Spectra ORB C++ Edition

IDL Guide

PrismTech

Spectra ORB

C++ Edition

IDL Guide

PrismTech

Part Number: EORBCPP-IDLG

Doc Issue 29A, 24 January 2014

CONTENTS

Table of Contents

Preface

About the IDL Guide ix
Contacts x

About IDL

Chapter 1 **Introduction to IDL** 3
1.1 **Defining the Object's Interface** 3

Chapter 2 **Concepts and Features** 5
2.1 **IDL Language Basics** 5
2.1.1 IDL Conventions 6
2.1.1.1 Comments 6
2.1.1.2 Identifiers 6
2.1.2 The typedef Mechanism 7
2.1.3 Data Types 7
2.1.3.1 Basic Types 7
2.1.3.2 Constructed Types 8
2.1.3.3 Template Types 9
2.1.3.4 Arrays 10
2.1.3.5 The any Type 11
2.1.4 Constant Types 11
2.1.5 Interfaces 12
2.1.5.1 Operations 12
2.1.5.2 Attributes 14
2.1.5.3 Exceptions 14
2.1.5.4 Inheritance 15
2.1.6 Modules 16
2.1.6.1 The CORBA Module 17

C++ Edition Specifics

Chapter 3 **Compiling an IDL Specification** 21
3.1 **Using the IDL Compiler** 21
3.1.1 Command Syntax 21
3.2 **IDL Compiler Output** 24
3.2.1 C++ Files Produced 25
3.2.2 Using the IDL Compiler Output 26
3.2.3 Creating a DLL 26

Chapter 4 **IDL to C++ Mapping** **29**

4.1 **Namespaces** **29**
4.1.1 Scoped Names and Namespaces 29
4.2 **Modules** **30**
4.3 **Interfaces** **31**
4.3.1 The _var Type 32
4.3.2 Object References 33
4.3.3 Object Reference Operations 33
4.3.3.1 Common Operations 33
4.3.3.2 Widening Object References 35
4.3.3.3 Narrowing Object References 36
4.4 **Data Type Mappings** **37**
4.4.1 Constants 37
4.4.2 Typedefs 38
4.4.3 Basic Data Types 38
4.4.4 Enums 39
4.4.5 String Types 39
4.4.5.1 String_var 40
4.4.6 Structured Types 41
4.4.6.1 Fixed versus Variable-Length 41
4.4.7 T_var Types 42
4.4.8 Struct Types 44
4.4.8.1 Examples 45
4.4.9 Union Types 46
4.4.10 Sequences 49
4.4.10.1 Unbounded Sequences 49
4.4.10.2 Bounded Sequences 53
4.4.10.3 The Sequence _var Type 55
4.4.11 Array Types 56
4.4.12 Any Types 59
4.4.12.1 Handling Typed Values 60
4.4.12.2 Insertion Into Any 60
4.4.12.3 Extraction from Any 62
4.4.12.4 Handling boolean, octet, char, and unbounded string 63
4.4.12.5 Arrays in an Any 66
4.4.12.6 Extracting to Object 67
4.4.12.7 Handling Untyped Values 67
4.4.12.8 Any_var and Any_out 68
4.5 **TypeCodes** **69**
4.6 **Exceptions** **71**
4.7 **Operations and Attributes** **76**

4.7.1 Operations 76
4.7.2 Attributes 76
4.7.3 Parameters 76
4.7.3.1 In Parameters 78
4.7.3.2 Inout Parameters 79
4.7.3.3 Out Parameters 79
4.7.3.4 Return Values 80
4.7.3.5 Primitives 80
4.7.3.6 Fixed-Length structs and unions 81
4.7.3.7 Fixed-length arrays 83
4.7.3.8 Strings 84
4.7.3.9 Object References 85
4.7.3.10 Variable-Length structs and unions 87
4.7.3.11 Variable-Length arrays 89
4.7.3.12 Sequence and Anys 91

Index **97**

Preface

About the IDL Guide

The *Spectra ORB C++ Edition IDL Guide* provides instructions and background information needed to understand the IDL language, data types, IDL to C++ mappings, and how to use the Spectra ORB C++ Edition IDL to C++ compiler.

The *IDL Guide* is intended to be used with the *User Guide*, as well as the other documents included with Spectra ORB C++ Edition: please refer to the *Product Guide* for a complete list of documents.

Intended Audience

The *IDL Guide* is intended to be used by developers and engineers, working in a distributed computing environment using Spectra ORB, who have a good level of knowledge and experience of CORBA and IDL.

Organisation

The *IDL Guide* is organized as follows:

- Chapter 1, *Introduction to IDL*, introduces the OMG's Interface Definition Language (IDL).
- Chapter 2, *Concepts and Features*, describes the IDL basics and provides essential background information.
- Chapter 3, *Compiling an IDL Specification* describes how to write IDL files.
- Chapter 4, *IDL to C++ Mapping*, describes the OMG's IDL to the C++ language mapping for the ORB.

Conventions

The conventions listed below are used to guide and assist the reader in understanding the IDL Guide.

⚠ Item of special significance or where caution needs to be taken.

i Item contains helpful hint or special information.

WIN Information applies to Windows (*e.g.* XP, Vista, Windows 7) only.

UNIX Information applies to Unix-based systems (*e.g.* Solaris) only.

C C language specific.

C++ C++ language specific.

Java Java language specific.

Hypertext links are shown as *blue italic underlined*.

On-Line (PDF) versions of this document: Items shown as cross references to other parts of the document, *e.g.* *Contacts* on page x, are hypertext links: jump to that section of the document by clicking on the cross reference.

```
% Commands or input which the user enters on the
  command line of their computer terminal
```

`Courier` fonts indicate programming code and file names.

Extended code fragments are shown as small `Courier` font in shaded boxes:

```
NameComponent newName[] = new NameComponent[1];

// set id field to “example” and kind field to an empty string
newName[0] = new NameComponent (“example”, “”);
```

Italics and ***Italic Bold*** are used to indicate new terms, or emphasise an item.

Arial Bold is used to indicate user-related actions, *e.g.* **File > Save** from a menu.

Step 1: One of several steps required to complete a task.

Contacts

PrismTech can be reached at the following contact points for information and technical support.

USA Corporate Headquarters	**European Head Office**
PrismTech Corporation 400 TradeCenter Suite 5900 Woburn, MA 01801 USA	PrismTech Limited PrismTech House 5th Avenue Business Park Gateshead NE11 0NG UK
Tel: +1 781 569 5819	Tel: +44 (0)191 497 9900 Fax: +44 (0)191 497 9901

Web: *http://www.prismtech.com*
Technical questions: *crc@prismtech.com* (Customer Response Center)
Sales enquiries: *sales@prismtech.com*

About IDL

CHAPTER

1 Introduction to IDL

The Interface Definition Language (IDL) enables interfaces to be defined which are independent of any particular programming language. This enables the components of CORBA-based distributed applications to be written in whichever programming language is most appropriate, where different components of the same application can even be written in different languages. For example, a client could be written in Java and a server in C, C++, or even COBOL.

This capability provides enormous power and flexibility, especially where legacy applications must be made to communication with new applications or components which have been written in a different language.

In CORBA, the object's interface defines the operations the object provides and how a client interacts with the object. The interface represents the contract between an object and a client that attempts to use that object. The interface defines the object's reference and allows an object to declare its type and the names of its inherited types. It also provides the names of the operations it supports and the parameters and return types of those operations.

To implement an object described in IDL, the interface description must be translated into the appropriate programming language constructs as defined by CORBA's IDL mapping for that language. Translation is performed by an IDL compiler supplied with the ORB. The IDL compiler is described in *Using the IDL Compiler on page 19**.*

The IDL to C language mapping is compliant with the OMG C Language Mapping Specification, OMG document 99-07-35.

1.1 Defining the Object's Interface

In IDL, an object's interface consists of a set of named operations and the parameters supplied to those operations. The following example shows how IDL would define the interface of an object in a simple application:

```
interface SimpleObject
{
   boolean oneOperation (in string aString);
   void anotherOperation (in string anotherString);
};
```

A description of an interface in IDL is called an IDL *specification* of that interface. IDL specifications should be created in a source file with a `.idl` file extension.

IDL source files are passed to the ORB's IDL compiler, to generate new files with programming-language constructs and implementations that can be used in an application. The IDL interface construct maps directly to the programming language's construct for an object reference:

The following sections describe IDL and the constructs you can use to define an object's interface.

CHAPTER

2 Concepts and Features

The OMG's Interface Definition Language (IDL) is a modelling language for defining interfaces. The interface definitions created are platform and language independent. The IDL interface definitions enable applications, which have been implemented in different programming languages, for example C, C++, Java, or even COBOL, to transparently communicate with each other through ORB middleware.

IDL compilers generate programming code for specific languages (such C, C++, Java, or COBOL) using the IDL interface definitions mentioned above. An individual IDL compiler is designed to generate programming code for a specific language and in accordance with an appropriate mapping specification defined by the OMG.

IDL compilers use the interface definitions to create the appropriate classes, methods, operations, parameters and attributes needed for the specific programming language used by the developer. The developers use the IDL-generated files as the basis for implementing client and server components of their distributed, CORBA-based applications.

This section describes the basics of the IDL language and how to use it to create programming language-specific applications. The complete description of the syntax and semantics of IDL are given in the OMG's Common Object Request Broker Architecture and Specification document.

2.1 IDL Language Basics

The IDL syntax is similar to C++ syntax. The key differences between IDL and C++ syntax are:

- a function return type is mandatory
- a name must be supplied with formal parameter in an operation declaration
- a parameter list consisting of a single void token does not act as an empty parameter list
- tags are required for structures, discriminated unions, and enumerations
- integer types cannot be declared as `int` or `unsigned`; they must be explicitly declared as `short`, `long`, or `long long`
- `char` types cannot be qualified as `signed` or `unsigned`

2.1.1 IDL Conventions

The Interface Definition Language uses the conventions described below.

2.1.1.1 Comments

There are two comment delimiters in IDL:

- The comment pair `/*` and `*/` is used to delimit comments that span multiple lines.
- The double slash `//` begins a comment that is terminated at the end of a line. The comment may be placed after a code statement on the same line or on a line by itself.

Comments do not nest.

2.1.1.2 Identifiers

Identifiers are names for IDL definitions such as constants, types, and operations. An identifier is an arbitrarily long sequence of letters, digits, and the underscore (_) character. Identifiers must begin with a letter.

In this example, `MaxVal` is an IDL identifier for a long integer:

```
long MaxVal;
```

Identifiers are scoped. See *Modules* on page 16 for information on how scoping works with identifiers.

Identifiers are case sensitive in that a given identifier must be spelled identically with respect to case throughout an IDL specification. However, using two different identifiers which differ only in case will produce a compilation error.

IDL reserves a set of keywords that cannot be used as identifiers. These reserved keywords are shown in Table 1, *IDL Keywords*.

Table 1 IDL Keywords

abstract	double	local	raises	truncatable
any	enum	long	readonly	typedef
attribute	exception	module	sequence	unsigned
boolean	factory	native	short	union
case	FALSE	Object	string	ValueBase
char	fixed	octet	struct	valuetype
const	float	oneway	supports	void
context	in	out	switch	wchar
custom	inout	private	TRUE	wstring
default	interface	public		

PRISMTECH

2.1.2 The typedef Mechanism

Use the `typedef` mechanism to name a data type. A `typedef` definition begins with the keyword `typedef`, followed by the data type and `typedef` name. You can name basic data types, constructed data types, and template types using `typedef` declarations. `typedefs` are typically used to name template types, such as sequences and arrays.

2.1.3 Data Types

IDL supports the basic data types such as char, short, long, and float; structured types such as structs, unions, and enumerations; template types such as sequences, arrays, and strings; and structured exceptions. The interface type can also be used as an operation parameter. An interface argument is passed as an object reference that refers to the instance of the object implementation it represents. CORBA defines an `any` type that can represent any IDL type and can be used as an argument or value type wherever multiple different types are acceptable. You can use typedefs or create an alias for any of these types.

An IDL compiler for a specific programming language maps basic IDL data types to the appropriate native data types of that programming language.

2.1.3.1 Basic Types

Table 2, *Basic IDL Data Types* lists the basic IDL data types.

Table 2 Basic IDL Data Types

IDL Identifier	Type	Description
`any`	Any type	Self-describing values that can express any IDL type
`boolean`	Boolean type	1 or 0 (zero)
`char`	Char type	8-bit ISO Latin 1 characters
`double`	Floating-point Type	IEEE double-precision floating-point numbers
`float`	Floating-point Type	IEEE single-precision floating-point numbers
`long`	32-bit integer	-2^31 ... 2^31-1
`long double`	Floating-point type	IEEE double-extended floating-point numbers
`long long`	64-bit integer	-2^63 ... 2^63-1

Table 2 Basic IDL Data Types (Continued)

IDL Identifier	Type	Description
`octet`	Octet type	8-bit quantity that is guaranteed not to undergo any conversion during communication
`short`	16-bit integer	-2^15 ... 2^15-1
`unsigned long`	32-bit integer	0 ... 2^32-1
`unsigned long long`	64-bit integer	0 ... 2^64-1
`unsigned short`	16-bit integer	0 ... 2^16-1

2.1.3.2 Constructed Types

IDL provides three constructed data types: structures, unions, and enumerations.

2.1.3.2.1 Structures

A structure allows related data to be grouped under a single name. Structures are declared with the keyword `struct`, followed by an identifier and a list of members enclosed in braces. Each struct member must be an IDL type[1].

This example of an IDL structure named `date` has three members: two of type `unsigned short` and one of type `short`.

```
struct date
{
   unsigned short month;
   unsigned short day;
   short year;
};
```

2.1.3.2.2 Discriminated Unions

A union groups items of different types and sizes, but only one member of the union has a useful value at any one time. This implies a saving in the amount of storage allocated to a union.

Discriminated unions use a discriminator (or switch) to help keep track of which union member has the most current value assigned to it. Each member of an IDL discriminated union has an associated case label which must match (or be castable to) the type of the switch field. The switch field uses the case label to determine which union member to use. An optional member with a `default` case label can be used. Access to the switch and the related element is language-mapping dependent.

1. A *type* means all types that can be defined as well as basic types.

An IDL discriminated union is declared with the keyword `union`, followed by an identifier for the union. This is followed by the keyword `switch` and a switch type enclosed in parentheses. Finally, a list of members that have case labels is enclosed in braces. Each union member must be an IDL type.

The switch type can be long, long long, short, unsigned long, unsigned long long, unsigned short, char, boolean, or enum.

This example shows an IDL discriminated union named `token`.

```
union token switch (long)
{
   case 1: char cval;
   case 2: float fval;
   case 3: double dval;
   default: long lval;
};
```

2.1.3.2.3 Enumerations

An enumeration is an ordered list of identifiers, referred to as enumerators. Enumerations are declared with the keyword `enum`, followed by an identifier and a comma-separated list of enumerators enclosed in braces.

The order in which the identifiers are named defines their relative order. This order allows two enumerators to be compared. By default, the first enumerator has a value of 0; the value of each subsequent enumerator is incremented by one.

This example shows an IDL enumerated type named `workday`.

```
enum workday { Monday, Tuesday, Wednesday, Thursday, Friday };
```

2.1.3.3 Template Types

IDL provides template data types: sequences, strings, and fixed.

2.1.3.3.1 Sequences

Example

A sequence is a one-dimensional array that can be declared with an optional maximum size. If the sequence is declared with a maximum size, it is referred to as a bounded sequence; if no maximum size is specified, it is referred to as an unbounded sequence. The length of a sequence can change dynamically but the length of a bounded sequence cannot exceed the maximum size fixed at compile time. Sequence elements can be any of the IDL types.

This example shows a bounded sequence. The keyword `typedef` names the sequence data type.

```
typedef sequence<long, 64> vec64;
```

This example shows an unbounded sequence named `vec`.

```
typedef sequence<long> vec;
```

2.1.3.3.2 Strings

A string is a one-dimensional array of 8-bit ISO Latin1 characters that can be declared with an optional maximum length. If the string is declared with a maximum length, it is referred to as a bounded string; if no maximum length is specified, it is referred to as an unbounded string. The length of a string can change dynamically but the length of a bounded string cannot exceed the maximum length fixed at compile time.

This example shows a bounded string. The keyword `typedef` names a bounded string data type:

```
typedef string<16> name16;
```

This example shows an unbounded string named `name`:

```
typedef string name;
```

2.1.3.3.3 Wide Strings

Wide types (wchar/wstring) are not supported. By default wide types are treated as the non-wide equivalent (string/char) type. If the *`-no_map_wide`* IDL compiler flag is used then the compiler generates an error if a wide type is used.

2.1.3.3.4 Fixed

The fixed type represents fixed point decimal numbers with a specified number of significant digits and scale factor. There can be no more than 31 significant digits in a given fixed point number. The scale factor is a non-negative integer less than or equal to the number of significant digits.

This example defines a `Money` type capable of storing 9-digit figures, 2 of which are to the right of the decimal point.

```
typedef fixed<9,2> Money;
```

The fixed point type can also be used to define integer types with a specified number of significant digits by setting the scale factor to 0. This is illustrated in the following example.

```
typedef fixed<31,0> BigInteger;
```

2.1.3.4 Arrays

IDL defines multi-dimensional, fixed-size arrays. Each dimension of the array has an explicit fixed size that cannot vary at runtime.

An array is declared with a type specifier, an identifier, and the dimensions enclosed in bracket pairs.

- any of the IDL types may be made into a multi-dimensional array.
- each dimension must be a positive integer constant expression.

This example defines a one-dimensional array of bounded strings:

```
typedef string<20> FiveStrings[5];
```

This example defines a two-dimensional matrix of longs:

```
typedef long LongMatrix[4][4];
```

2.1.3.5 The any Type

The `any` type can express any legal IDL value. An `any` is self-describing and is intended to be used as an argument or value type wherever multiple different types are acceptable. An `any` can be passed as an operation argument.

An `any` contains a `TypeCode` and a value which is of the type indicated by the `TypeCode`. The `any` type can be any IDL-specified type, including another `any`. The `TypeCode` allows applications that use it to interpret the type of the data the `any` contains.

The `any` maps into an implementation-language mapped type.

i Support for `anys` must be enabled in the `idlcpp` compiler by using the `-gen_any` command-line switch and linking in the correct `any` libraries. The C `Any` type mapping is defined in the `eOrbC/CORBA/any.h` header file and must be included.

2.1.4 Constant Types

A constant provides a way to declare types that are initialized with values that cannot be changed. Constants are declared with the keyword `const`, followed by a type specifier, an identifier, the assignment operator (=), and the value to which the constant is initialized.

Table 3, *Constants*, on page 11 lists the IDL types which can be used to declare constants.

Table 3 Constants

boolean	short
char	string
double	unsigned long
fixed	unsigned long long
float	unsigned short
long	
long long	
octet	

This example declares a constant `MaxVal` of type `long` with a fixed value of `100`:

```
const long MaxVal = 100;
```

i Long long and unsigned long long constants are limited to long and unsigned long values on platforms with compilers that do not have built-in long long support.

2.1.5 Interfaces

An interface is defined with the keyword `interface` followed by an identifier that names the interface. The interface identifier may optionally be followed by an inheritance specification and the interface body enclosed in braces. The body of an interface may include IDL type definitions.

A forward declaration of an interface consists simply of the keyword `interface` followed by an identifier.

An interface forms a naming scope for identifiers. An identifier that is defined within an interface can have the same name as an identifier that is defined outside the interface. An identifier that is defined within an interface can be used outside of the interface when it is qualified with the name of the interface and the name resolution operator (**::**).

This example shows an IDL specification with an interface object that provides options to start and stop, and returns the errors of a ManagedElement.

```
enum error{NO_ERROR, ELEMENT_BUSY, ELEMENT_NOT_RESPONDING};

typedef sequence<error>  errors;

interface ManagedElement
{

   // Management operations
   boolean start();
   boolean stop();

   errors get_errors();

};
```

The example uses `enum` to define `error`, which enumerates the possible errors. The `typedef` statement defines `errors` as a `sequence` of the `enum error`. The `ManagedElement` interface contains `start()` and `stop()` operations, which return boolean type values, and a `get_errors()` operation that has a return value of type `errors`.

2.1.5.1 Operations

Operation declarations occur in an interface body. Operation declarations consist of a return type followed by an identifier that names the operation, a parameter list enclosed in parentheses, and an optional `raises` expression.

2.1.5.1.1 Return Type

An operation must specify a return type. Return types can be any IDL type. If an operation does not return a result, it must specify the `void` type.

2.1.5.1.2 Parameter Declarations

A parameter list consists of zero or more parameter declarations separated by commas. A parameter declaration consists of a directional attribute followed by an IDL data type and an identifier that names the parameter. The directional attribute specifies the direction in which the parameter is to be passed. Table 4, *Directional Attributes* lists the IDL directional attributes.

Table 4 Directional Attributes

Attribute	Meaning
in	The parameter is passed from the client to the server
out	The parameter is passed back from the server to the client
inout	The parameter is passed in both directions

The following example illustrates an operation which uses the keyword `out` to specify that the `greetstr` string is passed from object to client.

```
string greeting (out string greetstr);
```

2.1.5.1.3 Oneway Operations

An operation declaration can be preceded by the optional keyword `oneway`. A oneway operation is a non-blocking call (it does not suspend the client when it makes the request). A oneway operation is invoked once at most, and delivery of the call is not guaranteed. If the request fails after being issued, the client is not notified. The return type of a oneway operation must be `void` and the parameter list must not contain any `inout` or `out` parameters. Oneway operations can only raise standard exceptions, and will only raise those exceptions if the client ORB encounters an error prior to sending the request.

The following example declares an interface with a single oneway operation:

```
interface alarms
{
   oneway void notify (in string event);
};
```

2.1.5.1.4 Raises Expressions

A `raises` expression specifies which exceptions may be raised by an operation. the expression consists of the keyword `raises` followed by a list of exceptions enclosed in parentheses. The exception list must consist of one or more previously-defined exceptions separated by commas.

The following example declares that a single exception, `NotRunning`, may be raised by the `stop` operation.

```
boolean stop () raises (NotRunning);
```

The `raises` expression specifies any operation-specific exceptions that may be raised by a call to the operation. In addition to these exceptions, IDL defines a set of standard system exceptions which can be raised by the ORB as a result of a call to the operation. These system exceptions must *not* be listed in a `raises` expression. System exceptions may be raised by the ORB even if no operation-specific exceptions have been defined.

2.1.5.2 Attributes

In addition to operation declarations, an interface body can have attribute declarations. An attribute is declared with the keyword `attribute`, which can be preceded by the optional keyword `readonly`, followed by a type specifier and an identifier.

To enable a client to access an attribute value, the IDL compiler maps an IDL attribute declaration to two functions: one to set the value of the attribute and one to retrieve the value of the attribute. The set function takes an input parameter of the same type as the attribute; the get function returns a value of the same type as the attribute. If an attribute is defined as `readonly`, it maps only to the get function.

The following example defines an interface with a single attribute, the string element_owner.

```
interface ManagedElement
{
   attribute string element_owner;
};
```

2.1.5.3 Exceptions

An exception is a data structure that is returned to indicate that a particular type of exceptional condition occurred as a result of a request on an object. There are two types of exceptions: user exceptions and standard exceptions.

2.1.5.3.1 User Exceptions

The exception data structure is similar to a struct in syntax and form. User exceptions are declared with the keyword `exception` and an identifier followed by an optional list of members enclosed in braces. The members of an exception provide additional information you can use to determine which exceptional condition occurred or more detail about the exception that occurred.

Exceptions can not be used as parameters to operations or members of elements or other data types.

To specify what types of exceptions an operation can raise, an optional `raises` expression can follow the parameter list of an operation declaration. See *Raises Expressions* on page 14 for details.

The following example declares an exception named `no_operating_info` in the `ManagedElement` interface. The `reason` member of the `no_operating_info` exception can be used to specify the reason a status could not be returned by the `get_state()` operation. The exception may be raised by the `status` operation.

```
interface ManagedElement
{
   exception no_operating_info { string reason };

   status get_state (out operating_info current_state)
      raises  (no_operating_info);
};
```

2.1.5.3.2 System Exceptions

CORBA defines a set of system exceptions that correspond to standard runtime errors that the ORB may signal as a result of any request. These exceptions are implicitly listed in every operation's `raises` expression.

The *User Guide* includes a list of system exceptions and their minor codes as well as where the exceptions are thrown.

2.1.5.4 Inheritance

Inheritance is a mechanism for defining an interface by adding new elements to an existing interface. An existing interface which the new interface inherits from is called a base interface of the new interface. An interface can be derived from any number of base interfaces.

To specify that one interface inherits from another, follow the interface name in the interface definition with a colon (`:`) and the name of the interface it inherits from. If an interface inherits from multiple interfaces, use commas to separate their names.

A derived interface inherits all the elements (constants, types, attributes, exceptions, and operations) of the base interfaces from which it is derived. If more than one base interface uses the same name for a constant, type, or exception, qualify the name with its interface name in the derived interface. A derived interface ***cannot*** inherit from two interfaces that contain the same operation or attribute name. Additionally, a derived interface ***cannot*** redefine an inherited operation or attribute name.

The following example specifies that the interface `GroupManagedElement` inherits from the previously-defined `ManagedElement` interface.

```
interface GroupManagedElement : ManagedElement
{
   // new operations and attributes
};
```

There can be multiple levels of inheritance. For example, interface X can inherit from base interface Y which in turn inherits from base interface Z. In this situation, Z is called an *indirect base* interface of X. Y is a *`direct base`* interface of X.

2.1.6 Modules

Modules can be used to control the naming scope of identifiers. By properly scoping IDL declarations within modules, you can avoid conflicts between identifiers in different modules. An identifier that is defined within a module can have the same name as an identifier that is defined outside the module.

The following example uses two `MaxVal` identifiers, one within a module named `business` and the other outside of the module.

```
const long MaxVal = 100;

module business
{
   const long MaxVal = 50;
};
```

An identifier defined within a module can be used outside of that module when it is qualified with the name of the module and the name resolution operator (`::`). In the example above, the `MaxVal` identifier defined within the `business` module can be referred to using `business::MaxVal`.

Modules can be used to scope identifiers of IDL data types, constants, exceptions, interfaces, and other modules.

The following example uses modules to control the scope of two interface definitions with the same name:

```
module Europe
{
   interface ManagedElement
   {
      // operations
   };
};

module USA
{
```

```
    interface ManagedElement
    {
       // operations
    };
};
```

An identifier can only be defined once in a module, however it can be redefined in nested modules. An identifier can be used in an unqualified form within a particular module and will be resolved by successively searching farther out in enclosing modules.

2.1.6.1 The CORBA Module

The CORBA specification includes pre-defined names which can be used in your IDL specification. This includes interface names such as `TypeCode`. To avoid conflicts between pre-defined CORBA names and user-defined names, CORBA names are defined in a `CORBA` module. To use CORBA names in your IDL specification, qualify them with `CORBA::`, for example, `CORBA::TypeCode`.

This does ***not*** apply to IDL keywords. For example, use `Object` instead of `CORBA::Object`.

C++ Edition Specifics

CHAPTER

3 Compiling an IDL Specification

The first step in developing an application is to define the IDL specification. The IDL files are then compiled into C++ source and header files using the ORB's IDL to C++ compiler, `idlcpp`*.*

3.1 Using the IDL Compiler

Interface specifications written in IDL are stored in IDL source files. These files must have an *`.idl`* extension in order to be recognised by the IDL compiler.

The ORB's IDL to C++ compiler, *`idlcpp`*, is located in the `$EORBHOME/bin/$EORBENV` directory (on Windows use `%EORBHOME%\bin\%EORBENV%`).

3.1.1 Command Syntax

The `idlcpp` compiler is run from the command line as follows:

```
%  idlcpp [options] <idl_files>
```

where

<*`idl_files`*> is a list of one or more developer-written IDL source files
[*`options`*] is a list of zero or more command-line options.

Using `idlcpp` with no parameters or with the *`-u`* option displays usage information.

The complete list of command-line parameters is described in Table 5, *IDL to C++ Compiler Options*, on page 22. Note that command-line parameters are case sensitive (`U` and `u` perform different functions, for example).

The IDL compiler's default behaviour is to create four C++ source files for each IDL file (the number, type and names of output files can be changed with the command line options). The standard generated source files include:

- a *client header* file (containing the *stub* declarations) with a *`.h`* extension, e.g. *`myfile.h`*
- a *client implementation* file (containing the *stub* definitions) with a *`.cpp`* extension, e.g. *`myfile.cpp`*
- a *server implementation base header* file with a *`_s`* suffix and *`.h`* extension, e.g. *`myfile_s.h`*, which contains the skeleton and/or tie declarations

- a *servant implementation base* file with a `_s` suffix and `.cpp` extension, e.g. `myfile_s.cpp`, which contains the skeleton base or tie class definitions

The first two files listed above are for clients; the second two are for servers. Only the client or server files can be generated if desired: refer to the *IDL Guide* for the appropriate command line option to use.

Example

To generate the client and server stub and skeleton files from an IDL source file called `myapp.idl` use:

```
%  idlcpp myapp.idl
```

This will generate:

- myapp.h and myapp.cpp, the C++ client header and implementation files (containing the stub)
- myapp_s.h and myapp_s.cpp, the C++ server header and implemention files (containing the skeleton)

Table 5 IDL to C++ Compiler Options

Option	Description
`-chs=`<*suffix*>	Generated client header suffix (default `.h`)
`-cis=`<*suffix*>	Generated client implementation suffix (default `.cpp`)
`-client_only`	Generates only the client files.
`-collocated_direct`	Generate code for direct servant invocation.
`-D`<*name*>`[=`*value*`]`	Defines `name` and an optional `value` for use with preprocessor conditional directives in the IDL specification file. White space between the `-D` and the name is optional. This option can be repeated to specify multiple preprocessor directives.
`-E`	Only runs the pre-processor, printing the output.
`-enum32`	Can be used when enumerated types map naturally to 32-bit values. Prevents generation of additional enumeration to force size.
`-[no]exceptions`	Generate code to support native or non-native exceptions (as opposed to portable macros).

Table 5 IDL to C++ Compiler Options (Continued)

Option	Description
-gen_any	Generates code to support insertion and extraction of IDL types using anys. This flag must be used for IDL files which contain anys.
-gen_names	Generates code to support operations that return id, name, and member_name for TypeCodes and dynamic anys. Setting this flag automatically sets the gen_any flag.
-gen_tie	Generates a tie template class for each interface. By default, idlcpp does not generate tie templates.
-I<*directory*>	Adds directory to the list of directories to be searched for included IDL files.
-ignore_interfaces	Do not generate interface code.
-import_export=<*macro*>	Generates all of the code necessary to import or export the generated classes from a user-built Dynamic Link Library for Windows. See *Creating a DLL* on page 26 for more details.
-M	Generate code for included files
-no_map_wide	Do not map wchar/wstring types to char/string. Generate compiler error instead.
-max_char=<*num*>	Sets the maximum number of characters per line for generated files. If a line in the IDL file exceeds num characters, the line is split by a backslash character (\) and continued on a new line. The default maximum number of characters per line is 1024.
-no_warn	Disable warning messages.
-o *or* -output <*dir*>	Generates output into specified directory.
-shs=<*suffix*>	Generated server header suffix (default _s.h)
-sis=<*suffix*>	Generated server implementation suffix (default _s.cpp)

Table 5 IDL to C++ Compiler Options (Continued)

Option	Description
`-strip_includes`	Strips path information from `#include` directives in the generated code. For example: `#include "common/types.idl"` in an IDL file will be translated into `#include "types.h"` in the generated code.
`-u`	Displays usage message and exits. Note that this is a *lower-case* u.
`-U`<*name*>	Removes any initial definition of the preprocessor directive `name`. This option can be repeated to undefine multiple names. This is the opposite of the `–D` option. White space between the `-U` and the name is optional. Note that this is an *upper-case* U.
`-v`	Traces compilation stages. Note that this is a *lower-case* v.
`-V`	Prints compiler version information and exits. Note that this is an *upper-case* V. `-version` can also be used to display version information.
`-w`	Suppresses compiler warning messages.

3.2 IDL Compiler Output

The IDL to C++ compiler will generate the four files shown in *Figure 1* for an IDL specification file called `hello.idl`.

PrismTech

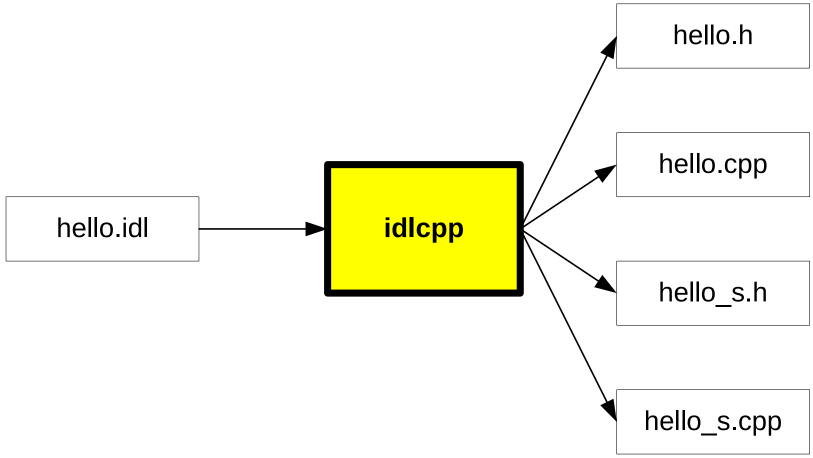

Figure 1 Example Generated C++ Files

Note that this is the default output from the IDL compiler. The number, type, and names of the output files can be changed with command-line options (given in Table 5, *IDL to C++ Compiler Options*, on page 22).

3.2.1 C++ Files Produced

- `hello.h` is the ***client header file*** containing *stub* declarations. It defines the C++ types associated with the IDL types defined in the IDL specification. The C++ types in this header file are used by client programs and object implementations. This header file provides a C++ abstract base class for every interface construct in the IDL specification. For clients, this header file provides remote access to object references through stubs that correspond to the IDL-defined `interface` operations. An ORB client program includes this header file and makes a request by calling one of the stub routines on an object reference.
- `hello.cpp` is the ***client implementation file*** containing *stub* definitions. It contains implementations of the types declared in `hello.h`. The interface implementation file is compiled and linked with an ORB client program and with object implementations.
- `hello_s.h` is the ***server implementation base header file*** containing the *skeleton* and *tie* declarations. It provides C++ types that you use in your object implementation. An ORB object implementation file includes this header file, which in turn includes the interface header file, `hello.h`. If the `-gen_tie` compiler switch is used, `hello_s.h` will also contain the tie template class definition.

- `hello_s.cpp` is the ***server implementation base file*** containing the *skeleton* and *tie* class definitions. It contains the implementations of operation dispatch functions required by object implementations. The object implementation dispatch file is compiled and linked with an ORB object implementation.

3.2.2 Using the IDL Compiler Output

The header files produced by the IDL compiler are used as follows:

- the client's modules must include the client header file (using a #include directive)
- the server's modules must include the server header file (using a #include directive)

The client and server implementation files produced by the IDL compiler should be compiled and linked with the application.

3.2.3 Creating a DLL

WIN A Win32 DLL (Dynamic Link Library) can be created by the IDL compiler by using the *-import_export=<macro>* command-line option.

. The following example illustrates this with the *MY_DLL* macro:

```
%  idlcpp -import_export=MY_DLL hello.idl
```

This inserts an import/export block into *hello.h* as follows:

```
#if defined(_WIN32)
#   if defined(EORB_BUILD_MY_DLL)
#      define MY_DLL __declspec(dllexport)
#   elif defined(EORB_NO_MY_DLL)
#      define MY_DLL
#   else
#      define MY_DLL __declspec(dllimport)
#   endif
#else
#   define MY_DLL
#endif
```

When compiling the generated source into a DLL, `-DEORB_BUILD_MY_DLL` must be used. This ensures that the appropriate symbols are declared as exported from the resulting DLL (by declaring `__declspec(dllexport)`).

When compiling the generated source into a static library, and when compiling other code that is to be linked with the resulting static library, `-DEORB_NO_MY_DLL` must be used to ensure that there is no `__declspec(dllimport)` or `__declspec(dllexport)` declaration.

If neither `-DEORB_BUILD_MY_DLL` nor `-DEORB_NO_MY_DLL` are used then the appropriate symbols will be declared as `__declspec(dllimport)`, which is effectively “standard” behaviour for a C++ file.

CHAPTER

4 IDL to C++ Mapping

This section describes the IDL to the C++ language mapping for Spectra ORB C++ Edition. Applications, clients or servers which conform to the constructs described here will be portable across conforming ORB implementations.

i The ORB's IDL to C++ compiler conforms to the OMG's *C++ Language Mapping Specification Version 1.0*.

4.1 Namespaces

Names in IDL can be scoped within a module or an interface in order to avoid identifier conflicts. This mapping ensures that you can use each of the types defined in the IDL specification without ambiguity.

4.1.1 Scoped Names and Namespaces

Names which are scoped in IDL are mapped to C++ scoped names as follows:

- modules can be mapped to C++ namespaces
- interfaces are mapped to C++ classes
- All IDL constructs which are scoped to an interface or module are accessed via C++ scoped names. For example, if a type `Mode` were defined in interface `Printer`, the type would be referred to as `Printer::Mode`.

These mappings allow corresponding mechanisms to be used in both IDL and C++ to build scoped names. For example, the IDL is logically mapped into C++ as:

```
// IDL
module M
{
  struct S
  {
    // definitions
  };

  union U
{
    // definitions
  };
};
```

```
// C++
namespace M
```

```
{
  struct S
  {
    // definitions
  };

  union U
  {
    // definitions
  };
};
```

The following are legal scoped names in C++:

```
// C++

M::S s;
M::U u;
```

Alternatively, employing the C++ *using* statement allows you to specify the name without explicitly using scoping.

```
// C++
using namespace M;
S s;
U u;
```

4.2 Modules

A module defines a scope and maps to a C++ namespace with the same name:

```
// IDL
module M
{
   // definitions
};
```

```
// C++
namespace M
{
   // definitions
}
```

Example Mapping to namespace

The IDL *module M* mapping to *namespace* is shown below.

```
// IDL
module M
{
  // definitions
};
```

4.3 Interfaces

IDL interfaces are mapped to abstract C++ classes using the same name and scope as the interface. The interface class includes methods for each of the operations and attributes that the interface supports. The interface class contains public C++ definitions for each of the IDL types defined in the scope of the interface in IDL.

Each interface class is derived from `CORBA::Object`, which defines behaviours common to all interface classes.

i Note that the ORB typedefs `CORBA::Object` to `CORBA::Stub`.

Example Interface mapping to Class

The example IDL *`interface A`* maps to the C++ *`class A`* as shown below:

```
interface A
{
   const float pi = 3.14159;
   struct S
   {
     short field;
   };
   void op();
};
```

```
class A
:
  virtual public CORBA::Object //e*ORB uses CORBA::Stub
{
public:
  // Nested definitions
  static const CORBA::Float pi;

  struct S {
    CORBA::Short field;
  };
  // operations
  virtual void op() = 0;
  // Object reference operations, explained later

protected:
  // restricted behaviours
  A();
  virtual ~A();

private:
  A(const A&);
  void operator = (const A&);
};
```

A CORBA-compliant C++ program cannot

- create or hold an instance of an interface class
- use a pointer (`A*`) or reference (`A&`) to an interface class

- reference ORB-specific interface classes (classes other than `A`, `A_ptr`, and `A_var`) that an ORB implementation might generate to implement interface behaviours in C++

Example Non-conformant uses

The following uses of *`class A`* are not conformant:

```
// C++: Non-conformant uses
A a;                  // cannot declare an instance of an
                      // interface class...

A * p;                // nor declare a pointer to an interface
                      // class...

void f(A & r);        // nor declare a reference to an
                      // interface class
```

A CORBA-compliant program may refer to nested types and constants using their fully-qualified scoped names as follows:

```
// C++
  A::S s;         // declare a struct variable
  s.field = 3;  // field access
```

4.3.1 The _var Type

The `_var` type for an interface both manages and behaves like an object reference. For the *`interface A`*, `A_var` behaves like the `A_ptr` object reference except that the `A_var` releases the real object reference when it goes out of scope or is assigned a new object reference.

Example _var type for an interface

The following code demonstrates using the interface `_var` type to invoke an operation on a managed object reference:

```
A_ptr ap = ...      // obtain a reference to an A
A_var av = ap     // av assumes ownership of ap.
av->op();         // Invoke op on av's object reference.
```

The `A_var` and `A_ptr` types may be assigned to each other without any explicit operations or casts. However, keep in mind that the `var` deallocates its pointer when it goes out of scope. Be careful to avoid using an object reference after it has been deallocated by a `var`. For example, the assignment statement in the code below results in the object reference held by `p` being released at the end of the block containing the declaration of `v`. Attempting to use the object reference after the end of the block causes errors because the `var` releases the object reference.

```
// C++

A_ptr p = // ...somehow obtain an A

{
```

```
  A_var v;
  v = p;
}  // v will release the object reference when it goes out
   // of scope

p->op();    // this call will fail since the object reference
            // has been released by v.
```

4.3.2 Object References

An IDL object reference maps to two C++ types. For an *`interface A`*, these types are named *`A_ptr`* and *`A_var`* where

- the name for the object reference is formed by appending *`_ptr`* or *`_var`* to the name of the interface
- the `_ptr` type behaves like a raw C++ pointer type
- the `_var` type behaves like the `_ptr` object reference except that the `_var` type releases the real object reference when it goes out of scope or is assigned a new object reference

i The ORB defines `A_ptr` for an interface `A` as a C++ pointer:

```
class A;
typedef A * A_ptr;
```

Operations on an object can be performed with an object reference by using the arrow operator (->) on the object reference (`_ptr`).

To invoke `op` on `ap`, which is a reference to an `A`:

```
A_ptr ap = ... // obtain a reference to an A
ap->op();      // invoke op on ap
```

An object reference is used the same way when it refers to a local object instance as when it refers to a remote object instance. Using the object reference type enables the caller of an object's services to make requests on an object without regard to the object's implementation or location, whether the object is local to the caller's process address space or an object instance on a remote host.

4.3.3 Object Reference Operations

The C++ mapping provides a number of standard functions for manipulating and comparing object references. These functions are either static functions in the CORBA namespace or are member functions of `CORBA::Object`.

4.3.3.1 Common Operations

CORBA defines three operations on any object reference: *duplicate*, *release*, and *is_nil*. Note that these are operations on the object reference, not the object implementation. Because the mapping does not require that object references to themselves be C++ objects, the `->` syntax cannot be employed to express the usage of these operations. Also, for convenience these operations are allowed to be performed on a nil object reference.

4.3.3.1.1 _duplicate and _nil

`_duplicate` and `_nil` are static member functions generated for each interface and are especially useful for initializing object references.

These functions are generated as follows:

```
// IDL
interface A {};

// C++
class A
{
  ...
  static A_ptr _duplicate(A_ptr obj);
  static A_ptr _nil();
  ...

};
```

The `_duplicate` function returns a new object reference with the same static type as the given reference and does not consume the given object reference. The caller is responsible for releasing the returned object reference.

A `nil` object reference is a special value that refers to no object. This value might be used to initialize a return value or as a return from an unsuccessful search request. The `_nil` function returns the `nil` object reference value for a given type. Whether to release the returned nil pointer is a stylistic decision that has no impact on code integrity. Conforming applications may not attempt to invoke methods on a `nil` object reference.

Example duplicate and nil usage

```
// IDL
interface A {};
// C++

A_ptr ap1 = ...                   // get an object reference
A_ptr ap2 = A::_duplicate(ap1);   // assign a copy of ap1 to ap2

A_ptr ap3 = A::_nil();            // assign a nil object
                                  // reference to np3
```

4.3.3.1.2 is_nil and release

`is_nil` and *`release`* are both static functions defined in the CORBA namespace as follows:

```
  // C++
  namespace CORBA
  {
```

```
    void release(Object_ptr obj);
    Boolean is_nil(Object_ptr obj);
  };
```

The `release` operation indicates that the caller will no longer access the reference, so associated resources may be deallocated. If the given object reference is `nil`, then `release` does nothing.

The `is_nil` operation returns `TRUE` if the object reference contains the special value for a nil object reference as defined by the ORB. Neither the release operation nor the `is_nil` operation may throw CORBA exceptions.

A compliant application need not call release on the object reference returned from the _nil function. References may not be compared using *`operator`*==; therefore, `is_nil` is the only compliant way an object reference can be checked to see if it is nil.

The `_nil` function may not throw any CORBA exceptions. A compliant program cannot attempt to invoke an operation through a nil object reference, since a valid C++ implementation of a nil object reference is a null pointer.

For each interface *`A`* the following is always guaranteed to return `TRUE`:

```
// IDL
interface A {};

// C++
Boolean true_result = CORBA::is_nil(A::_nil());
```

4.3.3.2 Widening Object References

An object reference for a derived interface can be widened to an object reference of a statically known parent type without any explicit operations or casts. For example, if interface *`B`* is derived from interface *`A`*, then the following implicit widening operations for `B` are supported:

- `B_ptr` to `A_ptr`
- `B_ptr` to `CORBA::Object_ptr`
- `B_var` to `A_ptr`
- `B_var` to `CORBA::Object_ptr`

When mixing `_var` and `_ptr` types, be careful to avoid the direct use of an object reference that has been assigned to a `_var`. Once the `_var` leaves scope, the object reference is no longer valid.

An attempt to implicitly widen from one `_var` type to another will cause a compile-time error. Assignment between two `_var` objects of the same type is supported, but widening assignments are not and will cause a compile-time error.

Widening assignments may be done using `_duplicate`. The same rules apply for object reference types that are nested in a complex type, such as a structure or sequence. The following code demonstrates these rules:

```
// C++
B_ptr bp = ...                    // acquire a reference to B
A_ptr ap = bp;                    // implicit widening
CORBA::Object_ptr objp = bp;      // implicit widening
objp = ap;                        // implicit widening
B_var bv = bp;                    // bv assumes ownership of bp
ap = bv;                     // implicit widening, bv retains ownership
obp = bv;                         // implicit widening,
                                  // bv retains ownership
A_var av = bv;                    // illegal, no implicit conversions
A_var av = B::_duplicate(bv);     // av and bv both refer to bp
B_var bv2 = bv;                   // implicit _duplicate
A_var av2;
av2 = av;                         // implicit _duplicate
```

4.3.3.3 Narrowing Object References

The mapping for an interface defines the *`_narrow`* static member function which returns a new object reference when passed an existing reference. Like `_duplicate`, the `_narrow` function returns a `nil` object reference if the given reference is nil. Unlike `_duplicate`, the parameter to `_narrow` is a reference of an object of any interface type. If the actual runtime type is of the requested interface's type, then `_narrow` returns a valid object reference. Otherwise, `_narrow` returns a `nil` object reference. For example, given the IDL shown below if the actual run-time type of the object referenced by a `CORBA::Object_ptr obj` is C, then:

```
// IDL
interface A {};
interface B : A {};
interface C : B {};
interface D : C {};
```

- `A::_narrow(obj)` returns a valid object reference
- `B::_narrow(obj)` returns a valid object reference
- `C::_narrow(obj)` returns a valid object reference
- `D::_narrow(obj)` returns a nil object reference

Narrowing to `A`, `B`, and `C` all succeed because the object referenced by `obj` supports all those interfaces. `D::_narrow(obj)` fails because the `C` object does not support the `D` interface.

i The client may need to make a distributed call in order to determine the actual runtime type of the object.

If successful, the `_narrow` function creates a new object reference and does not consume the given object reference, so the caller is responsible for releasing both the original and new references.

The `_narrow` operation can throw `CORBA` system exceptions.

4.4 Data Type Mappings

4.4.1 Constants

IDL constants map directly to a C++ constant definition that may or may not define storage, depending on the scope of the declaration. In the example below, a top-level IDL constant maps to a file-scope C++ constant, but a nested constant maps to a class-scope C++ constant. This inconsistency occurs because C++ file-scope constants may or may not require storage, or the storage may be replicated in each compilation unit, while class-scope constants always take storage. As a side effect, this difference means that the generated C++ header file might not contain values for constants defined in the IDL file.

Example

```
// IDL
const string some_string = "some string";
interface A
{
  const float pi = 3.14159;
};

// C++
static const CORBA::String some_string = "some string";

class A
{
public:
  static const CORBA::Float pi;
};
```

The generated code must use the constant's value instead of the constant's name in certain situations. For example:

```
// IDL
interface A
{
  const long n = 10;
  typedef long V[n];
};

// C++
class A
{
public:
  static const CORBA::Long n;
  typedef CORBA::Long V[10];
};
```

When generating constant values for `long long` types, compilers that do not support native `long long` types are limited to 32-bit constant values.

4.4.2 Typedefs

A `typedef` creates an alias for a type. If the original type maps to several types in C++, the code generator creates the corresponding alias for each type. The following example illustrates the mapping:

```
  // IDL
  typedef long long_t;
  interface i;
  typedef i i_t;

  typedef sequence<long> long_sequence;
  typedef long_sequence long_sequence_t;

// C++
  typedef CORBA::Long long_t;
  // Definitions for interface i, i_ptr, i_var
  typedef i i_t;
  typedef i_ptr i_t_ptr;
  typedef i_var i_t_var;
  // Definition for long_sequence, long_sequence_var
  typedef long_sequence long_sequence_t;
  typedef long_sequence_var long_sequence_t_var;
```

4.4.3 Basic Data Types

The following table shows mappings and sizes for the basic data types as defined in the `eOrb/CORBA.h` header file.

Table 6 Basic Data Types Mapping and Sizes

IDL	C++	Size
`short`	`CORBA::Short`	16 bits signed
`long`	`CORBA::Long`	32 bits
long long	`CORBA::LongLong`	64 bits
`unsigned short`	`CORBA::UShort`	16 bits unsigned
`unsigned long`	`CORBA::ULong`	32 bits
`unsigned long long`	`CORBA::ULongLong`	64 bits
`float`	`CORBA::Float`	32 bits
`double`	`CORBA::Double`	64 bits
`char`	`CORBA::Char`	8 bits signed
`boolean`	`CORBA::Boolean`	8 bits
`octet`	`CORBA::Octet`	8 bits unsigned

Because some C++ types, such as `short` and `long`, may have different representations on different platforms, each IDL basic type is mapped to a `typedef` in the CORBA namespace. The CORBA definitions reflect the appropriate representation. For example, on a 64-bit machine where a long integer is 64 bits, the definition of `CORBA::Long` still refers to a 32-bit integer. Accordingly, use the CORBA types to ensure portability.

The mappings for the basic types are distinguishable from each other for the purposes of overloading except for `boolean`, `char`, and `octet`: you can safely write overloaded C++ functions on `Short`, `UShort`, `Long`, `ULong`, `Float`, and `Double`.

For the `boolean` type, the values 1 (one, representing `TRUE`), 0 (zero, representing `FALSE`) are defined; other values produce unpredictable behaviour.

4.4.4 Enums

An IDL `enum` maps directly to the corresponding C++ type definition. The only difference is that the mapping adds an additional constant to force the C++ compiler to use exactly 32-bits for enumerated type declared values.

```
// IDL
enum Color { red, green, blue };

// C++
enum Color { red, green, blue, EORB_FORCE_ENUM32(__color) };
```

4.4.5 String Types

The IDL `string` type, whether bounded or unbounded, is mapped to `char*` in C++. String data is null-terminated.

For dynamic allocation of strings, compliant programs must use the following functions from the CORBA namespace.

```
// C++
namespace CORBA   // declared in eOrb/corba.h
{
   char* string_alloc(ULong len);
   char* string_dup(const char*);
   void string_free(char*);
}
```

The `string_alloc` function dynamically allocates a string or returns a `null` pointer if it cannot perform the allocation. It allocates `len+1` characters so the resulting string has enough space to hold a trailing `NUL` character. The `string_dup` function dynamically allocates enough space to hold a copy of its string argument, including the `NUL` character, copies its string argument into that memory, and returns a pointer to the new string. If allocation fails, it returns a `null` pointer.

The `string_free` function deallocates a string that was allocated with `string_alloc` or `string_dup`. Passing a `null` pointer to `string_free` is acceptable and results in no action being performed.

The `string_alloc`, `string_dup`, and `string_free` functions do not throw CORBA exceptions.

4.4.5.1 String_var

`String_var` is a convenience class that manages a `char` pointer (`char*`). An instance of `String_var` automatically frees its pointer when the `String_var` instance is deallocated. When a `String_var` is constructed or assigned from a `char*`, the `String_var` assumes ownership of the memory referenced by the `char*`. Assignment or construction from a `const char*` or from another `String_var` causes a copy. Additionally, assignment causes the `String_var` to free its current pointer.

On some compliers, a C++ string literal has type `char*`, so never construct or assign a string literal to a `String_var` unless the string literal is first cast to `const char*`.

Example

```
// C++
CORBA::String_var svar1("This will probably crash!");
CORBA::String_var svar2((const char *)"This is OK.");

CORBA::String_var svar3 = "This will probably crash!";
CORBA::String_var svar4 = (const char *)"This is OK.";
```

`svar1` and `svar3` will probably crash because `svar1` and `svar3` will attempt to free the memory, which has been allocated for their respective string literals, when they go out of scope. Casting the string literal to `const char*` causes the `String_var` to duplicate the data, which it can then free successfully.

`String_var` uses `CORBA::string_free` to release its pointer. Using a string that was not allocated using `CORBA::string_alloc` or `CORBA::string_dup` can result in unpredictable behaviour. This includes strings allocated with `malloc` or copied with `strdup`.

Example

```
// C++
String_var svar1 = (const char *)"This is a string";
String_var svar2 = CORBA::string_dup("This is a string");

int len = strlen(svar1)+1;
String_var svar3 = CORBA::string_alloc(len);

for(int i = 0; i < len; i++)
{
  svar3[i] = svar1[i];
}
svar2 = svar3;  // old svar2 storage is released; svar3 is copied to
svar2
// no deletes needed since each String_var releases its own storage
```

4.4.6 Structured Types

The mapping for `struct`, `union`, and `sequence` (but not `array`) is a C++ struct or class with a default constructor, a copy constructor, an assignment operator, and a destructor.

The default constructor initializes object reference members to appropriately-typed `nil` object references and string members to the empty string (""); all other members are initialized via their default constructors.

The copy constructor performs a deep-copy from the existing structure to create a new structure, including calling `_duplicate` on all object reference members and performing the necessary heap allocations for all string members.

The assignment operator first releases all object reference members and frees all string members, then performs a deep-copy to create a new structure.

The destructor releases all object reference members and frees all string members.

4.4.6.1 Fixed versus Variable-Length

The mapping for IDL structured types (structs, unions, arrays, and sequences) can vary slightly depending on whether the data structure is fixed-length or variable-length. A type is variable-length if it is one of the following types:

- The `any` type
- a `bounded` or `unbounded string`
- a `bounded` or `unbounded sequence`
- an object reference or reference to a transmissible pseudo-object
- a `struct` or `union` containing a member with a variable-length type
- an `array` with a variable-length element type
- a `typedef` to a variable-length type

The reason for treating fixed-and variable-length data structures differently is to allow more flexibility in the allocation of `out` parameters and return values from an operation. This flexibility allows a client-side stub for an operation that returns a sequence of strings, for example, to allocate all the string storage in one area that is deallocated in a single call.

4.4.7 T_var Types

A `T_var` convenience class is generated for each IDL structured type `T` in order to provide consistent argument-passing usage for both fixed- and variable-length structured types. This allows applications to be coded in terms of `T_var` types regardless of whether the underlying types are fixed- or variable-length. Each `T_var` type is defined at the same level of nesting as its `T` type.

The `T_var` is similar to the `String_var,` except that it maintains a pointer to a `T*` rather than a string. The general form of the `T_var` types is shown here:

```
// C++

  class T_var
  {
  public:
    T_var();
      // Constructs a T_var with a null T *.

    T_var(T * p);
      // Constructs a T_var with ownership of p.

    T_var(const T_var& that);
      // Constructs a T_var with a copy of that.

    ~T_var();
      // Frees the memory owned by an instance of T_var.

    T_var& operator=(T * p);
      // The T_var instance will assume ownership of p
      // after first freeing
      // the previous memory owned by the instance.

    T_var& operator=(const T_var & that);
      // The T_var instance will assume ownership of a copy
      // of that after first freeing the previous memory
      // owned by the instance.

    T& operator=(const T& that) { ptr_= new T(that); return *ptr_; }
      // The T_var instance will assume ownership of a copy
      // of that after first freeing the previous memory
      // owned by the instance.

    T* operator->();
      // Returns a pointer to the instance of T owned by T_var.
      // Note that this pointer is guaranteed to be null
      // if T_var was constructed with default constructor
      // and never assigned.

    operator T *();
```


PrismTech

```
      // Returns a pointer to the instance of T owned by T_var.
      // Note that this pointer is guaranteed to be null
      // if T_var was constructed with default constructor
      // and never assigned.

    operator T *() const;
      // Returns a const pointer to the instance of T owned by
      // T_var.  Note that this pointer is guaranteed to be
      // null if T_var was constructed with default
      // constructor and never assigned.

    operator T &();
      // Returns a reference to the instance of T owned by
      // T_var.  Note that this call will result in undefined
      // behavior if T_var was constructed with the default
      // constructor and never assigned.

    operator const T &() const
      // Returns a reference to the instance of T owned by
      // T_var.  Note that this call will result in
      // undefined behavior if T_var was constructed
      // with the default constructor and never assigned.

    const T& in()const;
      // Returns a reference to the instance of T owned by T_var.
    T& inout();
       // Returns a reference to the instance of T owned by T_var.
    T*& out()
       // Returns a pointer_reference, which allows the T_var to
receive ta new instance.
       // The previous reference is released.
    T* _retn();
       // Returns a pointer to the instance of T owned by T_var
       // and T_var will yield its ownership.
     T*& val();
       // Proprietary function for out.

 };
```

Example

```
// IDL
module M
{
  struct S
  {
    long L;
  };
};
```

```
// C++
M::S* s1 = CORBA_new M::S;

M::S_var s1_var = s1;      // s1_var assumes ownership of s1
M::S_var s2_var = s1_var;  // s2_var makes a copy of
                           // s1_var's data

s1_var->L = 99;
s2_var->L = 0;             // s1_var->L still equals 99
```

4.4.8 Struct Types

An IDL `struct` maps to a C++ `struct`, with each IDL `struct` member mapped to a self-managing member of the C++ `struct`. Having no constructors and public members allows simple field access as well as aggregate initialization of most fixed-length `structs`. Because all members map to self-managing types, all generated C++ `structs` rely upon the default constructors, copy constructors, destructors, and assignment operators.

The default constructor effectively initializes all strings and object references to `nil` (see below). The copy constructor and assignment operator perform deep copies, allocating additional storage as necessary. Additionally, the assignment operator releases any heap allocated storage before copying. The destructor releases all heap allocated storage.

With the exception of strings and object references, the type of C++ `struct` member is the normal mapping of the IDL member's type.

The assignment of a `const` pointer to a string member results in a release of its old storage and a copy, while assignment of a non-`const` pointer causes the member to release old storage and assume ownership of the pointer. A struct's default constructor relies upon the string type's default constructor to initialize itself to the empty string ("").

The assignment of a `const` pointer to an object reference member results in a release of its old storage and a copy, while assignment of a non-`const` pointer causes the member to release old storage and assume ownership of the pointer. A struct's default constructor relies upon the object reference type's default constructor to initialize itself to a nil object reference.

PrismTech

The generated `T_var` class for a `struct` has the same interface as shown in *T_var Types* on page 42.

4.4.8.1 Examples

The examples provided below use the IDL shown here:

```
// IDL
struct Fixed_struct
{
  long m_L;
  float m_F;
};

struct Variable_struct
{
  Fixed_struct m_fixed;
  Object m_objref;
  string m_name;
};

//Fixed_struct and Variable_struct map to:
// C++
struct Fixed_struct
{
  CORBA::Long m_L;
  CORBA::Float m_F;
};

struct Variable_struct
{
  Fixed_struct m_fixed;
  CORBA_Object_var m_objref;
  CORBA::String_mgr m_name;
};
```

`CORBA::String_mgr` is a proprietary class used by the ORB to implement the semantics of the C++ wrapping. Compliant code should not use this class.

This C++ code shows how to initialize and construct both a fixed- and variable-length `struct`.

The example uses the `CORBA::ORB::resolve_initial_references` function to obtain an object reference. This function offers a convenient way to obtain object references from the environment.

```
  // C++
  int main(int argc, char ** argv)
  {
    const char * server_name1 = "ExampleServer1";
    const char * server_name2 = "ExampleServer2";
    CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);

    Fixed_struct fs1 = { 99, 1.5 };

      // vs1.m_objref is initialized to a nil object reference
      // vs1.m_name is initialized to an empty string, ""
    Variable_struct vs1;
```

```
    vs1.m_fixed = fs1;

      // CORBA::ORB::resolve_initial_references() returns an
      // object reference allocated from the heap.
      // vs1.m_objref assumes ownership of that reference and
      // releases it when vs1 is deallocated.
    vs1.m_objref = orb->resolve_initial_references(
                                              server_name1);

    vs1.m_name = server_name1;  // server_name1 is const, so it
                               // will be copied
      // vs2 is a _var type.  It will use its copy
      // constructor to be initialized with its own
      // copies of vs1's data.

    Variable_struct_var vs2(vs1);

      // releases old storage, copies server_name2

    vs2->m_name = server_name2;
      // vs2->m_objref is released and assumes ownership of
      // the new object reference return by
      // CORBA::ORB::resolve_initial_references()
    vs2->m_objref = orb->resolve_initial_references(
                                              server_name2);

    CORBA::release(orb);  // be sure to release the orb's
                          // object reference

    return 0;
  };
```

4.4.9 Union Types

Unions map to C++ classes with access functions for the union members. The access functions are given the same name as the member names, and allow the members to be both set and retrieved. The C++ `union` class additionally provides a discriminant member, which provides information regarding the currently-held type of the `union`.

For example, the following IDL maps to the C++ shown below:

```
// IDL
typedef octet Bytes[64];  // Defines an array of octet.

struct S
{
  long len;
};

union U switch (long)
{
  case 1: long x;
  case 2: Bytes y;
  case 3: string z;
  case 4:
  case 5: S w;
  default: Object obj;
```

```
};
```

```
// C++
class U
{
 public:
  // Constructors, destructor and assignment
  U();
  U(const U& that);
  ~U();
  U & operator=(const U& that);

  // discriminant accessors
  void _d(CORBA::Long val);
  CORBA::Long _d() const;

  // CORBA::Long member accessors
  void x(CORBA::Long __val);
  CORBA::Long x() const;

  // Bytes member accessors
  void y(Bytes_slice* __val);
  Bytes_slice* y() const;

  // CORBA::String member accessors
  void z(char* __val);
  void z(const char* __val);
  void z(const CORBA::String_var& __val)
  const char* z() const

  // S member accessors
  void w(const S& __val);
  S& w();
  const S& w() const;

  // CORBA::Object member accessors
  void obj(CORBA_Object_ptr __val);
  CORBA_Object_ptr obj() const;
};
```

The default `union` constructor does not initialize the discriminant, nor does it initialize any `union` members to a useful state. It is therefore an error for an application to use a `union` before initializing it.

The copy constructor copies the discriminant value and makes a deep copy of the argument's data. The assignment operator first releases any storage, if necessary, copies the discriminant value, and finally, makes a deep copy of the argument's data.

The accessor functions of the discriminant have the name *_d* and allow the discriminant to be both set and retrieved. The type of the discriminant is given by the union's IDL switch type. The discriminant value always reflects either the last set value or a legal discriminant value for the last member that was set.

Setting the discriminant value to a non-legal discriminant value for a union's current type results in unpredictable behaviour. You will rarely set discriminant values.

Setting any member of the `union` through one of its accessors causes the `union` to set its discriminant and release any old storage, if necessary.

`Set` member functions for basic types accept arguments by value; the union performs a deep copy of the set argument.

`Set` member functions for constant string arguments perform copies. The `union` assumes ownership of any non-constant string argument passed to it.

Any attempt to get a member that is not consistent with the current discriminant value results in unpredictable behaviour. All returned member data, except for basic types, is owned by the `union` and must not be released.

Basic types are returned by value.

Array members are returned by slice pointers, where a slice is a subset of an array with all the dimensions of the original specified except the first one.

All other `get` member functions, including those for `structs`, `unions`, and `anys`, are returned by reference.

Example

This example shows the C++ `union` class interface shown above:

```
// C++
  S s = { 10 };
  U u;
  u.w(s)                // discriminant = 4
  u._d(4);              // OK.
  u._d(5);              // OK.
  //u._d(1);          // BAD.  This results in unpredictable behaviour
                        // since the current type is not long.
  CORBA::Object_ptr objref = ...     // Get an object reference.
  u.obj(objref)                      // u.obj takes a duplicate of
objref.
  u.obj()->_is_a("ExampleServer");   // using operator->() on an
                                     // Object_ptr

  u.z((const char *)"test string");  // u's object is released,
                                     // u.z gets copy
  u.x(10);                 // u's string is released

  U_var u_var = new U;     // Use the generated U_var.

  u_var = u;               // Copies u into u_var.
  u_var->_d();             // Returns 1.
  objref = ...             // Get another object reference.
  u_var->obj(objref);      // u_var.obj takes a duplicate of obhref.
  u_var = new U;           // old U (and objref) is
                           // released, u_var now holds the new U.
```

The generated `T_var` class for a `union` has the same interface as shown in *T_var Types* on page 42.

4.4.10 Sequences

A sequence is mapped to a C++ class that behaves like an array with a current length and a maximum length. For a bounded sequence, the maximum length is implicit in the sequence's type and cannot be explicitly controlled by the programmer.

Deallocation of a sequence's storage is controlled by a boolean release flag. When `TRUE`, this flag indicates that it is the sequence's responsibility to deallocate its storage. When `FALSE`, the sequence is not responsible for its storage. The operations and constructors that set or modify this flag are noted below.

Unbounded and bounded sequences are described in detail below, followed by a description of the *`sequence_var`* type.

4.4.10.1 Unbounded Sequences

Unbounded sequences allow maximum flexibility over sequence size and allocation. For an unbounded sequence, the initial value of the maximum length can be specified in the sequence constructor to allow control over the size of the initial buffer allocation. The programmer may always explicitly modify the current length of any sequence.

You can set the current length to a value greater than the actual current length, which may cause reallocation. Reallocation is conceptually equivalent to creating a new sequence of the new length, copying the old sequence elements zero through length minus one into the new sequence, and then assigning the old sequence to be the same as the new sequence. If the release flag is `TRUE` when reallocation occurs, the old sequence is released. The release flag is always set to `TRUE` after reallocation.

Setting the length to a smaller value than the actual current length does not affect how the storage associated with the sequence is allocated. However, the elements orphaned by this reduction are no longer accessible and their values cannot be recovered by increasing the sequence length to its original value. You cannot depend on the orphaned values being released immediately after you shorten the sequence.

The default constructor for unbounded sequences sets the sequence length equal to 0 and also sets the maximum length to 0.

Unbounded sequences provide a constructor that allows only the initial value of the maximum length to be set (the *maximum constructor* shown in the above example). This allows applications to control how much buffer space is initially allocated by the sequence. This constructor also sets the length to 0 and the release flag to `TRUE`.

The `T *data` constructor allows the length and contents of a unbounded sequence to be set. It also allows the initial value of the maximum length to be set for unbounded sequences. For this constructor, ownership of the buffer is determined by the release parameter: `FALSE`, means the caller owns the storage for the buffer and its elements; `TRUE` means that the sequence assumes ownership of the storage for the buffer and its elements. If release is `TRUE`, the buffer is assumed to have been allocated using the sequence `allocbuf` function, and the sequence will pass it to the sequence `freebuf` function when finished with it.

The copy constructor creates a new sequence with the same maximum and length as the given sequence, copies each of its current elements (items zero through length–1), and sets the release flag to `TRUE`.

The assignment operator deep-copies its parameter, releasing old storage if necessary. It behaves as if the original sequence is destroyed via its destructor and then the source sequence copied using the copy constructor.

If `release=TRUE`, then the destructor destroys each of the current elements (items zero through length–1), and destroys the underlying sequence buffer.

For an unbounded sequence, if a reallocation is necessary due to a change in the length and the sequence was created using the `release=TRUE` parameter in its constructor, the sequence will deallocate the old storage for all elements and the buffer. If release is `FALSE` under these circumstances, old storage will not be freed for either the elements or for the buffer before the reallocation is performed. After reallocation, the release flag is always set to `TRUE`.

The *`maximum()`* accessor function returns the total number of sequence elements that can be stored in the current sequence buffer for an unbounded sequence. This allows applications to know how many items they can insert into an unbounded sequence without causing a reallocation to occur.

The *`length()`* functions can be used to access and modify the length of the sequence. Increasing the length of a sequence adds new elements at the tail. The newly-added elements behave as if they are default-constructed when the sequence length is increased. However, a sequence implementation may delay actual default construction until a newly-added element is first accessed. For sequences of strings and wide strings, default element construction requires initialization of each element to the empty string or wide string. For sequences of object references, default element construction requires initialization of each element to a suitably-typed nil object reference.

The elements of sequences of complex types, such as structs and sequences, are initialized by their default constructors. Union sequences elements do not have any application-visible initialization; in particular, a default constructed union element is not safe for marshalling or access. Sequence elements of a basic type, such as `ULong`, have undefined default values.

The overloaded subscript operators (`operator[]`) return the item at the given index. The non-const version allows assignment into the item at the given index, while the const version allows read-only access to the item at the given index.

The overloaded subscript operators can not be used to access or modify any element beyond the current sequence length. Before either form of operator[] is used on a sequence, the length of the sequence must first be set using the length(ULong) modifier function, unless the sequence was constructed using the T *data constructor.

For strings and object references, operator[] for a sequence returns a type with the same semantics as the types used for string and object reference members of structs and arrays. Assignment to the string or object reference sequence member via operator=() releases old storage when appropriate. These return types honour the setting of the release parameter in the T *data constructor with respect to releasing old storage. The overloaded operator<< (insertion) and operator>> (extraction) operators for using string sequence elements and wide string sequence elements directly with C++ iostreams are provided.

The release() accessor function returns the state of the sequence release flag.

The overloaded get_buffer() accessor and reference functions allow direct access to the buffer underlying a sequence. This can be very useful when sending large blocks of data as sequences, such as sending image data as a sequence of octet, and the per-element access provided by the overloaded subscript operators is not sufficient.

The non-const get_buffer() reference function allows read-write access to the underlying buffer. If its orphan argument is FALSE (the default), the sequence returns a pointer to its buffer, allocating one if it has not yet done so. The size of the buffer can be determined using the maximum() accessor. The number of elements in the buffer can be determined from the sequence length() accessor. The sequence maintains ownership of the underlying buffer. Elements in the returned buffer may be directly replaced by the caller. For sequences of strings and object references, the caller must use the sequence release() accessor to determine whether elements should be freed (using CORBA::string_free or CORBA::release for string and object references, respectively) before being directly assigned to. Because the sequence maintains a notion of the length and size of the buffer, the caller of get_buffer() shall not lengthen or shorten the sequence by directly adding elements to the buffer or directly removing elements from the buffer. Changing the length of the sequence shall be performed only by invoking the sequence length() modifier function.

Alternatively, if the orphan argument to get_buffer() is TRUE, the sequence yields ownership of the buffer to the caller. If orphan is TRUE and the sequence does not own its buffer (i.e., its release flag is FALSE), the return value is a null pointer. If the buffer is taken from the sequence using this form of get_buffer(), the sequence reverts to the same state it would have if constructed using its default constructor. The caller becomes responsible for eventually freeing each element of the returned buffer (for strings and object references), and then freeing the returned buffer itself using freebuf.

The const `get_buffer()` accessor function allows read-only access to the sequence buffer. The sequence returns its buffer, allocating one if one has not yet been allocated. No direct modification of the returned buffer by the caller is permitted.

For the non-`const` `get_buffer()` reference function with an orphan argument of `FALSE`, and for the `const get_buffer()` accessor function, the return value remains valid until another non-const member function of the sequence is invoked, or until the sequence is destroyed, whichever occurs first.

The `replace()` function allows the buffer underlying a sequence to be replaced. The parameters to `replace()` are identical in type, order, and purpose to those for the `T *data` constructor for the sequence.

For the `T *data` sequence constructor and for the buffer parameter of the `replace()` function, the type of `T` for strings and object references is `char*` and `T_ptr`, respectively. In other words, string buffers are passed as *`char**`* and object reference buffers as `T_ptr*`. The return type of the non-const `get_buffer()` reference function for sequences of strings is `char**` and `T_ptr*` for sequences of object references. The return type of the `const get_buffer()` accessor function for sequences of strings is `const char* const*` and const T_ptr* for sequences of object reference.

Example

The following example shows declarations for an unbounded sequence.

```
// IDL
typedef sequence<T> V1; // unbounded sequence

// C++
class V1 // unbounded sequence
{
   public:
   V1();
   V1(ULong max);
   V1(ULong max, ULong length, T *data,
      Boolean release = FALSE);
   V1(const V1&);
   ~V1();
   V1 &operator=(const V1&);
   ULong maximum() const;
   void length(ULong);
   ULong length() const;
   T &operator[](ULong index);             // the exact type depends
on T
                                            // and is not specified
by the OMG
```

```
   const T &operator[](ULong index) const;   // the exact type depends
on T
                                              // and is not specified
by the OMG
   Boolean release() const;
   void replace(ULong max, ULong length, T *data,
   Boolean release = FALSE);
   T* get_buffer(Boolean orphan = FALSE);
   const T* get_buffer() const;
};
```

4.4.10.2 Bounded Sequences

The main difference between unbounded and bounded sequence types is that the maximum length of the bounded sequence is implicit in the sequence's type and cannot be modified. Like the unbounded sequence, however, the current length can be modified at any time. Attempting to set the current length to a value larger than the maximum length given in the IDL specification produces undefined behaviour.

Other minor differences are noted in the member function descriptions below.

The default constructor sets the sequence length equal to 0 (zero) and allocates a contents vector with the maximum size. the maximum length is part of the type and cannot be set or modified

The copy constructor creates a new sequence with the same maximum and length as the given sequence, copies each of its current elements (zero through length, minus one), and sets the release flag to `TRUE`.

The `T *data` constructor allows you to specify the contents as well as the initial length and release flag. The length should correspond to the size of the buffer, though the length and buffer need not be set to the maximum length. If the release flag is `FALSE` and a reallocation occurs, you are responsible for releasing the old storage. The new storage is allocated by the sequence and the release flag is set to `TRUE`. If the release flag is set to `TRUE`, storage for the given contents must have been allocated with the `allocbuf()` member function, described below.

If release equals `TRUE`, the destructor destroys each of the current elements (zero through length, minus one).

The assignment operator deep-copies its parameter, releasing old storage if release equals `TRUE`. It behaves as if the original sequence is destroyed via its destructor and the source sequence is copied using the copy constructor. Using the assignment operator always sets the release flag to `TRUE`.

The static `allocbuf` function returns a vector of `T` elements that can be passed to the `T* data` constructor. The length of the vector is given by the `nelems` function argument. The `allocbuf` function initializes each element using its default constructor, except for strings, which are initialized to empty strings, and object references, which are initialized to `nil` object references. A `null` pointer is returned if `allocbuf` cannot allocate the requested vector. Use the `freebuf` function to free vectors allocated by `allocbuf`.

The static `freebuf` function ensures that each element in the buffer is released appropriately. `freebuf` ignores `null` pointers passed to it.

The `maximum` function always returns the bound of the sequence as given in its IDL type declaration.

The length accessors allow you to manipulate the current length of the sequence. As noted above, setting the length to a value greater than the maximum length causes unpredictable behaviour. Setting the length to a value smaller than the maximum does not affect storage and only makes the sequence logically smaller.

The `index` operators, `operator[]`, return the item at the given index. The non-`const` version returns an `lvalue,` which allows read and write access to an item at a given index, while the `const` version allows only read access. The `index` operators cannot be used to access or modify any element beyond the current sequence length. Unless the sequence was constructed using the `T* data` constructor, you must set the length of the sequence using the `length(CORBA::ULong)` modifier function before using either form of `operator[]` on a sequence.

The `release()` accessor function returns the state of the sequence release flag.

The overloaded `get_buffer()` accessor and reference functions allow direct access to the buffer underlying a sequence. This can be very useful when sending large blocks of data as sequences, such as sending image data as a sequence of octet, where the per-element access provided by the overloaded subscript operators is not sufficient.

The non-const `get_buffer()` reference function allows read-write access to the underlying buffer. If its orphan argument is `FALSE` (the default), the sequence returns a pointer to its buffer, allocating one if it has not yet done so.

The const `get_buffer()` accessor function allows read-only access to the sequence buffer. The sequence returns its buffer, allocating one if one has not yet been allocated. No direct modification of the returned buffer by the caller is permitted.

The `replace()` function allows the buffer underlying a sequence to be replaced. The parameters for `replace()` are identical in type, order, and purpose to those for the `T *data` constructor for the sequence.

Example

The example below shows declarations for bounded sequence.

```
// IDL
```

```
typedef sequence<T, 2> V2; // bounded sequence

//C++
class V2 // bounded sequence
{
   public:
   V2();
   V2(ULong length, T *data, Boolean release = FALSE);
   V2(const V2&);
   ~V2();
   V2 &operator=(const V2&);
   ULong maximum() const;
   void length(ULong);
   ULong length() const;
   T &operator[](ULong index);             // the exact type depends
on T
                                             // and is not specified
by the OMG
   const T &operator[](ULong index) const;   // the exact type depends
on T
                                             // and is not specified
by the OMG
   Boolean release() const;
   void replace(ULong length, T *data,
   Boolean release = FALSE);
   T* get_buffer(Boolean orphan = FALSE);
   const T* get_buffer() const;
};
```

4.4.10.3 The Sequence _var Type

In addition to the regular operations defined for `T_var` types, the `T_var` for a sequence type also supports an overloaded *`operator[]`* that forwards requests to the `operator[]` of the underlying sequence. This subscript operator should have the same return type as that of the corresponding operator on the underlying sequence type.

Example

The following example demonstrates the use of the non-`const` `operator[]` for a sequence `_var` type for a bounded sequence of `CORBA::Long`.

```
  // IDL
  typedef sequence <long,100> Bnd_Long_Sequence;

  // C++
  Bnd_Long_Sequence_var bv = new Bnd_Long_Sequence;
  bv->length(100);
  for(int q=0; q<100; q++)
  {
    bv[q] = q;
  };
```

4.4.11 Array Types

Arrays are mapped to the corresponding C++ array definition, which allows the definition of statically-initialized data using the array. If the array element is a string, wide string, or an object reference, then the mapping uses the same type as for structure members. That is, the default constructor for string elements initializes them to the empty string ("") and assignment to an array element that is a string or object reference will release the storage associated with the old value.

Example

```
// IDL
typedef float F[10];
typedef string V[10];
typedef string M[1][2][3];
void op(out F p1, out V p2, out M p3);

// C++
typedef Float F[10];
typedef ... V[10]; // underlying type not shown
because
typedef ... M[1][2][3]; // it is implementation-dependent
F f1; F_var f2;
V v1; V_var v2;
M m1; M_var m2;
f(f2, v2, m2);
f1[0] = f2[1];
v1[1] = v2[1]; // free old storage, copy
m1[0][1][2] = m2[0][1][2]; // free old storage, copy
```

In the above example, the last two assignments result in the storage associated with the old value of the left-hand side being automatically released before the value from the right-hand side is copied.

Out and return arrays are handled via pointer to array slice, where a slice is an array with all the dimensions of the original specified except the first one. As a convenience for application declaration of slice types, the mapping also provides a typedef for each array slice type. The name of the slice typedef consists of the name of the array type followed by the suffix *_slice*.

Example

```
// IDL
typedef long LongArray[4][5];

// C++
typedef Long LongArray[4][5];
typedef Long LongArray_slice[5];
```

Both the `T_var` type and the `T_out` type for an array overload the `operator[]` instead of `operator->`. The use of array slices also means that the `T_var` type and the `T_out` type for an array have a constructor and assignment operator that each take a pointer to array slice as a parameter, rather than `T*`. The `T_var` for the previous example would be:

```
// C++
class LongArray_var
{
   public:
   LongArray_var();
   LongArray_var(LongArray_slice*);
   LongArray_var(const LongArray_var &);
   ~LongArray_var();
   LongArray_var &operator=(LongArray_slice*);
   LongArray_var &operator=(const LongArray_var &);
   LongArray_slice &operator[](ULong index);
   const LongArray_slice &operator[](Ulong index) const;
   const LongArray_slice* in() const;
   LongArray_slice* inout();
   LongArray_slice* out();
   LongArray_slice* _retn();
   // other conversion operators to support
   // parameter passing
};
```

A distinct C++ type whose name consists of the array name followed by the suffix *`_forany`* is provided for each array type in order to allow functions to be overloaded on them. Like `Array_var` types, `Array_forany` types allow access to the underlying array type, but unlike `Array_var`, the `Array_forany` type does not delete the storage of the underlying array upon its own destruction.

The interface of the *`Array_forany`* type is identical to that of the `Array_var` type Also, the `Array_forany` constructor taking an `Array_slice*` parameter also takes a Boolean `nocopy` parameter, which defaults to `FALSE`:

```
// C++
class Array_forany
{
   public:
   Array_forany(Array_slice*, Boolean nocopy = FALSE);
   ...
};
```

The `nocopy` flag allows for a non-copying insertion of an `Array_slice*` into an `Any`. Each `Array_forany` type must be defined at the same level of nesting as its `Array` type.

For dynamic allocation of arrays, compliant programs must use special functions defined at the same scope as the array type. For array `T`, the following functions will be available to a compliant program:

```
// C++
T_slice *T_alloc();
T_slice *T_dup(const T_slice*);
void T_copy(T_slice* to, const T_slice* from);
void T_free(T_slice *);
```

The *`T_alloc`* function dynamically allocates an array, or returns a null pointer if it cannot perform the allocation.

The *`T_dup`* function dynamically allocates a new array with the same size as its array argument, copies each element of the argument array into the new array, and returns a pointer to the new array. If allocation fails, a null pointer is returned.

The *`T_copy`* function copies the contents of the from array to the to array. If either argument is a null pointer, `T_copy` does not attempt a copy and results in no action being performed.

The *`T_free`* function deallocates an array that was allocated with `T_alloc` or `T_dup`. Passing a null pointer to `T_free` is acceptable and results in no action being performed. The `T_alloc`, `T_dup`, and `T_free` functions allow the ORB to utilize special memory management mechanisms for array types without replacing global operator `new` and operator `new[]`.

The `T_alloc`, `T_dup`, `T_copy`, and `T_free` functions do not throw `CORBA` exceptions.

Example Array_var

The following `_var` class is generated for an array called `A`:

```
class A_var
{
public:
  // Constructors
  A_var();
  A_var(A_slice* _slice);
  A_var(const A_var& that);

  // Destructor
  ~A_var();

  // Assignment Operators
  A_var& operator=(A_slice* s);
  A_var& operator=(const A_var& v);

  // Index Operators
  const A_slice& operator[](CORBA::ULong index) const;
```

```
    A_slice& operator[](CORBA::ULong index);

    const A_slice& operator[](int index) const;
    A_slice& operator[](int index);

    // Conversion Operators
    operator A_slice*();
    operator const A_slice*() const;

    //Access Operators
    const A_slice* in() const;
    const A_slice* inout();
    const A_slice* &out();
    const A_slice* _retn();
    const A_slice* &val(); //proprietary


  };
```

The difference between the `Array_forany` and the `Array_var` class is that `Array_forany` does not delete the contained array on destruction of the `Array_forany`. The only purpose for the `Array_forany` class is to give each array a distinct type for insertion into and extraction from an Any.

Example Array_forany

```
// IDL
typedef octet buffer10[10];

// C++
CORBA::Any          any;
const CORBA::ULong len = 10;
CORBA::ULong        u;
buffer10            b10;
buffer10_forany     b10fa;
for (u = 0; u < len; u++)
{
   b10[u] = (CORBA::Octet)u;
}
any <<= buffer10_forany(b10);

if (any >>= b10fa)
{
   for (u = 0; u < len; u++)
   {
      if (b10[u] == b10fa[u])
      {
         cout << "matched index: " << u << endl;
      }
    }
}
```

4.4.12 Any Types

The C++ mapping for the OMG IDL type *any* fulfils two different requirements:

- handling C++ types in a type-safe manner
- handling values whose types are not known at implementation compile time

The first item covers most normal usage of the any type: the conversion of typed values into and out of an any. The second item covers situations such as those involving the reception of a request or response containing an any that holds data of a type unknown to the receiver when it was created with a C++ compiler.

Since the Any type can contain any IDL-specified type (including base types, structs, unions, sequences, arrays, objects, interfaces, and anys), it also contains a TypeCode for the value so applications that can interpret the type of the data the any contains.

4.4.12.1 Handling Typed Values

Overloaded operators are used to insert and extract values into and from an Any in order to ensure that the Any contains a TypeCode and value that correctly matches the value's type. For primitive and predefined types, these operators already exist in the ORB libraries. For user-defined types, operators are generated by the IDL compiler. Overloaded operators depend on the C++ types being distinct. Special mechanisms are used to differentiate these non-distinct types.

- The basic IDL data types boolean, octet, and char are not necessarily mapped to distinct C++ data types. See *Handling boolean, octet, char, and unbounded string* on page 63 for information on using these types with an Any.
- In C++, all bounded and unbounded strings are mapped to char*, so a way to specify an bounded string is necessary. See *Handling boolean, octet, char, and unbounded string* on page 63 for more information on use of bounded strings with an Any.

C++ arrays decay into pointers to their first element when used within a function argument list. Overloaded functions cannot distinguish between arrays of different dimensions. See *Arrays in an Any* on page 66 for more information on use of arrays with an Any.

The IDL typedef keyword maps directly to the C++ typedef. Problems occur when user-defined types are typedef'd because the original and the new type are not distinct. In C++, the new typedef'd type is really of the original type. The Any will treat the type as the original type.

4.4.12.2 Insertion Into Any

Use predefined or generated overloaded operators for each distinct IDL type to ensure type-safe insertion of values into an Any.

Use the following form for each IDL type T for types that are typically passed by value, such as short, long, ushort, ulong, float, double, enumerations, unbounded strings, and object references:

```
// C++
void operator<<=(Any&, T);
```

For types that are not typically passed by value, such as `structs`, `unions`, `sequences`, `TypeCodes`, and `Anys`, the IDL compiler generates a copying and a non-copying form of insertion operator for each IDL type `T`:

```
// C++
   void operator<<=(Any&, const T&);    // copying form
   void operator<<=(Any&, T*);          // non-copying form
```

The copying form of the insertion operator makes a completely independent copy of the passed-in value. This means that the lifetimes of the value passed into the `Any` and the copy inside the `Any` are independent. With the non-copying form of the insertion operator, the `Any` assumes memory management for the passed-in value and deletes or releases the value when the `Any` goes out of scope. The `Any` always deallocates previous values, whether copied or not, when new values are inserted into it.

Some examples:

```
// IDL
struct mystruct
{
  long l;
  short s;
};

typedef unsigned long newulong;

interface anInterface
{
  void op();
};
// C++
CORBA::Any any;
CORBA::ULong ul = 77;
newulong nu = 88;
   any <<= ul;       // inserts ULong into the Any inserts
   any <<= nu;       // newulong into the Any, loses
                     // the identity - the Any thinks it
                     // contains a CORBA::ULong not a newulong

CORBA::String str = CORBA::string_dup(“Hello”);
any <<= str;
CORBA::string_free(str);    // free str since the Any copied it
mystruct ms;
ms.l = 10000;
ms.s = 44;
any <<= ms;           // copies the struct into the Any
mystruct * msp = CORBA_new mystruct;
msp->l = 87654;
msp->s = 22;
any <<= msp;          // the Any consumes the struct, so don’t
                      // delete it

CORBA::Any anotherany;
anotherany <<= any;   // copies one Any into another Any
anInterface_ptr aip;  // initialized somehow
any <<= aip;         // inserts anInterface_ptr into the Any (copies)
```

Insertion of `boolean`, `octet`, `char`, unbounded strings, and `arrays` are special cases described in following sections.

4.4.12.3 Extraction from Any

Use predefined or generated overloaded operators for each distinct IDL type in order to ensure type-safe extraction of values from an `Any`.

For types that are typically passed by value, such as `Short`, `Long`, `UShort`, `ULong`, `Float`, `Double`, enumerations, and unbounded strings, use the following form for each IDL type `T`:

```
// C++
Boolean operator>>=(const Any&, T&);
```

For types that are not typically passed by value, such as `structs`, `unions`, `sequences`, `TypeCodes`, and `Anys`, use the following form for each IDL type `T`:

```
// C++
Boolean operator>>=(const Any&, T*&);
```

For object references, the IDL compiler generates an extraction operator for each IDL interface type `T`:

```
// C++
Boolean operator>>=(const Any&, T_ptr&);
```

If the values are extracted successfully, the extraction operators return `TRUE`. Extracted values are always memory managed by the `Any`, so do not release or delete the extracted values. If those values need to exist beyond the scope of the `Any`, copy the values explicitly.

Some examples:

```
// IDL
struct mystruct
{
  long l;
  short s;
};

interface anInterface
{
  void op();
};
// C++
CORBA::Any any;
CORBA::ULong ul = 77;
CORBA::ULong nl = 0;
   any <<= ul;        // inserts ULong into the Any
   if (any >>= nl)    // extracts ULong from the Any
   {
  // use extracted value
   }
CORBA::String str = CORBA::string_dup("Hello");
CORBA::String nstr = nil;
```

```
any <<= str;
CORBA::string_free(str);    // free str since the Any copied it

if (any >>= nstr)           // extracts string from the Any
{
  // use extracted value, don't call CORBA::string_free() on nstr
}
mystruct ms;
ms.l = 10000;
ms.s = 44;
any <<= ms;           // copies the struct into the Any
mystruct * msp = nil;
if (any >>= msp)              // extracts struct from the Any
{
  // use extracted value, don't delete msp
}
anInterface_ptr aip;  // initialized somehow

any <<= aip;          // copies anInterface_ptr into the Any
anInterface_ptr naip = anInterface::_nil();

if (any >>= naip)     // extracts objref from the Any
{
  // use extracted value, don't release naip
}
```

Extraction of `boolean`, `octet`, `char`, `unbounded strings`, and `arrays` are special cases described below.

4.4.12.4 Handling boolean, octet, char, and unbounded string

Since the `boolean`, `octet`, and `char` OMG IDL types are not required to map to distinct C++ types, another means of distinguishing them from each other is necessary so that they can be used with the type-safe `any` interface. Similarly, since both bounded and unbounded strings map to `char*` another means of distinguishing them must be provided. Helper structs with defined insertion and extraction operators are provided in the `Any` class for distinguishing values of each type.

```
// C++
namespace CORBA
{
  class Any
  {
    ...
    struct from_boolean {
    from_boolean(Boolean b) : val(b) {}
    Boolean val;
  };

  struct from_octet {
    from_octet(Octet o) : val(o) {}
    Octet val;
  };

  struct from_char {
    from_char(Char c) : val(c) {}
    Char val;
  };
```

```
  struct from_string {
    from_string(char* s, ULong b, Boolean nocopy = FALSE)
    : bound(b) { if (nocopy) val = s; else val = string_dup(s); }
    char* val;
    ULong bound;
  };

  void operator<<=(from_boolean);
  void operator<<=(from_char);
  void operator<<=(from_octet);
  void operator<<=(from_string);

  struct to_boolean {
    to_boolean(Boolean &b) : ref(b) {}
    Boolean &ref;
  };

  struct to_char {
    to_char(Char &c) : ref(c) {}
    Char &ref;
  };

  struct to_octet {
    to_octet(Octet &o) : ref(o) {}
    Octet &ref;
  };

  struct to_string {
    to_string(char *&s, ULong b) : ref(s), bound(b) {}
    char *&ref;
    ULong bound;
  };


    struct to object
    {
      to_object(CORBA_Object_ptr &obj) : ref(obj)  {}
      CORBA_Object_ptr &ref;
    };

    struct to exception
    {
      to_exception(CORBA_Exception *&exc) : ref(exc) {}
      CORBA_Exception *&ref;
    };

    CORBA_Boolean   operator>>=(to_boolean)  const;
    CORBA_Boolean   operator>>=(to_char)  const;
    CORBA_Boolean   operator>>=(to_octet)  const;
    CORBA_Boolean   operator>>=(to_string)  const;
    CORBA_Boolean   operator>>=(to_object)  const;
    CORBA_Boolean   operator>>=(to_exception)  const;

...
 };
};
```

Use helper structs to insert and extract these potentially indistinct types to ensure portability with other ORBs. The helper structs for `boolean`, `octet` and `char` are virtually identical. For example:

```
// C++
CORBA::Any any;

CORBA::Boolean b1, b0 = TRUE;

any <<= CORBA::Any::from_boolean(b0);

if (any >>= CORBA::Any::to_boolean(b1))
{
  // use b1 (which is now CORBA_TRUE)
}

CORBA::Octet o1, o0 = 8;

any <<= CORBA::Any::from_octet(o0);

if (any >>= CORBA::Any::to_octet(o1))
{
  // use o1 (which is now 8)
}

CORBA::Char c1, c0 = 'A';

any <<= CORBA::Any::from_char(c0);

if (any >>= CORBA::Any::to_char(c1))
{
  // use c1 (which is now 'A')
}
```

In helper structs for bounded strings, a `bound` parameter specifies the bound of the string. To insert or extract an unbounded string, use the bounded string helper structs and set the bound value to zero (0). The insertion `from_string` struct also has a `nocopy` parameter, which specifies whether the `Any` should copy or consume the string. Setting the `nocopy` boolean to `FALSE`, the default, instructs the `Any` to copy the passed-in string. For example:

```
// IDL
typedef string<6> String6;


// C++
CORBA::Any any;

String6 s0 = CORBA::string_dup("hello");
String6 s1 = nil;

// copy insertion of a String6
any <<= CORBA::Any::from_string(s0, 6);

if (any >>= CORBA::Any::to_string(s1, 6))
{
  // use s1, don't call CORBA::string_free() on it
}
```

```
// nocopy insertion of a String6, any consumes s0
any <<= CORBA::Any::from_string(s0, 6,  TRUE);

if (any >>= CORBA::Any::to_string(s1, 6))
{
  // use s1, don't call CORBA::string_free() on it
}

CORBA::String us0 = CORBA::string_dup("my_unbounded_string");
CORBA::String us1;

// normal copy insertion of an unbounded string
any <<= us0;

// normal extraction of unbounded string
if (any >>= us1)
{
  // use us1, don't call CORBA::string_free() on it
}

// copy insertion of an unbounded string using from_string
any <<= CORBA::Any::from_string(us0, 0);

// nocopy insertion of an unbounded string, any consumes us0
any <<= CORBA::Any::from_string(us0, 0,  TRUE);

// use of to_string to extract an unbounded string
if (any >>= CORBA::Any::to_string(us1, 0))
{
  // use us1, don't call CORBA::string_free() on it
}
```

4.4.12.5 Arrays in an Any

Because C++ arrays decay into pointers to their first element when used within a function argument list, overloaded functions cannot distinguish between arrays of different dimensions. The IDL compiler generates an `Array_forany` helper class for each user-defined IDL array. The `Array_forany` class is named by pre-pending the IDL name to the `_forany` suffix. For example, in the code sample below, the `buffer10` array forms the `buffer10_forany` class.

`Array_forany` class behaviour is almost identical to `Array_var`, in that it allows access to its contained array. The difference between the `Array_forany` and `Array_var` classes is that the `Array_forany` does not delete the contained array upon destruction of the `Array_forany`. The only purpose for the `Array_forany` class is to give each array a distinct type for insertion into and extraction from an `Any`. For example:

```
// IDL
typedef octet buffer10[10];
// C++
CORBA::Any          any;
const CORBA::ULong len = 10;
CORBA::ULong        u;
buffer10            b10;
```


PRISMTECH

```
buffer10_forany    b10fa;
for (u = 0; u < len; u++)
{
  b10[u] = (CORBA::Octet)u;
}

any <<= buffer10_forany(b10);
if (any >>= b10fa)
{
  for (u = 0; u < len; u++)
  {
    if (b10[u] == b10fa[u])
    {
      cout << “matched index: “ << u << endl;
    }
  }
}
```

4.4.12.6 Extracting to Object

Extracting an object reference from an `Any` normally requires a `_ptr` to the object. This may not be a reasonable implementation for some applications because the application would need a `_ptr` instance for every object type.

A helper struct is available to extract object references from `Any`s in a generic fashion. If an `Any` contains an object reference, that reference can be explicitly widened to a `CORBA::Object_ptr` if it is extracted using the `to_object` struct. For example:

```
// IDL
interface anInterface
{
  void op();
};

// C++
CORBA::Any        any;
anInterface_ptr   aip;  // initialized somehow
CORBA::Object_ptr obj = CORB::Object::_nil();
any <<= aip;
if (any >>= CORBA::Any::to_object(obj))
{
  // use obj, don’t call CORBA_release() when done
  // could call anInterface::_narrow(obj) to get anInterface_ptr
}
```

4.4.12.7 Handling Untyped Values

i

The handling of untyped values is covered in version 1.0 (June 1999) of the *IDL to C++ Mapping Specification*. Spectra ORB C++ Edition includes this feature in accordance with the v1.0 specification.

However, we do not recommend using the `replace` and `value` functions (described below) since they are not type-safe and the `void *` values are defined by CORBA to be ORB-implementation-specific, so are not portable.

The default constructor creates an Any with the TypeCode set to CORBA::_tc_null and no value. Using the unsafe constructor, which takes a TypeCode, void * value, and boolean release parameter, is identical to creating an Any using the default constructor, then calling the Any's replace() function described in the previous section. We do not recommend use of this constructor because of the type-safety and portability issues described previously.

The copy constructor and the assignment operator both release the target's current TypeCode and value, duplicate the TypeCode from the source Any into the target Any, then deep-copy the value. The Any destructor releases the current TypeCode, then deletes or releases the current value.

Three methods are defined for the Any class for situations where the type-safe insertion and extraction operators are not sufficient.

```
// C++
namespace CORBA
{
  class Any
  {
    ...
    void replace(TypeCode_ptr, void * value,
                                     Boolean release = FALSE);
    TypeCode_ptr type() const;
    const void * value() const;
    ...
  };
};
```

The replace function allows you to insert a value into an Any without type safety. You are responsible for making sure the TypeCode and value are consistent. The behaviour of the Any is unpredictable if the value is not consistent with the TypeCode. If the release parameter is set to FALSE, the Any copies the value. Otherwise, the Any consumes the value. If the Any consumes the value, do not use the passed-in value pointer because the value may be copied and then deleted or released by the Any.

The type() function returns a duplicated TypeCode_ptr for the current contents of the Any that you must release when it is no longer needed.

The value() function returns a void pointer to the current contents of the Any.

4.4.12.8 Any_var and Any_out

Because Anys are returned via pointer as out and return parameters there exists an Any_var class similar to the T_var classes for object references. Any_var obeys the rules for T_var classes calling delete on its Any* when it goes out of scope or is otherwise destroyed. An Any_out class is also available that is similar in form to the T_out class.

4.5 TypeCodes

`TypeCodes` are CORBA pseudo-objects that contain structural information about IDL types. Use `TypeCodes` in the `Any` type to describe the contents of the `Any`.

A set of predefined `TypeCodes` covers all the primitive types and the predefined transmissible types such as `TypeCode` and `Object`.

Some of these predefined `TypeCodes` are:

```
// C++
namespace CORBA    // defined in eOrb/CORBA.h
{
      ...
  TypeCode_ptr _tc_short;
  TypeCode_ptr _tc_long;
      ...
  TypeCode_ptr _tc_octet;
  TypeCode_ptr _tc_any;
  TypeCode_ptr _tc_TypeCode;
  TypeCode_ptr _tc_Object;
 ...
};
```

The IDL compiler generates `TypeCodes` for all user-defined IDL types and IDL interfaces in the same scope as the type named `_tc_<typename>`.

For example:

```
// IDL
struct foo
{
  long aLong;
  short aShort;
};

typedef sequence <foo, 8> foo_seq;
interface anInterface
{
  oneway void op0(in foo f);
  boolean     op1(inout foo_seq fs);
};
```

The IDL compiler generates the following `TypeCodes` for the above IDL:

```
// C++
CORBA::TypeCode_ptr _tc_foo;
CORBA::TypeCode_ptr _tc_foo_seq;
CORBA::TypeCode_ptr _tc_anInterface;
```

The `TypeCode` `kind()` operation returns an enumerated value for the kind of type it describes. Calling `kind()` on the `_tc_foo` `TypeCode` generated in the example above returns `CORBA::tk_struct`. Typically, you would use the enumeration value returned from `kind()` to interpret the `TypeCode` and determine what other functions can be called. You can then call other `TypeCode` functions, which are specific to the different types of `TypeCodes`, to help describe the non-primitive types.

For example:

```
// C++
void
output_typecode(CORBA::TypeCode_ptr tc)
{
  CORBA::TCKind tck = tc->kind();

  switch (tck)
  {
    case CORBA::tk_short:
    {
      cout << “Short” << endl;
      break;
    }
    case CORBA::tk_long:
    {
      cout << “Long” << endl;
      break;
    }
    case CORBA::tk_objref:
    {
     CORBA::String_var * objname = tc->name();
      cout << “Object reference, name is: “ << objname << endl;
      break;
    }
    case CORBA::tk_struct:
    {
      CORBA::ULong u, len = tc->member_count();
     cout << “Struct, number of members: “ << len << “, members are:”
<< endl;
        for (u = 0; u < len; u++)
        {
          CORBA::TypeCode_var tmp_tc = tc->member_type(u);
          output_typecode(tmp_tc);
      }
      break;
    }
    case CORBA::tk_alias:
    {
      CORBA::TypeCode_var tmp_tc = tc->content_type();
        cout << “Alias, original type is:” << endl;
        output_typecode(tmp_tc);
        break;
    }
    case CORBA::tk_sequence:
    {
      CORBA::TypeCode_var tmp_tc = tc->content_type();
      cout << “Sequence, number of elements: “ << len
                                     << “, element type is:” << endl;
        output_typecode(tmp_tc);
        break;
    }
    default:
    cout << “Not an expected type.” << endl;
      break;
```

```
      }
    }
}
```

Calling `output_typecode(_tc_foo)` generates the following output:

```
Struct, number of members: 2, members are:
Long
Short
```

Calling `output_typecode(_tc_foo_seq)` generates the following output:

```
Alias, original type is:
Sequence, number of elements: 8, element type is:
Struct, number of members: 2, members are:
Long
Short
```

Calling `output_typecode(_tc_anInterface)` generates the following output:

```
Object reference, name is: anInterface
```

4.6 Exceptions

IDL exceptions are mapped to an exception class in C++ and behave like variable-length `structs`, regardless of whether or not the exception holds any variable-length members. Just as for variable-length `structs`, each exception member is self-managing with respect to its storage. All generated user-defined exceptions are derived from the `CORBA::UserException` base class, which allows you to write catch blocks that catch all user-defined exceptions.

The `UserException` class is derived from a base `CORBA::Exception` class, which is also defined in the `CORBA` module. All standard exceptions are derived from a `SystemException` class, also defined in the `CORBA` module. Like `UserException`, `SystemException` is derived from the base `Exception` class. The `SystemException` class interface is shown below.

```
// C++
enum CompletionStatus {
   COMPLETED_YES,
   COMPLETED_NO,
   COMPLETED_MAYBE
};
class SystemException : public Exception
{
   public:
   ~SystemException();
   ULong minor() const;
   void minor(ULong);
   virtual void _raise() const = 0;
   CompletionStatus completed() const;
   void completed(CompletionStatus);
   protected:
   SystemException();
   SystemException(const SystemException &);
   SystemException(ULong minor, CompletionStatus status);
```

```
    SystemException &operator=(const SystemException &);
};
```

The default constructor for `SystemException` causes *`minor()`* to return *`0`* and *`completed()`* to return *`COMPLETED_NO`*. Each specific system exception is derived from `SystemException`:

```
// C++
class UNKNOWN : public SystemException { ... };
class BAD_PARAM : public SystemException { ... };
// etc.
```

All specific system exceptions are defined within the `CORBA` module. This exception hierarchy allows any exception to be caught by simply catching the `Exception` type:

```
// C++
try {
...
} catch (const Exception &exc) {
...
}
```

Alternatively, all user exceptions can be caught by catching the `UserException` type, and all system exceptions can be caught by catching the `SystemException` type:

```
// C++
try {
...
} catch (const UserException &ue) {
...
} catch (const SystemException &se) {
...
}
```

The copy constructor, assignment operator, and destructor automatically copy or free the storage associated with the exception. For convenience, the code-generator also defines a constructor with one parameter for each exception member. This constructor initializes the exception members to the given values. The default constructor performs no explicit member initialization.

The following IDL for a user-defined exception maps to the C++ shown below:

```
  // IDL

  exception MyException
  {
    Object obj;
    string str;
  };
```

```
  // C++
```

```
class MyException
:
  public CORBA::UserException
{
public:
  // Members
  CORBA::Object_var obj;
  CORBA::String_var str;

  // Constructors
  MyException() {}
  MyException(CORBA::Object_var _obj, CORBA::String_var _str);
  MyException(const MyException& that);

  // Assignment operator
  MyException& operator=(const MyException& that);

  // Narrow
  static MyException* _downcast(CORBA::Exception *);
  static const MyException* _downcast(const CORBA::Exception*);

  // Duplicate
  virtual CORBA::Exception * _duplicate() const;

  // Repository id
  virtual const char * _rep_id() const;

  // Raise
  virtual void _raise() const;
};
```

In addition to the constructors, and assignment operator, the generated exception class provides a number of convenience functions.

The `_downcast` function is similar to the `_narrow` function used with object references. Unlike the `_narrow` for interfaces, however, the exception `_downcast` does not return a copy. If the exception is `nil`, `_downcast` returns `nil`. `_downcast` can be useful for determining the type of an exception in a `catch()` block, as is shown in the next example.

The `_rep_id()` allows for retrieval of an exception instance's repository ID. This can be useful for logging and debugging. The caller must not deallocate the returned string.

The `_raise` function allows an exception to be thrown and caught by its most derived type, even if the thrower does not have a pointer to the most derived type of the exception object.

Example

```
// C++
class MyException
:
  public CORBA::UserException
{
public:
```

```
    virtual void _raise() const;
    ...
  };

  The following code will not throw the exception as expected:

  try
  {
    CORBA::UserException * u_exc_p = new MyException;
    throw *u_exc_p;
  }
  catch(MyException &myexc)
  {
    cerr << "Caught MyException !" << endl;
  }
```

Compilers throw objects by their static type, not their dynamic type, so even though `u_exc_p` really points to an instance of `MyException`, the compiler throws an instance of `CORBA::UserException`.

Using `_raise`, however, will work because `_raise` is a virtual function that is implemented by each exception class derived from `CORBA::Exception` to throw itself. In this way, calling `_raise` on any `CORBA::Exception`-derived object always throws by the object's dynamic type:

```
  try
  {
    CORBA::UserException * u_exc_p = new MyException;
    u_exc_p->_raise();
  }
  catch(MyException &myexc)
  {
    cerr << "Caught MyException !" << endl;
  }
```

The following example demonstrates catching user-defined exceptions and the use of `_downcast` to determine the type of exception that was caught. Also shown is the use of `_rep_id` for debugging output. In CORBA, exceptions are thrown by value and caught by reference.

This approach lets the destructor release storage automatically.

```
  // IDL
  interface MyInterface
  {
    exception MyException1
    {
      string error_msg;
    };
    exception MyException2
    {
      string error_msg;
    };

    void op() raises(MyException1,MyException2);
  };
```

```
// C++
Object_var objref = ORB->resolve_initial_references("server");

MyInterface_var mi = MyInterface::_narrow(objref);
CORBA::release(objref);

if(!CORBA::is_nil(mi))
{
  try
  {
    mi->op();
  }
  catch(MyInterface::MyException1 &exc)
  {
    cerr << "Caught " << exc._repository_id() << ": ";
    cerr << exc.error_msg << endl;
  }
  catch(CORBA::UserException &exc)
  {
    // This catch block will use _downcast to try to identify
    // the exception
    // as a MyInterface::MyException2.
    MyInterface::MyException2 *me2 =
      MyInterface::MyException2::_downcast(&exc);
    if(me2)
    {
      cerr << "Caught " << me2._repository_id() << ": ";
      cerr << me2.error_msg << endl;
    }
    else
    {
      cerr << "Caught " << exc._repository_id() << endl;
    }
  }
  catch(CORBA::SystemException &exc)
  {
    cerr << "Caught " << exc._repository_id() << ": ";
    cerr << exc.minor() << " ";

    switch(exc.completed())
    {
      case CORBA::COMPLETED_YES:
      {
        cerr << "status: CORBA::COMPLETED_YES" << endl;
        break;
      }
      case CORBA::COMPLETED_NO:
      {
        cerr << "status: CORBA::COMPLETED_NO" << endl;
        break;
      }
      case CORBA::COMPLETED_MAYBE:
      {
        cerr << "status: CORBA::COMPLETED_MAYBE" << endl;
        break;
      }
    }
  }
}
CORBA::release(mi);
```

4.7 Operations and Attributes

Invoking operations on objects, plus attributes, operation signatures, and memory management topics are covered in this section.

4.7.1 Operations

An operation in IDL maps to a C++ method with the same name as the operation.

A *oneway* operation in IDL specifies a best-effort operation. A client invocation of a oneway operation is attempted once at most, and the delivery of the operation to the object implementation is not guaranteed. A oneway operation must have a void return value and in parameters only (no return value and no inout or out parameters). The operation also cannot have a raises expression, since the best-effort semantics do not allow the object implementation to notify the client of exceptional conditions. The C++ signature for a oneway operation does not differ from the signature for a normal operation with the same parameters.

4.7.2 Attributes

Each read-write attribute maps to a pair of overloaded C++ methods with the same name as the attribute. One method sets the attribute's value and one gets the attribute's value. The set method takes an input parameter with the same type as the attribute. The get method takes no parameters and returns the same type as the attribute.

For example, the time attribute in IDL maps to two overloaded C++ functions:

```
// IDL
interface timer
{
   attribute unsigned long time;
};
```

```
// C++ mapping of IDL
class timer ...
{
   virtual CORBA::ULong time(); // get
   virtual void time(CORBA::ULong); // set
};
```

An attribute marked *readonly* maps to the single C++ get function. Parameters and return types for attribute functions obey the same parameter passing rules as regular operations.

4.7.3 Parameters

There is a pre-defined behaviour for each mapped C++ parameter type. Inconsistencies between this behaviour and the use of the operation parameters can cause memory leaks or access violations.

The three parameter passing modes (`in`, `inout`, and `out`) determine operation signatures and memory management policies for the parameters. This section explains the memory management policies for parameter passing modes in the ORB.

The mapping for parameter passing attempts to optimise performance efficiency for each kind of IDL data type. Because fixed-length data type parameters can be stack-allocated in both the client and the server, they can be handled much more simply and efficiently than variable-length data types. Handling variable-length types is complicated by the issues of how and when parameter memory is allocated and deallocated.

Primitive and fixed-length arguments are passed by reference because they can be allocated on the stack for efficiency. However, a client passing an `inout` and `out` variable-length type, such as a string, may receive a value that requires more memory. To accommodate both kinds of allocation, the mapping is `T&` for a fixed-length aggregate type `T`, and `T*&` for a variable-length type `T`. This has the unfortunate consequence that use for a structured type depends on whether the type is fixed- or variable-length. The return value and parameter passing signatures for these types are listed in *Table 7*:

Table 7 Return Values and Signatures for non-_var Types

Data Type	In	Inout	Out	Return
`short`	`Short`	`Short&`	`Short&`	`Short`
`long`	`Long`	`Long&`	`Long&`	`Long`
`long long`	`LongLong`	`LongLong&`	`LongLong&`	`LongLong`
`unsigned short`	`UShort`	`UShort&`	`UShort&`	`UShort`
`unsigned long`	`ULong`	`ULong&`	`ULong&`	`ULong`
`unsigned long long`	`ULongLong`	`ULongLong&`	`ULongLong&`	`ULongLong`
`float`	`Float`	`Float&`	`Float&`	`Float`
`double`	`Double`	`Double&`	`Double&`	`Double`
`boolean`	`Boolean`	`Boolean&`	`Boolean&`	`Boolean`
`char`	`Char`	`Char&`	`Char&`	`Char`
`octet`	`Octet`	`Octet&`	`Octet&`	`Octet`
`enum`	`enum`	`enum&`	`enum&`	`enum`
`objref ptr`	`objref_ptr`	`objref_ptr&`	`objref_ptr&`	`objref_ptr`
`pseudo obj ptr`	`pobj_ptr`	`pobj_ptr&`	`pobj_ptr&`	`pobj_ptr`
`struct, fixed`	`const struct&`	`struct&`	`struct&`	`struct`
`struct, variable`	`const struct&`	`struct&`	`struct*&`	`struct*`

Table 7 Return Values and Signatures for non-_var Types(Continued) (Continued)

Data Type	In	Inout	Out	Return
union, fixed	const union&	union&	union&	union
union, variable	const union&	union&	union*&	union*
string	const char*	char*&	char*&	char*
sequence	const sequence&	sequence&	sequence*&	sequence*
array, fixed	const array	array	array	array_slice*
array, variable	const array	array	array_slice*&	array_slice*
any	const Any&	Any&	Any*&	Any*

Because `_var` types are available for interfaces, structs, unions, strings, sequences, arrays, and `Anys`, they can be passed as parameters in the same manner as their non-`_var` equivalent types. The `_var` types are self-memory-managing, so use of `_var` types by clients is slightly different than use of the non-`_var` types. The signatures for `_var` types are listed in *Table 8*

Table 8 Return Values and Signatures for _var Types

Data Type	In	Inout	Out	Return
objref var	const objref_var&	objref_var&	objref_var&	objref_var
pseudo obj var	const pobj_var&	pobj_var&	pobj_var&	pobj_var
struct_var	const struct_var&	struct_var&	struct_var&	struct_var
union_var	const union_var&	union_var&	union_var&	union_var
string_var	const string_var&	string_var&	string_var&	string_var
sequence_var	const sequence_var&	sequence&	sequence_var&	sequence_var
array_var	const array_var&	array_var&	array_var&	array_var
Any_var	const Any_var&	Any_var&	Any_var&	Any_var

4.7.3.1 In Parameters

`In` parameters are only passed from client to server, so parameter storage is caller-allocated and read-only.

On the client side, you must initialize all `in` arguments before the operation invocation. This is of particular interest in the case of parameters that are passed by pointer or pointer reference such as strings and arrays. Passing `_var` types as `in` parameters also requires the `_var` types to be initialized since they are de-referenced and passed by value.

On the server side, the ORB reconstructs a copy of the `in` parameter value before passing it to the object implementation. The ORB retains memory management responsibilities for the `in` parameters, so after the object implementation function returns, the local (server side) copies of the `in` parameter values are released by the ORB. The object implementation must, therefore, create a copy of an `in` value if it wants to retain the value after the method call returns.

4.7.3.2 Inout Parameters

`Inout` parameters are passed from client to object and from object back to client.

On the client side, `inout` parameters behave like `in` parameters and must be allocated and initialized to valid values before operation invocation. When the operation completes and returns, previous `inout` parameter values are released and memory is reallocated for the returned values. Releasing returned values when they are no longer needed is the client's responsibility.

When a `_var` type is passed as an `inout` parameter, the `_var` automatically deallocates any previously held value.

On the server side, the ORB reconstructs a copy of each `inout` parameter value before passing it to the function implementation. The object implementation can change value, but must release the value and reallocate it if the value is of a variable length type and the length is changed. After the object implementation function returns, the `inout` parameter values are passed back to the client and the local (server side) copies are released by the ORB. The object implementation must create a copy of an `inout` value to hold onto the value after the function call returns.

4.7.3.3 Out Parameters

`Out` parameters are passed from object to client.

On the client side, the caller is not required to initialize fixed-length parameter types. Variable-length parameter types (including Anys, strings, sequences, arrays, and variable-length structs and unions) that are returned by pointer, may be, but are not required to be, initialized to `nil`. When a `_var` type is passed as an out parameter, the `_var` automatically deallocates any previously held value.

```
// IDL
struct S
{
   string name;
   float age;
};
interface I
{
   void f(out S p);
};
// C++ (client-side code)
```

```
S_var s;
I_ptr i = ;      // initialized somehow

i->f(s);
... // use s
i->f(s);         // first result will be freed

S * sp;          // need not initialize before passing to out
i->f(sp);
// use sp
delete sp;       // cannot assume next call will free old value
i->f(sp);
```

For ensuring that types can be passed as `out` parameters by either their true type or their `_var` type, the ORB either pre-defines an `_out` helper class or the IDL compiler generates the `_out` helper class for the following `out` parameter types: string, object references, arrays, Anys, sequences, and variable length structs and unions. These `_out` classes are used in the generated operation signatures so that the memory management behaviour is correct regardless of whether the `_var` or non-`_var` types are passed as parameters to the operation.

On the server side, the object implementation must allocate and assign `out` parameter values. For values that are returned by pointer, (Anys, strings, sequences, arrays, and variable-length structs and unions) the server must always return a valid non-`nil` pointer to a value. After the object implementation function returns, the `out` parameter values are passed back to the client and the local (server side) copies are released by the ORB. The object implementation must create a copy of an `out` value to retain the value after the function call returns.

4.7.3.4 Return Values

The memory management policies for return values are generally the same as those for `out` parameters, with the exception of fixed-length array type parameters. A fixed length array is returned as a pointer to an array slice. See the array section for more information on array slices. For values that are returned by pointer, (Anys, strings, sequences, arrays, and variable length structs and unions) the object must always allocate and return a valid non-`nil` pointer to a value. After the object implementation function returns, the return values are passed back to the client and the local copies are released by the ORB. The object implementation must create a copy of a return value to retain the value after the function call returns.

4.7.3.5 Primitives

IDL primitive data types (boolean, octet, char, (unsigned) short, (unsigned) long, float, double and enumerations) are passed just as normal C++ built-in data types.

The IDL operation below demonstrates the memory management policies for primitive type parameters for `in`, `inout`, and `out` parameter modes and the return value.

```
   // IDL
   interface primitive_interface
   {
      long op(in short s, inout unsigned long ul, out float f);
   };

   // C++ mapping of operation op()
   virtual CORBA::Long op(CORBA::Short s,
                          CORBA::ULong& ul,
                          CORBA::Float& f);
```

For each `in` or `inout` parameter, the client creates and initializes an instance of the parameter type. For `inout` parameters, the client provides the initial value, and the object can change that value. For `out` parameters, the client allocates the storage but need not initialize it, and the object sets the value. Function returns are by value.

```
   // C++ (client code)
   CORBA::Long l;
   CORBA::Short s = 7;
   CORBA::ULong ul = 888;
   CORBA::Float f;

   l = srvr->op(s, ul, f);
```

`In` parameters are passed by value. `Inout` and `out` parameters are passed by reference. The object can change the value of the `inout` parameter, `ul`, set the value of the `out` parameter, `f`, and return a local variable value.

```
// C++ (object implementation)
   CORBA::Long
   prim_impl::op(CORBA::Short s, CORBA::ULong& ul, CORBA::Float& f)
   {
    CORBA::Long ret = (CORBA::Long)(s * 2);

      ul += ul;
      f = (CORBA::Float)(s * 4.4F);

      return ret;
   }
```

4.7.3.6 Fixed-Length structs and unions

Fixed-length `struct` and `union` types behave like C++ built-in types, except that `in` parameters are passed by `const` reference instead of by value.

The IDL operation below demonstrates the memory management policies for fixed-length `struct` type parameters for `in`, `inout`, and `out` parameter modes and the return value. The same rules apply to fixed-length `union`s.

```
// IDL
   // fl_struct is fixed-length because it contains no
   // variable-length members
```

```
   struct fl_struct
   {
      short row;
      short column;
   };

   interface fl_struct_interface
   {
      fl_struct op(in fl_struct fs1, inout fl_struct fs2, out
fl_struct fs3);
   };

// C++ mapping of the struct and op() operation
   struct fl_struct {
      CORBA::Short row;
      CORBA::Short column;
   };

   virtual fl_struct op(const fl_struct& fs1, fl_struct& fs2,
fl_struct& fs3);
```

For each `in` or `inout` parameter, the client creates and initializes an instance of the parameter type. For `inout` parameters, the client provides the initial value, and the object can change that value. For `out` parameters, the client allocates the storage but need not initialize it, and the object sets the value. Function returns are by value.

```
// C++ (client code)
   fl_struct fsret, fs1 = {3,4}, fs2 = {5,6}, fs3 = {0,0};

   fsret = srvr->op(fs1, fs2, fs3);
```

`In` parameters are passed by `const` reference. `Inout` and `out` parameters are passed by reference. The object can change the value of the `inout` parameter, `fs2`, set the value of the `out` parameter, `fs3`, and return a local variable value.

```
// C++ (object implementation)
   fl_struct
   fl_struct_impl::op(const fl_struct& fs1, fl_struct& fs2,
fl_struct& fs3)
   {
    fl_struct ret;

      ret.row = fs2.row + 1;
      ret.column = fs2.column + 1;

      fs2.row += fs1.row;
      fs2.column += fs1.column;

      fs3.row = fs1.row * 2;
      fs3.column = fs1.column * 2;

      return ret;
   }
```

4.7.3.7 Fixed-length arrays

Fixed-length array parameter types behave like C++ built-in types, except that the return value is passed as a pointer to a slice of the array, where a slice is an array with all the dimensions of the original specified except the first one.

The `op()` operation, shown below, demonstrates the memory management policies for fixed-length array type parameters for each of the parameter modes and the return value.

```
// IDL
   // segment is a fixed-length array because its element type
   // is a fixed-length type
   const unsigned long arraylen = 4;
   typedef long segment[arraylen];

   interface fl_array_interface
   {
      segment op(in segment s1, inout segment s2, out segment s3);
   };

// C++ mapping of the array and op() operation
   static const CORBA::ULong arraylen = 4;
   typedef CORBA::Long segment_slice;
   typedef CORBA::Long segment[4];
   extern segment_slice * segment_alloc();
   extern void segment_free(segment_slice *);
   extern void segment_copy(segment_slice* trg, const segment_slice*
src);
   extern segment_slice *segment_dup(const segment_slice* src);

   virtual segment_slice* op(segment s1, segment s2, segment s3);
```

For each `in` or `inout` parameter, the client creates and initializes an instance of the parameter type. Fixed-length array values can be statically initialized. For `inout` parameters, the client provides the initial value, and the object can modify that value. For `out` parameters, the client allocates the storage but need not initialize it, and the object sets the value. The function return value is dynamically allocated using the array’s `_alloc` function and the client must free this value when it is no longer needed using the array's `_free` function.

```
// C++ (client code)
   segment_slice * sret = nil;
   segment s1 = { 10, 11, 12, 13 };
   segment s2 = { 20, 21, 22, 23 };
   segment s3 = { 0, 0, 0, 0 };

   sret = srvr->op(s1, s2, s3);
   segment_free(sret);        // free the return value
```

`In`, `inout`, and `out` parameters are passed as the array type. The array value may be allocated from the stack. The server can change the value of the `inout` parameter, `s2`, and set the value of the `out` parameter, `s3`. The return value must be allocated dynamically because C++ does not allow a function to return an array. To dynamically allocate an array value, you must use the `_alloc` function for that array type. In the function implementation below the return `segment_slice *` is allocated with the segment type's `_dup` function, which calls the `_alloc` function to allocate the segment.

```
// C++ (object implementation)
   segment_slice*
   fl_array_impl::op(segment s1, segment s2, segment s3)
   {
    // duplicate s1 into the return segment slice
    segment_slice* ret = segment_dup(s1);

      for (CORBA::ULong u = 0; u < arraylen; u++)
      {
         s3[u] = s2[u] * 4;      // set s3 values to 4 times
                                 // s2 values
         s2[u] = s1[u] * 2;      // set s2 values to 2 times
                                 // s1 values
      }

      return ret;
   }
```

4.7.3.8 Strings

String parameters are variable-length types. `Inout`, `out`, and return types must be dynamically allocated using `string_alloc()` and freed using `string_free()`.

The `op()` operation, shown below, demonstrates the memory management policies for string parameters for each of the parameter modes and the return value.

```
   // IDL
   interface string_interface
   {
      string op(in string s1, inout string s2, out string s3);
   };


   // C++ mapping of the operation op()
   virtual CORBA::String op(const char* s1,
                            char* & s2,
                            CORBA::String_out s3);
```

The `String_out` type behaves like `char*&`, but manages string value memory for any `String_var` arguments. You should not declare an instance of one of these types. Treat the function as though it had this signature:

```
virtual char* op(const char* s1, char*& s2, char*& s3);
```

For each `in` or `inout` parameter, the client must create and initialize a non-`nil` string value that is `null`-terminated. An `in` string parameter can be a literal or `const`. `inout` string parameters should be allocated using `string_alloc()`. The object can change that value.

For `out` parameters, the client allocates the string pointer, but not the string. You may initialize the `out` value to `nil` since this value is not passed to the object.

The client is responsible for freeing `inout`, `out`, and return string values when they are no longer needed. String values should be freed using `string_free()`.

```
// C++ (client code)
CORBA::String sret = nil;
const char * s1 = "String1";
CORBA::String s2 = CORBA::string_dup("String2");
CORBA::String s3 = nil;

sret = srvr->op(s1, s2, s3);

CORBA::string_free(sret);
CORBA::string_free(s2);
CORBA::string_free(s3);
```

The object may deallocate the `inout` string and reassign the `char*` to point to new storage to hold the output value. `Inout` and `out` string values should be allocated using `CORBA::string_alloc` or `CORBA::string_dup`. The size of the `inout` string upon return is not limited by the size of the `inout` string when the function was called. The object is not allowed to return a `null` pointer for an `inout`, `out`, or return value.

```
// C++ (object implementation)
CORBA::String
string_impl::op(const char* s1,
                char* & s2,
                CORBA::String_out s3)
{
 // duplicate s1 into the return value
 CORBA::String ret = CORBA::string_dup(s1);
    // duplicate s2 into s3
   s3 = CORBA::string_dup(s2);

   // free s2, set it to a new string
   CORBA::string_free(s2);
   s2 = CORBA::string_dup("newstring");

   return ret;
}
```

4.7.3.9 Object References

The client allocates storage for the object reference. In the following example, the `op()` operation demonstrates the memory management policies for object reference parameters for each of the parameter modes and the return value.

```
// IDL
   interface objref_interface
   {
      objref_interface factory();

      objref_interface op(in    objref_interface oi1,
                          inout objref_interface oi2,
                          out   objref_interface oi3);
   };
```

```
   // C++ mapping for the factory() and op() operations
   virtual objref_interface_ptr factory();

   virtual objref_interface_ptr op(objref_interface_ptr oi1,
                                   objref_interface_ptr& oi2,
                                   objref_interface_out oi3);
```

The `objref_out` type behaves like `objref_ptr&`, but manages `objref` value memory for `objref_var` arguments. You should not declare an instance of one of these types. Treat the function as though it had this signature:

```
virtual objref_interface_ptr op(objref_interface_ptr oi1,
   objref_interface_ptr& oi2, objref_interface_ptr& oi3);
```

For each `in` or `inout` parameter, the client must create and initialize an object reference instance of the parameter type. For `inout` parameters, the client provides the initial value, and the object may change that value. To continue to safely use an object reference that is passed as an `inout`, the client must first duplicate the reference.

For `out` parameters, the client allocates the pointer for the object reference but need not initialize it, and the object sets the value. Function returns are by value.

The client is responsible for the release of all `inout`, `out` and return object references. Release of all object references embedded in other structures is performed automatically by the structures themselves.

```
// C++ (client code)
   objref_interface_ptr oiret = objref_interface::_nil();
   objref_interface_ptr oi1   =

orb->resolve_initial_references("ObjectReference1");
   objref_interface_ptr local =

orb->resolve_initial_references("ObjectReference2");
   objref_interface_ptr oi2   = objref_interface::_duplicate(local);
   objref_interface_ptr oi3   = objref_interface::_nil();

   oiret = srvr->op(oi1, oi2, oi3);
   CORBA::release(oiret);
   CORBA::release(oi1);
   CORBA::release(oi2);
   CORBA::release(oi3);
   CORBA::release(local);

   oiret = srvr->factory();

   CORBA::release(oiret);
```

`In` parameters are passed by value. `Inout` and `out` parameters are passed by reference. If the server object wants to reassign the `inout` parameter, it must first call `CORBA::release` on the original input value before reassignment.

```
// C++ object implementation for the factory operation
   //   The factory implementation assumes a objref_impl class
   //   data member:
   //
   //      objref_interface_ptr m_local_objref;
   //
   //   If this local object reference is currently set to a
   //   nil value when this operation is called, a new
   //   implementation is created. The return value is set to
   //   a duplicate of this objref. This also assumes that the
   //   local objref is released somewhere else.
   //
   objref_interface_ptr
   objref_impl::factory()
   {
      if (CORBA::is_nil(m_local_objref))
      {
         objref_impl*impl = new objref_impl;
         poa->activate_object(impl EORB_ENV_VARN);
      }
      return impl->_this;
   }
// C++ object implementation for the op operation
   objref_interface_ptr
   objref_impl::op(objref_interface_ptr oi1,
                   objref_interface_ptr& oi2,
                   objref_interface_out oi3)
   {
    // duplicate oi1 into the return value
    objref_interface_ptr ret = objref_interface::_duplicate(oi1);

      // duplicate oi2 into oi3
      oi3 = objref_interface::_duplicate(oi2);

      // release oi2, then duplicate oi1 into oi2
      CORBA::release(oi2);
      oi2 = objref_interface::_duplicate(oi1);
      return ret;
   }
```

4.7.3.10 Variable-Length structs and unions

Using variable-length aggregate types is more complex than using their fixed-length equivalents because of the additional memory-management issues involved.

The IDL operation below demonstrates the memory management policies for variable-length struct type parameters for `in`, `inout`, and `out` parameter modes and the return value. The same rules apply to variable-length unions.

```
// IDL
   struct vl_struct
   {
      short  row;
      short  column;
      string name;
   };

   interface vl_struct_interface
```

```
   {
      vl_struct op(in vl_struct vs1,
                   inout vl_struct vs2,
                   out vl_struct vs3);
   };

// C++ mapping for the struct and op() operation
   struct vl_struct {
      CORBA::Short row;
      CORBA::Short column;
      CORBA::String_var name;
   };

   virtual vl_struct* op(const vl_struct& vs1,
                         vl_struct& vs2,
                         vl_struct_out vs3);
```

The `vl_struct_out` type behaves like a `vl_struct*&`, but manages the variable-length struct's value memory for `vl_struct_var` arguments. You should not declare an instance of one of these types. Treat this function as though it had this signature:

```
virtual vl_struct* op(const vl_struct& vs1,
   vl_struct& vs2, vl_struct*& vs3);
```

For each in or inout parameter, the client creates and initializes an instance of the parameter type. For inout parameters, the client provides the initial value, and the server may change that value. For out parameters, the client allocates the storage but need not initialize it, and the object sets the value. Function returns are by pointer.

Aggregate types are complicated by the question of when and how parameter memory is allocated and deallocated. Mapping `in` parameters is straightforward because the parameter storage is client-allocated and read-only. For `out` parameters, the client allocates a pointer and passes it by reference to the object.

The client must always release the returned storage, regardless of whether the object is colocated or remote, in order to maintain location transparency. The client is not allowed to modify any values in the returned storage following completion of a request. If it is necessary to change the values, then the client must first copy the returned instance in a new instance, then modify the new instance.

```
   // C++ (client code)
   vl_struct * vsret = nil;
   vl_struct   vs1, vs2;
   vl_struct * vs3 = nil;

   vs1.row = 11;
   vs1.column = 12;
   vs1.name = CORBA::string_dup("somestructname1");
  // struct's name consumes the char* returned by CORBA::string_dup()
   vs2.row = 15;
   vs2.column = 16;
   vs2.name = (const char*)"somestructname2";
   // struct's name copies the const char*
```

```
   vsret = srvr->op(vs1, vs2, vs3);

   // delete the return and out structs
   delete vsret;
   delete vs3;
```

`in` parameters are passed by `const` reference. `Inout` and `out` parameters are passed by reference. The object can change the value of the `inout` parameter, `vs2`, set the value of the `out` parameter, `vs3`, and return a local variable value. The object is not allowed to return a `null` pointer for `out` parameters or return values. In both cases, the client is responsible for releasing the returned storage.

```
   // C++ (object implementation)
   vl_struct*
   vl_struct_impl::op(const vl_struct& vs1, vl_struct& vs2,
vl_struct_out vs3)
   {
    vl_struct* ret = new vl_struct;

      vs3 = new vl_struct;

      vs3->row = vs2.row + 2;
      vs3->column = vs2.column + 2;
      vs3->name = vs2.name;

      vs2.row = vs1.row * 2;
      vs2.column = vs1.column * 2;
      vs2.name = vs1.name;

      ret->row = 7;
      ret->column = 77;
      ret->name = CORBA::string_dup("newname");

      return ret;
   }
```

4.7.3.11 Variable-Length arrays

Variable-length `array` parameter types behave like C++ built-in types, except that the return value is passed as a pointer to a slice of the array, where a slice is an array with all the dimensions of the original specified except the first one.

The following example `op()` operation demonstrates the memory management policies for variable-length array type parameters for each of the parameter modes and the return value.

```
// IDL
// vsegment is a variable-length array because its element type
// is a variable-length type
struct vl_struct
{
   short  row;
   short  column;
   string name;
};
const unsigned long arraylen = 2;
typedef vsegment vl_struct[arraylen];
```

```
interface vl_array_interface
{
  vsegment op(in vsegment vs1, inout vsegment vs2, out vsegment vs3);
};

// C++ mapping for the struct, array and op() operation
struct vl_struct {
   CORBA::Short row;
   CORBA::Short column;
   CORBA::String_var name;
};

static const CORBA::ULong arraylen = 2;
typedef vl_struct vsegment_slice;
typedef vl_struct vsegment[2];
extern vsegment_slice * vsegment_alloc();
extern void vsegment_free(vsegment_slice *);
extern void vsegment_copy(vsegment_slice* trg, const vsegment_slice*
src);
extern vsegment_slice *vsegment_dup(const vsegment_slice* src);

virtual vsegment_slice* op(vsegment vs1,
                           vsegment vs2,
                           vsegment_out vs3);
```

The `vsegment_out` type behaves like a `vsegment_slice*&`, but manages `vsegment_slice` value memory for `vsegment_var` arguments. You should not declare an instance of one of these types. Treat this function as though it had this signature:

```
virtual vsegment_slice* op(vsegment vs1,
   vsegment vs2, vsegment_slice*& vs3);
```

For each `in` or `inout` parameter, the client must create and initialize an array instance. For `inout` parameters, the client provides the initial value, and the object may change that value. To continue to use an array passed as an `inout`, the client must first duplicate it before passing it as an operation parameter. For `out` parameters, the client must just pass a `vsegment_slice*&` initialized to nil.

```
   // C++ (client code)
   vsegment_slice * vsret = nil;
   vsegment         vs1, vs2;
   vsegment_slice * vs3 = nil;

   vs1[0].row = 20;
   vs1[0].column = 30;
   vs1[0].name = (const char*)"vs1[0].name";
   vs1[1].row = 21;
   vs1[1].column = 31;
   vs1[1].name = (const char*)"vs1[1].name";

   vs2[0].row = 40;
   vs2[0].column = 50;
   vs2[0].name = (const char*)"vs2[0].name";
   vs2[1].row = 41;
   vs2[1].column = 51;
```

```
   vs2[1].name = (const char*)"vs2[1].name";

   vsret = srvr->op(vs1, vs2, vs3);

   // free the return and out vsegment_slices
   vsegment_free(vsret);
   vsegment_free(vs3);
```

Out parameters and return values must be allocated dynamically by the object because C++ does not allow a function to return an array. To allocate an array value dynamically, use the _alloc function for that array type. In the function implementation below the return vsegment_slice * is allocated with the segment type's _dup function, which calls the _alloc function to allocate the vsegment. The out vsegment is allocated by the _alloc function, then filled in and returned.

```
// C++ (object implementation)
vsegment_slice*
vl_array_impl::op(vsegment vs1, vsegment vs2, vsegment_out vs3)
{
 // duplicate vs1 into the return value
 vsegment_slice* ret = vsegment_dup(vs1);

   // allocate the vs3 out vsegment
   vs3 = vsegment_alloc();

   for (CORBA::ULong u = 0; u < arraylen; u++)
   {
      // set the out struct's values to variations of vs2
      vs3[u].row = vs2[u].row * 4;
      vs3[u].column = vs2[u].column * 4;
      vs3[u].name = vs2[u].name;

      // modify the inout struct's values to variations of vs1
      vs2[u].row = vs1[u].row * 2;
      vs2[u].column = vs1[u].column * 2;
      vs2[u].name = vs1[u].name;
   }

   return ret;
}
```

4.7.3.12 Sequence and Anys

Sequence and Any parameters are variable-length types. They behave like variable length structs and unions.

The op() operation, shown below, demonstrates the memory management policies for sequence type parameters for each of the parameter modes and the return value. The same rules apply to anys.

```
// IDL
typedef sequence<long> vector;

interface sequence_interface
{
   vector op(in vector v1, inout vector v2, out vector v3);
```

```
};

// C++ mapping of the op() operation
   virtual vector* op(const vector& v1, vector& v2, vector_out v3);
```

The `vector_out` type behaves like a `vector*&`, but manages the sequence's value memory for `vector_var` arguments. You should not declare an instance of one of these types. Treat this function as though it had this signature:

```
virtual vector* op(const vector& v1, vector& v2, vector*&
v3);
```

For each `in` or `inout` parameter, the client creates and initializes an instance of the parameter type. For `inout` parameters, the client provides the initial value, and the server may change that value. For `out` parameters, the client allocates the storage but need not initialize it, and the object sets the value. Function returns are by pointer.

```
// C++ (client code)
vector * vret;
vector v1(4), v2(2);
vector * v3 = nil;
v1.length(4)
v1[0] = 10;
v1[1] = 11;
v1[2] = 12;
v1[3] = 13;
v2.length(2)
v2[0] = 20;
v2[1] = 21;

vret = srvr->op(v1, v2, v3);

delete vret;
delete v3;
```

For `inout` sequences and `Any`s, assignment or modification of the sequence or `Any` by the object may cause deallocation of owned storage before reallocation occurs. Therefore, if values passed in the sequence or `Any` from the client are used to set the sequence or `Any` for returning as the `out` value, the object should make a copy of the sequence or `Any` before re-assigning the `inout` values.

```
// C++ (object implementation)
vector*
sequence_impl::op(const vector& v1, vector& v2, vector_out v3)
{
 // create the return sequence and copy values from v1
 vector * ret = new vector(v1);
 CORBA::ULong u, len = 0;

   // create a new sequence v3 with same size as v2
   len = v2.length();
   v3 = new vector(len);

   // set v3 values to be 4 times v2 values
   for (u = 0; u < len; u++)
```

```
    {
        (*v3)[u] = v2[u] * 4;
    }

    // reset v2's length to be the same as v1
    len = v1.length();
    v2.length(len);

    // modify v2 values to be 2 times v1 values
    for (u = 0; u < len; u++)
    {
        v2[u] = v1[u] * 2;
    }

    return ret;
}
```

INDEX

Index

Symbols

_alloc function . . . 84, 91
_dup function . . . 84
_duplicate . . . 41
_narrow function . . . 73
_out helper class . . . 80
_rep_id . . . 74
_rep_id() . . . 73
_slice * return segment . . . 84
_tc_foo TypeCode . . . 69
_throw function . . . 73, 74
_var class . . . 58
_var type . . . 78, 80
  signatures . . . 78

A

access functions . . . 46
access violations . . . 76
Accessor
  length, unbounded sequence . . . 54
  maximum, bounded sequence . . . 54
Accessor function, discriminant . . . 47
Aggregate types . . . 88
  variable length . . . 87
allocbuf function . . . 53
Any
  Arrays . . . 66
  destructor . . . 68
  Extraction from . . . 62
  inserting a value . . . 68
  Insertion into . . . 60
  parameters . . . 91
Any class
  helper structs . . . 63
  unsafe operations . . . 68
Any type . . . 60, 69
Arguments
  fixed-length . . . 77
  in . . . 78
  objref_var . . . 86
  passing, consistent . . . 42
  primitive . . . 77
  string_var . . . 84
Array
  buffer10 . . . 66
  IDL . . . 10
  in an Any . . . 66
  members . . . 48
  parameter types
    fixed-length . . . 83, 89
Array_forany
  Array_var, differences . . . 59, 66
  helper class . . . 66
Array_var
  Array_forany, differences . . . 59, 66
Assignment operator . . . 41, 44, 68, 72
  sequence, bounded . . . 53
Attribute
  readonly . . . 76
  read-write . . . 76

B

Base class
  CORBA::UserException . . . 71
Basic data types
  mapping . . . 38
Best-effort operation . . . 76
boolean . . . 60
  data types
    IDL basic . . . 60
  helper structs . . . 65
  release flag . . . 49

types 39
Bounded
sequence, IDL 9
string, IDL 10
Bounded Sequence
Index Operators 54
Maximum Accessor 54
Memory Management Functions 53
Bounded sequence
assignment operator 53
constructors 53
destructor 53
mapping 53
Bounded strings 60
buffer10 array 66
buffer10_forany class 66
built-in types, C++ 81

C

C++
built-in types 81
namespace 30
scope names 29
union class 46
char 60
data types
IDL basic 60
helper structs 65
Classes
_var 58
buffer10_forany 66
C++ union 46
String_var 40
T_var 45
Client, object reference
storage allocation 85
Comments, IDL 6
Consistent argument passing 42
const pointer 44
Constant Types 11
Constructed Types
discriminated unions 8
enumerations 9
structures 8
Constructors
bounded sequence 53
copy 41, 44, 47, 53, 68, 72
default 41, 44, 68
object_var 44
sequence, bounded 53
String_var 44
unbounded sequence 49
union 47
unsafe 68
Copy constructor 41, 44, 47, 53, 68, 72
Copying form
insertion operator 61
CORBA::_tc_null 68
CORBA::Boolean 38
CORBA::Char 38
CORBA::Double 38
CORBA::Float 38
CORBA::Long 38
CORBA::Object_ptr
widening to 67
CORBA::Octet 38
CORBA::ORB::resolve_initial_references 45
CORBA::release 86
CORBA::Short 38
CORBA::string_alloc 40, 85
CORBA::string_dup 40, 85
CORBA::string_free 40
CORBA::tk_struct 69
CORBA::ULong 38
CORBA::UserException 71, 74
CORBA::UShort 38
Creating a type alias 38

D

Data Types
arrays 10

basic . . . 7
constructed . . . 8
distinct C++ . . . 60
IDL . . . 7
template . . . 9
Data, related group . . . 8
data, T * . . . 53, 54
Deallocation, sequence storage . . . 49
Debugging output . . . 74
Default constructor . . . 41, 44
Destructor . . . 41, 44, 72
Any . . . 68
bounded sequence . . . 53
sequence, bounded . . . 53
Discriminant
accessor function . . . 47
members . . . 46
Distinct C++ data types . . . 60

E

enum . . . 39
Exception class
convenience functions . . . 73
Exceptions
user-defined . . . 72
Extraction from Any . . . 62
Extraction operator
object references . . . 62
references, object . . . 62

F

Fixed
maps to . . . 45
Fixed-length
arguments . . . 77
array parameter types . . . 80, 83, 89
parameter types . . . 79
struct . . . 81
unions . . . 81
Flag, boolean release . . . 49
freebuf function . . . 54
from_string struct insertion . . . 65
Function
_alloc . . . 84, 91
_dup . . . 84
_narrow . . . 73
_throw . . . 73, 74
access . . . 46
accessor, discriminant . . . 47
allocbuf . . . 53
convenience
exception class . . . 73
CORBA . . . 45
freebuf . . . 54
member, Set . . . 48
memory management
sequence, bounded . . . 53
nelems . . . 53
overloaded . . . 60
replace . . . 68
replace() . . . 68
static member
_duplicate . . . 34
_nil . . . 34
is_nil . . . 34
release . . . 34
string_alloc . . . 39
string_dup . . . 39
string_free . . . 39
type() . . . 68
value() . . . 68
virtual
_throw . . . 74

G

Generated operation signatures . . . 80
get method . . . 76

H

Helper class
- _out 80
- Array_forany 66

Helper structs
- boolean 65
- char 65
- ensuring portability 65
- octet 65
- to_object 67

Helper structs, any class 63

I

Identifiers, IDL 6
IDL
- basic data types
  - boolean 60
  - char 60
  - octet 60
- comments 6
- constants
  - mapping 37
  - nested 37
  - top-level 37
- data types 7
- enum
  - mapping 39
- example
  - exception, user-defined 72
  - user-defined exception 72
- exceptions
  - mapping 71
- Identifiers 7
- identifiers 6
- keywords 6
- names, scoped 29
- reserved words 6
- scoped names 29
- sequence 9
- specification
  - mapping 29
- string 10
- string type
  - mapping 39
- struct types
  - mapping 44
- structure 8
- type
  - boolean 38
  - char 38
  - double 38
  - float 38
  - long 38
  - octet 38
  - overloading 39
  - short 38
  - typedef mapping 38
  - unsigned long 38
  - unsigned short 38
- typedef keyword 60

Implementation base header file 25
in arguments 78
in parameter 78, 81, 83, 86, 88, 92
Index Operators
- bounded sequence 54
- sequence,bounded 54

Index operators 54
inout parameter 79, 81, 83, 86, 88, 92
Insertion into Any 60
Insertion operator
- copying form 61

Insufficient type-safe insertion and extraction . . 68
Interface 29
- header file 25

Intializing object references 34
is_nil static function 34

K

Keyword
  IDL 6
  struct 8
kind() 69

L

Leaks, memory 76
Length accessor
  sequence, unbounded 54
  unbounded sequence 54
Local transparency 88

M

malloc 40
Mapping
  Any types 60
  basic data types 38
  bounded sequence 53
  IDL constants 37
  IDL enum 39
  IDL exceptions 71
  IDL string type 39
  parameter passing 77
  parameters 88
  sequences 49
  Structured Types 41
  Typedefs 38
  Types, structured 41
  unbounded sequences 49
Maximum accessor
  bounded sequence 54
  sequence, bounded 54
Member functions, Set 48
Member Get methods 48
Member Set methods 48
Members
  array 48
  discriminant 46
Memory
  leaks 76
  management
    responsibility, ORB 79
  management functions
    sequence, bounded 53
  management policies 80, 81
modes
  parameter passing 77
Module 29
MyException 74

N

Name
  group related data 8
  scoped 29
    building 29
    C++ 29
    legal 30
Namespace 29
  C++ 30
Naming 29
nelems function 53
nocopy parameter 65
Non-distinct types 60
null pointer 39

O

Object implementation dispatch file 26
Object operations

performing . . . 33
Object reference
extraction operator . . . 62
intializing . . . 34
name . . . 33
nil . . . 34
obtaining . . . 45
operations . . . 33
object_var . . . 44
Objects, pseudo . . . 69
objref_ptr& . . . 86
objref_var argument . . . 86
octet . . . 60
helper structs . . . 65
IDL basic data types . . . 60
oneway operation . . . 76
op() operation . . . 83, 84, 85
Operation
best-effort . . . 76
object
performing . . . 33
object reference . . . 33
oneway . . . 76
op() . . . 83, 84, 85
TypeCode kind() . . . 69
Operator
assignment . . . 41, 44, 68, 72
bounded sequence . . . 53
index . . . 54
bounded sequence . . . 54
insertion
copying form . . . 61
overloaded . . . 60
ORB . . . 79
memory management responsibility . . . 79
out parameter . . . 79, 80, 81, 83, 86, 88, 91, 92
flexibility . . . 42
out vsegment . . . 91
Output, debugging . . . 74
output_typecode(_tc_anInterface) . . . 71
output_typecode(_tc_foo) . . . 71
output_typecode(_tc_foo_seq) . . . 71
Overloaded
functions . . . 60
operators . . . 60

P

Parameter
Any . . . 91
fixed-length array type . . . 80
in . . . 78, 81, 83, 86, 88, 92
inout . . . 79, 81, 83, 86, 88, 92
memory
allocation . . . 88
deallocation . . . 88
nocopy . . . 65
out . . . 79, 80, 81, 83, 86, 88, 91, 92
flexibility . . . 42
primitive type . . . 81
Sequence . . . 91
string . . . 84
Parameter mapping . . . 88
Parameter passing
mapping . . . 77
modes . . . 77
signatures . . . 77
Parameter types
array, fixed-length . . . 89
Fixed-length, array . . . 83
Passing consistent arguments . . . 42
Pointer
const . . . 44
null . . . 39
Policies
memory management . . . 81
Predefined TypeCodes . . . 69
Primitive arguments . . . 77
Primitive type parameters . . . 81
Printer . . . 29
Pseudo-objects . . . 69

PrismTech

R

readonly attribute 76
read-write attribute 76
Reallocation 49
unbounded sequence 49
Reference, object
extraction operator 62
initializing 34
name 33
obtaining 45
operations 33
Release flag, boolean .49
Release static function .34
Remote transparency 88
replace function 68
replace() function 68
Reserved words
IDL 6
Return segment
_slice * 84
Return value
flexibility 42
signatures 77

S

Scoped names 29
building 29
C++ 29
legal. 30
sequence 41
bounded
assignment operator. 53
constructors 53
destructor. 53
index operators 54
mapping. 53
maximum accessor .54
memory management functions 53
bounded and unbounded, differences 53
IDL 9
manipulating current length 54
storage deallocation .49
unbounded
constructors 49
length accessor 54
sequence _var Type 55
sequence, unbounded
reallocation 49
Set member functions .48
set method. 76
Signatures
_var type 78
generated operation. .80
parameter passing 77
return value 77
static member function
_duplicate 34
_nil 34
is_nil 34
release 34
Storage allocation
client 85
str_dup 40
String 39
bounded 60
IDL 10
parameters. 84
unbounded 60
insertion and extraction. 65
string_alloc 39
string_alloc() 84
string_dup. 39
string_free. 39
string_free() 84, 85
String_var 40
string_var argument 84
struct. 41, 45
insertion from_string 65
keyword 8
Structure, IDL. 8
Structured Types
mapping 41

T

T * data . . . 53, 54
T_var class. . . . 45
T_var types . . . 42
  example . . . 42
Template Types . . . 9
  sequences . . . 9
  strings. . . . 10
Transparency
  local . . . 88
  remote . . . 88
Type
  Printer . . . 29
Type alias . . . 38
type() function . . . 68
TypeCode kind() operation . . . 69
TypeCode_ptr . . . 68
TypeCodes. . . . 69
  predefined . . . 69
Typedef
  mapping . . . 38
typedef
  IDL keyword . . . 60
Types
  _var . . . 78, 80
  aggregate . . . 88
  aggregate, variable length . . . 87
  Any . . . 60, 69
  Any, mapping . . . 60
  C++ built-in . . . 81
  fixed-length array parameter . . . 83
  fixed-length parameter . . . 79
  IDL struct
    mapping . . . 44
  non-distinct . . . 60
  sequence _var . . . 55
  structured
    mapping . . . 41
  T_var . . . 42
  union . . . 81
    mapping . . . 46
  variable-length. . . . 84, 91
  variable-length parameter . . . 79
  vector_out . . . 92
  vl_struct_out . . . 88
  vsegment_out. . . . 90
Type-safe insertion and extraction . . . 68

U

Unbounded sequence
  constructors . . . 49
  length accessor. . . . 54
  mapping . . . 49
  reallocation . . . 49
Unbounded string . . . 60
  insertion and extraction . . . 65
union . . . 41
  constructor . . . 47
  fixed-length . . . 81
  types. . . . 81
    mapping . . . 46
Unsafe constructor . . . 68
User-defined exception
  IDL example . . . 72
using statement . . . 30

V

value() function . . . 68
Values, return
  flexibility . . . 42
Variable
  maps to. . . . 45
Variable length aggregate types . . . 87
Variable-length parameter types . . . 79
Variable-length types . . . 84, 91
vector*& . . . 92
vector_out type . . . 92
Violations, access . . . 76
Virtual functions

_throw 74
vl_struct*& 88
vl_struct_out type 88
vsegment_out type 90
vsegment_slice*& 90