<tr>
<td>nvlist</td>
<td>A reference to the newly created argument list.</td>
</tr>
</tbody>
</table>

static CORBA::TypeCode_ptr create_recursive_sequence_tc(CORBA::Ulong bound, CORBA::Ulong offset);

This static method dynamically creates a TypeCode for a recursive sequence. The result of this method can be used to create other types. The offset parameter determines which enclosing TypeCode describes the elements of this sequence.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bound</td>
<td>The maximum number of sequence elements.</td>
</tr>
<tr>
<td>offset</td>
<td>Position within the buffer where the type code for the current element was previously generated.</td>
</tr>
</tbody>
</table>

static CORBA::TypeCode_ptr create_sequence_tc(CORBA::Ulong bound, CORBA::TypeCode_ptr element_type);

This static method dynamically creates a TypeCode for a sequence.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bound</td>
<td>The maximum number of sequence elements.</td>
</tr>
<tr>
<td>element_type</td>
<td>The type of elements stored in this sequence.</td>
</tr>
</tbody>
</table>

static CORBA::TypeCode_ptr create_string_tc(CORBA::Ulong bound);

This static method dynamically creates a TypeCode for a string.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bound</td>
<td>The maximum length of the string.</td>
</tr>
</tbody>
</table>
static CORBA::TypeCode_ptr create_struct_tc(const char *repository_id,
const char *type_name,
const CORBA::StructMemberSeq& members);

This static method dynamically creates a TypeCode for the structure with the specified type and members.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>repository_id</td>
<td>The identifier generated by the IDL compiler or constructed dynamically.</td>
</tr>
<tr>
<td>type_name</td>
<td>The name of the structure's type.</td>
</tr>
<tr>
<td>members</td>
<td>A list of values for the structure members.</td>
</tr>
</tbody>
</table>

static CORBA::TypeCode_ptr create_union_tc(const char *repository_id,
CORBA::TypeCode_ptr descriminator_type,
const CORBA::UnionMemberSeq& members);

This static method dynamically creates a TypeCode for a union with the specified type, discriminator and members.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>repository_id</td>
<td>The identifier generated by the IDL compiler or constructed dynamically.</td>
</tr>
<tr>
<td>type_name</td>
<td>The name of the union's type.</td>
</tr>
<tr>
<td>descriminator_type</td>
<td>The discriminating type for the union.</td>
</tr>
<tr>
<td>members</td>
<td>A list of values for the union members.</td>
</tr>
</tbody>
</table>

CORBA::Status get_default_context(CORBA::Context_ptr&);

This method returns the default per-process Context maintained by VisiBroker. The default Context is often used in constructing DII requests. See “Context” on page 5-9 for more information.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CORBA::Context_ptr&amp;</td>
<td>The property's value.</td>
</tr>
</tbody>
</table>

CORBA::Status get_next_response(CORBA::RequestSeq& req);

This method blocks waiting for the response associated with a deferred request. You can use the ORB::poll_next_response method to determine if there is a response waiting to be received before calling this method.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>req</td>
<td>Set to point to the request that has been received.</td>
</tr>
</tbody>
</table>
**CORBA::ORB methods**

```cpp
CORBA::StringSequence *list_initial_services();
```

This method returns a list of names of any object services that are available to your application. These services may include the Location Service, Interface Repository, Name Service, or Event Service. You can use any of the returned names with the `ORB::resolve_initial_references` method, described on page 5-25, to obtain the top-level object for that service.

```cpp
char *object_to_string(CORBA::Object_ptr obj);
```

This method converts the specified object reference to a string, a process referred to as “stringification” in the CORBA specification. Object references that have been converted to strings can be stored in files, for example. This is an ORB method because different ORB implementations may have different conventions for representing object references as strings.

**Note**

While an object reference can be made persistent by saving it to a file, the object itself is not made persistent.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>obj</td>
<td>Pointer to an object that is to be converted to a string.</td>
</tr>
</tbody>
</table>

```cpp
CORBA::BOA_ptr ORB::BOA_init(int& argc, char *const *argv,
                          const char *boa_identifier = (char *)NULL);
```

This ORB method returns a handle to the BOA and specifies optional networking parameters. The `argc` and `argv` parameters are the same parameters passed to the object implementation process when it is started. See Appendix A, “Using command-line options” for a complete description of the `BOA_init` options that may be specified.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>argc</td>
<td>The number of arguments passed.</td>
</tr>
<tr>
<td>argv</td>
<td>An array of char pointers to the arguments. All but two of the arguments take the form of a keyword and a value and are shown below. This method will ignore any keywords that it does not recognize.</td>
</tr>
<tr>
<td>boa_identifier</td>
<td>Identifies the type of BOA to be used. TPool is used if multiple thread support is desired. TSingle is used if the implementation does not use threads.</td>
</tr>
</tbody>
</table>

```cpp
static CORBA::ORB_ptr ORB_init(int& argc, char *const *argv,
                          const char *orb_id = NULL);
```

This method initializes the ORB and is used by both clients and object implementations. It returns a pointer to the ORB that can be used to invoke ORB methods. The `argc` and `argv` parameters passed to the application’s main function can be passed directly to this method. Arguments accepted by this method take the form of name-value pairs which allows them to be distinguished from other command line arguments. See Appendix A, “Using
command-line options” for a complete description of the ORB_init options that may be specified.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>argc</td>
<td>The number of arguments passed.</td>
</tr>
<tr>
<td>argv</td>
<td>An array of char pointers to the arguments. All but two of the arguments take the form of a keyword and a value. This method will ignore any keywords that it does not recognize.</td>
</tr>
<tr>
<td>boa_identifier</td>
<td>Identifies the type of ORB to be used. The default is IIOP.</td>
</tr>
</tbody>
</table>

CORBA::Boolean poll_next_response();

This method returns TRUE if a response to a deferred request has been received, otherwise FALSE is returned. This call does not block.

CORBA::Object_ptr resolve_initial_references(const char * identifier);

This method resolves one of the names returned by the ORB::list_initial_services method, described on page 5-24, to its corresponding implementation object. The resolved object which is returned can then be narrowed to the appropriate server type. If the specified service cannot be found, an InvalidName exception will be raised.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>identifier</td>
<td>The name of the service whose top-level object is to be returned. The identifier is not the name of the object to be returned.</td>
</tr>
</tbody>
</table>

CORBA::Status send_multiple_requests_deferred(const CORBA::RequestSeq& req);

This method sends all the client requests in the specified sequence as deferred requests. The ORB will not wait for any responses from the object implementation. The client application is responsible for retrieving the responses to each request using the ORB::get_next_response method.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>req</td>
<td>A sequence of deferred requests to be sent.</td>
</tr>
</tbody>
</table>

CORBA::Status send_multiple_requests_oneway(const CORBA::RequestSeq& req);

This method sends all the client requests in the specified sequence as one-way requests. The ORB does not wait for a response from any of the requests because one-way requests do not generate responses from the object implementation.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>req</td>
<td>A sequence of oneway requests to be sent.</td>
</tr>
</tbody>
</table>
**VisiBroker extensions to CORBA::ORB**

CORBA::Object_ptr **string_to_object**(const char *str);

This method converts a string representing an object into an object pointer. The string must have been created using the ORB::object_to_string method.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>str</td>
<td>A pointer to a string representing an object.</td>
</tr>
</tbody>
</table>

static CORBA::ORB_ptr **_duplicate**(CORBA::ORB_ptr ptr);

This static method duplicates the specified ORB pointer and returns a pointer to the duplicated ORB.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ptr</td>
<td>The ORB pointer to be duplicated.</td>
</tr>
</tbody>
</table>

static CORBA::ORB_ptr **_nil**();

This static method returns a NULL ORB pointer suitable for initialization purposes.

**VisiBroker extensions to CORBA::ORB**

CORBA::Object_ptr **bind**(const char *rep_id,
const char *object_name = NULL,
const char *host_name = NULL,
CORBA::BindOptions opt = NULL);

This method allows you obtain a generic object reference to an object by specifying the repository id of the object and optionally, its object name and host name where it is implemented.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rep_id</td>
<td>The identifier generated by the IDL compiler or constructed dynamically for the object.</td>
</tr>
<tr>
<td>object_name</td>
<td>The name of the object. This is an optional parameter.</td>
</tr>
<tr>
<td>host_name</td>
<td>The host name where the object implementation is located. This may be specified as an IP address or as a fully qualified host name.</td>
</tr>
<tr>
<td>opt</td>
<td>Any bind options for the object. Bind options are described on page 5-1.</td>
</tr>
</tbody>
</table>

CORBA::ULong **connection_count**();

This method is used by client applications to return the current number of active connections.
void connection_max(CORBA::ULong max_conn)

This method is used by client applications to set the maximum number of connections to be allowed. This property can also be set by using the command-line argument -OAConnectionMax, described in Appendix A, “Using command-line options.”

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>max_conn</td>
<td>The maximum number of connections to be allowed.</td>
</tr>
</tbody>
</table>

CORBA::ULong connection_max()

This method is used by client applications to return the maximum number of connections that will be allowed.

static CORBA::TypeCode_ptr create_wstring_tc(CORBA::Ulong bound);

This static method dynamically creates a TypeCode for a Unicode string.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bound</td>
<td>The maximum length of the string.</td>
</tr>
</tbody>
</table>

static void shutdown();

This method will cause a previous invocation of the impl_is_ready method to return. All object adapters will be shutdown and any associated memory will be freed.

typedef OctetSequence Principal

The Principal is used to represent the client application on whose behalf a request is being made. An object implementation can accept or reject a bind request, based on the contents of the client’s Principal.

Include file

You should include the file corba.h when using this typedef.

Principal methods

The BOA class provides the get_principal method, described on page 5-5, which returns a pointer to the Principal associated with an object. The Object class also provides methods for getting and setting the Principal.
SystemException

The SystemException class is used to report standard system errors encountered by the ORB or by the object implementation. This class is derived from the Exception class, described on page 5-11, which provides methods for printing the name and details of the exception to an output stream.

SystemException objects include a completion status which indicates if the operation that caused the exception was completed. SystemException objects also have a minor code that can be set and retrieved.

Include file

The corba.h file should be included when you use this class.

SystemException methods

CORBA::SystemException(CORBA::ULong minor = 0,
                        CORBA::CompletionStatus status = CORBA::COMPLETED_NO);

This method creates a SystemException object with the specified properties.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>minor</td>
<td>The minor code.</td>
</tr>
<tr>
<td>status</td>
<td>The completion status, one of CORBA::COMPLETED_YES, CORBA::COMPLETED_NO, or CORBA::COMPLETED_MAYBE. If a value is not specified, the default value of CORBA::COMPLETED_NO will be used.</td>
</tr>
</tbody>
</table>

CORBA::CompletionStatus completed() const;

This method returns TRUE if this object’s completion status is set to COMPLETED_YES.

void completed(CORBA::CompletionStatus status);

This method sets the completion status for this object.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>status</td>
<td>The completion status, one of COMPLETED_YES, COMPLETED_NO, or COMPLETED_MAYBE.</td>
</tr>
</tbody>
</table>

CORBA::ULong minor() const;

This method returns this object’s minor code.
### SystemException methods

**void minor(CORBA::ULong val);**

This method sets the minor code for this object.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>val</td>
<td>The minor code.</td>
</tr>
</tbody>
</table>

**static CORBA::SystemException * _narrow(CORBA::Exception * exc);**

This method attempts to narrow the specified Exception pointer to a SystemException pointer. If the supplied pointer points to a SystemException object or an object derived from SystemException, a pointer to the object is returned. If the supplied pointer does not point to a SystemException object, a NULL pointer is returned.

**Note** The reference count for the Exception object is not incremented by this method.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>exc</td>
<td>An Exception pointer to be narrowed.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Exception name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNKNOWN</td>
<td>Unknown exception.</td>
</tr>
<tr>
<td>BAD_PARAM</td>
<td>An invalid parameter was passed.</td>
</tr>
<tr>
<td>NO_MEMORY</td>
<td>Dynamic memory allocation failure.</td>
</tr>
<tr>
<td>IMP_LIMIT</td>
<td>Implementation limit violated.</td>
</tr>
<tr>
<td>COMM_FAILURE</td>
<td>Communication failure.</td>
</tr>
<tr>
<td>INV_OBJREF</td>
<td>Invalid object reference specified.</td>
</tr>
<tr>
<td>NO_PERMISSION</td>
<td>No permission for attempted operation.</td>
</tr>
<tr>
<td>INTERNAL</td>
<td>ORB internal error.</td>
</tr>
<tr>
<td>MARSHAL</td>
<td>Error marshalling parameter or result.</td>
</tr>
<tr>
<td>INITIALIZE</td>
<td>ORB initialization failure.</td>
</tr>
<tr>
<td>NO_IMPLEMENT</td>
<td>Operation implementation not available.</td>
</tr>
<tr>
<td>BAD_TYPECODE</td>
<td>Invalid typecode.</td>
</tr>
<tr>
<td>BAD_OPERATION</td>
<td>Invalid operation.</td>
</tr>
<tr>
<td>NO_RESOURCES</td>
<td>Insufficient resources to process request.</td>
</tr>
<tr>
<td>NO_RESPONSE</td>
<td>Response to request not yet available.</td>
</tr>
<tr>
<td>PERSIST_STORE</td>
<td>Persistent storage failure.</td>
</tr>
<tr>
<td>BAD_INV_ORDER</td>
<td>Routine invocations out of order.</td>
</tr>
<tr>
<td>TRANSIENT</td>
<td>Transient failure.</td>
</tr>
<tr>
<td>FREE_MEM</td>
<td>Unable to free memory.</td>
</tr>
<tr>
<td>INV_INDENT</td>
<td>Invalid identifier syntax.</td>
</tr>
<tr>
<td>INV_FLAG</td>
<td>Invalid flag was specified.</td>
</tr>
<tr>
<td>INTF_REPOS</td>
<td>Error accessing interface repository.</td>
</tr>
<tr>
<td>BAD_CONTEXT</td>
<td>Error processing context object.</td>
</tr>
</tbody>
</table>
UserException

The UserException base class is used to derive the user exceptions that your object implementations may want to raise. This class is derived from the Exception class, described on page 5-11, which provides methods for printing the name and details of the exception to an output stream.

Include file

The corba.h file should be included when you use this class.

UserException methods

CORBA::UserException();

This method creates a UserException object with the specified properties.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>minor</td>
<td>The minor code.</td>
</tr>
<tr>
<td>status</td>
<td>The completion status, one of the following values:</td>
</tr>
<tr>
<td></td>
<td>COMPLETED_YES</td>
</tr>
<tr>
<td></td>
<td>COMPLETED_NO</td>
</tr>
<tr>
<td></td>
<td>COMPLETED_MAYBE</td>
</tr>
</tbody>
</table>
This chapter describes the classes that support the Dynamic Invocation Interface used by client applications, and the Dynamic Skeleton Interface used by object servers. This chapter includes the following sections:

<table>
<thead>
<tr>
<th>Class</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Any</td>
<td>6-2</td>
</tr>
<tr>
<td>ContextList</td>
<td>6-4</td>
</tr>
<tr>
<td>DynamicImplementation</td>
<td>6-6</td>
</tr>
<tr>
<td>DynAny</td>
<td>6-7</td>
</tr>
<tr>
<td>DynArray</td>
<td>6-10</td>
</tr>
<tr>
<td>DynEnum</td>
<td>6-11</td>
</tr>
<tr>
<td>DynSequence</td>
<td>6-13</td>
</tr>
<tr>
<td>DynStruct</td>
<td>6-13</td>
</tr>
<tr>
<td>DynUnion</td>
<td>6-14</td>
</tr>
<tr>
<td>Environment</td>
<td>6-15</td>
</tr>
<tr>
<td>ExceptionList</td>
<td>6-17</td>
</tr>
<tr>
<td>NamedValue</td>
<td>6-19</td>
</tr>
<tr>
<td>NVList</td>
<td>6-20</td>
</tr>
<tr>
<td>Request</td>
<td>6-23</td>
</tr>
<tr>
<td>ServerRequest</td>
<td>6-27</td>
</tr>
<tr>
<td>TCKind</td>
<td>6-29</td>
</tr>
<tr>
<td>TypeCode</td>
<td>6-30</td>
</tr>
</tbody>
</table>
class CORBA::Any

This class is used to represent an IDL type so that its value may be passed in a type-safe manner. Objects of this class have a pointer to a TypeCode that defines the object’s type and a pointer to the value associated with the object. Methods are provided to construct, copy, and destroy an object as well as to initialize and query the object’s type and value. In addition, streaming operators are provided to read and write the object to a stream.

Include file

The any.h file should be included when you use this structure.

Any methods

CORBA::Any();

This is the default constructor that creates an empty Any object.

CORBA::Any(const CORBA::Any& val);

This is a copy constructor that creates an Any object that is a copy of the specified target.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>val</td>
<td>The object to be copied.</td>
</tr>
</tbody>
</table>

CORBA::Any(CORBA::TypeCode_ptr tc, void *value, CORBA::Boolean release = 0);

This constructor creates an Any object initialized with the specified value and TypeCode.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tc</td>
<td>The TypeCode of the value contained by this Any.</td>
</tr>
<tr>
<td>value</td>
<td>The value contained by this Any.</td>
</tr>
<tr>
<td>release</td>
<td>If set to TRUE, the memory associated with this Any object’s value will be released when this Any object is destroyed.</td>
</tr>
</tbody>
</table>

static CORBA::Any_ptr _duplicate(CORBA::Any_ptr ptr);

This static method increments the reference count for the specified object and then returns a pointer to it.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ptr</td>
<td>The Any to be duplicated.</td>
</tr>
</tbody>
</table>
static CORBA::Any_ptr _nil();

This static method returns a NULL pointer that can be used for initialization purposes.

static void _release(CORBA::Any_ptr *ptr);

This static method decrements the reference count for the specified object. If the count has reached zero, all memory managed by the object is released and the object is deleted.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ptr</td>
<td>The Any to be released.</td>
</tr>
</tbody>
</table>

**Initialization operators**

void operator<<(CORBA::Short);
void operator<<(CORBA::UShort);
void operator<<(CORBA::Long);
void operator<<(CORBA::ULong);
void operator<<(CORBA::Float);
void operator<<(CORBA::Double);
void operator<<(const CORBA::Any&);
void operator<<(const char *);
void operator<<(CORBA::LongLong);
void operator<<(CORBA::ULongLong);
void operator<<(CORBA::LongDouble);

These operators will initialize this object with the specified value, automatically setting the appropriate TypeCode for the value. If this Any object was constructed with the release flag set to true, the value previously stored in this Any object will be released before the new value is assigned.

void operator<<(CORBA::TypeCode_ptr tc);

This method initializes this object with the specified TypeCode of the value.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tc</td>
<td>The TypeCode to set for this Any.</td>
</tr>
</tbody>
</table>

**Extraction operators**

CORBA::Boolean operator>>(CORBA::Short&) const;
CORBA::Boolean operator>>(CORBA::UShort&) const;
CORBA::Boolean operator>>(CORBA::Long&) const;
CORBA::Boolean operator>>(CORBA::ULong&) const;
CORBA::Boolean operator>>(CORBA::Float&) const;
CORBA::Boolean operator>>(CORBA::Double&) const;
CORBA::Boolean operator>>(CORBA::Any&) const;
CORBA::Boolean operator>>(char *&) const;
CORBA::Boolean operator>>(CORBA::LongLong&) const;
CORBA::Boolean operator>>(CORBA::ULongLong&) const;
CORBA::Boolean operator>>(CORBA::LongDouble&) const;

These operators store the value from this object into the specified target. If the TypeCode of the target does not match the TypeCode of the stored value, false is returned and no value is extracted. Otherwise, the stored value will be assigned to the target and true will be returned.

CORBA::Boolean operator>>(CORBA::TypeCode_ptr& tc) const;

This method extracts the TypeCode of the value stored in this object.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tc</td>
<td>The object where the TypeCode for this Any is to be stored</td>
</tr>
</tbody>
</table>

### ContextList

class CORBA::ContextList

This class contains a list of contexts that may be associated with an operation request. See “Request” on page 6-23.

### ContextList methods

CORBA::ContextList();

This method constructs an empty Context list.

~CORBA::ContextList();

This method is the default destructor.

void add(const char *ctx);

This method adds the specified context to this object’s list.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tc</td>
<td>The context to be added to the list.</td>
</tr>
</tbody>
</table>

void add_consume(char *ctx);

This method adds the specified exception type code to this object’s list. Ownership of the passed argument is assumed by this ContextList. You should not attempt to access or free the argument after invoking this method.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tc</td>
<td>The type code to be added to the list.</td>
</tr>
</tbody>
</table>
ContextList methods

CORBA::ULong **count**() const;

This method returns the number of items currently stored in the list.

const char **item**(CORBA::Long **index**);

This method returns a pointer to the context that is stored in the list at the specified index. If the index is invalid, a NULL pointer is returned. You should not attempt to access or free the argument after invoking this method. To remove a context from the list, use the remove method.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>index</td>
<td>The index of the context to be returned. The index is zero-based.</td>
</tr>
</tbody>
</table>

void **remove**(CORBA::long **index**);

This method removes from the list the context with the specified index. If the index is invalid, no removal will occur.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>index</td>
<td>The index of the context to be removed. The index is zero-based.</td>
</tr>
</tbody>
</table>

static CORBA::ContextList_ptr **_duplicate**(CORBA::ContextList_ptr **ptr**);

This static method increments the reference count for the object and then returns a pointer to it.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ptr</td>
<td>The object to be duplicated.</td>
</tr>
</tbody>
</table>

static CORBA::ContextList_ptr **_nil**();

This static method returns a NULL pointer that can be used for initialization purposes.

static void **_release**(CORBA::ContextList **ptr**);

This static method decrements the reference count for this object. If the count has reached zero, all memory managed by the object is released and the object is deleted.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ptr</td>
<td>The Any to be released.</td>
</tr>
</tbody>
</table>
DynamicImplementation

class CORBA::DynamicImplementation : public CORBA::Object

This base class is used derive object implementations that wish to use the Dynamic Skeleton Interface instead of a skeleton class generated by the IDL compiler. You must provide an implementation of the invoke method when deriving from this class.

DynamicImplementation methods

CORBA::DynamicImplementation(const char *interface_name,
const char *object_name = NULL,
const char *repository_id = NULL);

This method constructs an object that implements a single interface with the specified interface name, object name, and repository identifier.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>interface_name</td>
<td>The interface name.</td>
</tr>
<tr>
<td>object_name</td>
<td>The object name.</td>
</tr>
<tr>
<td>repository_id</td>
<td>The repository identifier of this object.</td>
</tr>
</tbody>
</table>

CORBA::DynamicImplementation(const InterfaceNameSequence& interfaces,
const RepositoryIdSequence& ids,
const char *object_name = NULL);

This method constructs an object which implements more than one interface.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>interfaces</td>
<td>An array of interface names.</td>
</tr>
<tr>
<td>ids</td>
<td>An array of repository identifiers.</td>
</tr>
<tr>
<td>object_name</td>
<td>The object name.</td>
</tr>
</tbody>
</table>

virtual void invoke(CORBA::ServerRequest_ptr request) = 0;

This method will be invoked by the BOA whenever client operation requests are received for your object implementation. You must provide an implementation of this method which validates the ServerRequest object’s contents, performs the necessary processing to fulfill the request, and returns the results to the client. For more information on the ServerRequest class, see page 6-27.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>request</td>
<td>The ServerRequest object that represents the client’s operation request.</td>
</tr>
</tbody>
</table>
DynAny

static CORBA::DynamicImplementation_ptr _duplicate(CORBAsDynamicImplementation_ptr ptr);

This static method increments the reference count of the specified object and then returns a pointer to it.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ptr</td>
<td>The object to be duplicated.</td>
</tr>
</tbody>
</table>

static CORBA::DynamicImplementation_ptr _nil();

This static method returns a NULL pointer that can be used for initialization purposes.

static void _release(CORBAsDynamicImplementation_ptr ptr);

This static method decrements the reference count for the specified object. If the count has reached zero, all memory managed by the object is released and the object is deleted.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ptr</td>
<td>The object to be duplicated.</td>
</tr>
</tbody>
</table>

void * _safe_narrow(const CORBA::TypeInfo& obj) const;

Narrows a pointer to this object to this type specified by obj. If this object cannot be narrowed to the desired type, a NULL pointer is returned.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>obj</td>
<td>A TypeInfo object representing the desired type.</td>
</tr>
</tbody>
</table>

c const CORBA::TypeInfo _type_info() const;

This static method returns type information for this object.

DynAny

class CORBA::DynAny : public CORBA::Object

A DynAny object is used by a client application or server to create and interpret data types at run-time which were not defined at compile-time. A DynAny may contain a basic type (such as a boolean, int, or float) or a complex type (such as a struct or union). The type contained by a DynAny is defined when it is created and may not be changed during the lifetime of the object.

A DynAny object may represent a data type as one or more components, each with its own value. The next, seek, rewind, and current_component methods are provided to help you navigate through the components.
DynAny objects for basic types are created using the ORB::create_basic_dyn_any method, described in Chapter 5. A DynAny object may also be created and initialized from an Any object using the ORB::create_dyn_any method, also described in Chapter 5, “Core interfaces and classes.”

The following interfaces are derived from DynAny and provide support for constructed types that are dynamically managed.

### Important usage restrictions

DynAny objects cannot be used as parameters on operation requests or DII requests, nor can they be externalized using the ORB::object_to_string method. However, you may use the DynAny::to_any method to convert a DynAny object to an Any, which can be used as a parameter.

### DynAny methods

**void assign(CORBA::DynAny_ptr dyn_any);**

Initializes the value in this object from the specified DynAny.

An Invalid exception is raised if the type contained in the Any does not match the type contained by this object.

**CORBA::DynAny_ptr copy();**

Returns a copy of this object.

**virtual void destroy();**

Destroys this object.

**virtual CORBA::DynAny_ptr current_component();**

Returns the current component in this object.

**virtual void from_any(CORBA::Any& value);**

Initializes the current component of this object from the specified Any object.

An Invalid exception is raised if the TypeCode of value contained in the Any does not match the TypeCode defined for this object when it was created.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>An Any object containing the value to set for this object.</td>
</tr>
</tbody>
</table>
virtual boolean next();

Advances to the next component, if one exists, and returns true. If there are no more components, false is returned.

virtual void rewind();

Returns to the first component contained in this object’s sequence. A subsequent invocation of the current_component method will return the first component in the sequence.

If this object contains only one component, invoking this method will have no effect.

virtual CORBA::Boolean seek(CORBA::Long index);

If this object contains multiple components, this method advances to the component with the specified index and returns true. A subsequent invocation of the current_component method will return the component with the specified index.

If there is no component at the specified index, false is returned.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>index</td>
<td>The zero-base index of the desired component.</td>
</tr>
</tbody>
</table>

virtual CORBA::Any* to_any();

Returns an Any object containing the value of the DynAny.

CORBA::TypeCode_ptr type();

Returns the TypeCode for the value stored in the DynAny.

**Extraction methods**

A set of methods is provided which return the type contained in this DynAny object’s current component. Code sample 6.1 shows the name of each of the extraction methods.

A TypeMismatch exception is raised if the value contained in this DynAny does not match the expected return type for the extraction method used.

**Code sample 6.1**  Extraction methods offered by the DynAny class

```cpp
virtual CORBA::Any* get_any();
virtual CORBA::Boolean get_boolean();
virtual CORBA::Char get_char();
virtual CORBA::Double get_double();
virtual CORBA::Float get_float();
virtual CORBA::Long get_long();
virtual CORBA::LongDouble get_longdouble();
virtual CORBA::Long get_longlong();
virtual CORBA::Octet get_octet();
virtual CORBA::Object_ptr get_reference();
```
Insertion methods

A set of methods is provided that copies a particular type of value to this DynAny object’s current component. Code sample 6.2 shows the list of methods provided for inserting various types.

These methods will raise an InvalidValue exception if the inserted object’s type does not match the DynAny object’s type.

Code sample 6.2 Insertion methods offered by the DynAny class

```c++
virtual void insert_any(const CORBA::Any& value);
virtual void insert_boolean(CORBA::Boolean value);
virtual void insert_char(CORBA::Char value);
virtual void insert_double(CORBA::Double value);
virtual void insert_float(CORBA::Float value);
virtual void insert_long(CORBA::Long value);
virtual void insert_longdouble(CORBA::LongDouble value);
virtual void insert_longlong(CORBA::LongLong value);
virtual void insert_octet(CORBA::Octet value);
virtual void insert_reference(CORBA::Object_ptr value);
virtual void insert_short(CORBA::Short value);
virtual void insert_string(const char* value);
virtual void insert_typecode(CORBA::TypeCode_ptr value);
virtual void insert_ulong(CORBA::ULong value);
virtual void insert_ulonglong(CORBA::ULongLong value);
virtual void insert_ushort(CORBA::UShort value);
virtual void insert_wchar(CORBA::WChar value);
virtual void insert_wstring(const CORBA::WChar* value);
```
The **VISDynComplex** class is a helper class that allows the ORB to manage complex **DynAny** types.

### Important usage restrictions

**DynArray** objects cannot be used as parameters on operation requests or DII requests, nor can they be externalized using the `ORB::object_to_string` method. However, you may use the **DynAny::to_any** method to convert a **DynArray** object to a sequence of **Any** objects, which can be used as a parameter.

### DynArray methods

```cpp
virtual void destroy();

Destroys this object.
```

```cpp
CORBA::AnySeq* get_elements();

Returns a sequence of **Any** objects containing the values stored in this object.
```

```cpp
void set_elements(CORBA::AnySeq& _value);

Sets the elements contained in this object from the specified sequence of **Any** objects.

An **InvalidSeq** exception will be raised if the number of elements in **_value** is not equal to the number of elements in this **DynArray** or if the types of the **Any** values do not match the **TypeCode** of the **DynAny**.
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>_value</td>
<td>An array of <strong>Any</strong> objects whose values will be set in this <strong>DynArray</strong>.</td>
</tr>
</tbody>
</table>

### DynEnum

**class** **CORBA::DynEnum** : **CORBA::DynAny**

Objects of this class are used by a client application or server to create and interpret enumeration values at run-time which were not defined at compile-time.

Since this type contains a single component, invoking the **DynAny::rewind** and **DynAny::next** methods on a **DynEnum** object will always return **FALSE**.

### Important usage restrictions

**DynEnum** objects cannot be used as parameters on operation requests or DII requests, nor can they be externalized using the `ORB::object_to_string` method. However, you may use the **to_any** method to convert a **DynEnum** object to an **Any**, which can be used as a parameter.
DynEnum methods

**void from_any(const CORBA::Any& value);**

Initializes the value of this object from the specified Any object.

An **Invalid** exception is raised if the **TypeCode** of value contained in the Any does not match the **TypeCode** defined for this object when it was create.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>An Any object containing the value to set for this object.</td>
</tr>
</tbody>
</table>

**CORBA::ULong get_ulong();**

Returns the DynEnum object’s value as an unsigned long.

**void insert_ulong(CORBA::ULong _val);**

Initializes the current component of this object from the specified unsigned long.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>_val</td>
<td>The unsigned long value to set for this object.</td>
</tr>
</tbody>
</table>

**CORBA::Any* to_any();**

Returns an Any object containing the value of the current component.

**char* value_as_string();**

Returns the DynEnum object’s value as a string.

**void value_as_string(const char* value_as_string);**

Sets the value contained in this DynEnum from the specified string.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>value_as_string</td>
<td>A string that will be used to set the value in this DynEnum.</td>
</tr>
</tbody>
</table>

**CORBA::ULong value_as_ulong();**

Returns an int containing the DynEnum object’s value.

**void value_as_ulong(CORBA::ULong value_as_ulong);**

Sets the value contained in this DynEnum from the specified CORBA::ULong.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>value_as_ulong</td>
<td>An integer that will be used to set the value in this DynEnum.</td>
</tr>
</tbody>
</table>
DynSequence

**class** CORBA::DynSequence : public CORBA::DynArray

Objects of this class are used by a client application or server to create and interpret array data types at run-time which were not defined at compile-time. A DynSequence may contain a sequence of basic type (such as a boolean, int, or float) or a constructed type (such as a struct or union). The type contained by a DynSequence is defined when it is created and may not be changed during the lifetime of the object.

The next, rewind, seek, and current_component methods may be used to navigate through the components.

**Important usage restrictions**

DynSequence objects cannot be used as parameters on operation requests or DII requests, nor can they be externalized using the ORB::object_to_string method. However, you may use the to_any method to convert a DynSequence object to a sequence of Any objects, which can be used as a parameter.

**DynSequence methods**

- **CORBA::ULong length()**
  
  Returns the number of components contained in this DynSequence.

- **void length(CORBA::ULong length)**
  
  Sets the number of components contained in this DynSequence.

  If you specify a length that is less than the current number of components, the sequence will be truncated.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>length</td>
<td>The number of components to be contained in this DynSequence.</td>
</tr>
</tbody>
</table>

DynStruct

**class** CORBA::DynStruct : VISDynComplex

Objects of this class are used by a client application or server to create and interpret structures at run-time which were not defined at compile-time.

The next, rewind, seek, and current_component methods may be used to navigate through the structure members.

You create an DynStruct object by invoking the ORB::create_dyn_struct method.
Important usage restrictions

DynStruct objects cannot be used as parameters on operation requests or DII requests, nor can they be externalized using the CORBA::object_to_string method. However, you may use the to_any method to convert a DynStruct object to an Any object, which can be used as a parameter.

DynStruct methods

void destroy();

Destroys this object.

CORBA::FieldName current_member_name();

Returns the member name of the current component.

CORBA::TCKind current_member_kind();

Returns the TypeCode associated with the current component.

CORBA::NameValuePairSeq get_members();

Returns the members of the structure as a sequence of NameValuePair objects.

void set_members(const CORBA::NameValuePairSeq& value);

Sets the structure members from the array of NameValuePair objects.

An InvalidSeq exception will be raised if the order of the elements in value is not identical to the order of the members in this DynStruct or if the length of the value sequence is not equal to the number of members in the DynStruct.

DynUnion

class CORBA::DynUnion : VISDynComplex

This interface is used by a client application or server to create and interpret unions at run-time which were not defined at compile-time. The DynUnion contains a sequence of two elements; the union discriminator and the actual member.

The next, rewind, seek, and current_component methods may be used to navigate through the components.

You create an DynUnion object by invoking the CORBA::create_dyn_union method.

Important usage restrictions

DynUnion objects cannot be used as parameters on operation requests or DII requests, nor can they be externalized using the CORBA::object_to_string method.
However, you may use the \texttt{DynAny::to\_any} method to convert a \texttt{DynUnion} object to an \texttt{Any} objects, which can be used as a parameter.

### DynUnion methods

- \texttt{CORBA::DynAny\_ptr\ discriminator()};
  - Returns a \texttt{DynAny} object containing the discriminator for the union.

- \texttt{CORBA::TCKind\ discriminator\_kind()};
  - Returns the type code of the discriminator for the union.

- \texttt{CORBA::DynAny\_ptr\ member()};
  - Returns a \texttt{DynAny} object for the current component which represents a member in the union.

- \texttt{CORBA::TCKind\ member\_kind()};
  - Returns the type code for the current component, which represents a member in the union.

- \texttt{CORBA::FieldName\ member\_name()};
  - Returns the member name of the current component.

- \texttt{void\ member\_name(const\ char*\ _val)};
  - Sets the member name of the current component to the specified name.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>_val</td>
<td>The member name to set for the current component.</td>
</tr>
</tbody>
</table>

- \texttt{CORBA::Boolean\ set\_as\_default()};
  - Returns true if the current component is the default member.

- \texttt{void\ set\_as\_default(CORBA::Boolean\ _val)};
  - Enables or disables the current component as the default member, based on the value of the \texttt{set\_as\_default} parameter.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>_val</td>
<td>If set to true, the current component will become the default member.</td>
</tr>
</tbody>
</table>

### Environment

#### class \texttt{CORBA::Environment}

The \texttt{Environment} class is used for reporting and accessing both system and user exceptions on platforms where C++ language exceptions are not supported. When an interface specifies that user exceptions may be raised by the object’s
methods, the Environment class becomes an explicit parameter of that method. If an interface does not raise any exceptions, the Environment class is an implicit parameter and is only used for reporting system exceptions. If an Environment object is not passed from the client to a stub, the default of per-object Environment is used.

Multithreaded applications have a global Environment object for each thread that is created. Applications that are not multithreaded have just one global Environment object.

Include file

You should include the corba.h file when you use this class.

Environment methods

CORBA::Status ORB::create_environment(CORBA::Environment_ptr& ptr);

This method can be used to create a new Environment object.

This method is provided for CORBA compliance. You may find it easier to use the constructor provided for this class or the C++ new operator.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ptr</td>
<td>The pointer will be set to point to the newly created object.</td>
</tr>
</tbody>
</table>

Environment();

This method creates an Environment object. This is equivalent to calling the ORB::create_environment method.

~Environment();

This method deletes this Environment object. This is equivalent to calling the CORBA::release method and passing a pointer to the object to be deleted.

static CORBA::Environment& CORBA::current_environment();

This static method returns a reference to the global Environment object for the application process. In multithreaded applications, it returns the global Environment object for this thread.

void exception(CORBA::Exception *exp);

This method records the Exception object passed as an argument. The Exception object must be dynamically allocated because the specified object will assume ownership of the Exception object and will delete it when the
Environment itself is deleted. Passing a NULL pointer to this method is equivalent to invoking the clear method on the Environment.

```
CORBA::Exception *exception() const;
```

This method returns a pointer to the Exception currently recorded in this Environment. You must not invoke delete on the Exception pointer returned by this call. If no Exception has been recorded, a NULL pointer will be returned.

```
void clear();
```

This method will cause this Environment to delete any Exception object that it holds. If this object holds no exception, this method has no effect.

```
static CORBA::Environment_ptr _duplicate(CORBA::Environment_ptr ptr);
```

This static method increments the reference count for the specified object and then returns a pointer to it.

```
static CORBA::Environment_ptr _nil();
```

This static method returns a NULL pointer that can be used for initialization purposes.

**ExceptionList**

class CORBA::ExceptionList

This class contains a list of type codes that represent exceptions that may be raised by an operation request. See “Request” on page 6-23.

**ExceptionList methods**

**CORBA::ExceptionList();**

This method constructs an empty exception list.

**CORBA::ExceptionList(CORBA::ExceptionList& list);**

This is a copy constructor.

```
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>list</td>
<td>The list to be copied.</td>
</tr>
</tbody>
</table>
```
### ExceptionList methods

- **~CORBA::ExceptionList();**
  
  This method is the default destructor.

- **void add(CORBA::TypeCode_ptr tc);**
  
  This method adds the specified exception type code to this object’s list.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tc</td>
<td>The type code of an exception to be added to the list.</td>
</tr>
</tbody>
</table>

- **void add_consume(CORBA::TypeCode_ptr tc);**
  
  This method adds the specified exception type code to this object’s list. Ownership of the passed argument is assumed by this `ExceptionList`. You should not attempt to access or free the argument after invoking this method.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tc</td>
<td>The type code of an exception to be added to the list.</td>
</tr>
</tbody>
</table>

- **CORBA::ULong count() const;**
  
  This method returns the number of items currently stored in the list.

- **CORBA::TypeCode_ptr item(CORBA::Long index);**
  
  This method returns a pointer to the `TypeCode` stored in the list at the specified index. If the index is invalid, a `NULL` pointer is returned. You should not attempt to access or free the argument after invoking this method. To remove a `TypeCode` from the list, use the `remove` method.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>index</td>
<td>The index of the type code to be returned. The index is zero-based.</td>
</tr>
</tbody>
</table>

- **void remove(CORBA::long index);**
  
  This method removes from the list, the `TypeCode` with the specified index. If the index is invalid, no removal will occur.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>index</td>
<td>The index of the type code to be removed. The index is zero-based.</td>
</tr>
</tbody>
</table>

- **static CORBA::ExceptionList_ptr _duplicate(CORBA::ExceptionList_ptr ptr);**
  
  This static method increments the reference count for the specified object and then returns a pointer to it.
static CORBA::ExceptionList_ptr _nil();

This static method returns a NULL pointer that can be used for initialization purposes.

static void _release(CORBA::ExceptionList *ptr);

This static method decrements the reference count for the specified object. If the count has reached zero, all memory managed by the object is released and the object is deleted.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ptr</td>
<td>The object to be released.</td>
</tr>
</tbody>
</table>

NamedValue

class CORBA::NamedValue

The NamedValue class is used to represent a name-value pair used as a parameter or return value in a Dynamic Invocation Interface request. Objects of this class are grouped into an NVList, described on page 6-20. The Any class is used to represent the value associated with this object. The Request class is described on page 6-23.

Include file

You should include the file corba.h when using this class.

NamedValue methods

CORBA::Flags flags() const;

This method returns the flag defining how this name-value pair is to be used. One of the following is returned.

- **ARG_IN**—This object represents an input parameter.
- **ARG_OUT**—This object represents an output parameter.
- **ARG_INOUT**—This object represents both an input and output parameter.
- **IN_COPY_VALUE**—This value can be specified in combination with the ARG_INOUT flag to specify that the ORB should make a copy of the parameter. This allows the ORB to release memory associated with this parameter without impacting the client application’s memory.
const char *name() const;

This method returns the name portion of this object’s name-value pair. You should never release the storage pointed to by the return argument.

CORBA::Any *value() const;

This method returns the value portion of this object’s name-value pair. You should never release the storage pointed to by the return argument.

static CORBA::NamedValue_ptr _duplicate(CORBA::NamedValue_ptr ptr);

This static method increments the reference count for the specified object and then returns a pointer to it.

static CORBA::NamedValue_ptr _nil();

This static method returns a NULL pointer that can be used for initialization purposes.

static void _release(CORBA::NamedValue *ptr);

This static method decrements the reference count for the specified object. If the count has reached zero, all memory managed by the object is released and the object is deleted.

NVList

class CORBA::NVList

The NVList class is used to contain a list of NamedValue objects, described on page 6-19 in this guide, and is used to pass parameters associated with a Dynamic Invocation Interface request. The Request class is described on page 6-23.

Several methods are provided for adding items to the list. You should never release the storage pointed to by the return argument. Always use the remove method to delete an item from the list.

Include file

You should include the file corba.h when using this class.
NVList methods

CORBA::NamedValue_ptr add(CORBA::Flags flags);

This method adds a NamedValue object to this list, initializing only the flags. Neither the name or value of the added object are initialized. A pointer is returned which can be used to initialize the name and value attributes of the NamedValue. You should never release the storage associated with the return argument.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>flags</td>
<td>The flag indicating the intended use of the NamedValue object. It can be one of ARG_IN, ARG_OUT, or ARG_INOUT.</td>
</tr>
</tbody>
</table>

CORBA::NamedValue_ptr add_item(const char *name, CORBA::Flags flags);

This method adds a NamedValue object to this list, initializing the object’s flags and name attributes. A pointer is returned which can be used to initialize the value attribute of the NamedValue. You should never release the storage associated with the return argument.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>The name.</td>
</tr>
<tr>
<td>flags</td>
<td>The flag indicating the intended use of the NamedValue object. It can be one of ARG_IN, ARG_OUT, or ARG_INOUT.</td>
</tr>
</tbody>
</table>

NamedValue_ptr add_item_consume(char *nm, CORBA::Flags flags);

This method is the same as the add_item method, except that the NVList takes over the management of the storage pointed to by nm. You will not be able to access nm after this method is called because the list may have copied and released it. When this item is removed, the storage associated with it is automatically freed.

**Caution** You should never release the memory associated with this method’s return value.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>The name.</td>
</tr>
<tr>
<td>flags</td>
<td>The flag indicating the intended use of the NamedValue object. It must be one of ARG_IN, ARG_OUT, or ARG_INOUT.</td>
</tr>
</tbody>
</table>
**NVLList methods**

```c
CORBA::NamedValue_ptr add_value(const char *name,
     const CORBA::Any *value,
     CORBA::Flags flags);
```

This method adds a NamedValue object to this list, initializing the name, value, and flags. A pointer to the NamedValue object is returned. You should never release the storage associated with the return argument.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>The name.</td>
</tr>
<tr>
<td>value</td>
<td>The value.</td>
</tr>
<tr>
<td>flags</td>
<td>The flag indicating the intended use of the NamedValue object. It can be one of <code>ARG_IN</code>, <code>ARG_OUT</code>, or <code>ARG_INOUT</code>.</td>
</tr>
</tbody>
</table>

```c
NamedValue_ptr add_value_consume(char *nm, CORBA::Any *value,
                  CORBA::Flags flags);
```

This method is the same as the `add_value` method, except that the NVList takes over the management of the storage pointed to by `nm` and `value`. You will not be able to access `nm` or `value` after this method is called because the list may have copied and released them. When this item is removed, the storage associated with it is automatically freed.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nm</td>
<td>The name.</td>
</tr>
<tr>
<td>value</td>
<td>The value.</td>
</tr>
<tr>
<td>flags</td>
<td>The flag indicating the intended use of the NamedValue object. It must be one of <code>ARG_IN</code>, <code>ARG_OUT</code>, or <code>ARG_INOUT</code>.</td>
</tr>
</tbody>
</table>

```c
CORBA::Long count() const;
```

This method returns the number of NamedValue objects in this list.

```c
static CORBA::Boolean CORBA::is_nil(NVList_ptr obj);
```

This method returns true if the specified NamedValue pointer is NULL.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>obj</td>
<td>The object pointer to be checked.</td>
</tr>
</tbody>
</table>

```c
NamedValue_ptr item(CORBA::Long index);
```

This method returns the NamedValue in the list with the specified index. Never release the storage associated with the return argument.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>index</td>
<td>The index of the desired NamedValue object. Note that indexing is zero-based.</td>
</tr>
</tbody>
</table>
static void CORBA::release(CORBA::NVList_ptr obj);

This static method releases the specified object.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
</table>
| obj       | The object to be released.

Status remove(CORBA::Long index);

This method deletes the NamedValue object from this list, located at the specified index. Storage associated with items in the list that were added using the add_item_consume or add_value_consume methods will be released before the item is removed.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
</table>
| index     | The index of the NamedValue object. Note that indexing is zero-based.

static CORBA::NVList_ptr _duplicate(CORBA::NVList_ptr ptr);

This static method increments the reference count for the specified object and then returns a pointer to it.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
</table>
| ptr       | The object to be duplicated.

static CORBA::NVList_ptr _nil();

This static method returns a NULL pointer that can be used for initialization purposes.

static void _release(CORBA::NVList *ptr);

This static method decrements the reference count for the specified object. If the count has reached zero, all memory managed by the object is released and the object is deleted.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
</table>
| ptr       | The object to be released.

### Request

The **Request** class is used by client applications to invoke an operation on an ORB object using the Dynamic Invocation Interface. A single ORB object is associated with a given Request object. The Request represents an operation that is to be performed on the ORB object. It includes the arguments to be passed, the Context, and an Environment object, if any. Methods are provided for invoking...
the request, receiving the response from the object implementation, and retrieving the result of the operation.

You can create a Request object by using the Object::_create_request, described on page 5-12, and Object::_request, described on page 5-7, for creating a Request object.

Note that a Request object will retain ownership of all return parameters, so you should never attempt to free them. This behavior can be overridden by using the -ORBbackcomp option, described in Appendix A, “Using command-line options.”

Include file

The corba.h file should be included when you use this class.

Request methods

CORBA::Any& add_in_arg();

This method adds an unnamed input argument to this Request and returns a reference to the Any object so that you can set its name, type, and value.

CORBA::Any& add_in_arg(const char *name);

This method adds a named input argument to this Request and returns a reference to the Any object so that you can set its type and value.

Caution

You should never release the memory associated with this method’s return value.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>The name of the input argument to be added.</td>
</tr>
</tbody>
</table>

CORBA::Any& add_inout_arg();

This method adds an unnamed inout argument to this Request and returns a reference to the Any object so that you can set its name, type, and value.

CORBA::Any& add_inout_arg(const char *name);

This method adds a named inout argument to this Request and returns a reference to the Any object so that you can set its type and value.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>The name of the inout argument to be added.</td>
</tr>
</tbody>
</table>

CORBA::Any& add_out_arg();

This method adds an unnamed output argument to this Request and returns a reference to the Any object so that you can set its name, type, and value.
CORBA::Any& add_out_arg(const char *name);

This method adds a named output argument to this Request and returns a reference to the Any object so that you can set its type, and value.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>The name of the output argument to be added.</td>
</tr>
</tbody>
</table>

CORBA::NVList_ptr arguments();

This method returns a pointer to an NVList object containing the arguments for this request. The pointer can be used to set or retrieve the argument values. For more information on NVList, see page 6-20.

**Caution** You should never release the memory associated with this method’s return value.

CORBA::ContextList_ptr contexts();

This method returns a pointer to a list of all the Context objects that are associated with this Request. For more information on the Context class, see page 5-9.

**Caution** You should never release the memory associated with this method’s return value.

CORBA::Context_ptr ctx() const;

This method returns a pointer to the Context associated with this request.

void ctx(CORBA::Context_ptr ctx);

This method sets the Context to be used with this request. For more information on the Context class, see page 5-9.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ctx</td>
<td>The Context object to be associated with this request.</td>
</tr>
</tbody>
</table>

CORBA::Environment_ptr env();

This method returns a pointer to the Environment associated with this request. For more information on the Environment class, see page 6-15.

CORBA::ExceptionList_ptr exceptions();

This method returns a pointer to a list of all the exceptions that this request may raise.

**Caution** You should never release the memory associated with this method’s return value.

CORBA::Status get_response();

This method is used after the send_deferred method has been invoked to retrieve a response from the object implementation. If there is no response...
available, this method blocks the client application until a response is received.

```c++
CORBA::Status invoke();
```

This method invokes this `Request` on the ORB object associated with this request. This method will block the client until a response is received from the object implementation. This `Request` should be initialized with the target object, operation name and arguments before this method is invoked.

```c++
const char* operation() const;
```

This method returns the name of the operation that this request will represent.

```c++
CORBA::Boolean poll_response();
```

This non-blocking method is invoked after the `send_deferred` method to determine if a response has been received. This method returns true if a response has been received, otherwise false is returned.

```c++
CORBA::NamedValue_ptr result();
```

This method returns a pointer to a `NamedValue` object where the return value for the operation will be stored. The pointer can be used to retrieve the result value after the request has been processed by the object implementation. For more information on the `NamedValue` class, see page 6-19.

```c++
CORBA::Any& return_value();
```

This method returns a reference to an `Any` object that represents the return value of this `Request` object.

```c++
void set_return_type(CORBA::TypeCode_ptr tc);
```

This method sets the `TypeCode` of the return value that is expected. You must set the return value’s type before using the `invoke` method or one of the `send` methods.

```c++
CORBA::Status send_deferred();
```

Like the `invoke` method, this method sends this `Request` to the object implementation. Unlike the `invoke` method, this method does not block waiting for a response. The client application can retrieve the response using the `get_response` method.

```c++
CORBA::Status send_oneway();
```

This method invokes this `Request` as a `oneway` operation. Oneway operations do not block and do not result in a response being sent from the object implementation to the client application.
ServerRequest

CORBA::Object_ptr target() const;

This method returns a reference to the target object on which this request will operate.

static CORBA::Request_ptr _duplicate(CORBA::Request_ptr ptr);

This static method increments the reference count for the specified object and then returns a pointer to it.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ptr</td>
<td>The object to be duplicated.</td>
</tr>
</tbody>
</table>

static CORBA::Request_ptr _nil();

This static method returns a NULL pointer that can be used for initialization purposes.

static void _release(CORBA::Request_ptr ptr);

This static method decrements the reference count for the specified object. If the count has reached zero, all memory managed by the object is released and the object is deleted.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ptr</td>
<td>The object to be released.</td>
</tr>
</tbody>
</table>

ServerRequest

class CORBA::ServerRequest

The ServerRequest class is used to represent an operation request received by an object implementation that is using the Dynamic Skeleton Interface. When the BOA receives a client operation request, it invokes the object implementation’s invoke method and passes an object of this type.

This class provides the methods needed by the object implementation to determine the operation being requested and the arguments. It also provides methods for setting the return value and reflecting exceptions to the client application.

You should never attempt to free memory associated with any value returned by this class.

Include file

The corba.h file should be included when you use this class.
ServerRequest methods

void arguments(CORBA::NVList_ptr param);

This method obtains the parameter list for this request.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>params</td>
<td>The parameter list to be filled in. You must initialize this list with the appropriate number of Any objects and set their type and flag values prior to invoking this method.</td>
</tr>
</tbody>
</table>

CORBA::Context_ptr ctx()

This method returns the Context object associated with the request.

Caution
You should never release the memory associated with this method’s return value.

void exception(CORBA::Any_ptr exception);

This method is used to reflect the specified exception to the client application.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>exception</td>
<td>The exception that was raised. If this pointer is NULL, a CORBA::UnknownUserException will be reflected.</td>
</tr>
</tbody>
</table>

const char *operation() const;

Returns the name of the operation being requested.

const char* op_name() const

This method returns the operation name associated with the request. The object implementation uses this name to determine if the request is valid, to perform the appropriate processing to fulfill the request, and to return the appropriate value to the client.

void params(CORBA::NVList_ptr params);

This method accepts an NVList object, initialized with the appropriate number of Any objects, and fill it in with the parameters supplied by the client.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>params</td>
<td>The parameter list to be filled in. You must initialize this list with the appropriate number of Any objects and set their type and flag values prior to invoking this method.</td>
</tr>
</tbody>
</table>
void result(CORBA::Any_ptr result);

This method sets the result that is to be reflected to the client application.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>An Any object representing the return value.</td>
</tr>
</tbody>
</table>

void set_exception(const CORBA::Any& a);

This method sets the exception that is to be reflected to the client application.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>An Any object representing the exception.</td>
</tr>
</tbody>
</table>

void set_result(const CORBA::Any& a);

This method sets the result that is to be reflected to the client application.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>An Any object representing the return value.</td>
</tr>
</tbody>
</table>

static CORBA::ServerRequest_ptr _duplicate(CORBA::ServerRequest_ptr ptr);

This static method increments the reference count for the specified object and then returns a pointer to it.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ptr</td>
<td>The object to be duplicated.</td>
</tr>
</tbody>
</table>

static CORBA::ServerRequest_ptr _nil();

This static method returns a NULL pointer that can be used for initialization purposes.

static void _release(CORBA::ServerRequest *ptr);

This static method decrements the reference count for the specified object. If the count has reached zero, all memory managed by the object is released and the object is deleted.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ptr</td>
<td>The object to be released.</td>
</tr>
</tbody>
</table>

enum TCKind

This enumeration describes the various types that a TypeCode object, described on page 6-30, may represent. The values are shown in the following table.
TypeCode

<table>
<thead>
<tr>
<th>Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>tk_null</td>
<td>NULL</td>
</tr>
<tr>
<td>tk_void</td>
<td>void</td>
</tr>
<tr>
<td>tk_short</td>
<td>short</td>
</tr>
<tr>
<td>tk_long</td>
<td>long</td>
</tr>
<tr>
<td>tk_ushort</td>
<td>unsigned short</td>
</tr>
<tr>
<td>tk_ulong</td>
<td>unsigned long</td>
</tr>
<tr>
<td>tk_float</td>
<td>float</td>
</tr>
<tr>
<td>tk_double</td>
<td>double</td>
</tr>
<tr>
<td>tk_boolean</td>
<td>boolean</td>
</tr>
<tr>
<td>tk_char</td>
<td>char</td>
</tr>
<tr>
<td>tk_octet</td>
<td>octet string</td>
</tr>
<tr>
<td>tk_any</td>
<td>Any</td>
</tr>
<tr>
<td>tk_TypeCode</td>
<td>TypeCode</td>
</tr>
<tr>
<td>tk_Principal</td>
<td>Principal</td>
</tr>
<tr>
<td>tk_objref</td>
<td>object reference</td>
</tr>
<tr>
<td>tk_struct</td>
<td>struct</td>
</tr>
<tr>
<td>tk_union</td>
<td>union</td>
</tr>
<tr>
<td>tk_enum</td>
<td>enum</td>
</tr>
<tr>
<td>tk_string</td>
<td>string</td>
</tr>
<tr>
<td>tk_sequence</td>
<td>sequence</td>
</tr>
<tr>
<td>tk_array</td>
<td>array</td>
</tr>
<tr>
<td>tk_alias</td>
<td>alias</td>
</tr>
<tr>
<td>tk_except</td>
<td>exception</td>
</tr>
<tr>
<td>tk_longlong</td>
<td>long long</td>
</tr>
<tr>
<td>tk_ulonglong</td>
<td>unsigned long long</td>
</tr>
<tr>
<td>tk_longdouble</td>
<td>long double</td>
</tr>
<tr>
<td>tk_wchar</td>
<td>Unicode character</td>
</tr>
<tr>
<td>tk_wstring</td>
<td>Unicode string</td>
</tr>
</tbody>
</table>

class CORBA::TypeCode

The TypeCode class represents the various types that can be defined in IDL. Type codes are most often used to describe the type of value being stored in an Any object, described on page 6-2. Type codes may also be passed as parameters to method invocations.

TypeCode objects can be created using the various CORBA::ORB.create_<type>_tc methods, whose description begins on page 5-7. You may also use the constructors listed here.
Include file

The corba.h file should be included when you use this class.

TypeCode constructors

CORBA::TypeCode(const CORBA::TypeCode obj);

This is a copy constructor.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>obj</td>
<td>The TypeCode object to be copied.</td>
</tr>
</tbody>
</table>

CORBA::TypeCode(CORBA::TCKind kind, CORBA::Boolean is_constant);

This method constructs a TypeCode object for types that do not require any additional parameters. A BAD_PARAM exception is raised if kind is not a valid type for this constructor.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>kind</td>
<td>Describes the type of object being represented. Must be one of the following: CORBA::tk_null, CORBA::tk_void, CORBA::tk_short, CORBA::tk_long, CORBA::tk_ushort, CORBA::tk_ulong, CORBA::tk_float, CORBA::tk_double, CORBA::tk_boolean, CORBA::tk_char, CORBA::tk_octet, CORBA::tk_any, CORBA::tk_TypeCode, CORBA::tk_Principal, CORBA::tk_longlong, CORBA::tk_ulonglong, CORBA::tk_longdouble, or CORBA::tk_wchar.</td>
</tr>
<tr>
<td>is_constant</td>
<td>If true, the type being represented is to be considered a constant. Otherwise, the object is not a constant.</td>
</tr>
</tbody>
</table>

CORBA::TypeCode(CORBA::TCKind kind, const char *rep_id, const char *name, const CORBA::StructMemberSeq& members, CORBA::Boolean is_constant);

This method constructs a TypeCode object for structures and Exception objects. A BAD_PARAM exception is raised if kind is not a valid type for this constructor.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>kind</td>
<td>Describes the type of object being represented. Must be either CORBA::tk_struct or CORBA::tk_except.</td>
</tr>
<tr>
<td>rep_id</td>
<td>The object’s repository identifier.</td>
</tr>
<tr>
<td>name</td>
<td>The object’s name.</td>
</tr>
<tr>
<td>is_constant</td>
<td>If true, the type being represented is a constant. Otherwise, the object is not a constant.</td>
</tr>
</tbody>
</table>
**TypeCode constructors**

**CORBA::TypeCode(const char *rep_id, const char *name, CORBA::TypeCode_ptr discrim_type, const CORBA::UnionMemberSeq& members, CORBA::Boolean is_constant);**

This method constructs a TypeCode object for unions.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rep_id</td>
<td>The object’s repository identifier.</td>
</tr>
<tr>
<td>name</td>
<td>The object’s name.</td>
</tr>
<tr>
<td>discrim_type</td>
<td>The TypeCode of the union’s discriminator.</td>
</tr>
<tr>
<td>members</td>
<td>A list of union’s members</td>
</tr>
<tr>
<td>is_constant</td>
<td>If true, the type being represented is a constant. Otherwise, the object is not a constant.</td>
</tr>
</tbody>
</table>

**CORBA::TypeCode(const char *rep_id, const char *name, const CORBA::EnumMemberSeq& members, CORBA::Boolean is_constant);**

This method constructs a TypeCode object for an enumeration.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rep_id</td>
<td>The object’s repository identifier.</td>
</tr>
<tr>
<td>name</td>
<td>The object’s name.</td>
</tr>
<tr>
<td>members</td>
<td>A list of enumeration’s members</td>
</tr>
<tr>
<td>is_constant</td>
<td>If true, the type being represented is a constant. Otherwise, the object is not a constant.</td>
</tr>
</tbody>
</table>

**CORBA::TypeCode(const char *rep_id, const char *name, CORBA::TypeCode_ptr original_type, CORBA::Boolean is_constant);**

This method constructs a TypeCode object for an alias.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rep_id</td>
<td>The object’s repository identifier.</td>
</tr>
<tr>
<td>name</td>
<td>The object’s name.</td>
</tr>
<tr>
<td>original_type</td>
<td>The TypeCode of the object being aliased.</td>
</tr>
<tr>
<td>is_constant</td>
<td>If true, the type being represented is a constant. Otherwise, the object is not a constant.</td>
</tr>
</tbody>
</table>
**TypeCode constructors**

**CORBA::TypeCode(const char *rep_id, const char *name, CORBA::Boolean is_constant);**

This method constructs a TypeCode object for an interface.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rep_id</td>
<td>The object’s repository identifier.</td>
</tr>
<tr>
<td>name</td>
<td>The object’s name.</td>
</tr>
<tr>
<td>is_constant</td>
<td>If true, the type being represented is a constant. Otherwise, the object is not a constant.</td>
</tr>
</tbody>
</table>

**CORBA::TypeCode(CORBA::ULong bound, CORBA::Boolean is_constant, CORBA::Boolean wstr=0);**

This method constructs a TypeCode object for strings and Unicode strings.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bound</td>
<td>The string’s length.</td>
</tr>
<tr>
<td>is_constant</td>
<td>If true, the type being represented is a constant. Otherwise, the object is not a constant.</td>
</tr>
<tr>
<td>wstr</td>
<td>If set to true, a Unicode string is being represented.</td>
</tr>
</tbody>
</table>

**CORBA::TypeCode(CORBA::TCKind kind, CORBA::ULong len, CORBA::TypeCode_ptr element_type, CORBA::Boolean is_constant);**

This method constructs a TypeCode object for sequences and arrays.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>kind</td>
<td>The type of object being represented. Must be either CORBA::tk_sequence or CORBA::tk_array.</td>
</tr>
<tr>
<td>len</td>
<td>The length of the array or sequence.</td>
</tr>
<tr>
<td>element_type</td>
<td>The TypeCode of the elements in the array or sequence.</td>
</tr>
<tr>
<td>is_constant</td>
<td>If true, the type being represented is a constant. Otherwise, the object is not a constant.</td>
</tr>
</tbody>
</table>

**CORBA::TypeCode(CORBA::ULong bound, CORBA::ULong offset, CORBA::Boolean is_constant);**

This method constructs a TypeCode object for recursive sequences.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bound</td>
<td>The length of the sequence.</td>
</tr>
<tr>
<td>offset</td>
<td>The offset where this type was previously generated.</td>
</tr>
<tr>
<td>is_constant</td>
<td>If true, the type being represented is a constant. Otherwise, the object is not a constant.</td>
</tr>
</tbody>
</table>
**TypeCode methods**

**content_type() const;**

This method returns the TypeCode of the elements in a sequence or array. It also will return the type of an alias. A BadKind exception is raised if this object’s kind is not CORBA::tk_sequence, CORBA::tk_array, or CORBA::tk_alias.

**default_index() const;**

This method returns the default index of a TypeCode representing a union. If this object’s kind is not CORBA::tk_union, a BadKind exception is raised.

**discriminator_type() const;**

This method returns the discriminator type of a TypeCode representing a union. If this object’s kind is not CORBA::tk_union, a BadKind exception is raised.

**equal(CORBA::TypeCode_ptr tc) const;**

This method compares this object with the specified TypeCode. If they match in every respect, true is returned. Otherwise, false is returned.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tc</td>
<td>The object to be compared to this object.</td>
</tr>
</tbody>
</table>

**id() const;**

This method returns the repository identifier of the type being represented by this object. If the type being represented does not have a repository identifier, a BadKind exception is raised. Types that have a repository identifier include:

- CORBA::tk_struct
- CORBA::tk_union
- CORBA::tk_enum
- CORBA::tk_alias
- CORBA::tk_except
- CORBA::tk_objref

**kind() const**

This method returns this object’s kind.

**length() const;**

This method returns the length of the string, sequence, or array represented by this object. A BadKind exception is raised if this object’s kind is not CORBA::tk_string, CORBA::tk_sequence, or CORBA::tk_array.
**TypeCode methods**

**CORBA::ULong** **member_count()** **const;**

This method returns the member count of the type being represented by this object. If the type being represented does not have members, a BadKind exception is raised. Types that have members include:

- CORBA::tk_struct
- CORBA::tk_union
- CORBA::tk_enum
- CORBA::tk_except

**CORBA::Any_ptr** **member_label**(CORBA::ULong **index** **const;**

This method returns the label of the member with the specified index from a TypeCode object for a union. If this object's kind is not CORBA::tk_union, a BadKind exception is raised. If the index is invalid, a Bounds exception is raised.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>index</td>
<td>The label of the union member whose type is to be returned. This index is zero-based.</td>
</tr>
</tbody>
</table>

**const char ** **member_name**(CORBA::ULong **index** **const;**

This method returns the name of the member with the specified index from the type being represented by this object. If the type being represented does not have members, a BadKind exception is raised. If the index is invalid, a Bounds exception is raised. Types that have members include:

- CORBA::tk_struct
- CORBA::tk_union
- CORBA::tk_enum
- CORBA::tk_except

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>index</td>
<td>The index of the member whose name is to be returned. This index is zero-based.</td>
</tr>
</tbody>
</table>

**CORBA::TypeCode_ptr** **member_type**(CORBA::ULong **index** **const;**

This method returns the type of the member with the specified index from the type being represented by this object. If the type being represented does not have members with types, a BadKind exception is raised. If the index is invalid, a Bounds exception is raised. Types that have members include:

- CORBA::tk_union
- CORBA::tk_except

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>index</td>
<td>The index of the member whose type is to be returned. This index is zero-based.</td>
</tr>
</tbody>
</table>
**TypeCode methods**

```c++
const char *name() const;
```

This method returns the name of the type being represented by this object. If the type being represented does not have a name, a `BadKind` exception is raised. Types that have a name include:

- CORBA::tk_objref
- CORBA::tk_struct
- CORBA::tk_union
- CORBA::tk_enum
- CORBA::tk_alias
- CORBA::tk_except

```c++
CORBA::Any_ptr parameter(CORBA::Long index) const;
```

This method returns an `Any` object representing the parameter with the specified `index`. If the index is invalid, a `Bounds` exception is raised.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>index</td>
<td>The index of the parameter to be returned. This index is zero-based.</td>
</tr>
</tbody>
</table>

```c++
CORBA::Long param_count() const;
```

This method returns the number of parameters.

```c++
static CORBA::TypeCode_ptr _duplicate(CORBA::TypeCode_ptr obj);
```

This static method duplicates the specified `TypeCode`.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>obj</td>
<td>The object to be duplicated.</td>
</tr>
</tbody>
</table>

```c++
static CORBA::TypeCode_ptr _nil();
```

This static method returns a `NULL` `TypeCode` pointer that can be used for initialization purposes.

```c++
static void _release(CORBA::TypeCode_ptr obj);
```

This static method decrements the reference count to the specified object. If the reference count is zero, it also frees all memory that it is managing and then deletes the object.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>obj</td>
<td>The object to be released.</td>
</tr>
</tbody>
</table>
Chapter 7

Interface Repository classes and interfaces

This chapter describes the classes and interfaces that you can use to access the interface repository. The interface repository maintains information on modules and the interfaces they contain as well as other types like operations, attributes and constants. This chapter includes the following sections:

- AliasDef page 7-2
- AttributeDef page 7-3
- AttributeMode page 7-4
- ConstantDescription page 7-5
- Container page 7-8
- EnumDef page 7-15
- ExceptionDef page 7-16
- FullInterfaceDescription page 7-17
- InterfaceDef page 7-18
- IRObj ect page 7-21
- ModuleDescription page 7-22
- OperationDescription page 7-25
- ParameterDescription page 7-26
- PrimitiveDef page 7-27
- Repository page 7-28
- StringDef page 7-30
- ArrayDef page 7-2
- AttributeDescription page 7-4
- ConstantDef page 7-5
- Contained page 7-6
- Description page 7-14
- EstructDef page 7-15
- ExceptionDescription page 7-16
- IDLType page 7-18
- InterfaceDescription page 7-21
- ModuleDef page 7-22
- OperationDef page 7-23
- OperationMode page 7-26
- ParameterMode page 7-26
- PrimitiveKind page 7-27
- SequenceDef page 7-30
- StructDef page 7-31
This class is derived from the TypedefDef class and represents an alias for a typedef that is stored in the interface repository. This class provides methods for setting and obtaining the IDLType of the original typedef.

For more information on the TypedefDef class, see page 7-31. For more information on the IDLType class, see page 7-18.

**AliasDef methods**

```cpp
corba::idl_type original_type_def();
```

This method returns the IDLType of the original typedef for which this object is an alias.

```cpp
void original_type_def(corba::idl_type_ptr val);
```

This method sets the IDLType of the original typedef for which this object is an alias.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>val</td>
<td>The IDLType to set for this alias.</td>
</tr>
</tbody>
</table>

This class is derived from the IDLType class and represents an array that is stored in the interface repository. It provides methods for setting and obtaining the type of the elements in the array as well as the length of the array.

**ArrayDef methods**

```cpp
corba::type_code element_type();
```

This method returns the TypeCode of the array’s elements.

```cpp
corba::idl_type_ptr element_type_def();
```

This method returns the IDLType of the elements stored in this array.
AttributeDef

```cpp
void element_type_def(CORBASharedPtr<IDLType> element_type_def);
This method sets the IDLType of the elements stored in the array.
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>element_type_def</td>
<td>The IDLType of the elements in the array.</td>
</tr>
</tbody>
</table>

```cpp
CORBA::ULong length();
This method returns the number of elements in the array.
```

```cpp
void length(CORBASharedPtr<ULong> length);
This method sets the number of elements in the array.
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>length</td>
<td>The number of elements in the array.</td>
</tr>
</tbody>
</table>

AttributeDef

```cpp
class CORBA::AttributeDef : public CORBA::Contained
The interface is used to represent an interface attribute that is stored in the interface repository. This interface provides methods for setting and obtaining the attribute’s mode, typedef. A method is also provided for obtaining the attribute’s type.
```

AttributeDef methods

```cpp
CORBA::AttributeMode mode();
This method returns the mode of the attribute.
```

```cpp
void mode(CORBASharedPtr<AttributeMode> _val);
This method sets the mode of the attribute.
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>_val</td>
<td>The mode to set.</td>
</tr>
</tbody>
</table>

```cpp
CORBA::TypeCode_ptr typedef();
This method returns the TypeCode representing the attribute’s type.
```

```cpp
CORBA::IDLType_ptr typedef();
This method returns this object’s IDLType.
```
AttributeDescription

void type_def(CORBA::IDLType_ptr type_def);

This method sets the IDLType for which this object.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>type_def</td>
<td>The IDLType of this object.</td>
</tr>
</tbody>
</table>

AttributeDescription

struct AttributeDescription

The AttributeDescription structure describes an attribute that is stored in the interface repository.

AttributeDescription members

CORBA::Identifier_var name

This member represents the name of the attribute.

CORBA::RepositoryId_var id

This member represents the repository id of the attribute.

CORBA::RepositoryId_var defined_in

This member represents the name of the module or interface in which this attribute is defined.

CORBA::VersionSpec version

This member represents the attribute’s version.

CORBA::TypeCode_var type

This member represents the attribute’s IDL type.

CORBA::AttributeMode mode

This member represents the mode of this attribute.

AttributeMode

enum AttributeMode

The enumeration defines the values used to represent the mode of an attribute; either read-only or normal (read-write).
**AttributeMode values**

<table>
<thead>
<tr>
<th>Constant</th>
<th>Represents</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATTR_NORMAL</td>
<td>A read-write attribute.</td>
</tr>
<tr>
<td>ATTR_READONLY</td>
<td>A read-only attribute.</td>
</tr>
</tbody>
</table>

**ConstantDef**

class CORBA::ConstantDef : public CORBA::Contained

The class is used to represent a constant definition that is stored in the interface repository. This interface provides methods for setting and obtaining the constant’s type, value, and typedef.

**ConstantDef methods**

CORBA::TypeCode_ptr type();

This method returns the TypeCode representing the object’s type.

CORBA::IDLType_ptr type_def();

This method returns this object’s IDLType.

void type_def(CORBA::IDLType_ptr type_def);

This method sets the IDLType of the original typedef for this object.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>type_def</td>
<td>The IDLType of this object.</td>
</tr>
</tbody>
</table>

CORBA::Any *value();

This method returns a pointer to an Any object representing this object’s value.

void value(CORBA::Any& _val);

This method sets the value for this constant.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>_val</td>
<td>An Any object that represents this object’s value.</td>
</tr>
</tbody>
</table>

**ConstantDescription**

struct CORBA::ClassName

The ConstantDescription structure describes a constant that is stored in the interface repository.
ConstantDescription members

CORBA::Identifier_var name
This member represents the name of the constant.

CORBA::RepositoryId_var id
This member represents the repository id of the constant.

CORBA::RepositoryId_var defined_in
This member represents the name of the module or interface in which this constant is defined.

CORBA::VersionSpec version
This member represents the constant’s version.

CORBA::TypeCode_var type
This member represents the constant’s IDL type.

CORBA::Any value
This member represents the value of this constant.

Contained

class CORBA::Contained : public CORBA::IRObject

The Contained class is used to derive all interface repository objects that are themselves contained within another interface repository object. This class provides methods for:

- Setting and retrieving the object’s name and version.
- Determining the Container that contains this object.
- Obtaining the object’s absolute name, containing repository, and description.
- Moving an object from one container to another.

Include file

The corba.h file should be included when you use this class.

```
interfaceContained:IRObject{
    attribute RepositoryId id;
    attribute Identifier name;
    attribute VersionSpec version;

    readonly attribute Container defined_in;
    readonly attribute ScopedName absolute_name;
    readonly attribute Repository containing_Repository;
```
struct Description {
    DefinitionKind kind;
    any value;
};
Description describe();
void move(
    in Container new_Container,
    in Identifier new_name,
    in VersionSpec new_version
);
Container

void name(const char * val);

This method sets the name of the contained object.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>The object’s name.</td>
</tr>
</tbody>
</table>

CORBA::VersionSpec version();

This method returns the object’s version, which distinguishes this object from other objects that have the same name.

void version(CORBA::VersionSpec& val);

This method sets this object’s version.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>val</td>
<td>The exception’s id.</td>
</tr>
</tbody>
</table>

void move(CORBA::Container_ptr new_container, const char *new_name,
           CORBA::VersionSpec& new_version);

Moves this object from its current Container to the new_container.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>new_container</td>
<td>The Container to which this object is being moved.</td>
</tr>
<tr>
<td>new_name</td>
<td>The new name for the object.</td>
</tr>
<tr>
<td>new_version</td>
<td>The new version specification for the object.</td>
</tr>
</tbody>
</table>

Container

class CORBA::Container : public CORBA::IRObject

The Container class is used to create a containment hierarchy in the interface repository. A Container object holds object definitions derived from the Contained class. All object definitions derived from the Container class, with the exception of the Repository class, also inherit from the Contained class.

The Container provides methods to create all types of IDL types defined in orbtypes.h, including InterfaceDef, ModuleDef and ConstantDef classes. Each definition that is created will have its defined_in attribute initialized to point to this object.
Include file

The corba.h file should be included when you use this class.

```cpp
interface Container: IRObjet {    
    Contained lookup(in ScopedName search_name);    
    ContainedSeq contents{        
        in DefinitionKind limit_type,        
        in boolean exclude_inherited    
    };    
    ContainedSeq lookup_name{        
        in Identifier search_name,        
        in long levels_to_search,        
        in CORBA::DefinitionKind limit_type,        
        in boolean exclude_inherited    
    };    
    struct Description {        
        Contained Contained_object;        
        DefinitionKind kind;        
        any value;    
    };    
    typedef sequence<Description> DescriptionSeq;    
    DescriptionSeq describe_contents{        
        in DefinitionKind limit_type,        
        in boolean exclude_inherited,        
        in long max_returned_objs    
    };
```

**Container methods**

```cpp
CORBA::ContainedSeq * contents(CORBA::DefinitionKind limit_type,    
CORBA::Boolean exclude_inherited);
```

This method returns a list of contained object definitions directly contained or inherited into the container. You can use this method to navigate through the hierarchy of object definitions in the Repository. All object definitions contained by modules in the Repository are returned, followed by all object definitions contained within each of those modules.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>limit_type</td>
<td>The interface object types to be returned. Specifying <code>dk_all</code> will return objects of all types.</td>
</tr>
<tr>
<td>exclude_inherited</td>
<td>If set to TRUE, inherited objects will not be returned.</td>
</tr>
</tbody>
</table>
**Container methods**

**CORBA::AliasDef_ptr create_alias(const char * id,**
**const char * name,**
**const CORBA::VersionSpec& version,**
**CORBA::IDLType_ptr original_type);**

This method creates a `AliasDef` object in this `Container` with the specified attributes and returns a pointer to the newly created object.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>The alias’s id.</td>
</tr>
<tr>
<td>name</td>
<td>The alias’s name.</td>
</tr>
<tr>
<td>version</td>
<td>The alias’s version.</td>
</tr>
<tr>
<td>original_type</td>
<td>The type of the object for which this object is an alias.</td>
</tr>
</tbody>
</table>

**CORBA::ConstantDef_ptr create_constant(const char * id,**
**const char * name,**
**const CORBA::VersionSpec& version,**
**CORBA::IDLType_ptr type,**
**const CORBA::Any& value);**

This method creates a `ConstantDef` object in this `Container` with the specified attributes and returns a pointer to the newly created object.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>The constant’s id.</td>
</tr>
<tr>
<td>name</td>
<td>The constant’s name.</td>
</tr>
<tr>
<td>version</td>
<td>The constant’s version.</td>
</tr>
<tr>
<td>type</td>
<td>The type of the value specified below.</td>
</tr>
<tr>
<td>value</td>
<td>The constant’s value.</td>
</tr>
</tbody>
</table>

**CORBA::EnumDef_ptr create_enum(const char * id,**
**const char * name,**
**const CORBA::VersionSpec& version,**
**const CORBA::EnumMemberSeq& members);**

This method creates a `EnumDef` object in this `Container` with the specified attributes and returns a pointer to the newly created object.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>The enumeration’s id.</td>
</tr>
<tr>
<td>name</td>
<td>The enumeration’s name.</td>
</tr>
<tr>
<td>version</td>
<td>The enumeration’s version.</td>
</tr>
<tr>
<td>members</td>
<td>A list of the enumeration’s fields.</td>
</tr>
</tbody>
</table>
CORBA::EstructDef_ptr create_estruct( const char *id,
const char *name,
const CORBA::VersionSpec& version,
CORBA::EstructDef_ptr base,
const CORBA::EnumMemberSeq& members);

This method creates a EnumDef object in this Container with the specified attributes and returns a pointer to the newly created object.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>The extensible structure’s id.</td>
</tr>
<tr>
<td>name</td>
<td>The extensible structure’s name.</td>
</tr>
<tr>
<td>version</td>
<td>The extensible structure’s version.</td>
</tr>
<tr>
<td>base</td>
<td>The base class from which the extensible structure is derived. Use NULL to indicate that this is the root structure.</td>
</tr>
<tr>
<td>members</td>
<td>A list of the extensible structure’s fields.</td>
</tr>
</tbody>
</table>

CORBA::ExceptionDef_ptr create_exception(const char * id,
const char * name,
const CORBA::VersionSpec& version,
const CORBA::EnumMemberSeq& members);

This method creates a ExceptionDef object in this Container with the specified attributes and returns a pointer to the newly created object.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>The exception’s id.</td>
</tr>
<tr>
<td>name</td>
<td>The exception’s name.</td>
</tr>
<tr>
<td>version</td>
<td>The exception’s version.</td>
</tr>
<tr>
<td>members</td>
<td>A list of the types of the members of the exception, if any.</td>
</tr>
</tbody>
</table>

CORBA::InterfaceDef_ptr create_interface(const char * id,
const char * name,
const CORBA::VersionSpec& version,
const CORBA::InterfaceDefSeq& base_interfaces);

This method creates a InterfaceDef object in this Container with the specified attributes and returns a pointer to the newly created object.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>The interface’s id.</td>
</tr>
<tr>
<td>name</td>
<td>The interface’s name.</td>
</tr>
<tr>
<td>version</td>
<td>The interface’s version.</td>
</tr>
<tr>
<td>base_interfaces</td>
<td>A list of all interfaces that this interface inherits from.</td>
</tr>
</tbody>
</table>
Container methods

CORBA::ModuleDef_ptr create_module(const char * id,
const char *name,
const CORBA::VersionSpec& version);

This method creates a ModuleDef object in this Container with the specified attributes and returns a pointer to the newly created object.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>The module’s id.</td>
</tr>
<tr>
<td>name</td>
<td>The module’s name.</td>
</tr>
<tr>
<td>version</td>
<td>The module’s version.</td>
</tr>
</tbody>
</table>

CORBA::StructDef_ptr create_struct(const char * id,
const char *name,
const CORBA::VersionSpec& version,
const CORBA::StructMemberSeq& members);

This method creates a StructureDef object in this Container with the specified attributes and returns a pointer to the newly created object.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>The structure’s id.</td>
</tr>
<tr>
<td>name</td>
<td>The structure’s name.</td>
</tr>
<tr>
<td>version</td>
<td>The structure’s version.</td>
</tr>
<tr>
<td>members</td>
<td>The values for the structure’s fields.</td>
</tr>
</tbody>
</table>

CORBA::UnionDef_ptr create_union(const char * id,
const char *name,
const CORBA::VersionSpec& version,
CORBA::IDLType_ptr discriminator_type,
const CORBA::UnionMemberSeq& members);

This method creates a UnionDef object in this Container with the specified attributes and returns a pointer to the newly created object.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>The union’s id.</td>
</tr>
<tr>
<td>name</td>
<td>The union’s name.</td>
</tr>
<tr>
<td>version</td>
<td>The union’s version.</td>
</tr>
<tr>
<td>discriminator_type</td>
<td>The type of the union’s descriminant value.</td>
</tr>
<tr>
<td>members</td>
<td>A list of the types of each of the union’s fields.</td>
</tr>
</tbody>
</table>
**Container methods**

CORBA::DescriptionSeq * **describe_contents**(CORBA::DefinitionKind limit_type,
CORBA::Boolean exclude_inherited,
CORBA::Long max_returned_objs);

This method returns a description for all definitions directly contained by or inherited into this container.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>limit_type</td>
<td>The interface object types whose descriptions are to be returned. Specifying dk_all will return the descriptions for objects of all types.</td>
</tr>
<tr>
<td>exclude_inherited</td>
<td>If set to true, descriptions for inherited objects will not be returned.</td>
</tr>
<tr>
<td>max_returned_objs</td>
<td>The maximum number of descriptions to be returned. Setting this parameter to –1 will return all objects.</td>
</tr>
</tbody>
</table>

CORBA::Contained_ptr **lookup**(const char *search_name);

This method locates a definition relative to this container, given a scoped name. An absolute scoped name, one beginning with “::”, may be specified to locate a definition within the enclosing repository. If no object is found, a NULL value is returned.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>search_name</td>
<td>The object’s interface name.</td>
</tr>
</tbody>
</table>

CORBA::ContainedSeq * **lookup_name**(const char *search_name,
CORBA::Long levels_to_search,
CORBA::DefinitionKind limit_type,
CORBA::Boolean exclude_inherited);

This method locates an object by name within a particular object. The search can be constrained by the number of levels in the hierarchy to be searched, the object type, and whether inherited objects should be returned.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>search_name</td>
<td>The contained object’s name.</td>
</tr>
<tr>
<td>levels_to_search</td>
<td>The number of levels in the hierarchy to search. Setting this parameter to a value of –1 will cause all levels to be searched. Setting this parameter to 1 will search only this object.</td>
</tr>
<tr>
<td>limit_type</td>
<td>The interface object types to be returned. Specifying dk_all will return objects of all types.</td>
</tr>
<tr>
<td>exclude_inherited</td>
<td>If set to true, inherited objects will not be returned.</td>
</tr>
</tbody>
</table>
**DefinitionKind**

**enum** CORBA::DefinitionKind

The `DefinitionKind` enumeration contains the constants that define the possible types of interface repository objects. There are a set of integer constants, prefixed with `tk_`, that correspond to all the possible type codes.

**DefinitionKind values**

<table>
<thead>
<tr>
<th>Constant</th>
<th>Represents</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>dk_none</code></td>
<td>Exclude all types (used in repository lookup methods)</td>
</tr>
<tr>
<td><code>dk_all</code></td>
<td>All possible types (used in repository lookup methods)</td>
</tr>
<tr>
<td><code>dk_Attribute</code></td>
<td>Alias</td>
</tr>
<tr>
<td><code>dk_Constant</code></td>
<td>Constant</td>
</tr>
<tr>
<td><code>dk_Exception</code></td>
<td>Exception</td>
</tr>
<tr>
<td><code>dk_Interface</code></td>
<td>Interface</td>
</tr>
<tr>
<td><code>dk_Module</code></td>
<td>Module</td>
</tr>
<tr>
<td><code>dk_Operation</code></td>
<td>Interface Operation</td>
</tr>
<tr>
<td><code>dk_TypeDef</code></td>
<td>Typedef</td>
</tr>
<tr>
<td><code>dk_Alias</code></td>
<td>Alias</td>
</tr>
<tr>
<td><code>dk_Struct</code></td>
<td>Struct</td>
</tr>
<tr>
<td><code>dk_Union</code></td>
<td>Union</td>
</tr>
<tr>
<td><code>dk_Enum</code></td>
<td>Enum</td>
</tr>
<tr>
<td><code>dk_Primitive</code></td>
<td>Primitive type (such as int or long)</td>
</tr>
<tr>
<td><code>dk_String</code></td>
<td>String</td>
</tr>
<tr>
<td><code>dk_Sequence</code></td>
<td>Sequence</td>
</tr>
<tr>
<td><code>dk_Array</code></td>
<td>Array</td>
</tr>
<tr>
<td><code>dk_Repository</code></td>
<td>Repository</td>
</tr>
<tr>
<td><code>dk_Wstring</code></td>
<td>Unicode string</td>
</tr>
<tr>
<td><code>dk_Estruct</code></td>
<td>Extended structure</td>
</tr>
</tbody>
</table>

**Description**

**struct** Description

This structure provides a generic description for items in the interface repository that are derived from the `Contained` class.

**Description members**

CORBA::Contained_var contained_object

The object contained in this struct.
EnumDef

class CORBA::EnumDef : public CORBA::TypeDefDef

The class is used to represent an enumeration that is stored in the interface repository. This interface provides methods for setting and retrieving the enumeration’s list of members.

EnumDef methods

CORBA::EnumMemberSeq *members() ;

This method returns the enumeration’s list of members.

void members(CORBA::EnumMemberSeq members) ;

This method sets the enumeration’s list of members.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>members</td>
<td>The list of members.</td>
</tr>
</tbody>
</table>

EstructDef

class EstructDef : public CORBA::TypeDefDef

The class is used to represent an Estruct that is stored in the interface repository. Estruct objects represent classes that are passed as parameters on an operation request. This class provides methods for setting and retrieving the object’s list of members and the Estruct of the base class. A method is also provided for obtaining the TypeCode of the base class.

EstructDef methods

CORBA::TypeCode_ptr base() ;

This method returns the TypeCode of the base class of this Estruct.
ExceptionDef

void base_def(CORBA::EstructDef_ptr base_def);
This method sets the base class for this EstructDef object.

CORBA::StructMemberSeq *members() ;
This method returns this EstructDef object’s list of members.
void members(const CORBA::StructMemberSeq& members) ;
This method sets this EstructDef object’s list of members.

ExceptionDef

class ExceptionDef : public CORBA::Contained
The class is used to represent an exception that is stored in the interface repository. This class provides methods for setting and retrieving the exception’s list of members as well as a method for retrieving the exception’s TypeCode.

ExceptionDef methods

CORBA::StructMemberSeq *members() ;
This method returns this exception’s list of members.
void members(CORBA::StructMemberSeq& members) ;
This method sets the exception’s list of members.

Parameter Description
members The list of members.

CORBA::TypeCode_ptr type() ;
This method returns the TypeCode that represents this exception’s type.

ExceptionDescription

struct ExceptionDescription
This structure is used to represent information about an exception that is stored in the interface repository.
ExceptionDescription members

CORBA::String_var defined_in
This member represents the name of the module or interface in which this exception is defined.

CORBA::String_var id
This member represents the repository id of the exception.

CORBA::String_var name
This member represents the name of the exception.

CORBA::TypeCode_var type
This member represents the exception’s IDL type.

CORBA::VersionSpec version
This member represents the exception’s version.

FullInterfaceDescription

struct CORBA::FullInterfaceDescription
The FullInterfaceDescription structure describes a constant that is stored in the interface repository.

FullInterfaceDescription members

CORBA::String_var Name
This member represents the name of the interface.

CORBA::String_var id
This member represents the repository id of the interface.

CORBA::String_var defined_in
This member represents the name of the module or interface in which this interface is defined.

CORBA::VersionSpec version
This member represents the interface’s version.

CORBA::OpDescriptionSeq operations
This member represents a list of operations offered by this interface.

CORBA::AttrDescriptionSeq attributes
This member represents a list of attributes contained in this interface.
**IDLType**

**base_interfaces**

This member represents a list of interfaces from which this interface inherits.

**TypeCode_var type**

This member represents this interface's TypeCode.

---

**IDLType**

```cpp
class CORBA::IDLType : public CORBA::IRObject

The IDLType class provides an abstract interface that is inherited by all interface repository definitions that represent IDL types. This class provides a method for returning an object's TypeCode, which identifies the object's type. The IDLType is unique; the Typecode is not.

**Include file**

You should include the file corba.h when using this class.

```cpp
interface IDLType:IRObject {
    readonly attribute TypeCode type;
};
```

**IDLType methods**

**CORBA::Typecode_ptr type();**

This method returns the typecode of the current IRObject.

---

**InterfaceDef**

```cpp
class CORBA::InterfaceDef : public CORBA::Container,
    public CORBA::Contained,
    public CORBA::IDLType

The InterfaceDef class is used to define an ORB object's interface that is stored in the interface repository.

For more information, see "Container" on page 7-8, "Contained" on page 7-6, and "IDLType" on page 7-18 in this guide.

**Include file**

You should include the file corba.h when you use this class.
interface InterfaceDef: Container, Contained, IDLType {
    typedef sequence<RepositoryId> RepositoryIdSeq;
    typedef sequence<OperationDescription> OpDescriptionSeq;
    typedef sequence<AttributeDescription> AttrDescriptionSeq;
    attribute InterfaceDefSeq base_interfaces;
    boolean is_a(in RepositoryId interface_id);
    struct FullInterfaceDescription {
        Identifier name;
        RepositoryId id;
        RepositoryId defined_in;
        VersionSpec version;
        OpDescriptionSeq operations;
        AttrDescriptionSeq attributes;
        RepositoryIdSeq base_interfaces;
        TypeCode type;
    };
}

FullInterfaceDescription describe_interface();

AttributeDef create_attribute(
    in RepositoryId id,
    in Identifier name,
    in VersionSpec version,
    in IDLType type,
    in CORBA::AttributeMode mode
);

OperationDef create_operation(
    in RepositoryId id,
    in Identifier name,
    in VersionSpec version,
    in IDLType result,
    in OperationMode mode,
    in ParDescriptionSeq params,
    in ExceptionDefSeq exceptions,
    in ContextIdSeq contexts
);

struct InterfaceDescription {
    Identifier name;
    RepositoryId id;
    RepositoryId defined_in;
    VersionSpec version;
    RepositoryIdSeq base_interfaces;
};

InterfaceDef methods

CORBA::InterfaceDefSeq *base_interfaces();

This method returns a list of interfaces from which this class inherits.
InterfaceDef methods

void base_interfaces(const CORBA::InterfaceDefSeq& val);

This method sets the list of the interfaces from which this class inherits.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>val</td>
<td>The list of interfaces from which this interface inherits.</td>
</tr>
</tbody>
</table>

CORBA::AttributeDef_ptr create_attribute(const char * id,
const char * name,
const CORBA::VersionSpec & version,
CORBA::IDLType_ptr type,
CORBA::AttributeMode mode);

This method returns a pointer to a newly created AttributeDef that is contained in this object. The id, name, version, type, and mode are set to the specified value.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>The interface id to use.</td>
</tr>
<tr>
<td>name</td>
<td>The interface name to use.</td>
</tr>
<tr>
<td>version</td>
<td>The interface version to use.</td>
</tr>
<tr>
<td>mode</td>
<td>The interface mode. See page 7-4 for a list of possible values.</td>
</tr>
</tbody>
</table>

CORBA::OperationDef_ptr create_operation(const char * id,
const char * name,
CORBA::VersionSpec & version,
CORBA::IDLType_ptr result,
CORBA::OperationMode mode,
const CORBA::ParDescriptionSeq & params,
const CORBA::ExceptionDefSeq & exceptions,
const CORBA::ContextIdSeq & contexts);

This method creates a new OperationDef that is contained by this object, using the specified parameters. The defined_in attribute of the newly created OperationDef is set to identify this InterfaceDef.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>The interface id for this operation.</td>
</tr>
<tr>
<td>name</td>
<td>The name of this operation.</td>
</tr>
<tr>
<td>version</td>
<td>The operation’s version.</td>
</tr>
<tr>
<td>result</td>
<td>The IDL type returned by the operation.</td>
</tr>
<tr>
<td>mode</td>
<td>The mode of this operation—oneway or normal.</td>
</tr>
<tr>
<td>params</td>
<td>The list of parameters to pass to this operation.</td>
</tr>
<tr>
<td>exceptions</td>
<td>The list of exceptions raised by this operation.</td>
</tr>
<tr>
<td>contexts</td>
<td>Context lists are names of values expected in context and passed along with the request.</td>
</tr>
</tbody>
</table>
InterfaceDescription

CORBA::InterfaceDef::FullInterfaceDescription *describe_interface();

This method returns a FullInterfaceDescription, which describes this object’s interface.

CORBA::Boolean is_a(const char *interface_id);

This method returns true if this interface is identical to or inherits from, directly or indirectly, from the specified interface.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>interface_id</td>
<td>The id of the interface to be checked against this interface.</td>
</tr>
</tbody>
</table>

### InterfaceDescription

struct InterfaceDescription

This structure describes an object that is stored in the interface repository.

### InterfaceDescription members

CORBA::String_var name

The name of the interface.

CORBA::String_var id

The interface’s repository identifier.

CORBA::String_var defined_in

The name of the module or interface in which this interface is defined.

CORBA::VersionSpec version

The interface’s version.

CORBA::RepositoryIdSeq base_interfaces

A list of base interfaces for this interface.

### IRObect

class IRObect : CORBA::Object

The IRObect class offers the most generic interface for interface repository objects. The Container class, IDLType, Contained, and others are derived from this class.
Include file

You should include the file corba.h when you use this class.

```cpp
interface IRObject {
    readonly attribute DefinitionKind def_kind;
    void destroy();
};
```

**IRObject methods**

```cpp
CORBA::DefinitionKind def_kind();
```

This method returns the type of this interface repository object. See page 7-14 for a list of possible types.

```cpp
void destroy();
```

This method deletes this object from the interface repository. If this object is a Container, all of its contents will also be deleted. If the object is currently contained by another object, it will be removed. The `destroy` method returns the `Exception(CORBA::BAD_PARAM)` when invoked on a PrimitiveDef or Repository object. The Repository class is described on page 7-28 of this guide.

**ModuleDef**

```cpp
class ModuleDef : CORBA::Container,CORBA::Contained
```

The class is used to represent an IDL module in the interface repository.

**ModuleDescription**

```cpp
struct ModuleDescription
```

The `ModuleDescription` structure describes a constant that is stored in the interface repository.

**ModuleDescription members**

```cpp
CORBA::String_var name
```

This member represents the name of the module.

```cpp
CORBA::String_var id
```

This member represents the repository id of the module.

```cpp
CORBA::String_var defined_in
```

This member represents the name of the module in which this interface is defined.
CORBA::VersionSpec version

This member represents the module’s version.

The OperationDef class contains information about an interface operation that is stored in the interface repository. This class is derived from the Contained class, which is described on page 7-6. The inherited describe method returns a OperationDescription structure that provides complete information on the operation.

Include file

You should include the file corba.h when you use this class.

interface OperationDef: Contained {
    typedef sequence<ParameterDescription> ParDescriptionSeq;
    typedef Identifier ContextIdentifier;
    typedef sequence<ContextIdentifier> ContextIdSeq;
    typedef sequence<ExceptionDef> ExceptionDefSeq;
    typedef sequence<ExceptionDescription> ExcDescriptionSeq;
    readonly attribute TypeCode result;
    attribute IDLType result_def;
    attribute ParDescriptionSeq params;
    attribute CORBA::OperationMode mode;
    attribute ContextIdSeq contexts;
    attribute ExceptionDefSeq exceptions;
};

struct OperationDescription {
    Identifier name;
    RepositoryId id;
    RepositoryId defined_in;
    VersionSpec version;
    TypeCode result;
    OperationMode mode;
    ContextIdSeq contexts;
    ParDescriptionSeq parameters;
    ExcDescriptionSeq exceptions;
};

OperationDef methods

CORBA::ContextIdSeq * contexts();

This method returns a list of context identifiers that apply to the operation.
**OperationDef methods**

```cpp
void context(const CORBA::ContextIdSeq& val);

This method sets the list of the context identifiers that apply to this operation.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>val</td>
<td>The list of context identifiers.</td>
</tr>
</tbody>
</table>

CORBA::ExceptionDefSeq * exceptions();

This method returns a list of the exception types that can be raised by this operation.

void exceptions(const CORBA::ExceptionDefSeq& val);

This method sets the list of exception types that may be raised by this operation.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>val</td>
<td>The list of exceptions that this operation may raise.</td>
</tr>
</tbody>
</table>

CORBA::OperationMode mode();

This method returns the mode of the operation represented by this OperationDef. The mode may be normal or oneway. Operations that have a normal mode are synchronous and return a value to the client application. Oneway operations do not block and no response is sent from the object implementation to the client.

void mode(CORBA::OperationMode val);

This method sets the mode of the operation.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>val</td>
<td>The desired mode of this operation, either OP_ONEWAY or OP_NORMAL. See page 7-26 for more information.</td>
</tr>
</tbody>
</table>

CORBA::ParDescriptionSeq * params();

This method returns a pointer to a list of ParameterDescription structures that describe the parameters to this OperationDef.

void params(const CORBA::ParDescriptionSeq& val);

This method sets the list of the ParameterDescription structures for this OperationDef. The order of the structures is significant and should correspond to the order defined in the IDL definition for the operation.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>val</td>
<td>The list of ParameterDescription structures.</td>
</tr>
</tbody>
</table>
CORBA::TypeCode_ptr result();

This method returns a pointer to a TypeCode representing the type of the value returned by this Operation. The TypeCode is a read-only attribute.

CORBA::IDLType_ptr result_def();

This method returns a pointer to the definition of the IDL type returned by this OperationDef.

void result_def(CORBA::IDLType_ptr val);

This method sets the definition of the type returned by this OperationDef.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>val</td>
<td>A pointer to the type definition to use.</td>
</tr>
</tbody>
</table>

**OperationDescription**

struct OperationDescription

The OperationDescription structure describes an operation that is stored in the interface repository.

**OperationDescription members**

CORBA::String_var name

This variable represents the name the of the operation.

CORBA::String_var id

This variable represents the repository id of the operation.

CORBA::String_var defined_in

This variable represents the name of the operation in which this interface is defined.

CORBA::VersionSpec version

This variable represents the operation’s version.

CORBA::TypeCode_var result

This variable represents the operation’s result.

CORBA::OperationMode mode

This variable represents the operation’s mode.

CORBA::ContextIdSeq contexts

This variable represents the operation’s version.

CORBA::ParameterDescriptionSeq parameters

This variable represents the operation’s parameters.
**OperationMode**

The enumeration defines the values used to represent the mode of an operation; either oneway or normal. Oneway operations are those for which the client application does not expect a response. Normal requests involve a response being sent to the client by the object implementation that contains the results of the request.

**OperationMode values**

<table>
<thead>
<tr>
<th>Constant</th>
<th>Represents</th>
</tr>
</thead>
<tbody>
<tr>
<td>OP_NORMAL</td>
<td>A normal operation request.</td>
</tr>
<tr>
<td>OP_ONeway</td>
<td>A oneway operation request.</td>
</tr>
</tbody>
</table>

**ParameterDescription**

```plaintext
struct ParameterDescription
```

The `ParameterDescription` structure describes a parameter for an operation that is stored in the interface repository.

**ParameterDescription members**

- **CORBA::String_var name**
  
  This member represents the name of the parameter.

- **CORBA::TypeCode_var type**
  
  This member represents the parameter’s type.

- **CORBA::IDLType_var type_def**
  
  This member represents the parameter’s IDL type.

- **CORBA::ParameterMode mode**
  
  This member represents the parameter’s mode.

**ParameterMode**

```plaintext
genum ParameterMode
```

The enumeration defines the values used to represent the mode of a parameter for an operation.

**CORBA::ExceptionDescriptionSeq exceptions**

This variable represents the exceptions that this operation may raise.
ParameterMode values

<table>
<thead>
<tr>
<th>Constant</th>
<th>Represents</th>
</tr>
</thead>
<tbody>
<tr>
<td>PARAM_IN</td>
<td>Parameter is for input from the client to the server.</td>
</tr>
<tr>
<td>PARAM_OUT</td>
<td>Parameter is for output of results from the server to the client.</td>
</tr>
<tr>
<td>PARAM_INOUT</td>
<td>Parameter may be used for both input from the client and output from the server.</td>
</tr>
</tbody>
</table>

PrimitiveDef

class PrimitiveDef : public CORBA::IDLType

The class is used to represent a primitive (such as an int or a long) that is stored in the interface repository. It provides a method for retrieving the kind of primitive that is being represented.

PrimitiveDef methods

CORBA::PrimitiveKind kind();

This method returns the kind of primitive represented by this object.

PrimitiveKind

enum CORBA::PrimitiveKind

The PrimitiveKind enumeration contains the constants that define the primitive types of objects that may be stored in the interface repository.

PrimitiveKind values

<table>
<thead>
<tr>
<th>Constant</th>
<th>Represents</th>
</tr>
</thead>
<tbody>
<tr>
<td>pk_null</td>
<td>Null value</td>
</tr>
<tr>
<td>pk_void</td>
<td>Void</td>
</tr>
<tr>
<td>pk_short</td>
<td>Short</td>
</tr>
<tr>
<td>pk_long</td>
<td>Long</td>
</tr>
<tr>
<td>pk_ushort</td>
<td>Unsigned short</td>
</tr>
<tr>
<td>pk_ulong</td>
<td>Unsigned long</td>
</tr>
<tr>
<td>pk_float</td>
<td>Float</td>
</tr>
<tr>
<td>pk_double</td>
<td>Double</td>
</tr>
<tr>
<td>pk_boolean</td>
<td>Boolean</td>
</tr>
<tr>
<td>pk_char</td>
<td>Character</td>
</tr>
<tr>
<td>pk_octet</td>
<td>Octet</td>
</tr>
</tbody>
</table>
The Repository class provides access to the interface repository and is derived from the Container class, described on page 7-8.

### Include file

You should include the file corba.h when using this class.

```cpp
interface Repository : public Container {
    Contained lookup_id(in RepositoryId search_id);
    PrimitiveDef get_primitive(in CORBA::PrimitiveKind kind);
    StringDef create_string(in unsigned long bound);
    SequenceDef create_sequence(
        in unsigned long bound,
        in CORBA::IDLType_ptr element_type
    );
    ArrayDef create_array(
        in unsigned long length,
        in CORBA::IDLType_ptr element_type
    );
};
```

### Repository methods

CORBA::ArrayDef_ptr create_array(CORBA::ULong length,
                                 CORBA::IDLType_ptr element_type);

This method creates a new ArrayDef and returns a pointer to that object.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>length</td>
<td>The maximum number of elements in the array. This value must be greater than zero.</td>
</tr>
<tr>
<td>element_type</td>
<td>The IDL type of the elements stored in the array.</td>
</tr>
</tbody>
</table>
CORBA::SequenceDef_ptr create_sequence(CORBA::ULong bound, CORBA::IDLType_ptr element_type);

This method creates a new SequenceDef object and returns a pointer to that object.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bound</td>
<td>The maximum number of items in the sequence. This value must be greater than zero.</td>
</tr>
<tr>
<td>element_type</td>
<td>A pointer to the IDLType of the items stored in the sequence.</td>
</tr>
</tbody>
</table>

CORBA::StringDef_ptr create_string(CORBA::Ulong bound);

This method creates a new StringDef object and returns a pointer to that object.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bound</td>
<td>The maximum length of the string. This value must be greater than zero.</td>
</tr>
</tbody>
</table>

CORBA::StringDef_ptr create_wstring(CORBA::Ulong bound);

This method creates a new WstringDef object and returns a pointer to that object.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bound</td>
<td>The maximum length of the string. This value must be greater than zero.</td>
</tr>
</tbody>
</table>

CORBA::PrimitiveDef_ptr get_primitive(CORBA::PrimitiveKind kind);

This method returns a reference to a PrimitiveKind.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>kind</td>
<td>The reference to be returned.</td>
</tr>
</tbody>
</table>

CORBA::Contained_ptr lookup_id(const char * search_id);

This method searches for an object in the interface repository that matches the specified search id. If no match is found, a NULL value is returned.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>search_id</td>
<td>The identifier to use for the search.</td>
</tr>
</tbody>
</table>
**SequenceDef**

class **SequenceDef** : public CORBA::IDLType

The class is used to represent a sequence that is stored in the interface repository. This interface provides methods for setting and retrieving the sequence’s bound and element type.

**SequenceDef methods**

CORBA::ULong **bound**();

This method returns the bounds of the sequence.

void **bound**(CORBA::ULong **bound**);

This method sets the bound of the sequence.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>members</td>
<td>The list of members.</td>
</tr>
</tbody>
</table>

CORBA::TypeCode_ptr **element_type**();

This method returns a TypeCode representing the type of elements in this sequence.

CORBA::IDLType_ptr **element_type_def**();

This method returns the IDL type of the elements stored in this sequence.

void **element_type_def**(CORBA::IDLType_ptr **element_type_def**);

This method sets the IDL type for the elements stored in this sequence.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>element_type_def</td>
<td>The IDL type to set.</td>
</tr>
</tbody>
</table>

**StringDef**

class **StringDef** : public CORBA::IDLType

The class is used to represent a string that is stored in the interface repository. This interface provides methods for setting and retrieving the bounds of the string.

**StringDef methods**

CORBA::ULong **bound**();

This method returns the bounds of the String.
**StructDef**

```cpp
void members(CORBA::ULong bound) ;
```

This method sets the bounds of the `String`.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>members</td>
<td>The list of members.</td>
</tr>
</tbody>
</table>

**StructDef**

class `StructDef` : public `CORBA::TypedefDef`

The class is used to represent a structure that is stored in the interface repository. This class provides methods for setting and retrieving the structure’s list of members.

**StructDef methods**

```cpp
CORBA::StructMemberSeq *members() ;
```

This method returns the structure’s list of members.

```cpp
void members(CORBA::StructMemberSeq& members) ;
```

This method sets the structure’s list of members.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>members</td>
<td>The list of members.</td>
</tr>
</tbody>
</table>

**TypedefDef**

class `TypedefDef` : public `CORBA::Contained`, public `CORBA::IDLType`

This abstract base class represents a user-defined structure that is stored in the interface repository. The following interfaces all inherit from this interface:

- `AliasDef`, described on page 7-2.
- `EnumDef`, described on page 7-15.
- `EstructDef`, described on page 7-15.
- `StructDef`, described on page 7-31.
- `UnionDef`, described on page 7-32.
- `WStringDef`, described on page 7-34.

**TypeDescription**

structure `TypeDescription`

The `TypeDescription` structure contains the information that describes a type for an operation stored in the interface repository.
TypeDescription members

CORBA::String_var name
   This member represents the name of the type.

CORBA::String_var id
   This member represents the repository id of the type.

CORBA::String_var defined_in
   This member represents the name of the module or interface in which this type is defined.

CORBA::VersionSpec version
   This member represents the type’s version.

CORBA::TypeCode_var type
   This member represents the type’s IDL type.

UnionDef

class UnionDef : public CORBA::TypedefDef, public CORBA::Container

The class is used to represent a Union that is stored in the interface repository. This class provides methods for setting and retrieving the union’s list of members and discriminator type.

UnionDef methods

CORBA::TypeCode_ptr discriminator_type() ;

This method returns the TypeCode of the discriminator for the Union.

CORBA::IDLType_ptr discriminator_type_def() ;

This method returns the IDL type of the union’s discriminator.

void discriminator_type_def(CORBA::IDLType_ptr discriminator_type_def);

This method sets the IDL type of the union’s discriminator.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>discriminator_type_def</td>
<td>The list of members.</td>
</tr>
</tbody>
</table>

CORBA::UnionMemberSeq *members() ;

This method returns the union’s list of members.
void members(CORBA::UnionMembersSeq& members) ;

This method sets the union’s list of members.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>members</td>
<td>The list of members.</td>
</tr>
</tbody>
</table>

**UnionMember**

struct UnionMember

The `UnionMember` struct contains information that describes a union that is stored in the interface repository.

### UnionMember members

- CORBA::String_var `name`
  This member represents the name of the union.
- CORBA::Any `label`
  This member represents the label of the union.
- CORBA::TypeCode_var `type`
  This member represents the union’s type.
- CORBA::IDLType_var `type_def`
  This member represents the union’s IDL type.

**VersionSpec**

struct CORBA::VersionSpec

The `VersionSpec` structure contains information that describes the version of an object that is stored in the interface repository.

### VersionSpec members

- CORBA::UShort `major`
  This is the major version number.
- CORBA::UShort `minor`
  This is the minor version number.
WstringDef

This class is used to represent a Unicode string that is stored in the interface repository. It provides methods for setting and retrieving the bounds of the string.

WStringDef methods

- **CORBA::ULong bound()**;
  - This method returns the bounds of the Wstring.

- **void members(CORBA::ULong bound)**;
  - This method sets the bounds of the Wstring.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>members</td>
<td>The list of members.</td>
</tr>
</tbody>
</table>
Chapter 8

Activation interfaces and classes

This chapter describes the interfaces and classes used in the activation of object implementations. This chapter includes the following sections:

- ActivationImplDef: page 8-1
- Activator: page 8-2
- CreationImplDef: page 8-3
- ImplementationDef: page 8-6
- ImplementationStatus: page 8-8
- Include file: page 8-9

ActivationImplDef

```cpp
class ActivationImplDef : public CORBA::ImplementationDef
```

The `ActivationImplDef` class is used by the OAD in the activation and deactivation of object implementations. It is derived from `CORBA::ImplementationDef` which it expands by adding information necessary to launch an object server process.

See also

“Activator” on page 8-2, “CreationImplDef” on page 8-3, and “ImplementationDef” on page 8-6.

Include file

The `corba.h` file should be included when you use this class.
ActivationImplDef methods

**ActivationImplDef methods**

`Activator_ptr activator_obj()` const;

This method returns the object implementation’s `Activator` object.

`void activator_obj(Activator_ptr val);`

This method sets the `Activator` object for the object implementation.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>val</code></td>
<td>The new Activator object for the object implementation.</td>
</tr>
</tbody>
</table>

**Activator**

class `Activator`

The `Activator` base class is used by object servers to control the instantiation and deactivation of object implementations which they offer. You derive your own class from the Activator base class and pass it as an argument to the `BOA::impl_is_ready` method, instead of passing a reference to the object implementation.

The ORB will then use the `activate` and `deactivate` methods you provide to activate and deactivate the object.

See also “ImplementationDef” on page 8-6.

**Include file**

The `corba.h` file should be included when you use this class.

**Activator methods**

`virtual CORBA::Object_ptr activate(CORBA::ImplementationDef impl) = 0;`

You must implement this method, which should instantiate your object and provide any other processing that you want to occur when the ORB activates your object.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>impl</code></td>
<td>The <code>ImplementationDef</code> for the object being implemented, which contains the object name, interface name, and reference data for the object implementation.</td>
</tr>
</tbody>
</table>
virtual void deactivate(CORBA::Object_ptr obj, 
CORBA::ImplementationDef_ptr impl);

You may implement this method to provide clean-up processing when the ORB deactivates your object.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>obj</td>
<td>The object to be deactivated.</td>
</tr>
<tr>
<td>impl</td>
<td>The ImplementationDef associated with the object.</td>
</tr>
</tbody>
</table>

CreationImplDef

class CreationImplDef : public CORBA::ImplementationDef

The CreationImplDef class is used by the OAD in the activation and deactivation of object implementations. It is derived from CORBA::ImplementationDef which it expands by adding the information necessary to launch an object server process.

See also

“ActivationImplDef” on page 8-1 and “ImplementationDef” on page 8-6.

Include file

The corba.h file should be included when you use this class.

C++ example

The following excerpt shows how you would activate the VisiBroker for C++ application “factory_r” (located in your local directory) with one argument “ReentrantServer” and the environment LD_LIBRARY_PATH set properly for the OAD’s environment.

```
Example
path_name = "/home/developer/Project1/factory_r"
args = ['ReentrantServer']
env = ['LD_LIBRARY_PATH=/usr/ucblib:/usr/local/VisiCpp/lib']
```

It would correspond to the OAD spawning the following command:

```
Example
"/home/developer/Project1/factory_r ReentrantServer \
-Ooad_uid=<unique_id> -OactivateIOR=<oad's ior>"
```

In addition, the environment variables listed in the next section would be propagated from the OAD’s environment into that of the spawned server.
Environment variables

The following environment variables are propagated from the OAD’s environment into that of the spawned server or, if set, passed explicitly by the OAD.

- PATH
- CLASSPATH
- OSAGENT_PORT
- OSAGENT_ADDR
- VBROKER_ADM (or ORBELINE for backward compatibility)
- LD_LIBRARY_PATH as set in the CreationImplDef

All other environment variables must be registered using the env attribute in CreationImplDef. For example, if spawning a C++ implementation on Solaris, you would need to explicitly register an LD_LIBRARY_PATH in the CreationImplDef’s environment if the spawned executable requires the shared libraries.

For spawned C++ applications, the registration maps to the following command:

Syntax
```
exec-path { args1 ... argsN } -OAoad_uid=<unique_id>\n   -OAactivateIOR=<oad's ior>
```

The spawned environment contains all the specified environment variables from the implementation definition as well as definitions for PATH, CLASSPATH, OSAGENT_PORT, and OSAGENT_ADDR which are taken from the OAD’s own environment at startup. As with any OA parameter, those added by the OAD are stripped off during BOA_init and not seen by the client program. For more information, see Appendix A, “Using command-line options.”

CreationImplDef methods

```
CreationImplDef(const char *interface_name,
           const char *object_name,
           const CORBA::ReferenceData& id,
           const char *path_name,
           const CORBA::StringSequence& args,
           const CORBA::StringSequence& env,
           CORBA::Policy pol = CORBA::SHARED_SERVER);
```

This method creates an CreationImplDef object initialized with the specified parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>interface_name</td>
<td>The interface name for the object implementation.</td>
</tr>
<tr>
<td>object_name</td>
<td>The object name for the object implementation.</td>
</tr>
<tr>
<td>id</td>
<td>The reference data for the object implementation. Reference data is not interpreted by the ORB and can contain any application-specific data you desire.</td>
</tr>
</tbody>
</table>
### CreationImplDef methods

**activation_policy() const;**

This method returns the object implementation’s server policy.

**void activation_policy(CORBA::Policy p);**

This method sets the object implementation’s server policy.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>path_name</td>
<td>The path to the executable file that implements the server.</td>
</tr>
<tr>
<td>args</td>
<td>A StringSequence representing any arguments to be passed to the server.</td>
</tr>
<tr>
<td>env</td>
<td>A StringSequence representing any environment variables to be set for the server.</td>
</tr>
<tr>
<td>pol</td>
<td>The server's activation policy. Must be one of SHARED_SERVER, UNSHARED_SERVER, or SERVER_PER_METHOD. If no policy is specified, the default policy is SHARED_SERVER.</td>
</tr>
</tbody>
</table>

**CORBA::StringSequence *args() const;**

This method returns the argument list to be passed to the object implementation server at start up time.

**void args(CORBA::StringSequence& val);**

This method sets the argument list to be passed to the object implementation server at start up time.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>val</td>
<td>The new argument list.</td>
</tr>
</tbody>
</table>

**CORBA::StringSequence *env() const;**

This method returns the list of environment variables to be set for the object implementation server at start up time.

**void env(CORBA::StringSequence& val);**

This method sets the list of environment variables to be set for the object implementation server at start up time.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>val</td>
<td>The new environment variable list.</td>
</tr>
</tbody>
</table>

**const char *path_name() const;**

This method returns a string containing the path name of the executable file that implements the object.
void *path_name(const char *val);

This method sets the path name of the executable file that implements the object.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>val</td>
<td>The new path name for the object implementation.</td>
</tr>
</tbody>
</table>

static CORBA::ImplementationDef_ptr _duplicate(
    CORBA::CreationImplDef_ptr obj);

This method duplicates the specified CreationImplDef object and returns an ImplementationDef_ptr to the new object. If obj is not a valid pointer, NULL will be returned.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>obj</td>
<td>The ImplementationDef object to be duplicated.</td>
</tr>
</tbody>
</table>

static CORBA::CreationImplDef_ptr _narrow(
    CORBA::ImplementationDef_ptr ptr);

This method attempts to narrow the specified ImplementationDef pointer to a CreationImplDef pointer. If the pointer cannot be narrowed to a CreationImplDef pointer, a NULL pointer will be returned.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ptr</td>
<td>The ImplementationDef object to be narrowed.</td>
</tr>
</tbody>
</table>

static CORBA::CreationImplDef_ptr _nil();

This method returns a NULL CreationImplDef pointer that can be used for initialization purposes.

**ImplementationDef**

class CORBA::ImplementationDef

The ImplementationDef class is used in the activation and deactivation of object implementations. It contains the object name, interface name, and reference data associated with an object implementation.

See also “ActivationImplDef” on page 8-1 and “CreationImplDef” on page 8-3.

**Include file**

The corba.h file should be included when you use this class.
**ImplementationDef methods**

```cpp
ImplementationDef(const char *interface_name,
                    const char *object_name,
                    const CORBA::ReferenceData& id);
```

This method creates an `ImplementationDef` object, initialized with the specified parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>interface_name</td>
<td>The interface name for the object implementation.</td>
</tr>
<tr>
<td>object_name</td>
<td>The object name for the object implementation.</td>
</tr>
<tr>
<td>id</td>
<td>The reference data for the object implementation. Reference data is not interpreted by the ORB and can contain any application-specific data you desire.</td>
</tr>
</tbody>
</table>

```cpp
CORBA::ReferenceData_ptr id() const;
```

Returns the reference data identifier for the implementation. Reference data is not interpreted by the ORB and can contain any application-specific data you desire.

```cpp
void id(const CORBA::ReferenceData& data);
```

Sets the reference data identifier for the implementation. Reference data is not interpreted by the ORB and can contain any application-specific data you desire.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>data</td>
<td>The implementation’s reference data identifier.</td>
</tr>
</tbody>
</table>

```cpp
const char *interface_name() const;
```

This method returns a string containing the interface name of the object implementation.

```cpp
void *interface_name(const char *val);
```

This method sets the interface name for the object implementation.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>val</td>
<td>The new interface name for the object implementation.</td>
</tr>
</tbody>
</table>

```cpp
const char *object_name() const;
```

This method returns a string containing the object name of the object implementation.
void *object_name(const char *val);

This method sets the object name for the object implementation.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>val</td>
<td>The new object name for the object implementation.</td>
</tr>
</tbody>
</table>

**ImplementationStatus**

struct ImplementationStatus

*ImplementationStatus* is used to track the activation state for a server that is registered with the OAD.

```c
module Activation
{
    . . .
    struct ImplementationStatus {
        extension::CreationImplDefImpl;
        ObjectStatusList status;
    };
    . . .
};
```

**Include file**

The *corba.h* file should be included when you use this class.

**ImplementationStatus members**

CreationImplDef impl;

The *CreationImplDef* for the object implementation.

ObjectStatusList status;

Represents a list of status information for each object offered by the server.

See page 8-13 for information on the *ObjectStatusList* class.

**OAD**

The OAD interface provides access to the OAD (Object Activation Daemon). It is used by the administration tools for listing, registering, and unregistering objects. It can also be used by client code for programmatic administration of the OAD.
interface OAD {
    CreationImplDef create_CreationImplDef();
    Object reg_implementation(in CreationImplDef impl)
        raises(Activation::DuplicateEntry, Activation::InvalidPath);
    CreationImplDef get_implementation(in CORBA::RepositoryId repId,
        in string object_name)
        raises(Activation::NotRegistered);
    void change_implementation(in CreationImplDef old_info,
        in CreationImplDef new_info)
        raises(Activation::NotRegistered, Activation::InvalidPath,
            Activation::IsActive);
    attribute boolean destroy_on_unregister;
    void unreg_implementation(in CORBA::RepositoryId repId,
        in string object_name)
        raises(Activation::NotRegistered);
    void unreg_interface(in CORBA::RepositoryId repId)
        raises(Activation::NotRegistered);
    void unregister_all();
    Activation::ImplementationStatus get_status(in CORBA::RepositoryId repId,
        in string object_name)
        raises(Activation::NotRegistered);
    Activation::ImplStatusList get_status_interface(
        in CORBA::RepositoryId repId)
        raises(Activation::NotRegistered);
    Activation::ImplStatusList get_status_all();
};

Include file

The corba.h file should be included when you use this class.

OAD methods

void change_implementation(CORBA::CreationImplDef old_info,
    CORBA::CreationImplDef new_info)

This method dynamically changes an object’s implementation. You can use this method to change the registration’s activation policy, path name, argument settings, and environment settings.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>old_info</td>
<td>The information you want to change.</td>
</tr>
<tr>
<td>new_info</td>
<td>The information to replace the old info.</td>
</tr>
</tbody>
</table>
This method throws the following exceptions.

<table>
<thead>
<tr>
<th>Exception</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NotRegistered</td>
<td>The object you specify is not registered. You must specify a registered object.</td>
</tr>
<tr>
<td>IsActive</td>
<td>The object implementation is currently running. Deactivate the object and then try to change its information.</td>
</tr>
</tbody>
</table>

**Caution**

You cannot change information for a currently active implementation. Be sure to exercise caution when changing an object’s implementation name and object name with this method. Doing so will prevent client applications from locating the object with the old name.

```
CreationImplDef_ptr create_CreationImplDef();
```

Returns an instance of a `CreationImplDef` object. You can then set its attributes as explained in “CreationImplDef” on page 8-3.

```
void destroy_on_unregister(CORBA::Boolean val);
```

Sets the `destroy_on_unregister` attribute for the OAD.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>val</td>
<td>If set to TRUE, any active implementations are shut down when they are unregistered. Otherwise, they will not be shutdown when unregistered.</td>
</tr>
</tbody>
</table>

```
CORBA::Boolean destroy_on_unregister();
```

Returns the setting for the `destroy_on_unregister` attribute for an implementation. If the attribute is set to TRUE, any active implementations are shut down when unregistered.

```
CORBA::CreationImplDef_ptr get_implementation(const char *repId, const char *object_name)
```

This method retrieves information about implementations registered for the specified repository identifier and object name.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>repId</td>
<td>The repository identifier.</td>
</tr>
<tr>
<td>object_name</td>
<td>The object name.</td>
</tr>
</tbody>
</table>

This method throws the following exceptions.

<table>
<thead>
<tr>
<th>Exception</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NotRegistered</td>
<td>The object you specify is not registered. You must specify a registered object.</td>
</tr>
</tbody>
</table>
OAD methods

ImplementationStatus *get_status(const char *repId,
const char *object_name)

This method retrieves the status information about implementations
registered for the specified repository identifier and object name.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>repId</td>
<td>The repository identifier.</td>
</tr>
<tr>
<td>object_name</td>
<td>The object name.</td>
</tr>
</tbody>
</table>

ImplStatusList *get_status_all()

Returns an ImplStatusList containing the status information for all
implementations.

ImplStatusList *get_status_interface(
const char *repId)

This method gets the status information about implementations registered
for the specified repository identifier.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>repId</td>
<td>The repository identifier.</td>
</tr>
</tbody>
</table>

This method throws the following exceptions.

<table>
<thead>
<tr>
<th>Exception</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NotRegistered</td>
<td>The object you specify is not registered. You must specify a registered object.</td>
</tr>
</tbody>
</table>

CORBA::Object reg_implementation(
CORBA::CreationImplDef_ptr impl)

This method registers an implementation with the OAD and the VisiBroker
directory service.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>impl</td>
<td>The instance of CreationImplDef.</td>
</tr>
</tbody>
</table>

This method throws the following exceptions.

<table>
<thead>
<tr>
<th>Exception</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DuplicateEntry</td>
<td>The object you specify is a duplicate entry. You must specify an unregistered object.</td>
</tr>
</tbody>
</table>
void unreg_implementation(const char *repId, const char *object_name)

This method unregisters implementations by repository identifier and object name. If the destroy_on_unregister attribute is set to true, this method terminates all processes currently implementing the repository identifier and object name that is specified.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>repId</td>
<td>The repository identifier.</td>
</tr>
<tr>
<td>object_name</td>
<td>The object name.</td>
</tr>
</tbody>
</table>

This method throws the following exceptions.

<table>
<thead>
<tr>
<th>Exception</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NotRegistered</td>
<td>The object you specify is not registered. You must specify a registered object.</td>
</tr>
</tbody>
</table>

void unreg_interface(const char *repId)

This method unregisters all implementations for a repository identifier. If the destroy_on_unregister attribute is set to true, this method terminates all processes currently implementing the repository identifier specified.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>repId</td>
<td>The repository identifier.</td>
</tr>
</tbody>
</table>

This method throws the following exceptions.

<table>
<thead>
<tr>
<th>Exception</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NotRegistered</td>
<td>The object you specify is not registered. You must specify a registered object.</td>
</tr>
</tbody>
</table>

void unregister_all()

This method unregisters all implementations. Unless the attribute destroy_on_unregister is set to true, all active implementations continue to execute.

ObjectStatus

struct ObjectStatus

This structure is used to represent information about a particular object offered by an object implementation that is registered with the OAD. This structure is returned by the ObjectStatusList class, described on page 8-13.
module Activation
{
    . . .
    struct ObjectStatus {
        long unique_id;
        State activation_state;
        Object objRef;
    }
    . . .
};

ObjectStatus members

CORBA::Long unique_id;
    A unique identifier for the object.
State activation_state;
    The object’s current activation state, one of these values:
        • ACTIVE
        • INACTIVE
        • WAITING_FOR_ACTIVATION
CORBA::Object objRef;
    The object whose state is represented in the structure.

ObjectStatusList

class ObjectStatusList

This class implements a list of ObjectStatus structures and is used to represent
information about the objects offered by a server.

See also
“ObjectStatus” on page 8-12.

Include file

The corba.h file should be included when you use this class.
### ObjectStatusList methods

**void length(CORBA::ULong len);**

Sets the length of the list.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>len</td>
<td>The length of the list.</td>
</tr>
</tbody>
</table>

**CORBA::ULong length() const;**

Returns the length of the list.

**CORBA::ULong maximum() const;**

Returns the maximum length of the list.

**ObjectStatus& operator[](CORBA::ULong index);**

Returns the ObjectStatus structure from the list with the specified index.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>index</td>
<td>The index of the item in the list to be returned. This index is zero-based.</td>
</tr>
</tbody>
</table>

### StringSequence

**class CORBA::StringSequence**

The StringSequence class is used to contain a list of arguments or environment variables associated with a CreationImplDef object, which is described on page 8-3.

**Include file**

The corba.h file should be included when you use this class.

### StringSequence methods

**CORBA::StringSequence(CORBA::ULong max = 0);**

This method creates a StringSequence object with the specified length.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>max</td>
<td>The maximum number of arguments in the list. The default length is 0.</td>
</tr>
</tbody>
</table>
CORBA::StringSequence(CORBA::ULong max, CORBA::ULong length, char **data, CORBA::Boolean release = 0);

This method creates a StringSequence object with the specified parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>max</td>
<td>The maximum number of arguments in the list.</td>
</tr>
<tr>
<td>length</td>
<td>The length of the sequence.</td>
</tr>
<tr>
<td>data</td>
<td>The strings that will make up the sequence.</td>
</tr>
<tr>
<td>release</td>
<td>If set to 1, all memory associated with the list will be released when this object is destroyed.</td>
</tr>
</tbody>
</table>

~CORBA::StringSequence();

This method destroys this object.

Methods

static char **allocbuf(CORBA::ULong nelemes);

This method allocates memory to accommodate the number of list elements specified.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nelemes</td>
<td>The number of elements in the list.</td>
</tr>
</tbody>
</table>

CORBA::ULong compare(const CORBA::StringSequence& seq1, const CORBA::StringSequence& seq2);

This method compares two StringSequence objects and returns 0 if they are equal; otherwise a non-zero value is returned.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>seq1</td>
<td>The first object to be compared.</td>
</tr>
<tr>
<td>seq2</td>
<td>The second object to be compared.</td>
</tr>
</tbody>
</table>

static void freebuf(char **data);

This method frees the memory associated with the specified pointer.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>data</td>
<td>The list memory to be freed.</td>
</tr>
</tbody>
</table>

static void freebuf elems(char **data, CORBA::ULong nelemes);

This method allocates memory to accommodate the number of list elements specified.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>data</td>
<td>The list memory to be freed.</td>
</tr>
<tr>
<td>nelemes</td>
<td>The number of elements.</td>
</tr>
</tbody>
</table>
Methods

CORBA::ULong hash(CORBA::StringSequence&);
This returns a hash value for the specified object.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>StringSequence</td>
<td>The StringSequence for which a hash value is returned.</td>
</tr>
</tbody>
</table>

CORBA::ULong length() const;
This method returns the number of elements in the sequence.

void length(CORBA::ULong);
This method sets the number of elements in the sequence.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ULong</td>
<td>The new length.</td>
</tr>
</tbody>
</table>

CORBA::ULong maximum() const;
This method returns the number of arguments in the list.

CORBA::StringSequence& operator=(const CORBA::StringSequence& seq);
This operator allows a StringSequence to be copied through assignment.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>seq</td>
<td>The object to be copied.</td>
</tr>
</tbody>
</table>

CORBA::StringSequence& operator[](const CORBA::ULong index);
This operator allows arguments within a StringSequence to be accessed with an index.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>index</td>
<td>The zero-based index of the desired string sequence.</td>
</tr>
</tbody>
</table>

static void _release(CORBA::StringSequence* ptr);
This method releases the specified StringSequence object.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>obj</td>
<td>The ImplementationDef object to be duplicated.</td>
</tr>
</tbody>
</table>
This chapter describes the VisiBroker for C++ Event Handler interfaces and classes. Included in this chapter are the following sections:

- ClientEventHandler page 9-1
- ConnectionInfo page 9-3
- HandlerRegistry page 9-4
- ImplEventHandler page 9-6

**ClientEventHandler**

This base class allows you to derive your own client event handler objects which provide the methods which the ORB will invoke when one of the following events occurs:

- A bind to an object implementation succeeds.
- A bind to an object implementation fails.
- An object server aborts.
- A re-bind to an object implementation succeeds.
- A re-bind to an object implementation fails.

You register your client event handler object using the HandlerRegistry methods described on page 9-4. Client event handlers may be registered to handle events for all objects the client uses or to handle events for just one object.

See also “ImplEventHandler” on page 9-6
ClientEventHander methods

virtual void bind_failed(CORBA::Object_ptr obj)

This method provides the processing you wish to occur when a bind operation fails.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>obj</td>
<td>The object to which the bind attempt failed.</td>
</tr>
</tbody>
</table>

virtual void bind_succeeded(CORBA::Object_ptr obj, CORBA::ConnectionInfo& info)

This method provides the processing you wish to occur when a bind operation succeeds.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>obj</td>
<td>The object to which the client is now bound.</td>
</tr>
<tr>
<td>info</td>
<td>The connection information for this bound object.</td>
</tr>
</tbody>
</table>

virtual void rebind_succeeded(CORBA::Object_ptr obj, CORBA::ConnectionInfo& info)

This method provides the processing you wish to occur when a re-bind operation succeeds. A re-bind occurs when a connection to an object is broken and automatically re-established by the ORB to another, comparable object.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>obj</td>
<td>The object to which the client is now bound.</td>
</tr>
<tr>
<td>info</td>
<td>The connection information for this bound object.</td>
</tr>
</tbody>
</table>

virtual void rebind_failed(CORBA::Object_ptr obj)

This method provides the processing you wish to occur when a re-bind operation fails. A re-bind occurs when a connection to an object is broken and all attempts to automatically re-established a connection to another, comparable object have failed.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>obj</td>
<td>The object to which the re-bind attempt failed.</td>
</tr>
</tbody>
</table>
virtual void server_aborted(CORBA::Object_ptr obj)

You implement this method to provide the processing you wish to occur when an object server aborts.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>obj</td>
<td>The object to which the client was bound.</td>
</tr>
</tbody>
</table>

static CORBA::ClientEventHandler_ptr _duplicate(CORBA::ClientEventHandler_ptr ptr);

This static method increments the reference count for the specified object and then returns a pointer to it.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ptr</td>
<td>The object to be duplicated.</td>
</tr>
</tbody>
</table>

static CORBA::ClientEventHandler_ptr _nil();

This static method returns a NULL pointer that can be used for initialization purposes.

static void _release(CORBA::ClientEventHandler_ptr ptr);

This static method decrements the reference count for the specified object. If the count has reached zero, all memory managed by the object is released and the object is deleted.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ptr</td>
<td>The object to be released.</td>
</tr>
</tbody>
</table>

ConnectionInfo

struct ConnectionInfo

This structure is used to contain the details of a connection between a client and a server.

ConnectionInfo members

CORBA::String_var hostname

This member contains the name of the host where the client or server is located.

CORBA::UShort port

This member contains the port number being used by the client or server.
**HandlerRegistry**

This static class provides the methods for registering and unregistering both client and implementation event handlers.

*See also*  
“ClientEventHandler” on page 9-1 and “ImplEventHandler” on page 9-6.

**HandlerRegistry constructor**

static HandlerRegistry_ptr instance()  

There is no constructor for this class. You should use this static method to obtain a pointer to the HandlerRegistry. You can then use the pointer to invoke the methods offered by the HandlerRegistry.

**Client-side methods**

**void reg_glob_client_handler(CORBA::ClientEventHandle_ptr handler)**  

This method registers the specified ClientEventHandler for the all client objects. If a global event handler has already been registered for this client, a HandlerExists exception is raised.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handler</td>
<td>The client-side, global handler being registered.</td>
</tr>
</tbody>
</table>

**void reg_obj_client_handler(CORBA::Object_ptr obj, ClientEventHandler_ptr handler)**  

This method registers the specified ClientEventHandler for the specified object. If an event handler has already been registered for this object, a HandlerExists exception is raised. If the object reference provided is not valid, an InvalidObject exception is raised.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>obj</td>
<td>The specific object for which the handler is being registered.</td>
</tr>
<tr>
<td>handler</td>
<td>The client event handler to be registered.</td>
</tr>
</tbody>
</table>
**Implementation-side methods**

```c
void unreg_glob_client_handler()
```

This method unregisters the global ClientEventHandler previously registered for this client. If an event handler has not been registered for this client, a NoHandler exception is raised.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>obj</td>
<td>The object for which the handler is to be unregistered.</td>
</tr>
</tbody>
</table>

```c
void unreg_obj_client_handler(CORBA::Object_ptr obj)
```

This method unregisters the ClientEventHandler previously registered for the specified client object. If no event handler has been registered for the object, a NoHandler exception is raised. If the object reference provided is not valid, an InvalidObject exception is raised.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>obj</td>
<td>The object for which the handler is to be unregistered.</td>
</tr>
</tbody>
</table>

**Implementation-side methods**

```c
void reg_glob_impl_handler(CORBA::ImplEventHandler_ptr obj)
```

This method registers the specified ImplEventHandler for the all implementation objects offered by this server. If a global event handler has already been registered for this server, a HandlerExists exception is raised.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>obj</td>
<td>The server-side, global handler being registered.</td>
</tr>
</tbody>
</table>

```c
void reg_obj_impl_handler(CORBA::Object_ptr obj, ImplEventHandler_ptr handler)
```

This method registers the specified ImplEventHandler for the specified object. If an event handler has already been registered for this object, a HandlerExists exception is raised. If the object reference provided is not valid, an InvalidObject exception is raised.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>obj</td>
<td>The specific object for which the handler is being registered.</td>
</tr>
<tr>
<td>handler</td>
<td>The implementation event handler to be registered.</td>
</tr>
</tbody>
</table>
implEventHandler

void unreg_glob_impl_handler()

This method unregisters the global implEventHandler previously registered for this server. If an event handler has not been registered for this server, a NoHandler exception is raised.

void unreg_obj_impl_handler(CORBA::Object_ptr obj)

This method unregisters the ImplEventHandler previously registered for the specified implementation object. If no event handler has been registered for the object, a NoHandler exception is raised. If the object reference provided is not valid, an InvalidObject exception is raised.

ImplEventHandler

This base class allows you to derive your own implementation event handler object that provides the methods the ORB will invoke when one of the following events occurs:

• A bind request for this object implementation is received.
• An un-bind request for this object implementation is received.
• A previously connected client application has aborted.
• An operation request for this object implementation is received.
• An operation request for this object has been completed.

You register your implementation event handler object using the HandlerRegistry methods described on page 9-4. Implementation event handlers may be registered to handle events for all objects the server offers or to handle events for just one object implementation.

See also “ClientEventHandler” on page 9-1

ImplEventHandler methods

virtual void bind(const ConnectionInfo& info,
  CORBA::Principal_ptr ptr,
  CORBA::Object_ptr obj);

This method provides the processing you wish to occur when a bind request is received from a client application. This method will be invoked before the bind has completed.
**implEventHandler methods**

**Event Handler interfaces and classes**

```cpp
virtual void client_aborted(const ConnectionInfo& info, CORBA::Object_ptr obj);
```

This method provides the processing you wish to occur when a client that was previously bound to this server has aborted.

**Parameter**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>info</td>
<td>The object to which the client is now bound.</td>
</tr>
<tr>
<td>ptr</td>
<td>The Principal for the client making the operation request.</td>
</tr>
<tr>
<td>obj</td>
<td>The object to which the client is now bound.</td>
</tr>
</tbody>
</table>

```cpp
virtual void method_exception(const ConnectionInfo& info, const char *oper, CORBA::Object_ptr target, CORBA::Environment& env);
```

This method provides the processing you wish to occur if an exception is raised during an operation request but before the exception has been reflected to the client application.

**Parameter**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>info</td>
<td>The connection information for client.</td>
</tr>
<tr>
<td>obj</td>
<td>The object to which the client was bound.</td>
</tr>
</tbody>
</table>

```cpp
virtual void post_method(CORBA::ConnectionInfo& info, CORBA::Principal_ptr ptr, const char *name, CORBA::Object_ptr obj);
```

This method provides the processing you wish to occur after an operation request response has been processed but before the results are sent to the client application.

**Parameter**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>info</td>
<td>The connection information for this bound object.</td>
</tr>
<tr>
<td>oper</td>
<td>The name of the operation being invoked by the client.</td>
</tr>
<tr>
<td>target</td>
<td>The object to which the client is bound.</td>
</tr>
<tr>
<td>env</td>
<td>The Environment object for the client making the operation request.</td>
</tr>
<tr>
<td>ptr</td>
<td>The Principal for the client making the operation request.</td>
</tr>
<tr>
<td>name</td>
<td>The name of the operation being invoked by the client.</td>
</tr>
<tr>
<td>obj</td>
<td>The object to which the client is now bound.</td>
</tr>
</tbody>
</table>
**ImplEventHandler methods**

```cpp
virtual void pre_method(CORBA::ConnectionInfo& info,
CORBA::Principal_ptr ptr,
const char * name,
CORBA::Object_ptr obj);
```

This method provides the processing you wish to occur before an operation request received from a client application is processed.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>info</td>
<td>The connection information for this bound object.</td>
</tr>
<tr>
<td>ptr</td>
<td>The Principal for the client making the operation request.</td>
</tr>
<tr>
<td>name</td>
<td>The name of the operation being invoked by the client.</td>
</tr>
<tr>
<td>obj</td>
<td>The object to which the client is now bound.</td>
</tr>
</tbody>
</table>

```cpp
virtual void unbind(const ConnectionInfo& info,
CORBA::Object_ptr obj);
```

This method provides the processing you wish to occur when a client application releases its reference to your object. This method will be invoked before the `CORBA::Object::_release` method completes.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>info</td>
<td>The connection information for the client.</td>
</tr>
<tr>
<td>obj</td>
<td>The object to which the client is now bound.</td>
</tr>
</tbody>
</table>

```cpp
static CORBA::ImplEventHandler_ptr _duplicate(CORBA::ImplEventHandler_ptr ptr);
```

This static method increments the reference count for the specified object and then returns a pointer to it.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ptr</td>
<td>The object to be duplicated.</td>
</tr>
</tbody>
</table>

```cpp
static CORBA::ImplEventHandler_ptr _nil();
```

This static method returns a NULL pointer that can be used for initialization purposes.

```cpp
static void _release(CORBA::ImplEventHandler_ptr ptr);
```

This static method decrements the reference count for the specified object. If the count has reached zero, all memory managed by the object is released and the object is deleted.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ptr</td>
<td>The object to be released.</td>
</tr>
</tbody>
</table>
Interceptor and object wrapper interfaces and classes

This chapter describes the interfaces and classes that you can use to create and use interceptors and object wrappers. Read the chapters on interceptors and object wrappers in the VisiBroker for C++ Programmer’s Guide before using these interfaces. This chapter includes the following sections:

VISBindInterceptor page 10-2
VISChain BindInterceptor page 10-5
VISChainClientInterceptorFactory page 10-6
VISChainServerInterceptorFactory page 10-7
VISClientInterceptor page 10-7
VISClientInterceptorFactory page 10-11
VISClosure page 10-12
VISClosureData page 10-12
VISObjectWrapper::ChainUntypedObjectWrapperFactory page 10-13
VISObjectWrapper::UntypedObjectWrapper page 10-15
VISObjectWrapper::UntypedObjectWrapperFactory page 10-16
VISServerInterceptor page 10-17
VISServerInterceptorFactory page 10-22
VISBindInterceptor

class VISBindInterceptor : public VISResource

You can use this class to derive your own interceptor for handling bind and rebind events for a client or server application. Use the VISInit class, described in “VISInit” on page 14-1, to derive a class that will register your interceptor with the ORB.

This class also provides several static methods for testing and manipulating VISBindInterceptor pointers.

Include file

The vinter.h file should be included when you use this class.

VISBindInterceptor constructors/destructors

VISBindInterceptor();
    This is the default constructor.

virtual ~VISBindInterceptor();
    This is the default destructor.

VISBindInterceptor methods

virtual CORBA::Boolean bind(IOP::IOR& ior, CORBA::Object_ptr target, VISClosure& closure);

This method will be invoked by the ORB when the application for which this interceptor is registered is about to bind to an object. Your implementation of this method can provide features such as measuring the time required to complete a bind, printing a diagnostic message, or modifying the IOR used by the ORB. If the IOR is modified, it will only be used if this method returns TRUE.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ior</td>
<td>An Interoperable Object Reference for the object. Modification of the IOR by this method is allowed.</td>
</tr>
<tr>
<td>target</td>
<td>The object to which the application is being bound.</td>
</tr>
<tr>
<td>closure</td>
<td>May contain data saved by one interceptor method that can be retrieved later by another interceptor method.</td>
</tr>
</tbody>
</table>
VISBindInterceptor methods

virtual CORBA::Boolean bind_failed(IOR&, CORBA::Object_ptr target, VISClosure& closure);

This method will be invoked by the ORB when an attempt to bind to an object has failed. Your implementation of this method can log a failure message, attempt to bind to a different object, or modify the IOR used by the ORB. If the IOR is modified, it will only be used if this method returns TRUE.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ior</td>
<td>An Interoperable Object Reference for the object. Modification of the IOR by this method is allowed.</td>
</tr>
<tr>
<td>target</td>
<td>The object to which the application was attempting to bind.</td>
</tr>
<tr>
<td>closure</td>
<td>May contain data saved by one interceptor method that can be retrieved later by another interceptor method.</td>
</tr>
</tbody>
</table>

virtual void bind_succeeded(IOR&, CORBA::Object_ptr target, int fd, VISClosure& closure);

This method will be invoked by the ORB when a bind has successfully completed. Your implementation may simply log a message or measure the elapsed time used by the bind request.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ior</td>
<td>An Interoperable Object Reference for the object. Modification of the IOR by this method is not allowed.</td>
</tr>
<tr>
<td>target</td>
<td>The object to which the application was attempting to bind.</td>
</tr>
<tr>
<td>fd</td>
<td>The file descriptor associated with the bound object.</td>
</tr>
<tr>
<td>closure</td>
<td>May contain data saved by one interceptor method that can be retrieved later by another interceptor method.</td>
</tr>
</tbody>
</table>

virtual void exception_occurred(IOR&, CORBA::Object_ptr target, CORBA::Environment& env, VISClosure& closure);

This method is invoked by the ORB if an exception is raised by an interceptor method.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ior</td>
<td>An Interoperable Object Reference for the object. Modification of the IOR by this method is not allowed.</td>
</tr>
<tr>
<td>target</td>
<td>The object to which the application was attempting to bind.</td>
</tr>
<tr>
<td>env</td>
<td>Contains the details of the exception that was raised.</td>
</tr>
<tr>
<td>closure</td>
<td>May contain data saved by one interceptor method that can be retrieved later by another interceptor method.</td>
</tr>
</tbody>
</table>

static VISBindInterceptor *instance();

This method returns a pointer to this object.
VISBindInterceptor methods

static CORBA::Boolean is_nil(VISBindInterceptor i);

This method returns TRUE if the specified VISBindInterceptor pointer is NULL, otherwise FALSE is returned.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>The VISBindInterceptor pointer to be tested.</td>
</tr>
</tbody>
</table>

virtual CORBA::Boolean rebind(IOP::IOR& ior, CORBA::Object_ptr target, VISClosure& closure);

This method is invoked by the ORB when rebinding is enabled and an object reference is found to no longer be valid. Your implementation of this method can override the default rebind() behavior, returning a modified IOR for the ORB to use. If the IOR is modified, it will only be used if this method returns TRUE.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ior</td>
<td>An Interoperable Object Reference for the object. Modification of the IOR by this method is allowed.</td>
</tr>
<tr>
<td>target</td>
<td>The object to which the application is attempting to re-bind.</td>
</tr>
<tr>
<td>closure</td>
<td>May contain data saved by one interceptor method that can be retrieved later by another interceptor method.</td>
</tr>
</tbody>
</table>

virtual CORBA::Boolean rebind_failed(IOP::IOR& ior, CORBA::Object_ptr target, VISClosure& closure);

This method is invoked by the ORB when an attempt to rebind to an object has failed. Your implementation of this method can attempt to rebind() again or return a modified IOR for the ORB to use. If the IOR is modified, it will only be used if this method returns TRUE.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ior</td>
<td>An Interoperable Object Reference for the object. Modification of the IOR by this method is allowed.</td>
</tr>
<tr>
<td>target</td>
<td>The object to which the application was attempting to re-bind.</td>
</tr>
<tr>
<td>closure</td>
<td>May contain data saved by one interceptor method that can be retrieved later by another interceptor method.</td>
</tr>
</tbody>
</table>

virtual void rebind_succeeded(const IOP::IOR& ior, CORBA::Object_ptr target, int fd, VISClosure& closure);

This method is invoked by the ORB when a rebind attempt succeeds. Your implementation may simply log a message or measure the elapsed time used by the re-bind process.
### VISChainBindInterceptor

**VISBindInterceptor _duplicate(VISBindInterceptor i);**

This method duplicates the specified VISBindInterceptor pointer, increments the object’s reference count, and returns a pointer to the duplicated object.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ior</td>
<td>An Interoperable Object Reference for the object. Modification of the IOR by this method is not allowed.</td>
</tr>
<tr>
<td>target</td>
<td>The object to which the application was attempting to re-bind.</td>
</tr>
<tr>
<td>fd</td>
<td>The file descriptor associated with the re-bound object.</td>
</tr>
<tr>
<td>closure</td>
<td>May contain data saved by one interceptor method that can be retrieved later by another interceptor method.</td>
</tr>
</tbody>
</table>

**VISChainBindInterceptor**

class **VISChainBindInterceptor** : public VISBindInterceptor

You can use this class when your client application wishes to register more than one VISBindInterceptor for a given object. When the ORB prepares to invoke a VISBindInterceptor method, each interceptor’s method will be called in the order in which the interceptors were registered.

This class provides methods for adding or removing interceptor objects from the chain.

**Include file**

The **vcinter.h** file should be included when you use this class.

**VISChainBindInterceptor destructor**

~VISBindInterceptor();

This is the default constructor.

**VISChainBindInterceptor methods**

static void **add(VISBindInterceptor_ptr i);**

Adds the specified VISBindInterceptor to the chain.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>The bind interceptor to be added.</td>
</tr>
</tbody>
</table>
VISChainClientInterceptorFactory

static VISChainBindInterceptor * get_instance();

Returns a pointer to this object.

static void remove(VISBindInterceptor_ptr i);

Removes the specified VISBindInterceptor object from the chain.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>The bind interceptor to be removed.</td>
</tr>
</tbody>
</table>

VISChainClientInterceptorFactory

class VISChainClientInterceptorFactory : public VISClientInterceptorFactory

You can use this class when your client application wishes to register more than one VISClientInterceptor for a given object. When the ORB prepares to invoke a VISClientInterceptor method, each client interceptor’s method will be called in the order in which the interceptors were registered.

This class provides methods for adding or removing interceptor objects to or from the chain.

Include file

The vcinter.h file should be included when you use this class.

VISChainClientInterceptorFactory destructor

~VISClientInterceptorFactory();

This is the default constructor.

VISChainClientInterceptorFactory methods

static void add(VISClientInterceptorFactory * interp);

Adds the specified VISClientInterceptorFactory to the chain.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>interp</td>
<td>The bind interceptor to be added.</td>
</tr>
</tbody>
</table>

static void remove(VISClientInterceptorFactory * interp);

Removes the specified VISClientInterceptorFactory object from the chain.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>interp</td>
<td>The bind interceptor to be removed.</td>
</tr>
</tbody>
</table>
VISChainServerInterceptorFactory

class VISChainServerInterceptorFactory : public VISServerInterceptorFactory

You can use this class when your client application wishes to register more than one VISServerInterceptor for a given object. It also allows your server implementation to register more than one VISServerInterceptor for a given connection. When the ORB prepares to invoke a VISServerInterceptor method, each interceptor’s method will be called in the order in which the interceptors were registered.

This class provides methods for adding or removing interceptor objects to or from the chain.

Include file

The vcinter.h file should be included when you use this class.

VISChainServerInterceptorFactory destructor

~VISChainServerInterceptorFactory();

This is the default constructor.

VISChainServerInterceptorFactory methods

static void add(VISServerInterceptorFactory *intercep);

Adds the specified VISServerInterceptorFactory to the chain.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>intercept</td>
<td>The interceptor to be added.</td>
</tr>
</tbody>
</table>

static void remove(VISServerInterceptorFactory *i);

Removes the specified VISServerInterceptorFactory object from the chain.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>intercept</td>
<td>The interceptor to be removed.</td>
</tr>
</tbody>
</table>

VISClientInterceptor

class VISClientInterceptor : public VISResource

You use this class to derive your own client-side interceptor, providing implementations for those methods that you wish to override. The methods defined in your derived class will be invoked by the ORB during the
preparation or sending of an operation request, during the receipt of a reply message, or if an exception is raised.

Use the VISInit class, described on page 14-1, to derive a class that will register your interceptor with the ORB.

This class also provides several static methods for testing and manipulating VISClientInterceptor pointers.

Include file

The vinter.h file should be included when you use this class.

VISClientInterceptor constructors/destructors

VISClientInterceptor();

This is the default constructor.

~VISClientInterceptor();

This is the default destructor.

VISClientInterceptor methods

virtual void exception_occurred(CORBA::ULong req_id, CORBA_Environment& env, VISClosure& closure);

This method is invoked by the ORB whenever a method offered by this interceptor throws an exception. Your implementation of this method may provide additional processing, such as message logging or recovery processing.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>req_id</td>
<td>The operation request identifier.</td>
</tr>
<tr>
<td>env</td>
<td>Contains information on the exception that was raised.</td>
</tr>
<tr>
<td>closure</td>
<td>May contain data saved by one interceptor method that can be retrieved later by another interceptor method.</td>
</tr>
</tbody>
</table>

static CORBA::Boolean is_nil(VISClientInterceptor i);

This method returns TRUE if the specified VISClientInterceptor pointer is NULL, otherwise FALSE is returned.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>The VISClientInterceptor pointer to be tested.</td>
</tr>
</tbody>
</table>
virtual void prepare_request(GIOP::RequestHeader& hdr, VISClosure& closure);

This method is invoked by the ORB when it is preparing a request. Your implementation of this method may alter the RequestHeader to add service contexts.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hdr</td>
<td>The request header, which identifies the object, operation request, and other request information.</td>
</tr>
<tr>
<td>closure</td>
<td>May contain data saved by one interceptor method that can be retrieved later by another interceptor method.</td>
</tr>
</tbody>
</table>

virtual void receive_reply(const GIOP::RequestHeader& hdr, CORBA_MarshalInBuffer *buf, const CORBA_Exception *excep, VISClosure& closure);

This method is invoked by the ORB when it receives a reply to an operation request. This method is allowed to change the contents of buf, allowing your implementation to provide decryption or translation processing.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hdr</td>
<td>The request header, which identifies the object, operation request, and other request information. The contents of the header may be modified by this method.</td>
</tr>
<tr>
<td>buf</td>
<td>The buffer containing the reply to the operation request.</td>
</tr>
<tr>
<td>excep</td>
<td>Represents any exception that may have occurred while the object implementation-side was processing the operation request.</td>
</tr>
<tr>
<td>closure</td>
<td>May contain data saved by one interceptor method that can be retrieved later by another interceptor method.</td>
</tr>
</tbody>
</table>

virtual void receive_reply_failed(CORBA::ULong req_id, const CORBA_Exception& excep, VISClosure& closure);

This method is invoked by the ORB when an attempt to receive a reply to an operation request fails, usually due to a communications failure or a time-out interval being exceeded. The excep describes the reason for the failure. User exceptions will be reflected in the MarshalInBuffer passed to the receive_reply method.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>req_id</td>
<td>Identifies the operation that was requested.</td>
</tr>
<tr>
<td>excep</td>
<td>Describes the nature of the failure.</td>
</tr>
<tr>
<td>closure</td>
<td>May contain data saved by one interceptor method that can be retrieved later by another interceptor method.</td>
</tr>
</tbody>
</table>
**VISClientInterceptor methods**

```cpp
virtual CORBA.MarshalOutBuffer *send_request(const GIOP::RequestHeader& hdr,
                              CORBA.MarshalOutBuffer *buf,
                              VISClosure& closure);
```

This method is invoked by the ORB before it sends an operation request. Your implementation of this method may provide additional processing, such as performance monitoring, message logging, or message encryption. If this method returns a non-null `MarshalOutBuffer`, that buffer will be sent instead of `buf`.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>hdr</code></td>
<td>The request header, which identifies the object, operation request, and other request information.</td>
</tr>
<tr>
<td><code>buf</code></td>
<td>The buffer containing the operation request.</td>
</tr>
<tr>
<td><code>closure</code></td>
<td>May contain data saved by one interceptor method that can be retrieved later by another interceptor method.</td>
</tr>
</tbody>
</table>

```cpp
virtual void send_request_failed(const GIOP::RequestHeader& hdr,
                                 const CORBA_Exception& excep,
                                 VISClosure& closure);
```

This method is invoked by the ORB when an attempt to send an operation request fails. The failure may be related to a communication problem. The `excep` describes the reason for the failure.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>hdr</code></td>
<td>The request header, which identifies the object, operation request, and other request information.</td>
</tr>
<tr>
<td><code>excep</code></td>
<td>Describes the nature of the failure.</td>
</tr>
<tr>
<td><code>closure</code></td>
<td>May contain data saved by one interceptor method that can be retrieved later by another interceptor method.</td>
</tr>
</tbody>
</table>

```cpp
virtual void send_request_succeeded(const GIOP::RequestHeader& hdr,
                                    VISClosure& closure);
```

This method is invoked by the ORB when an operation request has been successfully sent. This method is for notification purposes only.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>hdr</code></td>
<td>The request header, which identifies the object, operation request, and other request information.</td>
</tr>
<tr>
<td><code>closure</code></td>
<td>May contain data saved by one interceptor method that can be retrieved later by another interceptor method.</td>
</tr>
</tbody>
</table>
VISClientInterceptorFactory

static VISClientInterceptor _duplicate(VISClientInterceptor i);

This method duplicates the specified VISClientInterceptor pointer, increments the object’s reference count, and returns a pointer to the duplicated object.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>The VISClientInterceptor pointer to be duplicated.</td>
</tr>
</tbody>
</table>

VISClientInterceptorFactory

class VISClientInterceptorFactory

You use this class to derive your own factory for creating instances of a particular client interceptor class that you have derived from VISClientInterceptor. You must provide an implementation for the create method that returns an instance of your VISClientInterceptor-derived class.

Include file

The vinter.h file should be included when you use this class.

VISClientInterceptorFactory constructor

VISClientInterceptorFactory();

This is the default constructor.

VISClientInterceptorFactory methods

virtual VISClientInterceptor *create(CORBA::Object_ptr obj) = 0;

This method creates an interceptor for the specified object. Your implementation of this method should return an instance of your VISClientInterceptor-derived class.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>obj</td>
<td>The object for which this interceptor is being created.</td>
</tr>
</tbody>
</table>

virtual VISClientInterceptor *create_instance(CORBA::Object_ptr obj);

This method creates an interceptor for the specified object. Your implementation of this method should return an instance of your VISClientInterceptor-derived class.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>obj</td>
<td>The object for which this interceptor is being created.</td>
</tr>
</tbody>
</table>
static VISClosureInterceptorFactory *instance();

This static method returns a pointer to this factory object.

**VISClosure**

struct VISClosure

This structure is used to store data so that it can be shared between different invocations of interceptor methods. The data that is stored is un-typed and can represent state information related to an operation request or a bind or locate request. It is used in conjunction with the **VISClosureData** class.

**Include file**

The *vinter.h* file should be included when you use this class.

**VISClosure members**

CORBA::ULong id;

You can use this data member to uniquely identify this object if you are using more than one **VISClosure** objects.

void *data;

This data member points to the un-typed data that may be stored or accessed by an interceptor method.

VISClosureData *managedData;

This data member points to the **VISClosureData** class that represents the actual data. You may cast your managed data to this type.

**VISClosureData**

class VISClosureData

This class represents managed data that can be shared between different invocations of interceptor methods.

**VISClosureData methods**

virtual void ~VISClosureData();

This is the default destructor.
virtual void _release();

Releases this object and decrements the reference count. If the reference count is then 0, the object is deleted.

**VISObjectWrapper::ChainUntypedObjectWrapperFactory**

class ChainUntypedObjectWrapperFactory : public UntypedObjectWrapperFactory

This interface is used by a client or server application to add or remove an UntypedObjectWrapperFactory object. An UntypedObjectWrapperFactory, described on page 10-16, is used to create an UntypedObjectWrapper for each object a client application binds to or for each object implementation created by a server application.

See the VisiBroker for C++ Programmer’s Guide for complete information using object wrappers.

**Include file**

The `vobjwrap.h` file should be included when you use this class.

**ChainUntypedObjectWrapperFactory methods**

```cpp
void add(UntypedObjectWrapperFactory_ptr factory, Location loc);
```

This method adds the specified un-typed object wrapper factory for a client application, server application, or both.

If you application is acting as both a client application and a server application, you can install an un-typed object wrapper factory so that the wrapper’s methods will be invoked for both invocations on bound objects and operation requests received by object implementations.

**Note**

On the client side, un-typed object wrapper factories must be defined before any objects are bound. On the server side, un-typed object wrapper factories must be defined before any implementation objects are created.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>factory</td>
<td>The name of the operation being requested.</td>
</tr>
<tr>
<td>loc</td>
<td>The location of the factory being added, which should be one of the following values:</td>
</tr>
<tr>
<td></td>
<td>VISObjectWrapper::Client</td>
</tr>
<tr>
<td></td>
<td>VISObjectWrapper::Server</td>
</tr>
<tr>
<td></td>
<td>VISObjectWrapper::Both</td>
</tr>
</tbody>
</table>

Interceptor and object wrapper interfaces and classes      10-13
**ChainUntypedObjectWrapperFactory methods**

```c
static ChainUntypedObjectWrapperFactory* instance(CORBA::Boolean doCreate = 1);
```

This static method returns a reference to the un-typed object wrapper factory.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>doCreate</td>
<td>If set to TRUE and a factory does not exist, one will be created. If set to FALSE and a factory does not exist, a null pointer will be returned.</td>
</tr>
</tbody>
</table>

```c
static CORBA::ULong num_installed(Location loc);
```

This static method returns the number of un-typed object wrapper factories installed for the specified location.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>loc</td>
<td>The location of the factories: VISObjectWrapper::Client VISObjectWrapper::Server VISObjectWrapper::Both</td>
</tr>
</tbody>
</table>

```c
void remove(UntypedObjectWrapperFactory_ptr factory, Location loc);
```

This method removes the specified un-typed object wrapper factory from the specified location.

If you application is acting as both a client application and a server application, you can selectively remove the object wrapper factories for all bound object or for all object implementations.

**Note**

Removing one or more object wrapper factories from a client will not affect objects of that class which are already bound by the client. Only subsequently bound objects will be affected. Removing object wrapper factories from a server will not affect object implementations that have already been created. Only subsequently created and object implementation will be affected.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>factory</td>
<td>The name of the operation being requested.</td>
</tr>
<tr>
<td>loc</td>
<td>The location of the factory being removed, which should be one of the following values: VISObjectWrapper::Client VISObjectWrapper::Server VISObjectWrapper::Both</td>
</tr>
</tbody>
</table>
VISObjectWrapper::UntypedObjectWrapper

class UntypedObjectWrapper : VISResource

You use this class to derive and implement an un-typed object wrapper for a client application, a server application, or both. When you derive an un-typed object wrapper from this class, you define a pre_method method that is invoked before an operation request is issued by a client application or before it is processed by an object implementation on the server-side. You also define a post_method method that will be invoked after an operation request is processed by an object implementation on the server-side or after an reply has been received by a client application.

You must also derive a factory class from the UntypedObjectWrapperFactory class, described on page 10-16, that will create your un-typed wrapper objects.

See the VisiBroker for C++ Programmer’s Guide for complete information using object wrappers.

Include file

The vobjwrap.h file should be included when you use this class.

UntypedObjectWrapper methods

virtual void *pre_method(const char *operation, CORBA::Object_ptr target, VISClosure& closure);

This method is invoked before an operation request is sent on the client-side or before it is processed by an object implementation on the server side.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>operation</td>
<td>The name of the operation being requested.</td>
</tr>
<tr>
<td>target</td>
<td>The object that is the target of the request.</td>
</tr>
<tr>
<td>closure</td>
<td>The Closure object can be used to pass data between object wrapper methods.</td>
</tr>
</tbody>
</table>

virtual void *post_method(const char *operation, CORBA::Object_ptr target, CORBA::Environment& env, VISClosure& closure);

This method is invoked after an operation request has been processed by the object implementation on the server-side or before the reply message is processed by the stub on the client side.
class UntypedObjectWrapperFactory

You use this interface to derive your own un-typed object wrapper factories. Your factory will be used to create an instance of your un-typed object wrapper for a client or server application whenever a new object is bound or an object implementation is created.

Include file

The `vobjwrap.h` file should be included when you use this class.

UntypedObjectWrapperFactory constructors

UntypedObjectWrapperFactory();

This default constructor creates an un-typed object wrapper factory, but does not automatically register it with the `ChainUntypedObjectWrapperFactory`.

Note

If you create an un-typed object wrapper factory in this manner, you must use the `ChainUntypedObjectWrapperFactory::add` method, described on page 10-13, to register your factory.

UntypedObjectWrapperFactory(Location loc);

Creates an un-typed object wrapper factory for the specified location and automatically registers it with the `ChainUntypedObjectWrapperFactory`. If your application is acting as both a client application and a server application, you can install an un-typed object wrapper factory so the wrapper’s methods will be invoked for both invocations on bound objects and operation requests received by object implementations.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>loc</td>
<td>The location of the factory being added, which should be one of the following values:</td>
</tr>
<tr>
<td></td>
<td><code>VISObjectWrapper::Client</code></td>
</tr>
<tr>
<td></td>
<td><code>VISObjectWrapper::Server</code></td>
</tr>
<tr>
<td></td>
<td><code>VISObjectWrapper::Both</code></td>
</tr>
</tbody>
</table>
UntypedObjectWrapperFactory methods

virtual UntypedObjectWrapper_ptr create(CORBA::Object_ptr target) = 0;

This method is called to create an instance of your type of UntypedObjectWrapper. Your implementation of this method can examine the type of bound object or object implementation to determine whether or not it wants to create an object wrapper for that object.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>target</td>
<td>The object being bound by a client application for which the un-typed object wrapper is being created. If this method is being invoked on the server-side, this represents the object implementation that is being created.</td>
</tr>
</tbody>
</table>

VISServerInterceptor

class VISServerInterceptor : VISResource

You use this class to derive your own server-side interceptor, providing implementations for those methods that you wish to override. The methods defined in your derived class will be invoked by the ORB during the receipt of an operation request, during the preparation or sending of a reply messages for an operation request, or if an exception is raised.

Use the VISServerInterceptor class, described on page 14-1, to derive a class that will register your interceptor with the ORB.

This class also provides several static methods for testing and manipulating VISServerInterceptor pointers.

Include file

The vinter.h file should be included when you use this class.

VISServerInterceptor enumerations

enum ShutdownReason { CLIENT_ABORTED, SERVER_RESOURCES_EXCEEDED };

This enumeration contains values that can be used to indicate the reason why a client-server connection was shutdown by the ORB.

VISServerInterceptor constructors/destructors

VISServerInterceptor ():

This is the default constructor.
VISServerInterceptor methods

~VISServerInterceptor();

This is the default destructor.

VISServerInterceptor methods

virtual void exception_occurred(const GIOP::RequestHeader& req_hdr,
CORBA::Environment& env,
VISClosure& closure);

This method is invoked by the ORB whenever a method offered by this
interceptor throws an exception. Your implementation of this method may
provide additional processing, such as message logging or recovery
processing.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>req_hdr</td>
<td>The request header, which identifies the object, operation request, and other information about the request.</td>
</tr>
<tr>
<td>env</td>
<td>Contains information on the exception that was raised.</td>
</tr>
<tr>
<td>closure</td>
<td>May contain data saved by one interceptor method that can be retrieved later by another interceptor method.</td>
</tr>
</tbody>
</table>

static CORBA::Boolean is_nil(VISServerInterceptor i);

This method returns TRUE if the specified VISServerInterceptor pointer is NULL, otherwise FALSE is returned.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>The VISServerInterceptor pointer to be tested.</td>
</tr>
</tbody>
</table>

virtual IOP::IOR *locate(CORBAC::ULong req_id, CORBA_OctetSequence object_key,
VISClosure& closure);

This method is invoked by the ORB whenever a locate request is received for
a particular object. This usually occurs as the result of a client application
issuing a bind(). Your implementation of this method can return a non-null
IOR to force the locate request to be forwarded to another server of your choosing.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>req_id</td>
<td>The request identifier.</td>
</tr>
<tr>
<td>object_key</td>
<td>Identifies the object to be located.</td>
</tr>
<tr>
<td>closure</td>
<td>May contain data saved by one interceptor method that can be retrieved later by another interceptor method.</td>
</tr>
</tbody>
</table>
virtual IOP::IOR *locate_failed(CORBA::ULong req_id,
CORBA_OctetSequence object_key,
VISClosure& closure);

This method is invoked by the ORB if the server no longer offers the requested object. Your implementation of this method can return a non-null IOR to cause the locate request to be forwarded to another server of your choosing.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>req_id</td>
<td>The request identifier.</td>
</tr>
<tr>
<td>object_key</td>
<td>Identifies the object to be located.</td>
</tr>
<tr>
<td>closure</td>
<td>May contain data saved by one interceptor method that can be retrieved later by another interceptor method.</td>
</tr>
</tbody>
</table>

virtual void locate_forwarded(CORBA::ULong req_id, IOP::IOR& forward_ior,
VISClosure& closure);

This method is invoked by the ORB before it sends an OBJECT_FORWARD reply to a locate request. Your implementation of this method can modify the IOR to be returned, thereby causing the locate request to be forwarded to the server of your choosing.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>req_id</td>
<td>The request identifier.</td>
</tr>
<tr>
<td>forward_ior</td>
<td>An Interoperable Object Reference for the object. Modification of the IOR by this method is allowed.</td>
</tr>
<tr>
<td>closure</td>
<td>May contain data saved by one interceptor method that can be retrieved later by another interceptor method.</td>
</tr>
</tbody>
</table>

virtual void locate_succeeded(CORBA::ULong req_id, VISClosure& closure);

This method is invoked by the ORB before an OBJECT_HERE reply message is to be sent. Your implementation of this method can provide additional processing, such as logging a message.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>req_id</td>
<td>The request identifier.</td>
</tr>
<tr>
<td>closure</td>
<td>May contain data saved by one interceptor method that can be retrieved later by another interceptor method.</td>
</tr>
</tbody>
</table>
**VISServerInterceptor methods**

virtual void **prepare_reply**(const GIOP::RequestHeader& *req_hdr*,
GIOP::ReplyHeader& *reply_hdr*,
CORBA::Object_ptr *target*,
VISClosure& *closure*);

This method is invoked by the ORB when it is about to prepare a reply message. Your implementation may modify the contents of the `ReplyHeader`.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>req_hdr</td>
<td>The request header, which identifies the object, operation request, and other request information.</td>
</tr>
<tr>
<td>reply_hdr</td>
<td>The reply header. You can modify the contents of this header.</td>
</tr>
<tr>
<td>target</td>
<td>The object associated with the reply.</td>
</tr>
<tr>
<td>closure</td>
<td>May contain data saved by one interceptor method that can be retrieved later by another interceptor method.</td>
</tr>
</tbody>
</table>

virtual CORBA_MarshalInBuffer * **receive_request**(GIOP::RequestHeader& *req_hdr*,
CORBA::Object *& *target*,
CORBA_MarshalInBuffer * *buf*,
VISClosure& *closure*);

This method is invoked by the ORB when an ORB request message is received. Your implementation of this method may alter the contents of the request header and the request message buffer. Your implementation may provide specialized processing, such as data decryption.

If this method returns a non-null `MarshalInBuffer` pointer, that buffer will be passed to the target object instead of `buf`.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>req_hdr</td>
<td>The request header, which identifies the object, operation request, and other request information. You may modify the contents of the header.</td>
</tr>
<tr>
<td>target</td>
<td>The object associated with the request.</td>
</tr>
<tr>
<td>buf</td>
<td>The buffer containing the marshalled request. You may modify the contents of this buffer.</td>
</tr>
<tr>
<td>closure</td>
<td>May contain data saved by one interceptor method that can be retrieved later by another interceptor method.</td>
</tr>
</tbody>
</table>

virtual void **request_completed**(GIOP::RequestHeader& *req_hdr*,
CORBA::Object_ptr *target*,
VISClosure& *closure*);

This method is invoked by the ORB after a reply message has been successfully sent in response to a request message. It will also be invoked when a `oneway` request, which does not entail the sending of a reply message, has completed successfully. Your implementation of this method can
provide additional processing, such as logging a message or stopping a timer.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>req_hdr</td>
<td>The request header, which identifies the object, operation request, and other request information.</td>
</tr>
<tr>
<td>target</td>
<td>The object associated with the reply.</td>
</tr>
<tr>
<td>closure</td>
<td>May contain data saved by one interceptor method that can be retrieved later by another interceptor method.</td>
</tr>
</tbody>
</table>

```
virtual CORBA_MarshalOutBuffer *send_reply(const GIOP::RequestHeader& req_hdr,
const GIOP::ReplyHeader& reply_hdr,
CORBA::Object_ptr target,
CORBA_MarshalOutBuffer *buf,
VISClosure& closure);
```

This method is invoked by the ORB before it sends a reply message. Your implementation of this method can provide additional processing, such as logging a message or starting a timer.

If this method returns a non-null MarshalOutBuffer pointer, that buffer will be sent to the client instead of buf.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>req_hdr</td>
<td>The request header, which identifies the object, operation request, and other request information.</td>
</tr>
<tr>
<td>reply_hdr</td>
<td>The reply header.</td>
</tr>
<tr>
<td>target</td>
<td>The object associated with the request.</td>
</tr>
<tr>
<td>buf</td>
<td>The buffer containing the marshalled reply. You may modify the contents of this buffer.</td>
</tr>
<tr>
<td>closure</td>
<td>May contain data saved by one interceptor method that can be retrieved later by another interceptor method.</td>
</tr>
</tbody>
</table>

```
virtual void *send_reply_failed(const GIOP::RequestHeader& req_hdr,
const GIOP::ReplyHeader& reply_hdr,
CORBA::Object_ptr target,
const CORBA_Exception& excep,
VISClosure& closure);
```

This method is invoked by the ORB if it was unable to send a reply. The inability to send a reply is usually related to a communications failure. Your implementation of this method can provide additional processing, such as logging a message.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>req_hdr</td>
<td>The request header, which identifies the object, operation request, and other request information.</td>
</tr>
<tr>
<td>reply_hdr</td>
<td>The reply header.</td>
</tr>
</tbody>
</table>
virtual void shutdown(ShutdownReason reason);

This method is invoked by the ORB when the connection between the client application and the server has been forcibly terminated. Your implementation of this method can provide additional processing, such as logging a message.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>reason</td>
<td>The reason for the connection being terminated. See page 10-17 for a list of possible values.</td>
</tr>
</tbody>
</table>

static VISServerInterceptor _duplicate(VISServerInterceptor i);

This method duplicates the specified VISServerInterceptor pointer, increments the object’s reference count, and returns a pointer to the duplicated object.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>The VISServerInterceptor pointer to be duplicated.</td>
</tr>
</tbody>
</table>

VISServerInterceptorFactory

class VISServerInterceptorFactory

You use this class to derive your own factory for creating instances of a particular server interceptor class that you have derived from VISServerInterceptor. You must provide an implementation for the create method that returns an instance of your VISServerInterceptor-derived class.

Include file

The vinter.h file should be included when you use this class.

VISServerInterceptorFactory constructor

VISServerInterceptorFactory();

This is the default constructor.
VISServerInterceptorFactory methods

**virtual VISServerInterceptor *create(int fd, const IOP::TaggedProfile& profile) = 0;**

This method creates an interceptor for the specified object. Your implementation of this method should return an instance of your VISServerInterceptor-derived class.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fd</td>
<td>The file descriptor associated with the client connection.</td>
</tr>
<tr>
<td>profile</td>
<td>The profile of the object for which the interceptor is being created.</td>
</tr>
</tbody>
</table>

**virtual VISServerInterceptor *create_instance(int fd, const IOP::TaggedProfile& profile) = 0;**

This method creates an interceptor for the specified object. Your implementation of this method should return an instance of your VISServerInterceptor-derived class.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fd</td>
<td>The file descriptor associated with the client connection.</td>
</tr>
<tr>
<td>profile</td>
<td>The profile of the object for which the interceptor is being created.</td>
</tr>
</tbody>
</table>

**static VISServerInterceptorFactory *instance();**

This static method returns a pointer to this factory object.
This chapter describes the VisiBroker implementation of the key General Inter-ORB Protocol interfaces and other structures defined by the CORBA specification. For a complete description of these interfaces, refer to Chapter 12 of the OMG CORBA Specification.

GIOP::MessageHeader

This structure is used to represent information about a GIOP message.

**MessageHeader members**

```cpp
struct MessageHeader {
    CORBA::Char magic[4];
    // This string should always contain “GIOP”.
}```
GIOP::CancelRequestHeader

Version GIOP_version:
Indicates the version of the protocol being used. This structure contains a
major and minor version number, as shown. The major version should be set
to 1 and the minor version should be set to 0.

    struct Version {
        CORBA::Octet major;
        CORBA::Octet minor;
    };

CORBA::Boolean byte_order;
Set to TRUE to indicate that little-endian byte ordering is used in the message.
If set to FALSE, big-endian byte ordering is used in the message.

CORBA::Octet message_type;
Indicates the type of message that follows the header. This should be one of
the following values.

    enum MsgType {
        Request,
        Reply,
        CancelRequest,
        LocateRequest,
        LocateReply,
        CloseConnection,
        MessageError,
        Fragment
    };

CORBA::ULong message_size;
Indicates the length of the message that follows this header.

GIOP::CancelRequestHeader

struct CancelRequestHeader
This structure is used to represent information about a cancel request message
header.

CancelRequestHeader members

CORBA::ULong request_id;
This data member represents the request identifier that is being cancelled.
GIOP::LocateReplyHeader

struct LocateReplyHeader

This structure is used to represent a message that is sent in reply to a locate request message. Additional data follows this header if the locate_status is set to OBJECT_FORWARD.

LocateReplyHeader members

CORBA::ULong request_id;

The request identifier of the original request.

LocateStatusType locate_status;

Indicates the disposition of the locate request as one of the following values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNKNOWN_OBJECT</td>
<td>Indicates the requested object could not be found. No other data is associated with this message.</td>
</tr>
<tr>
<td>OBJECT_HERE</td>
<td>Indicates the object is implemented by this server. No other data is associated with this message.</td>
</tr>
<tr>
<td>OBJECT_FORWARD</td>
<td>Indicates that the object is implemented by another server and a IOR for that server follows this header.</td>
</tr>
</tbody>
</table>

GIOP::LocateRequestHeader

structure LocateRequestHeader

This structure represents a message containing a request to locate an object.

LocateRequestHeader members

CORBA::ULong request_id;

Represents the request identifier for this message and is used to distinguish between multiple outstanding messages.

CORBA::OctetSequence object_key;

Represents the object to be located. This data is in a vendor-specific format.

GIOP::ReplyHeader

struct ReplyHeader {};

This structure represents the reply header of a reply message that is sent to a client in response to a request message.
Include file

The `vgiop.h` file should be included when you use this structure.

ReplyHeader members

IOP::ServiceContextList `service_info`;

A list of service context information that may be passed from the server to the client.

CORBA::ULong `request_id`;

Should be set to the same `request_id` as the request message for which this reply is associated.

ReplyStatusType `reply_status`;

Indicates the status of the reply and should be set to one of the following enum values:

- NO_EXCEPTION
- USER_EXCEPTION
- SYSTEM_EXCEPTION
- LOCATION_FORWARD

GIOP::RequestHeader

struct `RequestHeader` {};

This structure represents the request header of a request message that is sent to an object implementation.

Include file

The `vgiop.h` file should be included when you use this structure.

RequestHeader members

IOP::ServiceContextList `service_context`;

A list of service context information that may be passed from the client to the server.

CORBA::ULong `request_id`;

A unique identifier used to associate a reply message with a particular request message.
IIOP::ProfileBody

struct ProfileBody;

This structure contains information about the protocol supported by an object.

module IIOP {
    . . .
    struct ProfileBody {
        Version iiop_version;
        CORBA::String_var host;
        unsigned short port;
        CORBA::OctetSequence_var object_key;
        sequence<IOP::taggedComponent> components;
    }
    . . .

ProfileBody members

Version iiop_version;
    Represents the version of IIOP supported.

CORBA::String_var host;
    Represents the name of the host where the object is implemented.
IOP::IOR

CORBA::UShort port;

Indicates the port number to use for establishing a connection to the object.

CORBA::OctetSequence object_key;

Object keys are stored in a vendor-specific format and are generated when an IOR is created.

IIOP::MultiComponentProfile components;

A sequence of TaggedComponents which contain information about the protocols that are supported.

IOP::IOR

struct IOR {} 

This structure represents an Interoperable Object Reference and is used to provide important information about object references. Your client application can create an IOR by invoking the ORB::object_to_string method described on page 5-12.

Include file

The vgiop.h file should be included when you use this structure.

IOR members

CORBA::String_var type_id;

This data member describes the type of object reference that is represented by this IOR.

TaggedProfileSequence profiles;

This data member represents a sequence of one or more TaggedProfile structures, which contain information about the protocols that are supported.

static CORBA::Boolean is_nil(IOP::IOR *i);

This method returns TRUE if the specified pointer is NULL.

IOP::TaggedProfile

struct TaggedProfile

This structure represents a particular protocol that is supported by an Interoperable Object Reference (IOR).
TaggedProfile members

ProfileID tag:

This data member represents the contents of the profile data and should be one of the following values.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TAG_INTERNET_IOP</td>
<td>Indicates the protocol is standard IIOP.</td>
</tr>
<tr>
<td>TAG_MULTIPLE_COMPONENTS</td>
<td>Indicates the profile data contains a list of ORB services available using the protocol.</td>
</tr>
<tr>
<td>TAG_VSGN_LOCATOR</td>
<td>Indicates that the IOR is an interim, pseudo-object that is used until the real IOR is received by the osagent.</td>
</tr>
<tr>
<td>TAG_LOCAL_IPC_IOP</td>
<td>Indicates the protocol is IOP over a local IPC mechanism.</td>
</tr>
</tbody>
</table>

CORBA_OctetSequence profile_data:

This data member encapsulates all the protocol information needed to invoke an operation on an IOR.
Chapter 12

Marshal buffer classes

This chapter describes the buffer class used for marshalling data to a buffer when creating an operation request or a reply message. It also describes the buffer class used for extracting data from a received operation request or reply message.

CORBA::MarshalInBuffer class

This class represents a stream buffer that allows IDL types to be written to a buffer and may be used by interceptor methods that your implement. See Chapter 10, “Interceptor and object wrapper interfaces and classes,” for more information on the interceptor interfaces.

The CORBA::MarshalInBuffer class is used on the client side to marshal the data associated with an operation request. It is used on the server side to marshal the data associated with a reply message. This class provides a wide range of methods for adding various types of data to the buffer or for retrieving them from the buffer.

This class provides several static methods for testing and manipulating CORBA::MarshalInBuffer pointers.

A CORBA::MarshalInBuffer_var class is also offered, which provides a wrapper that automatically manages the contained object.

CORBA::MarshalOutBuffer page 12-5
Include file

The `mbuf.h` file should be included when you use this class.

**CORBA::MarshalInBuffer constructors/destructors**

```cpp
CORBA::MarshalInBuffer(char *read_buffer, CORBA::ULong length,
                        CORBA::Boolean release_flag=0,
                        CORBA::Boolean byte_order = CORBA::ByteOrder);
```

This is the default constructor.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>read_buffer</td>
<td>The buffer where the marshalled data will actually be stored.</td>
</tr>
<tr>
<td>length</td>
<td>The maximum number of bytes that may be stored in <code>read_buffer</code>.</td>
</tr>
<tr>
<td>release_flag</td>
<td>If set to TRUE, the memory associated with <code>read_buffer</code> will be freed when this object is destroyed. The default value is FALSE.</td>
</tr>
<tr>
<td>byte_order</td>
<td>Set this TRUE to indicate that little-endian byte ordering is being used. Set to FALSE to indicate that big-endian byte ordering is being used.</td>
</tr>
</tbody>
</table>

```cpp
virtual ~CORBA::MarshalInBuffer();
```

This is the default destructor. The buffer memory associated with this object will be released if the `release_flag` is set to TRUE. The `release_flag` may be set when the object is created or by invoking the `release_flag` method, described on page 12-4.

**CORBA::MarshalInBuffer methods**

```cpp
char *buffer() const;
```

Returns a pointer to the buffer associated with this object.

```cpp
void byte_order(CORBA::Boolean val) const;
```

Sets the byte ordering for this message buffer.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>val</td>
<td>Set this TRUE to indicate that little-endian byte ordering is being used. Set to FALSE to indicate that big-endian byte ordering is being used.</td>
</tr>
</tbody>
</table>

```cpp
CORBA::Boolean byte_order() const;
```

Returns TRUE if the buffer is using little-endian byte ordering. FALSE is returned if big-endian byte ordering is being used.
Corba::ULong cutoff() const;

Returns the current offset within the buffer associated with this object.

virtual VISistream& get(char& data);
virtul VISistream& get(unsigned char& data);

These methods allow you to retrieve a single character from the buffer at the current location.

This method returns a pointer to the location within the buffer immediately following the end of the data that was just retrieved.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>data</td>
<td>The location where the retrieved char or unsigned char is to be stored.</td>
</tr>
</tbody>
</table>

virtual VISistream& get(<data_type> data, unsigned size);

These methods allow you to retrieve a sequence of data from the buffer at the current location. There is a separate method for each of the listed target data types.

This method returns a pointer to the location within the buffer immediately following the end of the data that was just retrieved.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>data</td>
<td>The location where the retrieved data is to be stored.</td>
</tr>
<tr>
<td>size</td>
<td>The number of the specified data types to be retrieved.</td>
</tr>
<tr>
<td>data</td>
<td>The supported target data types are:</td>
</tr>
<tr>
<td></td>
<td>char* float*</td>
</tr>
<tr>
<td></td>
<td>unsigned char* double*</td>
</tr>
<tr>
<td></td>
<td>short* long double*</td>
</tr>
<tr>
<td></td>
<td>unsigned short* VISLongLong*</td>
</tr>
<tr>
<td></td>
<td>int* VISULongLong*</td>
</tr>
<tr>
<td></td>
<td>unsigned int* wchar_t*</td>
</tr>
<tr>
<td></td>
<td>long*</td>
</tr>
<tr>
<td></td>
<td>unsigned long*</td>
</tr>
</tbody>
</table>

virtual VISistream& getCString(char* data, unsigned maxlen);

This method allows you to retrieve a string character from the buffer at the current location. It returns a pointer to the location within the buffer immediately following the end of the data that was just retrieved.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>data</td>
<td>The location where the retrieved character string is to be stored.</td>
</tr>
<tr>
<td>maxlen</td>
<td>The maximum number of characters to be retrieved.</td>
</tr>
</tbody>
</table>
CORBA::MarshalInBuffer methods

virtual const CORBA::WChar *getWString(CORBA::ULong& len);

This method returns a pointer to a location within the buffer containing a Unicode string.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>len</td>
<td>The offset of the desired data within the buffer.</td>
</tr>
</tbody>
</table>

virtual int is_available(unsigned long size);

Returns 1 if the specified size is less than or equal to the size of the buffer associated with this object.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>Number of bytes that need to fit within this buffer.</td>
</tr>
</tbody>
</table>

virtual CORBA::ULong length() const;

Returns the total number of bytes in this object’s buffer.

virtual void new_encapsulation() const;

Resets the starting offset within the buffer to 0.

void release_flag( CORBA::Boolean val);

Enables or disables the automatic freeing of buffer memory when this object is destroyed.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>val</td>
<td>If val is set to TRUE, the buffer memory for this object will be freed when this object is destroyed. If val is set to FALSE, the buffer will not be freed when this object is destroyed</td>
</tr>
</tbody>
</table>

CORBA::Boolean release_flag() const;

Returns TRUE if the automatic freeing of this object’s buffer memory is enabled, otherwise FALSE is returned.

void reset();

Resets the starting offset, current offset and seek position to zero.

void rewind();

Resets the seek position to 0.

CORBA::ULong seekpos(CORBA::ULong pos);

Sets the current offset to the value contained in pos. If pos specifies an offset that is greater than the size of the buffer, a CORBA::BAD_PARAM exception is raised.
static CORBA::MarshalInBuffer * _duplicate(CORBA::MarshalInBuffer_ptr ptr);

Returns a duplicate pointer to this object pointed to by ptr and increments this object’s reference count.

static CORBA::MarshalInBuffer * _nil();

Returns a NULL pointer of type CORBA::MarshalInBuffer.

static void _release(CORBA::MarshalInBuffer_ptr ptr);

Reduces the reference count of the object pointed to by ptr. If the reference count is then 0, the object is destroyed. If the object’s release_flag was set to true when it was constructed, the buffer associated with the object will freed.

**CORBA::MarshalInBuffer operators**

virtual VISistream& operator>>(<data_type> data);

This stream operator allows you to add data of the specified source data_type to the buffer at the current location.

This method returns a pointer to the location within the buffer immediately following the end of the data that was just written.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>data</td>
<td>The data to be written to the buffer. The supported source data types are: char*</td>
</tr>
<tr>
<td></td>
<td>char &amp;</td>
</tr>
<tr>
<td></td>
<td>unsigned char &amp;</td>
</tr>
<tr>
<td></td>
<td>short &amp;</td>
</tr>
<tr>
<td></td>
<td>unsigned short &amp;</td>
</tr>
<tr>
<td></td>
<td>int &amp;</td>
</tr>
<tr>
<td></td>
<td>unsigned int &amp;</td>
</tr>
</tbody>
</table>

**CORBA::MarshalOutBuffer**

class CORBA::MarshalOutBuffer : public VISstream

This class represents a stream buffer that allows IDL types to be read from a buffer and may be used by interceptor methods that you implement. See Chapter 10, “Interceptor and object wrapper interfaces and classes,” for more information on the interceptor interfaces.

The CORBA::MarshalOutBuffer class is used on the client side to extract the data associated with a reply message. It is used on the server side to extract the data associated with an operation request. This class provides a wide range of methods for adding various types of data to the buffer or for retrieving them from the buffer.
This class provides several static methods for testing and manipulating CORBA::MarshalOutBuffer pointers.

A CORBA::MarshalInBuffer_var class is also offered, which provides a wrapper that automatically manages the contained object.

**Include file**

The mbuf.h file should be included when you use this class.

**CORBA::MarshalOutBuffer constructors/destructors**

CORBA::MarshalOutBuffer(CORBA::ULong initial_size = 255,
CORBA::Boolean release_flag = 0);

Creates an object with a 255 byte buffer.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>initial_size</td>
<td>The initial size of the buffer associated with this object. The default size is 255 bytes.</td>
</tr>
<tr>
<td>release_flag</td>
<td>If set to TRUE, the memory associated with read_buffer will be freed when this object is destroyed. The default value is FALSE.</td>
</tr>
</tbody>
</table>

CORBA::MarshalOutBuffer(char *read_buffer, CORBA::ULong len,
CORBA::Boolean release_flag=0);

Creates an object with the specified buffer, buffer length and release flag value.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>read_buffer</td>
<td>The buffer where the marshalled data will actually be stored.</td>
</tr>
<tr>
<td>length</td>
<td>The maximum number of bytes that may be stored in read_buffer.</td>
</tr>
<tr>
<td>release_flag</td>
<td>If set to TRUE, the memory associated with read_buffer will be freed when this object is destroyed. The default value is FALSE.</td>
</tr>
</tbody>
</table>

virtual ~CORBA::MarshalOutBuffer();

This is the default destructor. The buffer memory associated with this object will be released if the release_flag is set to TRUE. The release_flag may be set when the object is created or by invoking the release_flag method, described on page 12-8.

**CORBA::MarshalOutBuffer methods**

char *buffer() const;

Returns a pointer to the buffer associated with this object.
CORBA::MarshalOutBuffer methods

CORBA::ULong cutoff() const;

Returns the current offset within the buffer associated with this object.

virtual CORBA::ULong length() const;

Returns the total number of bytes in this object’s buffer.

virtual void new_encapsulation() const;

Resets the starting offset within the buffer to 0.

virtual VISostream& put(char data);

Adds a single character to the buffer at the current location.
This method returns a pointer to the location within the buffer immediately following the end of the data that was just added.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>data</td>
<td>The char to be stored.</td>
</tr>
</tbody>
</table>

virtual VISostream& put(const <data_type> data, unsigned size);

These methods allow you to store a sequence of data in the buffer at the current location.
This method returns a pointer to the location within the buffer immediately following the end of the data that was just added.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>data</td>
<td>The data is to be stored. The supported source data types are:</td>
</tr>
<tr>
<td></td>
<td>char* float*</td>
</tr>
<tr>
<td></td>
<td>unsigned char* double*</td>
</tr>
<tr>
<td></td>
<td>short* long double*</td>
</tr>
<tr>
<td></td>
<td>unsigned short* VISLongLong*</td>
</tr>
<tr>
<td></td>
<td>int* VISULongLong*</td>
</tr>
<tr>
<td></td>
<td>unsigned int* wchar_t*</td>
</tr>
<tr>
<td></td>
<td>long*</td>
</tr>
<tr>
<td></td>
<td>unsigned long*</td>
</tr>
<tr>
<td>size</td>
<td>The number of the specified data types to be stored.</td>
</tr>
</tbody>
</table>

virtual VISostream& putCString(const char* data);

This method allows you to store a character string into the buffer at the current location. It returns a pointer to the location within the buffer immediately following the end of the data that was just added.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>data</td>
<td>The character string to be stored.</td>
</tr>
</tbody>
</table>
**CORBA::MarshalOutBuffer operators**

```cpp
void release_flag(CORBA::Boolean val);
```

Enables or disables the automatic freeing of buffer memory when this object is destroyed.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>val</td>
<td>If val is set to TRUE, the buffer memory for this object will be freed when this object is destroyed. If val is set to FALSE, the buffer will not be freed when this object is destroyed</td>
</tr>
</tbody>
</table>

```cpp
CORBA::Boolean release_flag() const;
```

Returns **TRUE** if the automatic freeing of this object’s buffer memory is enabled, otherwise **FALSE** is returned.

```cpp
void reset();
```

Resets the starting offset, current offset and seek position to zero.

```cpp
void rewind();
```

Resets the seek position to 0.

```cpp
CORBA::ULong seekpos(CORBA::ULong pos);
```

Sets the current offset to the value contained in `pos`. If `pos` specifies an offset that is greater than the size of the buffer, a **CORBA::BAD_PARAM** exception is raised.

```cpp
static CORBA::MarshalOutBuffer * _duplicate(CORBA::MarshalOutBuffer_ptr ptr);
```

Returns a duplicate pointer to this object pointed to by `ptr` and increments this object’s reference count.

```cpp
static CORBA::MarshalOutBuffer * _nil();
```

Returns a **NULL** pointer of type **CORBA::MarshalOutBuffer**.

```cpp
static void _release(CORBA::MarshalOutBuffer_ptr ptr);
```

Reduces the reference count of the object pointed to by `ptr`. If the reference count is then 0, the object is destroyed. If the object’s `release_flag` was set to `true` when it was constructed, the buffer associated with the object will be freed.

**CORBA::MarshalOutBuffer operators**

```cpp
virtual VIStream& operator<<(data_type data);
```

This stream operator allows you to add data of the specified `data_type` to the buffer at the current location.
This method returns a pointer to the location within the buffer immediately following the end of the data that was just written.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>data</td>
<td>The data to be obtained to the buffer. The supported data types are:</td>
</tr>
<tr>
<td></td>
<td>const char*</td>
</tr>
<tr>
<td></td>
<td>char</td>
</tr>
<tr>
<td></td>
<td>unsigned char</td>
</tr>
<tr>
<td></td>
<td>short</td>
</tr>
<tr>
<td></td>
<td>unsigned short</td>
</tr>
<tr>
<td></td>
<td>int</td>
</tr>
<tr>
<td></td>
<td>int</td>
</tr>
<tr>
<td></td>
<td>long</td>
</tr>
<tr>
<td></td>
<td>unsigned long</td>
</tr>
</tbody>
</table>
This chapter describes the interfaces you can use to locate object instances on a network of Smart Agents. For more information on the Location Service, see Chapter 19, “Discovering object instances using the Location Service” in the VisiBroker for C++ Programmer’s Guide.

<table>
<thead>
<tr>
<th>Class</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Agent</td>
<td>13-1</td>
</tr>
<tr>
<td>Desc</td>
<td>13-6</td>
</tr>
<tr>
<td>Fail</td>
<td>13-7</td>
</tr>
<tr>
<td>TriggerDesc</td>
<td>13-7</td>
</tr>
<tr>
<td>TriggerHandler</td>
<td>13-8</td>
</tr>
<tr>
<td>&lt;type&gt;Seq</td>
<td>13-9</td>
</tr>
<tr>
<td>&lt;type&gt;SeqSeq</td>
<td>13-10</td>
</tr>
</tbody>
</table>

**Agent**

class Agent : public CORBA::Object

This class provides methods that enable you to locate all instances of a particular object on a network of Smart Agents. The methods offered by this class are divided into two categories; those that query a Smart Agent for data about objects and those that deal with triggers.

Your client application can obtain object information based on an interface repository ID alone or in combination with an instance name.

Triggers allow your client application to be notified of changes in the availability of one or more object instances.
Command-line options for applications using the Location Service are described in “Location service options” on page A-6.

```c
interface Agent {
    HostnameSeq all_agent_locations()
        raises (Fail);
    RepositoryIdSeq all_repository_ids()
        raises (Fail);
    ObjSeqSeq all_available()
        raises (Fail);
    ObjSeq all_instances (in string repository_id)
        raises (Fail);
    ObjSeq all_replica (in string repository_id, in string instance_name)
        raises (Fail);
    DescSeqSeq all_available_descs()
        raises (Fail);
    DescSeq all_instances_descs (in string repository_id)
        raises (Fail);
    DescSeq all_replica_descs (in string repository_id,
                             in string instance_name)
        raises (Fail);
    void reg_trigger(in TriggerDesc desc, in TriggerHandler handler)
        raises (Fail);
    void unreg_trigger(in TriggerDesc desc, in TriggerHandler handler)
        raises (Fail);
    attribute boolean willRefreshOADs;
};
```

Include file

You should include the `locate_c.hh` file when you use this class.

Agent methods

```
ObjLocation::HostnameSeq_ptr all_agent_locations();
```

Returns a sequence of host names representing the hosts on which osagent processes are currently executing.

See also

“<type>Seq” on page 13-9

This method throws the following exceptions:

<table>
<thead>
<tr>
<th>Exception</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fail</td>
<td>The <code>FailReason</code> values that may be presented include:</td>
</tr>
<tr>
<td></td>
<td>NO_AGENT_AVAILABLE</td>
</tr>
<tr>
<td></td>
<td>NO_SUCH_TRIGGER</td>
</tr>
<tr>
<td></td>
<td>AGENT_ERROR</td>
</tr>
</tbody>
</table>

See page 13-7 for a discussion of the `Fail` class.
Agent methods

ObjLocation::ObjSeq_ptr all_available();

Returns a sequence of object references for all objects currently registered with some Smart Agent on the network.

See also “<type>Seq” on page 13-9

This method throws the following exceptions:

<table>
<thead>
<tr>
<th>Exception</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fail</td>
<td>The FailReason values that may be presented include:</td>
</tr>
<tr>
<td></td>
<td>NO_AGENT_AVAILABLE</td>
</tr>
<tr>
<td></td>
<td>NO_SUCH_TRIGGER</td>
</tr>
<tr>
<td></td>
<td>AGENT_ERROR</td>
</tr>
<tr>
<td></td>
<td>See page 13-7 for a discussion of the Fail class.</td>
</tr>
</tbody>
</table>

ObjLocation::DescSeqSeq_ptr all_available_desc();

Returns descriptions for all objects currently registered with a Smart Agent on the network. The description information returned is organized by repository id.

See also “<type>SeqSeq” on page 13-10

This method throws the following exceptions:

<table>
<thead>
<tr>
<th>Exception</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fail</td>
<td>The FailReason values that may be presented include:</td>
</tr>
<tr>
<td></td>
<td>NO_AGENT_AVAILABLE</td>
</tr>
<tr>
<td></td>
<td>NO_SUCH_TRIGGER</td>
</tr>
<tr>
<td></td>
<td>AGENT_ERROR</td>
</tr>
<tr>
<td></td>
<td>See page 13-7 for a discussion of the Fail class.</td>
</tr>
</tbody>
</table>

ObjLocation::ObjSeq_ptr all_instances(const char *repository_id);

Returns a sequence of object references to all instances with the specified repository_id.

See also “<type>Seq” on page 13-9

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>repository_id</td>
<td>The repository ID of the object references to be retrieved.</td>
</tr>
</tbody>
</table>

This method throws the following exceptions:

<table>
<thead>
<tr>
<th>Exception</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fail</td>
<td>Any of the FailReason values, other than NO_SUCH_TRIGGER, may be presented. See page 13-7 for a discussion of the Fail class.</td>
</tr>
</tbody>
</table>
Agent methods

ObjLocation::DescSeq_ptr all_instances_descs(const char *repository_id);

Returns description information for all object instances with the specified repository_id.

See also “<type>Seq” on page 13-9

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>repository_id</td>
<td>The repository ID of the object descriptions to be retrieved.</td>
</tr>
</tbody>
</table>

This method throws the following exceptions:

<table>
<thead>
<tr>
<th>Exception</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fail</td>
<td>Any of the FailReason values, other than NO_SUCH_TRIGGER, may be presented. See page 13-7 for a discussion of the Fail class.</td>
</tr>
</tbody>
</table>

ObjLocation::ObjSeq_ptr all_replica(const char *repository_id, const char *instance_name);

Returns a sequence of object references for objects with the specified repository_id and instance_name.

See also “<type>Seq” on page 13-9

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>repository_id</td>
<td>The repository ID of the object references to be retrieved.</td>
</tr>
<tr>
<td>instance_name</td>
<td>The instance name of the object references to be returned.</td>
</tr>
</tbody>
</table>

This method throws the following exceptions:

<table>
<thead>
<tr>
<th>Exception</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fail</td>
<td>Any of the FailReason values, other than NO_SUCH_TRIGGER, may be presented. See page 13-7 for a discussion of the Fail class.</td>
</tr>
</tbody>
</table>

ObjLocation::DescSeq_ptr all_replica_descs(const char *repository_id, const char *instance_name);

Returns a sequence of description information for all object instances with the specified repository_id and instance_name.

See also “<type>Seq” on page 13-9

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>repository_id</td>
<td>The repository ID of the object descriptions to be retrieved.</td>
</tr>
<tr>
<td>instance_name</td>
<td>The instance name of the object descriptions to be retrieved.</td>
</tr>
</tbody>
</table>
This method throws the following exceptions:

<table>
<thead>
<tr>
<th>Exception</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fail</td>
<td>Any of the FailReason values, other than NO_SUCH_TRIGGER, may be presented. See page 13-7 for a discussion of the Fail class.</td>
</tr>
</tbody>
</table>

```cpp
void reg_trigger(const ObjLocation::TriggerDesc& desc,
                 ObjLocation::TriggerHandler_ptr hdlr);
```

Registers the trigger handler `hdlr` for object instances that match the description information specified in `desc`.

**Note** A `TriggerHandler` will be invoked every time an object that satisfies the trigger’s description becomes available. If you are only interested in learning when the first instance of the object becomes available, you should use the `unreg_trigger` method to remove the trigger after the first notification is received.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>desc</td>
<td>The object instance description information, which can contain combinations of the following information: repository ID, instance name, hostname. You can provide more or less information to narrow or widen the object instances to be monitored.</td>
</tr>
<tr>
<td>hdlr</td>
<td>The trigger handler object being registered.</td>
</tr>
</tbody>
</table>

This method throws the following exceptions:

<table>
<thead>
<tr>
<th>Exception</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fail</td>
<td>Any of the FailReason values, other than NO_SUCH_TRIGGER, may be presented. See page 13-7 for a discussion of the Fail class.</td>
</tr>
</tbody>
</table>

```cpp
void unreg_trigger(const ObjLocation::TriggerDesc& desc,
                    ObjLocation::TriggerHandler_ptr hdlr);
```

Unregisters the trigger handler `hdlr` for object instances that match the description information specified in `desc`.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>desc</td>
<td>The object description information.</td>
</tr>
<tr>
<td>hdlr</td>
<td>The trigger handler object being unregistered.</td>
</tr>
</tbody>
</table>

This method throws the following exceptions:

<table>
<thead>
<tr>
<th>Exception</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fail</td>
<td>The only FailReason value possible is NO_SUCH_TRIGGER. See page 13-7 for a discussion of the Fail class.</td>
</tr>
</tbody>
</table>
CORBA::Boolean willRefreshOADs();

Returns TRUE if the set of Object Activation Daemon is updated each time a method offered by this class is invoked, otherwise FALSE is returned. If the cache is not refreshed on each invocation, the following conditions may occur:

- All objects will still be reported, but their descriptor’s activable flag may be incorrect.
- Any attempt to verify the existence of an object registered with an OAD that has been started since the last refresh of the OAD cache will cause those objects to be activated by the OAD.

void willRefreshOADs(CORBA::Boolean val);

This class maintains a set of Object Activation Daemons. This method enables or disables the automatic refreshing of the OADs contained in this set.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>val</td>
<td>If TRUE, the OAD set will be refreshed whenever a method offered by this class is invoked.</td>
</tr>
</tbody>
</table>

struct Desc;

This structure contains information you use to describe the characteristics of an object. You pass this structure as an argument to several of the Location Service methods described in the chapter. The Desc structure, or a sequence of them, is returned by some of the Location Service methods.

See also “<type>Seq” on page 13-9

module ObjLocation {
    struct Desc {
        Object ref;
        IIOP::ProfileBody iiop_locator;
        string repository_id;
        string instance_name;
        boolean activable;
        string agent_hostname;
    };
    ...
}

Desc members

Object ref;

A reference to the object being described.
IIOP::ProfileBody iiop_locator;

   Represents profile data for the object, described in “IIOP::ProfileBody” on page 11-5.

CORBA::String_var repository_id;

   The object’s repository identifier.

CORBA::String_var instance_name;

   The object’s instance name.

CORBA::Boolean activable;

   Set to TRUE to indicate that this object is registered with the Object Activation Daemon. It is set to FALSE to indicate that the object was started manually and is registered with the osagent.

CORBA::String_var agent_hostname;

   The name of the host running the Smart Agent with which this object is registered.

---

class Fail : public CORBA::UserException

   This exception class may be thrown by the Agent class to indicate various errors. The data member FailReason is used to indicate the nature of the failure.

---

FailReason reason;

   Set to one of the following values to indicate the nature of the failure:

   enum FailReason {
      NO_AGENT_AVAILABLE,
      INVALID_REPOSITORY_ID,
      INVALID_OBJECT_NAME,
      NO_SUCH_TRIGGER,
      AGENT_ERROR
   };

---

struct TriggerDesc;

   This structure contains information you use to describe the characteristics of one or more objects for which you wish to register a TriggerHandler, described on page 13-5. The host_name and instance_name members may be set to NULL to
TriggerDesc members

monitor the widest possible set of objects. The more information that is specified, the smaller the set of objects will be.

module ObjLocation {
    ...
    struct TriggerDesc {
        string repository_id;
        string instance_name;
        string host_name;
    };
    ...
}

CORBA::String_var repository_id;

Represents the repository identifiers of the objects to be monitored by the TriggerHandler. May be set to NULL to include all possible repository identifiers.

CORBA::String_var instance_name;

Represents the instance name of the object to be monitored by the TriggerHandler. May be set to NULL to include all possible instance names.

CORBA::String_var host_name;

Represents the host name where the object or objects to be monitored by the TriggerHandler are located. May be set to NULL to include all hosts in the network.

TriggerHandler

You use this base class to derive your own callback object to be invoked every time an object becomes available or unavailable. You specify the criteria for the object or objects in which you are interested. You register your TriggerHandler object using the Agent::reg_trigger method, described on page 13-5.

You must provide implementations for the impl_is_ready and impl_is_down methods.

interface TriggerHandler {
    void impl_is_ready(in Desc desc);
    void impl_is_down(in Desc desc);
};

Include file

You should include the locate_c.hh file when you use this class.
TriggerHandler methods

virtual void impl_is_ready(const Desc& desc);

This method is invoked by the Location Service when an object instance matching the criteria specified in desc becomes accessible.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>desc</td>
<td>The object description information.</td>
</tr>
</tbody>
</table>

virtual void impl_is_down(const Desc& desc);

This method is invoked by the Location Service when an object instance matching the criteria specified in desc is no longer accessible.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>desc</td>
<td>The object description information.</td>
</tr>
</tbody>
</table>

<type>Seq

This is a generalized class description for the following sequence classes used by the Location Service:

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DescSeq</td>
<td>A sequence of Desc structures.</td>
</tr>
<tr>
<td>HostnameSeq</td>
<td>A sequence of host names.</td>
</tr>
<tr>
<td>ObjSeq</td>
<td>A sequence of object references.</td>
</tr>
<tr>
<td>RepositoryIdSeq</td>
<td>A sequence of repository identifiers.</td>
</tr>
</tbody>
</table>

Each class represents a particular sequence of <type>. The Location Service returns lists of information to your client application in the form of sequences which are mapped to one of these classes.

Each class offers operators for indexing items in the sequence just as you would a C++ array. The also offer methods for obtaining the length of the array, and for setting the array length.

Code sample 13.1 shows the correct way to index a HostnameSeq returned from the Agent::all_agent_locations method.

**Code sample 13.1  Indexing a HostnameSeq_var**

```cpp
//...
ObjLocation::HostnameSeq_var hostnames(myAgent->all_agent_locations());
for (CORBA::ULong i=0; i < hostnames->length(); i++) {
    cout << "Agent host #" << i+1 << " : " << hostnames[i] << endl;
} //...
```

See also  “<type>SeqSeq” on page 13-10
<type>Seq methods

<type>& operator[](CORBA::ULong index) const;

Returns a reference to the element in the sequence identified by index.

Caution

You must use a CORBA::ULong type for the index. Using an int type may lead to unpredictable results.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>index</td>
<td>The index of the element to be returned. This index is zero-based.</td>
</tr>
</tbody>
</table>

This method throws the following exceptions:

<table>
<thead>
<tr>
<th>Exception</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CORBA::BAD_PARAM</td>
<td>The index specified is less than zero or greater that the size of the sequence.</td>
</tr>
</tbody>
</table>

CORBA::ULong length() const;

Returns the number of elements in the sequence.

void length(CORBA::ULong len);

Sets the maximum length of the sequence to the value contained in len.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>len</td>
<td>The new length for the sequence.</td>
</tr>
</tbody>
</table>

<type>SeqSeq

This is a generalized class description for the following classes used by the Location Service:

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DescSeqSeq</td>
<td>A sequence of DescSeq objects.</td>
</tr>
<tr>
<td>ObjSeqSeq</td>
<td>A sequence of ObjSeq objects.</td>
</tr>
</tbody>
</table>

Each class represents a particular sequence of <type>Seq. Some Location Service methods return lists of information to your client application in the form of sequences of sequences which are mapped to one of these classes.

Each class offers operators for indexing items in the sequence just as you would a C++ array. The class also offer methods for obtaining the length of the array, and for setting the array length.

See also  “<type>Seq” on page 13-9
<type>SeqSeq methods

<type>Seq& operator[](CORBA::ULong index) const;

Returns a reference to the element in the sequence identified by `index`. The reference is to a one dimensional sequence, described on page 13-9.

Caution

You must use a CORBA::ULong type for the index. Using an int type may lead to unpredictable results.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>index</td>
<td>The index of the element to be returned. This index is zero-based.</td>
</tr>
</tbody>
</table>

This method throws the following exceptions:

<table>
<thead>
<tr>
<th>Exception</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CORBA::BAD_PARAM</td>
<td>The index specified is less than zero or greater than the size of the sequence.</td>
</tr>
</tbody>
</table>

CORBA::ULong length() const;

Returns the number of elements in the sequence.

void length(CORBA::ULong len);

Sets the maximum length of the sequence to the value contained in `len`.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>len</td>
<td>The new length for the sequence.</td>
</tr>
</tbody>
</table>
Chapter 14

Initialization interfaces and classes

This chapter describes the interfaces and classes that are provided for statically initializing ORB services, such as interceptors.

VISInit page 14-1
VISResolver page 14-3

VISInit

class VISInit

This abstract base class provides for the static initialization of service classes after the ORB and BOA have been initialized. By deriving your service class from VISInit and declaring it statically, you ensure that your service class instance will be properly initialized.

The ORB will invoke the VISInit::ORB_init and VISInit::BOA_init whenever the application calls CORBA::ORB_init or BOA_init methods. By providing your own implementations of these methods, you may add any needed initialization that must be performed for your service.

Include file

The vinit.h file should be included when you use this class.

VISInit constructors/destructors

VISInit();

This is the default constructor.
VISInit methods

VISInit(CORBA::Long init_priority);

This constructor creates a VISInit-derived object with the specified priority, which determines when it will be initialized relative to other VISInit-derived objects.

Internal VisiBroker classes which need to be initialized before user-defined classes have a negative priority value. The lowest priority value currently used by Visibroker internal classes is –10.

**Note**

You should set a priority value less than –10 if your class must be initialized before the VisiBroker internal classes.

If no priority value is specified, the default value is 0, which means that the class will be initialized after of the internal VisiBroker classes.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>init_priority</td>
<td>The initialization priority for this object. A negative priority value will cause this class to be initialized earlier. A positive priority value will cause this class to be initialized later.</td>
</tr>
</tbody>
</table>

virtual ~VISInit();

This is the default destructor.

VISInit methods

static void init_orb(int& argc, char *const * argv, CORBA::ORB_ptr orb);

This method will be called when the ORB is initialized. It will, in turn, invoke each of the ORB_init methods for each client interceptor factory that is to be initialized.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>argc</td>
<td>The count of arguments.</td>
</tr>
<tr>
<td>argv</td>
<td>An array of argument pointers.</td>
</tr>
<tr>
<td>orb</td>
<td>The ORB being initialized.</td>
</tr>
</tbody>
</table>

static void init_boa(int& argc, char *const * argv, CORBA::BOA_ptr boa);

This method will be called when the BOA is initialized. It will, in turn, invoke each of the BOA_init methods for each server interceptor factory this is to be initialized.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>argc</td>
<td>The count of arguments.</td>
</tr>
<tr>
<td>argv</td>
<td>An array of argument pointers.</td>
</tr>
<tr>
<td>boa</td>
<td>The BOA being initialized.</td>
</tr>
</tbody>
</table>
virtual void ORB_init(int& argc, char * const *argv, CORBA::ORB_ptr orb);

This method will be called when the ORB is initialized. Your implementation should provide for the initialization of the client-side interceptor factory that you wish to use.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>argc</td>
<td>The count of arguments.</td>
</tr>
<tr>
<td>argv</td>
<td>An array of argument pointers.</td>
</tr>
<tr>
<td>orb</td>
<td>The ORB being initialized.</td>
</tr>
</tbody>
</table>

virtual void BOA_init(int& argc, char * const *argv, CORBA::BOA_ptr boa);

This method will be called when the BOA is initialized. Your implementation should provide for the initialization of the server-side interceptor factory that you wish to use.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>argc</td>
<td>The count of arguments.</td>
</tr>
<tr>
<td>argv</td>
<td>An array of argument pointers.</td>
</tr>
<tr>
<td>boa</td>
<td>The BOA being initialized.</td>
</tr>
</tbody>
</table>

**VISResolver**

class VISResolver

This base class offers methods for resolving services that have been statically initialized with the ORB. You derive your own class from VISResolver for the service that you are implementing. You must provide an implementation of the resolve method.

This class also provides a static method for listing all of the services registered with an ORB.

**Include file**

The vinit.h file should be included when you use this class.

**VISResolver constructors/destructors**

VISResolver(const char *service_name);

Creates a VISResolver object for the specified service.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>service_name</td>
<td>The service name. This is the same name that client applications will use when invoking the ORB::resolve_initial_references method.</td>
</tr>
</tbody>
</table>
VISResolver methods

virtual ~VISResolver();

This is the default destructor.

VISResolver methods

static CORBA_ORB::ObjectIdList *list_services();

This static method returns a list of the services that are initialized for this ORB.

Note  ObjectIdList is simply a typedef for CORBA_StringSequence.

virtual CORBA::Object_ptr resolve(CORBA::ORB_ptr orb) = 0;

You provide an implementation of this method which returns a pointer to the top-level object that implements your service.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>orb</td>
<td>The ORB to be used to resolve the object for this service.</td>
</tr>
</tbody>
</table>

static CORBA::Object_ptr resolve(const char *service_name);

Returns a pointer to the object offering the specified service.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>service_name</td>
<td>The name of the service to be resolved.</td>
</tr>
</tbody>
</table>

const char *service_name() const;

Returns the name of the service associated with this VISResolver object.
This chapter describes the ORB Manager interfaces, which allow a client application to query and set the attributes of a server application.

This chapter includes the following major sections:

Adapter: page 15-1
Attribute: page 15-3
AttributeSet: page 15-4
Server: page 15-5

**Adapter**

class Adapter : public virtual AttributeSet

This interface allows you to obtain and set the attributes for a particular type of Object Adapter. The common attributes for each type of Object Adapter are described in Table 15.2. Attributes that are specific to the TPool adapter are described Table 15.2. Attributes common to the TPool and TSession adapters are described Table 15.3.

You obtain a Adapter object reference by invoking the Server::get_adapter method, described on page 15-6.

In addition to the methods described here, this interface inherits the methods defined for the base interface AttributeSet, described on page 15-4.
### Adapter attributes

**Table 15.1** Common attributes for TSingle, TPool, and TSession adapters

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OAactivatedConnections</td>
<td>readonly unsigned long</td>
<td>Current number of active client connections</td>
</tr>
<tr>
<td>OAactivatedRequests</td>
<td>readonly unsigned long</td>
<td>Current number of outstanding operations requests.</td>
</tr>
<tr>
<td>OArcvbufsize</td>
<td>unsigned long</td>
<td>Specifies the size of the buffer (in bytes) used to receive messages. If not specified, a default value (dependent upon your operating system) will be used.</td>
</tr>
<tr>
<td>OAsendbufsize</td>
<td>unsigned long</td>
<td>Specifies the size of the buffer (in bytes) used to receive messages. If not specified, a default value (dependent upon your operating system) will be used.</td>
</tr>
<tr>
<td>OAtcpnodelay</td>
<td>boolean</td>
<td>When set to 1, it sets all sockets to immediately send requests. The default value of 0 allows sockets to send requests in batches as buffers fill. This argument can be used to significantly impact performance or benchmark results.</td>
</tr>
</tbody>
</table>

**Table 15.2** Attributes for TPool adapters

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OAallocatedThreads</td>
<td>readonly unsigned long</td>
<td>Current number of allocated threads.</td>
</tr>
<tr>
<td>OAConnectionMax</td>
<td>unsigned long</td>
<td>Maximum number of incoming connections allowed.</td>
</tr>
<tr>
<td>OAConnectionMaxIdle</td>
<td>Read-Write</td>
<td>Number of seconds that connections are allowed to be idle before being shutdown. A value of 0 means that connections will never time-out.</td>
</tr>
<tr>
<td>OAshmsize</td>
<td>readonly unsigned long</td>
<td>The size of the send and receive segments (in bytes) in shared memory. If your client program and object implementation communicate via shared memory, you may use this option to enhance performance. This option is only supported on Windows platforms.</td>
</tr>
<tr>
<td>OAThreadMax</td>
<td>unsigned long</td>
<td>Maximum number of threads allowed.</td>
</tr>
<tr>
<td>OAThreadMaxIdle</td>
<td>unsigned long</td>
<td>Number of seconds a thread can exist without servicing any requests before it is returned to the system.</td>
</tr>
<tr>
<td>OAThreadMin</td>
<td>unsigned long</td>
<td>Minimum number of threads allowed.</td>
</tr>
</tbody>
</table>
Adapter methods

virtual char *adapter_id();

Returns the identifier for this adapter. The returned string will contain one of the following values:

- TSingle
- TPool
- TSession

virtual CORBA::ObjectSeq *persistent_objects();

Returns a sequence of object references to the persistent objects offered by this adapter. You can use this method to discover all the persistent objects currently active for the adapter.

Attribute

struct Attribute

This structure represents an attribute offered by a Server or Adapter object. This structure is used in the setting and retrieving of attributes.

```c
struct Attribute {
    string id;
    any value;
    boolean is_readonly;
};
```

Attribute members

CORBA::String_var id;

Uniquely identifies the attribute’s name for a particular class.

For a list of attribute identifiers for Adapter objects, see:

- Table 15.1 on page 15-2
- Table 15.2 on page 15-2
- Table 15.3 on page 15-3

---

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OAThreadStackSize</td>
<td>readonly unsigned long</td>
<td>Specifies the maximum thread stack size (in bytes) allowed.</td>
</tr>
</tbody>
</table>
The attribute’s value. See the previously listed tables for the appropriate data type for each attribute.

CORBA::Boolean is_readonly:

If set to TRUE, this attribute may be retrieved but not set. If set to FALSE, this attribute may be retrieved and set.

AttributeSet

class AttributeSet : public virtual CORBA::Object

This base interface is used to derive the Server and Adapter interfaces which are, in turn, used to manage ORB and Object Adapter attributes.

interface AttributeSet {
   exception InvalidAttributeId {};  
   exception InvalidAttributeType {};  
   exception InvalidAttributeValue {};  
   exception AttributeReadOnly {};  

   any get_attribute(  
       in string attribute_id)  
   raises (InvalidAttributeId);  

   void set_attribute(  
       in string attribute_id,  
       in any attribute_value)  
   raises (InvalidAttributeId,  
           InvalidAttributeType,  
           InvalidAttributeValue,  
           AttributeReadOnly );

   AttributeSeq get_all_attributes();
};

See also “Adapter” on page 15-1 and “Server” on page 15-5

AttributeSet methods

Virtual AttributeSeq 'get_all_attributes();

Returns a pointer to a sequence of all of the Attribute structures available for this object. Each Attribute structure contains the attribute’s identifier as a string, an Any containing the attribute’s type and value, and an indication of whether or not the attribute is read-only or may be altered using the set_attribute method.
virtual CORBA::Any *get_attribute( const char *attribute_id);

Returns a pointer to an Any containing the value of the attribute with the specified attribute_id.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>attribute_id</td>
<td>A string containing the identifier of the server attribute whose value is to be returned. The permitted values for this parameter depend on the object's type. For a list of Object Adapter attributes, see page 15-2.</td>
</tr>
</tbody>
</table>

This method may throw the following exceptions.

<table>
<thead>
<tr>
<th>Exception</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>InvalidAttributeId</td>
<td>The _attribute_id is not valid.</td>
</tr>
</tbody>
</table>

virtual void set_attribute( const char *attribute_id,
const CORBA::Any &attribute_value);

Sets the value of the attribute with the specified attribute_id.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>attribute_id</td>
<td>A string containing the identifier of the server attribute to be set. The permitted values for this parameter depend on the object's type. For a list of Object Adapter attributes, see page 15-2.</td>
</tr>
<tr>
<td>attribute_value</td>
<td>An Any containing the value to set for the attribute. The permitted values depend on the type of attribute. For a list of Object Adapter attribute values, see page 15-2.</td>
</tr>
</tbody>
</table>

This method may throw the following exceptions.

<table>
<thead>
<tr>
<th>Exception</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AttributeReadOnly</td>
<td>The attribute with the specified attribute_id is read-only and cannot be set.</td>
</tr>
<tr>
<td>InvalidAttributeId</td>
<td>The attribute_id is not valid.</td>
</tr>
<tr>
<td>InvalidAttributeType</td>
<td>The attribute_value object's TypeCode does not match the data type of the attribute being set.</td>
</tr>
<tr>
<td>InvalidAttributeValue</td>
<td>The attribute_value is invalid or out of range.</td>
</tr>
<tr>
<td>NO_PERMISSION</td>
<td>The server has not been started with the command-line option -ORBsecureSetAttr 0.</td>
</tr>
</tbody>
</table>

Server

class Server : public virtual AttributeSet

This interface allows you to obtain object references to the Adapter objects offered by a server application. You obtain a Server object reference by binding to an object and then invoking the Object::_resolve_reference method, specifying ORBServer as the value of the id parameter.
interface Server : AttributeSet {
    readonly attribute unsigned long process_id;
    readonly attribute long activation_policy;

    Adapter get_adapter(in string oa_id);
    AdapterSeq get_all_adapters();
    void shutdown();
};

Server methods

virtual CORBA::Long activation_policy();

Returns one of the following values which represents the activation policy of the server.

- 0 = SHARED_SERVER
- 1 = UNSHARED_SERVER
- 2 = SERVER_PER_METHOD

virtual Adapter_ptr get_adapter(const char *oa_id);

Returns the a pointer to an Adapter object for the Object Adapter with the specified identifier, which can then be used to set or obtain various attributes.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>_oa_id</td>
<td>Identifier of the Object Adapter for which you wish to obtain a management interface. The string must contain one of the following values: TSingle TPool TSession</td>
</tr>
</tbody>
</table>

virtual AdapterSeq *get_all_adapters();

This method returns a pointer to a sequence of Adapter object for all of the Object Adapters that the server is using.

virtual CORBA::ULong process_id();

Returns the process identifier belonging to the server or the vbj process used to start the server.

virtual void shutdown();

This method causes the server to shutdown and any object references associate with that server will no longer be valid.

Note: If a client application invokes this method on a server that was not started with the -ORBsecureShutdown 0 option, a CORBA::NO_PERMISSION exception will be thrown.
This appendix describes the options that may be set for the Basic Object Adaptor, the Object Request Broker, and the Location Service. It includes the following major sections:

- BOA_init() method  page A-1
- BOA options  page A-2
- ORB_init() method  page A-3
- ORB options  page A-4
- Location service options  page A-6

**BOA_init() method**

The `BOA_init()` method is used by your object implementation to set such options as the desired thread policy or the TCP/IP port number to be used. These parameters are passed as arguments to the object implementation’s server process when it is started.

```
prompt> server -OAipAddr 199.99.129.33 -OAport 19000
```

Code sample A.1 shows the definition of the `BOA_init()` method and the arguments it accepts. Like the `ORB_init()` method, the `argc` and `argv` parameters passed to `BOA_init()` are the same arguments that were passed to your object implementation’s main routine. The `BOA_init()` method will ignore any arguments it does not recognize.
**BOA options**

**Code sample A.1**  
BOA_init() method

```cpp
class CORBA {
...
    static BOA_ptr BOA_init(int& argc, char *const *argv,
                            const char *boa_identifier = "VIS_BOA");
...
};
```

After this method has been invoked, all the recognized BOA arguments will be stripped from the original parameter list so that they will not interfere with any other argument processing that your object implementation requires.

---

**BOA options**

Table A.1 summarizes the BOA_init() options.

<table>
<thead>
<tr>
<th>Type/Value pair</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>-OAagent &lt;0</td>
<td>1&gt;</td>
</tr>
<tr>
<td>-OAconnectionMax &lt;#&gt;</td>
<td>Specifies the maximum number of connections allowed when -Oaid TSession is selected.</td>
</tr>
<tr>
<td>-OAconnectionMaxIdle &lt;#&gt;</td>
<td>Specifies the time (in seconds) which a connection can idle without any traffic. Connections that idle beyond this time can be shutdown by VisiBroker. By default, this is set to 0 seconds which means that connections will never automatically timeout. This option should be set for Internet applications.</td>
</tr>
<tr>
<td>-OAgarbageCollectTimer &lt;#&gt;</td>
<td>Specifies the time (in seconds) that the adapter waits before checking for idle connections and threads to be cleaned up. The default period is 30 seconds. The adapter checks for threads that have been idle for longer than the time specified by -OAthreadMaxIdle and for connections that have been idle for longer than the time specified by -OAconnectionMaxIdle.</td>
</tr>
<tr>
<td>-OAlid &lt;TPool</td>
<td>TSession&gt;</td>
</tr>
<tr>
<td>-OAipAddr &lt;hostname</td>
<td>ip_address&gt;</td>
</tr>
</tbody>
</table>
The **ORB_init()** method is used by your client program to set such options as the IP address and port number of the Smart Agent to be used. These parameters are passed as arguments to the client program process when it is started.

```
prompt> client -ORBagentAddr 199.99.129.33 -ORBagentPort 19000
```

Code sample `A.2` shows the definition of the **ORB_init()** method and the arguments it accepts. Like the **BOA_init()** method, the `argc` and `argv` parameters passed to **ORB_init()** are the same arguments that were passed to your client.
ORB options

program’s main routine. The ORB_init() method will ignore any arguments it
does not recognize.

Code sample A.2  ORB_init() method definition

```cpp
class CORBA {
    ...
    static ORB_ptr ORB_init(int& argc, char *const *argv,
        const char *orb_id = (char*)NULL);
    ...
};
```

After this method has been invoked, all the recognized ORB arguments will be
stripped from the original parameter list so that they will not interfere with any
other argument processing that your client program requires.

ORB options

All but one of the ORB options take the form of type-value pairs. Table A.2
summarizes the ORB_init() options.

Table A.2  ORB_init() options used by client programs

<table>
<thead>
<tr>
<th>Type/Value pair</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>-ORBagent &lt;0</td>
<td>1&gt;</td>
</tr>
<tr>
<td>-ORBagentAddr &lt;hostname</td>
<td>ip_address&gt;</td>
</tr>
<tr>
<td>-ORBagentPort &lt;port_number&gt;</td>
<td>Specifies the port number of the Smart Agent. This option can be useful if multiple ORB domains are required, as described in the VisiBroker for C++ Programmer’s Guide, Chapter 11, “Smart Agent architecture.” If not specified, a default port number of 14000 will be used.</td>
</tr>
<tr>
<td>-ORBbackCompat &lt;0</td>
<td>1&gt;</td>
</tr>
<tr>
<td>-ORBbackdii &lt;0</td>
<td>1&gt;</td>
</tr>
</tbody>
</table>
### Using command-line options

#### ORB options

<table>
<thead>
<tr>
<th>Type/Value pair</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>-ORBconnectionMax &lt;#&gt;</td>
<td>Specifies the maximum number of outgoing connections that are allowed. If you do not specify this option, the default is to allow an unlimited number of connections.</td>
</tr>
<tr>
<td>-ORBconnectionMaxIdle &lt;#&gt;</td>
<td>This specifies the number of seconds that an outgoing connection can idle before it is shutdown by VisiBroker. By default, this is set to 0, which means that connections will never time-out. This option should be set for Internet applications.</td>
</tr>
<tr>
<td>-ORBir_name &lt;ir_name&gt;</td>
<td>Specifies the name of the Interface Repository to be accessed when the Object::get_interface() method is invoked on object implementations.</td>
</tr>
<tr>
<td>-ORBir_ior &lt;ior_string&gt;</td>
<td>Specifies the IOR of the Interface Repository to be accessed when the Object::get_interface() method is invoked on object implementations.</td>
</tr>
<tr>
<td>-ORBnullString &lt;0</td>
<td>1&gt;</td>
</tr>
<tr>
<td>-ORBrcvbufsize &lt;buffer_size&gt;</td>
<td>Specifies the size of the TCP buffer (in bytes) used to receive responses. If not specified, a default buffer size will be used. This argument can be used to significantly impact performance or benchmark results.</td>
</tr>
<tr>
<td>-ORBsecuresetattr &lt;0</td>
<td>1&gt;</td>
</tr>
<tr>
<td>-ORBsecureShutdown &lt;0</td>
<td>1&gt;</td>
</tr>
<tr>
<td>-ORBsendBind &lt;0</td>
<td>1&gt;</td>
</tr>
</tbody>
</table>

Table A.2  ORB_init() options used by client programs (continued)
Location service options

These command-line options can be used by your client program to control various Location Service features. When your client application invokes the ORB_init method, the Location Services will be initialized and will receive any command-line arguments you have specified. Command-line options for the Location Service will be processed and stripped from the argument list. All unrecognized options will be ignored.

As with the command line options for the BOA and ORB, the Location Service options take the form of type-value pairs.

```
prompt> client -LOCdebug 1 -LOCtimeout 10 -LOCverify 0
```

The table below summarizes the Location Service command-line options.

### Table A.2 ORB_init() options used by client programs (continued)

<table>
<thead>
<tr>
<th>Type/Value pair</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>-ORBsendbufsize &lt;buffer_size&gt;</td>
<td>Specifies the size of the TCP buffer (in bytes) used to send client requests. If not specified, a default buffer size will be used. This argument can be used to significantly impact performance or benchmark results.</td>
</tr>
<tr>
<td>-ORBshmsize &lt;size&gt;</td>
<td>Specifies the size of the send and receive segments (in bytes) in shared memory. If your client program and object implementation communicate via shared memory, you may use this option to enhance performance. This option is only supported on Windows platforms.</td>
</tr>
<tr>
<td>-ORBtcpNoDelay &lt;0</td>
<td>1&gt;</td>
</tr>
</tbody>
</table>

### Table A.3 Location Service command-line options

<table>
<thead>
<tr>
<th>Type/Value Pair</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>-LOCdebug &lt;0</td>
<td>1&gt;</td>
</tr>
<tr>
<td>-LOCtimeout &lt;seconds&gt;</td>
<td>Indicates the number of seconds to wait for a response from a server when verifying the existence of an object. This option is only used when -LOCverify has been set to one. The default value is one second.</td>
</tr>
<tr>
<td>-LOCverify &lt;0</td>
<td>1&gt;</td>
</tr>
</tbody>
</table>
Symbols

... ellipsis 1-2
[ ] brackets 1-2
| vertical bar 1-2

A
accessing
system exceptions 6-15
the interface repository 7-28
user exceptions 6-15
activating
object implementations 5-3
activation policy 15-6
ActivationImplDef 8-1
Activator 8-2
Adapter 15-1
methods 15-3
TPool 15-1
TSession 15-1
Agent
methods 13-2 to 13-6
AliasDef 7-2
Any 6-2, 6-19
extraction operators 6-3
initialization operators 6-3
methods 6-2 to 6-3
arguments
<Code Term>
type_code_info 2-4
-client_ext 2-2
corba_inc 2-2
-export 2-2
-export_skel 2-3
-hdr_suffix 2-3, 2-4
-impl_base_object 2-3
-incl_files_code 2-3
-LOCdebug A-6
-LOCtimeout A-6
-LOCverify A-6
-no_excep_spec 2-2
-no_exceptions 2-3
-no_tie 2-3
-OAid A-2
-OAipaddr A-2
-OAlocalIPC A-2, A-3
-OApartment A-3
-OArvcbufsize A-3
-OAsendbufsize A-3

-OB disproportionately A-3
-OAThreadMax A-3
-OAThreadMaxIdle A-3
-OAThreadStackSize A-3
-ORBagentaddr A-4
-ORBagentport A-4
-ORBbackcompat A-4
-ORBbackddi A-4
-ORBir_ior A-5
-ORBir_name A-5
-ORBnullstring A-5
-ORBrcvbufsize A-5
-ORBsendbufsize A-6
-ORBshminsize A-3, A-6
-pty 2-3
-server_ext 2-4
-src_suffix 2-4
-version 2-4
-virtual_impl_inh 2-4
array slice
passing parameters for multidimensional arrays 3-14
ArrayDef 7-2
arrays 3-13
managed types 3-14
memory management 3-15
type-safe 3-15
Attribute
members 15-3
AttributeDef 7-3
AttributeDescription 7-4
AttributeMode 7-4
AttributeSet 15-4
methods 15-4
B
backward compatibility
-ORBbackddi A-4
Basic Object Adaptor
(BOA) 5-3
bind options
collection_timeout 5-2
defer_bind 5-2
enable_rebind 5-2
max_bind_retries 5-2
receive_timeout 5-2
send_timeout 5-2
binding clients to objects 5-12
BindOptions 5-1
BOA  A-2, A-3
(Basic Object Adaptor) 5-3,
8-1, 8-2, 8-3, 8-6, 8-14
include file 8-1, 8-2, 8-3,
8-6, 8-8, 8-9, 8-13, 8-14
methods 5-3 to 5-7, 8-2,
8-15
VisiBroker extensions 5-7
to 5-8
BOA_init() method A-1
A
- OAConnectionMax A-2
- OAConnectionMaxIdle A-2
- OApartment A-3
- OArvcbufsize A-3
- OAsendbufsize A-3
- OATcpmodelay A-3
- OAThreadMax A-3
- OAThreadMaxIddle A-3
- OAThreadStackSize A-3
- passing arguments A-1
ORB_init() method A-3
BOA_init() method
- OAConnectionMax A-2
- OAConnectionMaxIdle A-2
- OAid A-2
- OAipAddr methods
  BOA_init()
    - OAipAddr A-2
- OApartment A-3
- OAsendbufsize A-3
- OATcpmodelay A-3
- OAThreadMax A-3
- OAThreadMaxIddle A-3
- OAThreadStackSize A-3
- passing arguments A-1
BOA_init() method A-1

C++
classes 2-2
compiler 2-1
C++ language exceptions 6-15
CancelRequestHeader 11-2
ChainUntypedObjectWrapper-Factory 10-13
EstructDef 7-15
Event handling
  client-side 9-1
  implementation-side 9-6
Exception 5-11, 5-28, 5-30
ExceptionDef 7-16
ExceptionDescription 7-16
  -export argument 2-2
  -export_skel argument 2-3
extended methods
  BOA
    (Basic Object Adaptor) 5-7 to 5-8
-export_skel 2-3
-hdr_suffix 2-3, 2-4
-impl_base_object 2-3
-incl_files_code 2-3
-no_excep_spec 2-2
-no_exceptions 2-3
-no_tie 2-3
-ptype 2-3
-server_ext 2-4
-src_suffix 2-4
-type_code_info 2-4
-version 2-4
-virtual_impl_inh 2-4
idl2cpp tool 2-1
idl2ir
  command info 2-4
  description 2-4
IDLType 7-18, 7-21
  include file 7-18
  methods 7-18
  -impl_base_object argument 2-3
  implementation
  inheritance
  using virtual
    inheritance 2-3
ImplementationDef 8-6
ImplementationStatus
  struct 8-8
ImplEventHandler 9-6
  -incl_files_code argument 2-3
  include file
    BOA
      (Basic Object Adaptor) 8-2, 8-15
      BOA_init() A-1
      -OAConnectionMax A-2
      -OAconnectionMax A-2
      -OAConnectionMax-Idle A-2
      Debugging A-6
      time-out A-6
      TriggerDesc 13-7
      verifying object
      existence A-6
LocationService
  Agent 13-1
  Fail 13-7
  Seq 13-9
  SeqSeq 13-10
  TriggerHandle 13-8
L
  linking errors 2-2
  LOCAL_SCOPE 5-6
  LocateReplyHeader 11-3
  LocateRequestHeader 11-3
G
  generated classes
    _ptie_ 4-3
    _sk_ 4-3
    _st_ 4-2
    _tie_ 4-3
    _var 4-3
  generating
    a String_var class 3-2
    Generating C++ code 2-1
    GLOBAL_SCOPE 5-6
H
  HandlerRegistry 9-4
  -hdr_suffix argument 2-3, 2-4
  hostname
    BOA, specifying for A-2
  LOCAL_SCOPE 5-6
I
  IDL
    arrays 3-13
    complex data types 3-6
    constants 3-3
    primitive data types 3-1
    to C++ language
    mapping 3-1
  IDL compiler 2-1
  idl2cpp compiler
    -arg 2-2
    -client_ext 2-2
    -corba_inc 2-2
    -export 2-2
    managed types
      arrays 3-14
      for sequences 3-12
      for unions 3-10
      manipulating object references 5-12
      mapping
        IDL enumerations to C++ 3-4
        IDL modules to C++ namespace 3-6
        IDL type definitions to C++ definitions 3-5
      MarshallInBuffer
        methods 12-2 to 12-5
      MarshallOutBuffer
        methods 12-6 to 12-8
      memory management
        arrays 3-15
        for sequences 3-12
        for structures 3-8
      memory management semantics 5-9
      MessageHeader 11-1
      methods
        BOA
          (Basic Object Adaptor) 5-3 to 5-7, 8-2, 8-15
          BOA_init() A-1
          -OAConnectionMax A-2
          -OAconnectionMax A-2
          -OAConnectionMax-Idle A-2
-OAid A-2
- OAlocalIPC A-2, A-3
- OAport A-3
- OArcvbufsize A-3
- OAsendbufsize A-3
- OAtcpNoDelay A-3
- OAThreadMax A-3
- OAThreadMaxIdle A-3
- OAThreadStackSize A-3

passing arguments A-1

change_implementation in
OAD 8-9

Contained 7-7
Container 7-9
Context 5-9 to 5-11
destroy_on_unregister in
OAD 8-10
extraction methods in
Any 6-9
get_implementation in
OAD 8-10
get_status in OAD 8-11
get_status_all in OAD 8-11
get_status_interface in
OAD 8-11
IDLType 7-18
insertion methods in
Any 6-10
InterfaceDef 7-19
IRObject 7-22
NamedValue 6-19
NVList 6-21
OperationDef 7-23
ORB 5-18, 5-26
ORB_init() A-3
-ORBAgentaddr A-4
-ORBAgentport A-4
-ORBBackcomapt A-4
-ORBBackddii A-4
-ORBir_ior A-5
-ORBir_name A-5
-ORBNullstring A-5
-ORBcvbufsize A-5
-ORBsendbufsize A-6
-ORBshmsize A-3, A-6

passing arguments A-3
Principal 5-27
reg_implementation in
OAD 8-11
Repository 7-28
Request 6-24
ServerRequest 6-28
SystemException 5-28 to
5-29, 5-30

unreg_implementation in
OAD 8-12

unreg_interface in OAD 8-12
unregister_all in OAD 8-12
ModuleDescription 7-22
ModuleFed 7-22
modules

mapping IDL modules to
C++ namespace 3-6
multithreaded
applications 6-16

N

NamedValue 6-19, 6-20
methods 6-19 to 6-20
-no_excep_spec argument 2-2
-no_exceptions argument 2-3
-no_tie argument 2-3
NVList 6-19, 6-20

methods 6-21 to 6-23

( Object Broker Request ) 5-17
methods 5-18 to 5-26

orb.lib 2-2
ORB_init() method
-ORBAgentaddr A-4
-ORBAgentport A-4
-ORBBackcomapt A-4
-ORBBackddii A-4
-ORBir_ior A-5
-ORBir_name A-5
-ORBNullstring A-5
-ORBcvbufsize A-5
-ORBsendbufsize A-6
-ORBshmsize A-3, A-6

passing arguments A-3
ORB_init() method A-3
-ORBAgentaddr argument A-4
-ORBBackddii argument A-4
ORBconnectionMax
ORB initialization
option A-5

P

parameter passing
for multi-dimensional
arrays 3-14
ParameterDescription 7-26
ParameterMode 7-26
persistent objects 4-3
platform designation with
icons 1-3
port number
BOA, used by A-3
Smart Agent, used by A-4
primitive data types 3-1
PrimitiveDef 7-27
PrimitiveKind 7-27
Principal 5-27
methods 5-27
principal
IDL interface 3-16
ProfileBody 11-5
Programmer Tools
general information 2-1
Programmer tools
dl2cpp 2-1
programming interface
ActivationImplDef 8-1
Activator 8-2
Adapter 15-1
Agent 13-1
AliasDef 7-2
Any 6-2, 6-19
ArrayDef 7-2
structures
  fixed-length 3-7
  memory management 3-8
  variable length 3-8
stubs 4-2
system exception classes 5-11
System Exceptions
  defined 5-29
SystemException 5-28, 5-30
  methods 5-28 to 5-29, 5-30

T
TaggedProfile 11-6
TCKind 6-29
thread policies
  choosing for BOA  A-2
tools
  idl2ir 2-4
TPool 5-6, 5-8, A-3
TPool_liop 5-6
TriggerDesc 13-7
TriggerHandler
  methods 13-9
TSession 5-6, 5-8, A-3
TSingle 5-6
type definitions
  mapping IDL types to C++
  type definitions 3-5
-type_code_info argument 2-4
TypeCode
  constructors 6-31 to 6-33
  methods 6-34 to 6-36
TypedefDef 7-31
TypeDescription 7-31
types
  complex data types 3-6
  IDL primitive data types 3-1
  primitive 3-1
  strings 3-2
type-safe
  arrays 3-15
typographic conventions 1-2

U
UnionDef 7-32
UnionMember 7-33
unions
  managed types 3-10
UntypedObjectWrapper 10-15
UntypedObjectWrapperFactory 10-16
user exception classes 5-11

V
Var classes 4-3
variable length
  structures 3-8
-version argument 2-4
VersionSpec 7-33
-virtual_impl_inh argument 2-4
VISBindInterceptor 10-2
VISChainBindInterceptor 10-5
VISChainClientInterceptor-Factory 10-6
VISChainServerInterceptor-Factory 10-7
VISClientInterceptor 10-7
VISClientInterceptor-Factory 10-11
VISClosure 10-12
VISClosureData 10-12
VISInit 14-1, 14-3
  methods 14-1 to 14-3
VISResolver
  methods 14-3 to 14-4
VISServerInterceptor 10-17
VISServerInterceptor-Factory 10-22

W
WstringDef 7-34