Orbix 6.3.7

CORBA Code Generation Toolkit Guide

Micro Focus
The Lawn
22-30 Old Bath Road
Newbury, Berkshire RG14 1QN
UK

http://www.microfocus.com

2014-06-12

Contents

Part IV Java Genies Library Reference

Preface

The Orbix Code Generation Toolkit is a flexible development tool that can help you become more productive right away by automating many repetitive coding tasks.

The toolkit contains an Interface Definition Language (IDL) parser, `idlgen`, and ready-made applications, or *genies*, that can generate Java or C++ code from CORBA IDL files automatically. The toolkit also contains libraries of useful commands that let you develop your own genies.

Audience

This guide is aimed at developers of Orbix applications. Before reading this guide, you should be familiar with the Object Management Group IDL, the C++ or Java language, and the Tcl scripting language.

Organization of this guide

This guide is divided into four parts and appendices:

Part I Using the Toolkit

Provides an overview of the Orbix code generation toolkit and describes its constituent components. This part describes how to run the demonstration genies bundled with the toolkit.

Part II Developing Genies

Offers an in-depth look at the Orbix code generation toolkit and shows how to develop genies that are tailored to your own needs.

Part III C++ Genies Library Reference

Provides a comprehensive reference to the library commands that you can use in your genies, to produce C++ code from CORBA IDL files.

Part IV Java Genies Library Reference

Provides a comprehensive reference to the library commands that you can use in your genies, to produce Java code from CORBA IDL files.

Appendices

Provide reference material on configuration options, command libraries, the IDL parser, and configuration file grammar.

Typographical conventions

This guide uses the following typographical conventions:

`Constant width`	Constant width (courier font) in normal text represents portions of code and literal names of items such as classes, functions, variables, and data structures. For example, text might refer to the `CORBA::Object` class.
	Constant width paragraphs represent code examples or information a system displays on the screen. For example:
	`#include <stdio.h>`
Italic	Italic words in normal text represent *emphasis* and *new terms*.
	Italic words or characters in code and commands represent variable values you must supply, such as arguments to commands or path names for your particular system. For example:
	`% cd /users/`*your_name*
	Note: Some command examples may use angle brackets to represent variable values you must supply. This is an older convention that is replaced with *italic* words or characters.

Keying conventions

This guide may use the following keying conventions:

No prompt	When a command's format is the same for multiple platforms, a prompt is not used.
`%`	A percent sign represents the UNIX command shell prompt for a command that does not require root privileges.

#	A number sign represents the UNIX command shell prompt for a command that requires root privileges.
>	The notation > represents the DOS or Windows command prompt.
... . . .	Horizontal or vertical ellipses in format and syntax descriptions indicate that material has been eliminated to simplify a discussion.
[ ]	Brackets enclose optional items in format and syntax descriptions.
{ }	Braces enclose a list from which you must choose an item in format and syntax descriptions.
|	A vertical bar separates items in a list of choices enclosed in { } (braces) in format and syntax descriptions.

Contacting Micro Focus

Our Web site gives up-to-date details of contact numbers and addresses.

Further Information and Product Support

Additional technical information or advice is available from several sources.

The product support pages contain a considerable amount of additional information, such as:

- The WebSync service, where you can download fixes and documentation updates.
- The Knowledge Base, a large collection of product tips and workarounds.
- Examples and Utilities, including demos and additional product documentation.

To connect, enter http://www.microfocus.com in your browser to go to the Micro Focus home page.

Note:
Some information may be available only to customers who have maintenance agreements.

If you obtained this product directly from Micro Focus, contact us as described on the Micro Focus Web site, http://www.microfocus.com. If you obtained the product from another source, such as an authorized distributor, contact them for help first. If they are unable to help, contact us.

Information We Need

However you contact us, please try to include the information below, if you have it. The more information you can give, the better Micro Focus SupportLine can help you. But if you don't know all the answers, or you think some are irrelevant to your problem, please give whatever information you have.

- The name and version number of all products that you think might be causing a problem.
- Your computer make and model.
- Your operating system version number and details of any networking software you are using.
- The amount of memory in your computer.
- The relevant page reference or section in the documentation.
- Your serial number. To find out these numbers, look in the subject line and body of your Electronic Product Delivery Notice email that you received from Micro Focus.

Contact information

Our Web site gives up-to-date details of contact numbers and addresses.

Additional technical information or advice is available from several sources.

The product support pages contain considerable additional information, including the WebSync service, where you can download fixes and documentation updates. To connect, enter http://www.microfocus.com in your browser to go to the Micro Focus home page.

If you are a Micro Focus SupportLine customer, please see your SupportLine Handbook for contact information. You can download it from our Web site or order it in printed form from your sales representative. Support from Micro Focus may be available only to customers who have maintenance agreements.

You may want to check these URLs in particular:

- http://www.microfocus.com/products/corba/orbix/orbix-6.aspx (trial software download and Micro Focus Community files)
- https://supportline.microfocus.com/productdoc.aspx. (documentation updates and PDFs)

To subscribe to Micro Focus electronic newsletters, use the online form at:

http://www.microfocus.com/Resources/Newsletters/infocus/newsletter-subscription.asp

Part I

Using the Toolkit

In this part

This part contains the following chapters:

Overview of the Code Generation Toolkit

The Orbix Code Generation Toolkit is a powerful development tool that can automatically generate code from IDL files.

The code generation toolkit offers ready-to-run genies that generate code from IDL files. You can use this code immediately in your development project. Used in this way, the toolkit can dramatically reduce the amount of time for development.

You can also use the code generation toolkit to write your own code generation scripts, or genies. For example, you can write genies to generate C++ or Java code from an IDL file, or to translate an IDL file into another format, such as HTML, RTF, or LaTeX.

IDL Compiler Architecture

Components of an IDL compiler

As shown in Figure 1, an IDL compiler typically contains three sub-components. A parser processes an input IDL file and constructs an in-memory representation, or parse tree. The parse tree can be queried to obtain arbitrary details about IDL declarations. A back end code generator then traverses the parse tree and generates C++ or Java stub code.

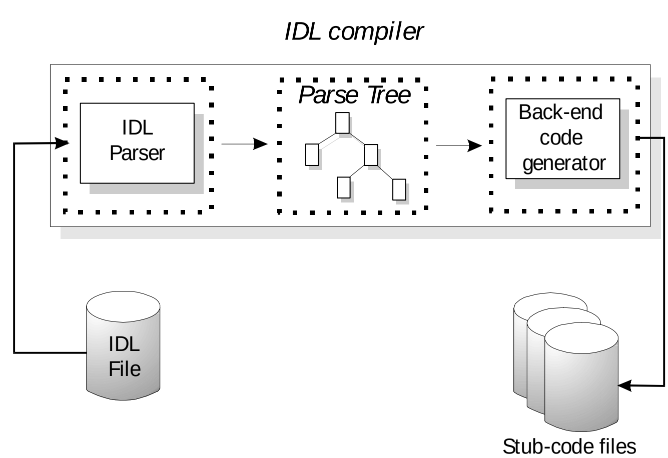

Figure 1: *Standard IDL Compiler Components*

Code Generation Toolkit Architecture

The idlgen executable

At the heart of the code generation toolkit is the `idlgen` executable. It uses an IDL parser and parse tree, but instead of a back end that generates stub code, the back end is a Tcl interpreter. The core Tcl interpreter provides the normal features of a language, such as flow-control statements, variables and procedures.

As shown in Figure 2, the Tcl interpreter inside `idlgen` is extended to manipulate the IDL parser and parse tree with Tcl commands. This lets you implement a customized back end, or *genie*, as a Tcl script.

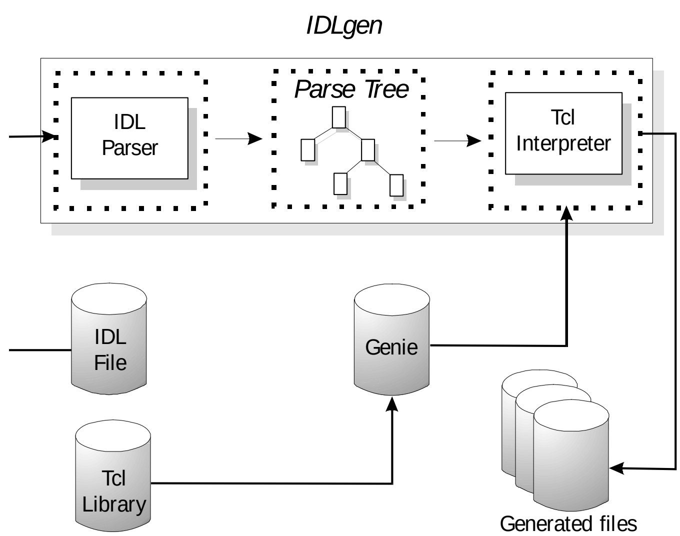

Figure 2: *Code Generation Toolkit Components*

Code Generation Toolkit components

The code generation toolkit consists of three components:

- The `idlgen` executable, which is the engine at the heart of the code generation toolkit.
- A number of pre-written genies that generate useful starting point code for Orbix applications, and perform other tasks. You can use these bundled genies straight away, without any knowledge of Tcl. For information on how to run the pre-written genies, refer to the *Orbix 2000 Programmer's Guide*.

  The Tcl source code for the pre-written genies can be used as a basis for writing your own genies.
- Tcl libraries that you can use to write your own genies. For example, a library is provided that maps IDL constructs to their C++ equivalents.

Running Demonstration Genies

Some ready-to-run genies are bundled with the code generation toolkit. This chapter describes the demo genies. For information about using the C++ and Java genies, see the Orbix CORBA Programmer's Guide for Java and for C++.

Orbix Genies

Genie categories

Bundled genies can be grouped into the following categories:

- Demo genies
  - `stats.tcl` provides statistical analysis of an IDL file's content.
  - `idl2html.tcl` converts IDL files into HTML files.
- Orbix C++ genies
  - `cpp_poa_genie.tcl` generates code for Orbix from an IDL file.
  - `cpp_poa_op.tcl` generates code for new operations from an IDL interface.
- Orbix Java genies
  - `java_poa_genie.tcl` generates code for Orbix from an IDL file.
  - `java_random.tcl` creates a number of functions that generate random values for all the types present in an IDL file.
  - `java_print.tcl` creates a number of functions that can display all the data types present in an IDL file.

General Genie Syntax

In general, you can run a genie through the `idlgen` interpreter like this:

```
idlgen genie-name [options]... target-idl-file[...]
```

Locating Genies

The `idlgen` executable locates the specified genie file by searching a list of directories. The list of directories comprises a list of default directories and the list of directories specified by `idlgen.genie_search_path` in the `idlgen.cfg` configuration file. See "General Configuration Options" on page 263.

You can alter the `genie_search_path` configuration setting to include other directories in the search path. For example, if you write your own genies, you can place them in a separate directory and add this directory to `idlgen.genie_search_path`.

Listing Genies

The `idlgen` executable provides a command-line option that lists all genies that are on the search path. For example:

```
idlgen -list

available genies are...

cpp_poa_equal.tcl    cpp_poa_random.tcl   list.tcl
cpp_poa_genie.tcl    idl2html.tcl         stats.tcl
cpp_poa_op.tcl       java_poa_genie.tcl
cpp_poa_print.tcl    java_print.tcl
```

You can also qualify the `-list` option with a filter argument. The filter argument displays only the genies whose names contain the specified text. For example, the following command shows all genies whose names contain the text `cpp`:

```
idlgen -list cpp

matching genies are...

cpp_poa_equal.tcl   cpp_poa_op.tcl      cpp_poa_random.tcl
cpp_poa_genie.tcl   cpp_poa_print.tcl
```

The stats.tcl Genie

The `stats.tcl` genie provides a number of statistics, based on an IDL file's content. It prints out a summary of how many times each IDL construct (such as `interface`, `operation`, `attribute`, `struct`, `union`, `module`) appears in the specified IDL files.

Example

The following is an example of running the `stats.tcl` genie:

```
idlgen stats.tcl bank.idl

statistics for 'bank.idl'
--------------------------------------------------
0         modules
5         interfaces
7         operations (1.4 per interface)
9         parameters (1.28571428571 per operation)
3         attributes (0.6 per interface)
0         sequence typedefs
0         array typedefs
0         typedef (not including sequences or arrays)
0         struct
0         fields inside structs (0 per struct)
0         unions
0         branches inside unions (0 per union)
1         exceptions
1         fields inside exceptions (1.0 per exception)
0         enum types
0         const declarations
5         types in total
```

Processing

The `stats.tcl` genie only processes the constructs it finds in the IDL file. It ignores `#include` statements. Supply the command-line option -include to recursively process all included IDL files.

For example, the IDL file `bank.idl` includes the `account.idl` IDL file. You can get statistics from both `account.idl` and `bank.idl` files using the following command:

```
idlgen stats.tcl -include bank.idl
```

This genie serve two purposes:

- Provides objective information that can help estimate the time required to implement some task, based on the IDL.
- Shows how to write genies that process IDL files.

The idl2html.tcl Genie

The `idl2html.tcl` genie converts an IDL file to an equivalent HTML file.

Example

Given the following IDL:

```
// IDL
interface bank {
    exception reject {
        string reason;
    };
    account newAccount(in string name)
        raises(reject);
    void deleteAccount(in account a)
        raises(reject);
};
```

You can convert this IDL file to HTML as follows:

```
idlgen idl2html.tcl bank.idl

idlgen: creating bank.html
```

This yields the following HTML output:

```
// HTML
interface bank {
            exception reject {
                string reason;
            };
            account newAccount(
                        in string name)
                        raises (bank::reject);
            void deleteAccount(
                        in account a)
                        raises (bank::reject);
}; // interface bank
```

Hypertext links resolve to the definition of the specified type. For example, clicking on `account` invokes the `account` interface definition.

You can set `default.html.file_ext` in the `idlgen.cfg` configuration file in order to specify the file extension preferred by your web browser—typically, `html` or `htm`.

Supplying Command-Line Options

Specifying command-line options

Bundled genies share several command-line options. To list all available options, supply the `-h` (help) option. For example:

```
idlgen idl2html.tcl -h

usage: idlgen idl2html.tcl [options] [file.idl]+
options are:
    -I<directory>       Passed to preprocessor
    -D<name>[=value]    Passed to preprocessor
    -h                  Prints this help message
    -v                  verbose mode
    -s                  silent mode
```

Preprocessor Options

Before the `idlgen` interpreter parses an IDL file, it sends the IDL file through an IDL preprocessor. The IDL preprocessor is similar to the well-known C, and C++ preprocessors.

Two command-line options let you pass information to the IDL preprocessor:

`-I` Specifies the include path for the preprocessor. For example:

```
idlgen idl2html.tcl -I/inc -I../std/inc bank.idl
```

`-D` Defines additional preprocessor symbols. For example:

```
idlgen idl2html.tcl -I/inc -DDEBUG bank.idl
```

Arguments that contain white space must be enclosed by quotation marks. For example:

```
idlgen idl2html.tcl -I"/Program Files" bank.idl
```

Verbosity Options

Two command-line options determine whether or not the genies run in verbose or silent mode.

`-v` Selects verbose mode. For example:

```
idlgen idl2html.tcl -v bank.idl
idlgen: creating bank.htm
```

`-s` Selects silent mode. For example:

```
idlgen idl2html.tcl -s bank.idl
```

The default setting is determined by the `default.all.want_diagnostics` value in the `idlgen.cfg` configuration file. If set to `yes`, Orbix defaults to verbose mode. If set to `no`, Orbix defaults to silent mode.

Part II

Developing Genies

In this part

This part contains the following chapters:

Basic Genie Commands

This chapter discusses some basic genie commands that are used to include other genie scripts and produce output text.

As described in "Code Generation Toolkit Architecture" on page 8, the `idlgen` interpreter provides a set of built-in commands that extend Tcl. Genies are Tcl scripts that use these extensions in parallel with the basic Tcl commands and features. These extensions allow you to parse IDL files easily and generate corresponding code to whatever specification you require.

To develop your own genies, you must be familiar with two languages: IDL and Tcl. You must also be familiar with the required output language and with the IDL mapping specification for that language.

Hello World Example

The idlgen interpreter

The `idlgen` interpreter processes Tcl scripts in the same way as any other Tcl interpreter. Tcl script files are fed into it and `idlgen` outputs the results to the screen or to a file.

The `idlgen` interpreter can only process Tcl commands stored in a script file. It does not have an interactive mode.

> **Note:** Although `idlgen` is a Tcl interpreter, the common Tcl extensions, such as Tk or Expect, are not built in. You cannot use `idlgen` to execute a Tk or Expect script.

Hello World Tcl Script

Consider this simple Tcl script:

```
# Tcl
puts "Hello, World"
```

Running this through the `idlgen` interpreter gives the following result:

```
idlgen hello.tcl

Hello, World
```

Adding Command Line Arguments

The `idlgen` interpreter adheres to the Tcl conventions for command-line argument support. This is demonstrated in the following script:

```
# Tcl
puts "argv0 is $argv0"
puts "argc is $argc"
foreach item $argv {
    puts "Hello, $item"
}
```

Running this through `idlgen` yields the following results:

```
idlgen arguments.tcl Fred Joe Mary

argv0 is arguments.tcl
argc is 3
Hello, Fred
Hello, Joe
Hello, Mary
```

Including Other Tcl Files

The `idlgen` interpreter provides two alternative commands for including other Tcl files into your genie script:

- The `source` command.
- The `smart_source` command.

The source Command

Standard Tcl has a command called `source`. The `source` command is similar to the `#include` compiler directive used in C++ and allows a Tcl script to use commands that are defined (and implemented) in other Tcl scripts. For example, to use the commands defined in the Tcl script `foobar.tcl` you can use the `source` command as follows (the C++ equivalent is given, for comparison):

```
# Tcl
source foobar.tcl
```

```
// C++
#include "foobar.h"
```

The `source` command has one limitation compared with its C++ equivalent: it has no search path for locating files. This requires you to specify full directory paths for other Tcl scripts, if the scripts are not in the same directory.

The smart_source Command

To locate an included file, using a search path, `idlgen` provides an enhanced version of the `source` command, called `smart_source`:

```
# Tcl
smart_source "myfunction.tcl"
myfunction "I can use you now"
```

Note: The search path is given in the `idlgen.genie_search_path` item in the `idlgen.cfg` configuration file. For more details, see "General Configuration Options" on page 263.

The `smart_source` command provides the following advantages over the simpler `source` command:

- It locates the specified Tcl file through a search path. This search path is specified in the `idlgen` configuration file and is the same one used by `idlgen` when it looks for genies.
- It has a built-in preprocessor for bilingual files. Bilingual files are described in the section "Embedding Text Using Bilingual Files" on page 22.
- It has a `pragma once` directive. This prevents repeated sourcing of library files and aids in overriding Tcl commands. This is described in "Re-Implementing Tcl Commands" on page 135.

Writing to a File

Alternatives to puts

Tcl scripts normally use the `puts` command for writing output. The default behavior of the `puts` command is to:

- Print to standard output.
- Print a new line after its string argument.

Both behaviors can be overridden. For example, if the output is to go to a file and no new line character is to be placed at the end of the output, you can use the `puts` command as follows:

```
# Tcl
puts -nonewline $some_file_id "Hello, world"
```

This syntax is too verbose to be useful. Genies regularly need to create output in the form of a text file. The code generation toolkit provides utility functions to create and write files that provide a more concise syntax for writing text to a file.

The utility functions are located in the `std/output.tcl` script. To use them you must use the `smart_source` command.

Example

The following example uses these utility commands:

```
# Tcl
smart_source "std/output.tcl"
set class_name "testClass"
set base_name "baseClass"

open_output_file "example.h"
output "class $class_name : public virtual "
output "$base_name\n"
output "{\n"
output "     public:\n"
output "        ${class_name}() {\n"
output "          cout << \"$class_name CTOR\";\n"
output "         }\n"
output "};\n"
close_output_file
```

When this script is run through the `idlgen` interpreter, it writes a file, `example.h`, in the current directory:

```
idlgen codegen.tcl

idlgen: creating example.h
The contents of this file are:
class testClass : public virtual baseClass
{
    public:
        testClass() {
            cout << "testClass CTOR";
        }
};
```

Note: Braces are placed around the `class_name` variable, so the Tcl interpreter does not assume `$class_name()` is an array.

Commands for creating a file

Table 1 shows the three commands that are used to create a file.

Table 1: *Creating a File*

Command	Result
`open_output_file` *`filename`*	Opens the specified file for writing. If the file does not exist, it is created. If the file exists, it is overwritten.
`output` *`string`*	Appends the specified string to the currently open file.
`close_output_file`	Closes the currently open file.

Embedding Text in Your Application

Although the `output` command is concise, the example in "Writing to a File" on page 19 is not easy to read. The number of output commands tends to obscure the structure and layout of the code being generated. It is better to place code in the Tcl script in a way that allows the layout and structure to be retained, while allowing the embedding of commands and variables.

The `idlgen` interpreter allows a large block of text to be quoted by:

- Embedding text in braces.
- Embedding text in quotation marks.
- Embedding text using bilingual files.

Embedding Text in Braces

Using braces allows the text to be placed over several lines:

```
# Tcl
smart_source "std/output.tcl"
set class_name "testClass"
set base_name "baseClass"

open_output_file "example.h"
output {
class $class_name : public virtual $base_name
{
    public:
        ${class_name}() {
            cout << "$class_name CTOR";
        }
};}
Running this script through idlgen results in the following
   example.h file:
class $class_name : public virtual $base_name
{
    public:
        ${class_name}() {
            cout << "$class_name CTOR";
        }
};
```

This code is easier to read than the code extract shown in "Writing to a File" on page 19. It does not, however, allow you to substitute variables.

Embedding Text in Quotation Marks

The second approach is to provide a large chunk of text to the `output` command using quotation marks:

```
# Tcl
smart_source "std/output.tcl"
set class_name "testClass"
set base_name "baseClass"
```

```
open_output_file "example.h"
output "
class $class_name : public virtual $base_name
{
    public:
        ${class_name}() {
            cout << \"$class_name CTOR\";
        }
};"
close_output_file
Running this script through the idlgen interpreter results in
   the following example.h file:
class testClass : public virtual baseClass
{
    public:
        testClass() {
            cout << "testClass CTOR";
        }
};
```

This is much better than using braces because the variables are substituted correctly. However, a disadvantage of using quotation marks is that you must remember to prefix embedded quotation marks with an escape character:

```
cout << \"$class_name CTOR\";
```

Embedding Text Using Bilingual Files

A bilingual file contains a mixture of two languages: Tcl and plain text. A preprocessor in the `idlgen` interpreter translates the plain text into `output` commands.

In the following example, plain text areas in bilingual scripts are marked using *escape sequences*. The escape sequences are shown in Table 2.

```
# Tcl
smart_source "std/output.tcl"
open_output_file "example.h"
set class_name "testClass"
set base_name "baseClass"

[***
class @$class_name@ : public virtual @$base_name@
{
    public:
        @$class_name@() {
            cout << "@$class_name@ CTOR";
        }
}
***]
close_output_file
```

Table 2: *Bilingual File Escape Sequences*

Escape Sequence	Use
`[***`	To start a block of plain text.
`***]`	To end a block of plain text.
`@$variable@`	To escape out of a block of plain text to a variable.
`@[nested command]@`	To escape out of a block of plain text to a nested command.

Compare this with the example in "Embedding Text in Braces" on page 21 that uses braces; the bilingual version is easier to read and substitutes the variables correctly.

It is much easier to write genies using bilingual files, especially if you have a syntax-highlighting text editor that uses different fonts or colors to distinguish the embedded text blocks of a bilingual file from the surrounding Tcl commands. Bold font is used throughout this guide to help you distinguish text blocks.

Note: Bilingual files normally have the extension `.bi`. This is not required, but is the convention used by all the genies bundled with the code generation toolkit.

Syntax Notes

- To print the `@` symbol inside a textual block use the following syntax:

```
# Tcl
set at "@"
[***...
support@$at@iona.com
...***]
```

- Similarly, if you want to print `[***` or `***]` in a file, print it in two parts so it is not treated as an escape sequence.
- The bilingual file preprocessor does not understand standard comment characters, such as `#`. For example, you cannot do the following:

```
# Tcl
#[***
#some text here
#***]
```

Instead, use an `if` statement to disable the plain text block:

```
# Tcl
if {0} {
[***
some text here
***]
}
```

Debugging and the bi2tcl Utility

Debugging a bilingual file can be awkward. The `idlgen` interpreter reports a line number where the problem exists but because the bilingual file has been altered by the preprocessor, this line number may not correspond to where the problem actually lies.

The `bi2tcl` utility helps you avoid this problem by replacing embedded text in a bilingual file with `output` commands, and generating a new but semantically equivalent script. This can be useful for debugging purposes because it is easier to understand runtime interpreter error messages with correct line numbers.

Example

If you run the bilingual example from "Embedding Text Using Bilingual Files" on page 22 through `bi2tcl`, a new file is created with `output` commands rather than the plain text area:

```
bi2tcl codegen.bi codegen.tcl
```

The contents of the `codegen.tcl` file are:

```
# Tcl
smart_source "std/output.tcl"
open_output_file "example.h"
set class_name "testClass"
set base_name "baseClass"
output "class ";
output $class_name;
output " : public virtual ";
output $base_name;
output "\n";
output "\{\n";
output "  public:\n";
output "     ";
output $class_name;
output "() \{\n";
output "       cout << \"";
output $class_name;
output " CTOR\";\n";
output "     \}\n";
output "\}\n";
close_output_file
```

The corresponding `.bi` and `.tcl` files are different in size. If a problem occurs inside the plain text section of the script, the interpreter gives a line number that, in certain cases, does not correspond to the original bilingual script.

Processing an IDL File

The IDL parser is a core component of the code generation toolkit. It allows IDL files to be processed into a parse tree and used by the Tcl application.

This chapter describes how the `idlgen` interpreter parses an IDL file and stores the results as a tree. This chapter details the structure of the tree and its nodes, and demonstrates how to build a sample IDL search genie, `idlgrep.tcl`. Appendix 1 on page 279 provides a reference to the commands discussed in this chapter.

IDL Files and idlgen

Parsing IDL

The IDL parsing extension provided by the `idlgen` interpreter gives the programmer a rich API that provides the mechanism to parse and process an IDL file with ease. When an IDL file is parsed, the output is stored in an internal format called a *parse tree*. The contents of this parse tree can be manipulated by a genie.

Parsing Example

Consider the following IDL, from `finance.idl`:

```
// IDL
interface Account {
    readonly attribute long accountNumber;
    readonly attribute float balance;
    void makeDeposit(in float amount);
};

interface Bank {
    Account newAccount();
};
```

Processing the contents of this IDL file involves two steps:

Step	Action
1	Parsing the IDL file.
2	Traversing the parse tree.

Parsing the IDL File

The built-in `idlgen` command, `idlgen_parse_idl_file`, provides the functionality for parsing an IDL file. It takes two parameters:

- The name of the IDL file.

- (optional) A list of preprocessor directives that are passed to the IDL preprocessor.

For example, you can use this command to process the `finance.idl` IDL file.

```
# Tcl
if {![idlgen_parse_idl_file "finance.idl"]}{
    exit 1
}
...# Continue with the rest of the application
```

If the IDL file is successfully parsed, the genie then has an internal representation of the IDL file ready for examination.

> **Note:** Warning or error messages that are generated during parsing are printed to standard error. If parsing fails, `idlgen_parse_idl_file` returns `0` (false).

Traversing the Parse Tree

After an IDL file is processed successfully by the parsing command, the root of the parse tree is placed into the global array variable `$idlgen(root)`.

The parse tree is a representation of the IDL, where each node in the tree represents an IDL construct. For example, parsing the `finance.idl` file forms the tree shown in Figure 3.

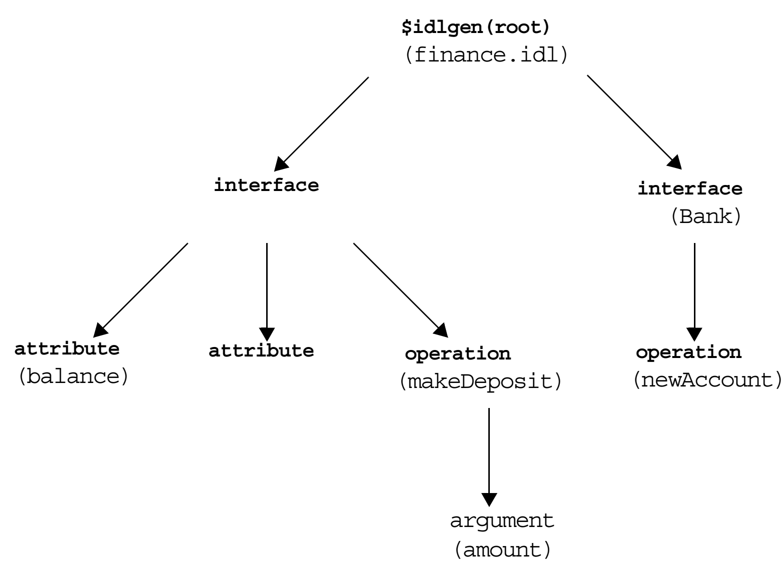

Figure 3: *The Finance IDL File's Parse Tree*

A genie can invoke commands on a node to obtain information about the corresponding IDL construct or to traverse to other parts of the tree related to the node on which the command was performed.

Example

Assume that you have traversed the parse tree and have located the node that represents the `balance` attribute. You can determine the information associated with this node by invoking commands on it:

```
# Tcl
set type_node [$balance_node type]
puts [$type_node l_name]

> float
```

This example uses the `type` node command, which returns a node that represents the attribute type. The `type` command is specific to `attribute` nodes. The `l_name` node command, which obtains the local name, is bcommon to all nodes.

Note: The parse tree incorporates the contents of all included IDL files, as well as the contents of the parsed IDL file.

You can use the `is_in_main_file` node command to find out whether a construct came from the parsed IDL file (as opposed to one of the included IDL files):

```
# Tcl
... # Assume interface_node has been initialised
set name [$interface_node l_name]
if {![$interface_node is_in_main_file]} {
    puts "$name is in the main file"
} else {
    puts "$name is not in the main file"
}
```

The Tcl script generates the following output:

```
Account is in the main file
```

Parse Tree Nodes

Node types

When creating the parse tree, `idlgen` uses a different type of node for each kind of IDL construct. For example, an interface node is created to represent an IDL interface, an operation node is created to represent an IDL operation and so on. Each node type provides a number of node commands.

Common node commands

Some node commands, such as the local name of the node, are common to all node types:

```
# Tcl
puts [$operation_node l_name]
```

The Tcl script generates the following output:

```
newAccount
```

Type-specific node commands

Some commands are specific to a particular type of node. For example, a node that represents an operation can be asked what the return type of that operation is:

```
# Tcl
set return_type_node [$operation_node return_type]
puts [$return_type_node l_name]
```

The Tcl script generates the following output:

```
Account
```

The different types of node are arranged into an inheritance hierarchy, as shown in Figure 4.

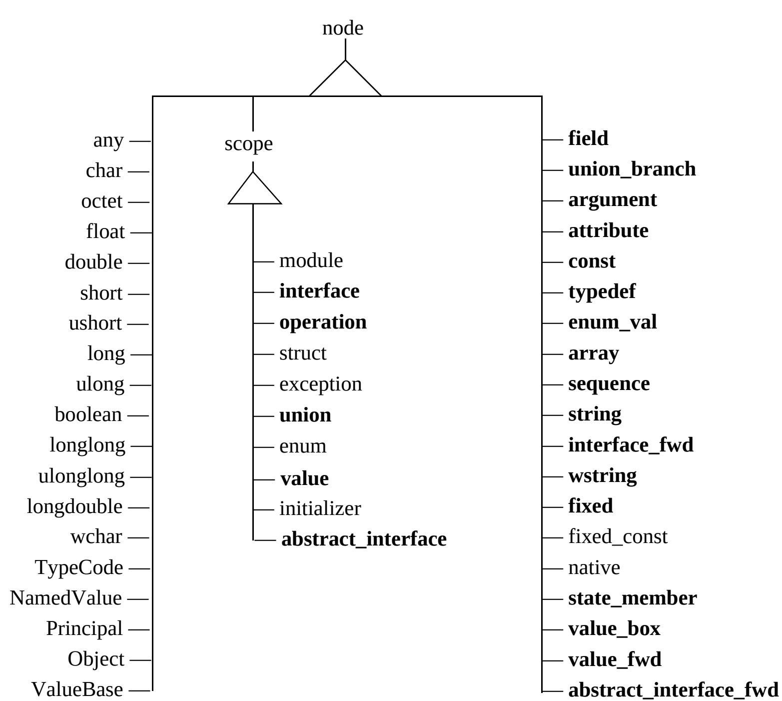

Figure 4: *Inheritance Hierarchy for Node Types*

New commands

Types in boldface define new commands. For example, the `field` node type inherits from the `node` node type, and defines some new commands, whereas the `char` node type also inherits from the `node` node type, but does not define any additional commands.

Abstract nodes

The following two abstract node types do not represent any IDL constructs, but encapsulate the common features of certain types of node:

- node
- scope

The node Abstract Node

Every node type inherits `node` commands. These commands can be used to find out about the common features of any construct.

> **Note:** Tcl is not an object-oriented programming language, so these `node` objects and their corresponding commands are described with a pseudo-code notation.

Definition of node

Here is a pseudo-code definition of the `node` abstract node:

```
class node {
    string              node_type()
    string              l_name()
    string              s_name()
    list<string>        s_name_list()
    string              file()
    integer             line()
    boolean             is_in_main_file()
}
```

> **Note:** This is a partial definition of the `node` abstract node. For a complete definition, see "IDL Parse Tree Nodes" on page 279.

Node commands

Two commonly used commands provided by the `node` abstract node are:

- `l_name()`, which returns the name of the node.
- `file()`, which returns the IDL file in which this node appears.

All node types inherit directly or indirectly from this abstract node. For example, the argument node, which represents an operation argument, inherits from node. It supplies additional commands that allow the programmer to determine the argument type and the direction modifier (in, inout, or out).

Here is a pseudo-code definition of the argument node type:

```
class argument : node {
    node                type()
    string              direction()
}
```

Assume that, in a genie, you have obtained a handle to the node that represents the argument highlighted in this parsed IDL file:

```
// IDL
interface Account {
    readonly attribute long accountNumber;
    readonly attribute float balance;

    void makeDeposit(in float amount);
};
```

The handle to the amount argument is placed in a variable called argument_node. To obtain information about the argument, the Tcl script can use any of the commands provided by the abstract node class or by the argument class:

```
# Tcl
... # Some code to locate argument_node
puts "Node type is '[$argument_node node_type]'"
puts "Local name is '[$argument_node l_name]'"
puts "Scoped name is '[$argument_node s_name]'"
puts "File is '[$argument_node file]'"
puts "Appears on line '[$argument_node line]'"
puts "Direction is '[$argument_node direction]'"
```

Run the idlgen interpreter from the command line:

```
idlgen arguments.tcl

Node type is 'argument'
Local name is 'amount'
Scoped name is 'Account::makeDeposit::amount'
File is 'finance.idl'
Appears on line '5'
Direction is 'in'
```

The scope Abstract Node

The other abstract node is the scope node. The scope node represents constructs that have *scoping behavior*—constructs that can contain nested constructs. The scope node provides the commands for traversing the parse tree.

For example, a `module` construct can have `interface` constructs inside it. A node that represented a `module` would therefore inherit from `scope` rather than `node`.

Note: The `scope` node inherits from the `node` abstract node.

Here is a pseudo-code definition of the `scope` abstract node:

```
class scope : node {
    node              lookup(string name)
    list<node>        contents(
                          list<string> constructs_wanted,
                          function filter_func=true_func)
    list<node>        rcontents(
                          list<string> constructs_wanted,
                          list<string> recurse_into,
                          function filter_func=true_func)
}
```

The `interface` and `module` constructs are concrete examples of node types that inherit from the `scope` node. An `interface` node type inherits from `scope` and extends the functionality of the `scope` node by providing a number of additional commands. These additional commands allow you to determine which interfaces can be inherited. They also permit you to search for and determine the ancestors of an interface.

The pseudo-code definition of the `interface` node is:

```
class interface : scope {
    list<node>                          inherits()
    list<node>                          ancestors()
    list<node>                          acontents()
}
```

To locate a node, a search command can be performed on an appropriate scoping node (in this case the root of the parse tree is used, as this is the primary scoping node that most searches originate from):

```
# Tcl
if {![idlgen_parse_idl_file "finance.idl"]} {
    exit 1
}
set node [$idlgen(root) lookup "Account::balance"]
puts [$node l_name]
puts [$node s_name]
```

Run the `idlgen` interpreter from the command line:

```
idlgen lookup.tcl

balance
Account::balance
```

The job of the `lookup` command is to locate a node by its fully or locally scoped lexical name.

Locating Nodes with contents and rcontents

There are two more scope commands that can be used to locate nodes in the parse tree:

- The contents command.
- The rcontents command.

Both of these commands can be used to search for nodes that are contained within a scoping node.

For example, to get to a list of the interface nodes from the root of the parse tree, you can use the contents command:

```
# Tcl
if {![idlgen_parse_idl_file "finance.idl"]} {
    exit
}
set want {interface}
set node_list [$idlgen(root) contents $want]
foreach node $node_list {
    puts [$node l_name]
}
```

Run the idlgen interpreter from the command line:

```
idlgen contents.tcl

Account
Bank
```

This command allows you to specify what type of constructs you want to search for, but it only searches for constructs that are directly under the given node (in this case the root of the parse tree).

The rcontents command extends the search so that it recurses into other scoping constructs.

For example:

```
# Tcl
if {![idlgen_parse_idl_file "finance.idl"]} {
    exit
}
set want {interface operation}
set recurse_into {interface}

set node_list [$idlgen(root) rcontents $want
   $recurse_into]
foreach node $node_list {
    puts "[$node node_type]: [$node s_name]"
}
```

Run the `idlgen` interpreter from the command line:

```
idlgen contents.tcl

interface: Account
operation: Account::makeDeposit
interface: Bank
operation: Bank::findAccount
operation: Bank::newAccount
```

This small section of Tcl code gives the scoped names of all the `interface` nodes that appear in the root scope and the scoped names of all the `operation` nodes that appear in any `interfaces`.

The all Pseudo-Node

For both `contents` and `rcontents` you can use a special pseudo-node name to represent all of the constructs you want to look for or recurse into. This name is `all` and you use it when you want to list all constructs:

```
# Tcl
set everynode_in_tree [rcontents all all]
```

It is now very easy to write a genie that can visit (almost) every node in the parse tree:

```
# Tcl
if {![idlgen_parse_idl_file "finance.idl"]} {
exit
}
set node_list [$idlgen(root) rcontents all all]
foreach node $node_list {
puts "[$node node_type]: [$node s_name]"
}
```

Try running the above script on an IDL file and see how the parse tree is traversed and what node types exist. Remember to change the argument to the parsing command to reflect the particular IDL file you want to traverse.

Note: This example genie visits most of the nodes in the parse tree. However, it will not visit any hidden nodes. See "Visiting Hidden Nodes" on page 36 for a discussion on how to access hidden nodes in the parse tree.

Nodes Representing Built-In IDL Types

Nodes that represent the built-in IDL types can be accessed with the `lookup` command defined on the `scope` node type. For example:

```
# Tcl
...
foreach type_name {string "unsigned long" char} {
    set node [$idlgen(root) lookup $type_name]
    puts "Visiting the '[$node s_name]' node"
}
```

Run the `idlgen` interpreter from the command line:

```
idlgen basic_types.tcl

Visiting the 'string' node
Visiting the 'unsigned long' node
Visiting the 'char' node
```

For convenience, the `idlgen` interpreter provides a utility command called `idlgen_list_builtin_types` that returns a list of all nodes representing the built-in types. You can use it as follows:

```
# Tcl
foreach node [idlgen_list_builtin_types] {
    puts "Visiting the [$node s_name] node"
}
```

It is rare for a script to process built-in types explicitly. However, nodes representing built-in types are accessed during normal traversal of the parse tree. For example, consider the following operation signature:

```
// IDL
interface Account {
    ...
    void makeDeposit(in float amount);
};
```

If a script traverses the parse tree and encounters the node for the amount parameter, then accessing the parameter's type returns the node representing the built-in type `float`:

```
#Tcl
... # Assume param_node has been initialized
set param_type [$param_node type]
puts "Parameter type is [$param_type s_name]"
```

Run the `idlgen` interpreter from the command line:

```
idlgen param_type.tcl

Parameter type is float
```

Typedefs and Anonymous Types

Consider the following IDL declarations:

```
// IDL
typedef sequence<long> longSeq;
typedef long longArray[10][20];
```

This segment of IDL defines a sequence called `longSeq` and an `array` called `longArray`.

The following is a pseudo-code definition of the `typedef` class:

```
class typedef : node {
    node base_type()
};
```

The `base_type` command returns the node that represents the `typedef`'s underlying type. In the case of:

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

The `base_type` command returns the node that represents the anonymous sequence.

When writing `idlgen` scripts, you might want to strip away all the layers of `typedef`s to get access to the raw underlying type. This can sometimes result in code such as:

```
# Tcl
proc process_type {type} {
    #--------
    # If "type" is a typdef node then get access to
    # the underlying type.
    #--------
    set base_type $type
    while {[$base_type node_type] == "typedef"} {
        set base_type [$base_type base_type]
    }

    #--------
    # Process it based on its raw type
    #--------
    switch [$base_type node_type] {
        struct       { ... }
        union        { ... }
        sequence     { ... }
        array        { ... }
        default      { ... }
    }
}
```

The need to write code to strip away layers of `typedef`s can arise frequently. To eliminate this tedious coding task, a command called `true_base_type` is defined in `node`. For most node types, this command simply returns the node directly. However, for `typedef` nodes, this command strips away all the layers of `typedef`s, and returns the underlying type. Thus, the above example could be rewritten more concisely as:

```
# Tcl
proc process_type {type} {
    set base_type [$type true_base_type]
    switch [$base_type node_type] {
        struct       { ... }
        union        { ... }
        sequence     { ... }
        array        { ... }
        default      { ... }
    }
}
```

Visiting Hidden Nodes

As mentioned earlier ("The all Pseudo-Node" on page 33), using the `all` pseudo-node as a parameter to the `rcontents` command is a convenient way to visit most nodes in the parse tree. For example:

```
# Tcl
foreach node [$idlgen(root) rcontents all all] {
    ...
}
```

However, the above code segment does not visit the nodes that represent:

- Built-in IDL types such as `long`, `short`, `boolean`, or `string`.
- Anonymous sequences or anonymous arrays.

The `all` pseudo-node does not really represent all types. However, it does represent all types that most scripts want to explicitly process.

It is possible to visit these hidden nodes explicitly. For example, the following code fragment processes all the nodes in the parse tree, including built-in IDL types and anonymous sequences and arrays.

```
# Tcl
set want {all sequence array}
set list [$idlgen(root) rcontents $want all]
set everything [concat $list [idlgen_list_builtin_types]]
foreach node $everything {
    ...
}
```

Other Node Types

Every construct in IDL maps to a particular type of node that either inherits from the `node` abstract node or from the `scope` abstract node. The examples given have only covered a small number of the IDL constructs that are available. The different types of node are arranged in an inheritance hierarchy. For a reference guide that lists all of the node types and available commands, see "IDL Parse Tree Nodes" on page 279.

Traversing the Parse Tree with contents

This section discusses how to create `idlgrep`, a genie that can search an IDL file, looking for any constructs that match a specified wild card. This genie is similar to the UNIX `grep` utility, but is specifically for IDL files.

Searching an IDL File with idlgrep

An example use of the idlgrep genie is to search the finance.idl for any construct that begins with an 'a' or an 'A':

```
idlgen idlgrep.tcl finance.idl "[A|a]*"

Construct   : interface
Local Name  : Account
Scoped Name : Account
File        : finance.idl
Line Number : 1

Construct   : attribute
Local Name  : accountNumber
Scoped Name : Account::accountNumber
File        : finance.idl
Line Number : 2
```

The genie should examine the whole parse tree and look for constructs that match the wild card criteria specified on the command line. It is limited to search only for the interface, operation, exception, and attribute constructs.

Development Iterations

The idlgrep genie is developed in a series of iterations:

Search Using contents

The following is a first attempt at writing the idlgrep genie:

```
# Tcl
if {![idlgen_parse_idl_file "finance.idl"]} {
    exit 1
}
set want {interface operation attribute exception}
set node_list [$idlgen(root) contents $want]
foreach node $node_list {
    puts [$node s_name]
}
```

Run the idlgen interpreter from the command line:

```
idlgen idlgrep.tcl

Account
Bank
```

Using the contents command on the root scope obtains a list of all the interface, operation, and attribute constructs that are in the root scope of the finance.idl file, and the root scope only. This set of results is incomplete as the search goes no further than the root scope. The next iteration refines the functionality of the idlgrep genie.

Search Using rcontents

The previous Tcl script could be expanded so that it traverses the whole parse tree using only the contents command. However, the rcontents command enables a more concise solution. The types of construct the genie is looking for appear only in the module and interface scopes, so the genie only needs to search those scopes.

This information is passed to the rcontents command in the following way:

```
# Tcl
if {![idlgen_parse_idl_file "finance.idl"]} {
    exit 1
}
set want {interface operation attribute exception}
set recurse_into {module interface}
set node_list [$idlgen(root) rcontents $want
   $recurse_into]
foreach node $node_list {
    puts "[$node node_type] [$node s_name]"
}
```

Run the idlgen interpreter from the command line:

```
idlgen idlgrep.tcl

interface Account
attribute Account::accountNumber
attribute Account::balance
operation Account::makeDeposit
interface Bank
operation Bank::findAccount
operation Bank::newAccount
```

Complete Search Genie

Assume that another requirement for this utility is to allow a user to specify whether or not the search should consider files in the #include statements. This can be accomplished with code similar to the following:

```
# Tcl
foreach node [$result_node_list] {
    if {![same_file_function $node]} {
        continue; # not interested in this node
    }
    .. # Do some processing
}
```

You can code this more elegantly by using a further feature of the `rcontents` command (this feature is also provided by `contents`). The general syntax of the `rcontents` command invoked on a *scope_node* scope node is:

$*scope_node* `rcontents` *`node_types scope_types`* [*`filter_func`*]

By passing the optional *`filter_func`* parameter to the `rcontents` command the resulting list of nodes can be filtered in-line. The *`filter_func`* parameter is the name of a function that returns either `true` or `false` depending on whether or not the node that was passed to it is to be added to the search list returned by `rcontents`.

To complete the basic `idlgrep` genie, the *`filter_func`* parameter is added to the `rcontents` command and support is added for the wild card and IDL file command line parameters:

```
# Tcl
proc same_file_function {node} {
    return [$node is_in_main_file]
}
if {$argc != 2} {
    puts "Usage idlgen.tcl <idlfile> <search_exp>"
    exit 1
}
set search_for [lindex $argv 1]
if {![idlgen_parse_idl_file [lindex $argv 0]]} {
    exit
}
set want {interface operation attribute exception}
set recurse_into {module interface}
set node_list [$idlgen(root) rcontents $want
   $recurse_into same_file_function]

foreach node $node_list {
    if [string match $search_for [$node l_name]] {
        puts "Construct   : [$node node_type]"
        puts "Local Name  : [$node l_name]"
        puts "Scoped Name : [$node s_name]"
        puts "File        : [$node file]"
        puts "Line Number : [$node line]"
        puts ""
    }
}
```

Run the completed genie on the `finance.idl` file:

```
idlgen idlgen.tcl finance.idl "[A|a]*"

Construct   : interface
Local Name  : Account
Scoped Name : Finance::Account
File        : finance.idl
Line Number : 22

Construct   : attribute
Local Name  : accountNumber
Scoped Name : Finance::Account::accountNumber
File        : finance.idl
Line Number : 23
```

```
To further test the genie, you can try it on a larger IDL
   file:
idlgen idlgen.tcl ifr.idl "[A|a]*"

Construct   : attribute
Local Name  : absolute_name
Scoped Name : Contained::absolute_name
File        : ifr.idl
Line Number : 73

Construct   : interface
Local Name  : AliasDef
Scoped Name : AliasDef
File        : ifr.idl
Line Number : 322

Construct   : interface
Local Name  : ArrayDef
Scoped Name : ArrayDef
File        : ifr.idl
Line Number : 343

Construct   : interface
Local Name  : AttributeDef
Scoped Name : AttributeDef
File        : ifr.idl
Line Number : 366
```

The next few chapters extend the ideas shown here and allow better genies to be developed. For example, `idlgrep.tcl` could be easily improved by allowing the user to specify more than one IDL file on the command line or by allowing further search options to be defined in a configuration file. The commands that allow the programmer to achieve such tasks are discussed in “Configuring Genies”.

Recursive Descent Traversal

The main method of traversing an IDL parse tree is to use the scoping nodes to locate and move to known nodes or known types of node. The previous examples in this chapter show how a programmer can selectively move down the parse tree and examine the sections that are relevant to the genie’s domain. However, a more complete traversal of the parse tree is needed by some genies.

One such blind, but complete, traversal technique is to use the `rcontents` command:

```
# Tcl
if {![idlgen_parse_idl_file "finance.idl"]} {
    exit
}
set node_list [$idlgen(root) rcontents all all]
foreach node $node_list {
    puts "[$node node_type]: [$node s_name]"
}
```

This search provides a long list of the nodes in the parse tree in the order of traversal. However, the traversal structure of the parse tree is harder to extract because this approach does not allow the parse tree to be analyzed on a node-by-node basis as the traversal progresses.

Recursive descent is a general technique for processing all (or most) of the nodes in the parse tree in a way that allows the nodes to be examined as the traversal progresses. However, before explaining how to use recursive descent in `idlgen` scripts, it is necessary to first explain how polymorphism is used in Tcl.

Polymorphism in Tcl

Consider this short application:

```
# Tcl
proc eat_vegetables {} {
    puts "Eating some veg"
}
proc eat_meat {} {
    puts "Eating some meat"
}
foreach item { meat vegetables vegetables } {
    eat_$item
}
```

Run this application through `idlgen`:

```
idlgen meatveg.tcl

Eating some meat
Eating some veg
Eating some veg
```

This demonstrates polymorphism using Tcl *string substitution*.

Recursive Descent Traversal through Polymorphism

Polymorphism through string substitution makes it easy to write recursive descent scripts. Imagine a genie that converts an IDL file into another file format. The target file is to be indented depending on how deep the IDL constructs are in the parse tree.

```
// Converted IDL
module aModule
(
        interface aInterface
        (
                void aOperation()
        )
)
```

This type of genie is perfect for the recursive descent mechanism. Consider the key command procedure that performs the polymorphism in this genie:

```
# Tcl
proc process_scope {scope} {
    foreach item [$scope contents all] {
        process_[$item node_type] $item
    }
}
```

As each scope node is examined it can be passed to the process_scope command procedure for further traversal. This procedure calls the appropriate node processing procedure by appending the node type name to the string process_. So, if a node that represents a module is passed to the process_scope procedure, it calls a procedure called process_module. This procedure is defined as follows:

```
# Tcl
proc process_module {m} {
    output "[indent] module [$m l_name]\n"
    output "(\n"

    increment_indent_level
    process_scope $m
    decrement_indent_level

    output "[indent] )"
}
```

If the module contains interfaces, process_scope then calls a command procedure called process_interface for each interface:

```
# Tcl
proc process_interface {i} {
    output "    [indent] interface [$i l_name]\n"
    output "(\n"

    increment_indent_level
    process_scope $i
    decrement_indent_level

    output "[indent] )"
}
```

This genie can then start the traversal by simply calling the process_scope command procedure on the root of the parsed IDL file:

```
# Tcl
process_scope $idlgen(root)
```

This example allows every construct in the IDL file to be examined and still allows you to be in control when it comes to the traversal of the parse tree.

Processing User-Defined Types

The idlgen_list_builtin_types command returns a list of all the built-in IDL types. The idlgen interpreter provides a similar command that returns a list of all the user-defined IDL types:

idlgen_list_user_defined_types exception

This command takes one argument that should be either exception or any other string (for example, no exception or ""). If the argument is exception then user-defined exceptions are included in the list of user-defined types that are returned. If the argument is any string other than exception, the user-defined exceptions are not included in the list of user-defined types that are returned. For example:

```
# Tcl
foreach type [idlgen_list_user_defined_types "exception"]
   {
         process_[$type node_type] $type
}
```

Another utility command provided by idlgen is:

idlgen_list_all_types exception

This command is a simple wrapper around calls to idlgen_list_builtin_types and idlgen_list_user_defined_types.

Recursive Structs and Unions

IDL permits the definition of recursive struct and recursive union types. A struct or union is said to be recursive if it contains a member whose type is an anonymous sequence of the enclosing struct or union. The following are examples of recursive types:

```
struct tree {
    long                          data;
    sequence<tree>                children;
};
union widget switch(long) {
    case 1: string                abc;
    case 2: sequence<widget>      xyz;
};
```

Some genies may have to do special-case processing for recursive types. The idlgen interpreter provides the following utility commands to aid this task:

Table 3: *Utility Functions for Special-Case Processing*

Command	Description
idlgen_is_recursive_type *type*	Returns: 1: if *type* is a recursive type. 0: if *type* is not recursive. For example, this command returns 1 for both the tree and widget types.

Table 3: *Utility Functions for Special-Case Processing*

Command	Description
idlgen_is_recursive_member *member*	Returns: 1: if *member* (a field of a struct or a branch of a union) has a recursive type. 0: if *member* does not have a recursive type. For example, the children field of the above tree is a recursive member, but the data field is not.
idlgen_list_recursive_member_ types	Traverses the parse tree and returns a list of all the anonymous sequences that are used as types of recursive members. For the above IDL definitions, this command returns a list containing the anonymous sequence<tree> and sequence<widget> types used for the children member of tree and the xyz member of widget, respectively.

Configuring Genies

This chapter describes how to write genies that are easily configurable for the genie user.

There are two related mechanisms that allow a genie user to specify their preferences and options,

- Command line-arguments
- Configuration files

This chapter discusses these two topics and describes how to make your genies flexible through configuration. Appendix 1 on page 273 provides a reference to the commands discussed in this chapter.

Using Command-Line Arguments

Most useful command-line programs take command-line arguments. Because `idlgen` is predominately a command-line application, your genies will invariably use command-line arguments as well. The code generation toolkit supplies functionality to parse command-line arguments easily.

In this section

This section covers the following topics:

Processing the Command Line

Enhancing the idlgrep Genie

Although the `idlgrep` application ("Processing an IDL File" on page 25) uses command-line options it assumes that the IDL file is the first parameter and the wild card is the second. Instead of hard coding these settings a more intelligent approach to command-line processing that does not make assumptions about argument ordering is preferable. It would also be useful if this application allowed multiple IDL files to be specified on the command-line.

Searching for IDL files to process

Taking these points into consideration, the first thing the idlgrep genie must do is find out which IDL files to process. It does this using the built-in idlgen_getarg command to search the command-line arguments for IDL files:

```
# Tcl
set idl_file_list {}
set cl_args_format {
    {".+\\.[iI][dD][lL]"     0    idl_file }
    {"-h"                    0    usage    }
}
while {$argc > 0} {
    # Extract one option at a time from the command
    # line using 'idlgen_getarg'
    idlgen_getarg $cl_args_format arg param symbol

    switch $symbol {
        idl_file  {lappend idl_file_list $arg}
        usage     {puts "Usage ..."; exit 1}
        default   {puts "Unknown argument $arg"
                   puts "Usage ..."
                   exit 1
                  }
    }
}
foreach file $idl_file_list {
    puts $file
}
```

Note: Each time the idlgen_getarg command is run, the $argc variable is decremented and the command-line argument removed from $argv.

How the idlgen_getarg command works

The idlgen_getarg command works by examining the command-line for any argument that matches the search criteria provided to it. It then extracts all the information associated with the matched argument and assigns the results to the given variables.

The following is an example of what the preceding Tcl script does with some IDL files passed as command-line parameters:

```
idlgen idlgrep.tcl bank.idl ifr.IDL daemon.iDl

bank.idl
ifr.IDL
daemon.iDl
```

If the genie user wants to see all of the available command-line options they can use the `-h` option for help:

```
idlgen idlgrep.tcl -h

Usage...
```

Syntax for the idlgen_getarg Command

The `idlgen_getarg` command takes four parameters:

```
idlgen_getarg cl_args_format arg param symbol
```

The first parameter, *cl_args_format*, is a data structure that describes which command-line arguments are being searched for. The three parameters—*arg*, *param*, and *symbol*—are variable names that are assigned values by the `idlgen_getarg` command, as described in Table 4.

Table 4: `idlgen_getarg` *Arguments*

Arguments	Purpose
arg	The text value of the command-line argument that was matched on this run of the command.
param	The parameter (if any) to the command-line argument that was matched. For example, a command-line option `-search a*` would have the parameter `a*`.
symbol	The symbol for the command-line argument that was specified in the format parameter. This can be used to find out which command-line argument was actually extracted.

Note: There is no need to use the `smart_source` command to access the `idlgen_getarg` command, because `idlgen_getarg` is a built-in command.

Searching for Command-Line Arguments

First parameter

This first parameter to the idlgen_getarg command is a data structure that describes the syntax of the command-line arguments to search for. In the idlgrep application example, see page 45, this first parameter is set to the following:

```
# Tcl
set cl_args_format {
    {".+\\.[iI][dD][lL]"    0    idl_file }
    {"-h"                   0    usage    }
}
```

This data structure is a list of sub-lists. Each sub-list is used to specify the search criteria for a type of command-line parameter.

First element of sublist

The first element of each sub-list is a regular expression that specifies the format of the command-line arguments. In the example shown above, the first sub-list is looking for any command-line argument that ends in .IDL or any case insensitive equivalent of .IDL.

Second element of sublist

The second element of each sub-list is a boolean value that specifies whether or not the command-line argument has a further parameter to it. A value 0 indicates that the command-line argument is self-contained. A value 1 indicates that the next command-line argument is a parameter to the current one.

Third element of sublist

The third element of each sub-list is a reference symbol. This symbol is what idlgen_getarg assigns to its fourth parameter if the regular expression element matches a command-line argument. Typically, if the regular expression does not contain any wild cards the symbol is identical to the first element. If the regular expression does contain wild cards the symbol can be used later on in the application to reference the command-line argument independently of its physical value.

More Examples of Command-Line Processing

Example of idlgen_getarg command

The following is another example of the `idlgen_getarg` command as it loops through some command-line arguments:

```
# Tcl
set inc_list {}
set idl_list {}
set extension "not specified"
set cmd_line_args_fmt {
    { "-I.+"                0    include }
    { "-ext"                1    ext     }
    { ".+\\.[iI][dD][lL]"   0   idlfile  }
}

while {$argc > 0} {
    idlgen_getarg $cmd_line_args_fmt arg param symbol

    switch $symbol {
     include   { lappend inc_list $arg }
     ext       { set extension $param }
     idlfile   { lappend idl_list $arg }
     default   { puts "Unknown argument $arg"
                 puts "Usage ..."
                 exit 1
               }
    }
}
foreach include_path $inc_list {
    puts "Include path is $include_path"
}
foreach idl_file $idl_list {
    puts "IDL file specified is $idl_file"
}
puts "Extension is $extension"
```

Running the application

Run this application with appropriate command-line arguments:

```
idlgen cla.tcl bank.idl car.idl -ext cpp

IDL file specified is bank.idl
IDL file specified is car.idl
Extension is cpp
```

The following is a different set of command-line parameters:

```
idlgen cla.tcl -I/home/iona -I/orbix/inc

Include path is /home/iona
Include path is /orbix/inc
Extension is not specified
```

Using idlgrep with Command-Line Arguments

Getting the search criteria

To finish the `idlgrep` utility the search criteria must also be taken from the command-line, as well as obtaining the list of IDL files to process:

```
# Tcl
set idl_file_list {}
set search_for "*"
set cl_args_format {
    {".+\\.[iI][dD][lL]"    0    idl_file }
    {-s                     1    reg_exp  }
}
while {$argc > 0} {
    idlgen_getarg $cl_args_format arg param symbol

    switch $symbol {
        idl_file  { lappend idl_file_list $arg }
        reg_exp   { set search_for $param }
        default   { puts "usage: ..."; exit }
    }
}
foreach file $idl_file_list {
    grep_file $file search_for
}
```

grep_file command

The following is the full listing for the `grep_file` command procedure:

```
# Tcl
proc grep_file {file searchfor} {
    global idlgen

    if {![idlgen_parse_idl_file $file]} {
        return
    }
    set want {interface operation attribute exception}
    set recurse_into {module interface}
    set node_list [$idlgen(root) rcontents $want
   $recurse_into]
    foreach node $node_list {

        if [string match $searchfor [$node l_name]] {
            puts "Construct   : [$node node_type]"
            puts "Local Name  : [$node l_name]"
            puts "Scoped Name : [$node s_name]"
            puts "File        : [$node file]"
            puts "Line Number : [$node line]"
            puts ""
        }
    }
}
```

Specifying multiple IDL files on the command line

Multiple IDL files can now be specified on the command-line, and the command-line arguments can be placed in any order:

```
idlgen idlgrep2.tcl finance.idl -s "a*" ifr.idl

Construct   : attribute
Local Name  : accountNumber
Scoped Name : Account::accountNumber
File        : finance.idl
Line Number : 21

Construct   : attribute
Local Name  : absolute_name
Scoped Name : Contained::absolute_name
File        : ifr.idl
Line Number : 73
```

Using std/args.tcl

The `std/args.tcl` library provides a command, `parse_cmd_line_args`, that processes the command-line arguments common to most genies. In particular, it picks out IDL file names from the command line and processes the following command-line arguments: `-I`, `-D`, `-v`, `-s`, `-dir`, and `-h`.

Example

The example below illustrates how to use this library:

```
# Tcl
smart_source "std/args.tcl"
parse_cmd_line_args idl_file options
if {![idlgen_parse_idl_file $idl_file $options]} {
        exit 1
}
... # rest of genie
```

Usage

Upon success, the `parse_cmd_line_args` command returns the name of the specified IDL file through the `idl_file` parameter, and preprocessor options through the `options` parameter. However, if the `parse_cmd_line_args` command encounters the `-h` option or any unrecognized option, or if there is no IDL file specified on the command-line, it prints out a usage statement and calls `exit` to terminate the genie. For example, if the above genie is saved to a file called `foo.tcl`, it could be run as follows:

```
idlgen foo.tcl -h

usage: idlgen foo.tcl [options] file.idl
options are:
    -I<directory>       Passed to preprocessor
    -D<name>[=value]    Passed to preprocessor
    -h                  Prints this help message
    -v                  Verbose mode
    -s                 Silent mode (opposite of -v option)
    -dir <directory>    Put generated files in <directory>
```

If you are writing a genie that needs only the above command-line arguments, you can use the unmodified `std/args.tcl` library in your genie. If, however, your genie requires some additional command-line arguments, you can copy the `std/args.tcl` library and modify the copy so that it can process additional command-line arguments. In this way, the `std/args.tcl` library provides a useful starting point for command-line processing in your genies.

Using Configuration Files

The `idlgen` interpreter and the bundled genies use information in a configuration file to enhance the range of options and preferences offered to the genie user. Examples of configurable options are:

- The search path for the `smart_source` command.
- Whether the genie user prefers the TIE or inheritance approach when implementing an interface.
- File extensions for C++ or Java files.

idlgen.cfg configuration file

The `idlgen` interpreter's core settings and preferences are stored in a standard configuration file that, by default, is called `idlgen.cfg`. This file is also used for storing preferences for the bundled applications. It is loaded automatically, but the built-in parser can be used to access other application-specific configuration files if the requirement arises.

In this section

This sections covers the following topics:

Syntax of an idlgen Configuration File

Configuration file syntax

A configuration file consists of a number of statements that assign a value to a name. The name, like a Tcl variable, can have its value assigned to either a string or a list. The syntax of such statements is summarized in Appendix 1 on page 297.

Comments

Text appearing between the # (number sign) character and the end of the line is a comment:

```
# This is a comment
x = "1" ;# Comment at the end
```

Assigning string values

Use the = (equal sign) symbol to assign a string value to a name. Use a ; (semi-colon) to terminate the assignment. The string literal must be enclosed by quotation marks:

```
local_domain = "iona.com";
```

Concatenating strings

Use the + (plus) symbol to concatenate strings. The following example sets the `host` configuration item to the value `amachine.iona.com`:

```
host = "amachine" + "." + local_domain;
```

Assigning a list to a name

Use the = (equals) symbol to assign a list to a name and put the items of the list inside matching [and] symbols:

```
initial_cache = ["times", "courier"];
```

Concatenating lists

Use the + (plus) symbol to concatenate lists. In this example, the `all` configuration item contains the list: `times, courier, arial, dingbats`.

```
all = initial_cache + ["arial", "dingbats"];
```

Scope

Items in a configuration file can be scoped. This can, for example, allow configuration items of the same name to be stored in different scopes.

In the following example, to access the value of dir, use the scoped named fonts.dir:

```
fonts {
    dir = "/usr/lib/fonts";
};
```

Reading the Contents of a Configuration File

You can use the idlgen_parse_config_file command to open a configuration file. The return value of this command is an object that can be used to examine the contents of the configuration file.

The following is a pseudo-code definition for the operations that can be performed on the return value of this configuration file parsing command:

```
class configuration_file {
    enum setting_type {string, list, missing}

    string                  filename()
    list<string>            list_names()
    void                    destroy()
    setting_type            type(
                                string cfg_name)
    string                  get_string(
                                string cfg_name)
    void                    set_string(
                                string cfg_name,
                                string cfg_value )
    list<string>            get_list(
                                string cfg_name)
    void                    set_list(
                                string cfg_item,
                                list<string> cfg_value )
}
```

There are operations to list the whole contents of the configuration file (list_names), query particular settings in the file (get_string, get_list), and alter values in the configuration file (set_string, set_list).

The following Tcl program uses the parse command and manipulates the results, using some of these operations:

```
# Tcl
if { [catch {
    set cfg [idlgen_parse_config_file "shop.cfg"]
   } err] } {
    puts stderr $err
    exit
}
```

```
puts "The settings in '[$cfg filename]' are:"
foreach name [$cfg list_names] {
    switch [$cfg type $name] {
        string {puts "$name:[$cfg get_string $name]"}
        list   {puts "$name:[$cfg get_list $name]"}
    }
}
$cfg destroy
```

Note: You should free associated memory by using the `destroy` operation when the configuration file has been completed.

Consider the case if the contents of the `shop` configuration file are as follows:

```
# shop.cfg
clothes = ["jeans", "jumper", "coat"];
sizes {
    waist       = "32";
    inside_leg  = "32";
};
```

Run this application through `idlgen`:

```
idlgen shopcfg.tcl

The settings in 'shop.cfg' are:
sizes.waist:32
sizes.inside_leg:32
clothes:jeans jumper coat
```

Note: For more detail about the commands and operations discussed in this section, Appendix 1 on page 273.

The Standard Configuration File

When `idlgen` starts, it reads the `idlgen.cfg` configuration file from the default configuration directory. To use an alternative configuration file, set the `IT_IDLGEN_CONFIG_FILE` environment variable to the absolute pathname of the alternative configuration file. The details of the configuration file are then stored in a global variable called `$idlgen(cfg)`. This variable can then be accessed at any time by your own genies.

Note: There is no restriction on the name of the standard configuration file but it is recommended that you follow the convention of naming it `idlgen.cfg`.

Using idlgrep with Configuration Files

Consider a new requirement to enhance the `idlgrep` genie once more to allow the genie user to specify which IDL constructs they want the search to include. The genie user might also want to specify which constructs to search recursively. It would be time consuming for the user to specify these details on the command-line; it is better to have these settings stored in the standard configuration file.

Assume that the standard configuration file contains the following scoped entries:

```
# idlgen.cfg
idlgrep {
    constructs    = [ "interface", "operation" ];
    recurse_into  = [ "module", "interface" ];
};
```

The following code from the `grep_file` command procedure must be replaced (for a full listing of this command procedure, see page 50):

```
# Tcl
set want {interface operation attribute exception}
set recurse_into {module interface}
```

The following code must be inserted as the replacement:

```
# Tcl
set want [$idlgen(cfg) get_list "idlgrep.constructs"]
set recurse_into [$idlgen(cfg) get_list
   "idlgrep.recurse_into"]
```

Running the `idlgen` interpreter with the new variation of the `idlgrep` genie gives a more precise search:

```
idlgen idlgrep3.tcl finance.idl -s "A*"

Construct   : interface
Local Name  : Account
Scoped Name : Account
File        : finance.idl
Line Number : 20
```

This is a good first step and gives the genie user a much more flexible application.

The current version of the application assumes that all of the configuration values are present in the configuration file. The application can be improved such that it automatically provides default values if entries are missing from the configuration file.

The following Tcl script shows the improved version of the application:

```
# Tcl
proc get_cfg_entry {cfg name default} {
    set type [$cfg type $name]
    switch $type {
        missing {return $default}
        default {return [$cfg get_$type $name]}
    }
}
...
set want [get_cfg_entry $idlgen(cfg) "idlgrep.constructs"
   \
            {interface operation}]
set recurse_into [get_cfg_entry $idlgen(cfg) \
            "idlgen.recurse_into" {module interface}]
```

The `type` operation allows you to determine whether the configuration item exists and whether it is a list entry or a string entry. The code provides a default value if the configuration entry is missing.

Default Values

There is another way you can provide a default value; the `get_string` and `get_list` operations can take an optional second parameter, which is used as a default if the entry is not found. An equivalent of the above code (ignoring the possibility that the entry could be a string entry) is:

```
# Tcl
set want [$idlgen(cfg) get_list "idlgrep.constructs" \
        {interface module}]
set recurse_into [$idlgen(cfg) get_list
   "idlgen.recurse_into" \
        module interface}]
```

Developing a C++ Genie

The `std/cpp_poa_lib.tcl` *file is a library of Tcl command procedures that map IDL constructs into their C++ counterparts. The server-side IDL-to-C++ mapping is based on the CORBA Portable Object Adapter specification.*

Identifiers and Keywords

There are a number of commands that help map IDL data types to their C++ equivalents.

The CORBA mapping generally maps IDL identifiers to the same identifier in C++, but there are some exceptions required, to avoid clashes. For example, if an IDL identifier clashes with a C++ keyword, it is mapped to an identifier with the prefix `_cxx_`.

Consider the following unusual, but valid, interface:

```
// IDL
interface Strange {
    string for( in long while );
};
```

The interface maps to a C++ class `Strange` in the following way:

```
// C++ - some details omitted
class strange : public virtual CORBA::Object
{
    virtual char*
    _cxx_for(
        CORBA::Long _cxx_while
    );
};
```

Note: Avoid IDL identifiers that clash with keywords in C++ or other programming languages that you use to implement CORBA objects. Although they can be mapped as described, it causes confusion.

The application programming interface (API) for generating C++ identifiers is summarized in Table 5. The `_s_` variants return fully-scoped identifiers whereas the `_l_` variants return non-scoped identifiers.

Table 5: *Commands for Generating Identifiers and Keywords*

Command	Description
`cpp_s_name` *node*	Returns the C++ mapping of a node's scoped name.
`cpp_l_name` *node*	Returns the C++ mapping of a node's local name.

Table 5: *Commands for Generating Identifiers and Keywords*

Command	Description
`cpp_typecode_s_name` *type*	Returns the scoped C++ name of the type code for *type*.
`cpp_typecode_l_name` *type*	Returns the local C++ name of the type code for *type*.

C++ Prototype

A typical approach to developing a C++ genie is to start with a working C++ example. This C++ example should exhibit most of the features that you want to incorporate into your generated code. You can then proceed by reverse-engineering the C++ example; developing a Tcl script that recreates the C++ example when it receives the corresponding IDL file as input.

The C++ example employed to help you develop the Tcl script is referred to here as a *C++ prototype*. In the following sections, two fundamental C++ prototypes are presented and analyzed in detail.

- The first C++ prototype demonstrates how to invoke a typical CORBA method (client-side prototype).
- The second C++ prototype demonstrates how to implement a typical CORBA method (server-side prototype).

The script derived from these fundamental C++ prototypes can serve as a starting point for a wide range of applications, including the automated generation of wrapping code for legacy systems.

The C++ prototypes described in this chapter use the following IDL:

```
// IDL
struct  widget            {long a;};
typedef sequence<long>   longSeq;
typedef long              long_array[10];

interface foo {
    longSeq op(
        in widget       p_widget,
        inout string    p_string,
        out longSeq     p_longSeq,
        out long_array  p_long_array);
};
```

Client-Side Prototype

The client-side prototype demonstrates a CORBA invocation of the `foo::op()` IDL operation. Parameters are allocated, a `foo::op()` invocation is made, and the parameters are freed at the end.

```
// C++
//--------
// Declare parameters for operation
//--------
widget p_widget;
char * p_string;
longSeq* p_longSeq;

long_array p_long_array;
longSeq* _result;

//--------
// Initialize "in" and "inout" parameters
//--------
p_widget = other_widget;
p_string = CORBA::string_dup(other_string);

//--------
// Invoke the operation
//--------
try {
        _result = obj->op(
        p_widget,
        p_string,
        p_longSeq,
        p_long_array);
} catch(const CORBA::Exception &ex) {
    ... // handle the exception
}

//--------
// Process the returned parameters
//--------
process_string(p_string);
process_longSeq(*p_longSeq);
process_long_array(p_long_array);
process_longSeq(*_result);

//--------
// Free memory associated with parameters
//--------
CORBA::string_free(p_string);
delete p_longSeq;
delete _result;
```

Server-Side Prototype

The server-side prototype demonstrates an implementation of the `foo::op()` IDL operation. This operation demonstrates the use of `in`, `inout` and `out` parameters and has a return value. The code shown in the implementation deals with deallocation, allocation, and initialization of parameters and return values.

```
// C++
longSeq*
fooImpl::op(
    const widget&                    p_widget,
    char *&                          p_string,
    longSeq_out                      p_longSeq,
    long_array                       p_long_array
) throw(CORBA::SystemException)
{

   //---------------------------------------------------
   --
    // Implement the logic of the operation...
    //
    // Process the input variables 'p_widget' and
   'p_string'
    //
    // Calculate, or find, the output data
    //         'other_string', 'other_longSeq',
   'other_long_array'

   //---------------------------------------------------
   --
    ...  // Not shown

    //--------
    // Declare a variable to hold the return value.
    //--------
    longSeq* _result;

    //--------
    // Allocate memory for "out" parameters
    // and the return value, if needed.
    //--------
    p_longSeq = new longSeq;
    _result = new longSeq;

    //--------
    // Assign new values to "out" and "inout"
    // parameters, and the return value, if needed.
    //--------
    CORBA::string_free(p_string);
    p_string = CORBA::string_dup(other_string);
    *p_longSeq = other_longSeq;
    for (CORBA::ULong i1 = 0; i1 < 10; i1 ++) {
        p_long_array[i1] = other_long_array[i1];
    }
    *_result = other_longSeq;
```

```
    if (an_error_occurs) {
        //--------
        // Before throwing an exception, we must
        // free the memory of heap-allocated "out"
        // parameters and the return value,
        // and also assign nil pointers to these
        // "out" parameters.
        //--------
        delete p_longSeq;
        p_longSeq = 0;
        delete _result;
        throw some_exception;
    }

    return _result;
}
```

Invoking an Operation

This section explains how to generate C++ code that invokes a given IDL operation. The process of making a CORBA invocation in C++ can be broken down into the following steps:

1. Declare variables to hold parameters and return value.

   The calling code must declare all `in`, `inout` and `out` parameters before making the invocation. If the return type of the operation is non-`void`, a return value must also be declared.
2. Initialize input parameters.

   The calling code must initialize all `in` and `inout` parameters. There is no need to initialize `out` parameters.
3. Invoke the IDL operation.

   The calling code invokes the operation, passing each of the prepared parameters and retrieving the return value (if any).
4. Process output parameters and return value.

   Assuming no exception has been thrown, the caller processes the returned `inout`, `out` and return values.
5. Release heap-allocated parameters and return value.

   When the caller is finished, any parameters that were allocated on the heap must be deallocated. The return value must also be deallocated.

The following subsections give a detailed example of how to generate complete code for an IDL operation invocation.

Step 1—Declare Variables to Hold Parameters and Return Value

The following example assumes that `_var` variables are not used, to show how explicit memory management statements are generated. In practice, it is usually better to use `_var` variables: their use automates cleanup and simplifies code, especially when exceptions can be thrown.

The following Tcl script illustrates how to declare C++ variables to be used as parameters to (and the return value of) an operation call:

Example 1:

```
# Tcl
set op          [$idlgen(root) lookup "foo::op"]
set is_var      0
set ind_lev     1
set arg_list    [$op contents {argument}]
[***
    //--------
    // Declare parameters for operation
    //--------
***]
foreach arg $arg_list {
1       cpp_gen_clt_par_decl $arg $is_var $ind_lev
}
2   cpp_gen_clt_par_decl $op $is_var $ind_lev
```

The Tcl script is explained as follows:

1. When an *argument* node appears as the first parameter of cpp_gen_clt_par_decl, the command outputs a declaration of the corresponding C++ parameter.
2. When an *operation* node appears as the first parameter of cpp_gen_clt_par_decl, the command outputs a declaration of a variable to hold the operation's return value. If the operation has no return value, the command outputs a blank string.

The previous Tcl code yields the following C++ code:

Example 2:

```
// C++
//--------
// Declare parameters for operation
//--------
widget p_widget;
1   char * p_string;
2   longSeq* p_longSeq;

long_array p_long_array;
3   longSeq* _result;
```

The $pref(cpp,ret_param_name) array element determines the name of the C++ variable that is declared to hold the return value, line 3. Its default value is _result. In lines 1, 2, and 3, the C++ variables are declared as raw pointers. This is because the is_var parameter is set to FALSE in calls to the cpp_gen_clt_par_decl command. If is_var is TRUE, the variables are declared as _var types.

Step 2—Initialize Input Parameters

The following Tcl script shows how to initialize `in` and `inout` parameters:

Example 3:

```
# Tcl
[***
    //--------
    // Initialize "in" and "inout" parameters
    //--------
***]
1 foreach arg [$op args {in inout}] {
    set type [$arg type]
2   set arg_ref [cpp_clt_par_ref $arg $is_var]
    set value "other_[$type s_uname]"

3   cpp_gen_assign_stmt $type $arg_ref $value $ind_lev 0
}
```

The Tcl script is explained as follows:

1. The `foreach` loop iterates over all the `in` and `inout` parameters.
2. The `cpp_clt_par_ref` command returns a reference (not a pointer) to the C++ parameter corresponding to the given argument node, `$arg`.
3. An assignment statement is generated by the `cpp_gen_assign_stmt` command for variables of the given `$type`. The `$arg_ref` argument is put on the left-hand side of the generated assignment statement and the `$value` argument on the right-hand side. Note that this command expects its second and third arguments to be references.

The previous Tcl script yields the following C++ code:

```
//--------
// Initialize "in" and "inout" parameters
//--------
p_widget = other_widget;
p_string = CORBA::string_dup(other_string);
```

Step 3—Invoke the IDL Operation

The following Tcl script shows how to invoke an IDL operation, pass parameters, and assign the return value to a variable:

Example 4:

```
# Tcl
1 set ret_assign    [cpp_ret_assign $op]
set op_name         [cpp_l_name $op]
set start_str       "\n\t\t\t"
set sep_str         ",\n\t\t\t"
```

Example 4:

```
2 set call_args [idlgen_process_list $arg_list \
                        cpp_l_name $start_str $sep_str]
[***
    //--------
    // Invoke the operation
    //--------
    try {
        @$ret_assign@obj->@$op_name@(@$call_args@);
    } catch(const CORBA::Exception &ex) {
        ... // handle the exception
    }
***]
```

The Tcl script is explained as follows:

1. The expression `[cpp_ret_assign $op]` returns the string, `"_result ="`. If the operation invoked does not have a return type, it returns an empty string, `""`.
2. The parameters to the operation call are formatted using the command `idlgen_process_list`. For more about this command, see “idlgen_process_list” on page 140.

The previous Tcl script yields the following C++ code:

```
// C++
//--------
// Invoke the operation
//--------
try {
       _result = obj->op(p_widget, p_string, p_longSeq,
                         p_long_array);
} catch(const CORBA::Exception &ex) {
    ... // handle the exception
}
```

Step 4—Process Output Parameters and Return Value

The following Tcl script shows that the techniques used to process output parameters are similar to those used to process input parameters.

Example 5:

```
# Tcl
[***
    //--------
    // Process the returned parameters
    //--------
***]
1 foreach arg [$op args {out inout}] {
    set type [$arg type]
    set name [cpp_l_name $arg]
2   set arg_ref [cpp_clt_par_ref $arg $is_var]
[***
    process_@[$type s_uname]@(@$arg_ref@);
***]
}
set ret_type [$op return_type]
if {[$ret_type l_name] != "void"} {
```

Example 5:

```
3    set ret_ref [cpp_clt_par_ref $op $is_var]
[***
    process_@[$ret_type s_uname]@(@$ret_ref@);
***]
}
```

The Tcl script is explained as follows:

1. The foreach loop iterates over all the out and inout parameters.
2. The command cpp_clt_par_ref returns a reference (not a pointer) to the C++ parameter corresponding to the given argument node, $arg.
3. When an operation node $op is supplied as the first parameter to cpp_clt_par_ref, the command returns a reference to the return value of the operation.

The previous Tcl script yields the following C++ code:

```
// C++
//--------
// Process the returned parameters
//--------
process_string(p_string);
process_longSeq(*p_longSeq);
process_long_array(p_long_array);
process_longSeq(*_result);
```

Step 5—Release Heap-Allocated Parameters and Return Value

The following Tcl script shows how to free memory associated with the parameters and return value of an operation call. To illustrate explicit memory management, the example assumes that is_var is set to FALSE.

Example 6:

```
# Tcl
[***
    //--------
    // Free memory associated with parameters
    //--------
***]
foreach arg $arg_list {
    set name [cpp_l_name $arg]

1    cpp_gen_clt_free_mem_stmt $arg $is_var $ind_lev
}
2 cpp_gen_clt_free_mem_stmt $op $is_var $ind_lev
```

The Tcl script is explained as follows:

1. The `cpp_gen_clt_free_mem_stmt` command generates a C++ statement to free memory for the parameter corresponding to `$arg`. If no memory management is needed (either because the parameter is a stack variable or because `$is_var` is equal to `1`) the command generates a blank string.
2. When an operation node is supplied as the first parameter to the `cpp_gen_clt_free_mem_stmt` command, a C++ statement is generated to free the memory associated with the return value. If no memory management is needed, the command generates a blank string.

The previous Tcl script yields the following C++ code to explicitly free memory:

```
// C++
//--------
// Free memory associated with parameters
//--------
CORBA::string_free(p_string);
delete p_longSeq;
delete _result;
```

Statements to free memory are generated only if needed. For example, there is no memory-freeing statement generated for `p_widget` or `p_long_array`, because these parameters had their memory allocated on the stack rather than on the heap.

> **Note:** It is good practice to set the `is_var` argument to TRUE so that parameters and the `_result` variable are declared as `_var` types. In this case memory management is automatic and no memory-freeing statements are generated. The resulting code is simpler and safer; `_vars` clean up automatically, even if an exception is thrown.

Invoking an Attribute

To invoke an IDL attribute, you must perform similar steps to those described in "Invoking an Operation" on page 63. However, a different form of the client-side Tcl commands are used:

```
cpp_clt_par_decl name type dir is_var
cpp_clt_par_ref name type dir is_var
cpp_clt_free_mem_stmt name type dir is_var
cpp_clt_need_to_free_mem name type dir is_var
```

Similar variants are available for the `gen_` counterparts of commands:

```
cpp_gen_clt_par_decl name type dir is_var ind_lev
cpp_gen_clt_free_mem_stmt name type dir is_var ind_lev
```

These commands are the same as the set of commands used to generate an operation invocation, except they take a different set of arguments. You specify the *name* and *type* of the attribute as the first two arguments. The *dir* argument can be `in` or `return`, indicating an attribute's modifier or accessor respectively. The *is_var* and *ind_level* arguments have the same effect as in "Step

1—Declare Variables to Hold Parameters and Return Value” on page 63.

Implementing an Operation

This section explains how to generate C++ code that provides the implementation of an IDL operation. The steps typically followed are:

1. Generate the operation signature.
2. Process input parameters.

   The function body first processes the in and inout parameters that it has received from the client.
3. Declare return value and allocate parameter memory.

   The return value is declared. Memory must be allocated for out parameters and the return value.
4. Initialize output parameters and return value.

   The inout and out parameters and the return value must be initialized.
5. Manage memory when throwing exceptions.

   It is important to deal with exceptions correctly. The inout and out parameters and return value must always be freed before throwing an exception.

Step 1—Generate the Operation Signature

There are two kinds of operation signature. The cpp_gen_op_sig_h command generates a signature for inclusion in a C++ header file. The command cpp_gen_op_sig_cc generates a signature for the method implementation.

The following Tcl script generates the signature for the implementation of the foo::op operation:

```
# Tcl
...
set op [$idlgen(root) lookup "foo::op"]

cpp_gen_op_sig_cc $op
```

The previous script generates the following C++ code:

```
// C++
longSeq*
fooImpl::op(
    const widget&                    p_widget,
    char *&                          p_string,
    longSeq_out                      p_longSeq,
    long_array                       p_long_array
) throw(
    CORBA::SystemException
)
```

The names of the C++ parameters are the same as the parameter names declared in IDL.

Step 2—Process Input Parameters

This step is similar to "Step 4—Process Output Parameters and Return Value" on page 66. It is, therefore, not described in this section.

Step 3—Declare the Return Value and Allocate Parameter Memory

The following Tcl script declares a local variable that can hold the return value of the operation. It then allocates memory for `out` parameters and the return value, if required.

Example 7:

```
# Tcl
set op          [$idlgen(root) lookup "foo::op"]
set ret_type    [$op return_type]
set is_var      0
set ind_lev     1
set arg_list    [$op contents {argument}]
if {[$ret_type l_name] != "void"} {
[***
        //--------
        // Declare a variable to hold the return value.
        //--------
1     @[cpp_srv_ret_decl $op 0]@;

***]
}
[***
        //--------
        // Allocate memory for "out" parameters
        // and the return value, if needed.
        //--------
***]
foreach arg [$op args {out}] {
2       cpp_gen_srv_par_alloc $arg $ind_lev
}
3 cpp_gen_srv_par_alloc $op $ind_lev
```

The Tcl script is explained as follows:

1. The `cpp_srv_ret_decl` command returns a statement that declares the return value of the an operation. The first argument, `$op`, is an operation node. The second (optional) argument is a boolean flag that indicates whether or not the returned declaration also allocates memory for the return value.
2. The `cpp_gen_srv_par_alloc` command allocates memory for the C++ parameter corresponding to the `$arg` argument node.
3. When the `$op` operation node is supplied as the first argument to the `cpp_gen_srv_par_alloc` command, the command allocates memory for the operation's return value.

The previous Tcl script generates the following C++ code:

```
// C++
//--------
// Declare a variable to hold the return value.
//--------
longSeq* _result;

//--------
// Allocate memory for "out" parameters
// and the return value, if needed.
//--------
p_longSeq = new longSeq;
_result = new longSeq;
```

The declaration of the `_result` variable (line 1 of the Tcl script) is separated from allocation of memory for it (line 3 of the Tcl script). This gives you the opportunity to throw exceptions before allocating memory, which eliminates memory management responsibilities associated with throwing an exception. If you prefer to allocate memory for the `_result` variable in its declaration, change line 1 of the Tcl script so that it passes `1` as the value of the *`alloc_mem`* parameter, and delete line 3 of the Tcl script. If you make these changes, the declaration of `_result` changes to:

```
longSeq* _result = new longSeq;
```

Step 4—Initialize Output Parameters and the Return Value

The following Tcl script iterates over all `inout` and `out` parameters and the return value, and assigns values to them:

Example 8:

```
# Tcl
[***
       //--------
       // Assign new values to "out" and "inout"
       // parameters, and the return value, if needed.
       //--------
***]
foreach arg [$op args {inout out}] {
        set type    [$arg type]
1       set arg_ref [cpp_srv_par_ref $arg]
        set name2   "other_[$type s_uname]"
        if {[$arg direction] == "inout"} {
2           cpp_gen_srv_free_mem_stmt $arg $ind_lev
        }
3       cpp_gen_assign_stmt $type $arg_ref $name2 \
                                 $ind_lev 0
}
if {[$ret_type l_name] != "void"} {
4       set ret_ref [cpp_srv_par_ref $op]
        set name2   "other_[$ret_type s_uname]"
5       cpp_gen_assign_stmt $ret_type $ret_ref \
                            $name2 $ind_lev 0
}
```

The Tcl script is explained as follows:

1. The `cpp_srv_par_ref` command returns a reference to the C++ parameter that corresponds to the `$arg` argument node.
2. Before an assignment can be made to an `inout` parameter, it is necessary to explicitly free the old value of the `inout` parameter. The `cpp_gen_srv_free_mem_stmt` command generates a C++ statement to free memory for the parameter corresponding to the `$arg` argument node.
3. An assignment statement is generated by the `cpp_gen_assign_stmt` command for variables of the given `$type`. The `$arg_ref` argument is put on the left-hand side of the generated assignment statement and the `$name2` argument on the right-hand side. This command expects its second and third arguments to be references. The last argument, the `scope` flag, works around a bug in some C++ compilers; see "cpp_assign_stmt" on page 163 for details.
4. When the `$op` operation node is supplied as the first argument to the `cpp_srv_par_ref` command, it returns a reference to the operation's return value.
5. This line generates an assignment statement to initialize the return value.

The previous Tcl script generates the following C++ code:

```
// C++
//--------
// Assign new values to "out" and "inout"
// parameters, and the return value, if needed.
//--------
CORBA::string_free(p_string);
p_string = CORBA::string_dup(other_string);
*p_longSeq = other_longSeq;
for (CORBA::ULong i1 = 0; i1 < 10; i1 ++) {
    p_long_array[i1] = other_long_array[i1];
}
*_result = other_longSeq;
```

Step 5—Manage Memory when Throwing Exceptions

If an operation throws an exception after it allocates memory for `out` parameters and the return value, some memory management must be carried out before throwing the exception. These duties are shown in the following Tcl code:

Example 9:

```
# Tcl
[***
       if (an_error_occurs) {
           //--------
           // Before throwing an exception, we must
           // free the memory of heap-allocated "out"
           // parameters and the return value,
           // and also assign nil pointers to these
           // "out" parameters.
           //--------
***]
foreach arg [$op args {out}] {
```

Example 9:

```
1          set free_mem_stmt [cpp_srv_free_mem_stmt $arg]
           if {$free_mem_stmt != ""} {
               set name [cpp_l_name $arg]
               set type [$arg type]
[***
               @$free_mem_stmt@;
2              @$name@ = @[cpp_nil_pointer $type]@;
***]
           }
}
3 cpp_gen_srv_free_mem_stmt $op 2
[***
               throw some_exception;
         }
***]
```

This Tcl script is explained as follows:

1. The `cpp_srv_free_mem_stmt` command returns a C++ statement to free memory for the parameter corresponding to `$arg`.
2. Nil pointers are assigned to `out` parameters using the `cpp_nil_pointer` command.
3. When the `$op` operation node is supplied as the first argument to `cpp_gen_srv_free_mem_stmt`, the command generates a C++ statement to free memory for the return value.

The previous Tcl script generates the following C++ code:

```
// C++
if (an_error_occurs) {
    //--------
    // Before throwing an exception, we must
    // free the memory of heap-allocated "out"
    // parameters and the return value,
    // and also assign nil pointers to these
    // "out" parameters.
    //--------
    delete p_longSeq;
    p_longSeq = 0;
    delete _result;
    throw some_exception;
}
```

Implementing an Attribute

Recall that the `cpp_srv_par_alloc` command is defined as follows:

```
cpp_srv_par_alloc arg_or_op
```

The `cpp_srv_par_alloc` command can take either one or three arguments.

- With one argument, the `cpp_srv_par_alloc` command allocates memory, if necessary, for an operation's `out` parameter or return value:

  `cpp_srv_par_alloc` *`arg_or_op`*

- With three arguments the `cpp_srv_par_alloc` command allocates memory for the return value of an attribute's accessor function:

  `cpp_srv_par_alloc` *`name type direction`*

  The *`direction`* argument must be equal to `return` in this case.

This convention of replacing `arg_or_op` with several arguments is also used in the other commands for server-side processing of parameters. Thus, the full set of commands for processing an attribute's implicit parameter and return value is:

```
cpp_srv_ret_decl name type ?alloc_mem?
cpp_srv_par_alloc name type direction
cpp_srv_par_ref name type direction
cpp_srv_free_mem_stmt name type direction
cpp_srv_need_to_free_mem type direction
```

It also applies to the `gen_` counterparts:

```
cpp_gen_srv_ret_decl name type ind_lev ?alloc_mem?
cpp_gen_srv_par_alloc name type direction ind_lev
cpp_gen_srv_free_mem_stmt name type direction ind_lev
```

Instance Variables and Local Variables

Previous sections show how to process variables used for parameters and an operation's return value. However, not all variables are used as parameters. For example, a C++ class that implements an IDL interface might contain some instance variables that are not used as parameters; or the body of an operation might declare some local variables that are not used as parameters. This section discusses commands for processing such variables. The following commands are provided:

```
cpp_var_decl name type is_var
cpp_var_free_mem_stmt name type is_var
cpp_var_need_to_free_mem type is_var
```

The `cpp_var_decl` and `cpp_var_free_mem_stmt` commands have `gen_` counterparts:

```
cpp_gen_var_decl name type is_var ind_lev
cpp_gen_var_free_mem_stmt name type is_var ind_lev
```

The following example shows how to use these commands:

Example 10:

```
# Tcl
set is_var 0
set ind_lev 1
[***
void some_func()
{
	// Declare variables
***]
foreach type $type_list {
	set name  "my_[$type l_name]"
```

Example 10:

```
1           cpp_gen_var_decl $name $type $is_var $ind_lev
}
[***

        // Initialize variables
***]
foreach type $type_list {
        set name  "my_[$type l_name]"
        set value "other_[$type l_name]"
2           cpp_gen_assign_stmt $type $name $value $ind_lev 0
}
[***

        // Memory management
***]
foreach type $type_list {
        set name  "my_[$type l_name]"
3           cpp_gen_var_free_mem_stmt $name $type $is_var
   $ind_lev
}
[***
} // some_func()
***]
```

The Tcl script is explained as follows:

1. The `cpp_gen_var_decl` command returns a C++ variable declaration with the specified `name` and `type`. The boolean `is_var` argument (equal to `0`) determines that the variable is not declared as a `_var` (smart pointer).
2. An assignment statement is generated by the `cpp_gen_assign_stmt` command for variables of the given `$type`. The `$name` argument is put on the left-hand side of the generated assignment statement and the `$value` argument on the right-hand side. This command expects its second and third arguments to be references. The last argument, the `scope` flag, is a workaround for a bug in some C++ compilers; see “cpp_assign_stmt” on page 163 for details.
3. The `cpp_gen_var_free_mem_stmt` command generates a C++ statement to free memory for the variable with the specified `name` and `type`.

If the `type_list` variable contains the types `string`, `widget` (a struct) and `long_array`, the Tcl code generates the following C++ code:

```
// C++
void some_func()
{
        // Declare variables
        char *                  my_string;
        widget                  my_widget;
        long_array              my_long_array;
```

```
        // Initialize variables
        my_string = CORBA::string_dup(other_string);
        my_widget = other_widget;
        for (CORBA::ULong i1 = 0; i1 < 10; i1 ++) {
            my_long_array[i1] = other_long_array[i1];
        }

        // Memory management
        CORBA::string_free(my_string);
} // some_func()
```

The `cpp_gen_var_free_mem_stmt` command generates memory-freeing statements only for the `my_string` variable. The other variables are stack-allocated, so they do not require their memory to be freed. If you modify the Tcl code so that `is_var` is set to TRUE, `my_string`'s type changes from `char *` to `CORBA::String_var` and suppresses the memory-freeing statement for that variable.

Processing a Union

When generating C++ code to process an IDL `union`, it is common to use a C++ `switch` statement to process the different cases of the `union`: the `cpp_branch_case_s_label` and `cpp_branch_case_l_label` commands are used for this task. Sometimes you might want to process an IDL `union` with a different C++ construct, such as an `if-then-else` statement: the `cpp_branch_s_label` and `cpp_branch_l_label` commands are used for this task. Table 6 summarizes the commands used for generating `union` labels.

Table 6: *Commands for Generating Union Labels*

Command	Description
`cpp_branch_case_s_label` *`union_branch`*	Returns the string `"case` *`scoped_label`*`"`, where *`scoped_label`* is the scoped name of the given *`union_branch`*, or `"default"` for the default union branch.
`cpp_branch_case_l_label` *`union_branch`*	Returns the string `"case` *`local_label`*`"`, where *`local_label`* is the local name of the given *`union_branch`*, or `"default"` for the default union branch.
`cpp_branch_s_label` *`union_branch`*	Returns the string `"`*`scoped_label`*`"`, where *`scoped_label`* is the scoped name of the given *`union_branch`*, or `"default"` for the default union branch.
`cpp_branch_l_label` *`union_branch`*	Returns the string `"`*`local_label`*`"`, where *`local_label`* is the local name of the given *`union_branch`*, or `"default"` for the default union branch.

For example, given the following IDL:

```
// IDL
module m {
        enum colour {red, green, blue};

        union foo switch(colour) {
            case red:    long     a;
            case green:  string   b;
            default:     short    c;
        };
};
```

The following Tcl script generates a C++ `switch` statement to process the union:

Example 11:

```
# Tcl
...
set union [$idlgen(root) lookup "m::foo"]
[***
void some_func()
{
       switch(u._d()) {
***]
1 foreach branch [$union contents {union_branch}] {
        set name [cpp_l_name $branch]
2       set case_label [cpp_branch_case_s_label $branch]
[***
       @$case_label@:
           ... // process u.@$name@()
           break;
***]
}; # foreach
[***
       };
} // some_func()
***]
```

The Tcl script is explained as follows:

1. The `foreach` loop iterates over every branch of the given `union`.
2. The `cpp_branch_case_s_label` command generates the case label for the given `$branch` branch node. If `$branch` is the default branch, the command returns `"default"`.

The previous Tcl script generates the following C++ code:

```
// C++
void some_func()
{
switch(u._d()) {
case m::red:
        ... // process u.a()
        break;
case m::green:
        ... // process u.b()
        break;
```

```
default:
        ... // process u.c()
        break;
};
} // some_func()
```

The `cpp_branch_case_s_label` command works for all union discriminant types. For example, if the discriminant is a `long` type, this command returns a string of the form `case 42` (where `42` is the value of the case label); if the discriminant is type `char`, the command returns a string of the form `case 'a'`.

Processing an Array

Arrays are usually processed in C++ using a `for` loop to access each element in the array. For example, consider the following definition of an array:

```
// IDL
typedef long long_array[5][7];
```

Assume that two variables, `foo` and `bar`, are both `long_array` types. C++ code to perform an element-wise copy from `bar` into `foo` might be written as follows:

Example 12:

```
    // C++
    void some_func()
    {
1       CORBA::ULong                    i1;
        CORBA::ULong                    i2;
2       for (i1 = 0; i1 < 5; i1 ++) {
            for (i2 = 0; i2 < 7; i2 ++) {
3               foo[i1][i2] = bar[i1][i2];
4           }
        }
    }
```

To write a Tcl script to generate the above C++ code, you need Tcl commands that perform these tasks:

1. Declare index variables.
2. Generate the `for` loop's header.
3. Provide the index for each element of the array `"[i1][i2]"`.
4. Generate the `for` loop's footer.

The following commands provide these capabilities:

```
cpp_array_decl_index_vars arr pre ind_lev
cpp_array_for_loop_header arr pre ind_lev ?decl?
cpp_array_elem_index arr pre
cpp_array_for_loop_footer arr indent
```

These commands use the following conventions:

- *arr* denotes an `array` node in the parse tree.
- *pre* is the prefix to use when constructing the names of index variables. For example, the prefix `i` is used to get index variables called `i1` and `i2`.

- *ind_lev* is the indentation level at which the `for` loop is to be created. In the above C++ example, the `for` loop is indented one level from the left side of the page.

The following Tcl script generates the `for` loop shown earlier:

```
# Tcl
set typedef  [$idlgen(root) lookup "long_array"]
set a        [$typedef true_base_type]
set indent   [cpp_indent [$a num_dims]]
set index    [cpp_array_elem_index $a "i"]
[***
void some_func()
{
    @[cpp_array_decl_index_vars $a "i" 1]@

    @[cpp_array_for_loop_header $a "i" 1]@
    @$indent@foo@$index@ = bar@$index@;

    @[cpp_array_for_loop_footer $a 1]@
}
***]
```

The amount of indentation to use inside the body of the `for` loop is calculated by using the number of dimensions in the array as a parameter to the `cpp_indent` command.

The `cpp_array_for_loop_header` command takes a boolean parameter called `decl`, which has a default value of `0` (FALSE). If `decl` is set to TRUE, the index variables are declared inside the header of the `for` loop. Thus, functionally equivalent (but slightly shorter) C++ code can be written as follows:

```
// C++
void some_func()
{
    for (CORBA::Ulong i1 = 0; i1 < 5; i1 ++) {
        for (CORBA::Ulong i2 = 0; i2 < 7; i2 ++) {
            foo[i1][i2] = bar[i1][i2];
        }
    }
}
```

The Tcl script to generate this is also slightly shorter because it can omit the `cpp_array_decl_index_vars` command:

```
# Tcl
set typedef  [$idlgen(root) lookup "long_array"]
set a        [$typedef true_base_type]
set indent   [cpp_indent [$a num_dims]]
set index    [cpp_array_elem_index $a "i"]
[***
void some_func()
{
    @[cpp_array_for_loop_header $a "i" 1 1]@
    @$indent@foo@$index@ = bar@$index@;
    @[cpp_array_for_loop_footer $a 1]@
}
***]
```

For completeness, some of the array processing commands have gen_ counterparts:

```
cpp_gen_array_decl_index_vars arr pre ind_lev
cpp_gen_array_for_loop_header arr pre ind_lev ?decl?
cpp_gen_array_for_loop_footer arr indent
```

Processing an Any

The commands to process the any type divide into two categories, for value insertion and extraction. The following subsections discuss each category.

- "Inserting Values into an Any".
- "Extracting Values from an Any".

Inserting Values into an Any

The cpp_any_insert_stmt command generates code that inserts a value into an any:

```
cpp_any_insert_stmt type any_name value
```

This command returns the C++ statement that inserts the specified value of the specified type into the any called any_name. An example of its use is:

```
# Tcl
foreach type $type_list {
        set var_name my_[$type s_uname]
[***
@[cpp_any_insert_stmt $type "an_any" $var_name]@;
***]
}
```

If the type_list variable contains the types widget (a struct), boolean and long_array, the above Tcl code will generate the following:

```
// C++
an_any <<= my_widget;
an_any <<= CORBA::Any::from_boolean(my_boolean);
an_any <<= long_array_forany(my_long_array);
```

Extracting Values from an Any

The following sub-subsections describe commands that you can use in Tcl scripts to extract values from an any:

cpp_any_extract_var_decl

cpp_any_extract_var_decl *type name*

The cpp_any_extract_var_decl command declares a variable into which values from an any are extracted. The parameters to this command are the variable's *type* and *name*. If the value to extract is a simple type such as a short, long, or boolean, the variable is declared as a normal variable of the specified type. However, if the value is a complex type such as struct or sequence, the variable is declared as a pointer to the specified type.

cpp_any_extract_var_ref

cpp_any_extract_var_ref *type name*

The cpp_any_extract_var_ref command returns a reference to the value in *name* of the specified *type*. The returned reference is either $name or *$name, depending on how the variable is declared by the cpp_any_extract_var_decl command.

cpp_any_extract_stmt

cpp_any_extract_stmt *type any_name name*

The cpp_any_extract_stmt command extracts a value of the specified type from the any called *any_name* into the variable *name*.

The following example shows how to use these commands:

```
# Tcl
foreach type $type_list {
    set var_name my_[$type s_uname]
[***
@[cpp_any_extract_var_decl $type $var_name]@;
***]
}
output "\n"
foreach type $type_list {
    set var_name my_[$type s_uname]
    set var_ref  [cpp_any_extract_var_ref $type $var_name]
[***
if (@[cpp_any_extract_stmt $type "an_any" $var_name]@) {
    process_@[$type s_uname]@(@$var_ref@);
}
***]
}
```

If the variable type_list contains the widget (a struct), boolean and long_array types then the above Tcl code generates the following C++:

```
// C++
widget * my_widget;
CORBA::Boolean my_boolean;
long_array_slice* my_long_array;

if (an_any >>= my_widget) {
        process_widget(*my_widget);
}
if (an_any >>= CORBA::Any::to_boolean(my_boolean)) {
        process_boolean(my_boolean);
}
if (an_any >>= long_array_forany(my_long_array)) {
        process_long_array(my_long_array);
}
```

Developing a Java Genie

The `std/java_poa_lib.tcl` *file is a library of Tcl command procedures that map IDL constructs into their Java counterparts. The server-side IDL-to-Java mapping is based on the CORBA Portable Object Adapter specification.*

Identifiers and Keywords

There are a number of commands that help map IDL data types to their Java equivalents.

The CORBA mapping generally maps IDL identifiers to the same identifier in Java, but there are some exceptions required to avoid clashes. For example, if an IDL identifier clashes with a Java keyword, it is mapped to an identifier with the prefix _.

Consider the following unusual, but valid, interface:

```
// IDL
interface Strange {
    string for( in long while );
};
```

The `for()` operation maps to a Java method with the following signature:

```
// Java
public java.lang.String Strange._for(int _while);
```

Note: Avoid IDL identifiers that clash with keywords in Java or other programming languages that you use to implement CORBA objects. Although they can be mapped as described, it causes confusion.

Another type of identifier clash arises because the IDL-to-Java mapping uses the convention of appending suffixes to type names, to form new class identifiers. For example, an IDL type called `Foo` maps to the Java `Foo` class and the associated `FooHelper` and `FooHolder` classes. A potential clash might occur if an IDL identifier happens to end in `Helper` or `Holder`. Consider the following IDL:

```
//IDL
interface Foo { ... };
interface FooHelper { ... };
interface FooHolder { ... };
```

Potential conflicts are avoided by prefixing an _ (underscore) character to the Java classes associated with the `FooHelper` and `FooHolder` IDL types. The `Foo`, `FooHelper` and `FooHolder` IDL interfaces are mapped to Java as shown in Table 7:

Table 7: *Naming Convention for IDL Identifiers Ending in Helper or Holder*

IDL Type	Java Class	Java Helper Class	Java Holder Class
Foo	Foo	FooHelper	FooHolder
FooHelper	_FooHelper	_FooHelperHelper	_FooHelperHolder
FooHolder	_FooHolder	_FooHolderHelper	_FooHolderHolder

The application programming interface (API) for generating Java identifiers is summarized in Table 8. The `_s_` variants return fully-scoped identifiers whereas the `_l_` variants return non-scoped identifiers.

Table 8: *Commands for Generating Identifiers and Keywords*

Command	Description
`java_s_name` *node*	Returns the Java mapping of a node's scoped name.
`java_l_name` *node*	Returns the Java mapping of a node's local name.
`java_typecode_s_name` *type*	Returns the scoped Java name of the type code for *type*.
`java_typecode_l_name` *type*	Returns the local Java name of the type code for *type*.
`java_helper_name` *type*	Returns the scoped name of the `Helper` class associated with *type*.
`java_holder_name` *type*	Returns the scoped name of the `Holder` class associated with *type*.

Java Prototype

A typical approach to developing a Java genie is to start with a working Java example. This Java example should exhibit most of the features that you want to incorporate into your generated code. You can then proceed by reverse-engineering the Java example; developing a Tcl script that recreates the Java example when it receives the corresponding IDL file as input.

The Java example employed to help you develop the Tcl script is referred to here as a *Java prototype*. In the following sections, two fundamental Java prototypes are presented and analyzed in detail.

- The first Java prototype demonstrates how to invoke a typical CORBA method (client-side prototype).
- The second Java prototype demonstrates how to implement a typical CORBA method (server-side prototype).

The script derived from these fundamental Java prototypes can serve as a starting point for a wide range of applications, including the automated generation of wrapping code for legacy systems.

The Java prototypes described in this chapter use the following IDL:

```
// IDL
// File: 'prototype.idl'
struct  widget          {long a;};
typedef sequence<long>   longSeq;
typedef long             long_array[10];

interface foo {
    longSeq op(
        in widget       p_widget,
        inout string    p_string,
        out longSeq     p_longSeq,
        out long_array  p_long_array);
};
```

Client-Side Prototype

The client-side prototype demonstrates a CORBA invocation of the `foo::op()` IDL operation. Parameters are allocated, a `foo::op()` invocation is made, and the parameters are freed at the end.

```
// Java
//--------
// Declare parameters for operation
//--------
Prototype.widget                 p_widget;
org.omg.CORBA.StringHolder       p_string;
Prototype.longSeq                p_longSeq;
Prototype.long_array             p_long_array;
int[]                            _result;

//--------
// Allocate Holder Object for "inout" and "out" Parameters
//--------
p_string = new org.omg.CORBA.StringHolder();
p_longSeq = new Prototype.longSeqHolder();
p_long_array = new Prototype.long_arrayHolder();

//--------
// Initialize "in" and "inout" parameters
//--------
p_widget = other_widget;
p_string.value = other_string;
```

```
//--------
// Invoke the operation
//--------
try {
    _result = obj.op(
                    p_widget,
                    p_string,
                    p_longSeq,
                    p_long_array);
} catch(Exception ex) {
    ... // handle the exception
}

//--------
// Process the returned parameters
//--------
process_string(p_string.value);
process_longSeq(p_longSeq.value);
process_long_array(p_long_array.value);
process_longSeq(_result);
```

Server-Side Prototype

The server-side prototype shows a sample implementation of the `foo::op()` IDL operation. This operation demonstrates the use of `in`, `inout` and `out` parameters. It also has a return value. The code shown in the implementation deals with deallocation, allocation and initialization of parameters and return values.

```
// Java
   public int[] op(
            Prototype.widget                p_widget,
            org.omg.CORBA.StringHolder      p_string,
            Prototype.longSeqHolder         p_longSeq,
            Prototype.long_arrayHolder      p_long_array
   )
   {
        //--------
        // Process 'in' and 'inout' parameters
        //--------
        process_widget(p_widget);
        process_string(p_string.value);

        //--------
        // Declare a variable to hold the return value.
        //--------
        int[]                           _result;

        //--------
        // Assign new values to "inout" and "out"
        // parameters, and the return value, if needed.
        //--------
        p_string.value = other_string;
        p_longSeq.value = other_longSeq;
        p_long_array.value = other_long_array;
        _result = other_longSeq;
        return _result;
    }
```

Invoking an Operation

This section explains how to generate Java code that invokes a given IDL operation. The process of making a CORBA invocation in Java can be broken down into the following steps:

1. Declare variables to hold parameters and return value.

   The calling code must declare all `in`, `inout`, and `out` parameters before making the invocation. If the return type of the operation is non-`void`, a return value must also be declared.
2. Allocate `Holder` objects for `inout` and `out` parameters.
3. Initialize input parameters.

   The calling code must initialize all `in` and `inout` parameters. There is no need to initialize `out` parameters.
4. Invoke the IDL operation.

   The calling code invokes the operation, passing each of the prepared parameters and retrieving the return value (if any).
5. Process output parameters and return value.

   Assuming no exception has been thrown, the caller processes the returned `inout`, `out`, and return values.

The following subsections give a detailed example of how to generate complete code for an IDL operation invocation.

Step 1—Declare Variables to Hold Parameters and Return Value

The Tcl script below illustrates how to declare Java variables to be used as parameters to (and the return value of) an operation call:

Example 13:

```
# Tcl
smart_source "std/output.tcl"
smart_source "std/java_poa_lib.tcl"

if { ! [idlgen_parse_idl_file "prototype.idl"] } {
    exit 1
}
idlgen_set_preferences $idlgen(cfg)
1 set pref(java_genie,package_name) "Prototype"

open_output_file "testClt.java"

set op         [$idlgen(root) lookup "foo::op"]
set ind_lev    2
set arg_list   [$op contents {argument}]
[***
    //--------
    // Declare parameters for operation
    //--------
***]
foreach arg $arg_list {
2   java_gen_clt_par_decl $arg $ind_lev
}
3 java_gen_clt_par_decl $op $ind_lev
```

The Tcl script is explained as follows:

1. Set the pref(java_genie,package_name) array element equal to the name of the Java package that contains the generated code.
2. When an *argument* node appears as the first parameter of java_gen_clt_par_decl, the command outputs a declaration of the corresponding Java parameter.
3. When an *operation* node appears as the first parameter of java_gen_clt_par_decl, the command outputs a declaration of a variable to hold the operation's return value. If the operation has no return value, the command outputs a blank string.

The previous Tcl script generates the following Java code:

```
//Java
//--------
// Declare parameters for operation
//--------
Prototype.widget                  p_widget;
org.omg.CORBA.StringHolder        p_string;
Prototype.longSeqHolder           p_longSeq;
Prototype.long_arrayHolder        p_long_array;
int[]                             _result;
```

The $pref(java,ret_param_name) array element determines the name of the Java variable that is declared to hold the return value. Its default value is _result.

Step 2—Allocate Holder Objects for inout and out Parameters

The following Tcl script shows how to allocate Holder objects for the inout and out parameters:

Example 14:

```
#Tcl
[***

    //--------
    // Allocate Holder objects for "inout" and "out" Parameters
    //--------
***]

1 foreach arg [$op args {inout out}] {
    set arg_name [java_l_name $arg]
    set type [$arg type]
    set dir      [$arg direction]

2   output "    [java_var_alloc_mem $arg_name $type $dir];
   \n"
}
```

The Tcl script is explained as follows:

1. The `foreach` loop iterates over all the `inout` and `out` parameters.
2. The `java_var_alloc_mem` command generates a statement that initializes the `$arg_name` variable with a `Holder` object of `$type` type.

The previous Tcl script generates the following Java code:

```
//Java
//--------
// Allocate Holder objects for "inout" and "out"
   Parameters
//--------
p_string = new org.omg.CORBA.StringHolder();
p_longSeq = new Prototype.longSeqHolder();
p_long_array = new Prototype.long_arrayHolder();
```

Step 3—Initialize Input Parameters

The following Tcl script shows how to initialize `in` and `inout` parameters:

Example 15:

```
# Tcl
[***
    //--------
    // Initialize "in" and "inout" parameters
    //--------
***]
1 foreach arg [$op args {in inout}] {
    set arg_name [java_l_name $arg]
    set type [$arg type]
    set dir      [$arg direction]
    set value "other_[$type s_uname]"
2   java_gen_assign_stmt $type $arg_name $value $ind_lev
   $dir
}
```

The Tcl script is explained as follows:

1. The `foreach` loop iterates over all the `in` and `inout` parameters.
2. An assignment statement is generated by the `java_gen_assign_stmt` command for variables of the given `$type`. The `$arg_ref` argument is put on the left-hand side of the generated assignment statement and the `$value` argument on the right-hand side.

The previous Tcl script generates the following Java code:

```
// Java
//--------
// Initialize "in" and "inout" parameters
//--------
p_widget = other_widget;
p_string.value = other_string;
```

Step 4—Invoke the IDL Operation

The following Tcl script shows how to invoke an IDL operation, pass parameters, and assign the return value to a variable:

Example 16:

```
1 # Tcl
set ret_assign    [java_ret_assign $op]
set op_name       [java_l_name $op]
set start_str     "\n\t\t\t"
set sep_str       ",\n\t\t\t"
2 set call_args [idlgen_process_list $arg_list \
                        java_l_name $start_str $sep_str]
[***
    //--------
    // Invoke the operation
    //--------
    try {
        @$ret_assign@obj.@$op_name@(@$call_args@);
    } catch(Exception ex) {
        ... // handle the exception
    }
***]
```

The Tcl script is explained as follows:

1. The `[java_ret_assign $op]` expression returns the `"_result ="` string. If the operation invoked does not have a return type, it returns an empty string, `""`.
2. The parameters to the operation call are formatted using the command `idlgen_process_list`. For more about this command, “idlgen_process_list” on page 140.

The previous Tcl script generates the following Java code:

```
//Java
//--------
// Invoke the operation
//--------
try {
    _result = obj.op(
                    p_widget,
                    p_string,
                    p_longSeq,
                    p_long_array);
}
 catch(Exception ex) {
    ... // handle the exception
}
```

Step 5—Process Output Parameters and Return Value

The techniques used to process output parameters are similar to those used to process input parameters, as in the following Tcl script:

Example 17:

```
# Tcl
[***
    //--------
    // Process the returned parameters
    //--------
***]
1 foreach arg [$op args {out inout}] {
     set type [$arg type]
     set name [java_l_name $arg]
     set dir  [$arg direction]
2    set arg_ref [java_clt_par_ref $arg]
[***
    process_@[$type s_uname]@(@$arg_ref@);
***]
}
set ret_type [$op return_type]
set name     [java_l_name $arg]
if {[$ret_type l_name] != "void"} {
3    set ret_ref [java_clt_par_ref $op]
[***
    process_@[$ret_type s_uname]@(@$ret_ref@);
***]
}

close_output_file
```

The Tcl script is explained as follows:

1. The `foreach` loop iterates over all the `out` and `inout` parameters.
2. The `java_clt_par_ref` command returns a reference to the Java parameter corresponding to the given argument node `$arg`.
3. When an operation node `$op` is supplied as the first parameter to `java_clt_par_ref`, the command returns a reference to the return value of the operation.

The previous Tcl script generates the following Java code:

```
//Java
//--------
// Process the returned parameters
//--------
process_string(p_string.value);
process_longSeq(p_longSeq.value);
process_long_array(p_long_array.value);
process_longSeq(_result);
```

Invoking an Attribute

To invoke an IDL attribute, you must perform similar steps to those described in “Invoking an Operation” on page 87. However, a different form of the client-side Tcl commands are used:

```
java_clt_par_decl name type dir
java_clt_par_ref name type dir
```

Similar variants are available for the `gen_` counterparts of commands:

```
java_gen_clt_par_decl name type dir ind_lev
```

These commands are the same as the set of commands used to generate an operation invocation, except they take a different set of arguments. You specify the *name* and *type* of the attribute as the first two arguments. The *dir* argument can be `in` or `return`, indicating an attribute’s modifier or accessor, respectively. The *ind_level* argument has the same effect as in “Step 1—Declare Variables to Hold Parameters and Return Value” on page 87.

Implementing an Operation

This section explains how to generate Java code that provides the implementation of an IDL operation. The steps are:

1. Generate the operation signature.
2. Process input parameters.

   The method body first processes the `in` and `inout` parameters that it has received from the client.
3. Declare the return value.
4. Initialize output parameters and return value.

   The `inout` and `out` parameters and the return value must be initialized.

Step 1—Generate the Operation Signature

The java_gen_op_sig command generates a signature for the Java method that implements an IDL operation.

The following Tcl script generates the signature for the implementation of the foo::op operation:

```
# Tcl
smart_source "std/output.tcl"
smart_source "std/java_poa_lib.tcl"

idlgen_set_preferences $idlgen(cfg)
set pref(java_genie,package_name) "Prototype"

if { ! [idlgen_parse_idl_file "prototype.idl"] } {
    exit 1
}

open_output_file "testSrv.java"
set op [$idlgen(root) lookup "foo::op"]
java_gen_op_sig $op
...
```

The previous Tcl script generates the following Java code:

```
// Java
   public int[] op(
            Prototype.widget                p_widget,
            org.omg.CORBA.StringHolder      p_string,
            Prototype.longSeqHolder         p_longSeq,
            Prototype.long_arrayHolder      p_long_array
   )
   throws org.omg.CORBA.SystemException
```

The names of the Java parameters are the same as the parameter names declared in IDL.

Step 2—Process Input Parameters

This step is similar to “Step 5—Process Output Parameters and Return Value” on page 91. It is, therefore, not described in this subsection.

Step 3—Declare the Return Value

The following Tcl script declares a local variable that can hold the return value of the operation:

Example 18:

```
# Tcl
...
set op         [$idlgen(root) lookup "foo::op"]
set ret_type   [$op return_type]
set ind_lev    3
set arg_list   [$op contents {argument}]
if {[$ret_type l_name] != "void"} {
    set type [$op return_type]
    set ret_ref [java_srv_par_ref $op]
[***
       //--------
       // Declare a variable to hold the return value.
       //--------
1      @[java_srv_ret_decl $ret_ref $type]@;

***]
}
```

1. The java_srv_ret_decl command returns a statement that declares the return value of the operation. The first argument is the name of the operation node. The second argument is the type of the return value.

The output of the above Tcl is as follows:

```
//Java
//--------
// Declare a variable to hold the return value.
//--------
int[]                           _result;
```

Step 4—Initialize Output Parameters and the Return Value

The following Tcl script iterates over all inout and out parameters and, if needed, the return value, and assigns values to them:

Example 19:

```
# Tcl
[***
       //--------
       // Assign new values to "out" and "inout"
       // parameters, and the return value, if needed.
       //--------
***]
foreach arg [$op args {inout out}] {
        set type    [$arg type]
1       set arg_ref [java_srv_par_ref $arg]
        set name2   "other_[$type s_uname]"
[***
       @$arg_ref@ = @$name2@;
***]
}
if {[$ret_type l_name] != "void"} {
```

Example 19:

2
```
        set ret_ref [java_srv_par_ref $op]
        set name2   "other_[$ret_type s_uname]"
[***
       @$ret_ref@ = @$name2@;
       return @$ret_ref@;
***]
}
```

The Tcl script is explained as follows:

1. The `java_srv_par_ref` command returns a reference to the Java parameter corresponding to the `$arg` argument node. If the argument is an `inout` or `out` parameter the reference is of the form *ArgName*`.value`, as is appropriate for assignment to `Holder` types.
2. When the `$op` operation node is supplied as the first argument to the `java_srv_par_ref` command, it returns a reference to the operation's return value.

The previous Tcl script generates the following Java code:

```
//Java
//--------
// Assign new values to "out" and "inout"
// parameters, and the return value, if needed.
//--------
p_string.value = other_string;
p_longSeq.value = other_longSeq;
p_long_array.value = other_long_array;
_result = other_longSeq;
return _result;
```

Implementing an Attribute

The `java_srv_par_alloc` command is defined as follows:

```
java_srv_par_alloc arg_or_op
```

The `java_srv_par_alloc` command can take either one or three arguments.

- With one argument, the `java_srv_par_alloc` command allocates memory, if necessary, for an operation's `out` parameter or return value:

  `java_srv_par_alloc` *`arg_or_op`*
- With three arguments the `java_srv_par_alloc` command can allocate memory for the return value of an attribute's accessor method:

  `java_srv_par_alloc` *`name type direction`*

  The *`direction`* attribute must be set equal to `return` in this case.

This convention of replacing *arg_or_op* with several arguments is also used in the other commands for server-side processing of parameters. Thus, the full set of commands for processing an attribute's implicit parameter and return value is:

```
java_srv_ret_decl name type ?alloc_mem?
java_srv_par_alloc name type direction
java_srv_par_ref name type direction
```

It also applies to the `gen_` counterparts:

```
java_gen_srv_ret_decl name type ind_lev ?alloc_mem?
java_gen_srv_par_alloc name type direction ind_lev
```

Instance Variables and Local Variables

Previous subsections show how to process variables used for parameters and an operation's return value. However, not all variables are used as parameters. For example, a Java class that implements an IDL interface might contain some instance variables that are not used as parameters; or the body of an operation might declare some local variables that are not used as parameters. This section discusses commands for processing such variables. The following command is provided:

```
java_var_decl name type direction
```

The `java_var_decl` command has a `gen_` counterpart:

```
java_gen_var_decl name type direction ind_lev
```

The following example shows how to use these commands:

Example 20:

```
# Tcl
smart_source "std/output.tcl"
smart_source "std/java_poa_lib.tcl"

if { ! [idlgen_parse_idl_file "prototype.idl"] } {
    exit 1
}
idlgen_set_preferences $idlgen(cfg)
open_output_file "variables.java"

lappend type_list [$idlgen(root) lookup string]
lappend type_list [$idlgen(root) lookup widget]
lappend type_list [$idlgen(root) lookup long_array]

set ind_lev 1
[***
void some_func()
{
       // Declare variables
***]
foreach type $type_list {
        set name  "my_[$type l_name]"
```

Example 20:

```
1         java_gen_var_decl $name $type "in" $ind_lev
}
[***

        // Initialize variables
***]
foreach type $type_list {
        set name  "my_[$type l_name]"
        set value "other_[$type l_name]"
2         java_gen_assign_stmt $type $name $value $ind_lev
   "in"
}
[***
} // some_func()
***]

close_output_file
```

The Tcl script is explained as follows:

1. The `java_gen_var_decl` command returns a Java variable declaration with the specified `name` and `type`. The `"in"` argument specifies the direction of the variable, as if it was a parameter. If the direction is `"out"` or `"inout"` a `Holder` type is declared.
2. An assignment statement is generated by the `java_gen_assign_stmt` command for variables of the given `$type`. The `$name` argument is put on the left-hand side of the generated assignment statement and the `$value` argument on the right-hand side.

If the `type_list` variable contains the `string`, `widget` (a struct) and `long_array` types, the Tcl code generates the following Java code:

```
// Java
void some_func()
{
    // Declare variables
    java.lang.String                my_string;
    NoPackage.widget                my_widget;
    int[]                           my_long_array;

    // Initialize variables
    my_string = other_string;
    my_widget = other_widget;
    {
        for (int i1 = 0; i1 < 10  ; i1 ++) {
            my_long_array[i1] = other_long_array[i1];
        }
    }
} // some_func()
```

Processing a Union

When generating Java code to process an IDL `union`, it is common to use a Java `switch` statement to process the different cases of the `union`: the `java_branch_case_s_label` command is used for this task. Sometimes you might want to process an IDL `union` with a different Java construct, such as an `if-then-else` statement: the `java_branch_l_label` command is used for this task. Table 9 summarizes the commands used for generating `union` labels.

Table 9: *Commands for Generating Union Labels*

Command	Description
`java_branch_case_l_label` *`union_branch`*	Returns the `"case` *`local_label`*`"` string, where *`local_label`* is the local label of the *`union_branch`*, or `"default"`, for the default union branch.
`java_branch_case_s_label` *`union_branch`*	Returns the `"case` *`scoped_label`*`"` string, where *`scoped_label`* is the scoped label of the *`union_branch`*, or `"default"`, for the default union branch.
`java_branch_l_label` *`union_branch`*	Returns the `"`*`local_label`*`"` string, where *`local_label`* is the local label of the given *`union_branch`*, or `"default"`, for the default union branch.
`java_branch_s_label` *`union_branch`*	Returns the `"`*`scoped_label`*`"` string, where *`scoped_label`* is the scoped label of the given *`union_branch`*, or `"default"`, for the default union branch.

For example, given the following IDL:

```
// IDL
module m {
        enum colour {red, green, blue};

        union foo switch(colour) {
            case red:    long     a;
            case green:  string   b;
            default:     short    c;
        };
};
```

The following Tcl script generates a Java `switch` statement to process the union:

Example 21:

```
# Tcl
smart_source "std/output.tcl"
smart_source "std/java_poa_lib.tcl"

if { ! [idlgen_parse_idl_file "union.idl"] } {
    exit 1
}
idlgen_set_preferences $idlgen(cfg)

open_output_file "union.java"

set union [$idlgen(root) lookup "m::foo"]
[***
void some_func()
{
        //...
       switch(u.discriminator().value()) {
***]
1 foreach branch [$union contents {union_branch}] {
        set name [java_l_name $branch]
2       set case_label [java_branch_case_s_label $branch]
[***
       @$case_label@:
           ... // process u.@$name@()
           break;
***]
}; # foreach
[***
       };
} // some_func()
***]
close_output_file
```

The Tcl script is explained as follows:

1. The `foreach` loop iterates over every branch of the given `union`.
2. The `java_branch_case_s_label` command generates the case label for the given `$branch` branch node. If `$branch` is the default branch, the command returns `"default"`.

This Tcl script generates the following Java code:

```
// Java
void some_func()
{
        //...
        switch(u.discriminator().value()) {
        case NoPackage.m.colour._red:
            ... // process u.a()
            break;
        case NoPackage.m.colour._green:
            ... // process u.b()
            break;
        default:
            ... // process u.c()
            break;
        };
} // some_func()
```

Case labels are generated in the form `NoPackage.m.colour._red`, of integer type, instead of `NoPackage.m.colour.red`, of `NoPackage.m.colour` type, because an integer type must be used in the branches of the switch statement.

The `java_branch_case_s_label` command works for all union discriminant types. For example, if the discriminant is a `long` type, the command returns a string of the form `case 42` (where `42` is the value of the case label); if the discriminant is type `char`, the command returns a string of the form `case 'a'`.

Processing an Array

Arrays are usually processed in Java using a `for` loop to access each element in the array. For example, consider the following definition of an array:

```
// IDL
typedef long long_array[5][7];
```

Assume that two variables, `foo` and `bar`, are both `long_array` types. Java code to perform an element-wise copy from `bar` into `foo` might be written as follows:

Example 22:

```
// Java
void some_method()
{
1       int                             i1;
        int                             i2;

2       for (i1 = 0; i1 < 5  ; i1 ++) {
            for (i2 = 0; i2 < 7  ; i2 ++) {
3               foo[i1][i2] = bar[i1][i2];
4           }
        }
}
```

To write a Tcl script to generate the above Java code, you need Tcl commands that perform the following tasks:

1. Declare index variables.
2. Generate the `for` loop's header.
3. Provide the index for each element of the array `"[i1][i2]"`.
4. Generate the `for` loop's footer.

The following commands provide these capabilities:

```
java_array_decl_index_vars arr pre ind_lev
java_array_for_loop_header arr pre ind_lev ?decl?
java_array_elem_index arr pre
java_array_for_loop_footer arr ind_lev
```

These commands use the following conventions:

- *`arr`* denotes an `array` node in the parse tree.
- *`pre`* is the prefix to use when constructing the names of index variables. For example, the prefix `i` is used to get index variables called `i1` and `i2`.
- *`ind_lev`* is the indentation level at which the `for` loop is to be created. In the above Java example, the `for` loop is indented one level from the left side of the page.

The following Tcl script generates the `for` loop shown earlier:

```
# Tcl
smart_source "std/output.tcl"
smart_source "std/java_poa_lib.tcl"

if { ! [idlgen_parse_idl_file "array.idl"] } {
    exit 1
}
idlgen_set_preferences $idlgen(cfg)
open_output_file "array.java"

set typedef  [$idlgen(root) lookup "long_array"]
set a        [$typedef true_base_type]
set indent   [java_indent [$a num_dims]]
set index    [java_array_elem_index $a "i"]
[***
void some_method()
{
   @[java_array_decl_index_vars $a "i" 1]@

   @[java_array_for_loop_header $a "i" 1]@
   @$indent@foo@$index@ = bar@$index@;
   @[java_array_for_loop_footer $a 1]@
}
***]

close_output_file
```

The amount of indentation to use inside the body of the `for` loop is calculated by using the number of dimensions in the array as a parameter to the `java_indent` command.

The `java_array_for_loop_header` command takes a boolean parameter called `decl`, which has a default value of `0` (`FALSE`). If `decl` is set to `1` (`TRUE`), the index variables are declared inside the header of the `for` loop.

Functionally equivalent (but slightly shorter) Java code can be written as follows:

```
// Java
void some_method()
{
    for (int i1 = 0; i1 < 5  ; i1 ++) {
        for (int i2 = 0; i2 < 7  ; i2 ++) {
            foo[i1][i2] = bar[i1][i2];
        }
    }
}
```

The Tcl script to generate this is also slightly shorter, because it can omit the `java_array_decl_index_vars` command:

```
# Tcl
...
set typedef  [$idlgen(root) lookup "long_array"]
set a        [$typedef true_base_type]
set indent   [java_indent [$a num_dims]]
set index    [java_array_elem_index $a "i"]
[***
void some_method()
{
    @[java_array_for_loop_header $a "i" 1 1]@
    @$indent@foo@$index@ = bar@$index@;
    @[java_array_for_loop_footer $a 1]@
}
***]
```

For completeness, some of the array processing commands have `gen_` counterparts:

```
java_gen_array_decl_index_vars arr pre ind_lev
java_gen_array_for_loop_header arr pre ind_lev ?decl?
java_gen_array_for_loop_footer arr indent
```

Processing a Sequence

Because sequences map to Java arrays, they are processed in a similar way to IDL `array` types. The following commands are provided for processing sequences:

```
java_sequence_for_loop_header seq pre ind_lev ?decl?
java_sequence_elem_index seq pre
java_sequence_for_loop_footer seq ind_lev
```

The command parameters are:

- *seq* denotes a `sequence` node in the parse tree.
- *pre* is the prefix to use when constructing the names of index variables. For example, the prefix `i` is used to get index variables called `i1` and `i2`.

- *ind_lev* is the indentation level at which the `for` loop is to be created.
- *decl* is a flag that causes loop indices to be declared in the `for` loop header when equal to `1` (`TRUE`). No indices are declared when *decl* is equal to `0` (`FALSE`).

These commands are used in an similar way to the array commands.

Processing an Any

The commands to process the `any` type divide into two categories, for value insertion and extraction. The following subsections discuss each category.

- "Inserting Values into an Any"
- "Extracting Values from an Any"

Inserting Values into an Any

Table 10 summarizes the command that is used to generate code that inserts values into an `any`.

Table 10: *Command for Generating any Insertion Statements*

Command	Description
`java_any_insert_stmt` *type any_name value*	Returns a Java statement that inserts the *value* variable of the specified *type* into the `any` called *any_name*.

The following example Tcl script shows how to use this command:

```
# Tcl
smart_source "std/output.tcl"
smart_source "std/java_poa_lib.tcl"

if { ! [idlgen_parse_idl_file "prototype.idl"] } {
    exit 1
}
idlgen_set_preferences $idlgen(cfg)
open_output_file "any_insert.java"

lappend type_list [$idlgen(root) lookup widget]
lappend type_list [$idlgen(root) lookup boolean]
lappend type_list [$idlgen(root) lookup long_array]

foreach type $type_list {
        set var_name my_[$type s_uname]
[***
    @[java_any_insert_stmt $type "an_any" $var_name]@;
***]
}
close_output_file
```

If the `type_list` variable contains the `widget` (a struct), `boolean` and `long_array` types, the above Tcl code generates the following:

```
// Java
NoPackage.widgetHelper.insert(an_any,my_widget);
an_any.insert_boolean(my_boolean);
NoPackage.long_arrayHelper.insert(an_any,my_long_array);
```

Extracting Values from an Any

Table 11 summarizes the commands that are used to generate code that extracts values from an `any`.

Table 11: *Commands for Generating any Extraction Statements*

Command	Description
`java_any_extract_var_decl` *`type name`*	Declares a variable called *`name`*, of the specified *`type`*, into which an `any` value can be extracted.
`java_any_extract_var_ref` *`type name`*	Returns a reference to the variable called *`name`* of the specified *`type`*.
`java_any_extract_stmt` *`type any_name name`*	Extracts a value of the specified `type` from the `any` called *`any_name`* into the variable *`name`*.

The following example Tcl script shows how to use these commands:

```
# Tcl
smart_source "std/output.tcl"
smart_source "std/java_poa_lib.tcl"

if { ! [idlgen_parse_idl_file "prototype.idl"] } {
    exit 1
}
idlgen_set_preferences $idlgen(cfg)
open_output_file "any_extract.java"

lappend type_list [$idlgen(root) lookup widget]
lappend type_list [$idlgen(root) lookup boolean]
lappend type_list [$idlgen(root) lookup long_array]

[***
try {
***]
foreach type $type_list {
    set var_name my_[$type s_uname]
[***
   @[java_any_extract_var_decl $type $var_name]@;
***]
}
output "\n"
foreach type $type_list {
    set var_name my_[$type s_uname]
    set var_ref  [java_any_extract_var_ref $type $var_name]
```

```
[***
    @[java_any_extract_stmt $type "an_any" $var_name]@
    process_@[$type s_uname]@(@$var_ref@);

***]
}
[***
}
catch(Exception e){
    System.out.println("Error: extract from any.");
    e.printStackTrace();
};
***]

close_output_file
```

If the variable `type_list` contains the `widget` (a `struct`), `boolean` and `long_array` types, the above Tcl code generates the following Java code:

```
// Java
try {
    NoPackage.widget                my_widget;
    boolean                         my_boolean;
    int[]                           my_long_array;

    my_widget = NoPackage.widgetHelper.extract(an_any)
    process_widget(my_widget);

    my_boolean = an_any.extract_boolean()
    process_boolean(my_boolean);

    my_long_array =
   NoPackage.long_arrayHelper.extract(an_any)
    process_long_array(my_long_array);

}
catch(Exception e){
    System.out.println("Error: extract from any.");
    e.printStackTrace();
};
```

Using the C++ Print and Random Utility Libraries

This chapter shows how to use the cpp_poa_print and cpp_poa_random libraries, using some example Tcl scripts.

Two additional genies, `cpp_poa_print.tcl` and `cpp_poa_random.tcl` are provided with the code generation toolkit: the `cpp_poa_print.tcl` genie, to generate code that prints out CORBA data types and the `cpp_poa_random.tcl` genie, to generate code that creates random values for CORBA data types. The genies are discussed in the *CORBA Programmer's Guide*.

Associated with these genies are two libraries—the print and random utility libraries—that can be used in your own Tcl scripts to generate print statements or to initialize variables with random data.

Sample IDL for Examples

The examples in this chapter use the following sample IDL:

```
// IDL
struct  widget            {long a;};
typedef sequence<long>    longSeq;
typedef long              long_array[10];
interface foo {
    longSeq op(
        in widget       p_widget,
        inout string    p_string,
        out longSeq     p_longSeq,
        out long_array  p_long_array);
};
```

The cpp_poa_print Utility Library

The minimal API of the `cpp_poa_print` library is made available by the following command:

```
smart_source "cpp_poa_print/lib-min.tcl"
```

The minimal API defines the following commands:

```
cpp_gen_print_stmt type name ?indent? ?ostream?
cpp_print_delete ?printer?
cpp_print_func_name type
cpp_print_gen_init ?orb?
cpp_print_stmt type name ?indent? ?ostream?
```

See “cpp_poa_print Commands” on page 207 for details.

For access to the full API of the `cpp_poa_print` library, use the following command:

```
smart_source "cpp_poa_print/lib-full.tcl"
```

The full library includes commands from the minimal library and the following commands:

```
gen_cpp_print_funcs_h
gen_cpp_print_funcs_cc ?ignored?
```

These commands generate the `it_print_funcs.h` and `it_print_funcs.cc` files, respectively. See “cpp_poa_print Commands” on page 207 for details.

Example Script

The following script shows how to use the commands in the `cpp_poa_print` library.

Example 23:

```
  # Tcl
  smart_source "std/sbs_output.tcl"
  smart_source "std/cpp_poa_lib.tcl"
1 smart_source "cpp_poa_print/lib-full.tcl"

  if {$argc != 1}  {
          puts "usage: ..."; exit 1
  }
  set file [lindex $argv 0]
  set ok [idlgen_parse_idl_file $file]
  if {!$ok} { exit }

2 #--------
  # Generate it_print_funcs.{h,cc}
  #--------
  gen_cpp_print_funcs_h
  gen_cpp_print_funcs_cc 1

  #--------
  # Generate a file which contains
  # calls to the print functions
  #--------
  set h_file_ext  $pref(cpp,h_file_ext)
  set cc_file_ext $pref(cpp,cc_file_ext)
  open_output_file "main$cc_file_ext"

  set type_list [idlgen_list_all_types "exception"]
  [***
  #include "it_print_funcs@$h_file_ext@

  //--------
  // Declare global objects
  //--------
  CORBA::ORB_var  global_orb = CORBA::ORB::_nil();
```

Example 23:

```
3 IT_GeniePrint*  global_print = 0;

int
main(int argc, char **argv)
{
        //--------
        // Initialise the ORB.
        //--------
        cout << "Initializing the ORB" << endl;
        global_orb = CORBA::ORB_init(argc, argv);

        //--------
        // Declare variables of each type
        //--------
***]
foreach type $type_list {
        set name my_[$type s_uname]
[***
        @[cpp_var_decl $name $type 1]@;
***]
}; # foreach type

[***

        ... //Initialize variables

***]
4 cpp_print_gen_init

[***
        //--------
        // Print out the value of each variable
        //--------
***]
foreach type $type_list {
5    set print_func [cpp_print_func_name $type]
    set name my_[$type s_uname]
[***
        cout << "@$name@ =";
        @$print_func@(cout, @$name@, 1);
        cout << endl;

***]
}; # foreach type

[***
        //--------
        // Delete the 'global_print' object
        //--------
6       @[cpp_print_delete]@;

} // end of main()
***]
close_output_file
```

The lines relevant to the cpp_poa_print library can be explained as follows:

1. The full version of the cpp_poa_print library is included, using smart_source.
2. This line and the following line generate the it_print_funcs.h and it_print_funcs.cxx files, respectively. These files define a set of functions that can be used to print out user-defined IDL types. A function is defined for each IDL type declared in the $file that is parsed at the outset.
3. The IT_GeniePrint class is the printer class defined in the it_print_funcs.h and it_print_funcs.cxx files. The IT_GeniePrint member functions print out CORBA data types.
4. The cpp_print_gen_init command initializes a pointer to an IT_GeniePrint object. The default name of the pointer is global_print.
5. The code in this foreach loop uses the generated print functions to print out sample instances of each CORBA data type. The print function invocation corresponding to each $type type is generated, using cpp_print_func_name.
6. The cpp_print_delete command deletes the IT_GeniePrint object with the default name global_print.

C++ Generated Code

The example script generates the following C++ code when run against the sample IDL:

```
//C++
#include "it_print_funcs.h

//--------
// Declare global objects
//--------
CORBA::ORB_var  global_orb = CORBA::ORB::_nil();
IT_GeniePrint*  global_print = 0;

int main(int argc, char **argv)
{
        //--------
        // Initialise the ORB.
        //--------
        cout << "Initializing the ORB" << endl;
        global_orb = CORBA::ORB_init(argc, argv);
```

```
//--------
// Declare variables of each type
//--------
CORBA::Short                    my_short;
CORBA::Long                     my_long;
CORBA::UShort                   my_unsigned_short;
CORBA::ULong                    my_unsigned_long;
CORBA::Float                    my_float;
CORBA::Double                   my_double;
CORBA::Boolean                  my_boolean;
CORBA::Octet                    my_octet;
CORBA::Char                     my_char;
CORBA::String_var               my_string;
CORBA::Any                      my_any;
CORBA::Object_var               my_Object;
widget                          my_widget;
longSeq                         my_longSeq;
long_array                      my_long_array;
foo_var                         my_foo;

... //Initialize variables

// Initialise the global printer object.
//
global_print = new IT_GeniePrint(global_orb);
```

```
        //--------
        // Print out the value of each variable
        //--------
        cout << "my_short =";
        global_print->print_short(cout, my_short, 1);
        cout << endl;

        cout << "my_long =";
        global_print->print_long(cout, my_long, 1);
        cout << endl;

        ... // and so on (some data types skipped)

        cout << "my_widget =";
       global_print->genie_print_widget(cout, my_widget,
   1);
        cout << endl;

        cout << "my_longSeq =";
        global_print->genie_print_longSeq(cout,
   my_longSeq, 1);
        cout << endl;

        cout << "my_long_array =";
        global_print->genie_print_long_array(cout,
   my_long_array, 1);
        cout << endl;

        cout << "my_foo =";
        global_print->print_object(cout, my_foo, 1);
        cout << endl;

        //--------
        // Delete the 'global_print' object
        //--------
        delete global_print;

} // end of main()
```

The cpp_poa_random Utility Library

The minimal API of the cpp_poa_random library is made available by the following command:

```
smart_source "cpp_poa_random/lib-min.tcl"
```

The minimal API defines the following commands:

```
cpp_gen_random_assign_stmt type name indent
cpp_random_assign_stmt type name
cpp_random_delete ?random?
cpp_random_gen_init ?orb? ?seed? ?random?
```

See “cpp_poa_random Commands” on page 209 for details.

For access to the full API of the cpp_poa_random library, use the following command:

```
smart_source "cpp_poa_random/lib-full.tcl"
```

The full library includes the command from the minimal library and the following commands:

```
gen_cpp_random_funcs_h
gen_cpp_random_funcs_cc ?ignored?
```

These commands generate the `it_random_funcs.h` and `it_random_funcs.cc` files, respectively. See "cpp_poa_random Commands" on page 209 for details.

Example Script

The following script shows how to use the commands of the `cpp_poa_random` library. This example is an extension of the example shown earlier (see page 108).

Example 24:

```
# Tcl
smart_source "std/sbs_output.tcl"
smart_source "std/cpp_poa_lib.tcl"
smart_source "cpp_poa_print/lib-full.tcl"
1 smart_source "cpp_poa_random/lib-full.tcl"

if {$argc != 1}  {
        puts "usage: ..."; exit
}
set file [lindex $argv 0]
set ok [idlgen_parse_idl_file $file]
if {!$ok} { exit }

#--------
# Generate it_print_funcs.{h,cc}
#--------
gen_cpp_print_funcs_h
gen_cpp_print_funcs_cc 1

2 #--------
# Generate it_random_funcs.{h,cc}
#--------
gen_cpp_random_funcs_h
gen_cpp_random_funcs_cc 1

#--------
# Generate a file which contains
# calls to the print and random functions
#--------
set h_file_ext  $pref(cpp,h_file_ext)
set cc_file_ext $pref(cpp,cc_file_ext)
open_output_file "main$cc_file_ext"

set type_list [idlgen_list_all_types "exception"]
[***
#include "it_print_funcs@$h_file_ext@
```

Example 24:

```
3 #include "it_random_funcs@$h_file_ext@

//--------
// Declare global objects
//--------
CORBA::ORB_var  global_orb = CORBA::ORB::_nil();
IT_GeniePrint*  global_print = 0;
4 IT_GenieRandom* global_random = 0;

int
main(int argc, char **argv)
{
        //--------
        // Initialise the ORB.
        //--------
        cout << "Initializing the ORB" << endl;
        global_orb = CORBA::ORB_init(argc, argv);

        //--------
        // Declare variables of each type
        //--------
***]
foreach type $type_list {
        set name my_[$type s_uname]
[***
        @[cpp_var_decl $name $type 1]@;
***]
}; # foreach type

output "\n"
cpp_print_gen_init

output "\n"
5 cpp_random_gen_init

[***

        //--------
        // Assign random values to each variable
        //--------
***]
foreach type $type_list {
        set name my_[$type s_uname]
[***
```

Example 24:

```
6       @[cpp_random_assign_stmt $type $name]@;
***]
}; # foreach type

[***

        //--------
        // Print out the value of each variable
        //--------
***]
foreach type $type_list {
    set print_func [cpp_print_func_name $type]
    set name my_[$type s_uname]
[***
        cout << "@$name@ =";
        @$print_func@(cout, @$name@, 1);
        cout << endl;

***]
}; # foreach type

[***
        //--------
        // Delete global objects
        //--------
        @[cpp_print_delete]@;
7       @[cpp_random_delete]@;

} // end of example_func()
***]
close_output_file
```

The lines relevant to the cpp_poa_random library can be explained as follows:

1. The full cpp_poa_random library is included, using smart_source.
2. This line and the following line generate the it_random_funcs.h and it_random_funcs.cxx files. These files define a class with member functions that generate random values for user-defined IDL types. A function is defined for each IDL type declared in the $file, which is parsed at the outset.
3. This include line ensures that the generated code has access to the declarations in it_random_funcs.h.
4. The IT_GenieRandom class is defined in the it_random_funcs.h and it_random_funcs.cxx files. The IT_GenieRandom member functions are used to generate random values for CORBA data types.
5. The cpp_random_gen_init command initializes a pointer to an IT_GenieRandom object. The default pointer name is global_random.
6. The cpp_random_assign_stmt command is used to generate a statement that initializes a variable with random data. The generated statement calls an IT_GenieRandom member function of the appropriate type.
7. The cpp_random_delete command deletes the IT_GenieRandom object with the default name global_random.

C++ Generated Code

The example script generates the following C++ code when run against the sample IDL:

```
//C++
#include "it_print_funcs.h
#include "it_random_funcs.h
//--------
// Declare global objects
//--------
CORBA::ORB_var  global_orb = CORBA::ORB::_nil();
IT_GeniePrint*  global_print = 0;
IT_GenieRandom* global_random = 0;
int main(int argc, char **argv)
{
        //--------
        // Initialise the ORB.
        //--------
        cout << "Initializing the ORB" << endl;
        global_orb = CORBA::ORB_init(argc, argv);
        //--------
        // Declare variables of each type
        //--------
        CORBA::Short                    my_short;
        CORBA::Long                     my_long;
        CORBA::UShort                   my_unsigned_short;
        CORBA::ULong                    my_unsigned_long;
        CORBA::Float                    my_float;
        CORBA::Double                   my_double;
        CORBA::Boolean                  my_boolean;
        CORBA::Octet                    my_octet;
        CORBA::Char                     my_char;
        CORBA::String_var               my_string;
        CORBA::Any                      my_any;
        CORBA::Object_var               my_Object;
        widget                          my_widget;
        longSeq                         my_longSeq;
        long_array                      my_long_array;
        foo_var                         my_foo;
        // Initialise the global printer object.
        //
        global_print = new IT_GeniePrint(global_orb);

        // Initialise the global random generator object.
        //
        global_random = new IT_GenieRandom(global_orb);
```

```
        //--------
        // Assign random values to each variable
        //--------
        my_short = global_random->get_short();
        my_long = global_random->get_long();
        my_unsigned_short = global_random->get_ushort();
        my_unsigned_long = global_random->get_ulong();
        my_float = global_random->get_float();
        my_double = global_random->get_double();
        my_boolean = global_random->get_boolean();
        my_octet = global_random->get_octet();
        my_char = global_random->get_char();
        my_string = global_random->get_string( 0);
        global_random->get_any(my_any);
        my_Object = global_random->get_reference();
        global_random->genie_widget(my_widget);
        global_random->genie_longSeq(my_longSeq);
        global_random->genie_long_array(my_long_array);
        global_random->genie_foo(my_foo);

        //--------
        // Print out the value of each variable
        //--------
        cout << "my_short =";
        global_print->print_short(cout, my_short, 1);
        cout << endl;

        ... // identical to 'cpp_poa_print' example

        //--------
        // Delete global objects
        //--------
        delete global_print;
        delete global_random;

} // end of main()
```

Using the Java Print and Random Utility Libraries

This chapter shows how to use the java_poa_print and java_poa_random libraries, using some example Tcl scripts.

Two additional genies, `java_poa_print.tcl` and `java_poa_random.tcl` are provided with the code generation toolkit: the `java_poa_print.tcl` genie, to generate code that prints out CORBA data types and the `java_poa_random.tcl` genie, to generate code that creates random values for CORBA data types. The genies are discussed in the *Orbix 2000 Programmer's Guide*.

Associated with these genies are two libraries—the print and random utility libraries—that can be used in your own Tcl scripts to generate print statements or to initialize variables with random data.

Sample IDL for Examples

The examples in this chapter use the following sample IDL:

```
// IDL
struct  widget           {long a;};
typedef sequence<long>   longSeq;
typedef long             long_array[10];
interface foo {
    longSeq op(
        in widget       p_widget,
        inout string    p_string,
        out longSeq     p_longSeq,
        out long_array  p_long_array);
};
```

The java_poa_print Utility Library

The minimal API of the `java_poa_print` library is made available by the following command:

```
smart_source "java_poa_print/lib-min.tcl"
```

The minimal API defines the following commands:

```
java_gen_print_stmt type name print_obj_loc ?indent? ?ostream?
java_print_func_name type print_obj_loc
java_print_gen_init ?orb?
java_print_stmt type name print_obj_loc ?indent? ?ostream?
```

See “java_poa_print Commands” on page 259 for details.

For access to the full API of the `java_print` library, use the following command:

```
smart_source "java_poa_print/lib-full.tcl"
```

The full library includes commands from the minimal library and the following commands:

```
gen_java_print_funcs ?ignored?
```

This command generates the `IT_GeniePrint.java` file. See “java_poa_print Commands” on page 259 for details.

Example Script

The following script shows how to use the commands in the `java_poa_print` library.

Example 25:

```
#Tcl
smart_source "std/sbs_output.tcl"
smart_source "std/java_poa_lib.tcl"
1 smart_source "java_poa_print/lib-full.tcl"
smart_source "std/java_config_defaults.tcl"

if {$argc != 1} {
    puts "usage: ..."; exit 1
}

set file [lindex $argv 0]
set ok [idlgen_parse_idl_file $file]
if {!$ok} { exit }
```

Example 25:

```
2 #-----------
# Generate IT_GeniePrint.java
#-----------
gen_java_print_funcs

#--------
# generate a file which contains
# calls to the print functions
#--------

set java_file_ext $pref(java,java_file_ext)
open_output_file "Printer$java_file_ext"

set type_list [idlgen_list_all_types "exception"]

[***
import @$pref(java,printpackage_name)@.*;

import org.omg.CORBA.*;

public class Printer
{

    // global_orb -- make ORB global so all code can find it.
    //
    public static org.omg.CORBA.ORB global_orb = null;

3   @[java_indent 1]@public static IT_GeniePrint
   global_printer;

    public static void main (String args[])
    {
        // Initialise the ORB.
        //
        System.out.println ("Initializing the ORB");
        global_orb = ORB.init(args, null);

        // ----------
        // Declare variables of each type
        // ----------
***]

    foreach type $type_list {
        set name my_[$type s_uname]

        java_gen_var_decl $name $type in 1
    }; # foreach type

[***

        ... //Initialise variables
***]
```

Example 25:

```
4 java_print_gen_init "global_orb"

[***
        // --------
        // Print out the value of each variable
        // --------
***]

    foreach type $type_list {
5           set print_func [java_print_func_name $type
   "Printer"]
            set name my_[$type s_uname]
[***
        System.out.println ("@$name@ =");
        @$print_func@(System.out,@$name@, 1);
        System.out.println();
***]
    }; #foreach type

[***
    } //end of main()

}
***]

close_output_file
```

The lines relevant to the java_poa_print library can be explained as follows:

1. The full version of the java_poa_print library is included, using smart_source.
2. This line generates the IT_GeniePrint.java file. This file defines a set of functions that can be used to print out user-defined IDL types. A function is defined for each IDL type declared in the $file that is parsed at the outset.
3. The IT_GeniePrint class is the printer class defined in the IT_GeniePrint.java file. The IT_GeniePrint member functions print out basic CORBA data types and user-defined types.
4. The java_print_gen_init command initializes a pointer to an IT_GeniePrint object. The default name of the pointer is global_printer.
5. The code in this foreach loop uses the generated print functions to print out sample instances of each CORBA data type. The print function invocation corresponding to each $type type is generated, using java_print_func_name.

Java Generated Code

The example script generates the following Java code when run against the sample IDL:

```
//Java
import idlgen.*;
import org.omg.CORBA.*;
```

```
public class Printer
{
    // global_orb -- make ORB global so all code can find
   it.
    //
    public static org.omg.CORBA.ORB global_orb = null;

    public static IT_GeniePrint global_printer;

    public static void main (String args[])
    {
        // Initialise the ORB.
        //
        System.out.println ("Initializing the ORB");
        global_orb = ORB.init(args, null);

        // ----------
        // Declare variables of each type
        // ----------
        short                                my_short;
        int                                  my_long;
        short                               my_unsigned_short;
        int                                  my_unsigned_long;
        long                                 my_long_long;
        long
   my_unsigned_long_long;
        float                                my_float;
        double                               my_double;
        boolean                              my_boolean;
        byte                                 my_octet;
        char                                 my_char;
        java.lang.String                     my_string;
        char                                 my_wchar;
        java.lang.String                     my_wstring;
        org.omg.CORBA.Any                    my_any;
        org.omg.CORBA.Object                 my_Object;
        NoPackage.widget                     my_widget;
        int[]                                my_longSeq;
        int[]                                my_long_array;
        NoPackage.foo                        my_foo;
```

```
        //Initialise variables
        global_printer = new IT_GeniePrint(global_orb,
"");
        // --------
        // Print out the value of each variable
        // --------
        System.out.println ("my_short =");

Printer.global_printer.print_short(System.out,my_short
, 1);
        System.out.println();
        System.out.println ("my_long =");

Printer.global_printer.print_long(System.out,my_long,
1);
        System.out.println();
        System.out.println ("my_widget =");

        ... // and so on (some data types skipped)

 Printer.global_printer.genie_print_NoPackage_widget(
                       System.out,my_widget, 1);
 System.out.println();
 System.out.println ("my_longSeq =");
 Printer.global_printer.genie_print_NoPackage_longSeq(
                      System.out,my_longSeq, 1);
 System.out.println();
 System.out.println ("my_long_array =");

Printer.global_printer.genie_print_NoPackage_long_arra
y(
                         System.out,my_long_array, 1);
 System.out.println();
 System.out.println ("my_foo =");
Printer.global_printer.print_object(System.out,my_foo,
1);
 System.out.println();
} //end of main()
}
```

The java_poa_random Utility Library

The minimal API of the `java_poa_random` library is made available by the following command:

```
smart_source "java_poa_random/lib-min.tcl"
```

The minimal API defines the following commands:

```
java_gen_random_assign_stmt type name ?dir? random_obj_loc
   indent
java_random_assign_stmt type name ?dir? random_obj_loc
java_random_gen_init ?orb? ?seed? ?random?
```

See “java_poa_random Commands” on page 260 for details.

For access to the full API of the `java_poa_random` library, use the following command:

```
smart_source "java_poa_random/lib-full.tcl"
```

The full library includes the command from the minimal library and the following commands:

```
gen_java_random_funcs ?ignored?
```

This command generates the `IT_GenieRandom.java` file. See “java_poa_random Commands” on page 260 for details.

Example Script

The following script shows how to use the commands of the `java_poa_random` library. This example is an extension of the example shown earlier (see page 120).

Example 26:

```
#Tcl
smart_source "std/sbs_output.tcl"
smart_source "std/java_poa_lib.tcl"
smart_source "java_poa_print/lib-full.tcl"
1 smart_source "java_poa_random/lib-full.tcl"
smart_source "std/java_config_defaults.tcl"

if {$argc != 1} {
    puts "usage: ..."; exit 1
}

set file [lindex $argv 0]
set ok [idlgen_parse_idl_file $file]
if {!$ok} { exit }

#-----------
# Generate IT_GeniePrint.java
#-----------
gen_java_print_funcs

#-----------
# Generate IT_GenieRandom.java
#-----------
```

Example 26:

2
```
gen_java_random_funcs

#--------
# generate a file which contains
# calls to the print and random functions
#--------

set java_file_ext $pref(java,java_file_ext)
open_output_file "Random$java_file_ext"

set type_list [idlgen_list_all_types "exception"]

[***
import @$pref(java,printpackage_name)@.*;
***]

if { $pref(java,printpackage_name) !=
   $pref(java,randompackage_name) } {
[***
import @$pref(java,randompackage_name)@.*;
***]
}

[***
import org.omg.CORBA.*;

public class Random
{

    // global_orb -- make ORB global so all code can find it.
    //
    public static org.omg.CORBA.ORB global_orb = null;

    @[java_indent 1]@public static IT_GeniePrint
   global_printer;
```

Example 26:

3
```
    @[java_indent 1]@public static IT_GenieRandom
   global_random;

    public static void main (String args[])
    {
        // Initialise the ORB.
        //
        System.out.println ("Initializing the ORB");
        global_orb = ORB.init(args, null);

        // ----------
        // Declare variables of each type
        // ----------
***]

    foreach type $type_list {
        set name my_[$type s_uname]

        java_gen_var_decl $name $type in 1
    }; # foreach type


output "\n"
java_print_gen_init "global_orb"

output "\n"
java_random_gen_init "global_orb"

[***
        // --------
        // Assign random values to each variable
        // --------
***]

    foreach type $type_list {
        set name my_[$type s_uname]
[***
```
4 (margin marker beside line `java_random_gen_init "global_orb"`)

Example 26:

5
```
        @[java_random_assign_stmt $type $name "in"
    global_random]@;
***]
    }; #foreach type

[***

        // --------
        // Print out the value of each variable
        // --------
***]

    foreach type $type_list {
        set print_func [java_print_func_name $type
    "global_printer"]
        set name my_[$type s_uname]
[***
        System.out.println ("@$name@ =");
        @$print_func@(System.out,@$name@, 1);
        System.out.println();
***]
    }; #foreach type

[***
    } //end of main()

}
***]

close_output_file
```

The lines relevant to the `java_poa_random` library can be explained as follows:

1. The full `java_poa_random` library is included, using `smart_source`.
2. This line generates the `IT_GenieRandom.java` file. This file defines a class with member functions that generate random values for user-defined IDL types. A function is defined for each IDL type declared in the `$file`, which is parsed at the outset.
3. The `IT_GenieRandom` class is defined in the `IT_GenieRandom` file. The `IT_GenieRandom` member functions are used to generate random values for basic CORBA data types and user-defined types.
4. The `java_random_gen_init` command initializes a pointer to an `IT_GenieRandom` object. The default pointer name is `global_random`.
5. The `java_random_assign_stmt` command is used to generate a statement that initializes a variable with random data. The generated statement calls an `IT_GenieRandom` member function of the appropriate type.

Java Generated Code

The example script generates the following Java code when run against the sample IDL:

```
//Java
import idlgen.*;
import idlgen.*;
import org.omg.CORBA.*;

public class Random
{
    // global_orb -- make ORB global so all code can find it.
    //
    public static org.omg.CORBA.ORB global_orb = null;

    public static IT_GeniePrint global_printer;
    public static IT_GenieRandom global_random;

    public static void main (String args[])
    {
        // Initialise the ORB.
        //
        System.out.println ("Initializing the ORB");
        global_orb = ORB.init(args, null);

        // ----------
        // Declare variables of each type
        // ----------
        short                           my_short;
        int                             my_long;
        short                           my_unsigned_short;
        int                             my_unsigned_long;
        long                            my_long_long;
        long                            my_unsigned_long_long;
        float                           my_float;
        double                          my_double;
        boolean                         my_boolean;
        byte                            my_octet;
        char                            my_char;
        java.lang.String                my_string;
        char                            my_wchar;
        java.lang.String                my_wstring;
        org.omg.CORBA.Any               my_any;
        org.omg.CORBA.Object            my_Object;
        NoPackage.widget                my_widget;
        int[]                           my_longSeq;
        int[]                           my_long_array;
        NoPackage.foo                   my_foo;

        global_printer = new IT_GeniePrint(global_orb, "");

        // Initialise the global random generator object.
        //
        global_random = new IT_GenieRandom(global_orb, 0);
```

```
        // --------
        // Assign random values to each variable
        // --------
        my_short = Random.global_random.get_short();
        my_long = Random.global_random.get_long();
        my_unsigned_short = Random.global_random.get_ushort();
        my_unsigned_long = Random.global_random.get_ulong();
        my_long_long = Random.global_random.get_longlong();
        my_unsigned_long_long =
            Random.global_random.get_ulonglong();
        my_float = Random.global_random.get_float();
        my_double = Random.global_random.get_double();
        my_boolean = Random.global_random.get_boolean();
        my_octet = Random.global_random.get_octet();
        my_char = Random.global_random.get_char();
        my_string = Random.global_random.get_string( 10);
        my_wchar = Random.global_random.get_wchar();
        my_wstring = Random.global_random.get_wstring( 10);
        my_any = Random.global_random.get_any(1);
        my_Object = Random.global_random.get_reference();
        my_widget =
   Random.global_random.genie_NoPackage_widget();
        my_longSeq =
            Random.global_random.genie_NoPackage_longSeq();
        my_long_array =
            Random.global_random.genie_NoPackage_long_array();
        my_foo = Random.global_random.genie_NoPackage_foo();

        // --------
        // Print out the value of each variable
        // --------
        System.out.println ("my_short =");
        Random.global_printer.print_short(System.out,my_short,
   1);
        System.out.println();
        System.out.println ("my_long =");
       Random.global_printer.print_long(System.out,my_long, 1);
        System.out.println();
        System.out.println ("my_widget =");
        Random.global_printer.genie_print_NoPackage_widget(
            System.out,my_widget, 1);
        System.out.println();
        System.out.println ("my_longSeq =");
        Random.global_printer.genie_print_NoPackage_longSeq(
            System.out,my_longSeq, 1);
        System.out.println();
        System.out.println ("my_long_array =");
        Random.global_printer.genie_print_NoPackage_long_array(
            System.out,my_long_array, 1);
        System.out.println();
        System.out.println ("my_foo =");
        Random.global_printer.print_object(System.out,my_foo,
   1);
        System.out.println();
    } //end of main()
}
```

Further Development Issues

This chapter details further development facets of the code generation toolkit that help you to write genies more effectively.

Global Arrays

The code generation toolkit employs a number of global arrays to store common information.

Some of these global arrays are discussed in previous chapters. For example, `$idlgen(root)`, see "Traversing the Parse Tree" on page 26, holds the results of parsing an IDL file.

> **Note:** When using arrays make sure you do not place spaces inside the parentheses, otherwise Tcl will treat it as a different array index to the one you intended. For example, `$variable(index)` is not the same as `$variable( index )`.

This sections covers the following topics:

The $idlgen Array

This array contains entries that are related to the core `idlgen` executable.

- $idlgen(root)
- $idlgen(cfg)
- $idlgen(exe_and_script_name)

$idlgen(root)

This variable holds the root of an IDL file parsed with the built-in parser. For example:

```
# Tcl
if {![idlgen_parse_idl_file "finance.idl"]} {
    exit
}
set node [$idlgen(root) lookup Account]
```

For more information, see "Processing an IDL File".

$idlgen(cfg)

This variable represents all the configuration settings from the standard configuration file `idlgen.cfg`:

```
# Tcl
set version [$idlgen(cfg) get_string
   orbix.version_number]
```

For more information, see "Using Configuration Files" on page 52.

$idlgen(exe_and_script_name)

This variable contains the name of the `idlgen` executable together with the name of the Tcl script being run. This variable is convenient for printing usage statements:

```
# Tcl
puts "Usage: $idlgen(exe_and_script_name) -f <file>"
```

Run the `idlgen` interpreter from the command line:

```
idlgen globalvars.tcl

Usage: idlgen globalvars.tcl -f <file>
```

The $pref Array

It is best to avoid embedding coding preferences in a script that will be re-used in many different circumstances. Passing numerous parameters to each command procedure is impractical, so it is better to use a global repository of coding preferences.

Genie preferences

The code generation toolkit provides a number of mechanisms to support genie preferences:

- Command line arguments.
- Configuration files.

Configuration files

Configuration files can, in coding terms, be time consuming to access. The preference array caches the more common preferences found in a configuration file. Users can specify values in the `default` scope of the standard configuration file and they are placed in the `$pref` array during initialization of the `idlgen` interpreter. This allows quick access to the main options without the overhead of using the configuration file commands and operations. Command-line arguments can then override any of the more static preferences specified in configuration files.

Example configuration file

This is an example configuration file, with some entries in the `default` scope:

```
default {
        trousers {
            waist = "32";
            inside_leg = "32";
        };
        jacket {
            chest = "42";
            colour = "pink";
        };
};
```

The corresponding entries in the preference array are as follows:

```
$pref(trousers,waist)
$pref(trousers,inside_leg)
$pref(jacket,chest)
$pref(jacket,colour)
```

The `idlgen` interpreter automatically creates preference array values for all the `default` scoped entries in the standard configuration file using the following command:

```
# Tcl
idlgen_set_preferences $idlgen(cfg)
```

Note: This command assumes that all names in the configuration file containing `is_` or `want_` have boolean values. If such an entry has a value other than `0` or `1`, or `true` or `false`, an exception is thrown.

This command takes the `default` scoped entries from the specified configuration file and copies them into the preference array. This command can also be run on configuration files that you have processed explicitly:

```
# Tcl
if { [catch {
        set cf [idlgen_parse_config_file "shop.cfg"]
        idlgen_set_preferences $cf
        } err]
    } else {
        puts stderr $err
        exit
    }
}
parray pref
```

Running this script on the described configuration file results in the following output:

```
idlgen prefs.tcl

pref(trousers,waist)        = 32
pref(trousers,inside_leg)   = 32
pref(jacket,chest)          = 42
pref(jacket,colour)         = pink
```

It is good practice to ensure that the defaults in a configuration file take precedence over default values in a genie. This behavior can be accomplished by using the Tcl `info exists` command to ensure that a preference is set only if it does not exist in the configuration file.

```
if { ![info exists pref(trousers,waist)] } {
        set pref(trousers,waist) "30"
}
```

You should extend the default `scope` of the configuration file when your genie requires an additional preference entry or new category. You can complement the extended `scope` by using the described commands to place quick access preferences in the preferences array.

The command procedures in the `std/output.tcl` library examine the entries described in Table 12:

Table 12: $pref(...) Array Entries

$pref(...) Array Entry	Purpose
`$pref(all,output_dir)`	A file generated with the `open_output_file` command file is placed in the directory specified by this entry. If this entry has the value "." or "" (an empty string), the file is generated in the current working directory. The default value of this entry is an empty string.
`$pref(all,want_diagnostics)`	If this has the value `1`, diagnostic messages, such as `idlgen: creating foo_i.h`, are written to standard output whenever a genie generates an output file. If this entry has the value `0`, no diagnostic messages are written. The `-v` (verbose) command-line option sets this entry to `1` and the `-s` (silent) command-line option sets this entry to `0`. The default value of this entry is `1`.

The $cache Array

If a command is called frequently, caching its result can speed up a genie. Caching the results of frequently called commands can speed up genies by up to twenty per cent. Many of the commands supplied with the code generation toolkit perform caching. This mechanism is useful for speeding up your own genies.

Consider this simple command procedure that takes three parameters and returns a result:

```
# Tcl
proc foobar {a b c} {
    set result ...; # set to the normal body
                    # of the procedure here
    return $result
}
```

To cache the results in the cache array the command procedure can be altered as below:

```
# Tcl
proc foobar {a b c} {
    global cache
    if { [info exists cache(foobar,$a,$b,$c)]} {
        return $cache(foobar,$a,$b,$c)
    }
    set result ...; # set to the normal body
                    # of the procedurehere
    set cache(foobar,$a,$b,$c) $result
    return $result
}
```

You should only cache the results of *idempotent* procedures; that is, procedures that always return the same result when invoked with the same parameters. For example, a random-number generator function is not idempotent, and hence its result should not be cached.

Note: A side-effect of the `idlgen_parse_idl_file` command is that it destroys `$cache(...)`. This is to prevent a genie from having stale cache information if it processes several IDL files.

Re-Implementing Tcl Commands

Consider a genie which uses a particular Tcl command procedure extensively, but you must now alter its behavior. The genie uses the following command procedure a number of times:

```
# Tcl
proc say_hello {message} {
    puts $message
}
```

There are a number of different ways you could alter the behavior of this command procedure:

- Re-code the procedure’s body.
- Replace all instances where the genie calls this procedure with calls to a new procedure.
- Use a feature of the Tcl language that allows you to re-implement procedures without affecting the original procedure.

The third option allows the genie to use the new implementation of the command procedure, while still allowing the process to be reversed if required. The new implementation of the command procedure can be slotted in and out, when required, without having to alter the calling code.

This is the new implementation of the `say_hello` command procedure:

```
# Tcl
proc say_hello {message} {
    puts "Hello '$message'"
}
```

If a genie used `say_hello` from the original script, it can use the original procedure's functionality:

```
# Tcl
smart_source "original.tcl"
say_hello Tony
```

Run the `idlgen` interpreter from the command line:

```
idlgen application.tcl

Tony
```

However, to override the command procedure, the programmer only needs to `smart_source` the new command procedure instead:

```
# Tcl
smart_source override.tcl
say_hello Tony
```

Run the `idlgen` interpreter from the command line:

```
idlgen application.tcl

Hello 'Tony'
```

More Smart Source

When commands are re-implemented, there is still a danger that a script might `smart_source` the replaced command back in. This would cause the original (and unwanted) version of the command to be re-instated.

```
# Tcl
smart_source "override.tcl"
smart_source "original.tcl" ;# Oops
say_hello Tony
```

Run the `idlgen` interpreter from the command line:

```
idlgen application.tcl

Tony
```

Smart source provides a mechanism to prevent this. This mechanism is accomplished by using the `pragma once` directive to nullify repeated attempts to `smart_source` a file.

For example, the following implementation prohibits the use of `smart_source` multiple times on the original command procedure. Here is the original implementation with the new `pragma` directive added:

```
# Tcl
smart_source pragma once
proc say_hello {message} {
    puts $message
}
```

The following Tcl script is the new implementation, but note that it uses `smart_source` on the original file as well. This is to ensure that if anyone uses the new implementation, the old implementation is guaranteed not to override the new implementation later on.

```
# Tcl
smart_source "original.tcl"
smart_source pragma once

proc say_hello {message} {
    puts "Hello '$message'"
}
```

Now, when the genie accidentally uses `smart_source` on the original command procedure, the new procedure is not overridden by the original.

```
# Tcl
smart_source "override.tcl"
smart_source "original.tcl" ;# Will not override
say_hello Tony
```

Run the `idlgen` interpreter from the command line:

```
idlgen application.tcl

Hello 'Tony'
```

More Output

An alternative set of output commands is found in `std/sbs_output.tcl`. The `sbs` prefix stands for *Smart But Slower* output. The Tcl commands that are available in this alternative script have the same API as the ones available in `std/output.tcl`, but they have a different implementation.

The main advantage of using this alternative library of commands is that it can dramatically cut down on the re-compilation time of a project that contains auto-generated files. A change to an IDL file might affect only a few of the generated files, but if all the files are written out, the makefile of the project can attempt to rebuild portions of the project unnecessarily.

The std/sbs_output.tcl commands only rewrite a file if the file has changed. These overridden commands are slower because they write a temporary file and run a diff with the target file. This is typically 10% slower than the equivalent commands in std/output.tcl.

Miscellaneous Utility Commands

The following sections discuss miscellaneous utility commands provided by the idlgen interpreter.

This section covers the following topics:

idlgen_read_support_file

Scripts often generate lots of repetitive code, and also copy some pre-written code to the output file. For example, consider a script that generates utility functions for converting IDL types into corresponding Widget types. Such a script might be useful if you want to build a CORBA-to-Widget gateway, or if you are adding a CORBA wrapper to an existing Widget-based application. Typically, such a script:

- Contains procedures that generate data-type conversion functions for user-defined type such as structs, unions, and sequences.
- Copies (to the output files) pre-written functions that perform data-type conversion for built-in IDL types such as short, long, and string.

You can ensure that pre-written code is copied to an output file by taking advantage of the idlgen interpreter's bilingual capability: simply embed all the pre-written code inside a text block as shown below:

```
proc foo_copy_pre_written_code {} {
[***
    ... put all the pre-written code here ...
***]
}
```

This approach works well if there is only a small amount of pre-written code, say fifty lines. However, if there are several hundred lines of pre-written code this approach becomes unwieldy. The script might contain more lines of embedded text than of Tcl code, making it difficult to follow the steps in the Tcl code.

The idlgen_read_support_file command is provided to tackle this scalability issue. It is used as follows:

```
proc foo_copy_pre_written_code {} {
    output [idlgen_read_support_file
   "foo/pre_written.txt"]
}
```

The idlgen_read_support_file command searches for the specified file relative to the directories in the script_search_path entry in the idlgen.cfg configuration file (which makes it possible for you to keep pre-written code files in the same directory as your genies). If idlgen_read_support_file cannot find the file, it throws an exception. If it can find the file, it reads the file and returns its entire contents as a string. This string can then be used as a parameter to the output command.

As shown in the above example, idlgen_read_support_file can be used to copy chunks of pre-written text into an output file. However, you can also use it to copy entire files, as the following example illustrates:

```
proc foo_copy_all_files {} {
    foo_copy_file "pre_written_code.h"
    foo_copy_file "pre_written_code.cc"
    foo_copy_file "Makefile"
}

proc foo_copy_file {file_name} {
    open_output_file $file_name
    output [idlgen_read_support_file "foo/$file_name"]
    close_output_file
}
```

Some programming projects can be divided into two parts:

- A genie that generates lots of repetitive code.
- Five or ten handwritten files containing non-repetitious code that cannot be generated easily.

By using the idlgen_read_support_file command as shown in the above example, it is possible to shrink-wrap such a project into a genie that both generates the repetitious code and copies the hand-written files (including a Makefile). Shrink-wrapped scripts are a very convenient format for distribution. For example, suppose that different departments in your organization have genies implemented using the Widget toolkit/database. If you have written a genie that enables you to put a CORBA wrapper around an arbitrary Widget-based genie, you can shrink-wrap this genie (and its associated pre-written files) and distribute it to the different departments in your organization, so that they can easily use it to wrap their genies.

idlgen_support_file_full_name

This command is used as follows:

```
idlgen_support_file_full_name local_name
```

This command is related to `idlgen_read_support_file`, but instead of returning the contents of the file, it just locates the file and returns its full pathname. This command can be useful if you want to use the file name as a parameter to a shell command that is executed with the `exec` command.

idlgen_gen_comment_block

Many organizations require that all source-code files contain a standard comment, such as a copyright notice or disclaimer. The `idlgen_gen_comment_block` command is provided for this purpose. Suppose that the `default.all.copyright` entry in the `idlgen.cfg` configuration file is a list of strings containing the following text:

```
Copyright ACME Corporation 1998.
All rights reserved.
```

When the `idlgen` interpreter is started, the above configuration entry is automatically copied into `$pref(all,copyright)`. If a script contains the following commands

```
set text $pref(all,copyright)
idlgen_gen_comment_block $text "//" "-"
```

the following is written to the output file:

```
// ----------------------------------------
// Copyright ACME Corporation 1998.
// All rights reserved.
// ----------------------------------------
```

The `idlgen_gen_comment_block` command takes three parameters:

- The first parameter is a list of strings that denotes the text of the comment to be written.
- The second parameter is the string used to start a one-line comment, for example, `//` in C++ and Java, `#` in Makefiles and shell-scripts, and `--` in Ada.
- The third parameter is the character that is used for the horizontal lines that form a box around the comment.

idlgen_process_list

Genies frequently process lists. If each item in a list is to be processed identically, this can be achieved with a Tcl `foreach` loop:

```
foreach item $list {
    process_item $item
}
```

However, some lists require slightly more complex logic. The classic case is a list of parameters separated by commas. In this case, the `foreach` loop can be written in the form:

```
set arg_list [$op contents {argument}]
set len [llength $arg_list]
set i 1
foreach arg $arg_list {
    process_item $arg
    if {$i < $len} { output "," }
    incr i
}
```

This example shows that generating a separator (for example, a comma) between each item of a list requires substantially more code. Furthermore, empty lists might require special-case logic.

The `idlgen` interpreter provides the `idlgen_process_list` command to ease the burden of list processing. This command takes six parameters:

```
idlgen_process_list list func start_str sep_str end_str
   empty_str
```

The `idlgen_process_list` command returns a string that is constructed as follows:

If the *list* is empty, *empty_str* is returned. Otherwise:

1. The `idlgen_process_list` command initializes its result with *start_str*.
2. It then calls *func* repeatedly (each time passing it an item from *list* as a parameter).
3. The strings returned from these calls are appended onto the result, followed by *sep_str* if the item being processed is not the last one in the list.
4. When all the items in *list* have been processed, *end_str* is appended to the result, which is then returned.

The *start_str, sep_str, end_str* and *empty_str* parameters have a default value of `""`. Therefore you need to specify explicitly only the parameters that you need. The following code snippet illustrates how `idlgen_process_list` can be used:

```
proc l_name {node} {
    return [$node l_name]
}
proc gen_call_op {op} {
    set arg_list [$op contents {argument}]
    set call_args [idlgen_process_list $arg_list \
                     l_name "\n\t\t\t" ",\n\t\t\t"]
[***
    try {
        obj->@[$op l_name]@(@$call_args@);
    } catch (...) { ... }
***]
}
```

If the above gen_call_op command procedure is invoked on two operations, one that takes three parameters and another that does not take any parameters, then the output generated might be something like:

```
try {
    obj->op1(
            stock_id,
            quantity,
            unit_price);
} catch (...) { .. }
try {
    obj->op2();
} catch (...) { ... }
```

idlgen_pad_str

The idlgen_pad_str command takes two parameters:

idlgen_pad_str *string pad_len*

This command calculates the length of the string parameter. If it is less than pad_len, it adds spaces onto the end of string to make it pad_len characters long. The padded string is then returned. This command can be used to obtain vertical alignment of parameter/variable declarations. For example, consider the following example:

```
foreach arg $op {
    set type [[$arg type] s_name]
    set name [$arg l_name]
    puts "[idlgen_pad_str $type 12] $name;"
}
```

For a given operation, the output of the above code might be as follows:

```
long         wages;
string       names;
Finance::Account acc;
Widget       foo;
```

As can be seen, the names of most of the parameters are vertically aligned. However, the type name of the *acc* parameter is longer than 12 (the pad_len) causing acc to be misaligned. Using a relatively large value for *pad_len*, such as 32, minimizes the likelihood of misalignment occurring. However, IDL syntax does not impose any limit on the length of identifiers, so it is impossible to pick a value of pad_len large enough to guarantee alignment in all cases. For this reason, it is a good idea for scripts to determine pad_len from an entry in a configuration file. In this way, users can modify it easily to suit their needs. Some commands in the cpp_boa_lib.tcl library use $pref(cpp,max_padding_for_types) for alignment of parameters and variable declarations.

Recommended Programming Style

The bundled genies share a common programming style. The following section highlights some aspects of this programming style and explains how adopting the same style can help you when developing your own genies.

This section covers the following topics:

Organizing Your Files

The following code illustrates several recommendations for organizing the files in your genies:

```
#--------
# File: foo.tcl
#--------
smart_source "foo/args.tcl"
process_cmd_line_args idl_file preproc_opts

set ok [idlgen_parse_idl_file $idl_file $preproc_opts]
if {!$ok} { exit }

if {$pref(foo,want_client)} {
    smart_source "foo/gen_client_cc.bi"
    gen_client_cc
}

if {$pref(foo,want_server)} {
    smart_source "foo/gen_server_cc.bi"
    gen_server_cc
}

if {$pref(foo,want_impl_class)} {
    smart_source "foo/gen_impl_class_h.bi"
    smart_source "foo/gen_impl_class_cc.bi"
    set want {interface}
    set rec_into {module}
    foreach i [$idlgen(root) rcontents $want $rec_into] {
        gen_impl_class_h  $i
        gen_impl_class_cc $i
    }
}
```

The above example demonstrates the following points:

- Do not define all the genie's logic in a single file. Instead, write a small mainline script that uses `smart_source` to access commands in other files. This helps to keep the genie code modular.

- If the mainline script of your genie is called foo.tcl, any associated files should be in a sub-directory called foo. This helps to avoid clashing file names. It also ensures that running the command idlgen -list lists the foo.tcl genie, but does not list any of the associated files that are used to help implement foo.tcl.
- Command procedures to process command-line arguments should be put into a file called args.tcl (in the genie's sub-directory). The results of processing command-line arguments should be passed back to the caller either with Tcl upvar parameters or with the $pref array (or a combination of both). If you use the $pref array then use the name of the genie as a prefix for entries in $pref. For example, the args.tcl command procedures in the cpp_genie.tcl genie uses the entry $pref(cpp_genie,want_client) to indicate the value of the -client command-line option.
- If your genie has several options (such as -client, -server) for selecting different kinds of code that can be generated, place the command procedures for generating each type of code into separate files, and smart_source a file only if the corresponding command-line option has been provided. This speeds up the genie if only a few options have been generated because it avoids unnecessary use of smart_source on files.

Organizing Your Command Procedures

The following code illustrates several recommendations for organizing the command procedures in your genies:

```
#--------
# File: foo/gen_impl_class_cc.bi
#--------
...
proc gen_impl_class_cc {i} {
    global pref
    set file [cpp_impl_class $i]$pref(cpp,cc_file_ext)
    open_output_file $file

    gen_impl_class_cc_file_header
    gen_impl_class_cc_constructor
    gen_impl_class_cc_destructor

    foreach op [$i contents {operation}] {
        gen_impl_class_cc_operation $op
    }
    close_output_file
}
```

The above example demonstrates the following points:

1. Large procedures are broken into a collection of smaller procedures.
2. Avoid name space pollution of procedure names:
   - Use a common prefix for names of all procedures defined in a file.
   - You can use (an abbreviation of) the file name as the prefix.

3. Use gen_ as part of the prefix if the procedure outputs its result.
   - Example: cpp_gen_operation_h outputs an operation's signature.
4. Procedures without gen_ in their name return their result.
   - Example: cpp_is_fixed_size returns a value.

Writing Library Genies

Let us suppose that your organization has many existing genies that are implemented with the aid of a product called ACME. In order to aid the task of putting CORBA wrappers around these genies, you decide to write a genie called idl2acme.tcl that generates C++ conversion functions to convert IDL types to their ACME counterparts, and vice versa. For example, if there is an IDL type called foo and a corresponding ACME type called acme_foo, idl2acme.tcl generates the following two functions:

```
void idl_to_acme_foo(const foo &from, acme_foo &to);
void acme_to_idl_foo(const acme_foo &from, foo &to);
```

The genie generates similar conversion functions for all IDL types. It can be run as follows:

```
idlgen idl2acme.tcl some_file.idl

idlgen: creating idl2acme.h
idlgen: creating idl2acme.cc
```

The idl2acme.tcl script can look something like this:

```
#--------
# File: idl2acme.tcl
#--------
smart_source "idl2acme/args.tcl"

parse_cmd_line_args file opts
set ok [idlgen_parse_idl_file $file $opts]
if {!$ok} { exit }

smart_source "std/sbs_output.tcl"
smart_source "idl2acme/gen_idl2acme_h.bi"
smart_source "idl2acme/gen_idl2acme_cc.bi"

gen_idl2acme_h
gen_idl2acme_cc
```

Calling a Genie from Other Genies

Although being able to run idl2acme.tcl as a stand-alone genie is useful, you might decide that you would also like to call upon its functionality from inside other genies. For example, you might modify a copy of the bundled cpp_genie.tcl script in order to develop acme_genie.tcl, which is a genie that is tailored specifically for the needs of people who want to put CORBA wrappers around existing ACME-based genies. In order to access the API of idl2acme.tcl, the following lines of code can be embedded inside acme_genie.tcl:

```
smart_source "idl2acme/gen_idl2acme_h.bi"
smart_source "idl2acme/gen_idl2acme_cc.bi"

gen_idl2acme_h
gen_idl2acme_cc
```

This might seem like an elegant approach to take. However, it suffers from two defects:

1. Scalability.

   In the above example, acme_genie.tcl requires just two smart_source commands to get access to the API of idl2acme.tcl. However, a more feature-rich library might have its functionality implemented in ten or twenty files. Accessing the API of such a library from inside acme_genie.tcl would require ten or twenty smart_source commands, which is somewhat unwieldy. It is better if a genie can access the API of a library with just one smart_source command, regardless of how feature-rich that library is.
2. Lack of encapsulation.

   Any genie that wants to access the API of idl2acme.tcl must be aware of the names of the files in the idl2acme directory. If the names of these files ever change, it breaks any genies that make use of them.

Both of these problems can be solved.

When writing the idl2acme.tcl genie, create the following two files:

```
idl2acme/lib-full.tcl
idl2acme/lib-min.tcl
```

The idl2acme/lib-full.tcl file contains the necessary smart_source commands to access the full API of the idl2acme library. Therefore, a genie can access this API with just one smart_source command.

The idl2acme/lib-min.tcl file contains the necessary smart_source commands to access the minimal API of the idl2acme library. In general, the difference between the full and minimal APIs varies from one library to another and should be clearly specified in the library's documentation.

The Full API

In the case of the idl2acme library, the full API might define five procedures:

```
gen_idl2acme_h
gen_idl2acme_cc
gen_acme_var_decl_stmt type name
gen_idl2acme_stmt type from_var to_var
gen_acme2idl_stmt type from_var to_var
```

These command procedures are used as follows:

- The gen_idl2acme_h and gen_idl2acme_cc procedures generate the idl2acme.h and idl2ame.cc files, respectively.
- The gen_acme_var_decl_stmt procedure generates a C++ variable declaration of an ACME type corresponding to the specified IDL type.
- The gen_idl2acme_stmt procedure generates a C++ statement that converts an IDL type to an ACME type, and the gen_acme2idl_stmt procedure generates a C++ statement that performs the data-type translation in the opposite direction.

The Minimal API

The minimal API (as exposed by idl2acme/lib-min.tcl) includes the latter three command procedures. A genie can smart_source the minimal API, to generate code that makes calls to data-type conversion routines. A genie can access the full API with smart_source if it also needs to generate the implementation of the data-type conversion routines. The reason for providing both full and minimal libraries is that the minimal library is likely to contain only a small amount of code, and hence can be accessed much faster with smart_source than the full library, which typically contains hundreds or thousands of lines of code. Thus, genies that require only the minimal API can start up faster.

The concept of a minimal API might not make sense for some libraries. In such cases, only the full library should be provided.

Commenting Your Generated Code

As your genies have a high likelihood of containing code written in another language, it is even more important to comment both sets of code when creating genies.

Putting block comments into the generated code:

- Documents your genie scripts.
- Documents the generated code.
- Shows the relationship between scripts and generated code.
- Is a very useful debugging aid.

The following is an example section of a Tcl (bilingual) script that has been commented:

```
# Tcl
proc gen_impl_class_cc_operation{ op } {
[***
//---------------------------------------------
// Function:          @[cpp_ident_s_name $op]@
// Description:       Implements the corresponding
//                    IDL operation
//---------------------------------------------
***]
    cpp_gen_operation_cc $op ;# C++ signature of op
    ...
}
```

Part III

C++ Genies Library Reference

In this part

This part contains the following chapters:

C++ Development Library

The code generation toolkit comes with a rich C++ development library that makes it easy to create code generation applications that map IDL to C++ code.

Naming Conventions in API Commands

Abbreviations

The abbreviations shown in Table 13 are used in the names of commands defined in the `std/cpp_poa_lib.tcl` library.

Table 13: *Abbreviations Used in Command Names.*

Abbreviation	Meaning
`clt`	Client
`srv`	Server
`var`	Variable
`var_decl`	Variable declaration
`is_var`	See "Naming Conventions for is_var"
`gen_`	See "Naming Conventions for gen_"
`par/param`	Parameter
`ref`	Reference
`stmt`	Statement
`mem`	Memory
`op`	Operation
`attr_acc`	An IDL attribute's accessor
`attr_mod`	An IDL attribute's modifier.
`sig`	Signature.
`_cc`	A C++ code file—normally `.cxx`,. but the extension is configurable.
`_h`	A C++ header file.

Command names in `std/cpp_poa_lib.tcl` start with the `cpp_` prefix. For example, the following statement generates the C++ signature of an operation (for use in a header file) and assigns the result to the `foo` variable:

```
set foo [cpp_op_sig_h $op]
```

Naming Conventions for is_var

The CORBA mapping from IDL to C++ provides smart pointers whose names end in _var. For example, an IDL struct called widget has a C++ smart pointer type called widget_var. Sometimes, the syntactic details of declaring and using C++ variables depends on whether or not you use these _var types. For this reason, some of the commands in std/cpp_poa_lib.tcl take a boolean parameter called is_var, which indicates whether or not the variable being processed was declared as a _var type.

Naming Conventions for gen_

The names of some commands contain gen_, to indicate that they generate output into the current output file. For example, cpp_gen_op_sig_h outputs the C++ signature of an operation for use in a header file. Commands whose names omit gen_ return a value—which you can use as a parameter to the output command.

Examples

Some commands whose names do not contain gen_ also have gen_ counterparts. Both forms are provided to offer greater flexibility in how you write scripts. In particular, commands without gen_ are easy to embed inside textual blocks (that is, text inside [*** and ***]), while their gen_ counterparts are sometimes easier to call from outside textual blocks. Some examples follow:

- The following segment of code prints the C++ signatures of all the operations of an interface, for use in a .h file:

  ```
  # Tcl
  foreach op [$inter contents {operation}] {
      output "    [cpp_op_sig_h $op];\n"
  }
  ```

  Note that the output statement uses spaces to indent the signature of the operation, and follows it with a semicolon and newline character. The printing of all this white space and syntactic baggage is automated by the gen_ counterpart of this command, so the above code snippet could be rewritten in the following, slightly more concise format:

  ```
  # Tcl
  foreach op [$inter contents {operation}] {
      cpp_gen_op_sig_h $op
  }
  ```

- The cpp_gen_ commands tend to be useful inside foreach loops to, for example, declare operation signatures or variables. However, when generating the bodies of operations in .cpp files, it is likely that you will be making use of a textual block.

In such cases, it can be a nuisance to have to exit the textual block just to call a Tcl command, and then enter another textual block to print more text. For example:

```
# Tcl
[***
//--------
// Function: ...
//--------
***]
cpp_gen_op_sig_cc $op
[***
{
    ... // body of the operation
}
***]
```

- The use of commands without `gen_` can often eliminate the need to toggle in and out of textual blocks. For example, the above segment of code can be written in the following, more concise form:

```
# Tcl
[***
//--------
// Function: ...
//--------
@[cpp_op_sig_cc $op]@
{
    ... // body of the operation
}
***]
```

Indentation

To allow programmers to choose their preferred indentation, all command procedures in `std/cpp_poa_lib.tcl` use the string in `$pref(cpp,indent)` for each level of indentation they need to generate.

Some commands take a parameter called `ind_lev`. This parameter is an integer that specifies the indentation level at which output should be generated.

$pref(cpp,...) Entries

Some entries in the `$pref(cpp,...)` array are used to specify various user preferences for the generation of C++ code, as shown in Table 14. All of these entries are have a default values if there is no setting in the `idlgen.cfg` file. You can also force the setting by explicit assignment in a Tcl script.

Table 14: *$pref(cpp,...) Array Entries*

$pref(...) Array Entry	Purpose
`$pref(cpp,h_file_ext)`	Specifies the filename extension for header files. Its default value is `.h`.

Table 14: *$pref(cpp,...) Array Entries*

$pref(...) Array Entry	Purpose
`$pref(cpp,cc_file_ext)`	Specifies the filename extension for code files. Its default value is `.cxx`.
`$pref(cpp,indent)`	Specifies the amount of white space to be used for one level of indentation. Its default value is four spaces.
`$pref(cpp,impl_class_suffix)`	Specifies the suffix that is added to the name of a class that implements an IDL interface. Its default value is `Impl`.
`$pref(cpp,factory_suffix)`	Specifies the suffix that is added to the name of a valuetype factory class. Its default value is `Factory`.
`$pref(cpp,attr_mod_param_name)`	Specifies the name of the parameter in the C++ signature of an attribute's modifier operation. Its default value is `_new_value`.
`$pref(cpp,ret_param_name)`	Specifies the name of the variable that is to be used to hold the return value from a non-`void` operation call. Its default value is `_result`.
`$pref(cpp,max_padding_for_types)`	Specifies the padding to be used with C++ type names when declaring variables or parameters. This padding helps to ensure that the names of variables and parameters are vertically aligned, which makes code easier to read. Its default value is `32`.

Groups of Related Commands

To help you find the commands needed for a particular task, each heading below lists a group of related commands.

Identifiers and Keywords

`cpp_l_name` *node*
`cpp_s_name` *node*
`cpp_typecode_s_name` *type*
`cpp_typecode_l_name` *type*

General Purpose Commands

`cpp_assign_stmt` *type name value ind_lev ?scope?*
`cpp_indent` *number*
`cpp_is_fixed_size` *type*
`cpp_is_keyword` *name*
`cpp_is_var_size` *type*
`cpp_nil_pointer` *type*
`cpp_sanity_check_idl`

Servant/Implementation Classes

`cpp_impl_class` *interface_node*
`cpp_poa_class_s_name` *interface_node*

cpp_poa_tie_s_name *interface_node*

Operation Signatures

cpp_gen_op_sig_cc *operation_node ?class_name?*
cpp_gen_op_sig_h *operation_node*
cpp_op_sig_cc *operation_node ?class_name?*
cpp_op_sig_h *operation_node*

Attribute Signatures

cpp_attr_acc_sig_cc *attribute_node ?class_name?*
cpp_attr_acc_sig_h *attribute_node*
cpp_attr_mod_sig_cc *attribute_node ?class_name?*
cpp_attr_mod_sig_h *attribute_node*
cpp_gen_attr_acc_sig_cc *attribute_node ?class_name?*
cpp_gen_attr_acc_sig_h *attribute_node*
cpp_gen_attr_mod_sig_cc *attribute_node ?class_name?*
cpp_gen_attr_mod_sig_h *attribute_node*

Types and Signatures of Parameters

cpp_param_sig *name type direction*
cpp_param_sig *op_or_arg*
cpp_param_type *type direction*
cpp_param_type *op_or_arg*

Invoking Operations

cpp_assign_stmt *type name value ind_lev ?scope?*
cpp_clt_free_mem_stmt *arg_or_op is_var*
cpp_clt_need_to_free_mem *arg_or_op is_var*
cpp_clt_par_decl *arg_or_op is_var*
cpp_clt_par_ref *arg_or_op is_var*
cpp_gen_clt_free_mem_stmt *arg_or_op is_var ind_lev*
cpp_gen_clt_par_decl *arg_or_op is_var ind_lev*
cpp_ret_assign *op*

Invoking Attributes

cpp_clt_free_mem_stmt *name type dir is_var*
cpp_clt_need_to_free_mem *name type dir is_var*
cpp_clt_par_decl *name type dir is_var*
cpp_clt_par_ref *name type dir is_var*
cpp_gen_clt_free_mem_stmt *name type dir is_var ind_lev*
cpp_gen_clt_par_decl *name type dir is_var ind_lev*

Implementing Operations

cpp_gen_srv_free_mem_stmt *arg_or_op ind_lev*
cpp_gen_srv_par_alloc *arg_or_op ind_lev*
cpp_gen_srv_ret_decl *op ind_lev ?alloc_mem?*

cpp_srv_free_mem_stmt *arg_or_op*
cpp_srv_need_to_free_mem *arg_or_op*
cpp_srv_par_alloc *arg_or_op*
cpp_srv_par_ref *arg_or_op*
cpp_srv_ret_decl *op ?alloc_mem?*

Implementing Attributes

cpp_gen_srv_free_mem_stmt *name type direction ind_lev*
cpp_gen_srv_par_alloc *name type direction ind_lev*
cpp_gen_srv_ret_decl *name type ind_lev ?alloc_mem?*
cpp_srv_free_mem_stmt *name type direction*
cpp_srv_need_to_free_mem *type direction*
cpp_srv_par_alloc *name type direction*
cpp_srv_par_ref *name type direction*
cpp_srv_ret_decl *name type ?alloc_mem?*

Instance Variables and Local Variables

cpp_var_decl *name type is_var*
cpp_var_free_mem_stmt *name type is_var*
cpp_var_need_to_free_mem *type is_var*

Processing Unions

cpp_branch_case_l_label *union_branch*
cpp_branch_case_s_label *union_branch*
cpp_branch_l_label *union_branch*
cpp_branch_s_label *union_branch*

Processing Arrays

cpp_array_decl_index_vars *arr pre ind_lev*
cpp_array_elem_index *arr pre*
cpp_array_for_loop_footer *arr indent*
cpp_array_for_loop_header *arr pre ind_lev ?decl?*
cpp_gen_array_decl_index_vars *arr pre ind_lev*
cpp_gen_array_for_loop_footer *arr indent*
cpp_gen_array_for_loop_header *arr pre ind_lev ?decl?*

Processing Any

cpp_any_insert_stmt *type any_name value ?is_var?*
cpp_any_extract_stmt *type any_name name*
cpp_any_extract_var_decl *type name*
cpp_any_extract_var_ref *type name*

cpp_poa_lib Commands

This section gives detailed descriptions of the Tcl commands in the `cpp_poa_lib` library in alphabetical order.

cpp_any_extract_stmt

Synopsis

`cpp_any_extract_stmt` *`type any_name var_name`*

Description

This command generates a statement that extracts the value of the specified *`type`* from the `any` called *`any_name`* into the *`var_name`* variable.

Parameters

type	A type node of the parse tree.
any_name	The name of the `any` variable.
var_name	The name of the variable into which the `any` is extracted.

Notes

`var_name` must be a variable declared by `cpp_any_extract_var_decl`.

Examples

The following example shows how to use the `any` extraction commands:

```
# Tcl
foreach type $type_list {
    set var_name my_[$type s_uname]
[***
@[cpp_any_extract_var_decl $type $var_name]@;
***]
}
output "\n"
foreach type $type_list {
    set var_name my_[$type s_uname]
    set var_ref  [cpp_any_extract_var_ref $type $var_name]
[***
if (@[cpp_any_extract_stmt $type "an_any" $var_name]@) {
    process_@[$type s_uname]@(@$var_ref@);
}
***]
}
```

If the variable `type_list` contains the type nodes for `widget` (a `struct`), `boolean` and `long_array`, the previous Tcl script generates the following C++ code:

```
// C++
widget * my_widget;
CORBA::Boolean my_boolean;
long_array_slice* my_long_array;

if (an_any >>= my_widget) {
        process_widget(*my_widget);
}
if (an_any >>= CORBA::Any::to_boolean(my_boolean)) {
        process_boolean(my_boolean);
}
if (an_any >>= long_array_forany(my_long_array)) {
        process_long_array(my_long_array);
}
```

See Also `cpp_any_insert_stmt`
`cpp_any_extract_var_decl`
`cpp_any_extract_var_ref`

cpp_any_extract_var_decl

Synopsis `cpp_any_extract_var_decl` *`type name`*

Description This command declares a variable into which values from an `any` are extracted. The parameters to this command are the variable's *`type`* and *`name`*.

Parameters

type	A type node of the parse tree.
name	The name of the variable.

Notes If the value to be extracted is a simple type, such as a `short`, `long`, or `boolean`, the variable is declared as a normal variable of the specified `type`. However, if the value is a complex type such as `struct` or `sequence`, the variable is declared as a pointer to the specified `type`.

Examples The following example shows how to use the `cpp_any_extract_var_decl` command:

```
# Tcl
foreach type $type_list {
    set var_name my_[$type s_uname]
[***
@[cpp_any_extract_var_decl $type $var_name]@;
***]
}
```

If the `type_list` variable contains the type nodes for `widget` (a `struct`), `boolean`, and `long_array`, the previous Tcl script generates the following C++ code:

```
// C++
widget * my_widget;
CORBA::Boolean my_boolean;
long_array_slice* my_long_array;
```

See Also `cpp_any_insert_stmt`
`cpp_any_extract_var_ref`
`cpp_any_extract_stmt`

cpp_any_extract_var_ref

Synopsis `cpp_any_extract_var_ref` *`type name`*

Description This command returns a reference to the value in *`name`* of the specified *`type`*.

Parameters

type	A type node of the parse tree.
name	The name of the variable.

Notes

The returned reference is either `$name` or `*$name`, depending on how the variable is declared by the `cpp_any_extract_var_decl` command. If `type` is a struct, union, or sequence type, the command returns `*$name`; otherwise it returns `$name`.

Examples

The following example shows how to use the `cpp_any_extract_var_ref` command:

```
# Tcl
foreach type $type_list {
    set var_name my_[$type s_uname]
    set var_ref  [cpp_any_extract_var_ref $type $var_name]
[***
process_@[$type s_uname]@(@$var_ref@);
***]
}
```

If the `type_list` variable contains the type nodes for `widget` (a struct), `boolean`, and `long_array` then the previous Tcl script generates the following C++ code:

```
// C++
process_widget(*my_widget);
process_boolean(my_boolean);
process_long_array(my_long_array);
```

See Also

`cpp_any_insert_stmt`
`cpp_any_extract_var_decl`
`cpp_any_extract_stmt`

cpp_any_insert_stmt

Synopsis

`cpp_any_insert_stmt` *type any_name value ?is_var?*

Description

This command returns the C++ statement that inserts the specified *value* of the specified *type* into the `any` called *any_name*.

Parameters

type	A type node of the parse tree.
any_name	The name of the `any` variable.
value	The name of the variable that is being inserted into the `any`.
is_var	TRUE if *value* is a `_var` variable.

Notes

If the *is_var* parameter is TRUE, the generated statement includes a call to the `in()` operation defined for all `_var` types, if necessary. This is necessary in some cases for some C++ compilers, to prevent compiler errors related to ambiguous operation overloading or multiple implicit conversions.

Examples

The following Tcl fragment shows how the command is used:

```
# Tcl
...
foreach type $type_list {
        set var_name my_[$type s_uname]
[***
@[cpp_any_insert_stmt $type "an_any" $var_name]@;
***]
}
```

If the type_list variable contains the type nodes for widget (a struct), boolean, and long_array, the previous Tcl script will generate the following C++ code:

```
// C++
an_any <<= my_widget;
an_any <<= CORBA::Any::from_boolean(my_boolean);
an_any <<= long_array_forany(my_long_array);
```

See Also

cpp_any_extract_var_decl
cpp_any_extract_var_ref
cpp_any_extract_stmt

cpp_array_decl_index_vars

Synopsis

cpp_array_decl_index_vars *array prefix ind_lev*

cpp_gen_array_decl_index_vars *array prefix ind_lev*

Description

This command declares the set of index variables that are used to index the specified *array*.

Parameters

array	An array node in the parse tree.
prefix	The prefix to be used when constructing the names of index variables. For example, the prefix i is used to get index variables called i1 and i2.
ind_lev	The indentation level at which the for loop is to be created.

Notes

The array indices are declared to be of type CORBA::ULong.

Examples

The following Tcl script illustrates the use of the command:

Example 27:

```
# Tcl
set typedef  [$idlgen(root) lookup "long_array"]
set a        [$typedef true_base_type]
1 set indent   [cpp_indent [$a num_dims]]
```

Example 27:

```
2 set index    [cpp_array_elem_index $a "i"]
[***
void some_func()
{
    @[cpp_array_decl_index_vars $a "i" 1]@

    @[cpp_array_for_loop_header $a "i" 1]@
    @$indent@foo@$index@ = bar@$index@;
    @[cpp_array_for_loop_footer $a 1]@
}
***]
```

The amount of indentation to be used inside the body of the `for` loop, line 2, is calculated by using the number of dimensions in the array as a parameter to the `cpp_indent` command, line 1. The above Tcl script generates the following C++ code:

```
// C++
void some_func()
{
    CORBA::ULong                    i1;
    CORBA::ULong                    i2;
    for (i1 = 0; i1 < 5; i1 ++) {
        for (i2 = 0; i2 < 7; i2 ++) {
            foo[i1][i2] = bar[i1][i2];
        }
    }
}
```

See Also

cpp_gen_array_decl_index_vars
cpp_array_for_loop_header
cpp_array_elem_index
cpp_array_for_loop_footer

cpp_array_elem_index

Synopsis

`cpp_array_elem_index` *array prefix*

Description

This command returns, in square brackets, the complete set of indices required to index a single element of *array*.

Parameters

array	An array node in the parse tree.
prefix	The prefix to use when constructing the names of index variables. For example, the prefix `i` is used to get index variables called `i1` and `i2`.

Examples

If `arr` is a two-dimensional array node, the following Tcl fragment:

```
# Tcl
...
set indices  [cpp_array_elem_index $arr "i"]
```

sets `indices` equal to the string, `"[i1][i2]"`.

See Also

cpp_array_decl_index_vars
cpp_array_for_loop_header

cpp_array_for_loop_footer

Synopsis

`cpp_array_for_loop_footer` *`array ind_lev`*

`cpp_gen_array_for_loop_footer` *`array ind_lev`*

Description

This command generates the `for` loop footer for the given *`array`* node, with indentation specified by *`ind_level`*.

Parameters

array	An array node in the parse tree.
ind_lev	The indentation level at which the `for` loop is created.

Notes

This command generates a number of close braces `'}'` that equals the number of dimensions of the array.

See Also

`cpp_array_decl_index_vars`

`cpp_array_for_loop_header`
`cpp_array_elem_index`

cpp_array_for_loop_header

Synopsis

`cpp_array_for_loop_header` *`array prefix ind_lev ?declare?`*

`cpp_gen_array_for_loop_header` *`array prefix ind_lev ?declare?`*

Description

This command generates the `for` loop header for the given *`array`* node.

Parameters

array	An array node in the parse tree.
prefix	The prefix to be used when constructing the names of index variables. For example, the prefix `i` is used to get index variables called `i1` and `i2`.
ind_lev	The indentation level at which the `for` loop is created.
declare	(Optional) This boolean argument specifies that index variables are declared locally within the `for` loop. Default value is `0`.

Examples

Given the following IDL definition of an array:

```
// IDL
typedef long long_array[5][7];
```

You can use the following Tcl fragment to generate the for loop header:

```
# Tcl
...
set typedef  [$idlgen(root) lookup "long_array"]
set a        [$typedef true_base_type]
[***
    @[cpp_array_for_loop_header $a "i" 1]@
***]
```

This generates the following C++ code:

```
// C++
    for (i1 = 0; i1 < 5; i1 ++) {
        for (i2 = 0; i2 < 7; i2 ++) {
```

Alternatively, using the command `cpp_array_for_loop_header $a "i" 1 1` results in the following C++ code:

```
// C++
    for (CORBA::ULong i1 = 0; i1 < 5; i1 ++) {
        for (CORBA::ULong i2 = 0; i2 < 7; i2 ++) {
```

See Also

cpp_array_decl_index_vars
cpp_gen_array_for_loop_header
cpp_array_elem_index
cpp_array_for_loop_footer

cpp_assign_stmt

Synopsis

cpp_assign_stmt *type name value ind_lev ?scope?*

cpp_gen_assign_stmt *type name value ind_lev ?scope?*

Description

This command returns the C++ statement (with the terminating ';') that assigns *value* to the variable *name*, where both are of the same *type*.

Parameters

type	A type node of the parse tree.
name	The name of the variable that is assigned to (left hand side of assignment).
value	A variable reference that is assigned from (right hand side of assignment).
ind_lev	The number of levels of indentation.
scope	(Optional) When performing assignment of arrays, the scope flag determines whether or not the body of the generated for loop is enclosed in curly braces '{', '}'. The default value is 1 (TRUE).

Notes

The assignment performs a deep copy. For example, if `type` is a `string` or `interface` then a `string_dup()` or `_duplicate()`, respectively, is performed on the *value*.

The *ind_lev* and *scope* parameters are ignored for all assignment statements, except those involving arrays. In the case of array assignments, a `for` loop is generated, to perform an element-wise copy of the array's contents. The `ind_lev` (indentation level) parameter is required, because the returned `for` loop spans several lines of code, and these lines of code need to be indented consistently. The *scope* parameter is a boolean (with a default value of 1) that specifies whether or not an extra scope (that is, a pair of braces '{', '}') should surround the `for` loop. This extra level of scoping is a workaround for a scoping-related bug in some C++ compilers.

Examples

The following example illustrates the use of this command:

```
# Tcl
set is_var 0
set ind_lev 1
[***
void some_func()
{
***]
foreach type $type_list {
        set name  "my_[$type l_name]"
        set value "other_[$type l_name]"
[***
       @[cpp_assign_stmt $type $name $value $ind_lev 0]@
***]
}
[***
} // some_func()
***]
```

If the variable `type_list` contains the type nodes for `string`, `widget` (a struct), and `long_array`, the above Tcl script generates the following C++ code:

```
// C++
void some_func()
{
        my_string = CORBA::string_dup(other_string);
        my_widget = other_widget;
        for (CORBA::ULong i1 = 0; i1 < 10; i1 ++) {
            my_long_array[i1] = other_long_array[i1];
        }
} // some_func()
```

Note that the `cpp_assign_stmt` command (and its `gen_` counterpart) expect the `name` and `value` parameters to be references (rather than pointers). For example, if the variable `my_widget` is a pointer to a struct (rather than an actual struct) then the `name` parameter to `cpp_gen_assign_stmt` should be `*my_widget` instead of `my_widget`.

See Also

`cpp_gen_assign_stmt`

`cpp_assign_stmt_array`
`cpp_clt_par_ref`

cpp_attr_acc_sig_h

Synopsis

`cpp_attr_acc_sig_h` *`attribute`*

`cpp_gen_attr_acc_sig_h` *`attribute`*

Description

This command returns the signature of an attribute accessor operation for inclusion in a `.h` file.

Parameters

`attribute`	An attribute node in the parse tree.

Notes

The `cpp_attr_acc_sig_h` command has no `';'` (semicolon) at the end of its generated statement.

The `cpp_gen_attr_acc_sig_h` command includes a `';'` (semicolon) at the end of its generated statement.

Examples

Consider the following sample IDL:

```
// IDL
// File: 'finance.idl'
interface Account {
    attribute long accountNumber;
    attribute float balance;
    void makeDeposit(in float amount);
};
```

The following Tcl script illustrates the use of the command:

```
# Tcl
smart_source "std/sbs_output.tcl"
smart_source "std/cpp_poa_lib.tcl"

if { ! [idlgen_parse_idl_file "finance.idl"] } {
    exit 1
}

set attr [$idlgen(root) lookup "Account::balance"]
set attr_acc_sig_h  [cpp_attr_acc_sig_h  $attr]

output "$attr_acc_sig_h \n\n"

cpp_gen_attr_acc_sig_h  $attr
```

The following output is generated by the Tcl script:

```
virtual CORBA::Float balance()
   throw(CORBA::SystemException)

virtual CORBA::Float balance()
   throw(CORBA::SystemException);
```

See Also

`cpp_gen_attr_acc_sig_h`
`cpp_attr_acc_sig_cc`
`cpp_attr_mod_sig_h`
`cpp_attr_mod_sig_cc`

cpp_attr_acc_sig_cc

Synopsis

`cpp_attr_acc_sig_cc` *`attribute ?class?`*

`cpp_gen_attr_acc_sig_cc` *`attribute ?class?`*

Description

This command returns the signature of an attribute accessor operation, for inclusion in a `.cc` file.

Parameters

attribute	An attribute node in the parse tree.
?class?	(Optional) The name of the class in which the accessor operation is defined. If no class is specified, the default implementation class name is used instead (given by `[cpp_impl_class [$op defined_in]]`).

Notes

Neither the `cpp_attr_acc_sig_cc` nor the `cpp_gen_attr_acc_sig_cc` command put a `';'` (semicolon) at the end of the generated statement.

Examples

Consider the following sample IDL:

```
// IDL
// File: 'finance.idl'
interface Account {
    attribute long accountNumber;
    attribute float balance;
    void makeDeposit(in float amount);
};
```

The following Tcl script illustrates the use of the command:

```
# Tcl
smart_source "std/sbs_output.tcl"
smart_source "std/cpp_poa_lib.tcl"

if { ! [idlgen_parse_idl_file "finance.idl"] } {
    exit 1
}

set attr [$idlgen(root) lookup "Account::balance"]
set attr_acc_sig_cc [cpp_attr_acc_sig_cc $attr]

output "$attr_acc_sig_cc \n\n"

cpp_gen_attr_acc_sig_cc $attr
```

The following output is generated by the Tcl script:

```
CORBA::Float
AccountImpl::balance() throw(CORBA::SystemException)

CORBA::Float
AccountImpl::balance() throw(CORBA::SystemException)
```

See Also

`cpp_attr_acc_sig_h`
`cpp_gen_attr_acc_sig_cc`
`cpp_attr_mod_sig_h`
`cpp_attr_mod_sig_cc`

cpp_attr_mod_sig_h

Synopsis

`cpp_attr_mod_sig_h` *`attribute`*

`cpp_gen_attr_mod_sig_h` *`attribute`*

Description

This command returns the signature of an attribute modifier operation for inclusion in a `.h` file.

Parameters

`attribute`	Attribute node in parse tree.

Notes

The command `cpp_attr_mod_sig_h` has no `';'` (semicolon) at the end of its generated statement.

The related command cpp_gen_attr_mod_sig_h does include a ';' (semicolon) at the end of its generated statement.

Examples

Consider the following sample IDL:

```
// IDL
// File: 'finance.idl'
interface Account {
    attribute long accountNumber;
    attribute float balance;
    void makeDeposit(in float amount);
};
```

The following Tcl script illustrates the use of the command:

```
# Tcl
smart_source "std/sbs_output.tcl"
smart_source "std/cpp_poa_lib.tcl"

if { ! [idlgen_parse_idl_file "finance.idl"] } {
    exit 1
}

set attr_mod_sig_h  [cpp_attr_mod_sig_h  $attr]
output "$attr_mod_sig_h \n\n"

cpp_gen_attr_mod_sig_h  $attr
```

The following output is generated by the Tcl script:

```
virtual void balance(CORBA::Float _new_value)
   throw(CORBA::SystemException)

    virtual void balance(CORBA::Float _new_value)
      throw(CORBA::SystemException);
```

See Also

cpp_attr_acc_sig_h

cpp_attr_acc_sig_cc
cpp_attr_mod_sig_cc

cpp_attr_mod_sig_cc

Synopsis

cpp_attr_mod_sig_cc *attribute ?class?*

cpp_gen_attr_mod_sig_cc *attribute ?class?*

Description

This command returns the signature of the attribute modifier operation for inclusion in a .cc file.

Parameters

attribute	An attribute node in the parse tree.
?class?	(Optional) The name of the class in which the modifier operation is defined. If no class is specified, the default implementation class name is used instead (given by [cpp_impl_class [$op defined_in]]).

Notes

Neither the cpp_attr_mod_sig_cc nor the cpp_gen_attr_mod_sig_cc put a ';' (semicolon) at the end of the generated statement.

Examples

Consider the following sample IDL:

```
// IDL
// File: 'finance.idl'
interface Account {
    attribute long accountNumber;
    attribute float balance;
    void makeDeposit(in float amount);
};
```

The following Tcl script illustrates the use of the command:

```
# Tcl
smart_source "std/sbs_output.tcl"
smart_source "std/cpp_poa_lib.tcl"

if { ! [idlgen_parse_idl_file "finance.idl"] } {
    exit 1
}

set attr_mod_sig_cc [cpp_attr_mod_sig_cc $attr]
output "$attr_mod_sig_cc \n\n"

cpp_gen_attr_mod_sig_cc $attr
```

The following output is generated by the Tcl script:

```
void AccountImpl::balance(
    CORBA::Float                    _new_value
) throw(CORBA::SystemException)

void AccountImpl::balance(
    CORBA::Float                    _new_value
) throw(CORBA::SystemException)
```

See Also

`cpp_attr_acc_sig_h`
`cpp_attr_acc_sig_cc`
`cpp_attr_mod_sig_h`
`cpp_gen_attr_mod_sig_cc`

cpp_branch_case_l_label

Synopsis

`cpp_branch_case_l_label` *`union_branch`*

Description

This command returns a non-scoped C++ case label for the union branch *`union_branch`*. The `case` keyword prefixes the label unless the label is `default`. The returned value omits the terminating `':'` (colon).

Parameters

`union_branch`	A `union_branch` node of the parse tree.

Notes

This command generates case labels for all union discriminator types.

Examples Consider the following IDL:

```
// IDL
module m {
        enum colour {red, green, blue};

        union foo switch(colour) {
            case red:    long     a;
            case green:  string   b;
            default:     short    c;
        };
};
```

The following Tcl script illustrates the use of the command:

```
# Tcl
...
set union [$idlgen(root) lookup "m::foo"]
foreach branch [$union contents {union_branch}] {
        output [cpp_branch_case_l_label $branch]
        output "\n"
}; # foreach
```

The following output is generated by the Tcl script:

```
// C++
case red
case green
default
```

See Also

cpp_branch_l_label
cpp_branch_case_s_label
cpp_branch_s_label

cpp_branch_l_label

Synopsis cpp_branch_l_label *union_branch*

Description This command returns the non-scoped C++ case label for the union branch *union_branch*. The case keyword and the terminating ':' (colon) are both omitted.

Parameters

union_branch	A union_branch node of the parse tree.

Notes This command generates case labels for all union discriminator types.

Examples

Consider the following IDL:

```
// IDL
module m {
        enum colour {red, green, blue};

        union foo switch(colour) {
            case red:    long     a;
            case green:  string   b;
            default:     short    c;
        };
};
```

The following Tcl script illustrates the use of the command:

```
# Tcl
...
set union [$idlgen(root) lookup "m::foo"]
foreach branch [$union contents {union_branch}] {
        output [cpp_branch_l_label $branch]
        output "\n"
}; # foreach
```

The following output is generated by the Tcl script:

```
// C++
red
green
default
```

See Also

cpp_branch_case_l_label

cpp_branch_case_s_label
cpp_branch_s_label

cpp_branch_case_s_label

Synopsis

cpp_branch_case_s_label *union_branch*

Description

This command returns a scoped C++ case label for the union branch *union_branch*. The `case` keyword prefixes the label unless the label is `default`.The returned value omits the terminating `':'` (colon).

Parameters

union_branch	A `union_branch` node of the parse tree.

Notes

This command generates case labels for all union discriminator types.

Examples

Consider the following IDL:

```
// IDL
module m {
        enum colour {red, green, blue};

        union foo switch(colour) {
            case red:    long     a;
            case green:  string   b;
            default:     short    c;
        };
};
```

The following Tcl script illustrates the use of the command:

```
# Tcl
...
set union [$idlgen(root) lookup "m::foo"]
foreach branch [$union contents {union_branch}] {
        output [cpp_branch_case_s_label $branch]
        output "\n"
}; # foreach
```

The following output is generated by the Tcl script:

```
// C++
case m::red
case m::green
default
```

See Also

cpp_branch_case_l_label
cpp_branch_l_label
cpp_branch_s_label

cpp_branch_s_label

Synopsis

`cpp_branch_s_label` *union_branch*

Description

Returns a scoped C++ case label for the *union_branch* union branch. The `case` keyword and the terminating `':'` (colon) are both omitted.

Parameters

union_branch	A `union_branch` node of the parse tree.

Notes

This command generates case labels for all union discriminator types.

Examples

Consider the following IDL:

```
// IDL
module m {
        enum colour {red, green, blue};

        union foo switch(colour) {
            case red:    long     a;
            case green:  string   b;
            default:     short    c;
        };
};
```

The following Tcl script illustrates the use of the command:

```
# Tcl
...
set union [$idlgen(root) lookup "m::foo"]
foreach branch [$union contents {union_branch}] {
        output [cpp_branch_s_label $branch]
        output "\n"
}; # foreach
```

The following output is generated by the Tcl script:

```
// C++
m::red
m::green
default
```

See Also

`cpp_branch_case_l_label`
`cpp_branch_l_label`
`cpp_branch_case_s_label`

cpp_clt_free_mem_stmt

Synopsis

`cpp_clt_free_mem_stmt` *name type direction is_var*
`cpp_clt_free_mem_stmt` *arg is_var*
`cpp_clt_free_mem_stmt` *op is_var*
`cpp_gen_clt_free_mem_stmt` *name type direction is_var*
`cpp_gen_clt_free_mem_stmt` *arg is_var*
`cpp_gen_clt_free_mem_stmt` *op is_var*

Description

This command returns a C++ statement that frees the memory associated with the specified parameter (or return value) of an operation.

Parameters

name	The name of the parameter or return value variable.
type	A type node of the parse tree that describes the type of this parameter or return value.
direction	The parameter passing mode—one of `in`, `inout`, `out`, or `return`.
is_var	A boolean flag to indicate whether the parameter variable is a `_var` type or not. A value of `1` indicates a `_var` type.
arg	An argument node of the parse tree.
op	An operation node of the parse tree.

Notes

The following variants of the command are supported:

- The first form of the command is used to free memory associated with an explicitly named parameter variable.
- The second form of the command is used to free memory associated with parameters.
- The third form of the command is used to free memory associated with return values.

- The non-gen forms of the command omit the terminating ';' (semicolon) character.
- The gen forms of the command include the terminating ';' (semicolon) character.

If there is no need to free memory for the parameter (for example, if *is_var* is 1 or if the parameter's type or direction does not require any memory management) this command returns an empty string.

Examples

This example uses the following sample IDL:

```
// IDL
struct  widget           {long a;};
typedef sequence<long>   longSeq;
typedef long             long_array[10];

interface foo {
    longSeq op(
        in widget       p_widget,
        inout string    p_string,
        out longSeq     p_longSeq,
        out long_array  p_long_array);
};
```

The following Tcl script shows how to free memory associated with the parameters and the return value of the foo::op() union branch.

Example 28:

```
# Tcl
...
[***
    //--------
    // Free memory associated with parameters
    //--------
***]
foreach arg $arg_list {
    set name [cpp_l_name $arg]
1   cpp_gen_clt_free_mem_stmt $arg $is_var $ind_lev
}
2 cpp_gen_clt_free_mem_stmt $op $is_var $ind_lev
```

The $arg_list contains the list of argument nodes corresponding to the foo::op() operation. To illustrate explicit memory management, the example assumes that is_var is set to FALSE. Notice how the cpp_gen_clt_free_mem_stmt command is used to free memory both for the parameters, line 1, and the return value, line 2.

The Tcl code yields the following statements that explicitly free memory:

```
//--------
// Free memory associated with parameters
//--------
CORBA::string_free(p_string);
delete p_longSeq;
delete _result;
```

Statements to free memory are generated only if needed. For example, there is no memory-freeing statement generated for `p_widget` or `p_long_array`, because these parameters have their memory allocated on the stack rather than on the heap.

See Also

`cpp_gen_clt_free_mem_stmt`

`cpp_clt_need_to_free_mem`

cpp_clt_need_to_free_mem

Synopsis

`cpp_clt_need_to_free_mem` *`arg is_var`*

`cpp_clt_need_to_free_mem` *`op is_var`*

Description

This command returns `1` (TRUE) if the client programmer has to take explicit steps to free memory. Returns `0` (FALSE) otherwise.

Parameters

arg	An argument node of the parse tree.
op	An operation node of the parse tree.
is_var	A boolean flag to indicate whether the parameter variable is a `_var` type or not. A value of `1` indicates a `_var` type.

Notes

The following variants of the command are supported:

- The first form of the command is used to check parameters.
- The second form of the command is used to check return values.

See Also

`cpp_clt_free_mem_stmt`

cpp_clt_par_decl

Synopsis

`cpp_clt_par_decl` *`name type direction is_var`*

```
cpp_clt_par_decl arg is_var
cpp_clt_par_decl op is_var
cpp_gen_clt_par_decl name type direction is_var ind_lev
cpp_gen_clt_par_decl arg is_var ind_lev
cpp_gen_clt_par_decl op is_var ind_lev
```

Description

This command returns a C++ statement that declares a parameter or return value variable.

Parameters

name	The name of a parameter or return value variable.
type	A type node of the parse tree that describes the type of this parameter or return value.
direction	The parameter passing mode—one of `in`, `inout`, `out` or `return`.
is_var	A boolean flag to indicate whether the parameter variable is a `_var` type or not. A value of `1` indicates a `_var` type.
arg	An argument node of the parse tree.
op	An operation node of the parse tree.

ind_lev	Number of levels of indentation (gen variants only).

Notes

The following variants of the command are supported:

- The first form of the command is used to declare an explicitly named parameter variable.
- The second form is used to declare a parameter. The third form is used to declare a return value.
- The non-gen forms of the command omit the terminating ';' (semicolon) character.
- The gen forms of the command include the terminating ';' (semicolon) character.

For most parameter declarations, *is_var* is ignored and space for the parameter is allocated on the stack. However, if the parameter is a string or an object reference being passed in any direction, or if it is one of several types of out parameter that must be heap-allocated, the *is_var* parameter determines whether to declare the parameter as a _var or a normal pointer.

Examples

The following IDL is used in this example:

```
// IDL
struct  widget           {long a;};
typedef sequence<long>   longSeq;
typedef long             long_array[10];

interface foo {
    longSeq op(
        in widget       p_widget,
        inout string    p_string,
        out longSeq     p_longSeq,
        out long_array  p_long_array);
};
```

The following Tcl script illustrates how to declare C++ variables that are intended to be used as parameters to (or the return value of) an operation call:

```
# Tcl
...
set op         [$idlgen(root) lookup "foo::op"]
set is_var     0
set ind_lev    1
set arg_list   [$op contents {argument}]
[***
    //--------
    // Declare parameters for operation
    //--------
***]
foreach arg $arg_list {
    cpp_gen_clt_par_decl $arg $is_var $ind_lev
}
cpp_gen_clt_par_decl $op $is_var $ind_lev
```

This Tcl script generates the following C++ code:

Example 29:

```
//--------
// Declare parameters for operation
//--------
widget p_widget;
1 char * p_string;
2 longSeq* p_longSeq;

long_array p_long_array;
3 longSeq* _result;
```

Line 3 declares the name of the return value to be `_result`, which is the default value of the variable `$pref(cpp,ret_param_name)`. In lines 1, 2, and 3, the C++ variables are declared as raw pointers. This is because the `is_var` parameter is FALSE in calls to `cpp_gen_clt_par_decl`. If `is_var` is TRUE, the variables are declared as `_var` types.

See Also

`cpp_gen_clt_par_decl`

`cpp_clt_par_ref`

cpp_clt_par_ref

Synopsis

`cpp_clt_par_ref` *`name type direction is_var`*

`cpp_clt_par_ref` *`arg is_var`*
`cpp_clt_par_ref` *`op is_var`*

Description

This command returns either `$name` or `*$name`, whichever is necessary to get a reference to the actual data (as opposed to a pointer to the data).

Parameters

name	The name of the parameter or return value variable.
type	A type node of the parse tree that describes the type of this parameter or return value.
direction	The parameter passing mode—one of `in`, `inout`, `out`, or `return`.
is_var	A boolean flag to indicate whether the parameter variable is a `_var` type or not. A value of `1` indicates a `_var` type.
arg	An argument node of the parse tree.
op	An operation node of the parse tree.

Notes

This command is intended to be used in conjunction with `cpp_clt_par_decl` and `cpp_assign_stmt`. If a parameter (or return value) variable has been declared, using the command `cpp_clt_par_decl`, a reference to that parameter (or return value) is obtained, using the command `cpp_clt_par_ref`.

References returned by `cpp_clt_par_ref` are intended for use in the context of assignment statements, in conjunction with the command `cpp_gen_assign_stmt`. See the following example.

Examples

Given the following IDL:

```
// IDL
struct  widget            {long a;};
typedef sequence<long>    longSeq;
typedef long              long_array[10];

interface foo {
    longSeq op(
        in widget       p_widget,
        inout string    p_string,
        out longSeq     p_longSeq,
        out long_array  p_long_array);
};
```

The following Tcl script shows how to initialize `in` and `inout` parameters:

Example 30:

```
# Tcl
...
[***
    //--------
    // Initialize "in" and "inout" parameters
    //--------
***]
1 foreach arg [$op args {in inout}] {
    set type [$arg type]
2     set arg_ref [cpp_clt_par_ref $arg $is_var]
    set value "other_[$type s_uname]"
3     cpp_gen_assign_stmt $type $arg_ref $value $ind_lev 0
}
```

1. The `foreach` loop iterates over all the `in` and `inout` parameters.
2. The `cpp_clt_par_ref` command is used to obtain a reference to a parameter.
3. The parameter reference can then be used to initialize the parameter using the `cpp_gen_assign_stmt` command.

The previous Tcl script yields the following C++ code:

```
//--------
// Initialize "in" and "inout" parameters
//--------
p_widget = other_widget;
p_string = CORBA::string_dup(other_string);
```

See Also

`cpp_clt_par_decl`
`cpp_assign_stmt`
`cpp_gen_assign_stmt`
`cpp_l_name`

cpp_gen_array_decl_index_vars

`cpp_gen_array_decl_index_vars` *`array prefix ind_lev`*

Description

This command is a variant of “cpp_array_decl_index_vars” that prints its result directly to the current output.

cpp_gen_array_for_loop_footer

Synopsis `cpp_gen_array_for_loop_footer` *array ind_lev*

Description This command is a variant of "cpp_array_for_loop_footer" that prints its result directly to the current output.

cpp_gen_array_for_loop_header

Synopsis `cpp_gen_array_for_loop_header` *array prefix ind_lev ?declare?*

Description This command is a variant of "cpp_array_for_loop_header" that prints its result directly to the current output.

cpp_gen_assign_stmt

Synopsis `cpp_gen_assign_stmt` *type name value ind_lev ?scope?*

Description This command is a variant of "cpp_assign_stmt" that prints its result directly to the current output.

cpp_gen_attr_acc_sig_h

Synopsis `cpp_gen_attr_acc_sig_h` *attribute*

Description This command is a variant of "cpp_attr_acc_sig_h" that prints its result directly to the current output.

cpp_gen_attr_acc_sig_cc

Synopsis `cpp_gen_attr_acc_sig_cc` *attribute ?class?*

Description This command is a variant of "cpp_attr_acc_sig_cc" that prints its result directly to the current output.

cpp_gen_attr_mod_sig_h

Synopsis `cpp_gen_attr_mod_sig_h` *attribute*

Description This command is a variant of "cpp_attr_mod_sig_h" that prints its result directly to the current output.

cpp_gen_attr_mod_sig_cc

Synopsis `cpp_gen_attr_mod_sig_cc` *attribute ?class?*

Description This command is a variant of "cpp_attr_mod_sig_cc" that prints its result directly to the current output.

cpp_gen_clt_free_mem_stmt

Synopsis `cpp_gen_clt_free_mem_stmt` *name type direction is_var*

`cpp_gen_clt_free_mem_stmt` *arg is_var*
`cpp_gen_clt_free_mem_stmt` *op is_var*

Description This command is a variant of "cpp_clt_free_mem_stmt" that prints its result directly to the current output.

cpp_gen_clt_par_decl

Synopsis

```
cpp_gen_clt_par_decl name type direction is_var ind_lev
cpp_gen_clt_par_decl arg is_var ind_lev
cpp_gen_clt_par_decl op is_var ind_lev
```

Description

This command is a variant of "cpp_clt_par_decl" that prints its result directly to the current output.

cpp_gen_op_sig_h

Synopsis

```
cpp_gen_op_sig_h op
cpp_gen_op_sig_h initializer
```

Description

This command is a variant of "cpp_op_sig_h" that prints its result directly to the current output.

cpp_gen_op_sig_cc

Synopsis

```
cpp_gen_op_sig_cc op ?class?
cpp_gen_op_sig_cc initializer ?class?
```

Description

This command is a variant of "cpp_op_sig_cc" that prints its result directly to the current output.

cpp_gen_srv_free_mem_stmt

Synopsis

```
cpp_gen_srv_free_mem_stmt name type direction ind_lev
cpp_gen_srv_free_mem_stmt arg ind_lev
cpp_gen_srv_free_mem_stmt op ind_lev
```

Description

This command is a variant of "cpp_srv_free_mem_stmt" that prints its result directly to the current output.

cpp_gen_srv_par_alloc

Synopsis

```
cpp_gen_srv_par_alloc name type direction ind_lev
cpp_gen_srv_par_alloc arg ind_lev
cpp_gen_srv_par_alloc op ind_lev
```

Description

This command is a variant of "cpp_srv_par_alloc" that prints its result directly to the current output.

cpp_gen_srv_ret_decl

Synopsis

```
cpp_gen_srv_ret_decl name type ind_lev ?alloc_mem?
cpp_gen_srv_ret_decl op ind_lev ?alloc_mem?
```

Description

This command is a variant of "cpp_srv_ret_decl" that prints its result directly to the current output.

cpp_gen_var_decl

Synopsis

```
cpp_gen_var_decl name type is_var ind_lev
```

Description

This command is a variant of "cpp_var_decl" that prints its result directly to the current output.

cpp_gen_var_free_mem_stmt

Synopsis

`cpp_gen_var_free_mem_stmt` *`name type is_var`*

Description

This command is a variant of "cpp_var_free_mem_stmt" that prints its result directly to the current output.

cpp_impl_class

Synopsis

`cpp_impl_class` *`interface`*

`cpp_impl_class` *`valuetype`*

Description

This command returns the name of a C++ class that implements the specified IDL interface.

Parameters

interface	An interface node of the parse tree.
valuetype	A valuetype node of the parse tree

Notes

The class name is constructed by getting the fully scoped name of the IDL interface or valuetype, replacing all occurrences of `'::'` with `'_'` (the namespace is flattened) and appending `$pref(cpp,impl_class_suffix)`, which has the default value `Impl`.

Examples

Consider the following Tcl script:

```
# Tcl
set class [cpp_impl_class $inter]
[***
class @$class@ {
    public:
        @$class@();
};
***]
```

The following interface and valuetype definitions result in the generation of the corresponding C++ code:

Table 15: *C++ Implementation Classes*

<table>
<tr><th>Interface Definition</th><th>C++ Code</th></tr>
<tr><td><pre>//IDL
interface Cow {
    ...
};</pre></td><td><pre>// C++
class CowImpl {
    public:
            CowImpl();
};</pre></td></tr>
<tr><td><pre>//IDL
module Farm {
    interface Cow {
        ...
    };
};</pre></td><td><pre>// C++
class Farm_CowImpl {
    public:
            Farm_CowImpl();
};</pre></td></tr>
</table>

Table 15: *C++ Implementation Classes*

Interface Definition	C++ Code

`// IDL`
`valuetype Account {`
`    ...`
`};`
	`// C++`
	`class AccountImpl {`
	`    public:`
	`            AccountImpl();`
	`};`

cpp_indent

Synopsis

`cpp_indent` *`ind_lev`*

Description

This command returns the string given by `$pref(cpp,indent)`, concatenated with itself `$ind_lev` times. The default value of `$pref(cpp,indent)` is four spaces.

Parameters

`ind_lev` The number of levels of indentation required.

Examples

Consider the following Tcl script:

```
#Tcl
puts "[cpp_indent 1]One"
puts "[cpp_indent 2]Two"
puts "[cpp_indent 3]Three"
```

This produces the following output:

```
    One
        Two
            Three
```

cpp_is_fixed_size

Synopsis

`cpp_is_fixed_size` *`type`*

Description

This command returns TRUE if the node is a fixed-size node; otherwise it returns FALSE. It is an error if the node does not represent a type.

Parameters

`type` A type node of the parse tree.

Notes

The mapping of IDL to C++ has the concept of *fixed size* types and *variable size* types. This command returns a boolean value that indicates whether the specified `type` is fixed size.

The command is called internally from other commands in the `std/cpp_poa_lib.tcl` library.

See Also

`cpp_is_keyword`

`cpp_is_var_size`

cpp_is_keyword

Synopsis

`cpp_is_keyword` *`string`*

Description This command returns TRUE if the specified *string* is a C++ keyword; otherwise it returns FALSE.

Parameters

string	The string containing the identifier to be tested.

Notes This command is called internally from other commands in the std/cpp_poa_lib.tcl library.

Examples For example:

```
# Tcl
cpp_is_keyword "new"; # returns 1
cpp_is_keyword "cow"; # returns 0
```

See Also cpp_is_fixed_size

cpp_is_var_size

cpp_is_var_size

Synopsis cpp_is_var_size *type*

Description This command returns TRUE if the node is a variable-size node; otherwise it returns FALSE. It is an error if the node does not represent a type.

Parameters

type	A type node of the parse tree.

Notes The mapping of IDL to C++ has the concept of *fixed size* types and *variable size* types. This command returns a boolean value that indicates whether the specified *type* is variable size.

The command is called internally from other commands in the std/cpp_poa_lib.tcl library.

See Also cpp_is_fixed_size

cpp_is_keyword

cpp_l_name

Synopsis cpp_l_name *node*

Description This command returns the C++ mapping of the node's local name.

Parameters

node	A node of the parse tree.

Notes For user-defined types, the return value of cpp_l_name is usually the same as the node's local name, but prefixed with _cxx_ if the local name conflicts with a C++ keyword.

If the node represents a built-in IDL type, the result is the C++ mapping of the type; for example:

Table 16: *C++ Local Names for the built-in IDL Types*

IDL Type	C++ Type
short	CORBA::Short
unsigned short	CORBA::UShort
long	CORBA::Long
unsigned long	CORBA::ULong
char	CORBA::Char
octet	CORBA::Octet
boolean	CORBA::Boolean
string	char *
float	CORBA::Float
double	CORBA::Double
any	CORBA::Any
Object	CORBA::Object

When `cpp_l_name` is invoked on a parameter node, it returns the name of the parameter variable as it appears in IDL. You can use `cpp_l_name` in conjunction with `cpp_clt_par_decl` to help generate an operation invocation: the command `cpp_clt_par_decl` is used to declare the parameters, and `cpp_l_name` returns the name of the parameter in a form suitable for passing in the invocation.

See Also

`cpp_s_name`
`cpp_s_uname`
`cpp_clt_par_decl`
`cpp_gen_clt_par_decl`

cpp_nil_pointer

Synopsis

`cpp_nil_pointer` *`type`*

Description

This command returns a C++ expression that denotes a nil pointer value for the specified type.

Parameters

`type` A type node of the parse tree. The node must represent a type that can be heap-allocated.

Notes

The command returns a C++ expression that is a nil pointer (or a nil object reference) for the specified `type`. It should be used *only* for types that might be heap-allocated; that is, `struct`, `exception`, `union`, `sequence`, `array`, `string`, `Object`, `interface`, or `TypeCode`. If used for any other type, for example, a `long`, this command throws an exception.

This command can be used to initialize pointer variables. There is rarely a need to use this command if you make use of `_var` types in your applications.

cpp_obv_class_s_name

Synopsis

`cpp_obv_class_s_name` *`valuetype`*

Description

This command returns the fully scoped name of the OBV base class for the given *valuetype*.

Parameters

valuetype	A value node of the parse tree.

Examples

Given a *valuetype* called `Account`, the string returned by the `[cpp_obv_class_s_name` *valuetype*`]` expression is `OBV_Account`.

See Also

`cpp_poa_tie_s_name`

cpp_op_sig_h

Synopsis

`cpp_op_sig_h` *op*

`cpp_op_sig_h` *initializer*
`cpp_gen_op_sig_h` *op*
`cpp_gen_op_sig_h` *initializer*

Description

This command generates the signature of an operation or valuetype initializer for inclusion in `.h` files.

Parameters

op	An operation node of the parse tree.
initializer	An initializer node of the parse tree.

Notes

The command `cpp_op_sig_h` has no `';'` (semicolon) at the end of its generated statement.

The related command `cpp_gen_op_sig_h` does include a `';'` (semicolon) at the end of its generated statement.

Examples

Consider the following sample IDL:

```
// IDL
// File: 'finance.idl'
interface Account {
    attribute long accountNumber;
    attribute float balance;
    void makeDeposit(in float amount);
};
```

The following Tcl script illustrates the use of the command:

```
# Tcl
smart_source "std/sbs_output.tcl"
smart_source "std/cpp_poa_lib.tcl"

if { ! [idlgen_parse_idl_file "finance.idl"] } {
    exit 1
}

set op [$idlgen(root) lookup "Account::makeDeposit"]

set op_sig_h  [cpp_op_sig_h  $op]
output "$op_sig_h \n\n"

cpp_gen_op_sig_h  $op
```

The following output is generated by the Tcl script:

```
    virtual void makeDeposit(CORBA::Float amount)
    throw(CORBA::SystemException)

    virtual void
    makeDeposit(
        CORBA::Float                    amount
    ) throw(
        CORBA::SystemException
    );
```

See Also

cpp_gen_op_sig_h

cpp_op_sig_cc

cpp_op_sig_cc

Synopsis

cpp_op_sig_cc *op ?class?*

cpp_op_sig_cc *initializer ?class?*
cpp_gen_op_sig_cc *op ?class?*
cpp_gen_op_sig_cc *initializer ?class?*

Description

This command generates the signature of the operation or valuetype initializer for inclusion in .cxx files.

Parameters

op	An operation node of the parse tree.
initializer	An initializer node of the parse tree.
?class?	(Optional) The name of the class in which the method is defined. If no class is specified, the default implementation class name is used instead (given by [cpp_impl_class [$op defined_in]]).

Notes

Neither the cpp_op_sig_cc nor the cpp_gen_op_sig_cc command put a ';' (semicolon) at the end of the generated statement.

Examples

Consider the following sample IDL:

```
// IDL
// File: 'finance.idl'
interface Account {
    attribute long accountNumber;
    attribute float balance;
    void makeDeposit(in float amount);
};
```

The following Tcl script illustrates the use of the command:

```
# Tcl
smart_source "std/sbs_output.tcl"
smart_source "std/cpp_poa_lib.tcl"

if { ! [idlgen_parse_idl_file "finance.idl"] } {
    exit 1
}

set op [$idlgen(root) lookup "Account::makeDeposit"]

set op_sig_cc [cpp_op_sig_cc $op]
output "$op_sig_cc \n\n"

cpp_gen_op_sig_cc $op
```

The following output is generated by the Tcl script:

```
void
AccountImpl::makeDeposit(
    CORBA::Float                    amount
) throw(
    CORBA::SystemException
)

void
AccountImpl::makeDeposit(
    CORBA::Float                    amount
) throw(
    CORBA::SystemException
)
```

See Also

`cpp_op_sig_h`

`cpp_gen_op_sig_cc`

cpp_param_sig

Synopsis

`cpp_param_sig` *`name type direction`*

`cpp_param_sig` *`arg`*

Description

This command returns the C++ signature of the given parameter.

Parameters

name	The name of a parameter or return value variable.
type	A type node of the parse tree that describes the type of this parameter or return value.
direction	The parameter passing mode—one of `in`, `inout`, `out`, or `return`.
arg	An argument node of the parse tree.

Notes

This command is useful when you want to generate signatures for functions that use IDL data types. The following variants of the command are supported:

- The first form of the command returns the appropriate C++ type for the given *type* and *direction*, followed by the given *name*.
- The second form of the command returns output similar to the first but extracts the *type*, *direction* and *name* from the argument node *arg*.

The result contains white space padding, to vertically align parameter names when parameters are output one per line. The amount of padding is determined by `$pref(cpp,max_padding_for_types)`.

Examples

Consider the following Tcl extract:

```
# Tcl
...
set type [$idlgen(root) lookup "string"]
set dir "in"
puts "[cpp_param_sig "foo" $type $dir]"
```

The output generated by this script is:

```
const char *          foo
```

See Also

`cpp_param_type`

`cpp_gen_operation_h`
`cpp_gen_operation_cc`

cpp_param_type

Synopsis

`cpp_param_type` *type direction*

`cpp_param_type` *arg*
`cpp_param_type` *op*

Description

This command returns the C++ parameter type for the node specified in the first argument.

Parameters

type	A type node of the parse tree that describes the type of this parameter or return value.
direction	The parameter passing mode—one of `in`, `inout`, `out`, or `return`.
arg	An argument node of the parse tree.
op	An operation node of the parse tree.

Notes

This command is useful when you want to generate signatures for functions that use IDL data types. The following variants of the command are supported:

- The first form of the command returns the appropriate C++ type for the given *type* and *direction*.
- The second form of the command returns output similar to the first but extracts the *type* and *direction* from the argument node *arg*.
- The third form of this command is a shorthand for `[cpp_param_type [$op return_type] "return"]`. It returns the C++ type for the return value of the given *op*.

The result contains white space padding, to vertically align parameter names when parameters are output one per line. The amount of padding is determined by `$pref(cpp,max_padding_for_types)`.

Examples

The following Tcl extract prints out `const char *`:

```
# Tcl
...
set type [$idlgen(root) lookup "string"]
set dir "in"
puts "[cpp_param_type $type $dir]"
```

See Also

cpp_param_sig
cpp_gen_operation_h
cpp_gen_operation_cc

cpp_poa_class_s_name

Synopsis

cpp_poa_class_s_name *interface*

Description

This command returns the fully scoped name of the POA skeleton class for that interface.

Parameters

interface	An interface node of the parse tree.

Examples

Given an interface node `$inter`, the following Tcl extract shows how the command is used:

```
# Tcl
...
set class [cpp_impl_class $inter]
[***
class @$class@ :
    public virtual @[cpp_poa_class_s_name $inter]@
{
  public:
    @$class@();
};
***]
```

The following interface definitions results in the generation of the corresponding C++ code:.

Table 17: *C++ Implementation Classes*

IDL Definition	Generated C++

// IDL
interface Cow {
    ...
};
	// C++
	class CowImpl :
	    public virtual POA_Cow{
	public:
	    CowImpl();
	};

Table 17: *C++ Implementation Classes*

IDL Definition	Generated C++

`// IDL`
`module Farm {`
`    interface Cow{`
`    ...`
`    };`
`};`
	`// C++`
	`class Farm_CowImpl :`
	`    public virtual`
	`   POA_Farm::Cow`
	`{`
	`  public:`
	`    Farm_CowImpl();`
	`};`

See Also cpp_poa_tie_s_name

cpp_poa_tie_s_name

Synopsis cpp_poa_tie_s_name *interface*

Description This command returns the name of the POA tie template for the IDL interface.

Parameters

interface An interface node of the parse tree.

Examples Given an interface node `$inter`, the following Tcl extract shows how the command is used:

```
# Tcl
...
set class [cpp_impl_class $inter]
[***
    @$class@ * tied_object = new @$class@();
    @[cpp_poa_class_s_name $inter]@* the_tie =
        new @[cpp_poa_tie_s_name
   $inter]@<@$class@>(tied_object);
***]
```

If `$inter` is set to the node representing the IDL interface `Cow`, the Tcl code produces the following output:

```
    CowImpl * tied_object = new CowImpl();
    POA_Cow* the_tie =
        new POA_Cow_tie<CowImpl>(tied_object);
```

See Also cpp_poa_class_s_name

cpp_ret_assign

Synopsis cpp_ret_assign *op*

Description This command returns the `"_result ="` string (or a blank string, `""`, if *op* has a `void` return type).

Parameters

op An operation node of the parse tree.

Notes The name of the result variable is given by `$pref(cpp,ret_param_name)`. The default is `_result`.

See Also `cpp_assign_stmt`

`cpp_gen_assign_stmt`

cpp_s_name

Synopsis `cpp_s_name` *node*

Description This command returns the C++ mapping of the node's scoped name.

Parameters

node	A node of the parse tree.

Notes This command is similar to the `cpp_l_name` command, but it returns the fully scoped name of the C++ mapping type, rather than the local name.

Built-in IDL types are mapped as they are in the `cpp_l_name` command.

See Also `cpp_l_name`

`cpp_s_uname`

cpp_s_uname

Synopsis `cpp_s_uname` *node*

Description This command returns the node's scoped name, with each occurrence of the `::` separator replaced by an underscore `'_'` character.

Parameters

node	A node of the parse tree.

Notes The command is similar to `[$node s_uname]` except, for special-case handling of anonymous sequence and array types, to give them unique names.

Examples This routine is useful if you want to generate data types or operations for every IDL type. For example, the names of operations corresponding to each IDL type could be generated with the following statement:

```
set op_name "op_[cpp_s_uname $type]"
```

Some examples of IDL types and the corresponding identifier returned by cpp_s_uname:

Table 18: *Scoped Names with Underscore Scope Delimiter*

IDL Type	Scoped Name
foo m::foo m::for unsigned long sequence<foo>	foo m_foo m_for unsigned_long _foo_seq

See Also

cpp_l_name

cpp_s_name

cpp_sanity_check_idl

Synopsis

cpp_sanity_check_idl

Description

This command traverses the parse tree looking for unnecessary anonymous types that can cause portability problems in C++.

Notes

Consider the following IDL typedef:

```
typedef sequence< sequence<long> > longSeqSeq;
```

The mapping states that the IDL type longSeqSeq maps into a C++ class with the same name. However, the mapping does not state how the embedded anonymous sequence sequence<long> is mapped to C++. The net effect of loopholes like these in the mapping from IDL to C++ is that use of these anonymous types can hinder readability and portability of C++ code.

To avoid these problems, use extra typedef declarations in IDL files. For example, the previous IDL can be rewritten as follows:

```
typedef sequence<long> longSeq;
typedef sequence<longSeq>   longSeqSeq;
```

If cpp_sanity_check_idl finds anonymous types that might cause portability problems, it prints out a warning message.

Examples

The following Tcl script shows how the command is used:

```
# Tcl
...
smart_source "std/args.tcl"
smart_source "std/cpp_poa_lib.tcl"
parse_cmd_line_args file options
if {![idlgen_parse_idl_file $file $options]} {
    exit 1
}
cpp_sanity_check_idl
... # rest of script
```

cpp_srv_free_mem_stmt

Synopsis

```
cpp_srv_free_mem_stmt name type direction
cpp_srv_free_mem_stmt arg
cpp_srv_free_mem_stmt op
cpp_gen_srv_free_mem_stmt name type direction ind_lev
cpp_gen_srv_free_mem_stmt arg ind_lev
cpp_gen_srv_free_mem_stmt op ind_lev
```

Description

This command returns a C++ statement that frees the memory associated with the specified parameter (or return value) of an operation on the server side.

Parameters

name	The name of the parameter or return value variable.
type	A type node of the parse tree that describes the type of this parameter or return value.
direction	The parameter passing mode—one of `in`, `inout`, `out`, or `return`.
arg	An argument node of the parse tree.
op	An operation node of the parse tree.
ind_lev	Number of levels of indentation (`gen` variants only).

Notes

The following variants of the command are supported:

- The first form of the command is used to free memory associated with an explicitly named parameter variable.
- The second form of the command is used to free memory associated with parameters.
- The third form of the command is used to free memory associated with return values.
- The non-`gen` forms of the command omit the terminating `';'` (semicolon) character.
- The `gen` forms of the command include the terminating `';'` (semicolon) character.

There are only two cases in which a server should free the memory associated with a parameter:

- When assigning a new value to an `inout` parameter, it might be necessary to release the previous value of the parameter.
- If the body of the operation decides to throw an exception after memory has been allocated for `out` parameters and the return value, then the operation should free the memory of these parameters (and return value) and also assign nil pointers to these `out` parameters for which memory has previously been allocated. If the exception is thrown before memory has been allocated for the `out` parameters and the return value, then no memory management is necessary.

Examples

Given the following sample IDL:

```
// IDL
struct  widget           {long a;};
typedef sequence<long>   longSeq;
typedef long             long_array[10];

interface foo {
        longSeq op(
            in widget        p_widget,
            inout string     p_string,
            out longSeq      p_longSeq,
            out long_array   p_long_array);
};
```

If an operation throws an exception after it has allocated memory for `out` parameters and the return value, some memory management must be carried out before throwing the exception. These duties are shown in the following Tcl code:

Example 31: *Generating Code to Free Memory*

```
# Tcl
...
[***
        if (an_error_occurs) {
            //--------
            // Before throwing an exception, we must
            // free the memory of heap-allocated "out"
            // parameters and the return value,
            // and also assign nil pointers to these
            // "out" parameters.
            //--------
***]
foreach arg [$op args {out}] {
1       set free_mem_stmt [cpp_srv_free_mem_stmt $arg]
        if {$free_mem_stmt != ""} {
            set name [cpp_l_name $arg]
            set type [$arg type]
[***
            @$free_mem_stmt@;
2           @$name@ = @[cpp_nil_pointer $type]@;
***]
        }
}
3 cpp_gen_srv_free_mem_stmt $op 2
[***
            throw some_exception;
        }
***]
```

This script shows how `cpp_srv_free_mem_stmt` and `cpp_gen_srv_free_mem_stmt`, lines 1 and 3, respectively, can free memory associated with `out` parameters and the return value. Nil pointers can be assigned to `out` parameters by using the `cpp_nil_pointer` command , line 2.

The previous Tcl script generates the following C++ code:

```
// C++
if (an_error_occurs) {
    //--------
    // Before throwing an exception, we must
    // free the memory of heap-allocated "out"
    // parameters and the return value,
    // and also assign nil pointers to these
    // "out" parameters.
    //--------
    delete p_longSeq;
    p_longSeq = 0;
    delete _result;
    throw some_exception;
}
```

See Also

`cpp_gen_srv_free_mem_stmt`

`cpp_srv_need_to_free_mem`

cpp_srv_need_to_free_mem

Synopsis

`cpp_srv_need_to_free_mem` *`type direction`*

`cpp_srv_need_to_free_mem` *`arg`*
`cpp_srv_need_to_free_mem` *`op`*

Description

This command returns `1` (TRUE) if the server program has to take explicit steps to free memory when the operation is being aborted, by throwing an exception. It returns `0` (FALSE) otherwise.

Parameters

type	A type node of the parse tree that describes the type of this parameter or return value.
direction	The parameter passing mode—one of `in`, `inout`, `out`, or `return`.
arg	An argument node of the parse tree.
op	An operation node of the parse tree.

Notes

The following variants of the command are supported:

- The first form of the command is used to check whether the given *type* of parameter (or return value), passed in the given *direction*, must be explicitly freed when an exception is thrown.
- The second form of the command is used to check parameters.
- The third form of the command is used to check return values.

See Also

`cpp_srv_free_mem_stmt`

cpp_srv_par_alloc

Synopsis

`cpp_srv_par_alloc` *`name type direction`*

`cpp_srv_par_alloc` *`arg`*
`cpp_srv_par_alloc` *`op`*
`cpp_gen_srv_par_alloc` *`name type direction ind_lev`*
`cpp_gen_srv_par_alloc` *`arg ind_lev`*

```
cpp_gen_srv_par_alloc op ind_lev
```

Description

This command returns a C++ statement to allocate memory for an `out` parameter (or return value), if needed. If there is no need to allocate memory, this command returns an empty string.

Parameters

type	A type node of the parse tree that describes the type of this parameter or return value.
direction	The parameter passing mode—one of `in`, `inout`, `out`, or `return`.
arg	An argument node of the parse tree.
op	An operation node of the parse tree.
ind_lev	The number of levels of indentation (`gen` variants only).

Examples

Given the following sample IDL:

```
// IDL
struct  widget          {long a;};
typedef sequence<long>   longSeq;
typedef long             long_array[10];

interface foo {
        longSeq op(
            in widget       p_widget,
            inout string    p_string,
            out longSeq     p_longSeq,
            out long_array  p_long_array);
};
```

The following Tcl script declares a local variable that can hold the return value of the operation. It then allocates memory for `out` parameters and the return value, if required.

Example 32: *Allocating Memory for Parameters*

```
# Tcl
...
set op         [$idlgen(root) lookup "foo::op"]
set ret_type   [$op return_type]
set is_var     0
set ind_lev    1
set arg_list   [$op contents {argument}]
if {[$ret_type l_name] != "void"} {
[***
       //--------
       // Declare a variable to hold the return value.
       //--------
```

Example 32: *Allocating Memory for Parameters*

```
1       @[cpp_srv_ret_decl $op 0]@;

***]
}
[***
        //--------
        // Allocate memory for "out" parameters
        // and the return value, if needed.
        //--------
***]
foreach arg [$op args {out}] {
        cpp_gen_srv_par_alloc $arg $ind_lev
}
2 cpp_gen_srv_par_alloc $op $ind_lev
```

The previous Tcl script generates the following C++ code:

```
// C++
//--------
// Declare a variable to hold the return value.
//--------
longSeq* _result;

//--------
// Allocate memory for "out" parameters
// and the return value, if needed.
//--------
p_longSeq = new longSeq;
_result = new longSeq;
```

The declaration of the `_result` variable, line 1, is separated from allocation of memory for it, line 2. This gives you the opportunity to throw exceptions before allocating memory, which eliminates the memory management responsibilities associated with throwing an exception. If you prefer to allocate memory for the `_result` variable in its declaration, change line `1` in the Tcl script so that it passes `1` as the value of the `alloc_mem` parameter, then delete line `2` of the Tcl script. If you make these changes, the declaration of `_result` changes as follows:

```
longSeq* _result = new longSeq;
```

See Also

`cpp_gen_srv_par_alloc`
`cpp_srv_par_ref`
`cpp_srv_ret_decl`

cpp_srv_par_ref

Synopsis

`cpp_srv_par_ref` *`name type direction`*
`cpp_srv_par_ref` *`arg`*
`cpp_srv_par_ref` *`op`*

Description

This command returns a reference to the value of the specified parameter (or return value) of an operation. The returned reference is either `$name` or `*$name`, depending on whether the parameter is passed by reference or by pointer.

Parameters

name	The name of a parameter or return value variable.
type	A type node of the parse tree that describes the type of this parameter or return value.
direction	The parameter passing mode—one of `in`, `inout`, `out`, or `return`.
arg	An argument node of the parse tree.
op	An operation node of the parse tree.

Notes

References returned by `cpp_clt_par_ref` are intended for use in the context of assignment statements, in conjunction with the `cpp_gen_assign_stmt` command. See the following example.

Examples

Given the following sample IDL:

```
// IDL
struct  widget           {long a;};
typedef sequence<long>   longSeq;
typedef long             long_array[10];

interface foo {
        longSeq op(
            in widget       p_widget,
            inout string    p_string,
            out longSeq     p_longSeq,
            out long_array  p_long_array);
};
```

The following Tcl script iterates over all `inout` and `out` parameters and the return value, and assigns values to them:

Example 33: *Assigning Values to Parameters and Return Value*

```
# Tcl
[***
       //--------
       // Assign new values to "out" and "inout"
       // parameters, and the return value, if needed.
       //--------
***]
foreach arg [$op args {inout out}] {
        set type    [$arg type]
1       set arg_ref [cpp_srv_par_ref $arg]
        set name2   "other_[$type s_uname]"
        if {[$arg direction] == "inout"} {
2           cpp_gen_srv_free_mem_stmt $arg $ind_lev
        }
3       cpp_gen_assign_stmt $type $arg_ref $name2 \
                                    $ind_lev 0
}
if {[$ret_type l_name] != "void"} {
4       set ret_ref [cpp_srv_par_ref $op]
        set name2   "other_[$ret_type s_uname]"
5       cpp_gen_assign_stmt $ret_type $ret_ref \
                            $name2 $ind_lev 0
}
```

The cpp_srv_par_ref command, lines 1 and 4, can be used to obtain a reference to both the parameters and the return value. For example, in the IDL operation used in this example, the parameter p_longSeq is passed by pointer. Thus, a reference to this parameter is *p_longSeq. A reference to a parameter (or the return value) can then be used to initialize it using the cpp_gen_assign_stmt command, lines 3 and 5.

It is sometimes necessary to free the old value associated with an inout parameter before assigning it a new value. This can be achieved using the cpp_gen_srv_free_mem_stmt command, line 2. However, this should be done only for inout parameters; hence the if statement around this command.

The previous Tcl script generates the following C++ code:

```
// C++
//--------
// Assign new values to "out" and "inout"
// parameters, and the return value, if needed.
//--------
CORBA::string_free(p_string);
p_string = CORBA::string_dup(other_string);
*p_longSeq = other_longSeq;
for (CORBA::ULong i1 = 0; i1 < 10; i1 ++) {
    p_long_array[i1] = other_long_array[i1];
}
*_result = other_longSeq;
```

See Also

cpp_srv_par_alloc

cpp_srv_ret_decl

cpp_srv_ret_decl

Synopsis

cpp_srv_ret_decl *name type ?alloc_mem?*

cpp_srv_ret_decl *op ?alloc_mem?*
cpp_gen_srv_ret_decl *name type ind_lev ?alloc_mem?*
cpp_gen_srv_ret_decl *op ind_lev ?alloc_mem?*

Description

This command returns a C++ declaration of a variable that holds the return value of an operation. If the operation does not have a return value this command returns an empty string.

Parameters

name	The name of the parameter or return value variable.
type	A type node of the parse tree that describes the type of this parameter or return value.
alloc_mem	(Optional) The flag indicating whether memory should be allocated. Default value is 1, meaning allocate.
op	An operation node of the parse tree.
ind_lev	The number of levels of indentation (gen variants only).

Notes

Assuming that the operation does have a return value, if `alloc_mem` is `1`, the variable declaration also allocates memory to hold the return value, if necessary. If `alloc_mem` is `0`, no allocation of memory occurs, and instead you can allocate the memory later with the `cpp_srv_par_alloc` command. The default value of `alloc_mem` is `1`.

Examples

Given the following sample IDL:

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

interface foo {
        longSeq op();
};
```

The following Tcl script declares a local variable that can hold the return value of the operation. It then allocates memory for the return value, if required.

Example 34: *Declare a Variable to Hold a Return Value*

```
# Tcl
...
set op          [$idlgen(root) lookup "foo::op"]
set ret_type    [$op return_type]
set ind_lev     1
if {[$ret_type l_name] != "void"} {
[***
       //--------
       // Declare a variable to hold the return value.
       //--------
1    @[cpp_srv_ret_decl $op 0]@;

***]
}

2  cpp_gen_srv_par_alloc $op $ind_lev
```

The previous Tcl script generates the following C++ code:

```
// C++
//--------
// Declare a variable to hold the return value.
//--------
longSeq* _result;

_result = new longSeq;
```

The declaration of the `_result` variable, line 1, is separated from the allocation of memory for it, line 2. This gives you the opportunity to throw exceptions before allocating memory, which eliminates the memory management responsibilities associated with throwing an exception. If you prefer to allocate memory for the `_result` variable in its declaration, change line 1 in the Tcl script so that it passes `1` as the value of the `alloc_mem` parameter, then delete line 2 of the Tcl script. If you make these changes, the declaration of `_result` changes as follows:

```
longSeq* _result = new longSeq;
```

See Also

`cpp_srv_par_alloc`
`cpp_srv_par_ref`
`cpp_gen_srv_ret_decl`

cpp_typecode_l_name

Synopsis

`cpp_typecode_l_name` *`type`*

Description

This command returns the local C++ name of the `typecode` for the specified `type`.

Parameters

`type`	A type node of the parse tree.

Notes

For user-defined types, the command forms the type code by prefixing the local name of the type with `_tc_`. For the built-in types (such as `long`, and `short`), the type codes are defined inside the CORBA module.

Examples

Examples of the local names of C++ type codes for IDL types:

Table 19: *Generating C++ Type Code Identifiers*

IDL Type	C++ Type Code
`cow`	`_tc_cow`
`farm::cow`	`_tc_cow`
`long`	`CORBA::_tc_long`

See Also

`cpp_typecode_s_name`

cpp_typecode_s_name

Synopsis

`cpp_typecode_s_name` *`type`*

Description

This command returns the fully-scoped C++ name of the `typecode` for the specified `type`.

Parameters

`type`	A type node of the parse tree.

Notes

For user-defined types, an IDL type of the form *`scope`*`::`*`localName`* has the scoped type code *`scope`*`::_tc_`*`localName`*. For the built-in types (such as `long` and `short`), the type codes are defined inside the CORBA module.

Examples

Examples of the fully-scoped names of C++ type codes for IDL types:

Table 20: *Generating C++ Scoped Type Code Identifiers*

IDL Type	C++ Type Code
`cow`	`_tc_cow`
`farm::cow`	`farm::_tc_cow`
`long`	`CORBA::_tc_long`

See Also cpp_typecode_l_name

cpp_value_factory_base_class

Synopsis

cpp_value_factory_base_class *valuetype*

cpp_value_factory_base_class *valuebox*

Description

This command returns the scoped name of a generated factory base class (for a regular valuetype) or a valuebox factory class (for a boxed valuetype).

Parameters

valuetype	A value node of the parse tree
valuebox	A value_box node of the parse tree.

Examples

Given the following IDL:

```
//IDL
module Finance {
    valuetype Account {
        factory init(in float openingBalance);

        public float balance;
        private string password;

        boolean make_deposit(in float amount);
        boolean make_withdrawal(in float amount);
    };
};
```

Consider the following Tcl script, which assumes that the `$value` variable is set equal to the `Account` value node:

```
#Tcl
...
    set factory_class [cpp_value_factory_impl_class
   $value]
[***
//  @$factory_class@ --
//  Factory to create instances of valuetype @[cpp_s_name
   $value]@
//
class @$factory_class@ :
    public virtual @[cpp_value_factory_base_class $value]@
{
public:
    // _register_with_orb - create and register a factory.
    //
    static void
    _register_with_orb(
        CORBA::ORB_ptr orb
    );
```

```
    virtual CORBA::ValueBase*
    create_for_unmarshal();

***]
    foreach op [$value contents initializer] {
        cpp_gen_op_sig_h $op
        output "\n"
    }

[***
};
***]
```

The previous Tcl script generates the following C++ code:

```
// C++
//  Finance_AccountFactory --
//  Factory to create instances of valuetype
   Finance::Account
//
class Finance_AccountFactory :
    public virtual Finance::Account_init
{
public:
    // _register_with_orb - create and register a factory.
    //
    static void
    _register_with_orb(
        CORBA::ORB_ptr orb
    );

    virtual CORBA::ValueBase*
    create_for_unmarshal();

    virtual Finance::Account*
    init(
        CORBA::Float                    openingBalance
    );

};
```

cpp_value_factory_impl_class

Synopsis

`cpp_value_factory_base_class` *`valuetype`*

Description

This command returns the name of the C++ class that implements the *`valuetype`* factory.

Parameters

`valuetype`	A value node of the parse tree

Notes

The class name is constructed by getting the fully scoped name of the IDL interface or valuetype, replacing all occurrences of `'::'` with `'_'` (the namespace is flattened) and appending `$pref(cpp,factory_suffix)`, which has the default value `Factory`.

cpp_var_decl

Synopsis

`cpp_var_decl` *`name type is_var`*

cpp_gen_var_decl *name type is_var ind_lev*

Description

This command returns a C++ variable declaration with the specified *name* and *type*.

Parameters

name	The name of the variable.
type	A type node of the parse tree that describes the type of this variable.
is_var	The boolean flag indicates whether the variable is a _var type. A value of 1 indicates a _var type.
ind_lev	The number of levels of indentation (gen variants only).

Notes

For most variables, the *is_var* parameter is ignored, and the variable is allocated on the stack. However, if the variable is a string or an object reference, it must be heap allocated, and the *is_var* parameter determines whether the variable is declared as a _var (smart pointer) type or as a raw pointer.

All variables declared via cpp_var_decl are references, and hence can be used directly with cpp_assign_stmt.

Examples

The following Tcl script illustrates how to use this command:

```
# Tcl
...
set is_var 0
set ind_lev 1
[***
        // Declare variables
***]
foreach type $type_list {
        set name  "my_[$type l_name]"
        cpp_gen_var_decl $name $type $is_var $ind_lev
}
```

If variable type_list contains the types string, widget (a struct), and long_array, the Tcl code generates the following C++ code:

```
// C++
        // Declare variables
        char *                      my_string;
        widget                      my_widget;
        long_array                  my_long_array;
```

See Also

cpp_gen_var_decl

cpp_var_free_mem_stmt
cpp_var_need_to_free_mem

cpp_var_free_mem_stmt

Synopsis

cpp_var_free_mem_stmt *name type is_var*

cpp_gen_var_free_mem_stmt *name type is_var*

Description

This command returns a C++ statement that frees the memory associated with the variable of the specified name and type. If there is no need to free memory for the variable, the command returns an empty string.

Parameters

name	The name of the variable.
type	A type node of the parse tree that describes the type of this variable.
is_var	A boolean flag to indicate whether the variable is a `_var` type or not. A value of `1` indicates a `_var` type.

Examples

The following Tcl script illustrates how to use the command:

```
# Tcl
set is_var 0
set ind_lev 1
[***
       // Memory management
***]
foreach type $type_list {
        set name  "my_[$type l_name]"
        cpp_gen_var_free_mem_stmt $name $type $is_var
   $ind_lev
}
```

If variable `type_list` contains the types `string`, `widget` (a struct) and `long_array`, the Tcl script generates the following C++ code:

```
// C++
        // Memory management
        CORBA::string_free(my_string);
```

The `cpp_gen_var_free_mem_stmt` command generates memory-freeing statements only for the `my_string` variable. The other variables are stack-allocated, so they do not require their memory to be freed. If you modify the Tcl code so that `is_var` is set to TRUE, `my_string`'s type changes from `char *` to `CORBA::String_var` and the memory-freeing statement for that variable is suppressed.

See Also

`cpp_var_decl`
`cpp_gen_var_free_mem_stmt`
`cpp_var_need_to_free_mem`

cpp_var_need_to_free_mem

Synopsis

`cpp_var_need_to_free_mem` *type is_var*

Description

This command returns `1` (`TRUE`) if the programmer has to take explicit steps to free memory for a variable of the specified type; otherwise it returns `0` (`FALSE`).

Parameters

type	A type node of the parse tree that describes the type of this variable.

is_var	A boolean flag that indicates whether the variable is a _var type or not. A value of 1 indicates a _var type.

See Also

cpp_var_decl

cpp_var_free_mem_stmt

C++ Utility Libraries

This reference describes two libraries—the cpp_poa_print and cpp_poa_random utility libraries—that can be used in your own Tcl scripts to generate print statements or to initialize variables with random data.

cpp_poa_print Commands

This section gives detailed descriptions of the Tcl commands in the `cpp_poa_print` library.

cpp_gen_print_stmt

Synopsis

`cpp_gen_print_stmt` *`type name ?indent? ?ostream?`*

Description

This command is a variant of "cpp_print_stmt" that prints its result directly to the current output.

cpp_print_delete

Synopsis

`cpp_print_delete` *`?printer?`*

Description

This command generates a statement to deallocate the printer object (of `IT_GeniePrint` type). No terminating '`;`' (semicolon) is generated.

Parameters

printer	(Optional) The name of the printer object pointer. Default is `global_print`.

Notes

There is no complementary command to declare the printer object pointer. A printer object pointer can be declared using the following line of C++ code:

```
//C++
IT_GeniePrint* global_print = 0;
```

cpp_print_func_name

Synopsis

`cpp_print_func_name` *`type`*

Description

This command returns the name of the function that prints the given *type*.

Parameters

type	A type node of the parse tree.

Notes

The printer member function is invoked on `global_print` by default.

cpp_print_gen_init

Synopsis

`cpp_print_gen_init` *`?orb?`*

Description

This command generates a statement to initialize the `global_print` pointer.

Parameters

orb	(Optional) The name of a pointer to an ORB object (of `CORBA::ORB_ptr` type) passed to the `IT_GeniePrint` constructor. Default is `global_orb`.

Notes This command must be called before generating any print statements.

cpp_print_stmt

Synopsis `cpp_print_stmt` *type name ?indent? ?ostream?*

`cpp_gen_print_stmt` *type name ?indent? ?ostream?*

Description This command generates a statement that prints the *name* variable, which is of *type* type, to the *ostream* output stream with *indent* levels of indentation.

Parameters

type	A type node of the parse tree.
name	The name of the variable to be printed (must be a variable reference).
indent	(Optional) The number of units of indentation. Default is `0`.
ostream	(Optional) The name of an output stream. Default is the `cout` standard output stream.

Notes No terminating '`;`' (semicolon) is generated in the `cpp_print_stmt` version of the command.

A terminating '`;`' (semicolon) is generated in the `cpp_gen_print_stmt` version of the command.

gen_cpp_print_funcs_cc

Synopsis `gen_cpp_print_funcs_cc ?`*ignored?*

Description This command generates an `it_print_funcs.cxx` file containing the implementation of the `IT_GeniePrint` class.

Parameters

ignored	(Optional) Retained for backwards compatibility.

gen_cpp_print_funcs_h

Synopsis `gen_cpp_print_funcs_h`

Description This command generates an `it_print_funcs.h` file containing the declaration of the `IT_GeniePrint` class.

cpp_poa_random Commands

This section gives detailed descriptions of the Tcl commands in the `cpp_poa_random` library.

cpp_gen_random_assign_stmt

Synopsis

`cpp_gen_random_assign_stmt` *`type name indent`*

Description

This command is a variant of "cpp_random_assign_stmt" that prints its result directly to the current output.

cpp_random_assign_stmt

Synopsis

`cpp_random_assign_stmt` *`type name`*

`cpp_gen_random_assign_stmt` *`type name indent`*

Description

This command generates a statement that assigns a random value to the *name* variable, which is of *type* type.

Parameters

type	A type node of the parse tree.
name	The name of the variable to which a random value is assigned.
indent	The number of units of indentation.

Notes

No terminating ';' (semicolon) is generated in the `cpp_random_assign_stmt` version of the command.

A terminating ';' (semicolon) is generated in the `cpp_gen_random_assign_stmt` version of the command.

cpp_random_delete

Synopsis

`cpp_random_delete ?`*`random?`*

Description

This command generates a statement to deallocate the random object (of `IT_GenieRandom` type). No terminating ';' (semicolon) is generated.

Parameters

random	(Optional) The name of the random object pointer. Default is `global_random`.

Notes

There is no complementary command to declare the random object pointer. A random object pointer can be declared using the following line of C++ code:

```
//C++
IT_GenieRandom* global_random = 0;
```

cpp_random_gen_init

Synopsis

`cpp_random_gen_init` *`?orb? ?seed? ?random?`*

Description

This command generates a statement to initialize the `global_random` pointer.

Parameters

orb	(Optional) The name of a pointer to an ORB object (of `CORBA::ORB_ptr` type) passed to the `IT_GenieRandom` constructor. Default is `global_orb`.
seed	(Optional) An integer seed to initialize the random number generator. Default is `0`.
random	(Optional) The name of the random object pointer. Default is `global_random`.

Notes A terminating '`;`' (semicolon) is generated.

gen_cpp_random_funcs_cc

Synopsis `gen_cpp_random_funcs_cc ?`*`ignored?`*

Description This command generates an `it_random_funcs.cxx` file containing the implementation of the `IT_GenieRandom` class.

Parameters

ignored	(Optional) Retained for backwards compatibility.

gen_cpp_random_funcs_h

Synopsis `gen_cpp_random_funcs_h`

Description This command generates an `it_random_funcs.h` file containing the declaration of the `IT_GenieRandom` class.

Part IV

Java Genies Library Reference

In this part

This part contains the following chapters:

Java Development Library

The code generation toolkit comes with a rich Java development library that makes it easy to create code generation applications that map IDL to Java code.

Naming Conventions in API Commands

java_poa_lib.tcl library commands

The abbreviations shown in Table 21 are used in the names of commands defined in the `std/java_poa_lib.tcl` library.

Table 21: *Abbreviations Used in Command Names.*

Abbreviation	Meaning
`clt`	Client
`srv`	Server
`var`	Variable
`var_decl`	Variable declaration
`gen_`	See “Naming Conventions for gen_”
`par/param`	Parameter
`ref`	Reference
`stmt`	Statement
`op`	Operation
`attr_acc`	An IDL attribute's accessor
`attr_mod`	An IDL attribute's modifier
`sig`	Signature

Command names in `std/java_poa_lib.tcl` start with the `java_` prefix.

For example, the following statement generates the Java signature of an operation:

```
[java_op_sig $op]
```

Naming Conventions for gen_

The names of some commands contain gen_, to indicate that they generate output into the current output file. For example, java_gen_op_sig outputs the Java signature of an operation. Commands whose names omit gen_ return a value—which you can use as a parameter to the output command.

Examples

Some commands whose names do not contain gen_ also have gen_ counterparts. Both forms are provided to offer greater flexibility in how you write scripts. In particular, commands without gen_ are easy to embed inside textual blocks (that is, text inside [*** and ***]), while their gen_ counterparts are sometimes easier to call from outside textual blocks. Some examples follow:

- The following segment of code prints the Java signatures of all the operations of an interface:

  ```
  # Tcl
  foreach op [$inter contents {operation}] {
      output "    [java_op_sig $op]\n"
  }
  ```

  The output statement uses spaces to indent the signature of the operation, and follows it with a newline character. The printing of this white space is automated by the gen_ counterpart of this command. The above code snippet could be rewritten in the following, slightly more concise, format:

  ```
  # Tcl
  foreach op [$inter contents {operation}] {
      java_gen_op_sig $op
  }
  ```

- The use of commands without gen_ can often eliminate the need to toggle in and out of textual blocks. For example:

  ```
  # Tcl
  [***
  //--------
  // Function: ...
  //--------
  @[java_op_sig $op]@
  {
      ... // body of the operation
  }
  ***]
  ```

Indentation

To allow programmers to choose their preferred indentation, all commands in std/java_poa_lib.tcl use the string in $pref(java,indent) for each level of indentation they need to generate.

Some commands take a parameter called `ind_lev`. This parameter is an integer that specifies the indentation level at which output should be generated.

$pref(java,...) Entries

Some entries in the `$pref(java,…)` array are used to specify various user preferences for the generation of Java code, as shown in Table 22. All of these entries have default values if there is no setting in the `idlgen.cfg` file. You can also force the setting by explicit assignment in a Tcl script.

Table 22: *$pref(java,...) Array Entries*

$pref(...) Array Entry	Purpose
`$pref(java,java_file_ext)`	Specifies the filename extension for Java source code files. Its default value is `.java`.
`$pref(java,java_class_ext)`	Specifies the filename extension for Java class files. Its default value is `.class`.
`$pref(java,indent)`	Specifies the amount of white space to be used for one level of indentation. Its default value is four spaces.
`$pref(java,impl_class_suffix)`	Specifies the suffix that is added to the name of a class that implements an IDL interface. Its default value is `Impl`.
`$pref(java,attr_mod_param_name)`	Specifies the name of the parameter in the Java signature of an attribute's modifier operation. Its default value is `_new_value`.
`$pref(java,ret_param_name)`	Specifies the name of the variable that is to be used to hold the return value from a non-`void` operation call. Its default value is `_result`.
`$pref(java,max_padding_for_types)`	Specifies the padding to be used with Java type names when declaring variables or parameters. This padding helps to ensure that the names of variables and parameters are vertically aligned, which makes code easier to read. Its default value is `32`.

Groups of Related Commands

To help you find the commands needed for a particular task, each heading below lists a group of related commands.

Identifiers and Keywords

`java_l_name` *node*
`java_s_name` *node*
`java_typecode_s_name` *type*
`java_typecode_l_name` *type*

General Purpose Commands

`java_assign_stmt` *type name value ind_lev ?scope?*
`java_indent` *number*
`java_is_keyword` *name*

Servant/Implementation Classes

`java_impl_class` *interface_node*
`java_poa_class_s_name` *interface_node*
`java_poa_tie_s_name` *interface_node*

Operation Signatures

`java_gen_op_sig` *operation_node ?class_name?*
`java_op_sig` *operation_node ?class_name?*

Attribute Signatures

`java_attr_acc_sig` *attribute_node ?class_name?*
`java_attr_mod_sig` *attribute_node ?class_name?*
`java_gen_attr_acc_sig` *attribute_node ?class_name?*
`java_gen_attr_mod_sig` *attribute_node ?class_name?*

Types and Signatures of Parameters

`java_param_sig` *name type direction*
`java_param_sig` *op_or_arg*
`java_param_type` *type direction*
`java_param_type` *op_or_arg*

Invoking Operations

`java_assign_stmt` *type name value ind_lev ?scope?*
`java_clt_par_decl` *arg_or_op is_var*
`java_clt_par_ref` *arg_or_op is_var*
`java_gen_clt_par_decl` *arg_or_op is_var ind_lev*
`java_ret_assign` *op*

Invoking Attributes

`java_clt_par_decl` *name type dir is_var*
`java_clt_par_ref` *name type dir is_var*
`java_gen_clt_par_decl` *name type dir is_var ind_lev*

Implementing Operations

`java_gen_srv_par_alloc` *arg_or_op ind_lev*
`java_gen_srv_ret_decl` *op ind_lev ?alloc_mem?*
`java_srv_par_alloc` *arg_or_op*

java_srv_par_ref *arg_or_op*
java_srv_ret_decl *op ?alloc_mem?*

Implementing Attributes

java_gen_srv_par_alloc *name type direction ind_lev*
java_gen_srv_ret_decl *name type ind_lev ?alloc_mem?*
java_srv_par_alloc *name type direction*
java_srv_par_ref *name type direction*
java_srv_ret_decl *name type ?alloc_mem?*

Instance Variables and Local Variables

java_var_decl *name type is_var*

Processing Unions

java_branch_case_l_label *union_branch*
java_branch_case_s_label *union_branch*
java_branch_l_label *union_branch*
java_branch_s_label *union_branch*

Processing Arrays

java_array_decl_index_vars *arr pre ind_lev*
java_array_elem_index *arr pre*
java_array_for_loop_footer *arr indent*
java_array_for_loop_header *arr pre ind_lev ?decl?*
java_gen_array_decl_index_vars *arr pre ind_lev*
java_gen_array_for_loop_footer *arr indent*
java_gen_array_for_loop_header *arr pre ind_lev ?decl?*

Processing Any

java_any_extract_stmt *type any_name name*
java_any_extract_var_decl *type name*
java_any_extract_var_ref *type name*
java_any_insert_stmt *type any_name value ?is_var?*

java_poa_lib Commands

This section gives detailed descriptions of the Tcl commands in the java_poa_lib library.

java_any_extract_stmt

Synopsis

java_any_extract_stmt *type any_name var_name*

Description

This command generates a statement that extracts the value of the specified *type* from the any called *any_name* into the *var_name* variable.

Parameters

type	A type node of the parse tree.
any_name	The name of the `any` variable.
var_name	The name of the variable into which the `any` is extracted.

Notes

var_name must be a variable declared by `java_any_extract_var_decl`.

Examples

The following Tcl script illustrates the use of the `java_any_extract_stmt` command:

```
# Tcl
smart_source "std/output.tcl"
smart_source "std/java_poa_lib.tcl"

if { ! [idlgen_parse_idl_file "prototype.idl"] } {
    exit 1
}

idlgen_set_preferences $idlgen(cfg)
open_output_file "any_extract.java"

lappend type_list [$idlgen(root) lookup widget]
lappend type_list [$idlgen(root) lookup boolean]
lappend type_list [$idlgen(root) lookup long_array]

[***
try {
***]
```

```
foreach type $type_list {
    set var_name my_[$type s_uname]
[***
    @[java_any_extract_var_decl $type $var_name]@;
***]
}
output "\n"
foreach type $type_list {
    set var_name my_[$type s_uname]
    set var_ref  [java_any_extract_var_ref $type $var_name]
[***
    @[java_any_extract_stmt $type "an_any" $var_name]@
    process_@[$type s_uname]@(@$var_ref@);

***]
}
[***
};
catch(Exception e){
    System.out.println("Error: extract from any.");
    e.printStackTrace();
};
***]
close_output_file
If the type_list variable contains the type nodes for widget (a
   struct), boolean and long_array, the previous Tcl script
   generates the following Java code::
// Java
try {
    NoPackage.widget                my_widget;
    boolean                         my_boolean;
    int[]                           my_long_array;

    my_widget = NoPackage.widgetHelper.extract(an_any)
    process_widget(my_widget);

    my_boolean = an_any.extract_boolean()
    process_boolean(my_boolean);

    my_long_array = NoPackage.long_arrayHelper.extract(an_any)
    process_long_array(my_long_array);
};

catch(Exception e){
    System.out.println("Error: extract from any.");
    e.printStackTrace();
};
java_any_insert_stmt
java_any_extract_var_decl
java_any_extract_var_ref
```

java_any_extract_var_decl

Synopsis

`java_any_extract_var_decl` *type name*

Description

This command declares a variable, into which values from an `any` are extracted. The parameters to this command are the variable's *type* and *name*.

Parameters

type	A type node of the parse tree.
name	The name of the variable.

Examples

The following Tcl script illustrates the use of the `java_any_extract_var_decl` command:

```
# Tcl
foreach type $type_list {
    set var_name my_[$type s_uname]
[***
   @[java_any_extract_var_decl $type $var_name]@;
***]
}
```

If the variable `type_list` contains the type nodes for `widget` (a `struct`), `boolean`, and `long_array`, then the previous Tcl script generates the following Java code::

```
//Java
    NoPackage.widget                my_widget;
    boolean                         my_boolean;
    int[]                           my_long_array;
```

See Also

`java_any_insert_stmt`

`java_any_extract_var_ref`
`java_any_extract_stmt`

java_any_extract_var_ref

Synopsis

`java_any_extract_var_ref` *type name*

Description

This command returns a reference to the value in *name* of the specified *type*.

Parameters

type	A type node of the parse tree.
name	The name of the variable.

Notes

The returned reference is always `$name`.

Examples

The following Tcl script illustrates the use of the `java_any_extract_var_ref` command:

```
# Tcl
foreach type $type_list {
    set var_name my_[$type s_uname]
    set var_ref  [java_any_extract_var_ref $type
   $var_name]
[***
    process_@[$type s_uname]@(@$var_ref@);
***]
}
```

If the variable `type_list` contains the type nodes for `widget` (a struct), `boolean`, and `long_array` then the previous Tcl script generates the following Java code:

```
// Java
    process_widget(my_widget);
    process_boolean(my_boolean);
    process_long_array(my_long_array);
```

See Also

`java_any_insert_stmt`
`java_any_extract_var_decl`
`java_any_extract_stmt`

java_any_insert_stmt

Synopsis

`java_any_insert_stmt` *`type any_name value`*

Description

This command returns the Java statement that inserts the specified *value* of the specified *type* into the `any` called *any_name*.

Parameters

type	A type node of the parse tree.
any_name	The name of the `any` variable.
value	The name of the variable that is being inserted into the `any`.

Examples

The following Tcl script illustrates the use of the `java_any_insert_stmt` command:

```
# Tcl
smart_source "std/output.tcl"
smart_source "std/java_poa_lib.tcl"
if { ! [idlgen_parse_idl_file "prototype.idl"] } {
    exit 1
}
idlgen_set_preferences $idlgen(cfg)
open_output_file "any_insert.java"
lappend type_list [$idlgen(root) lookup widget]
lappend type_list [$idlgen(root) lookup boolean]
lappend type_list [$idlgen(root) lookup long_array]
foreach type $type_list {
        set var_name my_[$type s_uname]
[***
@[java_any_insert_stmt $type "an_any" $var_name]@;
***]
}
close_output_file
If the type_list variable contains the type nodes for
   widget (a struct), boolean, and long_array, the
   previous Tcl script generates the following Java code:
// Java
NoPackage.widgetHelper.insert(an_any,my_widget);
an_any.insert_boolean(my_boolean);
NoPackage.long_arrayHelper.insert(an_any,my_long_array);
```

See Also

`java_any_extract_var_decl`

`java_any_extract_var_ref`
`java_any_extract_stmt`

java_array_decl_index_vars

Synopsis

`java_array_decl_index_vars` *array prefix ind_lev*

`java_gen_array_decl_index_vars` *array prefix ind_lev*

Description

This command declares a set of index variables that are used to index the specified *array*.

Parameters

array	An `array` node of the parse tree.
prefix	The prefix to be used when constructing the names of index variables. For example, the prefix `i` is used to get index variables called `i1` and `i2`.
ind_lev	The indentation level at which the `for` loop is to be created.

Notes

The array indices are declared to be of the `int` type.

Examples

Given the following IDL:

```
//IDL
typedef long            long_array[5][7];
```

The following Tcl script illustrates the use of the `java_array_decl_index_vars` command:

Example 35:

```
# Tcl
smart_source "std/output.tcl"
smart_source "std/java_poa_lib.tcl"

if { ! [idlgen_parse_idl_file "array.idl"] } {
    exit 1
}

idlgen_set_preferences $idlgen(cfg)

open_output_file "array.java"

set typedef  [$idlgen(root) lookup "long_array"]
set a        [$typedef true_base_type]
1 set indent   [java_indent [$a num_dims]]
2 set index    [java_array_elem_index $a "i"]
[***
void some_method()
{
    @[java_array_decl_index_vars $a "i" 1]@

    @[java_array_for_loop_header $a "i" 1]@
    @$indent@foo@$index@ = bar@$index@;
    @[java_array_for_loop_footer $a 1]@
}
***]
close_output_file
```

The amount of indentation to be used inside the body of the `for` loop, line 2, is calculated by using the number of dimensions in the array as a parameter to the `java_indent` command, line 1. The above Tcl script generates the following Java code:

```
// Java
void some_method()
{
    int                              i1;
    int                              i2;
    for (i1 = 0; i1 < 5  ; i1 ++) {
        for (i2 = 0; i2 < 7  ; i2 ++) {
            foo[i1][i2] = bar[i1][i2];
        }
    }
}
```

See Also

`java_gen_array_decl_index_vars`
`java_array_for_loop_header`
`java_array_elem_index`
`java_array_for_loop_footer`

java_array_elem_index

Synopsis

`java_array_elem_index` *`array prefix`*

Description

This command returns, in square brackets, the complete set of indices required to index a single element of *array*.

Parameters

array	An array node of the parse tree.
prefix	The prefix to use when constructing the names of index variables. For example, the prefix `i` is used to get index variables called `i1` and `i2`.

Examples

If `arr` is a two-dimensional array node, the following Tcl fragment:

```
# Tcl
...
set indices  [java_array_elem_index $arr "i"]
```

returns the string `"[i1][i2]"`.

See Also

`java_array_decl_index_vars`

`java_array_for_loop_header`
`java_array_for_loop_footer`

java_array_for_loop_footer

Synopsis

`java_array_for_loop_footer` *array ind_lev*

`java_gen_array_for_loop_footer` *array ind_lev*

Description

This command generates a `for` loop footer for the given *array* node with indentation given by *ind_level*.

Parameters

array	An array node of the parse tree.
ind_lev	The indentation level at which the `for` loop is created.

Notes

This command prints a number of close braces `'}'` that equals the number of dimensions of the array.

See Also

`java_array_decl_index_vars`

`java_array_for_loop_header`
`java_array_elem_index`

java_array_for_loop_header

Synopsis

`java_array_for_loop_header` *array prefix ind_lev ?declare?*

`java_gen_array_for_loop_header` *array prefix ind_lev ?declare?*

Description

This command generates the `for` loop header for the given *array* node.

Parameters

array	An array node of the parse tree.
prefix	The prefix to be used when constructing the names of index variables. For example, the prefix `i` is used to get index variables called `i1` and `i2`.

ind_lev — The indentation level at which the `for` loop is created.

declare — This optional argument is set to `1` to specify that index variables are declared locally within the `for` loop. Default value is `0`.

Examples

Given the following IDL definition of an array:

```
// IDL
typedef long long_array[5][7];
```

The following Tcl script illustrates the use of the `java_array_for_loop_header` command:

```
# Tcl
...
set typedef  [$idlgen(root) lookup "long_array"]
set a        [$typedef true_base_type]
[***
    @[java_array_for_loop_header $a "i" 1]@
***]
```

This produces the following Java code::

```
// Java
    for (i1 = 0; i1 < 5; i1++) {
        for (i2 = 0; i2 < 7; i2++) {
```

Alternatively, using the command `java_array_for_loop_header $a "i" 1 1` results in the following Java code:

```
// Java
    for (int i1 = 0; i1 < 5; i1++) {
        for (int i2 = 0; i2 < 7; i2++) {
```

See Also

`java_array_decl_index_vars`

`java_gen_array_for_loop_header`
`java_array_elem_index`
`java_array_for_loop_footer`

java_assign_stmt

Synopsis

`java_assign_stmt` *type name value ind_lev direction*

`java_gen_assign_stmt` *type name value ind_lev direction*

Description

This command returns the Java statement (with the terminating `';'`) that assigns *value* to the variable *name*, where both are of the same *type*.

Parameters

type — A type node of the parse tree.

name — The name of the variable that is assigned to (left-hand side of assignment).

value — A variable reference that is assigned from (right-hand side of assignment).

ind_lev — Ignored.

direction	The parameter passing mode—one of `in`, `inout`, `out`, or `return`.

Notes

The command generates a shallow copy assignment for all types except arrays, for which it generates a deep copy assignment.

If the *direction* is specified as `inout` or `out`, the left-hand side of the generated assignment statement becomes *name*`.value`, as is appropriate for `Holder` types.

Examples

The following Tcl script illustrates the use of the `java_assign_stmt` command:

```
# Tcl

smart_source "std/output.tcl"
smart_source "std/java_poa_lib.tcl"

if { ! [idlgen_parse_idl_file "prototype.idl"] } {
    exit 1
}
idlgen_set_preferences $idlgen(cfg)
open_output_file "assign_stmt.java"

set op         [$idlgen(root) lookup "foo::op"]
set ind_lev    1

[***
    //--------
    // Initialize "in" and "inout" parameters
    //--------
***]
foreach arg [$op args {in inout}] {
    set arg_name [java_l_name $arg]
    set type [$arg type]
    set dir      [$arg direction]
    set value "other_[$type s_uname]"
    java_gen_assign_stmt $type $arg_name $value $ind_lev
   $dir
}
close_output_file
```

The Tcl script initializes the `in` and `inout` parameters of the `foo::op` operation. There is one `in` parameter, of `widget` type, and one `inout` parameter, of `string` type.

```
// Java
    //--------
    // Initialize "in" and "inout" parameters
    //--------
    p_widget = other_widget;
    p_string.value = other_string;
```

Assignment to the `p_string` parameter, which is declared as a `Holder` type, is done by assigning to `p_string.value`.

See Also

`java_gen_assign_stmt`

`java_assign_stmt_array`
`java_clt_par_ref`

java_assign_stmt_array

Synopsis

`java_assign_stmt_array` *`type name value ind_lev ?scope?`*

Description

This command generates nested `for` loops that assign *`value`* to the *`name`*, where both are *`type`* arrays.

Parameters

type	A type node of the parse tree.
name	The name of the variable that is assigned to (left hand side of assignment).
value	The name of the variable that is assigned from (right hand side of assignment).
ind_lev	Initial level of indentation for the generated code.
scope	(Optional) If equal to `1`, the lines of generated code are enclosed in curly braces. Otherwise the braces are omitted. The default is `1`.

Examples

The following Tcl script illustrates the use of the `java_assign_stmt_array` command:

```
# Tcl
smart_source "std/output.tcl"
smart_source "std/java_poa_lib.tcl"

if { ! [idlgen_parse_idl_file "array.idl"] } {
    exit 1
}

idlgen_set_preferences $idlgen(cfg)
open_output_file "assign_array.java"

set typedef  [$idlgen(root) lookup "long_array"]
set a        [$typedef true_base_type]
set indent   [java_indent [$a num_dims]]
set index    [java_array_elem_index $a "i"]
set assign_stmt [java_assign_stmt_array $a "arr1" "arr2"
   1]

[***
void some_method()
{
    @$assign_stmt@
}
***]
close_output_file
```

Given the following IDL definition of long_array:

```
// IDL
typedef long             long_array[5][7];
```

The Tcl script generates the following Java code:

```
// Java
void some_method()
{
    {
        for (int i1 = 0; i1 < 5  ; i1 ++) {
            for (int i2 = 0; i2 < 7  ; i2 ++) {
                arr1[i1][i2] = arr2[i1][i2];
            }
        }
    }
}
```

An extra set of braces is generated to enclose the `for` loops because *scope* has the default value `1`.

See Also

`java_gen_assign_stmt`

`java_assign_stmt`
`java_clt_par_ref`

java_attr_acc_sig

Synopsis

`java_attr_acc_sig` *attribute*

`java_gen_attr_acc_sig` *attribute*

Description

This command returns the signature of an attribute accessor operation.

Parameters

attribute	An attribute node of the parse tree.

Notes

Neither the `java_attr_acc_sig` nor the `java_gen_attr_acc_sig` command put a `';'` (semicolon) at the end of the generated signature.

Examples

Consider the following sample IDL:

```
// IDL
// File: 'finance.idl'
interface Account {
    attribute long accountNumber;
    attribute float balance;
    void makeDeposit(in float amount);
};
```

The following Tcl script illustrates the use of the java_attr_acc_sig command:

```
# Tcl
smart_source "std/output.tcl"
smart_source "std/java_poa_lib.tcl"
if { ! [idlgen_parse_idl_file "finance.idl"] } {
    exit 1
}
idlgen_set_preferences $idlgen(cfg)
open_output_file "signatures.java"
set attr [$idlgen(root) lookup "Account::balance"]
set attr_acc_sig [java_attr_acc_sig $attr]
output "$attr_acc_sig \n\n"
close_output_file
```

The previous Tcl script generates the following Java code:

```
// Java
public float balance()
```

See Also

java_attr_acc_sig_h
java_gen_attr_acc_sig_cc
java_attr_mod_sig_h
java_attr_mod_sig_cc

java_attr_mod_sig

Synopsis

java_attr_mod_sig *attribute*

java_gen_attr_mod_sig *attribute*

Description

This command returns the signature of the attribute modifier operation.

Parameters

attribute	Attribute node in parse tree.

Notes

Neither the java_attr_mod_sig nor the java_gen_attr_mod_sig put a ';' (semicolon) at the end of the generated statement.

Examples

Consider the following sample IDL:

```
// IDL
// File: 'finance.idl'
interface Account {
    attribute long accountNumber;
    attribute float balance;
    void makeDeposit(in float amount);
};
```

The following Tcl script illustrates the use of the `java_attr_mod_sig` command:

```
# Tcl
smart_source "std/output.tcl"
smart_source "std/java_poa_lib.tcl"

if { ! [idlgen_parse_idl_file "finance.idl"] } {
    exit 1
}
idlgen_set_preferences $idlgen(cfg)

open_output_file "signatures.java"

set attr [$idlgen(root) lookup "Account::balance"]
set attr_mod_sig [java_attr_mod_sig $attr]

output "$attr_mod_sig \n\n"
java_gen_attr_mod_sig $attr

close_output_file
```

The previous Tcl script generates the following Java code:

```
// Java
public void balance(float  _new_value)

public void balance(float  _new_value)
```

See Also

`java_attr_acc_sig_h`
`java_attr_acc_sig_cc`
`java_attr_mod_sig_h`
`java_gen_attr_mod_sig_cc`

java_branch_case_l_label

Synopsis

`java_branch_case_l_label` *`union_branch`*

Description

This command returns a non-scoped label for the *`union_branch`* union branch. The `case` keyword prefixes the label unless the label is `default`. The returned value omits the terminating `':'` (colon).

Parameters

`union_branch` A `union_branch` node of the parse tree.

Notes

This command generates labels for all union discriminator types. Labels that clash with Java keywords are prefixed with an _ (underscore) character.

Examples

Consider the following IDL:

```
// IDL
module m {
        enum colour {red, green, blue};

        union foo switch(colour) {
            case red:    long     a;
            case green:  string   b;
            default:     short    c;
        };
};
```

The following Tcl script illustrates the use of the `java_branch_case_l_label` command:

```
# Tcl
...
set union [$idlgen(root) lookup "m::foo"]
foreach branch [$union contents {union_branch}] {
        output [java_branch_case_l_label $branch]
        output "\n"
}; # foreach
```

The previous Tcl script generates the following Java code:

```
//Java
red
green
default
```

See Also

`java_branch_case_s_label`
`java_branch_l_label`
`java_branch_s_label`

java_branch_case_s_label

Synopsis

`java_branch_case_s_label` *union_branch*

Description

This command returns a scoped label for the *union_branch* union branch. The `case` keyword prefixes the label unless the label is `default`. The returned value omits the terminating `':'` (colon).

Parameters

union_branch	A `union_branch` node of the parse tree.

Notes

This command generates labels for all union discriminator types. Labels that clash with Java keywords are prefixed with an _ (underscore) character.

Examples

Consider the following IDL:

```
// IDL
module m {
        enum colour {red, green, blue};
        union foo switch(colour) {
            case red:    long     a;
            case green:  string   b;
            default:     short    c;
        };
};
```

The following Tcl script illustrates the use of the `java_branch_case_s_label` command:

```
# Tcl
...
set union [$idlgen(root) lookup "m::foo"]
foreach branch [$union contents {union_branch}] {
        output [java_branch_case_s_label $branch]
        output "\n"
}; # foreach
```

The following output is generated by the Tcl script:

```
//Java
case NoPackage.m.colour._red
case NoPackage.m.colour._green
default
```

Case labels are generated in the form `NoPackage.m.colour._red` (of integer type) instead of `NoPackage.m.color.red` (of `NoPackage.m.colour` type) because an integer type must be used in the branches of the switch statement.

See Also

`java_branch_case_l_label`

`java_branch_l_label`
`java_branch_s_label`

java_branch_l_label

Synopsis

`java_branch_l_label` *`union_branch`*

Description

This command returns a non-scoped label for the *`union_branch`* union branch.

Parameters

union_branch	A union_branch node of the parse tree.

Notes

This command generates labels for all union discriminator types. Labels that clash with Java keywords are prefixed with an _ (underscore) character.

Examples Consider the following IDL:

```
// IDL
module m {
        enum colour {red, green, blue};
        union foo switch(colour) {
            case red:    long     a;
            case green:  string   b;
            default:     short    c;
        };
};
```

The following Tcl script illustrates the use of the `java_branch_l_label` command:

```
# Tcl
...
set union [$idlgen(root) lookup "m::foo"]
foreach branch [$union contents {union_branch}] {
        output [java_branch_l_label $branch]
        output "\n"
}; # foreach
```

The previous Tcl script generates the following Java code:

```
//Java
red
green
default
```

See Also

`java_branch_case_l_label`

`java_branch_case_s_label`
`java_branch_s_label`

java_branch_s_label

Synopsis `java_branch_s_label` *union_branch*

Description Returns a scoped label for a *union_branch* union branch.

Parameters

union_branch A `union_branch` node of the parse tree.

Notes This command generates labels for all union discriminator types.

Examples Consider the following IDL:

```
// IDL
module m {
        enum colour {red, green, blue};

        union foo switch(colour) {
            case red:    long     a;
            case green:  string   b;
            default:     short    c;
        };
};
```

The following Tcl script illustrates the use of the `java_branch_s_label` command:

```
# Tcl
...
set union [$idlgen(root) lookup "m::foo"]
foreach branch [$union contents {union_branch}] {
        output [java_branch_s_label $branch]
        output "\n"
}; # foreach
```

The previous Tcl script generates the following Java code:

```
// Java
NoPackage.m.colour._red
NoPackage.m.colour._green
default
```

See Also

`java_branch_case_l_label`

`java_branch_case_s_label`
`java_branch_l_label`

java_clt_par_decl

Synopsis

`java_clt_par_decl` *`name type direction`*

`java_clt_par_decl` *`arg`*
`java_clt_par_decl` *`op`*
`java_gen_clt_par_decl` *`name type direction ind_lev`*
`java_gen_clt_par_decl` *`arg ind_lev`*
`java_gen_clt_par_decl` *`op ind_lev`*

Description

This command returns a Java statement that declares a client-side parameter or return value variable.

Parameters

name	The name of the parameter or return value variable.
type	A type node of the parse tree that describes the type of this parameter or return value.
direction	The parameter passing mode—one of `in`, `inout`, `out`, or `return`.
arg	An argument node of the parse tree.
op	An operation node of the parse tree.
ind_lev	The number of levels of indentation (`gen` variants only).

Notes

The following variants of the command are supported:

- The first form of the command is used to declare an explicitly named parameter variable.
- The second form is used to declare a parameter.
- The third form is used to declare a return value.
- The non-`gen` forms of the command omit the terminating `';'` (semicolon) character.

- The `gen` forms of the command include the terminating `';'` (semicolon) character.

Examples

The following IDL is used in this example:

```
// IDL
struct  widget           {long a;};
typedef sequence<long>   longSeq;
typedef long             long_array[10];

interface foo {
    longSeq op(
        in widget       p_widget,
        inout string    p_string,
        out longSeq     p_longSeq,
        out long_array  p_long_array);
};
```

The Tcl script below illustrates how to declare Java variables that are intended to be used as parameters to (or the return value of) an operation call:

```
# Tcl
...
set op         [$idlgen(root) lookup "foo::op"]
set ind_lev    2
set arg_list   [$op contents {argument}]
[***
    //--------
    // Declare parameters for operation
    //--------
***]
foreach arg $arg_list {
   java_gen_clt_par_decl $arg $ind_lev
}
java_gen_clt_par_decl $op $ind_lev
```

The previous Tcl script generates the following Java code:

```
//Java
//--------
// Declare parameters for operation
//--------
NoPackage.widget                p_widget;
org.omg.CORBA.StringHolder      p_string;
NoPackage.longSeqHolder         p_longSeq;
NoPackage.long_arrayHolder      p_long_array;
int[]                           _result;
```

The last line declares the name of the return value to be `_result`, which is the default value of the variable `$pref(java,ret_param_name)`.

See Also

`java_gen_clt_par_decl`

`java_clt_par_ref`

java_clt_par_ref

Synopsis

`java_clt_par_ref` *name type direction*

`java_clt_par_ref` *arg*

```
java_clt_par_ref op
```

Description

This command returns *name*.value, if the parameter *direction* is inout or out (as is appropriate for Holder types). Otherwise it returns *name*.

Description

The single argument forms of this command derive the *name*, *type*, and *direction* from the given *arg* argument node or *op* operation node.

Parameters

name	The name of the parameter or return value variable.
type	A type node of the parse tree that describes the type of this parameter or return value.
direction	The parameter passing mode—one of in, inout, out or return.
is_var	The boolean flag to indicate that the parameter variable is a _var type. A value of 1 indicates a _var type.
arg	An argument node of the parse tree.
op	An operation node of the parse tree.

Examples

Given this IDL:

```
// IDL
struct  widget           {long a;};
typedef sequence<long>   longSeq;
typedef long             long_array[10];

interface foo {
    longSeq op(
        in widget       p_widget,
        inout string    p_string,
        out longSeq     p_longSeq,
        out long_array  p_long_array);
};
```

The following Tcl script shows how to initialize in and inout parameters:

Example 36: *Initializing in and inout Parameters*

```
# Tcl
...
[***
    //--------
    // Initialize "in" and "inout" parameters
    //--------
***]
1 foreach arg [$op args {in inout}] {
    set arg_name [java_l_name $arg]
    set type [$arg type]
    set dir      [$arg direction]
2   set arg_ref [java_clt_par_ref $arg_name $type $dir]
    set value "other_[$type s_uname]"
3   java_gen_assign_stmt $type $arg_ref $value $ind_lev
   $dir
}
```

1. The foreach loop iterates over all the in and inout parameters.
2. The java_clt_par_ref command is used to obtain a reference to a parameter
3. This reference can then be used to initialize the parameter with the java_gen_assign_stmt command.

The previous Tcl script generates the following Java code:

```
//Java
//--------
// Initialize "in" and "inout" parameters
//--------
p_widget = other_widget;
p_string.value = other_string;
```

See Also

java_clt_par_decl

java_assign_stmt
java_gen_assign_stmt
java_l_name

java_gen_array_decl_index_vars

java_gen_array_decl_index_vars *array prefix ind_lev*

Description

This command is a variant of "java_array_decl_index_vars" that prints its result directly to the current output.

java_gen_array_for_loop_footer

Synopsis

java_gen_array_for_loop_footer *array ind_lev*

Description

This command is a variant of "java_array_for_loop_footer" that prints its result directly to the current output.

java_gen_array_for_loop_header

Synopsis

java_gen_array_for_loop_header *array prefix ind_lev ?declare?*

Description

This command is a variant of "java_array_for_loop_header" that prints its result directly to the current output.

java_gen_assign_stmt

Synopsis

java_gen_assign_stmt *type name value ind_lev ?scope?*

Description

This command is a variant of "java_assign_stmt" that prints its result directly to the current output.

java_gen_attr_acc_sig

Synopsis

java_gen_attr_acc_sig *attribute*

Description

This command is a variant of "java_attr_acc_sig" that prints its result directly to the current output.

java_gen_attr_mod_sig

Synopsis

java_gen_attr_mod_sig *attribute*

Description This command is a variant of "java_attr_mod_sig" that prints its result directly to the current output.

java_gen_clt_par_decl

Synopsis `java_gen_clt_par_decl` *name type direction ?ind_lev?*

Description This command is a variant of "java_clt_par_decl" that prints its result directly to the current output.

java_gen_op_sig

Synopsis `java_gen_op_sig` *op*

Description This command is a variant of "java_op_sig" that prints its result directly to the current output.

java_gen_srv_par_alloc

Synopsis `java_gen_srv_par_alloc` *name type direction ind_lev*

`java_gen_srv_par_alloc` *arg ind_lev*
`java_gen_srv_par_alloc` *op ind_lev*

Description This command is a variant of "java_srv_par_alloc" that prints its result directly to the current output.

java_gen_srv_ret_decl

Synopsis `java_gen_srv_ret_decl` *name type ind_lev*

Description This command is a variant of "java_srv_ret_decl" that prints its result directly to the current output.

java_gen_var_decl

Synopsis `java_gen_var_decl` *name type ind_lev*

Description This command is a variant of "java_var_decl" that prints its result directly to the current output.

java_helper_name

Synopsis `java_helper_name` *type*

Description This command returns the scoped name of the `Helper` class associated with *type*.

Parameters

type	A type node of the parse tree.

Notes A special cases arises if an IDL interface is called, for example, `FooHelper`. Because `FooHelper` risks clashing with the `Helper` class for the IDL `Foo` type, the OMG IDL-to-Java mapping has a special rule for identifiers of this type. The IDL identifier `FooHelper` is mapped to `_FooHelper`, its associated `Helper` class maps to `_FooHelperHelper`, and its associated `Holder` class maps to `_FooHelperHolder`.

Primitive IDL types (such as `long` and `boolean`) do not have associated `Helper` classes.

Examples

Given the following IDL:

```
//IDL
struct Widget {
    short s;
};

typedef string StringAlias;

interface Foo {
    void dummy();
};

interface FooHelper {
    void dummy();
};

interface FooHolder {
    void dummy();
};
```

Examples of Java identifiers returned by `[java_helper_name $type]` are given in Table 23:

Table 23: *Helper Classes for User-Defined Types*

Java Name of `$type`	**Output from** `java_helper_name` **Command**
`NoPackage.Widget`	`NoPackage.WidgetHelper`
`NoPackage.StringAlias`	`NoPackage.StringAliasHelper`
`NoPackage.Foo`	`NoPackage.FooHelper`
`NoPackage._FooHelper`	`NoPackage._FooHelperHelper`
`NoPackage._FooHolder`	`NoPackage._FooHolderHelper`

See Also

`java_holder_name`

java_holder_name

Synopsis

`java_holder_name` *`type`*

Description

This command returns the scoped name of the `Holder` class associated with *`type`*.

Parameters

`type`	A type node of the parse tree.

Notes

A special cases arises if an IDL interface is called, for example, `FooHolder`. Because `FooHolder` risks clashing with the `Holder` class for the IDL `Foo` type, the OMG IDL-to-Java mapping has a special rule for identifiers of this type. The IDL identifier `FooHolder` is mapped to `_FooHolder`, its associated `Helper` class maps to `_FooHolderHelper`, and its associated `Holder` class maps to `_FooHolderHolder`.

Examples

Given the following IDL:

```
//IDL
struct Widget {
    short s;
};

typedef string StringAlias;

interface Foo {
    void dummy();
};

interface FooHelper {
    void dummy();
};

interface FooHolder {
    void dummy();
};
```

Examples of Java identifiers returned by `[java_holder_name $type]` are given in Table 24:

Table 24: *Holder Classes for User-Defined Types*

Java Name of `$type`	Output from `java_holder_name` Command
long	IntHolder
boolean	BooleanHolder
NoPackage.Widget	NoPackage.WidgetHolder
NoPackage.StringAlias	NoPackage.StringAliasHolder
NoPackage.Foo	NoPackage.FooHolder
NoPackage._FooHelper	NoPackage._FooHelperHolder
NoPackage._FooHolder	NoPackage._FooHolderHolder

See Also `java_helper_name`

java_impl_class

Synopsis `java_impl_class` *`interface`*

Description This command returns the name of the Java class that implements the specified IDL interface.

Parameters

interface	An interface node of the parse tree.

Notes The class name is constructed by getting the fully scoped name of the IDL interface, replacing all occurrences of `'::'` with `'.'` and appending `$pref(java,impl_class_suffix)`, which has the default value `Impl`.

Examples

Consider the following Tcl script:

```
# Tcl
...
set class [java_impl_class $inter]
[***
public class @$class@ {
    //...
};
***]
```

The following interface definitions result in the generation of the corresponding Java code.

Interface	Java Code

`interface Cow {`
`    //...`
`};`
	`public class NoPackage.CowImpl {`
	`    //...`
	`};`

`module Farm {`
`   interface Cow`
`   {`
`        //...`
`    };`
`};`
	`public class NoPackage.Farm.CowImpl {`
	`    //...`
	`};`

java_indent

Synopsis

`java_indent` *`ind_lev`*

Description

This command returns the string given by `$pref(java,indent)` concatenated with itself `$ind_lev` times. The default value of `$pref(java,indent)` is four spaces.

Parameters

`ind_lev` The number of levels of indentation required.

Examples

Consider the following Tcl script:

```
#Tcl
puts "[java_indent 1]One"
puts "[java_indent 2]Two"
puts "[java_indent 3]Three"
```

This produces the following output:

```
    One
        Two
            Three
```

java_is_basic_type

Synopsis

`java_is_basic_type` *`type`*

Description

This command returns TRUE if *`type`* represents a built-in IDL type.

Parameters

`type` A type node of the parse tree.

Notes This command is the opposite of "java_user_defined_type". It is TRUE when java_user_defined_type is FALSE, and vice-versa.

See Also java_user_defined_type

java_is_keyword

Synopsis java_is_keyword *string*

Description This command returns TRUE if the specified *string* is a Java keyword, otherwise it returns FALSE.

Parameters

string	The string containing the identifier to be tested.

Notes This command is called internally from other commands in the std/java_poa_lib.tcl library.

Examples For example:

```
# Tcl
java_is_keyword "new"; # returns 1
java_is_keyword "cow"; # returns 0
```

java_list_recursive_member_types

Synopsis java_list_recursive_member_types

Description This command returns a list of all user-defined type nodes that represent IDL recursive member types.

Examples Consider the following IDL:

```
//IDL
struct Recur {
    string name;
    sequence<Recur> RecurSeq;
};

struct Ordinary {
    string name;
    short  s;
};

interface TestRecursive {
    Recur get_recursive_struct();
};
```

The Recur struct is a recursive type because one of its member types, sequence<Recur>, refers to the struct in which it is defined. The sequence<Recur> member type is an example of a recursive member type.

The following Tcl script is used to parse the IDL file:

```
# Tcl
smart_source "std/output.tcl"
smart_source "std/java_poa_lib.tcl"

if { ! [idlgen_parse_idl_file "recursive.idl"] } {
    exit 1
}
idlgen_set_preferences $idlgen(cfg)

open_output_file "recursive.java"

set type_list [java_list_recursive_member_types]

foreach type $type_list {
    output "recursive type: "
    output [java_s_name $type]
    output "\n"
    set parent [$type defined_in]
    output "parent of recursive type: "
    output [java_s_name $parent]
    output "\n\n"
}
close_output_file
```

The output of this Tcl script is as follows:

```
recursive type: <anonymous-sequence>
parent of recursive type: Recur
```

One recursive member type, corresponding to `sequence<Recur>`, is found and this member is defined in the `Recur` struct.

java_l_name

Synopsis

`java_l_name` *node*

Description

This command returns the Java mapping of the node's local name.

Parameters

node	A node of the parse tree.

Notes

For user-defined types the return value of `java_l_name` is usually the same as the node's local name, but prefixed with _ (underscore) if the local name conflicts with a Java keyword.

If the node represents a built-in IDL type then the result is the Java mapping of the type. For example:

IDL Type	Java Mapping
short	short
unsigned short	short
long	int
unsigned long	int
char	char
octet	byte
boolean	boolean
string	java.lang.String
float	float
double	double
any	org.omg.CORBA.Any
Object	org.omg.CORBA.Object

When `java_l_name` is invoked on a parameter node, it returns the name of the parameter variable as it appears in IDL.

See Also

`java_s_name`

`java_s_uname`
`java_clt_par_decl`
`java_gen_clt_par_decl`

java_op_sig

Synopsis

`java_op_sig` *op*

`java_gen_op_sig` *op*

Description

This command generates the Java signature of the *op* operation.

Parameters

op	An operation node of the parse tree.

Notes

Neither the java_op_sig nor the java_gen_op_sig command put a `';'` (semicolon) at the end of the generated statement.

Examples

Consider the following sample IDL:

```
// IDL
// File: 'finance.idl'
interface Account {
    attribute long accountNumber;
    attribute float balance;
    void makeDeposit(in float amount);
};
```

The following Tcl script illustrates the use of the command:

```
# Tcl
smart_source "std/output.tcl"
smart_source "std/java_poa_lib.tcl"

if { ! [idlgen_parse_idl_file "finance.idl"] } {
    exit 1
}
idlgen_set_preferences $idlgen(cfg)

open_output_file "signatures.java"

set op [$idlgen(root) lookup "Account::makeDeposit"]
set op_sig [java_op_sig $op]
output "$op_sig \n\n"

java_gen_op_sig $op

close_output_file
```

The previous Tcl script generates the following Java code:

```
//Java
public void makeDeposit(
        float                           amount
)

public void makeDeposit(
        float                           amount
)
```

See Also

`java_op_sig_h`

`java_gen_op_sig_cc`

java_package_name

Synopsis

`java_package_name` *node*

Description

This command returns the Java package name within which this *node* occurs.

Parameters

node	A node of the parse tree.

Notes

User-defined IDL types are prefixed by the default scope.

java_param_sig

Synopsis

`java_param_sig` *name type direction*

`java_param_sig` *arg*

Description

This command returns the Java signature of the given parameter.

Parameters

name	The name of a parameter or return value variable.

type	A type node of the parse tree that describes the type of this parameter or return value.
direction	The parameter passing mode—one of `in`, `inout`, `out`, or `return`.
arg	An argument node of the parse tree.

Notes

This command is useful when you want to generate signatures for functions that use IDL data types. The following variants of the command are supported:

- The first form of the command returns the appropriate Java type for the given *type* and *direction*, followed by the given *name*.
- The second form of the command returns output similar to the first but extracts the *type*, *direction* and *name* from the *arg* argument node.

The result contains white space padding to vertically align parameter names when parameters are output one per line. The amount of padding is determined by `$pref(java,max_padding_for_types)`.

Examples

Consider the following Tcl extract:

```
# Tcl
...
set type [$idlgen(root) lookup "string"]
set dir "in"
puts "[java_param_sig "foo" $type $dir]"
```

The previous Tcl script generates the following Java code:

```
//Java
java.lang.String foo
```

See Also

`java_param_type`

`java_gen_operation_h`
`java_gen_operation_cc`

java_param_type

Synopsis

`java_param_type` *type direction*

`java_param_type` *arg*
`java_param_type` *op*

Description

This command returns the Java parameter type for the node specified in the first argument.

Parameters

type	A type node of the parse tree that describes the type of this parameter or return value.
direction	The parameter passing mode—one of `in`, `inout`, `out`, or `return`.
arg	An argument node of the parse tree.
op	An operation node of the parse tree.

Notes

This command is useful when you want to generate signatures for methods that use IDL data types. The following variants of the command are supported:

- The first form of the command returns the appropriate Java type for the given *type* and *direction*.
- The second form of the command returns output similar to the first but extracts the *type* and *direction* from the argument node *arg*.
- The third form of this command is a shorthand for `[java_param_type [$op return_type] "return"]`. It returns the Java type for the return value of the given *op*.

The result contains white space padding to vertically align parameter names when parameters are output one per line. The amount of padding is determined by `$pref(java,max_padding_for_types)`.

Examples

The following Tcl extract prints out `java.lang.String`:

```
# Tcl
...
set type [$idlgen(root) lookup "string"]
set dir "in"
puts "[java_param_type $type $dir]"
```

See Also

`java_param_sig`

`java_gen_operation`

java_poa_class_l_name

Synopsis

`java_poa_class_l_name` *`interface`*

Description

This command returns the local name of the POA skeleton class for that interface.

Parameters

interface	An interface node of the parse tree.

Examples

Given an interface node `$inter`, the following Tcl extract shows how the command is used:

```
#Tcl
...
set class [java_impl_class $inter]
[***
public class @$class@ extends @[java_poa_class_l_name
   $inter]@
{
    //...
};
***]
```

The following interface definitions results in the generation of the corresponding Java code:.

Interface	Java Mapping

`interface Cow {`
`    //...`
`};`
	`public class NoPackage.CowImpl`
	`    extends CowPOA`
	`{`
	`    //...`
	`};`

`module Farm {`
`    interface`
`   Cow{`
`    //...`
`    };`
`};`
	`public class NoPackage.Farm.CowImpl`
	`    extends CowPOA`
	`{`
	`    //...`
	`};`

See Also java_poa_class_s_name

java_poa_class_s_name

Synopsis `java_poa_class_s_name` *`interface`*

Description This command returns the fully scoped name of the POA skeleton class for that interface.

Parameters

`interface` An interface node of the parse tree.

Examples Given an interface node `$inter`, the following Tcl extract shows how the command is used:

```
# Tcl
...
set class [java_impl_class $inter]
[***
public class @$class@ extends @[java_poa_class_s_name
   $inter]@
{
    //...
};
***]
```

The following interface definitions results in the generation of the corresponding Java code:.

Interface	Java Mapping

`interface Cow {`
`    //...`
`};`
	`public class NoPackage.CowImpl`
	`    extends NoPackage.CowPOA`
	`{`
	`    //...`
	`};`

`module Farm {`
`    interface`
`   Cow{`
`    //...`
`    };`
`};`
	`public class NoPackage.Farm.CowImpl`
	`    extends NoPackage.Farm.CowPOA`
	`{`
	`    //...`
	`};`

See Also java_poa_class_l_name

java_poa_tie_s_name

Synopsis java_poa_tie_s_name *interface*

Description This command returns the name of the POA tie template for the IDL interface.

Parameters

interface An interface node of the parse tree.

Examples Given an interface node $inter, the following Tcl extract shows how the command is used:

```
# Tcl
...
set class [java_impl_class $inter]
[***
    @$class@ tied_object = new @$class@();
    @[java_poa_class_s_name $inter]@ the_tie =
        new @[java_poa_tie_s_name $inter]@(tied_object);
***]
```

If $inter is set to the node representing the IDL interface Cow, the Tcl code produces the following output:

```
// Java
    CowImpl tied_object = new CowImpl();
    NoPackage.CowPOA the_tie =
            new NoPackage.CowPOAtie(tied_object);
```

See Also java_poa_class_s_name

java_ret_assign

Synopsis java_ret_assign *op*

Description This command returns the "_result =" string (or a blank string, "", if *op* has a void return type).

Parameters

op An operation node of the parse tree.

Notes The name of the result variable is given by $pref(java,ret_param_name). The default is _result.

See Also java_assign_stmt

java_gen_assign_stmt

java_s_name

Synopsis java_s_name *node*

Description This command returns the Java mapping of the node's scoped name.

Parameters

node A node of the parse tree.

Notes

This command is similar to the `java_l_name` command, but it returns the fully scoped name of the Java mapping type, rather than the local name.

Built-in IDL types are mapped as they are in the `java_l_name` command.

See Also

`java_l_name`

`java_s_uname`

java_s_uname

Synopsis

`java_s_uname` *node*

Description

This command returns the node's scoped name, with each occurrence of the `::` separator replaced by an underscore `'_'` character.

Parameters

node	A node of the parse tree.

Notes

The command is similar to `[$node s_uname]` except for special-case handling of anonymous sequence and array types to give them unique names.

Examples

This routine is useful if you want to generate data types or operations for every IDL type. For example, the names of operations corresponding to each IDL type could be generated with the following statement:

```
set op_name "op_[java_s_uname $type]"
```

Some examples of IDL types and the corresponding identifier returned by `java_s_uname`:

Table 25: *Scoped Names with an Underscore Scope Delimiter*

IDL Type	Scoped Name
foo	foo
m::foo	m_foo
m::for	m_for
unsigned long	unsigned_long
sequence<foo>	_foo_seq

See Also

`java_l_name`

`java_s_name`

java_sequence_elem_index

Synopsis

`java_sequence_elem_index` *seq prefix*

Description

This command returns, in square brackets, the index of a *seq* node.

Parameters

seq	A sequence node of the parse tree.
prefix	The prefix to use when constructing the names of index variables. For example, the prefix `i` is used to get an index variable called `i1`.

Examples The following Tcl fragment:

```
# Tcl
...
set index  [java_sequence_elem_index $seq "i"]
```

returns the string, `"[i1]"`.

See Also `java_array_decl_index_vars`

`java_array_for_loop_header`
`java_array_for_loop_footer`

java_sequence_for_loop_footer

Synopsis `java_sequence_for_loop_footer` *array ind_lev*

Description This command generates a `for` loop footer for the given *seq* node with indentation given by *ind_level*.

Parameters

seq	A sequence node of the parse tree.
ind_lev	The indentation level at which the `for` loop is created.

Notes This command prints a single close brace `'}'`.

See Also `java_sequence_for_loop_header`

`java_sequence_elem_index`

java_sequence_for_loop_header

Synopsis `java_sequence_for_loop_header` *seq prefix ind_lev ?declare?*

Description This command generates the `for` loop header for the given *array* node.

Parameters

seq	Asequence node of the parse tree.
prefix	The prefix used when constructing the names of index variables. For example, the prefix `i` is used to get an index variables called `i1`.
ind_lev	The indentation level at which the `for` loop is created.
declare	(Optional) This boolean argument specifies that index variables are declared locally within the `for` loop. Default value is `0`.

Examples Given the following IDL definition of a sequence:

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

You can use the following Tcl fragment to generate the for loop header:

```
# Tcl
...
set typedef  [$idlgen(root) lookup "longSeq"]
set a        [$typedef true_base_type]
[***
    int len = foo.length;
   @[java_sequence_for_loop_header $a "i" 1 1]@
***]
```

This produces the following Java code:

```
// Java
    int len = foo.length;
    for (int i1 = 0; i1 < len; i1++) {
```

See Also

java_sequence_for_loop_footer

java_sequence_elem_index

java_srv_par_alloc

Synopsis

java_srv_par_alloc *name type direction*

java_srv_par_alloc *arg*
java_srv_par_alloc *op*
java_gen_srv_par_alloc *name type direction ind_lev*
java_gen_srv_par_alloc *arg ind_lev*
java_gen_srv_par_alloc *op ind_lev*

Description

This command returns a Java statement to allocate memory for an out parameter (or return value), if needed. If there is no need to allocate memory, this command returns an empty string.

Parameters

type	The type node of the parse tree that describes the type of this parameter or return value.
direction	The parameter passing mode—one of in, inout, out, or return.
arg	An argument node of the parse tree.
op	An operation node of the parse tree.
ind_lev	The number of levels of indentation (gen variants only).

Examples

Given the following sample IDL:

```
// IDL
struct  widget           {long a;};
typedef sequence<long>   longSeq;
typedef long             long_array[10];

interface foo {
        longSeq op(
            in widget       p_widget,
            inout string    p_string,
            out longSeq     p_longSeq,
            out long_array  p_long_array);
};
```

The following Tcl script allocates memory for `out` parameters.

```
# Tcl
smart_source "std/output.tcl"
smart_source "std/java_poa_lib.tcl"

idlgen_set_preferences $idlgen(cfg)
smart_source "std/args.tcl"

if { ! [idlgen_parse_idl_file "prototype.idl"] } {
    exit 1
}
open_output_file "srv_par_alloc.java"
set op [$idlgen(root) lookup "foo::op"]

set ind_lev    3
set arg_list   [$op contents {argument}]

[***
       //--------
       // Allocate memory for "out" parameters.
       //--------
***]
foreach arg [$op args {out}] {
        java_gen_srv_par_alloc $arg $ind_lev
}
close_output_file
```

The previous Tcl script generates the following Java code:

```
// Java
        //--------
        // Allocate memory for "out" parameters.
        //--------
         p_longSeq = new NoPackage.longSeqHolder();
         p_long_array = new NoPackage.long_arrayHolder();
```

See Also

`java_gen_srv_par_alloc`

`java_srv_par_ref`
`java_srv_ret_decl`

java_srv_par_ref

Synopsis

`java_srv_par_ref` *name type direction*

```
java_srv_par_ref arg
java_srv_par_ref op
```

Description

This command returns *name*`.value`, if the *direction* parameter is `inout` or `out` (as is appropriate for `Holder` types). Otherwise it returns *name*.

Description

The single argument forms of this command derive the *name*, *type*, and *direction* from the given *arg* argument node or *op* operation node.

Parameters

name	The name of the parameter or return value variable.
type	The type node of the parse tree that describes the type of this parameter or return value.
direction	The parameter passing mode—one of `in`, `inout`, `out`, or `return`.
arg	An argument node of the parse tree.
op	An operation node of the parse tree.

Examples

Given the following sample IDL:

```
// IDL
struct  widget           {long a;};
typedef sequence<long>   longSeq;
typedef long             long_array[10];

interface foo {
        longSeq op(
            in widget       p_widget,
            inout string    p_string,
            out longSeq     p_longSeq,
            out long_array  p_long_array);
};
```

The following Tcl script iterates over all `inout` and `out` parameters and the return value, and assigns values to them:

```
# Tcl
...
[***
        //--------
        // Assign new values to "out" and "inout"
        // parameters, and the return value, if needed.
        //--------
***]
foreach arg [$op args {inout out}] {
        set type    [$arg type]
        set arg_ref [java_srv_par_ref $arg]
        set name2   "other_[$type s_uname]"
[***
        @$arg_ref@ = @$name2@;
***]
}
if {[$ret_type l_name] != "void"} {
        set ret_ref [java_srv_par_ref $op]
        set name2   "other_[$ret_type s_uname]"
[***
        @$ret_ref@ = @$name2@;
***]
}
```

The `java_srv_par_ref` command returns a reference to both the parameters and the return value.

The previous Tcl script generates the following Java code:

```
//Java
//--------
// Assign new values to "out" and "inout"
// parameters, and the return value, if needed.
//--------
p_string.value = other_string;
p_longSeq.value = other_longSeq;
p_long_array.value = other_long_array;
_result = other_longSeq;
```

See Also

`java_srv_par_alloc`

`java_srv_ret_decl`

java_srv_ret_decl

Synopsis

`java_srv_ret_decl` *name type*

`java_gen_srv_ret_decl` *name type ind_lev*

Description

This command returns the Java declaration of a variable that holds the return value of an operation. If the operation does not have a return value this command returns an empty string.

Parameters

name	The name of a parameter or return value variable.

type	The type node of the parse tree that describes the type of this parameter or return value.
ind_lev	The number of levels of indentation (`gen` variants only).

Notes

Assuming that the operation does have a return value, if `alloc_mem` is `1`, the variable declaration also allocates memory to hold the return value, if necessary. If `alloc_mem` is `0`, no allocation of memory occurs, and instead you can allocate the memory later with the `java_srv_par_alloc` command. The default value of `alloc_mem` is `1`.

Examples

Given the following sample IDL:

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

interface foo {
        longSeq op();
};
```

The following Tcl script declares a local variable that can hold the return value of the operation. It then allocates memory for the return value, if required.

```
# Tcl
...
set op          [$idlgen(root) lookup "foo::op"]
set ret_type    [$op return_type]
set ind_lev     1
set arg_list    [$op contents {argument}]
if {[$ret_type l_name] != "void"} {
    set type [$op return_type]
    set ret_ref [java_srv_par_ref $op]
[***
       //--------
       // Declare a variable to hold the return value.
       //--------
       @[java_srv_ret_decl $ret_ref $type]@;

***]
}
```

The previous Tcl script generates the following Java code:

```
// Java
//--------
// Declare a variable to hold the return value.
//--------
int[]     _result;
```

See Also

`java_srv_par_alloc`

`java_srv_par_ref`
`java_gen_srv_ret_decl`

java_typecode_l_name

Synopsis

`java_typecode_l_name` *type*

Description This command returns the local Java name of the `typecode` for the specified `type`.

Parameters

type	A type node of the parse tree.

Notes For user-defined types, the command returns *localName*`Helper.type()`. For the built-in types (such as `long` and `short`), the `get_primitive_tc()` method is used to get the type code.

Examples Examples of the local names of Java type codes for IDL types:

IDL types	Local Java Type Codes
`Cow`	`CowHelper.type()`
`Farm::Cow`	`CowHelper.type()`
`long`	`org.omg.CORBA.ORB.init().get_primitive_tc` `(org.omg.CORBA.TCKind.tk_long)`

See Also `java_typecode_s_name`

java_typecode_s_name

Synopsis `java_typecode_s_name` *type*

Description This command returns the fully-scoped Java name of the `typecode` for the specified type.

Parameters

type	A type node of the parse tree.

Notes For user-defined types, an IDL type of the form *scope*`::`*localName* has the scoped type code *scope*`::`*localName*`Helper.type()`. For the built-in types (such as `long`, and `short`), the `get_primitive_tc()` method is used to get the type code.

Examples Examples of the fully-scoped names of Java type codes for IDL types:

Table 26: *Scoped Java Names of IDL Type Codes*

IDL Type	Fully Scoped Java Type Code
`Cow`	`NoPackage.CowHelper.type()`
`Farm::Cow`	`NoPackage.Farm.CowHelper.type()`
`long`	`org.omg.CORBA.ORB.init().get_primitive_tc` `(org.omg.CORBA.TCKind.tk_long)`

See Also `java_typecode_l_name`

java_user_defined_type

Synopsis `java_user_defined_type` *type*

Description This command returns TRUE if *type* represents a user-defined IDL type.

Parameters

type	A type node of the parse tree.

See Also `java_is_basic_type`

java_var_decl

Synopsis

`java_var_decl` *`name type direction`*

`java_gen_var_decl` *`name type direction ind_lev`*

Description

This command returns the Java variable declaration with the specified *`name`* and *`type`*. The *`direction`* parameter determines whether a plain type or a `Holder` type is declared.

Parameters

name	The name of the variable.
type	The type node of the parse tree that describes the type of this variable.
direction	The parameter passing mode—one of `in`, `inout`, `out`, or `return`.
ind_lev	The number of levels of indentation (`gen` variants only).

Examples

The following Tcl script illustrates how to use this command:

```
# Tcl
...
set ind_lev 1
[***
       // Declare variables
***]
foreach type $type_list {
        set in_name  "in_[$type l_name]"
        java_gen_var_decl $in_name $type "in" $ind_lev

        set inout_name  "inout_[$type l_name]"
        java_gen_var_decl $inout_name $type "inout"
   $ind_lev
}
```

If variable `type_list` contains the types `string`, `widget` (a struct), and `long_array`, the Tcl code generates the following Java code:

```
//Java
    // Declare variables
    java.lang.String                in_string;
    org.omg.CORBA.StringHolder      inout_string;
    NoPackage.widget                in_widget;
    NoPackage.widgetHolder          inout_widget;
    int[]                           in_long_array;
    NoPackage.long_arrayHolder      inout_long_array;
```

See Also `java_gen_var_decl`

Java Utility Libraries

This reference describes two libraries—the java_poa_print and java_poa_random utility libraries—that can be used in your own Tcl scripts to generate print statements or to initialize variables with random data.

java_poa_print Commands

This section gives detailed descriptions of the Tcl commands in the `java_poa_print` library.

java_gen_print_stmt

Synopsis

`java_gen_print_stmt` *`type name print_obj_loc ?indent? ?ostream?`*

Description

This command is a variant of "java_print_stmt" that prints its result directly to the current output.

java_print_func_name

Synopsis

`java_print_func_name` *`type print_obj_loc`*

Description

This command returns the name of the function that prints the given *`type`*.

Parameters

`type`	A type node of the parse tree.
`print_obj_loc`	The scope in which the `global_printer` object is defined.

Notes

The printer member function is invoked on `global_printer`.

java_print_gen_init

Synopsis

`java_print_gen_init` *`?orb?`*

Description

This command generates a statement to initialize the `global_printer` pointer.

Parameters

`orb`	(Optional) The name of a pointer to an ORB object (of `CORBA::ORB_ptr` type) passed to the `IT_GeniePrint` constructor. Default is `global_orb`.

Notes

This command must be called before generating any print statements.

java_print_stmt

Synopsis

`java_print_stmt` *`type name print_obj_loc ?indent? ?ostream?`*

`java_gen_print_stmt` *`type name print_obj_loc ?indent? ?ostream?`*

Description This command generates a statement that prints the *name* variable, which is of *type* type, to the *ostream* output stream with *indent* levels of indentation.

Parameters

type	A type node of the parse tree.
name	The name of the variable to be printed (must be a variable reference).
`print_obj_loc`	The scope in which the `global_printer` object is defined.
indent	(Optional) The number of units of indentation. Default is `0`.
`ostream`	(Optional) The name of an output stream. Default is the `System.out` standard output stream.

Notes No terminating '`;`' (semicolon) is generated in the `java_print_stmt` version of the command.

A terminating '`;`' (semicolon) is generated in the `java_gen_print_stmt` version of the command.

gen_java_print_funcs

Synopsis `gen_java_print_funcs ?`*`ignored?`*

Description This command generates an `IT_GeniePrint.java` file containing the implementation of the `IT_GeniePrint` class.

Parameters

ignored	(Optional) Retained for backwards compatibility.

java_poa_random Commands

This section gives detailed descriptions of the Tcl commands in the `java_poa_random` library.

java_gen_random_assign_stmt

Synopsis `java_gen_random_assign_stmt` *type name ?dir? random_obj_loc indent*

Description This command is a variant of "java_random_assign_stmt" that prints its result directly to the current output.

java_random_assign_stmt

Synopsis `java_random_assign_stmt` *type name ?dir? random_obj_loc*

`java_gen_random_assign_stmt` *type name ?dir? random_obj_loc indent*

Description This command generates a statement that assigns a random value to the *name* variable, which is of *type* type.

Parameters

type	A type node of the parse tree.

name	The name of the variable to which a random value is assigned.
dir	One of in, inout or out. If inout or out is specified, the .value suffix is appended to the variable name.
random_obj_loc	The scope in which the global_random object is defined.
indent	The number of units of indentation.

Notes

No terminating ';' (semicolon) is generated in the java_random_assign_stmt version of the command.

A terminating ';' (semicolon) is generated in the java_gen_random_assign_stmt version of the command.

java_random_gen_init

Synopsis

`java_random_gen_init ?orb? ?seed? ?random?`

Description

This command generates a statement to initialize the global_random pointer.

Parameters

orb	(Optional) The name of a pointer to an ORB object (of CORBA.ORB type) passed to the IT_GenieRandom constructor. Default is global_orb.
seed	(Optional) An integer seed to initialize the random number generator. Default is 0.
random	(Optional) The name of the random object pointer. Default is global_random.

Notes

A terminating ';' (semicolon) is generated.

gen_java_random_funcs

Synopsis

`gen_java_random_funcs ?ignored?`

Description

This command generates an IT_GenieRandom.java file containing the implementation of the IT_GenieRandom class.

Parameters

ignored	(Optional) Retained for backwards compatibility.

User's Reference

This appendix presents reference material about all the configuration and usage details for the idlgen interpreter and for the genies provided with the Orbix Code Generation Toolkit.

General Configuration Options

Standard configuration options

Table 27 describes the general purpose configuration options available in the standard configuration file `idlgen.cfg`.

Table 27: *Configuration File Options*

Configuration Option	Description
`idlgen.tmp_dir`	Directory for creating temporary files. If the `TMP` environment variable is set, it is used instead of the entry in the configuration file.
`default.all.want_diagnostics`	Setting for diagnostics: `true`: Genies print diagnostic messages. `false`: Genies stay silent.
`default.all.copyright`	List of lines to put in the copyright notice.
`default.html.file_ext`	File extension preferred by your web browser (`".html"` for most platforms).
`idlgen.genie_search_path`	Directories searched for genies. If the `IT_GENIE_PATH` environment variable is set, the directories specified in `IT_GENIE_PATH` are searched, followed by the default genie directory. If the `IT_GENIE_PATH` environment variable is *not* set, the directories specified in this configuration option are searched, followed by the default genie directory. This configuration option also specifies the directories searched by the `smart_source` command.

Configuration Options for C++ Genies

C++ genie configuration options

Table 28 describes the configuration options for genies that generate C++ code in the standard configuration file, idlgen.cfg:

Table 28: *Configuration File Options for C++ Genies*

Configuration Option	Purpose
default.cpp.cc_file_ext	File extension preferred by your C++ compiler (for example, cxx, cc, cpp, or C).
default.cpp.h_file_ext	File extension preferred by your C++ compiler; usually h.
default.cpp.impl_class_suffix	Suffix for your C++ classes that implement IDL interfaces.
default.cpp.factory_suffix	Suffix for used C++ valuetype factory classes.
default.cpp.max_padding_for_types	Maximum amount of padding used to align parameters.
idlgen.preprocessor.cmd	Allows you to override the location of a C++ preprocessor If not set, it defaults to a standard location relative to the environment setting, IT_PRODUCT_DIR. You should not have to change this entry.
idlgen.preprocessor.args	Arguments to pass to the preprocessor. If not set, it defaults to appropriate arguments for the IDL pre-processor. You should not have to change this entry.

cpp_poa_genie configuration options

Table 29 describes the configuration options for the cpp_poa_genie in the standard configuration file, idlgen.cfg. The options of want_*option* form can each be overridden by command line parameters when the cpp_poa_genie.tcl is run. The idlgen.cfg configuration file determines default values when the command line options are not specified

Table 29: *Configuration File Options for the cpp_poa_genie Genie*

Configuration Option	Purpose
default.cpp_poa_genie.ref_file_ext	File extension for object reference files.
default.cpp_poa_genie.idl_flags	Extra flags passed to the IDL compiler when run by the generated Makefile.

Table 29: *Configuration File Options for the cpp_poa_genie Genie*

Configuration Option	Purpose
default.cpp_poa_genie.cpp_flags	Extra flags passed to the C++ compiler when run by the generated Makefile.
default.cpp_poa_genie.link_flags	Extra link flags passed to the C++ linker when run by the generated Makefile.
default.cpp_poa_genie.want_include	Generate code for #included IDL files.
default.cpp_poa_genie.want_servant	Generate code for servant classes to implement IDL interfaces.
default.cpp_poa_genie.want_client	Generate a test client main.
default.cpp_poa_genie.want_server	Generate a test server main.
default.cpp_poa_genie.want_makefile	Generate a makefile to build the test client and server.
default.cpp_poa_genie.want_tie	Use the POA tie approach instead of direct inheritance from POA skeleton classes.
default.cpp_poa_genie.want_inherit	If IDL interfaces inherit from each other, make the servants for those interfaces have corresponding inheritance.
default.cpp_poa_genie. want_default_poa	Override the _default_POA function on each servant.
default.cpp_poa_genie.want_refcount	Make servants reference-counted.
default.cpp_poa_genie.want_var	Use _var variables in the generated code.
default.cpp_poa_genie.want_complete	Generate a complete client and complete servants.
default.cpp_poa_genie.want_threads	Use the multi-threaded POA policy.
default.cpp_poa_genie.want_ns	Use the naming service.

Configuration Options for Java Genies

Java genie configuration options

Table 30 describes the configuration options specific to Java genies in the standard configuration file idlgen.cfg:

Table 30: *Configuration File Options for Java Genies*

Configuration Option	Purpose
default.java.java_file_ext	File extension preferred by your Java compiler.
default.java.java_class_ext	Class name extension preferred by your Java compiler.
default.java.server_name	Default server name.

Table 30: *Configuration File Options for Java Genies*

Configuration Option	Purpose
default.java.impl_class_suffix	Suffix for your Java classes that implement IDL interfaces.
default.java.print_prefix	Prefix for your Java classes that implement print methods for IDL types.
default.java.random_prefix	Prefix for your Java classes that implement random methods for IDL types.
default.java.printpackage_name	Package name in which the print and random classes are put—default is idlgen.
default.java.want_javadoc_comments	Flag that specifies if you want javadoc comments in the generated code—default is "false".
default.java.max_padding_for_types	Maximum amount of padding used to align parameters.
default.java.attr_mod_param_name	Variable name used when declaring an attribute if none is supplied—default is "_new_value".
default.java.ret_param_name	Variable name used to hold a return value—default is "_result".
default.java.ant_home	The install directory for the ant build utility.

java_poa_genie configuration options

Table 31 describes the configuration options for the java_poa_genie in the standard configuration file, java_idlgen.cfg. The options of the want_*option* form can each be overridden by command line parameters when the java_poa_genie.tcl is run. The java_idlgen.cfg configuration file determines default values when the command-line options are not specified

Table 31: *Configuration File Options for the java_poa_genie Genie*

Configuration Option	Purpose
default.java_poa_genie.ref_file_ext	File extension for object reference files. Test clients and servers use these files to pass object references.
default.java_poa_genie.package_name	The package that the generated code will be contained in.
default.java_poa_genie.want_include	Generate code for #included IDL files.
default.java_poa_genie.want_servant	Generate code for servant classes to implement IDL interfaces.
default.java_poa_genie.want_client	Generate a test client main.
default.java_poa_genie.want_server	Generate a test server main.

Table 31: *Configuration File Options for the java_poa_genie Genie*

Configuration Option	Purpose
default.java_poa_genie.want_antfile	Generate files used by the ant utility to build the test client and server.
default.java_poa_genie.want_tie	Use the POA tie approach instead of direct inheritance from POA skeleton classes.
default.java_poa_genie.want_inherit	If IDL interfaces inherit from each other, make the servants for those interfaces have corresponding inheritance.
default.java_poa_genie.want_default_poa	Override the _default_POA function on each servant.
default.java_poa_genie.want_throw	Generate a throws clause in operation signatures. Default is "true".
default.java_poa_genie.want_complete	Generate a complete client and complete servants.
default.java_poa_genie.want_threads	Use the multi-threaded POA policy.
default.java_poa_genie.want_ns	Use the naming service.

Command-Line Usage

This section summarizes the command-line arguments used by the genies that are bundled with the code generation toolkit.

stats

Synopsis

`idlgen stats.tcl [`*`options`*`] [`*`file`*`.idl]+`

Options

The command line options are:

-I<directory>	Passed to preprocessor. Specifies the directories to search for files included by the *file*.idl file.
-D<name>[=value]	Passed to preprocessor. Defines preprocessor variables that can be used by *file*.idl.
-h	Print a help message.
-include	Count statistics for files that are included by *file*.idl as well.

idl2html

Synopsis

`idlgen idl2html.tcl [`*`options`*`] [`*`file`*`.idl]+`

Options

The command line options are:

-I<directory>	Passed to preprocessor. Specifies the directories to search for files included by the *file*.idl file.

`-D<name>[=value]`	Passed to preprocessor. Defines preprocessor variables that can be used by *file*`.idl`.
`-dir <directory>`	Put generated files in the specified directory
`-h`	Print help message.
`-v`	Verbose mode (default).
`-s`	Silent mode.

Orbix C++ Genies

cpp_poa_genie

Synopsis `idlgen cpp_poa_genie.tcl [`*options*`]` *`file`*`.idl [`*interface wildcard*`]*`

Options

The command line options are as follows. The defaults are set by configuration file entries:

`-I<directory>`	Passed to preprocessor. Specifies the directories to search for files included by the *file*`.idl` file.
`-D<name>[=value]`	Passed to preprocessor. Defines preprocessor variables that can be used by *file*`.idl`.
`-h`	Print help message.
`-v`	Verbose mode (default).
`-s`	Silent mode.
`-dir <directory>`	Put generated files in the specified directory.
`-(no)include`	Process interfaces for files that are included by *`file`*`.idl` as well.
`-(no)servant`	Generate servant class that implements IDL interface.
`-(no)client`	Generate test client main program.
`-(no)server`	Generate test server main program.
`-(no)makefile`	Generate makefile to build test application.
`-all`	Shorthand for `-servant -client -server -makefile`.
`-(in)complete`	Generate complete applications (default no).
`-(no)ns`	Generate code that uses the naming service (default no).
`-(no)tie`	Use the tie approach (default no).
`-(no)inherit`	Servants follow IDL inheritance (default yes).
`-(no)refcount`	Servants are reference counted.
`-(no)threads`	Server is multi-threaded (default no). Implies `-refcount`.
`-strategy simple`	Create servants in main.
`-strategy activator`	Create on demand with `ServantActivator`.
`-strategy locator`	Create servants per-call with `ServantLocator`.
`-strategy dfltsrv`	Use a single default servant for many objects.
`-default_poa per_servant`	Each servant is associated with a POA.

-default_poa exception	_default_poa asserts or throws exception.

cpp_poa_op

Synopsis idlgen cpp_poa_op.tcl [*options*] *file*.idl [*operation or attribute wildcard*]*

Options The command line options are:

-I<directory>	Passed to preprocessor. Specifies the directories to search for files included by the *file*.idl file.
-D<name>[=value]	Passed to preprocessor. Defines preprocessor variables that can be used by *file*.idl.
-dir <directory>	Put generated files in the specified directory.
-h	Print help message.
-v	Verbose mode (default).
-s	Silent mode.
-o <file>	Write the output to the specified file.
-include	Process operations and attributes for files that are included by *file*.idl as well.
-(no)var	Use _var types in generated code (default).
-(in)complete	Generate bodies of operations and attributes (default).

Orbix Java Genies

java_poa_genie

Synopsis idlgen java_poa_genie.tcl [*options*] *file*.idl [*interface wildcard*]*

Options The command line options are as follows. The defaults are set by configuration file entries:

-jP <package_name>	Name of the package containing the generated Java source code prefixed to the IDL module name if provided.
-I<directory>	Passed to preprocessor. Specifies the directories to search for files included by the *file*.idl file.
-D<name>[=value]	Passed to preprocessor. Defines preprocessor variables that can be used by *file*.idl.
-h	Print help message.
-v	Verbose mode (default).
-s	Silent mode.
-(no)log	Log invocations to standard output.

-dir <directory>	Put generated files in the specified directory.
-include	Process interfaces for files that are included by *file*.idl as well.
-(no)servant	Generate servant class that implements IDL interface.
-(no)client	Generate test client main program.
-(no)server	Generate test server main program.
-(no)antfile	Generate files to build test application.
-all	Shorthand for -servant -client -server -antfile.
-(in)complete	Generate complete applications (default no).
-(no)ns	Generate code that uses the naming service (default no).
-(no)tie	Use the tie approach (default no).
-(no)inherit	Servants follow IDL inheritance (default yes).
-(no)threads	Server is multi-threaded (default no).
-strategy simple	Create servants in main.
-strategy activator	Create on demand with ServantActivator.
-strategy locator	Create servants per-call with ServantLocator.
-strategy dfltsrv	Use a single default servant for many objects.
-default_poa per_servant	Each servant is associated with a POA.
-default_poa exception	_default_poa asserts or throws exception.

Command Library Reference

This appendix presents reference material on all the commands that the code generation toolkit provides, in addition to the standard Tcl interpreter.

File Output API

Commands

The following commands provide support for file output:

- `std/output.tcl`: Normal output.
- `std/sbs_output.tcl`: Smart but slow output.

open_output_file

Synopsis `open_output_file` *`filename`*

Description Opens the specified file for writing.

Notes If the file already exists, it is overwritten.

Examples `open_output_file "my_code.cpp"`

See Also `close_output_file`, `output`

close_output_file

Synopsis `close_output_file`

Description Closes the currently open file.

Notes Throws an exception if there is no currently open file.

Examples `close_output_file`

See Also `close_output_file`, `flush_output`

output

Synopsis `output` *`string`*

Description Writes the specified string to the currently open file.

Notes Throw an exception if there is no currently open file.

Examples `output "Write a line to a file"`

See Also `close_output_file`, `open_output_file`

Configuration File API

This section lists and describes all the commands associated with configuration files. These commands are discussed in "Configuring Genies".

Configuration commands have the following general format:

`$cfg` *operation* [*arguments*]

The `$cfg` variable is a reference to a configuration object that can be used as a Tcl command. The first argument of the `$cfg` command, *operation*, specifies a particular action performed by the `$cfg` command.

A pseudo-code notation is used for the operation definitions of the `$cfg` configuration file variable:

```
class derived_node : base_node {
    return_type operation(param_type param_name)
}
```

idlgen_parse_config_file

Synopsis

`idlgen_parse_config_file` *filename*

Description

Parses the given configuration file. If parsing fails this command throws an exception, the text of which indicates the problem. If parsing is successful this command returns a handle to a Tcl object that is initialized with the contents of the specified configuration file. The pseudo-code representation of the resultant object is:

```
class configuration_file {
    enum setting_type {string, list, missing}

    string         filename()
    list<string>   list_names()
    void  destroy()  setting_type type(string cfg_name)
    string get_string(string cfg_name)
    void set_string(string cfg_name, string cfg_value)
    list<string> get_list(string cfg_name)
   void set_list(string cfg_name, list<string> cfg_value)
}
```

Examples

```
if {
    [catch {
        set my_cfg_file [idlgen_parse_config_file "mycfg.cfg"]
    } err]
} {
    puts stderr $err
    exit
}
```

See Also

`destroy`, `filename`

destroy

Synopsis

`$cfg destroy`

Description

Frees any memory taken up by the parsed configuration file.

Examples `$my_cfg_file destroy`

See Also `idlgen_parse_config_file`

filename

Synopsis `$cfg filename`

Description Returns the name of the configuration file that was parsed.

Examples $my_cfg_file filename

The preceding Tcl command returns:

```
mycfg.cfg
```

See Also `idlgen_parse_config_file`

list_names

Synopsis `$cfg list_names`

Description Returns a list that contains the names of all the entries in the parsed configuration file.

Notes No assumptions should be made about the order of names in the returned list.

Examples

```
puts "[$my_cfg_file filename] contains the following entries..."
foreach name [$my_cfg_file list_names] {
    puts "\t$name"
}
```

The preceding Tcl script generates the following output:

```
orbix.version
orbix.is_multithreaded
cpp.file_ext
```

See Also `filename`

type

Synopsis `$cfg type`

Description A configuration file entry can have a value that is either a string or a list of strings. This command is used to determine the type of the value associated with the name.

Notes If the specified name is not in the configuration file, this command returns `missing`.

Examples

```
switch [$my_cfg_file type "foo.bar"] {
    string     { puts "The 'foo.bar' entry is a string" }
    list       { puts "The 'foo.bar' entry is a list" }
    missing    { puts "There is no 'foo.bar' entry" }
}
```

See Also `list_names`

get_string

Synopsis `$cfg get_string` *`name [default_value]`*

Description Returns the value of the specified name. If there is no name entry, the default value (if supplied) is returned.

Notes An exception is thrown if any of the following errors occur:

- There is no entry for name and no default value is supplied.
- The entry for name exists but is of type `list`.

Examples

```
puts [$my_cfg get_string "foo_bar"]
```

The preceding Tcl script generates the following output:

```
my_value
```

See Also `get_list`, `set_string`

get_list

Synopsis `$cfg get_list` *`name [default_list]`*

Description Returns the list value of the specified name. If there is no name entry, the default list (if supplied) is returned.

Notes An exception is thrown if any of the following errors occur:

- There is no entry for name and no default list is supplied.
- The entry for name exists but is of type `string`.

Examples

```
foreach item [$my_cfg get_list my_list] { puts $item }
```

The preceding Tcl script generates the following output:

```
value1
value2
value3
```

See Also `get_string`, `set_list`

set_string

Synopsis `$cfg set_string` *`name value`*

Description Assigns *`value`* to the specified *`name`*.

Notes If the *name* entry already exists, it is overwritten. The updated configuration settings are not written back to the file.

Examples

```
$my_cfg set_string "foo.bar" "another_value"
```

See Also `get_string`

set_list

Synopsis `$cfg set_list` *`name value`*

Description Assigns *`value`* to the specified *`name`*.

Notes If the entry `name` already exists, it is overwritten. The updated configuration settings are not written back to the file.

Examples

```
$my_cfg set_list my_string ["this", "is", "a", "list"]
```

See Also `get_list`

idlgen_set_preferences

Synopsis `idlgen_set_preferences $cfg`

Description This procedure iterates over all the entries in the specified configuration file and, for each entry that exists in the `default` scope, it creates an entry in the `$pref` array. For example, the `$cfg` entry `default.foo.bar = "apples"` results in `$pref(foo,bar)` being set to `"apples"`.

Notes

This procedure assumes that all names in the configuration file that contain `is_` or `want_` have boolean values. If such an entry has a value other than `"true"` or `"false"`, an exception is thrown.

During initialization, the `idlgen` interpreter executes the following statement:

`idlgen_set_preferences $idlgen(cfg)`

As such, `default` scoped entries in the configuration file are always copied into the `$pref` array.

```
if {
    [catch {
      set my_cfg [idlgen_parse_config_file "mycfg.cfg"]
        idlgen_set_preferences $my_cfg
    } err]
} {
    puts stderr $err
    exit
}
```

See Also

`idlgen_parse_config_file`

Command Line Arguments API

This sections details commands that support command-line parsing. These commands are discussed in “Configuring Genies”.

idlgen_getarg

Synopsis

`idlgen_getarg` *`$format arg param symbol`*

Description

Extracts the command line arguments from `$argv` using a user-defined search data structure.

format (in)	A data structure that describes which command-line parameters you wish to extract.
arg (out)	The command-line argument that was matched on this run of the command.
param (out)	The parameter (if any) of the command-line argument that was matched.
symbol (out)	The symbol for the command-line argument that was specified in the format parameter. This can be used to find out which command-line argument was actually extracted.

Notes

Format must be of the following form:

```
set format {
    {"regular expression"   [ 0|1] symbol }
    ...
    ...
}
```

Examples

```
set cmd_line_args_format {
    { "-I.+"                0       -I       }
    { "-D.+"                0       -D       }
    { "-v"                  0       -v       }
    { "-h"                  0       usage    }
    { "-ext"                1       -ext     }
    { ".+\\.[iI][dD][lL]    0       idl_file }
}

while { $argc > 0 } {

    idlgen_getarg $cmd_line_args_format arg param symbol

    switch $symbol {
        -I      -
        -D      { puts  "Preprocessor directive: $arg"}
        idlfile { puts  "IDL file: $arg" }
        -v      { puts  "option: -v" }
        -ext    { puts  "option: -ext; parameter $param" }
        usage   { puts  "usage: ..."
                    exit 1
                }
        default { puts "unknown argument $arg"    }
        puts "usage: ...
        exit 1
    }
}
```

See Also

idlgen_parse_config_file

IDL Parser Reference

This appendix presents reference material on all the commands that the code generation toolkit provides to parse IDL files and manipulate the results.

idlgen_parse_idl_file

Synopsis

`idlgen_parse_idl_file` *`file preprocessor_directives`*

Description

Parses the specified IDL *`file`* with the specified preprocessor directives being passed to the preprocessor. The *`preprocessor_directives`* parameter is optional. Its default value is an empty list.

Notes

If parsing is successful, the root node of the parse tree is placed into the global variable `$idlgen(root)`, and `idlgen_parse_idl_file` returns `1` (`true`). If parsing fails, error messages are written to standard error and `idlgen_parse_idl_file` returns `0`.

Examples

```
# Tcl
if { [idlgen_parse_idl_file "bank.idl" {-DDEBUG}]}{
        puts "parsing succeeded"
} else {
        puts "parsing failed"
}
```

IDL Parse Tree Nodes

This section lists and describes all the possible node types that can be created from parsing an IDL file.

This section uses the following typographical conventions:

- A pseudo-code notation is used for the operation definitions of the different nodes that can exist in the parse tree:

```
class derived_node : base_node {
    return_type       operation(param_type param_name)
}
```

- In the examples given, the highlighted line in the IDL corresponds to the node used in the Tcl script. In this example, the module `Finance` is the node referred to in the Tcl script as the variable `$module`.

Figure 0.1:

<table>
<tr><th>IDL Module</th><th>Tcl Node</th></tr>
<tr><td><pre>module Finance {
    interface Account{
    ...
    };
};</pre></td><td><pre>puts [$module
   l_name]
> Finance</pre></td></tr>
</table>

Table of Node Types

Inheritance hierarchy

All the different types of nodes are arranged into an inheritance hierarchy as shown in Figure 5:

Types shown in bold define new operations. For example, type `field` inherits from type `node` and defines some new operations, while type `char` also inherits from `node` but does not define any additional operations. There are two abstract node types that do not represent any IDL constructs, but encapsulate the common features of certain types of node. These two abstract node types are called *node* and *scope*.

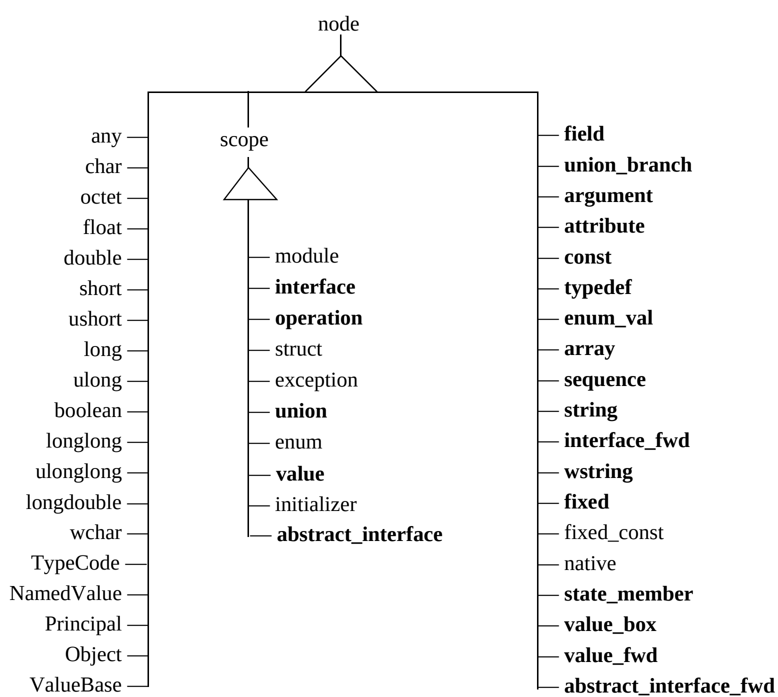

Figure 5: *Inheritance Hierarchy for Node Types*

node

Synopsis

```
class node {
    string          node_type()
    string          l_name()
    string          s_name()
    string          s_uname()
    list<string>    s_name_list()
    string          file()
    integer         line()
    boolean         is_in_main_file()
    node            defined_in()
    node            true_base_type()
```

```
    list<string>    pragma_list()
    boolean         is_imported()
}
```

Description

This is the abstract base type for all the nodes in the IDL parse tree. For example, the nodes `interface`, `module`, `attribute`, `long` are all sub-types of `node`.

Parameters

node_type	The name of parse-tree node's class.
l_name	Local name of the node for example, `balance`.
s_name	Fully scoped name of the node for example `account::balance`.
s_uname	Fully scoped name of the node, but with all occurrences of `"::"` replaced with an underscore for example, `account_balance`.
s_name_list	Fully scoped name of the node in list form.
defined_in	The node of the enclosing scope.
true_base_type	For almost all node types, this operation returns a handle to the node itself. However, for a typedef node, this operation strips off all the layers of typedef and returns a handle to the underlying type (see page 34).
file	IDL file that contained the node.
line	Line number in the IDL file where the construct was defined.
pragma_list	A list of the relevant pragmas in the IDL file.
is_in_main_file	True if not in an IDL file referred to in an `#include` statement.
is_imported	Opposite of `is_in_main_file`.

Examples

```
// IDL
module Finance {
        exception noFunds {
            string reason;
        };
};

# Tcl
puts [$node node_type]          > exception
puts [$node l_name]             > noFunds
puts [$node s_name]             > Finance::noFunds
puts [$node s_uname]            > Finance_noFunds
puts [$node s_name_list]        > Finance noFunds
set module [$node defined_in]
puts [$module l_name]           > Finance
```

scope

Synopsis

```
class scope : node {
    node        lookup(string name)
    list<node>  contents(
                    list<string> constructs_wanted,
                    function filter_func = "")
    list<node>  rcontents(
                    list<string> constructs_wanted,
                    list<string> recurse_nto,
                    function filter_func = "")
}
```

Description

Abstract base type for all the scoping constructs in the IDL file. An IDL construct is a `scope` if it can contain other IDL constructs. For example, a `module` is a `scope` because it can contain the declaration of other IDL types. Likewise, a `struct` is a `scope` because it contains the fields of the `struct`.

lookup

`lookup name`

Get a handle to the named node.

contents

`contents` *`node_types`* `[`*`func`*`]`

```
proc func { node } {
# return 1 if node is to be included
# return 0 if node is to be excluded
}
```

Obtain a list of handles to all the nodes that match the types in the `node_types` list. An optional function name `func` can be provided for extra filtering. This function must take one parameter, and returns either `1` or `0` (true or false). The parameter is the handle to a located node, the function can then return `1` if it wants that node in the results list or `0` if it is to be excluded.

rcontents

`rcontents` *`node_types scope_types [func]`*

The same as `contents` but also recursively traverses the contained scopes specified in the *`scope_types`* list. The pseudo-type `all` can be used as a value for the *`node_types`* and *`scope_types`* parameters of the `contents` and `rcontents` operations.

Examples

```
// IDL
module Finance {
    exception noFunds {
        string reason;
    };
    interface account {
```

```
        ...
    };
};
```

```
# Tcl
set exception [$finance lookup noFunds]
puts [$exception l_name]                              > noFunds

foreach node [$finance contents {all}] {
    puts [$node l_name]                               > noFunds
}                                                       account

foreach node [$finance rcontents {all} {exception}]{
    puts [$node l_name]
}                                                     > noFunds
                                                        reason
                                                        account
```

Built-In IDL types

All the built-in IDL types (`long`, `short`, `string`, and so on) are represented by types that inherit from `node` and do not define any additional operations.

```
class any        : node {}
class char       : node {}
class octet      : node {}
class float      : node {}
class double     : node {}
class short      : node {}
class ushort     : node {}
class long       : node {}
class boolean    : node {}
class longlong   : node {}
class ulonglong  : node {}
class longdouble : node {}
class wchar      : node {}
class TypeCode   : node {}
class NamedValue : node {}
class Principal  : node {}
class Object     : node {}
class ValueBase  : node {}
```

Notes

The `Principal` type is deprecated.

Examples

```
// IDL
interface bank {
    void findAccount( in long accNumber, inout branch brchObj );
};
```

```
# Tcl
puts [$long_type l_name]          > long
```

abstract_interface

Synopsis

```
class abstract_interface : scope {
    list<node> inherits()
    list<node> ancestors()
```

```
    list<node> acontents(
        list<string> constructs_wanted
        function filter_func = ""
    )
}
```

Description

An abstract interface.

Parameters

`inherits`	The list of abstract interfaces this one derives from.
`ancestors`	The list of abstract interfaces that are ancestors of this one, plus the abstract interface itself.
`acontents`	Like the normal `scope::contents` command but searches ancestor abstract interfaces as well.

Notes

An abstract interface is an ancestor of itself.

Examples

```
// IDL
module Finance {
    abstract interface AbsBank {
        ...
    };
};

# Tcl
puts [$interface l_name]          > bank
```

abstract_interface_fwd

Synopsis

```
class abstract_interface_fwd : node {
    node full_definition
}
```

Description

A forward declaration of an abstract interface.

Parameters

`full_definition`	The actual abstract interface.

Examples

```
// IDL
valuetype Account;

# Tcl
puts [$value node_type]          > value_fwd
puts [$value l_name]             > Account
```

argument

Synopsis

```
class argument : node {
    node    type()
    string  direction()
}
```

Description

An individual argument to an operation.

Parameters

`type`	The data type of the argument.
`direction`	The passing direction of the argument: `in`, `out` or `inout`.

Examples

```
// IDL
interface bank {
    void findAccount( in long accNumber, inout branch brchObj );
};
```

```
# Tcl
puts [$argument direction]       > in
set type [$argument type]
puts [$type l_name]              > long
puts [$argument l_name]          > accNumber
```

array

Synopsis

```
class array : node {
    node           elem_type()
    list<integer>  dims()
}
```

Description

Represents an array type. Array types are *anonymous*—they have no valid name. Arrays must be declared using `typedef` in order to have a name that can be referred to in IDL or C++ code. It is the corresponding `typedef` node that provides the array's name.

Parameters

`elem_type`	The data type of the array.
`dims`	The dimensions of the array.

Examples

```
// IDL
module Finance {
    typedef long longArray[10][20];
};
```

```
# Tcl
set array_typedef
   [$idlgen(root) lookup
   Finance::LongArray]
set array [$array_typedef base_type] Finance::LongArray
puts [$array_typedef s_name]         <anonymous-array>
puts [$array l_name]                 10 20
puts [$array dims]                   long
puts [[$array elem_type] s_name]
```

attribute

Synopsis

```
class attribute : node {
    boolean  is_readonly()
    node     type()
}
```

Description

An attribute.

Parameters

`is_readonly`	Determines whether or not the attribute is read only.
`type`	The type of the attribute.

Examples

```
// IDL
interface bank {
    attribute readonly string bankName;
};
```

```
# Tcl
puts [$attribute is_readonly]      > 1
set type [$attribute type]
puts [$type l_name]                > string
puts [$attribute l_name]           > bankName
```

const

Synopsis

```
class const : node {
    string  value()
    node    type()
}
```

Description A const.

Parameters

`value`	The value of the constant.
`type`	The data type of the constant.

Examples

```
// IDL
module Finance {
    const long bankNumber= 57;
};
```

```
# Tcl
puts [$const value]                > 57
set type [$const type]
puts [$type l_name]                > long
puts [$const l_name]               > bankNumber
```

enum_val

Synopsis

```
class enum_val : node {
    string  value()
    string  type()
}
```

Description A single entry in an enumeration.

Parameters

`value`	The value of the enumerated entry.
`type`	A name given to the whole enumeration.

Examples

```
// IDL
```

```
enum colour {red, green, blue};

# Tcl
puts [$enum_val value]            > 2
puts [$enum_val l_name]           > blue
puts [[$enum_val type] l_name]    > colour
```

enum

Synopsis

```
class enum : scope {
}
```

Description

The enumeration.

Examples

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

# Tcl
puts [$enum s_name]               > colour
```

exception

Synopsis

```
class exception : scope {
}
```

Description

An exception.

Examples

```
// IDL
module Finance{
    exception noFunds {
        string  reason;
        float   amountExceeded;
    };
};

# Tcl
puts [$exception l_name]          > noFunds
```

field

Synopsis

```
class field : node {
    node  type()
}
```

Description

An item inside an exception or structure.

Parameters

`type` The type of the field.

Examples

```
// IDL
struct cardNumber {
    long binNumber;
    long accountNumber;
};

# Tcl
set type [$field type]
puts [$type l_name]               > long
puts [$field l_name]              > binNumber
```

fixed

Synopsis

```
class fixed : node {
    integer digits()
    integer scale()
}
```

Description

A fixed type.

Parameters

`digits`	The number of significant digits (a positive integer).
`scale`	The number of places the decimal point is shifted to the left (positive integer) or to the right (negative integer).

Examples

```
// IDL
struct accountDetails {
    fixed< 20, 2 > balance;
};
```

```
# Tcl
set type [$field type]
puts [$type digits]              > 20
puts [$type scale]               > 2
```

fixed_const

Synopsis

```
class fixed_const : node {
}
```

Description

A fixed type constant.

Notes

A const fixed type does not have explicit `digits` and `scale` parameters.

Examples

```
// IDL
const fixed = "45.678";
```

```
# Tcl
set type [$const type]
puts [$type node_type]           > fixed_const
puts [$type l_name]              > fixed
puts [$const value]              > 45.678
```

initializer

Synopsis

```
class initializer : scope {
}
```

Description

An `initializer` node represents an initializer member of a stateful value type.

Notes

The `initializer` scope contains `argument` nodes.

Examples

```
// IDL
valuetype Account {
    factory init(in fixed< 20, 2> openingBalance);
```

```
    public fixed< 20, 2> balance;
};
```

```
# Tcl
puts [$member node_type]          > initializer
puts [$member l_name]             > init
```

interface

Synopsis

```
class interface : scope {
    list<node>  inherits()
    list<node>  ancestors()
    list<node>  acontents(
                    list<string> constructs_wanted
                    function filter_func = "" )
    boolean  is_local()
}
```

Description

An interface.

Parameters

inherits	The list of interfaces this one derives from.
ancestors	The list of all interfaces that are ancestors of this one, plus the interface itself.
acontents	Like the normal scope::contents command, but searches ancestor interfaces as well.
is_local	True if the interface was declared local in the IDL.

Notes

An interface is an ancestor of itself.

Examples

```
// IDL
module Finance {
    interface bank {
        ...
    };
};
```

```
# Tcl
puts [$interface l_name]          > bank
```

interface_fwd

Synopsis

```
class interface_fwd : node {
    node  full_definition()
}
```

Description

A forward declaration of an interface.

Parameters

full_definition	The actual interface.

Examples

```
// IDL
interface bank;
...
interface bank {
```

```
    account findAccount( in string accountNumber );
};
```

```
# Tcl
set interface [$interface_fwd
   full_definition]
set operation [$interface lookup           > findAccount
   "findAccount"]
puts [ $operation l_name ]
```

module

Synopsis

```
class module : scope {
}
```

Description

A module.

Examples

```
// IDL
module Finance {
    interface bank {
        ...
    };
};
```

```
# Tcl
puts [$module l_name]            > Finance
```

native

Synopsis

```
class native : node {
}
```

Description

A native type.

Notes

Used to represent language specific data types in IDL (particularly in pseudo-IDL). It does not normally occur in user-defined IDL.

Examples

```
// IDL
module PortableServer {
    native Servant;
    ...
};
```

```
# Tcl
puts [$type node_type]           > native
puts [$type l_name]              > Servant
```

operation

Synopsis

```
class operation : scope{
    node          return_type()
    boolean       is_oneway()
    list<node>    raises_list()
    list<string>  context_list()
    list<node>    args(list<string> dir_list,
                       function filter_func = "")
}
```

Description

An interface operation.

Parameters

`return_type`	The return type of the operation.
`is_oneway`	Determines whether or not the operation is a `oneway`.
`raises_list`	A list of handles to the exceptions that can be raised.
`context_list`	A list of the context strings.
`args`	The `operation` class is a subtype of `scope`, and hence it inherits the `contents` operation. Invoking `contents` on an operation returns a list of all the argument nodes contained in the operation. Sometimes you may want to get back a list of only the arguments that are passed in a particular direction. The `args` operation allows you to specify a list of directions for which you want to inspect the arguments. For example, specifying `{in inout}` for the `dir_list` parameter causes args to return a list of all the `in` and `inout` arguments.

Examples

```
// IDL
interface bank
{
    long newAccount( in string accountName )
        raises( duplicate, blacklisted ) context( "branch" );
};


# Tcl
set type [$operation return_type]
puts [$type l_name]                 > long
puts [$operation is_oneway]         > 0
puts [$operation l_name]            > newAccount
puts [$operation context_list]      > branch
```

sequence

Synopsis

```
class sequence : node {
    node     elem_type()
    integer  max_size()
}
```

Description

An anonymous sequence.

`elem_type`	The type of the sequence.
`max_size`	The maximum size, if the sequence is bounded. Otherwise the value is 0.

Examples

```
// IDL
module Finance {
    typedef sequence<long, 10> longSeq;
```

```
};

# Tcl
set typedef [$idlgen(root) lookup
   "Finance::longSeq"]
set seq [$typedef base_type]
set elem_type [$seq elem_type]
puts [$elem_type l_name]          > long
puts [$typedef l_name]            > longSeq
puts [$seq max_size]              > 10
puts [$seq l_name]                > <anonymous_sequence>
```

state_member

Synopsis

```
class state_member : node {
    string protection()
    node   type()
}
```

Description

A `state_member` node represents a data member of a stateful value type.

`protection`	Either `"public"` or `"private"`.
`type`	The type of the state member.

Examples

```
// IDL
valuetype Account {
    public fixed< 20, 2> balance;
};

# Tcl
puts [$member node_type]          > state_member
puts [$member protection]         > public
puts [$member l_name]             > balance
```

string

Synopsis

```
class string : node {
    integer  max_size()
}
```

Description

A bounded or unbounded string data type.

`max_size`	The maximum size, if the string is bounded. Otherwise the value is `0`.

Examples

```
// IDL
struct branchDetails {
    string<100> branchName;
};


# Tcl
set type [$field type]
puts [$field l_name]              > branchName
puts [$type max_size]             > 100
puts [$type l_name]               > string
```

struct

Synopsis

```
class struct : scope {
}
```

Description

A structure.

Notes

The `struct` scope contains `field` nodes.

Examples

```
// IDL
module Finance {
    struct branchCode
    {
        string  cateogory;
        long    zoneCode;
    };
};
```

```
# Tcl
puts [$structure s_name]          > Finance::branchCode
```

typedef

Synopsis

```
class typedef : node {
    node  base_type()
}
```

Description

A type definition.

`base_type` The data type of the typedef.

Examples

```
// IDL
module Finance{
    typedef sequence<account, 100> bankAccounts;
};
```

```
# Tcl
set $sequence [$typedef
   base_type]                     > 100
puts [$sequence max_size]         > bankAccounts
puts [$typedef l_name]
```

union

Synopsis

```
class union : scope {
    node  disc_type()
}
```

Description

A union.

`disc_type` The data type of the discriminant.

Examples

```
// IDL
union accountType switch( long ) {
    case 1:   string   accountName;
    case 2:   long     accountNumber;
    default:  account  accountObj;
```

```
};

# Tcl
puts [$union l_name]              > accountType
set type [$union disc_type]
puts [$type l_name]               > long
```

union_branch

Synopsis

```
class union_branch : node {
    string  l_label()
    string  s_label()
    string  s_label_list()
    string  type()
}
```

Description

A single branch in a union.

`l_label`	The case label.
`s_label`	The scoped case label.
`s_label_list`	The scoped label in list form.
`type`	The data type of the branch.

Examples

```
// IDL
module Finance {
    union accountType switch( long ) {
        case 1:   string   accountName;
        case 2:   long     accountNumber;
        default:  account  accountObj;
    };
};

# Tcl
set type [$union_branch    type]
puts [$type l_name]               > long
puts [$union_branch l_name]       > accountNumber
puts [$union_branch l_label]      > 2
puts [$union_branch s_label]      > 2
```

value

Synopsis

```
class value : scope {
    list<node> inherits()
    list<node> supports()
    list<node> ancestors()

    list<node> acontents(
        list<string> constructs_wanted
        function filter_func = ""
    );

    boolean is_custom()
    boolean is_truncatable()
    boolean is_abstract()
}
```

Description

A value type.

inherits	The list of value types this one derives from.
supports	The list of interfaces supported by this value.
ancestors	The list of all interfaces and value types that are ancestors of this one, plus the value type itself.
acontents	Like the normal scope::contents command, but searches ancestor interfaces and value types as well.
is_custom	True, if the value is declared as custom.
is_truncatable	True, if this value inherits from a stateful value and the inheritance is specified to be truncatable.
is_abstract	True, if this value is declared abstract.

Notes

The value type is an ancestor of itself. The value scope contains initializer, state_member and operation nodes.

Examples

```
// IDL
valuetype Account { };

# Tcl
puts [$value node_type]          > value
puts [$value l_name]             > Account
```

value_box

Synopsis

```
class value_box : node {
    node base_type()
}
```

Description

A value box type.

base_type	The boxed type.

Examples

```
// IDL
valuetype StringBox string;

# Tcl
puts [$value node_type]          > value_box
puts [$value l_name]             > StringBox
```

value_fwd

Synopsis

```
class value_fwd : node {
    node full_definition
}
```

Description

A forward declaration of a value type.

full_definition	The actual value type.

Examples

```
// IDL
```

```
valuetype Account;
```

```
# Tcl
puts [$value node_type]           > value_fwd
puts [$value l_name]              > Account
```

wstring

Synopsis

```
class wstring : node {
    integer max_size()
}
```

Description

A wide string (international string).

`max_size` The maximum size, if the wide string is bounded. Otherwise the value is `0`.

Examples

```
// IDL
struct customer {
    wstring<100> customerName;
};
```

```
# Tcl
set type [$field type]
puts [$field l_name]              > customerName
puts [$type max_size]             > 100
puts [$type l_name]               > wstring
```

Configuration File Grammar

This appendix summarizes the syntax of the configuration file used with the code generation toolkit.

```
config_file      = [ statement ]*

statement        = named_scope ';'
                   | assign_statement ';'

named_scope      = identifier '{' [ statement ]* '}'

assign_statement = identifier '=' string_expr
                   | identifier '=' array_expr

string_expr      = string [ '+' string ]*
array_expr       = array  [ '+' array ]*
string           = "..." | identifier

array          = '[' string_expr [ ',' string_expr ]* ']'
                   |    identifier

identifier       = [ [a-z] | [A-Z] | [0-9] | '_' | '-' |
   ':'
                   | '.' ]*
```

Comments start with # and extend to the end of the line.

Glossary

C

CFR

See configuration repository.

client

An application (process) that typically runs on a desktop and requests services from other applications that often run on different machines (known as server processes). In CORBA, a client is a program that requests services from CORBA objects.

configuration

A specific arrangement of system elements and settings.

configuration domain

Contains all the configuration information that Orbix ORBs, services and applications use. Defines a set of common configuration settings that specify available services and control ORB behavior. This information consists of configuration variables and their values. Configuration domain data can be implemented and maintained in a centralised Orbix configuration repository or as a set of files distributed among domain hosts. Configuration domains let you organise ORBs into manageable groups, thereby bringing scalability and ease of use to the largest environments. See also configuration file and configuration repository.

configuration file

A file that contains configuration information for Orbix components within a specific configuration domain. See also configuration domain.

configuration repository

A centralised store of configuration information for all Orbix components within a specific configuration domain. See also configuration domain.

configuration scope

Orbix configuration is divided into scopes. These are typically organized into a root scope and a hierarchy of nested scopes, the fully-qualified names of which map directly to ORB names. By organising configuration properties into various scopes, different settings can be provided for individual ORBs, or common settings for groups of ORB. Orbix services, such as the naming service, have their own configuration scopes.

CORBA

Common Object Request Broker Architecture. An open standard that enables objects to communicate with one another regardless of what programming language they are written in, or what operating system they run on. The CORBA specification is produced and maintained by the OMG. See also OMG.

CORBA naming service

An implementation of the OMG Naming Service Specification. Describes how applications can map object references to names. Servers can register object references by name with a naming service repository, and can advertise those names to clients. Clients, in turn, can resolve the desired objects in the naming service by supplying the appropriate name. The Orbix naming service is an example.

CORBA objects

Self-contained software entities that consist of both data and the procedures to manipulate that data. Can be implemented in any programming language that CORBA supports, such as C++ and Java.

D

deployment

The process of distributing a configuration or system element into an environment.

I

IDL

Interface Definition Language. The CORBA standard declarative language that allows a programmer to define interfaces to CORBA objects. An IDL file defines the public API that CORBA objects expose in a server application. Clients use these interfaces to access server objects across a network. IDL interfaces are independent of operating systems and programming languages.

IIOP

Internet Inter-ORB Protocol. The CORBA standard messaging protocol, defined by the OMG, for communications between ORBs and distributed applications. IIOP is defined as a protocol layer above the transport layer, TCP/IP.

installation

The placement of software on a computer. Installation does not include configuration unless a default configuration is supplied.

Interface Definition Language

See IDL.

invocation

A request issued on an already active software component.

IOR

Interoperable Object Reference. See object reference.

N

naming service

See CORBA naming service.

O

object reference

Uniquely identifies a local or remote object instance. Can be stored in a CORBA naming service, in a file or in a URL. The contact details that a client application uses to communicate with a CORBA object. Also known as interoperable object reference (IOR) or proxy.

OMG

Object Management Group. An open membership, not-for-profit consortium that produces and maintains computer industry specifications for interoperable enterprise applications, including CORBA. See www.omg.com.

ORB

Object Request Broker. Manages the interaction between clients and servers, using the Internet Inter-ORB Protocol (IIOP). Enables clients to make requests and receive replies from servers in a distributed computer environment. Key component in CORBA.

P

POA

Portable Object Adapter. Maps object references to their concrete implementations in a server. Creates and manages object references to all objects used by an application, manages object state, and provides the infrastructure to support persistent objects and the portability of object implementations between different ORB products. Can be transient or persistent.

protocol

Format for the layout of messages sent over a network.

S

server

A program that provides services to clients. CORBA servers act as containers for CORBA objects, allowing clients to access those objects using IDL interfaces.

T

Tcl

Tool Command Language. A scripting language and an interpreter for that language.

TCP/IP

Transmission Control Protocol/Internet Protocol. The basic suite of protocols used to connect hosts to the Internet, intranets, and extranets.

Index

B

C

D

E

F

G

H

I

J

K

L

O

P

R

S

T

U

V

W