Orbix
Code Generation Toolkit
Programmer's Guide

IONA Technologies PLC
September 2000

M 2 4 6 7

The Orbix Code Generation contains the language Tcl which is used subject to the following license:

This software is copyrighted by the Regents of the University of California, Sun Microsystems, Inc., and other parties.

The following terms apply to all files associated with the software unless explicitly disclaimed in individual files.

The authors hereby grant permission to use, copy, modify, distribute, and license this software and its documentation for any purpose, provided that existing copyright notices are retained in all copies and that this notice is included verbatim in any distributions. No written agreement, license, or royalty fee is required for any of the authorized uses. Modifications to this software may be copyrighted by their authors and need not follow the licensing terms described here, provided that the new terms are clearly indicated on the first page of each file where they apply.

IN NO EVENT SHALL THE AUTHORS OR DISTRIBUTORS BE LIABLE TO ANY PARTY FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RISING OUT OF THE USE OF THIS SOFTWARE, ITS DOCUMENTATION, OR ANY DERIVATIVES THEREOF, EVEN IF THE AUTHORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

THE AUTHORS AND DISTRIBUTORS SPECIFICALLY DISCLAIM ANY WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND NON-INFRINGEMENT. THIS SOFTWARE IS PROVIDED ON AN "AS IS" BASIS, AND THE AUTHORS AND DISTRIBUTORS HAVE NO OBLIGATION TO PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS.

GOVERNMENT USE: If you are acquiring this software on behalf of the U.S. government, the Government shall have only "Restricted Rights" in the software and related documentation as defined in the Federal Acquisition Regulations (FARs) in Clause 52.227.19 (c) (2). If you are acquiring the software on behalf of the Department of Defense, the software shall be classified as "Commercial Computer Software" and the Government shall have only "Restricted Rights" as defined in Clause 252.227-7013 (c) (1) of DFARs. Notwithstanding the foregoing, the authors grant the U.S. Government and others acting in its behalf permission to use and distribute the software in accordance with the terms specified in this license.

Contents

Part II Developing Genies

Preface

The Orbix Code Generation Toolkit is a flexible development tool that increases programmer productivity by automating repetitive coding tasks. It is aimed at both novice and expert users of Orbix, IONA Technologies' implementation of the Object Management Group's (OMG) Common Object Request Broker Architecture (CORBA).

The Orbix Code Generation Toolkit contains an IDL parser, `idlgen`, and ready-made applications called *genies* that allow you to generate Java or C++ code from CORBA IDL files automatically. The Toolkit also contains command libraries that you can use to develop your own genies.

Audience

There are two intended audiences for this book: *genie users* and *genie developers*.

A genie user is a developer of Orbix applications, who uses the bundled genies to accelerate development. Part I of this book is addressed at this audience. Genie users need to be familiar with the OMG Interface Definition Language (IDL), and the C++ or Java language.

A genie developer customizes the bundled genies or develops completely new genies to perform specialized tasks. Part II of this book is addressed at this audience. Genie developers need to be familiar with the OMG 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

This section of the guide is a user's guide to the Orbix Code Generation Toolkit. It provides an overview of the product and describes its constituent components. It describes how to run the demonstration genies and the ready-to-run genies that produce C++ and Java starting point code.

Part II Developing Genies

This section of the guide takes an in-depth look at the Orbix Code Generation Toolkit and describes how to develop your own genies that are tailored to specific needs.

Part III C++ Genies Library Reference

This section of the guide is a reference to the commands that you use to produce C++ code from OMG IDL files.

Part IV Java Genies Library Reference

This section of the guide is a reference to the commands that you use to produce Java code from OMG IDL files.

Appendices

The appendices provide reference material on configuration options, command libraries, the IDL parser and configuration file grammar.

Document 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 <iostream.h>`
`Constant width (bold)`	Constant width (courier font) in bold text represents either command-line input or portions of code from Tcl bilingual files. See "Embedding Text Using Bilingual Files" on page 87.
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`*

This guide may use the following keying conventions:

No prompt	When a command's format is the same for multiple platforms, no prompt is 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, Windows NT, or Windows 98 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.

Note that the examples in this guide include file names in UNIX format. However, unless otherwise stated, all examples in this guide apply to the Orbix Code Generation Toolkit on both UNIX and Windows platforms.

Part I

Using the Toolkit

1

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

As shown in Figure 1.2, 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.

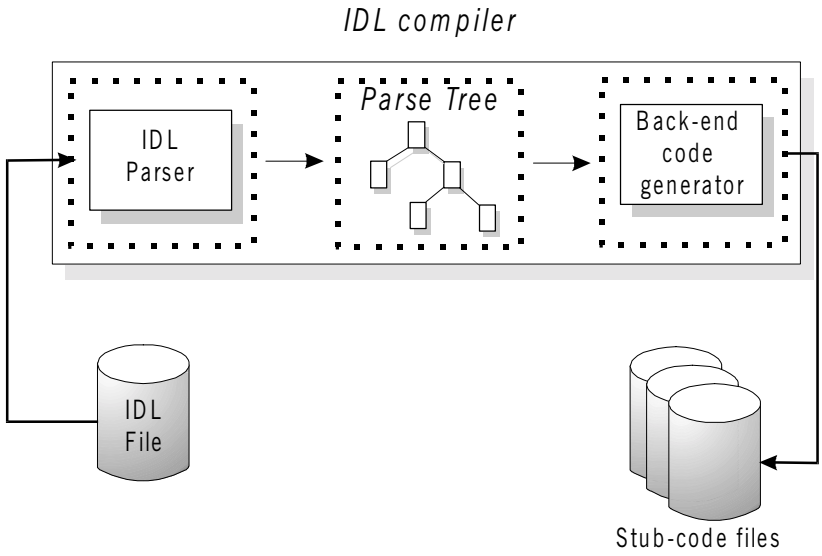

Figure 1.1: *Standard IDL Compiler Components*

Code Generation Toolkit Architecture

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 1.2, 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.

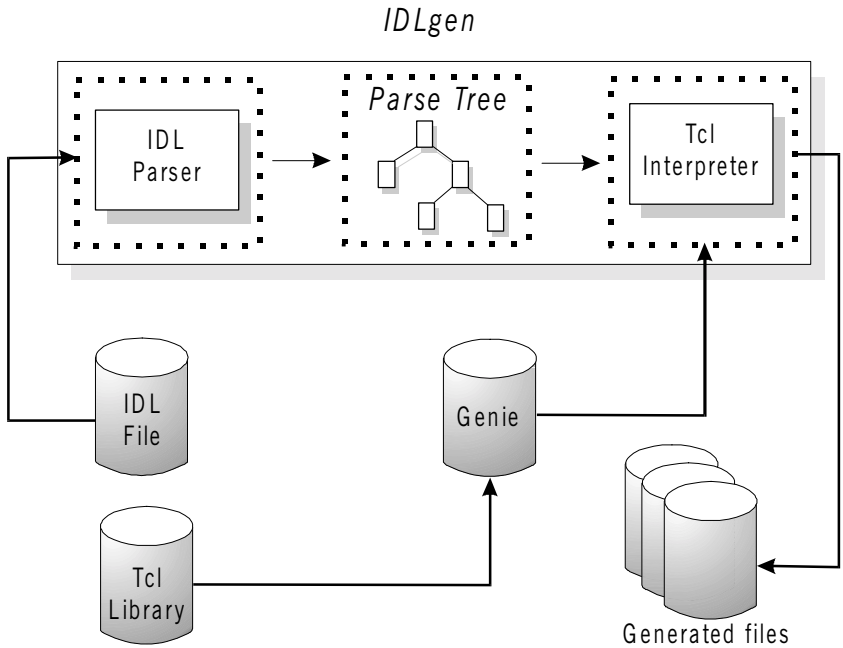

Figure 1.2: *Code Generation Toolkit Components*

Orbix Code Generation Toolkit Components

The Orbix Code Generation Toolkit consists of three components:

1. The `idlgen` executable: this is the engine at the heart of the Code Generation Toolkit.
2. A number of pre-written genies: these genies generate useful starting point code to help developers of Orbix applications.
3. Libraries of Tcl procedures: these libraries help users who want to write their own genies. For example, there is a library which maps IDL constructs into their C++ equivalents.

The Bundled Applications

Orbix Code Generation Toolkit comes with a number of bundled genies that can be fed into Orbix Code Generation Toolkit to accomplish a number of different tasks. The genies are also provided in source code form and so can be used as reference material when writing your own genies. The full details of these genies are discussed in Chapter 3, "Ready-to-Use Genies for Orbix C++ Edition".

Note: The bundled genies can be used straight away. A genie user does not need to know anything about Tcl or programming in Tcl.

Approaches to Using the Toolkit

The Code Generation Toolkit is a powerful addition to the CORBA developer's toolkit. However it is not essential to master all the available features of the Toolkit to make good use of it. As a starting point, it is a good idea to get to know the capabilities of the bundled applications and decide whether or not these can provide all that you want. If they cannot it is straightforward to extend or write new genies that meet the exact requirements of a task.

The first two parts of this guide are addressed at different groups of users:

- Part I—A genie user's guide covering installation, configuration, and a full description of the bundled applications.
- Part II—A genie developer's guide describing how to write new genies.

You need to read the second part only if you wish to extend the bundled genies or write new ones.

Known Limitations of Code Generation Toolkit

The IDL parser within IDLgen has some known limitations which will be addressed in a future release:

- It does not support the re-opening of modules.
- It does not support the following types: `long long`, `unsigned long long`, `wchar`, `wstring` and `fixed`.
- It allows only one case label per union branch. For example, the following is not allowed inside a union:

  ```
  case 1: case 2: long a;
  ```

IDLgen does support opaque types.

Finally, the IDL specification permits the use of anonymous sequences and arrays in some circumstances. For example, the following is legal IDL:

```
struct tree {
        long data;
1       sequence<tree>  children;
};
```

```
2     typedef sequence< sequence<long> > longSeqSeq;

      struct foo {
3            long bar[10];
      };
```

The `tree` struct requires the use of an anonymous sequence 1 in order to define a recursive type.

IDLgen provides full support for the use of anonymous sequences used in recursive types. However, IDLgen does not provide full support for unnecessary uses of anonymous types such as 2 or 3. IDLgen scripts can generate bad code for such uses of unnecessary anonymous types. As such, we recommend that you rewrite your IDL files to remove unnecessary anonymous types. For example, the examples of anonymous types 2 and 3 above could be rewritten as follows:

```
typedef sequence<long> longSeq;
typedef longSeq longSeqSeq;

typedef long longArray[10];
struct foo {
   longArray       bar;
};
```

2

Running the Demonstration Genies

A number of ready-to-run genies are bundled with the Code Generation Toolkit. This chapter describes these example genies.

The Orbix Code Generation Toolkit comes with a collection of genies that can accomplish a number of different tasks. This chapter discusses how these genies work with `idlgen`:

- How to run a genie.
- What genies are supplied.
- A description of the demonstration genies.

Running Genies

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

`idlgen` ***`name-of-genie args-to-genie`***

For example, one of the demo genies converts IDL files to HTML. This genie is held in the file `idl2html.tcl`. You can run it as follows:

```
idlgen idl2html.tcl bank.idl shop.idl acme.idl
idlgen: creating bank.html
idlgen: creating shop.html
idlgen: creating acme.html
```

Specifying the Application Location

The idlgen executable locates the specified genie file by searching a list of directories. The list of these directories is defined in the standard configuration file idlgen.cfg under the setting idlgen.genie_search_path. The default setting for this is:

```
idlgen.genie_search_path = [
      "."
      ,"./genie"
      ,install_root + "/genies"
      ,install_root + "/demo_genies"
];
```

This default setting is to search:

1. The current directory.
2. The genies directory under the current directory.
3. The genies directory under the toolkit installation directory.
4. The demos directory under the toolkit installation directory.

The order of these directories in the list is the order in which idlgen searches for the genie.

Note: You can alter this configuration setting to add additional directories. For instance, if you write your own genies you could place them into a separate directory and add this directory to idlgen.genie_search_path.

Looking For Applications

The `idlgen` executable provides a command-line option that lists all of the available genies, in all of the directories that are specified in the search path:

```
idlgen -list

available genies are...
cpp_genie.tcl    cpp_random.tcl       java_print.tcl
cpp_op.tcl       idl2html.tcl         java_random.tcl
cpp_print.tcl    java_genie.tcl       stats.tcl
```

You can pass a filter string to the `-list` option. For example, to show all the genies whose names contain the cpp string, enter the following command:

```
idlgen -list cpp

matching genies are...
cpp_genie.tcl   cpp_op.tcl      cpp_print.tcl   cpp_random.tcl
```

Common Command-Line Arguments

The bundled genies have some common command-line arguments. The simplest one is the help command-line argument `-h`:

```
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
```

There are also command-line arguments for passing information onto the IDL preprocessor.

- `-I` The include path for preprocessor.
- `-D` Any additional preprocessor symbols to define.

For example:

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

or:

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

You may have to place quote marks around the parameters to these command-line arguments if they contain white space:

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

The final couple of common command-line arguments determine whether or not the genies run in *verbose* or *silent* mode.

Running in verbose mode causes `idlgen` to tell you what it is doing:

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

```
idlgen: creating bank.htm
```

Running in silent mode suppresses the output:

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

If neither of these command-line settings are specified the default setting is determined by the `default.all.want_diagnostics` value in the `idlgen.cfg` configuration file. If this is set to `yes`, `idlgen` defaults to verbose mode. If this is set to `no`, `idlgen` defaults to silent mode.

What are the Bundled Genies?

The genies bundled with the Orbix Code Generation Toolkit can be grouped into a number of categories:

Demonstration Genies

`stats.tcl`	Provides statistical analysis of an IDL file's content.
`idl2html.tcl`	Converts IDL files into HTML files.

Orbix C++ Specific Genies

`cpp_genie.tcl`	Generates C++ code from an IDL file.
`cpp_op.tcl`	Generates C++ code for new operations from an IDL interface.
`cpp_random.tcl`	Creates a number of C++ functions that generate random values for all the types present in an IDL file.
`cpp_print.tcl`	Creates a number of C++ functions that can display all the data types present in an IDL file.
`cpp_equal.tcl`	Creates utility functions that test IDL types for equality.

Orbix Java Specific Genies

`java_genie.tcl`	Generates Java code from an IDL file.
`java_random.tcl`	Creates a number of Java methods that generate random values for all the types present in an IDL file.
`java_print.tcl`	Creates a number of Java methods that can display all the data types present in an IDL file.

This chapter describes the demo genies. Chapter 3, "Ready-to-Use Genies for Orbix C++ Edition" discusses the Orbix C++ specific genies.
Chapter 4"Ready-to-Use Genies for Orbix Java Edition"discusses the Orbix Java specific genies.

For a full genie user's reference, please refer to Appendix A on page 387. This describes the configuration and command-line options that are available.

Demonstration Genies

Two demonstration genies are shipped with the Orbix Code Generation Toolkit:

- `stats.tcl`
- `idl2html.tcl`

stats.tcl

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

For example:

```
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
```

The statistics genie, by default, only processes the constructs it finds in the IDL file specified. It does not take into consideration any IDL files that are referred to with `#include` statements. You can use the `-include` command-line option to process, recursively, all such IDL files as well. For example, the IDL file `bank.idl` includes the IDL file `account.idl`:

```
// IDL
#include "account.idl"

interface Bank
{
   ...
};
```

You can gain statistics from both `account.idl` and `bank.idl` files together with this command:

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

This genie serve two purposes:

- This genie provides objective information which can be used to help estimate the time it will take to implement some task based on the IDL.
- The implementation of this genie provides a useful demonstration of how to write genies that process IDL files.

idl2html.tcl

This genie takes an IDL file and converts it to an equivalent HTML file.

Consider this simple extract from an IDL file:

```
// 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 by running it through idlgen:

```
idlgen idl2html.tcl bank.idl
```

```
idlgen: creating bank.html
```

This is the resultant HTML file, when viewed in an appropriate HTML browser:

```
// 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
```

The underlined words are the hypertext links that, when selected, move you to the definition of the specified type. For example, clicking on account makes the definition for the account interface appear in the browser's window.

There is one configuration setting in the standard configuration file for this genie:

default.html.file_ext	File extension preferred by your web browser. This is usually .html.

3

Ready-to-Use Genies for Orbix C++ Edition

The Orbix Code Generation Toolkit is packaged with several genies for use with the Orbix C++ product. This chapter explains what these genies are and how to use them effectively.

Using the C++ Genie to Kickstart New Projects

Many people start a new project by copying some code from an existing project and then editing this code to change the names of variables, signatures of operations, and so on. This is boring and time-consuming work. The C++ genie (`cpp_genie.tcl`) is a powerful utility that eliminates this task. If you have an IDL file that defines the interfaces for your new project, the C++ genie can generate a demonstration, client-server application that contains all the starting point code that you are likely to need for your project.

Generating a Client-Server Application

You can use the C++ genie to generate a complete client-server application. It produces a `makefile` and a complete set of compilable code for both a client and a server for the specified interfaces. For example:

```
idlgen cpp_genie.tcl -all finance.idl
```

```
finance.idl:
idlgen: creating account_i.h
idlgen: creating account_i.cxx
idlgen: creating bank_i.h
idlgen: creating bank_i.cxx
idlgen: creating smart_account.h
idlgen: creating smart_account.cxx
idlgen: creating smart_bank.h
idlgen: creating smart_bank.cxx
idlgen: creating loader.h
idlgen: creating loader.cxx
idlgen: creating server.cxx
idlgen: creating client.cxx
idlgen: creating call_funcs.h
idlgen: creating call_funcs.cxx
idlgen: creating it_print_funcs.h
idlgen: creating it_print_funcs.cxx
idlgen: creating it_random_funcs.h
idlgen: creating it_random_funcs.cxx
idlgen: creating Makefile
idlgen: creating Makefile.inc
```

The generated client application calls every operation in the server application and passes random values as parameters to the operations and attribute get/set methods. The server application then passes random values back in the `inout`, `out`, and `return` values of the operations.

Choosing an Object Reference Distribution Method

To establish initial contact between a client and a server application, the server has to distribute initial object references to its clients. The `cpp_genie.tcl` genie lets you select the object reference distribution method using a command-line option. You can choose between three mutually exclusive methods of object distribution, as described in Table 3.1.

Command-Line Option	Description
`-file`	(Default) Generate server code that distributes object references by writing stringified object references to files. Generate client code that reads the stringified object references from the server-created files.
`-ns`	Generate server code that distributes object references by creating object bindings in the naming service. Generate client code that reads the server-created bindings by resolving the object names.
`-bind`	(Deprecated) Generate client code that creates object references, based on the arguments passed to `_bind()`. The generated server performs no special steps.

Table: 3.1: *Object Distribution Methods*

Compiling and Running the Application

The `Makefile` generated by the Orbix Code Generation Toolkit has a complete set of rules for building both the client and server applications. To build the client and server:

1. Compile the generated application. At a command prompt, enter the following commands:

   Windows

   ```
   > nmake
   ```

   UNIX

   ```
   % make
   ```

2. Run the Orbix daemon. Open a new MS-DOS prompt or `xterm` window (UNIX) and enter the following command:

   Windows

   ```
   > orbixd
   ```

   UNIX

   ```
   % orbixd
   ```

   The Orbix daemon runs in the foreground and logs its activities to this window.

3. Register the server with the Orbix daemon. At a command prompt, enter the following command:

   Windows

   ```
   > nmake putit
   ```

   UNIX

   ```
   % make putit
   ```

4. Run the server. At a command prompt, enter the following command:

   ```
   server
   ```

5. Run the client.

   If you have generated the client code using either the `-file` or `-ns` option, open a new MS-DOS prompt or `xterm` window (UNIX) and enter the following command:

   ```
   client
   ```

If you have generated the client code using the `-bind` option, open a new MS-DOS prompt or `xterm` window (UNIX) and enter the following command:

`client` *ServerHostName*

where *ServerHostName* is the name of the host where the server process is running.

The client application invokes every operation, invokes all the attribute's get and set methods and displays the whole process to standard output.

This client-server application can be used to accomplish any of the following:

- Demonstrating or testing an Orbix client-server application for a particular interface or interfaces.
- Generating sample code to see how to initialize and pass parameters.
- Generating a starting point for an application.

Generating a Partial Application

The genie can generate a whole client-server application or it can just generate the parts desired by the programmer. To generate any kind of starting-point code from an IDL file (or files) you must first choose which kind of code you wish to generate.

One area of repetitive coding in Orbix occurs when the programmer wants to write the classes that implement IDL interfaces. To generate the skeleton implementation class for the `account` interface in the `finance.idl` file, run the genie application as follows:

```
idlgen cpp_genie.tcl -interface -incomplete account
finance.idl

finance.idl:
idlgen: creating account_i.h
idlgen: creating account_i.cxx
```

The -interface option tells the genie to generate the classes that implement IDL interfaces. The -incomplete option specifies that the operations and attributes of the generated classes have empty bodies. Specifying the name of an interface (for example, account) causes the genie to consider only that interface when generating code.

The previous command generates the account_i.h and account_i.cxx files that contain the outline of a class, account_i, that implements the account interface.

For example, given the following definition of the account interface:

```
// IDL
interface account {
    readonly attribute float balance;

    void makeLodgement(in float f);
    void makeWithdrawal(in float f);
};
```

The following code is generated:

```
// C++
class account_i : public virtual accountBOAImpl
{
public:
   ...
   virtual void makeLodgement(
      CORBA::Float f,
      CORBA::Environment&_env =
                  CORBA::IT_chooseDefaultEnv())
      throw(CORBA::SystemException);

   virtual void makeWithdrawal(
      CORBA::Float f,
      CORBA::Environment&_env =
                  CORBA::IT_chooseDefaultEnv())
      throw(CORBA::SystemException);

   virtual CORBA::Float balance(
      CORBA::Environment&_env =
                  CORBA::IT_chooseDefaultEnv())
   ...
};
```

Command-Line Options to Generate Parts of an Application

The C++ genie generates a complete application by generating different files, such as a client main file (`client.cxx`), server main file (`server.cxx`), smart proxies, classes that implement IDL interfaces, a makefile and so on. The C++ genie provides command-line options to selectively turn the generation of each type of code on and off. In this way, you can instruct the C++ genie to generate as much or as little of an application as you want. Table 3.2 describes the various command-line options:

Command-Line Option	Description
`-(no)interface`	Generates the classes that implement the interfaces in the IDL.
`-(no)smart`	Generates smart proxy classes.
`-(no)loader`	Generates a single loader class for all the interfaces in an IDL.
`-(no)server`	Generate a simple server main file.
`-(no)client`	Generate a simple client application.
`-(in)complete`	Generates skeletal clients and servers.
`-(no)makefile`	Generates a Makefile that can build the server and client applications.

Table: 3.2: *C++ Genie Command-Line Options*

Each of these command-line options is available in two forms, which can switch the feature either on or off. For example, the `-interface` option generates implementation classes; whereas the `-nointerface` option suppresses generation of implementation classes.

These command-line options are described in the following sections.

-interface: Classes that Implement Interfaces

You can generate the classes that implement the interfaces in an IDL file using the `-interface` option:

```
idlgen cpp_genie.tcl -interface bank.idl
```

This generates a class header and implementation code for each interface that appears in the IDL file.

Consider the `account` interface that appears in the `bank.idl` file. The `account` interface is implemented by a generated class, `account_i`. The `_i` suffix is specified by the `default.cpp.impl_class_suffix` setting in the `idlgen.cfg` configuration file. The `account_i` class is defined in the `account_i.cxx` file.

There are two mechanisms for implementing an interface: the *TIE approach* and the *BOAImpl approach*. The genie allows you to specify which one is to be used. The option `-boa` specifies the BOAImpl approach, for example:

```
idlgen cpp_genie.tcl -interface -boa bank.idl
```

The option `-tie` specifies the TIE approach, for example:

```
idlgen cpp_genie.tcl -interface -tie bank.idl
```

The default approach is specified by the `default.cpp_genie.want_boa` entry in `idlgen.cfg`.

By default, a function called `_this()` is generated for each implementation class. This operation provides a reference to the CORBA object. For interfaces implemented using the BOA approach, `_this()` simply returns `this`. For interfaces implemented using the TIE approach, `_this()` returns the back pointer which was initialized in a static `create()` method (described in the next paragraph). The `_this()` function makes it possible for a TIE object to pass itself as a parameter to an IDL operation.

Note: The `-no_this` command-line option can be used to suppress the generation of the `_this()` operation.

A related matter is how the constructors of an implementation class are used. In the code generated by the C++ genie, constructors are protected and hence cannot be called directly from application code. Instead, objects are created by calling a `public static` operation called `_create()`. If the TIE approach is used for implementing interfaces, the algorithm used in the implementation of this operation is as follows:

```
// C++
foo_ptr foo_i::_create(const char *marker,
                       CORBA::LoaderClass *l=0)
{
   foo_i* obj
   foo_ptr tie_obj;

1     obj = new foo_i(marker, l);
2     tie_obj = new TIE_foo(foo_i)(obj, marker, l);
3     obj->m_this = tie_obj; // set the back ptr
   return tie_obj;
}
```

The `_create()` operation calls the constructor, 1. It then creates the TIE wrapper object, 2, and sets a back pointer from the implementation object to its TIE wrapper, 3. If the BOA approach is used instead then steps 2 and 3 are omitted. By providing this `_create()` operation, you can ensure that there is a consistent way for application code to create CORBA objects, irrespective of whether the TIE or BOA approach is used.

Another matter to be aware of is how modules affect the name of the implementation class. The C++ genie flattens interface names that appear in modules.

Consider this short extract of IDL:

```
// IDL
module finance {
   interface account {
      ...
   };
};
```

The account interface here is implemented by a class `finance_account_i`. The interface name has been flattened with the module name.

-smart: Smart Proxies

Use the -smart option to generate smart proxy classes for all the interfaces in an IDL file:

```
idlgen cpp_genie.tcl -smart bank.idl
```

This generates a smart proxy class header and corresponding skeletal implementation for each interface that appears in the IDL file.

Consider the account interface that appears in the bank.idl file. The account interface will have a smart proxy class called smart_account. The smart_ prefix is specified by the entry default.cpp.smart_proxy_prefix in idlgen.cfg. The smart_account class is also created in a file of the same name and with a class definition of the following form:

```
// C++
class smart_account : public virtual account
{
public:
   smart_account(
      char            *OR,
      CORBA::Boolean   diagnostics);
   virtual ~smart_account();

   virtual void makeLodgement(
      CORBA::Float f,
      CORBA::Environment& _env =
               CORBA::IT_chooseDefaultEnv())
      throw(CORBA::SystemException);

   virtual void makeWithdrawal(
      CORBA::Float f,
      CORBA::Environment& _env =
            CORBA::IT_chooseDefaultEnv())
      throw(CORBA::SystemException);

   virtual CORBA::Float balance(
      CORBA::Environment& _env =
            CORBA::IT_chooseDefaultEnv())
};
```

A corresponding smart proxy factory class is also created and appears in the same file. In the case of the `smart_account` proxy class, the corresponding factory class is of the form:

```
// C++
class smart_accountProxyFactoryClass
        : public virtual accountProxyFactoryClass
{
public:
   smart_accountProxyFactoryClass(
      CORBA::Boolean  factoryDiagnostics,
      CORBA::Boolean  proxyDiagnostics);
   virtual ~smart_accountProxyFactoryClass();

   virtual void *New(
      char *OR,
      CORBA::Environment&);
   virtual void *New(
      ObjectReferenceImpl *OR,
      CORBA::Environment&);
};
```

The constructor for this smart proxy factory takes two boolean parameters. The first is used to turn diagnostic messages on and off in the `New()` operation of the factory object. The second parameter is used to turn diagnostic messages on and off in the operations of smart proxy objects. These diagnostic messages can be useful both as a teaching aid and as a debugging aid.

A single instance of the smart proxy factory class is created at the end of the generated source file, which in this case is the `smart_account.cxx` file:

```
// C++
smart_accountProxyFactoryClass
        my_smart_accountProxyFactoryClass(1,1);
```

The parameters passed to the constructor of this smart proxy factory activate both forms of diagnostics. You can edit these parameters to turn off the diagnostics if required.

-loader: Loaders

Use the `-loader` option to generate a single loader class for all the interfaces in an IDL file:

```
idlgen cpp_genie.tcl -loader bank.idl
```

This generates a single class that can be used as a loader for all the interface types that exist in the processed IDL file.

The loader class is of the form:

```
// C++
class loader : public CORBA::LoaderClass
{
public:
   loader(CORBA::Boolean printDiagnostics);
   virtual ~loader();

   virtual CORBA::Object_ptr load(
      const char              *it_interface,
      const char              *marker,
      CORBA::Boolean           isLocalBind,
      CORBA::Environment&);

   virtual void save(
      CORBA::Object_ptr       obj,
      CORBA::saveReason       reason,
      CORBA::Environment&);

   virtual void record(
      CORBA::Object_ptr         obj,
      char                     *&marker,
      CORBA::Environment&);

   virtual CORBA::Boolean rename(
      CORBA::Object_ptr         obj,
      char                     *&marker,
      CORBA::Environment&);
};
```

Like the smart proxy factory, the constructor for a loader takes a boolean parameter which is used to turn diagnostic messages on and off.

Note: The creation of the loader is in the generated `server.cxx` main file and uses a `true` value when creating the loader, thereby enabling diagnostic messages. You can alter this if required.

The `load()` operation on this loader recreates an object by calling the static `create` operation of the appropriate implementation class. The `save()` operation on a loader delegates its responsibility by calling the `_loaderSave()` operation on the specified object. Each implementation class generated by the genie is given this operation `_loaderSave()`.

-server: Server Main File

Use the `-server` option to generate a simple server main file:

```
idlgen cpp_genie.tcl -server bank.idl
```

This generates a file called `server.cxx` which is of the form:

```
// C++
int main(int argc, char **argv)
{
   // Local Variables
   CORBA::ORB_var orbVar;
   CORBA::BOA_var boaVar;
   try {
      orbVar = CORBA::ORB_init ( argc , argv, "Orbix");
      boaVar = orbVar->BOA_init ( argc, argv, "Orbix_BOA");
   } catch (CORBA::SystemException e) {
      cerr << "Unexpected System Exception :" << e << endl;
      exit (1);
   } catch (...) {
      cerr << "Unexpected Exception." << endl;
      exit (1);
   }

   bank_var     obj1;
   account_var  obj2;

   //--------
   // Initialize Orbix.
   //--------
```

```
    orbVar->setDiagnostics(1);
    try {
       boaVar->impl_is_ready("banksimpleSrv", 0);
    } catch(CORBA::SystemException &ex) {
       cerr  << "impl_is_ready() failed" << endl
             << ex << endl;
       exit(1);
    }
    obj1 = bank_i::_create("bank-1");
    obj2 = account_i::_create("account-1");

    //--------
    // Application-specific initialisation.
    //--------
    ofstream ofile;
    ofile.open ("bank.ior");
    ofile << obj1->_object_to_string();
    ofile.close ();

    ofile.open ("account.ior");
    ofile << obj2->_object_to_string();
    ofile.close ();


    //--------
    // Main event loop.
    //--------
    try {
       CORBA::Orbix.processEvents();
    } catch(CORBA::SystemException &ex) {
       cerr << "processEvents() failed" << endl
            << ex << endl;
       exit(1);
    }

    //--------
    // Terminate.
    //--------
    return 0;
}
```

This server makes object references available to clients by writing them to files. The object references for the bank object and the account object are converted to string format and written to the files bank.ior and account.ior respectively.

If a loader had been requested by using the -loader option:

```
idlgen cpp_genie.tcl -server bank.idl
```

The server code would have included the following lines:

```
// C++
   loader*      srvLoader;
   ...
   srvLoader = new loader(1);
   obj1 = bank_i::_create("bank-1", srvLoader);
   obj2 = account_i::_create("account-1", srvLoader);
```

-client: Client Application

Use the -client option to generate a simple client application:

```
idlgen cpp_genie.tcl -client bank.idl
```

This generates a source file client.cxx with a simple main(). The client source file is of the form:

```
// C++
int main(int argc, char **argv)
{
   bank_var     obj1;
   account_var  obj2;
   //--------
   // Set Orbix diagnostics level
   //--------
   CORBA::Orbix.setDiagnostics(1);

   ifstream iorfile;
   char myIor [ 2048 ];
   CORBA::Object_var tObj;

   try {
      iorfile.open ("bank.ior");
      iorfile >> myIor;
      iorfile.close();
```

```
        tObj = CORBA::Orbix.string_to_object(myIor);
        obj1 = bank::_narrow(tObj);

        iorfile.open ("account.ior");
        iorfile >> myIor;
        iorfile.close();
        tObj = CORBA::Orbix.string_to_object(myIor);
        obj2 = account::_narrow(tObj);

    } catch (CORBA::SystemException sysEx ) {
        cerr << "Unexpected Exception: " << sysEx << endl;
        exit(1);
    }

    //--------
    // Invoke the operations and attributes
    //--------
    call_account_get_balance(obj1);
    call_account_makeLodgement(obj1);
    call_account_makeWithdrawal(obj1);
    call_bank_newAccount(obj2);
    call_bank_deleteAccount(obj2);

    //--------
    // Terminate gracefully.
    //--------
    return 0;
}
```

The client obtains references to each of the CORBA objects by reading stringified object references from the files created by the server (`bank.ior` and `account.ior`). The client then invokes every operation and attribute with random parameter values.

-incomplete: Skeletal Clients and Servers

The `-incomplete` option is used to suppress the generation of dummy implementation code for the generated client and server applications.

By default (or using the `-complete` option), the C++ genie produces dummy implementations for the client and server whenever the `-client`, `-server`, and `-interface` options are specified. The dummy implementation provides the following functionality:

- The client `main()` function contains code to invoke every operation and attribute on every interface (`-client` option).
- The server `main()` function contains code to create one instance of a CORBA object for every interface and to distribute the object references to clients (`-server` option).
- The bodies of operations and attributes are implemented by code that prints out the parameters and generates random return values (`-interface` option).

If the `-incomplete` option is specified, the generated code is reduced to the minimum amount of boilerplate code in each case. For example, clients do not invoke any remote operations and the bodies of operations and attributes are left empty.

-makefile: Makefile

Use the `-makefile` option to obtain a makefile that can build the server and client applications. The makefile also provides two other targets: `clean` and `putit`.

```
make clean
make putit
```

The `putit` target registers the server in the Implementation Repository and the `clean` target removes any files generated during compilation and linking.

Other Command-Line Options

For a full list of the command-line options for the Orbix C++ Genie please refer to the Appendix A, "User's Reference" on page 387.

Other C++ Genies

In addition to the `cpp_genie.tcl`, a number of other C++ genies are supplied with the Orbix Code Generation Toolkit, as shown in Table 3.3.

C++ Genie	Description
`cpp_op.tcl`	Generates C++ code for new operations from an IDL interface.
`cpp_print.tcl`	Creates a number of C++ functions that can display all the data types present in an IDL file.
`cpp_random.tcl`	Creates a number of C++ functions that generate random values for all the types present in an IDL file.
`cpp_equal.tcl`	Creates utility functions that test IDL types for equality.

Table: 3.3: *Additional C++ Genies*

The output from these genies can generate extra C++ source code that you might find useful when you are writing your own applications. The following sections discuss each of these genies in more detail.

cpp_op.tcl—Generating Signatures of Individual Operations

The C++ genie is useful when starting a new project. However, IDL interfaces often change during application development. For example, a new operation might be added to an interface, or the signature of an existing operation might be changed. Whenever such a change occurs, you have to update existing C++ code with the signatures of the new or modified operations. This is where the `cpp_op.tcl` genie is useful. This genie prints the C++ signatures of specified operations and attributes to a file. The user can then paste these operations back into the target source files.

Imagine that the operation `newAccount()` is added to the interface `bank`. To generate the new operation run the genie as follows:

```
idlgen cpp_op.tcl bank.idl "*::newAccount"
```

```
idlgen: creating tmp
Generating signatures for bank::newAccount
```

As this example shows, you can use wild cards to specify the names of operations or attributes. If you do not explicitly specify any operations or attributes, the `*` wild card is used by default, which causes the signatures of all operations and attributes to be generated. By default, this genie writes the generated operations into the file `tmp`. You can specify an alternative file name by using the `-o` command-line option:

```
idlgen cppsig.tcl bank.idl -o ops.txt "*::newAccount"
```

```
idlgen: creating ops.txt
Generating signatures for bank::newAccount
```

By default, wild cards are matched only against the names of operations and attributes in the specified file. If you specify the `-include` option then the wildcards are matched against all operations and attributes in the included IDL files too.

cpp_print.tcl—Creating Print Functions for IDL Types

This genie generates utility functions to print IDL data types. It is run as follows:

```
idlgen cpp_print.tcl foo.idl
```

```
idlgen: creating it_print_funcs.h
idlgen: creating it_print_funcs.cxx
```

The names of the generated files are always `it_print_funcs.{h,cxx}`, regardless of the name of the input IDL file. The functions in these generated files all have names of the form `IT_print_`*XXX* where *XXX* is the name of an IDL type.

To illustrate the print functions, consider the following IDL definitions:

```
// IDL
enum employee_grade {temporary, junior, senior};

struct EmployeeDetails {
    string           name;
    long             id;
    double           salary;
    employee_grade   grade;
};

typedef sequence<EmployeeDetails> EmployeeDetailsSeq;
```

When you run `cpp_print.tcl` on this IDL, utility print functions are generated for all the user-defined IDL types (and also for built-in IDL types). The generated print utility function for the `EmployeeDetailsSeq` type has the following signature:

```
// C++
void IT_print_EmployeeDetailsSeq(ostream &out,
       const EmployeeDetailsSeq &seq,
       int indent = 0);
```

The signatures of print functions for the other IDL types are similar. This function takes three parameters. The first parameter is the `ostream` to be used for printing. The second parameter is the IDL type to be printed. The final parameter, `indent`, specifies the indentation level at which the IDL type is to be

printed. This parameter is ignored when printing simple types such as `long`, `short`, `string`, and so on. It is only used when printing a compound type such as a `struct`, in which case the members *inside* the `struct` should be indented one level deeper than the enclosing `struct`.

An example of using the print functions is shown below:

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

void foo_i::op(const EmployeeDetailsSeq &emp, ...)
{
   if (m_do_logging) {
      //--------
      // Write parameter values to a log file.
      //--------
      cout    << "op() called; 'emp' = ";
      IT_print_EmployeeDetailsSeq(m_log, emp, 1);
      cout    << endl;
   }
   ... // Rest of operation.
}
```

The contents of the log file written by the above snippet of code might look like the following:

```
op() called; 'emp' parameter =
    sequence EmployeeDetailsSeq length = 2 {
        [0] =
            struct EmployeeDetails {
                name = "Joe Bloggs"
                id = 42
                salary = 29000
                grade = 'senior'
            } //end of struct EmployeeDetails
        [1] =
            struct EmployeeDetails {
                name = "Joan Doe"
                id = 96
                salary = 21000
                grade = 'junior'
            } //end of struct EmployeeDetails
    } //end of sequence EmployeeDetailsSeq
```

Aside from their use as a logging aid, these print functions can also be a very useful debugging aid. For example, consider a client application that reads information from a database, stores this information in an IDL `struct` and then passes this `struct` as a parameter to an operation in a remote server. If you wanted to confirm that the code to populate the fields of the `struct` from information in a database was correct then you could use a generated print function to examine the contents of the `struct`.

The C++ genie makes use of `cpp_print.tcl` so that the generated client and server applications can print diagnostics showing the values of parameters that are passed to operations.

cpp_random.tcl—Creating Random Functions for IDL Types

This application generates utility functions to assign random values to IDL data types. It is run as follows:

```
idlgen cpp_random.tcl foo.idl
```

```
idlgen: creating it_random_funcs.h
idlgen: creating it_random_funcs.cxx
```

The names of the generated files are always `it_random_funcs.{h,cxx}`, regardless of the name of the input IDL file. The functions in these generated files all have names of the form `IT_random_XXX` where *XXX* is the name of an IDL type. The functions generated for small IDL types (`long`, `short`, `enum`, and so on) return the random value. Thus, you can write code as follows:

```
// C++
CORBA::Long l;
CORBA::Double d;
colour col;   // an enum type
CORBA::String_var str;

l   = IT_random_long();
d   = IT_random_double();
col = IT_random_col();
str = IT_random_string();
```

However, in the case of compound types (struct, union, sequence, and so on), it would be inefficient to return the random value (since this would involve copying a potentially large data-type on the stack). Instead, for these compound types, the generated function assigns a random value directly to a reference parameter. For example:

```
// C++
CORBA::Any any;
EmployeeDetails emp;  // a struct
EmployeeDetailsSeq seq;  // a sequence

IT_random_any(any);
IT_random_EmployeeDetails(emp);
IT_random_EmployeeDetailsSeq(seq);
```

Aside from the functions to assign random values for various IDL types, the following are also defined in the generated files:

```
// C++
void IT_random_set_seed(unsigned long new_seed);
unsigned long IT_random_get_seed();
long IT_random_get_rand(unsigned long range = 65536UL);
void IT_random_reset_recursive_limits();
```

IT_random_set_seed() is used to set the seed for the random number generator.

IT_random_get_seed() returns the current value of this seed.

IT_random_get_rand() returns a new random number in the specified range.

IDL allows the declaration of recursive types. For example:

```
// IDL
struct tree {
   long            data;
   sequence<tree>  children;
};
```

When generating a random tree, the IT_random_tree() function calls itself recursively. Care must be taken to ensure that the recursion terminates. This is done by putting a limit on the depth of the recursion. IT_random_reset_recursive_limits() is used to reset the limit for a recursive struct, a recursive union and type any (which can recursively contain other any objects).

The generated random functions can be a very useful prototyping tool. For example, when developing a client-server application, you often want to concentrate your efforts initially on developing the server. You can write a client quickly that uses random values for parameters when invoking operations on the server. In doing this, you will have a primitive client that can be used to test the server. Then when you have made sufficient progress in implementing and debugging the server, you can concentrate your efforts on implementing the client application so that it uses non-random values for parameters.

The C++ genie makes use of `cpp_random.tcl` so that the generated client can invoke operations (albeit with random parameter values) on objects in the server.

cpp_equal.tcl—Creating Equality Functions for IDL Types

The C++ language provides a built-in `operator==()` for the basic types such as `long` and `float`. C++ also allows you to define `operator==()` in classes. However, the OMG mapping from IDL to C++ does not specify that `operator==()` is provided in the C++ data-types representing IDL types. Thus, if `EmployeeDetails` is an IDL `struct` then, unfortunately, you *cannot* write C++ code such as:

```
// C++
EmployeeDetails    emp1;
EmployeeDetails    emp2;
... // initialise emp1 and emp2
if (emp1 == emp2) { ... }
```

Instead, you have to write code which laboriously compares each field inside `emp1` and `emp2`. The `cpp_equal.tcl` application addresses this issue by generating functions to test for equality of IDL data types. It is run as follows:

```
idlgen cpp_equal.tcl foo.idl
```

```
idlgen: creating it_equal_funcs.h
idlgen: creating it_equal_funcs.cxx
```

The names of the generated files are always `it_equal_funcs.{h,cxx}`, regardless of the name of the input IDL file. The functions in these generated files all have names of the form `IT_is_eq_XXX` where XXX is the name of an IDL type. You can use these functions as follows:

```
// C++
EmployeeDetails    emp1;
EmployeeDetails    emp2;
... // initialise emp1 and emp2
if (IT_is_eq_EmployeeDetails(emp1,emp2)) { ... }
```

These equality testing functions are generated for type `any`, `TypeCode`, and every `IDL struct`, `union`, `sequence`, `array`, and `exception`. The function `IT_is_eq_obj_refs()`is provided to test the equality of two object references.

Configuration Settings

The configuration settings for the C++ genie are contained in the scope `default.cpp_genie` in the `idlgen.cfg` configuration file.

Some other settings are not, technically speaking, settings specifically for the C++ genie, but are settings used by the `std/cpp_boa_lib.tcl` library, which maps IDL constructs to their C++ equivalents. As the C++ genie uses this library extensively, its outputs are affected by these settings. They are held in the scope `default.cpp`.

For a full listing of these settings please refer to Appendix A, “User’s Reference” on page 387.

4

Ready-to-Use Genies for Orbix Java Edition

The Orbix Code Generation Toolkit is packaged with several genies for use with IONA's product OrbixWeb which maps OMG IDL to the Java language. This chapter explains what these genies are and how to use them effectively.

Using the Java Genie to Kickstart New Projects

Many people start a new project by copying some code from an existing project and then editing this code to change the names of variables, signatures of operations, and so on. This is boring and time-consuming work. The Java genie (`java_genie.tcl`) is a powerful utility which eliminates this task. If you have an IDL file that defines the interfaces for your new project then the Java genie can generate a demonstration, client-server application that contains all the starting-point code that you are likely to need for your project. In just a few seconds, the Java genie can give your project a kickstart, and make you productive immediately.

Generating a Client-Server Application

You can use the Java genie to generate a complete client-server application. It produces a `makefile` and a complete set of compilable code for both a client and server for the specified interfaces. For example:

```
idlgen java_genie.tcl -all -jp MyPackage finance.idl
```

```
finance.idl:
java_genie.tcl: no change to idlgen/PrintFuncs.java
java_genie.tcl: no change to idlgen/MyPackage/Printbank.java
java_genie.tcl: no change to idlgen/MyPackage/Printaccount.java
java_genie.tcl: no change to idlgen/MyPackage/PrintCashAmount.java
java_genie.tcl: no change to idlgen/RandomFuncs.java
java_genie.tcl: no change to idlgen/MyPackage/Randombank.java
java_genie.tcl: no change to idlgen/MyPackage/Randomaccount.java
java_genie.tcl: no change to idlgen/MyPackage/
RandomCashAmount.java
java_genie.tcl: no change to idlgen/RandomMyPackage.java
java_genie.tcl: no change to MyPackage/bankCaller.java
java_genie.tcl: no change to MyPackage/accountCaller.java
java_genie.tcl: no change to MyPackage/bankLog.java
java_genie.tcl: no change to MyPackage/_clt_opbankLog.java
java_genie.tcl: no change to MyPackage/_srv_opbankLog.java
java_genie.tcl: no change to MyPackage/accountLog.java
java_genie.tcl: no change to MyPackage/_clt_opaccountLog.java
java_genie.tcl: no change to MyPackage/_srv_opaccountLog.java
java_genie.tcl: no change to MyPackage/Log.java
java_genie.tcl: no change to MyPackage/bankImpl.java
java_genie.tcl: no change to MyPackage/accountImpl.java
java_genie.tcl: no change to MyPackage/Smartbank.java
java_genie.tcl: no change to MyPackage/Smartbank_Factory.java
java_genie.tcl: no change to MyPackage/Smartaccount.java
java_genie.tcl: no change to MyPackage/Smartaccount_Factory.java
java_genie.tcl: no change to MyPackage/Loader.java
java_genie.tcl: no change to client.java
java_genie.tcl: no change to server.java
java_genie.tcl: no change to Makefile
```

The generated client application calls every operation in the server application and passes random values as parameters to the operations and attribute get/set methods. The server application then passes random values back in the `inout`, `out` and `return` values of the operations.

Choosing an Object Reference Distribution Method

To establish initial contact between a client and a server application, the server has to distribute initial object references to its clients. The `java_genie.tcl` genie lets you select the object reference distribution method using a command-line option. You can choose between three mutually exclusive methods of object distribution, as described in Table 4.1.

Command-Line Option	Description
unspecified	(Default) Generate server code that distributes object references by writing stringified object references to files. Generate client code that reads the stringified object references from the server-created files.
`-ns`	Generate server code that distributes object references by creating object bindings in the naming service. Generate client code that reads the server-created bindings by resolving the object names.
`-bind`	(Deprecated) Generate client code that creates object references, based on the arguments passed to `_bind()`. The generated server performs no special steps.

Table: 4.1: *Object Distribution Methods*

Compiling and Running the Application

The `Makefile` generated by the code generation toolkit has a complete set of rules for building both the client and server applications. To build the client and server:

1. Compile the generated application. At a command prompt, enter the following commands:

   Windows

   ```
   > nmake depend
   > nmake
   ```

   UNIX

   ```
   % make depend
   % make
   ```

2. Run the Orbix daemon. Open a new MS-DOS prompt or `xterm` window (UNIX) and enter the following command:

   Windows

   ```
   > orbixd
   ```

   UNIX

   ```
   % orbixd
   ```

   The Orbix daemon runs in the foreground and logs its activities to this window.

3. Register the server with the Orbix daemon. At a command prompt, enter the following command:

   Windows

   ```
   > nmake putit
   ```

   UNIX

   ```
   % make putit
   ```

   If you are using the default approach to object reference distribution (writing stringified object references to file) or the `_bind()` approach (using the `-bind` option), proceed to directly to step 4.

If you are using the naming service approach to object reference distribution (using the `-ns` option), you have to set up the naming service as well.

Run the naming service. Open a new MS-DOS prompt or `xterm` window (UNIX) and enter the following command:

Windows

```
> nmake runns
```

UNIX

```
% make runns
```

Create the `IT_GenieDemo` context in the naming service. At a command prompt, enter the following command:

Windows

```
> nmake setup_ns
```

UNIX

```
% make setup_ns
```

4. Run the server. At a command prompt, enter the following command:

   Windows

   ```
   > nmake runserver
   ```

   UNIX

   ```
   % make runserver
   ```

5. Run the client. Open a new MS-DOS prompt or `xterm` window (UNIX) and enter the following command:

   Windows

   ```
   > nmake runclient
   ```

   UNIX

   ```
   % make runclient
   ```

The client application invokes every operation, invokes all the attribute's get and set methods and displays the whole process to standard output.

This client-server application can be used to accomplish any of the following:

- Demonstrating or testing an Orbix client-server application for a particular interface or interfaces.
- Generating sample code to see how to initialize and pass parameters.
- Generating a starting point for an application.

Generating a Partial Application

The genie can generate a whole client-server application or it can just generate the parts desired by the programmer. To generate any kind of starting-point code from an IDL file (or files) you must first choose which kinds of code you wish to generate.

One area of repetitive coding in OrbixWeb occurs when the programmer wishes to write the classes that implement the interfaces in the IDL file. To generate the skeleton implementation class for the `account` interface in the `finance.idl` file, you can run the genie in this way:

```
idlgen java_genie.tcl -interface -incomplete account
finance.idl
```

```
finance.idl:
idlgen: creating NoPackage/accountImpl.java
```

The `-interface` option tells the genie to generate the classes that implement IDL interfaces. The `-incomplete` option means that such generated classes will be "incomplete", that is, their operations and attributes will have empty bodies (rather than generated bodies which illustrate how to initialize parameters). Specifying the name of an interface (`account` in the above example) causes the genie to consider only that interface when generating code.

The previous command generates the file `accountImpl.java` that provides a skeleton class called `accountImpl` for implementing the `account` interface.

For example, assume that the account interface is defined as follows:

```
// IDL
interface account {
    readonly attribute float balance;

    void makeLodgement(in float f);
    void makeWithdrawal(in float f);
};
```

The corresponding extract of generated code is:

```
// Java
...
public class accountImpl
    implements _accountOperations, java.io.Serializable
{
    ...
    public void makeLodgement(float f)
    {
    }

    public void makeWithdrawal(float f)
    {
    }

    public float balance()
    {
    }
    ...
};
```

This saves the developer the time it would normally take to write this class by hand.

You can either explicitly enable specific code-generation options or you can use the `-all` option to turn them all on and then disable whichever options you do not want. For instance, the previous example could have been typed as:

```
idlgen java_genie.tcl bank.idl -all -nosmart -noloader
-nomakefile -noclient -noserver -jp MyPackage
```

By default, any wildcards specified on the command line are matched only against IDL interfaces in the specified file but if you specify the `-include` option then the wild cards are matched against IDL interfaces in all the included IDL files too.

Command-Line Options to Generate Parts of an Application

The Java genie generates a complete application by generating different files, such as a client main class (`client.java`), server main class (`server.java`), smart proxies, classes that implement IDL interfaces, a makefile and so on. The Java genie provides command-line options to selectively turn the generation of each type of code on or off. In this way, you can instruct the Java genie to generate as much or as little of an application as you want. Table 4.2 summarizes the Java genie command-line arguments:

Command-Line Option	Description
`-(no)interface`	Generates the classes that implement the interfaces in the IDL.
`-(no)smart`	Generates smart proxy classes.
`-(no)loader`	Generates a single loader class for all the interfaces in an IDL.
`-(no)server`	Generates a simple server main class.
`-(no)client`	Generates a simple client application.
`-(in)complete`	Generates skeletal clients and servers.
`-(no)makefile`	Generates a makefile that can build the server and client applications.
`-jp`	Specifies the package into which the generated Java code is placed. If you do not specify a package, the generated code is placed into a package called `NoPackage` by default.

Table: 4.2: *Java Genie Command-Line Options*

These command-line options are detailed in the following sections.

-interface: Classes that Implement Interfaces

You can generate the classes that implement the interfaces in an IDL file, using the `-interface` option:

```
idlgen java_genie.tcl -interface bank.idl -jp MyPackage
```

This generates a class and implementation code for each interface that appears in the IDL file.

Consider the interface, `account`, that appears in the `bank.idl` file. The `account` interface is implemented by a class of the same name but suffixed by `Impl`. The suffix is specified by the `default.java.impl_class_suffix` setting in the `idlgen.cfg` configuration file. The `accountImpl` class is also created in a file of the same name.

There are two mechanisms for implementing an interface: the TIE approach and the BOAImpl approach. The genie allows you to specify which one is to be used. The option `-boa` specifies the BOAImpl approach, for example:

```
idlgen java_genie.tcl -interface -boa bank.idl -jp MyPackage
```

The option `-tie` specifies the TIE approach, for example:

```
idlgen java_genie.tcl -interface -tie bank.idl -jp MyPackage
```

The default approach is specified by the `default.cpp_genie.want_boa` entry in `idlgen.cfg`.

The `_this()` method provides a reference to the CORBA object. For interfaces implemented using the BOA approach, `_this()` simply returns `this`. For interfaces implemented using the TIE approach, `_this()` returns the back pointer that was initialized in a static `_create` operation (described in the next paragraph). The `_this()` method makes it possible for a TIE object to pass itself as a parameter to an IDL operation.

Note: The `-no_this` command-line option can be used to suppress the generation of the `_this()` method.

A related matter is how implementation class constructors are used. In the code generated by the Java genie, constructors are protected and hence cannot be called directly from application code. Instead, objects should be created by

calling a `public static` operation called `_create`. If the TIE approach is used for implementing interfaces, the algorithm used in the implementation of this method is as follows:

```
// Java
foo _create(String marker, LoaderClass l)
{
    fooImpl obj
    foo tie_obj;

1       obj = new fooImpl(marker, l);
2       tie_obj = new _tie_foo(obj, marker, ());
3       obj.m_this = tie_obj; // set the back ptr
    return tie_obj;
}
```

The `create` operation calls the constructor, 1. It then creates the TIE wrapper object, 2, and sets a back pointer from the implementation object to its TIE wrapper, 3. If the BOA approach is used instead then steps 2 and 3 are omitted. By providing this `_create()` method, you can ensure that there is a consistent way for application code to create CORBA objects, irrespective of whether the TIE or BOA approach is used.

Another matter to be aware of is how modules affect the name of the implementation class. The Java genie chooses to flatten interface names that appear in modules.

Consider this short extract of IDL:

```
// IDL
module finance {
    interface account {
        ...
    };
};
```

The `account` interface here is implemented by a class `accountImpl` in the `finance` package.

-smart: Smart Proxies

Use the `-smart` option to generate smart proxy classes for all the interfaces in an IDL file:

```
idlgen java_genie.tcl -client -smart bank.idl
```

This generates a smart proxy class for each interface that appears in the IDL file.

Consider the `account` interface that appears in the `bank.idl` file. The smart proxy class for the `account` interface is called `Smartaccount`. The `Smart` prefix is specified by the `default.java.smart_proxy_prefix` entry in `idlgen.cfg`. The `Smartaccount` class is also created in a file of the same name with a class definition of the following form:

```
// Java
...
class Smartaccount extends _accountStub
{
    ...
    public Smartaccount()
        throws org.omg.CORBA.SystemException;
        { ... }

    public void makeLodgement(float f)
        { ... }

    public void makeWithdrawal(float f)
        { ... }

    public Float balance()  { ... }
    ...
};
```

A corresponding smart proxy factory class is also created and appears in the same file. In the case of the `Smartaccount` proxy class, the corresponding factory class is of the form:

```
// Java
...
class Smartaccount_Factory extends ProxyFactory
{
    public Smartaccount_Factory()
    { ... }

    public org.omg.CORBA.Object
    New(org.omg.CORBA.portable.Delegate del)
    { ... }
};
```

A single instance of the smart proxy factory class is created in the `createSmartProxyFactories()` method in the `client.java` file.

-loader: Loaders

Use the `-loader` option to generate a single loader class for all the interfaces in an IDL file:

```
idlgen java_genie.tcl -loader bank.idl
```

This generates a single class that can be used as a loader for all the interface types that exist in the processed IDL file.

The loader class is of the form:

```
// Java
...
public class Loader extends LoaderClass {
    public Loader()
    {
        super(true);
    }

    public
    org.omg.CORBA.Object load(String  it_interface,
                              String  marker,
                              boolean isLocalBind)
    { ... }
```

```
    public
    void save(org.omg.CORBA.Object  obj,
            int reason)
    { ... }

    public
    void record(org.omg.CORBA.Object    obj,
              StringHolder            marker)
    { ... }

    public
    boolean rename(org.omg.CORBA.Object       obj,
                 StringHolder               marker)
    { ... }
}
```

The `load()` method uses Java serialization to recreate previously saved objects. If it cannot find a previously saved object it makes a new instance using `_create()`. The `save()` method uses Java serialization to write an object to file.

-server: Server Main Function

Use the `-server` option to generate a simple server main function:

```
idlgen java_genie.tcl -server bank.idl
```

This generates a file called `server.java` which is of the form:

```
// Java
...
public static
void main(String[] args)
{
    server                          _srv = null;
    org.omg.CORBA.ORB               orb  = null;

    process_cmd_line_args(args);
    java.util.Properties p = null;

    orb = org.omg.CORBA.ORB.init(args,p);
    _OrbixWeb.ORB(orb).setConfigItem(
        "IT_IMPL_READY_IF_CONNECTED", "false"
```

```
);
_srv = new server(orb,args);

System.out.println(
    "Calling impl_is_ready(" +"genieSrv" + ",0)"
);
try {
    _OrbixWeb.ORB(_CORBA.Orbix).impl_is_ready("genieSrv",0);
}
catch (SystemException se) {
   System.out.println(
       "Exception during impl_is_ready : " + se.toString()
    );
    System.exit (1);
}

try {
    System.out.println(
        "Creating object obj1 = NoPackage.bankImpl"
    );
    obj1 = NoPackage.bankImpl._create("bank-1");
     _OrbixWeb.ORB(_CORBA.Orbix).connect(obj1);
    System.out.println(
        "Creating object obj2 = NoPackage.accountImpl"
    );
    obj2 = NoPackage.accountImpl._create("account-1");
     _OrbixWeb.ORB(_CORBA.Orbix).connect(obj2);
}
catch (SystemException se) {
    System.out.println (
      "Exception during creation of Implementation objects : "
      + se.toString()
    );
    System.exit (1);
}
if ( !writeReference( obj1, "bank.ref")) {
    System.out.println(
        "Failed to write object reference for bank"
    );
}
if ( !writeReference( obj2, "account.ref")) {
    System.out.println(
        "Failed to write object reference for account"
```

```
        );
    }

    System.out.println(
        "Calling impl_is_ready (" + "genieSrv" + ", " + "-1" + ")"
    );
    try {
        _OrbixWeb.ORB(_CORBA.Orbix).impl_is_ready("genieSrv",-1);
    }
    catch(org.omg.CORBA.SystemException se)
    {
        System.out.println(
            "Exception during impl_is_ready :" + se.toString());
        System.exit(1);
    }

    System.out.println("Server Exiting ... ");
    System.exit(1);
}
```

This server makes object references available to clients by writing them to files. The object references for the bank object and the account object are converted to string format and written to the files `bank.ref` and `account.ref` respectively.

If a loader had been requested by using the `-loader` option:

```
idlgen java_genie.tcl -server -loader bank.idl
```

The server code would have included the following lines:

```
// Java
srvLoader = new NoPackage.Loader();
...
obj1 = NoPackage.bankImpl._create("bank-1",srvLoader);
...
obj2 = NoPackage.accountImpl._create("account-1",srvLoader);
```

-client: Client Application

Use the `-client` option to generate a simple client application:

```
idlgen java_genie.tcl -client bank.idl
```

This generates a source file, `client.java`, with a simple `main()` function. The client source file is of the form:

```
// Java
...
public static
void main(String [] args)
{
    client _this = null;

    client.process_cmd_line_args(args);
    org.omg.CORBA.ORB orb = null;

    try
    {
        Properties p = System.getProperties();
        orb = org.omg.CORBA.ORB.init(args,p);

        _this = new client(orb,args);
        _this.getServerObjectsViaIORReferenceFiles();
        // call all the methods
        _this.run();
    }
    catch(Exception ex)
    {
        ex.printStackTrace();
    }
}
...
public void run () {
    try
    {
        MyPackage.accountCaller.get_balance(obj1);
        MyPackage.accountCaller.makeLodgement(obj1);
        MyPackage.accountCaller.makeWithdrawal(obj1);

        MyPackage.bankCaller.newAccount(obj2);
        MyPackage.bankCaller.deleteAccount(obj2);
    }
    catch(Exception ex)
    {
        System.out.println("Remote call failed\n");
        ex.printStackTrace();
```

```
    }
    return;
}
...
```

The client obtains references to each of the CORBA objects by reading stringified object references from the files created by the server (`bank.ref` and `account.ref`). The client then invokes every operation and attribute with random parameter values.

-incomplete: Skeletal Clients and Servers

The `-incomplete` option is used to suppress the generation of dummy implementation code for the generated client and server applications.

By default (or using the `-complete` option), the Java genie produces dummy implementations for the client and server whenever the `-client`, `-server`, and `-interface` options are specified. The dummy implementation provides the following functionality:

- The client `main()` function contains code to invoke every operation and attribute on every interface (`-client` option).
- The server `main()` function contains code to create one instance of a CORBA object for every interface and to distribute the object references to clients (`-server` option).
- The bodies of operations and attributes are implemented by code that prints out the parameters and generates random return values (`-interface` option).

If the `-incomplete` option is specified, the generated code is reduced to the minimum amount of boilerplate code in each case. For example, clients do not invoke any remote operations and the bodies of operations and attributes are left empty.

-makefile: Makefile

Use the `-makefile` option to obtain a makefile that can build the server and client applications. The makefile also provides two other targets: `clean` and `putit`.

```
make clean
make putit
```

The `putit` target registers the server in the Implementation Repository and the `clean` target removes any files generated during compilation and linking.

Other Command-Line Options

For a full list of the command-line options for the Java genie please refer to the Appendix A, "User's Reference" on page 387.

Other Java Genies

In addition to the `java_genie.tcl`, a number of other Java genies are supplied with the Orbix Code Generation Toolkit, as shown in Table 4.3.

C++ Genie	Description
`java_print.tcl`	Creates a number of Java functions that can display all the data types present in an IDL file.
`java_random.tcl`	Creates a number of Java functions that generate random values for all the types present in an IDL file.

Table: 4.3: *Additional Java Genies*

The output from these genies can generate extra Java source code that you might find useful when you are writing your own applications. The following sections discuss each of these genies in more detail.

java_print.tcl—Creating Print Functions for IDL Types

The genie `java_print.tcl` generates utility functions to print IDL data types. It is run as follows:

```
idlgen java_print.tcl foo.idl

idlgen: creating PrintFuncs.java
```

The name of the generated file is `PrintFuncs.java` regardless of the name of the input IDL file. The functions are generated in a Java class called `NoPackage.Print`*TypeName*, and the print method is called *TypeName* (the package, `NoPackage`, is specified by the `default.java.printpackage_name` configuration variable). To illustrate these print functions, consider the following IDL definitions:

```
// IDL
enum EmployeeGrade {temporary, junior, senior};

struct EmployeeDetails {
    string           name;
    long             id;
    double           salary;
    EmployeeGrade    grade;
};

typedef sequence<EmployeeDetails> EmployeeDetailsSeq;
```

When you run `java_print.tcl` on the file containing the above IDL types, utility print functions are generated for all the user-defined IDL types in that IDL file (and also for the built-in IDL types). The generated print utility functions for the `EmployeeDetailsSeq` type is placed in a class `idlgen.NoPackage.PrintEmployeeDetailsSeq`.

Two print methods are provided by the PrintEmployeeDetailsSeq class:

```
// Java
public class PrintEmployeeDetailsSeq
{
    public static
    void EmployeeDetailsSeq(
        java.io.PrintStream _os,
        NoPackage.EmployeeDetails[] IT_seq,
        int indent
    ) { ... }
...
    public static
    void EmployeeDetailsSeq(
        java.io.PrintStream _os,
        NoPackage.EmployeeDetailsSeqHolder IT_seq,
        int indent
    ) { ... }
}
```

The methods are overloaded on the type of the second parameter. The first method prints an EmployeeDetails sequence and the second method prints the corresponding holder type, EmployeeDetailsSeqHolder.

The first parameter, _os, is the stream used for printing. The second parameter, IT_seq, is the IDL type to be printed. The final parameter, indent, specifies the indentation level at which the IDL type is to be printed. This parameter is ignored when printing simple types such as long, short, string and so on. It is used only when printing a compound type such as a struct, in which case the members *inside* the struct should be indented one level deeper than the enclosing struct.

An example using the print functions is shown below:

```
// Java
import idlgen.NoPackage.*;
...
void op( EmployeeDetailsSeq emp, ...)
{
    if (m_do_logging) {
        //--------
        // Write parameter values to a log file.
        //--------
        System.out.println("op() called; 'emp' = ");
```

```
        PrintEmployeeDetailsSeq.EmployeeDetailsSeq(
            m_log, emp, 1
        );
    }
    ... // rest of operation
}
```

The contents of the log file written by the above snippet of code might look like the following:

```
op() called; 'emp' parameter =
    sequence EmployeeDetailsSeq length = 2 {
        [0] =
            struct EmployeeDetails {
                name = "Joe Bloggs"
                id = 42
                salary = 29000
                grade = 'senior'
            } //end of struct EmployeeDetails
        [1] =
            struct EmployeeDetails {
                name = "Joan Doe"
                id = 96
                salary = 21000
                grade = 'junior'
            } //end of struct EmployeeDetails
    } //end of sequence EmployeeDetailsSeq
```

Aside from their use as a logging aid, these print functions can also be a very useful debugging aid. For example, consider a client application that reads information from a database, stores this information in an IDL `struct` and then passes this `struct` as a parameter to an operation in a remote server. If you wanted to confirm that the code to populate the fields of the `struct` from information in a database was correct then you could use a generated print function to examine the contents of the `struct`.

The Java genie makes use of `java_print.tcl` so that the generated client and server applications can print diagnostics showing the values of parameters that are passed to operations.

java_random.tcl—Creating Random Functions for IDL Types

The genie `java_random.tcl` generates utility functions to assign random values to IDL data types. It is run as follows:

```
idlgen java_random.tcl foo.idl

idlgen: creating RandomFuncs.java
```

The name of the generated file is `RandomFuncs.java`, regardless of the name of the input IDL file. The functions are generated in a Java class called `idlgen.RandomFuncs`*TypeName*, and the random method is simply called `Random`*TypeName*. The functions generated for small IDL types (`long`, `short`, `enum`, and so on) return the random value.

Thus, you can write code as follows:

```
// Java
int l;
Double d;
colour col;     // an enum type
String str;

l   = idlgen.RandomFuncs.randomlong();
d   = idlgen.RandomFuncs.randomdouble();
col = idlgen.RandomFuncs.randomcol();
str = idlgen.RandomFuncs.randomString();
```

Aside from the functions to assign random values for various IDL types, the following are also defined in the generated files:

```
// Java
void idlgen.IT_Random.set_seed(long new_seed);
long idlgen.IT_Random.get_seed();
long idlgen.IT_Random.get_rand(long range);

long idlgen.RandomFuncs.limitReached();
```

The methods can be explained as follows:

- `set_seed()` is used to set the seed for the random number generator.
- `get_seed()` returns the current value of this seed.

- `get_rand()` returns a new random number in the specified range.
- `limitReached()` returns TRUE when the maximum recursion depth is reached during the generation of a random value for a recursive type.

IDL allows the declaration of recursive types. For example:

```
// IDL
struct tree {
    long            data;
    sequence<tree>  children;
};
```

When generating a random `tree`, the `randomtree()` function calls itself recursively. Care must be taken to ensure that the recursion terminates. This is done by putting a limit on the depth of the recursion. The `max_recursive_depth` member variable defines the limit for a recursive `struct`, a recursive `union` and type `any` (which can recursively contain other `any` objects).

The Java genie makes use of `java_random.tcl` so that the generated client can invoke operations (albeit with random parameter values) on operations in the server.

Configuration Settings

The configuration settings for the Java genie are contained in the scopes:

- `default.orbix_web`
- `default.java_genie`

Some other settings are not, technically speaking, settings specifically for the Java genie, but are settings used by the development libraries. As the Java genie uses these command libraries extensively, its outputs are affected by these settings. They are held in the scope:

- `default.java`

For a full listing of these settings please refer to Appendix A on page 387.

5

Orbix C++ Client/Server Wizard

The Orbix C++ Client/Server Wizard is a graphical user interface that allows you to develop and compile an entire C++ application from an IDL file – both client and server. It is easy to use: you just point and click.

The Orbix C++ genie described in Chapter "Ready-to-Use Genies for Orbix C++ Edition" on page 17 allows you to develop C++ applications from the command-line. But if you use Microsoft Visual Studio 6.0 in Windows to build your C++ applications, you do not even need to use the command-line.

The Orbix C++ Client/Server Wizard is a Windows tool that you can use within the Visual Studio environment. It acts as a wrapper for the C++ genie, and permits you to:

- Choose which IDL files you want to convert to C++.
- Convert them to C++ client or server files with the Orbix C++ genie.
- Build the files into a working application using Visual Studio's normal build mechanism.

Using the Wizard

Using the wizard to build a C++ application from an IDL file is a simple four stage process:

1. Start the wizard from within Visual Studio.
2. Choose your IDL file.
3. Decide whether to generate client or server code.
4. Build the application with the generated code.

Starting the Wizard

The wizard files are inserted into your Developer Studio directory automatically during the Orbix C++ installation process. Therefore to start the wizard:

1. Run Visual Studio.
2. Select **File→New.**
3. Select **IONA Orbix C++ Client/Server Wizard** in the Projects window, giving your new project an appropriate name. The **IONA Orbix C++ Wizard – Step 1of 2** window is displayed as shown in Figure 5.1.
4. Select **Browse** to choose the IDL file from which you want to generate C++ code.
5. Select **Client** or **Server**, to generate C++ for client or server application. If you want to generate a set of code for both types of application, simply run the wizard twice.
6. Select one of **Stringified Object References**, **Naming Service**, or **Bind Call (Deprecated)** as the object distribution method.
7. Select **Advanced** if you want to set some advanced options.
8. Select **Next** to continue the process.

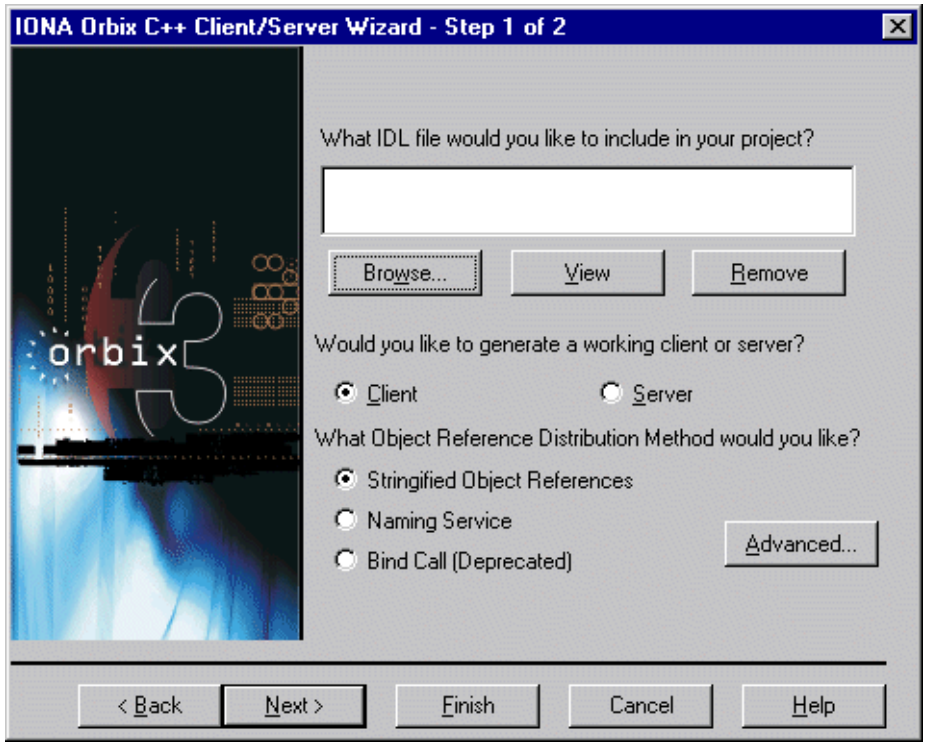

Figure 5.1: *Client/Server Wizard: Browsing for IDL Files*

Object Distribution Method

You can choose between three mutually exclusive methods of object distribution, as shown in Figure 5.1. You should choose the same option for both the client and the server projects; otherwise the generated client and server are not able to talk to each other. The object distribution methods are described in Table 5.1.

Object Distribution Method	Description
`Stringified Object References`	The generated server distributes object references by writing stringified object references to files. The generated client reads the stringified object references from the server-created files.
`Naming Service`	The generated server distributes object references by creating object bindings in the naming service. The generated client reads the server-created bindings by resolving the object names.
`Bind Call (Deprecated)`	The generated server performs no special steps. The generated client creates object references, based on the arguments passed to `_bind()`.

Table: 5.1: *Object Distribution Methods*

Advanced Code Generation Options

When you select **Advanced** from the window shown in Figure 5.1, the **Advanced Code Generation Options** window is displayed, as shown in Figure 5.2:

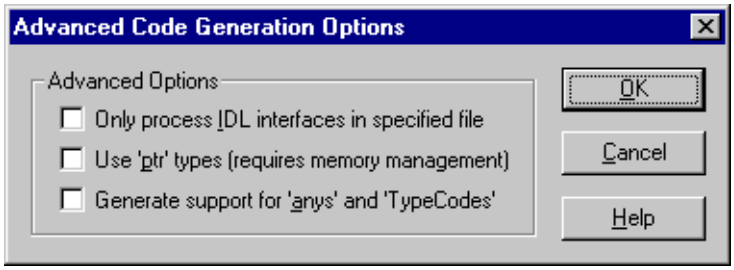

Figure 5.2: *Advanced Code Options Window*

You can set three options, which are described in Table 5.2:

Advanced Code Option	Effect
`Only process IDL interfaces in specified file`	By default, the wizard generates code for IDL interfaces for the both the specified file and all files in `#include` statements. Choosing this option forces the IDL compiler to generate stub or skeleton code only for the specified IDL file, ignoring any other IDL files from `#include` statements.

Table: 5.2: *Advanced Code Options*

Advanced Code Option	Effect
`Use 'ptr' types (requires memory management)`	By default, all object references (both proxies and implementation objects) in your generated code are managed by `_var` types. A `_var` type is a smart pointer that has the ability to manage the memory associated with the object reference. You can, however, choose to use the more primitive `_ptr` type, which performs no memory management on the object that it refers to.
`Generate support for 'anys' and 'TypeCodes'`	If your IDL uses the any CORBA data type, you should select this option to generate helper types and methods that enable you to insert and extract your complex types into and out of an any.

Table: 5.2: *Advanced Code Options*

Generating Client Code

When you generate an Orbix C++ client, you are presented with the window shown in Figure 5.3. You can choose to generate Smart Proxy code for your client by selecting **Generate Smart Proxies**.

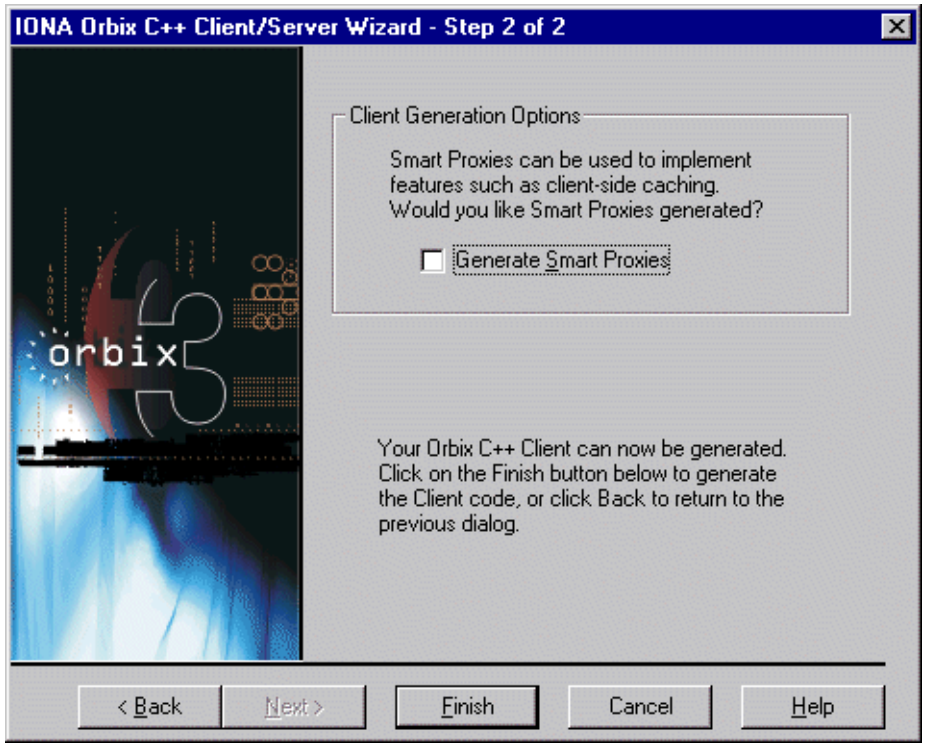

Figure 5.3: *Client Generation Options*

Smart Proxies are useful enhancements that enable you to override client requests made through IDL stubs. They can be added to and removed from your project without any changes required to the usual client code. Smart Proxies are typically used to implement features like client side caching, reporting or monitoring.

Once you have chosen whether or not you would like to have Smart Proxies generated, select the **Finish** button to complete the generation of your CORBA client.

Generating Server Code

When you generate an Orbix C++ server, you are presented with the window shown in Figure 5.4. You have several options to tailor the generated server code.

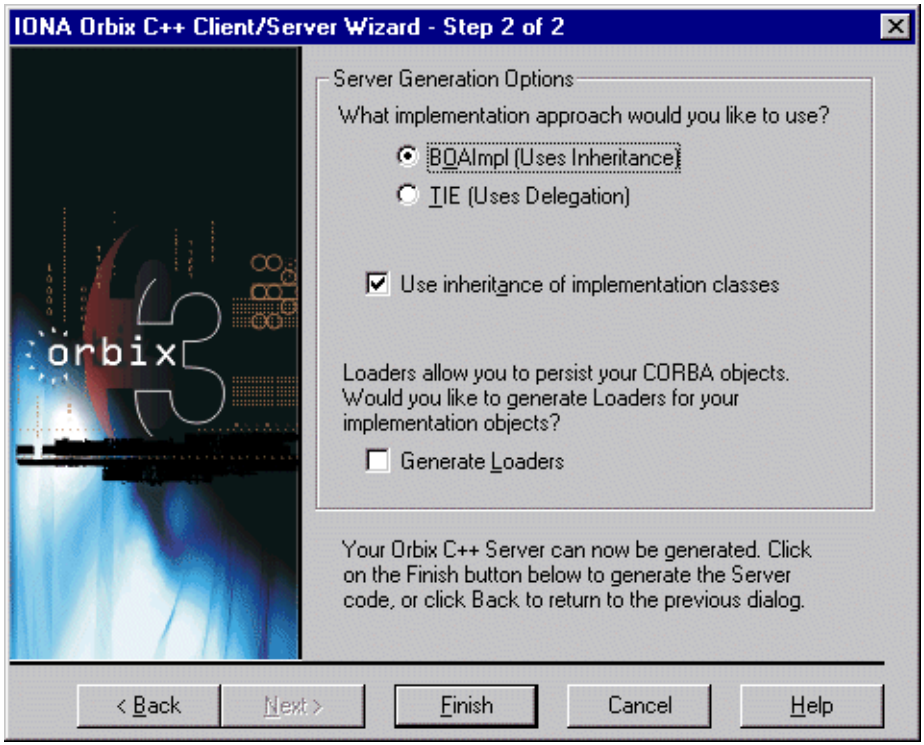

Figure 5.4: *Server Generation Options*

By default, your implementation objects use the **BOAImpl** approach to associate them with the corresponding interfaces. This approach uses inheritance to make the association. However, you can choose to use the **TIE** approach instead—this employs delegation to associate the implementation objects with the IDL interfaces. The choice is purely one of personal preference and has no implications for client code.

If you select **Uses inheritance of implementation classes**, your implementation objects inherit from each other. For example, if you have an IDL interface called `Account` and derived the `CheckingAccount` interface from it, then the generated C++ implementation of the `CheckingAccount` interface (usually called `CheckingAccount_i`) inherits its base functionality from the C++ implementation class, `Account_i`.

You can choose to create loaders for your implementation objects by selecting **Generate Loaders**. Loaders are useful for serializing (reading or writing) your objects to and from files or a database. The generated loader simply delegates loading and saving to the implementation objects that need to be serialized.

Once you have tailored your server options, select the **Finish** button to complete the generation of your CORBA server.

Building Your CORBA C++ Application

The wizard is now ready to generate the project. The **New Project Information** window, shown in Figure 5.5, lets you preview the files that are about to be generated. Click **OK** to proceed.

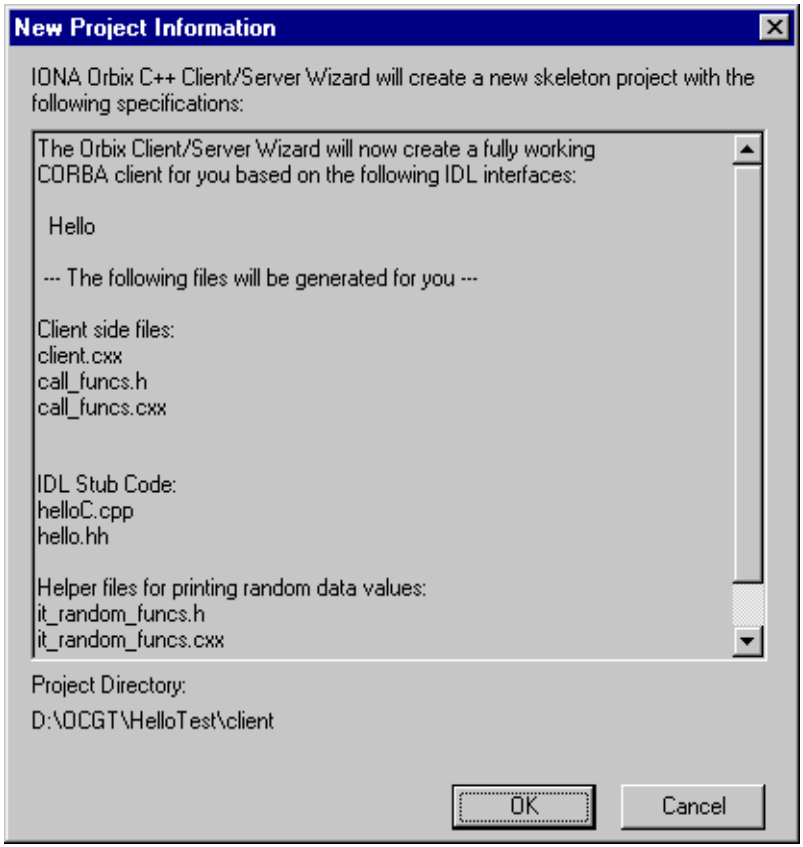

Figure 5.5: *Building a C++ Project with the Wizard*

Once this process is complete, the generated C++ files are available as a Visual Studio project. You can now begin to customize the application by editing the files or proceed directly to build the application by selecting **Build→Build.**

Part II

Developing Genies

6

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 5, 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.

The following topics are covered in this chapter:

- Hello World example.
- Including other Tcl files.
- Writing to a file.
- Embedding text in your application.
- Debugging and the bi2tcl utility.

Hello World Example

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 387.

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 87.
- 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 205.

Writing to a File

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.

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

```
// C++
class testClass : public virtual baseClass
{
     public:
            testClass() {
                cout << "testClass CTOR";
            }
};
```

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

Table 6.1 shows the three commands that are used to create 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.

Table: 6.1: *Creating a File*

Embedding Text in Your Application

Although the `output` command is concise, the example in "Writing to a File" on page 83 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:

```
// C++
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 83. 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:

```
// C++
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 6.2.

```
# 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
```

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.

Table: 6.2: *Bilingual File Escape Sequences*

Compare this with the example in "Embedding Text in Braces" on page 85 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.

If you run the bilingual example from "Embedding Text Using Bilingual Files" on page 87 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.

7

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 C on page 413 provides a reference to the commands discussed in this chapter.

The following topics are covered in this chapter:

- IDL files and idlgen.
- Traversing the parse tree with `rcontents`.
- Recursive descent traversal.
- Processing user-defined types.
- Recursive structs and unions.

IDL Files and idlgen

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.

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:

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 7.1.

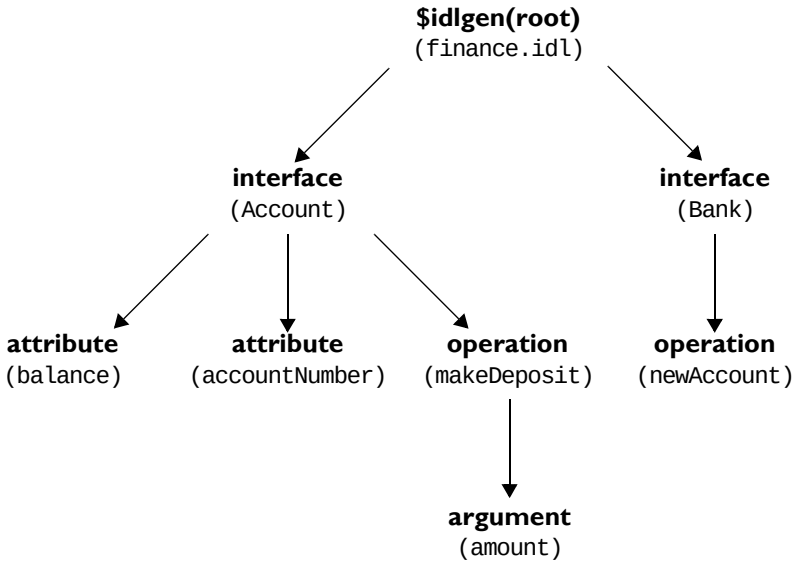

Figure 7.1: *The Finance IDL File's Parse Tree*

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.

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 common 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

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. 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
```

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 7.2.

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.

Two abstract node types 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*.

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.

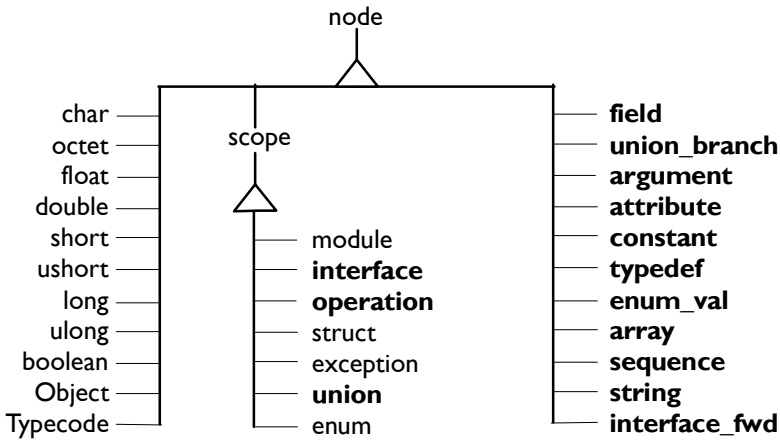

Figure 7.2: *Inheritance Hierarchy for Node Types*

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

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 414.

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 105 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 typedefs 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 typedefs can arise frequently. To eliminate this 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 typedefs, and returns the underlying type.

Thus, the previous 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 101), 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 414.

Traversing the Parse Tree with rcontents

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.

The `idlgrep` genie is developed in a series of iterations:

- Search using `contents`.
- Search using `rcontents`.
- Complete search genie.

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 Chapter 8 on page 117.

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:

```
// IDL
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: 7.1: *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.
`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.

8

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. These two mechanisms are:

- Processing command-line arguments.
- Parsing configuration files.

This chapter discusses these two topics and describes how to make your genies flexible through configuration. Appendix B on page 403 provides a reference to the commands discussed in this chapter.

Processing 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.

Enhancing the idlgrep Genie

Although the `idlgrep` application ("Processing an IDL File" on page 91) 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.

Processing the Command Line

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`.

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 symbol*, are variable names that are assigned values by the idlgen_getarg command, as described in Table 8.1.

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.

Table: 8.1: idlgen_getarg *Arguments*

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

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 117, 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.

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`.

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.

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

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"
```

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

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
}
```

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 ""
        }
    }
}
```

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`. 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
```

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.

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.

Syntax of an idlgen Configuration File

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 D on page 433.

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
```

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";
```

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;
```

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"];
```

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"];
```

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 B on page 403.

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 124):

```
# 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}]
```

9

Developing a C++ Genie

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.

The `std/cpp_boa_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 Basic Object Adapter (BOA) specification.

The following topics are covered in this chapter:

- Identifiers and keywords.
- C++ prototype.
- Client side: invoking an operation.
- Client side: invoking an attribute.
- Server side: implementing an operation.
- Server side: implementing an attribute.
- Instance variables and local variables.
- Processing a union.
- Processing an array.
- Processing an Any.

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 _.

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*
    _for(
        CORBA::Long _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 9.1. The _s_ variants return fully-scoped identifiers whereas the _l_ variants return non-scoped identifiers.

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.
`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*.

Table: 9.1: *Commands for Generating Identifiers and Keywords*

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*
foo_i::op(
    const widget&                   p_widget,
    char *&                         p_string,
    longSeq*&                       p_longSeq,
    long_array                      p_long_array,
    CORBA::Environment &)
        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;
}
```

Client Side: 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:

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

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

idlgen_set_preferences $idlgen(cfg)

open_output_file "testClt.cxx"

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:

```
// 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 name of the C++ variable that is declared to hold the return value, line 3, 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:

```
# 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:

```
    # 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"
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 211.

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.

```
# 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"} {
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.

```
# 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.

Client Side: Invoking an Attribute

To invoke an IDL attribute, you must perform similar steps to those described in "Client Side: Invoking an Operation" on page 139. 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 140.

Server Side: 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.
3. The function body first processes the `in` and `inout` parameters that it has received from the client.
4. Declare return value and allocate parameter memory.
5. The return value is declared. Memory must be allocated for `out` parameters and the return value.
6. Initialize output parameters and return value.
7. The `inout` and `out` parameters and the return value must be initialized.
8. Manage memory when throwing exceptions.
9. 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
smart_source "std/output.tcl"
smart_source "std/cpp_boa_lib.tcl"

idlgen_set_preferences $idlgen(cfg)
if { ! [idlgen_parse_idl_file "prototype.idl"] } {
    exit 1
}
open_output_file "testSrv.cxx"
set op [$idlgen(root) lookup "foo::op"]
cpp_gen_op_sig_cc $op
```

The previous script generates the following C++ code:

```
// C++
longSeq*
foo_i::op(
        const widget&                   p_widget,
        char *&                         p_string,
        longSeq*&                       p_longSeq,
        long_array                      p_long_array,
        CORBA::Environment &)
            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 143. 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.

```
# 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:

```
# 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 241 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:

```
# 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 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;
}
```

Server Side: 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:

```
# Tcl
set is_var 0
set ind_lev 1
[***
void some_func()
{
        // Declare variables
***]
foreach type $type_list {
        set name  "my_[$type l_name]"
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 241 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 9.2 summarizes the commands used for generating union labels.

Table: 9.2: *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:

```
# 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:

```
// 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
smart_source "std/output.tcl"
smart_source "std/cpp_boa_lib.tcl"

if { ! [idlgen_parse_idl_file "prototype.idl"] } {
    exit 1
}
idlgen_set_preferences $idlgen(cfg)
open_output_file "any_insert.cxx"
lappend type_list [$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]
[***
@[cpp_any_insert_stmt $type "an_any" $var_name]@;
***]
}

close_output_file
```

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

Table 9.3 summarizes the commands that are used to generate code that extracts values from an any.

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

Table: 9.3: *Commands for Generating any Extraction Statements*

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);
}
```

10

Developing a Java Genie

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.

The `std/java_boa_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.

The following topics are covered in this chapter:

- Identifiers and keywords.
- Java prototype.
- Client side: invoking an operation.
- Client side: invoking an attribute.
- Server side: implementing an operation.
- Server side: implementing an attribute.
- Instance variables and local variables.
- Processing a union.
- Processing an array.
- Processing a sequence.
- Processing an Any.

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.

The application programming interface (API) for generating Java identifiers is summarized in Table 10.1. The `_s_` variants return fully-scoped identifiers whereas the `_l_` variants return non-scoped identifiers.

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*.

Table: 10.1: *Commands for Generating Identifiers and Keywords*

Command	Description
`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*.

Table: 10.1: *Commands for Generating Identifiers and Keywords*

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.longSeqHolder         p_longSeq;
Prototype.long_arrayHolder      p_long_array;
int[]                           _return;

//--------
// 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);
process_longSeq(p_longSeq);
process_long_array(p_long_array);
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);

        //--------
        // 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;
    }
```

Client Side: 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:

```
# Tcl
smart_source "std/output.tcl"
smart_source "std/java_boa_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;
```

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:

```
#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:

```
# 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:

```
# Tcl
1   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 211.

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:

```
# 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);
process_longSeq(p_longSeq);
process_long_array(p_long_array);
process_longSeq(_result);
```

Client Side: Invoking an Attribute

To invoke an IDL attribute, you must perform similar steps to those described in "Client Side: Invoking an Operation" on page 172. 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 173.

Server Side: 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_boa_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 178. 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:

```
# 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]@;

***]
}
```

The preceding Tcl script can be explained as follows:

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:

```
# 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]"
        set dir     [$arg direction]
        java_gen_assign_stmt $type $arg_ref $name2 $ind_lev $dir
}
if {[$ret_type l_name] != "void"} {
2       set ret_ref [java_srv_par_ref $op]
        set name2   "other_[$ret_type s_uname]"
        set dir     "return"
        java_gen_assign_stmt $type $ret_ref $name2 $ind_lev $dir
[***
        return @$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;
{
    for (int i1 = 0; i1 < 10  ; i1 ++) {
        p_long_array[i1] = other_long_array[i1];
    }
}
{
    for (int i1 = 0; i1 < 10  ; i1 ++) {
        _return[i1] = other_longSeq[i1];
    }
}
return _result;
```

Server Side: 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:

```
# Tcl
smart_source "std/output.tcl"
smart_source "std/java_boa_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]"
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 10.2 summarizes the commands used for generating `union` labels.

Table: 10.2: *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:

```
# Tcl
smart_source "std/output.tcl"
smart_source "std/java_boa_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:

```
// 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_boa_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 10.3 summarizes the command that is used to generate code that inserts values into an any.

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`*.

Table: 10.3: *Command for Generating any Insertion Statements*

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

```
# Tcl
smart_source "std/output.tcl"
smart_source "std/java_boa_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 10.4 summarizes the commands that are used to generate code that extracts values from an any.

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*.

Table: 10.4: *Commands for Generating any Extraction Statements*

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

```
# Tcl
smart_source "std/output.tcl"
smart_source "std/java_boa_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();
};
```

11

Further Development Issues

This chapter details further development facets of the code generation toolkit that help you to write genies more effectively.

This chapter describes the following topics in detail:

- Global arrays.
- Re-implementing Tcl commands.
- Miscellaneous utility commands.
- Recommended programming style.

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 93, 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 )`.

The $idlgen Array

This array contains entries that are related to the core idlgen executable.

$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 Chapter on page 91.

$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 126.

$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.

The code generation toolkit provides a number of mechanisms to support genie preferences:

- Command line arguments.
- 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.

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 11.1:

$pref(...) Array Entry	Purpose
`$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`.

Table: 11.1: $pref(...) Array Entries

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.

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.

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

12

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

The abbreviations shown in Table 12.1 are used in the names of commands defined in the `std/cpp_boa_lib.tcl` library.

Abbreviation	Meaning
`clt`	Client
`srv`	Server
`var`	Variable
`var_decl`	Variable declaration
`is_var`	See "Naming Conventions for is_var" on page 224
`gen_`	See "Naming Conventions for gen_" on page 225
`par/param`	Parameter

Table: 12.1: *Abbreviations Used in Command Names.*

Abbreviation	Meaning
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.

Table: 12.1: *Abbreviations Used in Command Names.*

Command names in std/cpp_boa_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_boa_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.

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_boa_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 12.2. 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.

$pref(...) Array Entry	Purpose
`$pref(cpp,h_file_ext)`	Specifies the filename extension for header files. Its default value is `.h`.
`$pref(cpp,cc_file_ext)`	Specifies the filename extension for code files. Its default value is `.cxx`.
`$pref(cpp,indent)`	Specifies the amount of white space to be used for one level of indentation. Its default value is four spaces.
`$pref(cpp,impl_class_suffix)`	Specifies the suffix that is added to the name of a class that implements an IDL interface. Its default value is `_i`.
`$pref(cpp,smart_proxy_suffix)`	Specifies the prefix that is added to an IDL interface to give the name of a smart proxy class. Its default value is `smart_`.

Table: 12.2: *$pref(cpp,...) Array Entries*

$pref(...) Array Entry	Purpose
`$pref(cpp,want_throw)`	A boolean value that specifies whether or not the C++ signatures of operations and attributes should have a `throw` clause. Its default value is `true`. It should be set to `false` only if generating C++ code for an old C++ compiler that does not support exceptions.
`$pref(cpp,server_timeout)`	Timeout (milliseconds) passed to `impl_is_ready()` in the generated `server.cxx` file. A value of `-1` represents infinity.
`$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.

Table: 12.2: *$pref(cpp,...) Array Entries*

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_l_name type
cpp_typecode_s_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_boa_class_s_name interface_node
cpp_impl_class interface_node
cpp_tie_class 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_boa_lib Commands

This section gives detailed descriptions of the Tcl commands in the cpp_boa_lib library in alphabetical order.

cpp_any_extract_stmt

cpp_any_extract_stmt *type any_name var_name*

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

cpp_any_extract_var_decl *type name*

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

cpp_any_extract_var_ref *type name*

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

cpp_any_insert_stmt *type any_name value ?is_var?*

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.

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

cpp_array_decl_index_vars *array prefix ind_lev*
cpp_gen_array_decl_index_vars *array prefix ind_lev*

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 Consider the following sample IDL:

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

The following Tcl script illustrates the use of the command:

```
# Tcl
set typedef  [$idlgen(root) lookup "long_array"]
set a        [$typedef true_base_type]
1  set indent   [cpp_indent [$a num_dims]]
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, `2`, is calculated by using the number of dimensions in the array as a parameter to the `cpp_indent` command, `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

`cpp_array_elem_index` *`array prefix`*

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
```

cpp_array_for_loop_footer

```
cpp_array_for_loop_footer array ind_lev
cpp_gen_array_for_loop_footer array ind_lev
```

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

```
cpp_array_for_loop_header array prefix ind_lev ?declare?
cpp_gen_array_for_loop_header array prefix ind_lev ?declare?
```

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

`cpp_assign_stmt` *`type name value ind_lev ?scope?`*
`cpp_gen_assign_stmt` *`type name value ind_lev ?scope?`*

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 { and }) 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

`cpp_attr_acc_sig_h` *`attribute`*
`cpp_gen_attr_acc_sig_h` *`attribute`*

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_boa_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(
    CORBA::Environment &_env=CORBA::IT_chooseDefaultEnv())

    virtual CORBA::Float balance(
        CORBA::Environment &_env=CORBA::IT_chooseDefaultEnv());
```

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

cpp_attr_acc_sig_cc *attribute ?class?*
cpp_gen_attr_acc_sig_cc *attribute ?class?*

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_boa_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
Account_i::balance(
        CORBA::Environment &)

CORBA::Float
Account_i::balance(
        CORBA::Environment &)
```

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

```
cpp_attr_mod_sig_h attribute
cpp_gen_attr_mod_sig_h attribute
```

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_boa_lib.tcl"

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

set attr [$idlgen(root) lookup "Account::balance"]
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,
    CORBA::Environment &_env=CORBA::IT_chooseDefaultEnv())

    virtual void balance(
        CORBA::Float                    _new_value,
        CORBA::Environment &_env=CORBA::IT_chooseDefaultEnv());
```

See Also `cpp_attr_acc_sig_h`
`cpp_attr_acc_sig_cc`
`cpp_attr_mod_sig_cc`

cpp_attr_mod_sig_cc

`cpp_attr_mod_sig_cc` *`attribute ?class?`*
`cpp_gen_attr_mod_sig_cc` *`attribute ?class?`*

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_boa_lib.tcl"

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

set attr [$idlgen(root) lookup "Account::balance"]
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
Account_i::balance(
        CORBA::Float                    _new_value,
        CORBA::Environment &)

void
Account_i::balance(
        CORBA::Float                    _new_value,
        CORBA::Environment &)
```

See Also

```
cpp_attr_acc_sig_h
cpp_attr_acc_sig_cc
cpp_attr_mod_sig_h
cpp_gen_attr_mod_sig_cc
```

cpp_boa_class_l_name

cpp_boa_class_l_name *interface*

This command returns the local name of the BOA 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_boa_class_l_name $inter]@
{
  public:
    @$class@();
};
***]
```

The following interface definitions results in the generation of the corresponding C++ code:.

// IDL
interface Cow {
    ...
};
	// C++
	class Cow_i :
	    public virtual CowBOAImpl
	{
	  public:
	    Cow_i();
	};

// IDL
module Farm {
    interface Cow{
    ...
    };
};
	// C++
	class Farm_Cow_i :
	    public virtual CowBOAImpl
	{
	  public:
	    Farm_Cow_i();
	};

See Also cpp_tie_class
cpp_boa_s_name

cpp_boa_class_s_name

cpp_boa_class_s_name *interface*

This command returns the fully scoped name of the BOA 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_boa_class_s_name $inter]@
{
  public:
    @$class@();
};
***]
```

The following interface definitions results in the generation of the corresponding C++ code:.

`// IDL`
`interface Cow {`
`    ...`
`};`
	`// C++`
	`class Cow_i :`
	`    public virtual CowBOAImpl`
	`{`
	`  public:`
	`    Cow_i();`
	`};`

`// IDL`
`module Farm {`
`    interface Cow{`
`    ...`
`    };`
`};`
	`// C++`
	`class Farm_Cow_i :`
	`    public virtual Farm::CowBOAImpl`
	`{`
	`  public:`
	`    Farm_Cow_i();`
	`};`

See Also `cpp_tie_class`
`cpp_boa_l_name`

cpp_branch_case_l_label

`cpp_branch_case_l_label` *`union_branch`*

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

`cpp_branch_l_label` *`union_branch`*

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

cpp_branch_case_s_label *union_branch*

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

`cpp_branch_s_label` *`union_branch`*

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

```
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
```

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.

```
# 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

`cpp_clt_need_to_free_mem` *`arg is_var`*
`cpp_clt_need_to_free_mem` *`op is_var`*

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

```
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
```

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:

```
//--------
// 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`. 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

```
cpp_clt_par_ref name type direction is_var
cpp_clt_par_ref arg is_var
cpp_clt_par_ref op is_var
```

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:

```
# 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 preceding script can be explained as follows:

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*

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

cpp_gen_array_for_loop_footer *array ind_lev*

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

cpp_gen_array_for_loop_header *array prefix ind_lev ?declare?*

This command is a variant of cpp_array_for_loop_header that prints its result directly to the current output.

cpp_gen_assign_stmt

cpp_gen_assign_stmt *type name value ind_lev ?scope?*

This command is a variant of cpp_assign_stmt that prints its result directly to the current output.

cpp_gen_attr_acc_sig_h

cpp_gen_attr_acc_sig_h *attribute*

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

cpp_gen_attr_acc_sig_cc *attribute ?class?*

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

cpp_gen_attr_mod_sig_h *attribute*

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

cpp_gen_attr_mod_sig_cc *attribute ?class?*

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

```
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
```

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

```
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
```

This command is a variant of `cpp_clt_par_decl` that prints its result directly to the current output.

cpp_gen_op_sig_h

```
cpp_gen_op_sig_h op
cpp_gen_op_sig_h initializer
```

This command is a variant of `cpp_op_sig_h` that prints its result directly to the current output.

cpp_gen_op_sig_cc

```
cpp_gen_op_sig_cc op ?class?
cpp_gen_op_sig_cc initializer ?class?
```

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

```
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*

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

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*

This command is a variant of cpp_srv_par_alloc that prints its result directly to the current output.

cpp_gen_srv_ret_decl

cpp_gen_srv_ret_decl *name type ind_lev ?alloc_mem?*
cpp_gen_srv_ret_decl *op ind_lev ?alloc_mem?*

This command is a variant of cpp_srv_ret_decl that prints its result directly to the current output.

cpp_gen_var_decl

cpp_gen_var_decl *name type is_var ind_lev*

This command is a variant of cpp_var_decl that prints its result directly to the current output.

cpp_gen_var_free_mem_stmt

cpp_gen_var_free_mem_stmt *name type is_var*

This command is a variant of cpp_var_free_mem_stmt that prints its result directly to the current output.

cpp_impl_class

`cpp_impl_class` *`interface`*

This command returns the name of a C++ 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 `'_'` (the namespace is flattened) and appending `$pref(cpp,impl_class_suffix)`, which has the default value `_i`.

Examples Consider the following Tcl script:

```
# Tcl
...
set class [cpp_impl_class $inter]
[***
class @$class@ {
    public:
        @$class@();
};
***]
```

The following interface definitions result in the generation of the corresponding C++ code:

`//IDL`
`interface Cow {`
`  ...`
`};`
	`// C++`
	`class Cow_i {`
	`  public:`
	`  Cow_i();`
	`};`

`//IDL`
`module Farm {`
`  interface Cow {`
`  ...`
`  };`
`};`
	`// C++`
	`class Farm_Cow_i {`
	`  public:`
	`  Farm_Cow_i();`
	`};`

cpp_indent

`cpp_indent` *`ind_lev`*

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

`cpp_is_fixed_size` *`type`*

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_boa_lib.tcl` library.

See Also `cpp_is_keyword`
`cpp_is_var_size`

cpp_is_keyword

```
cpp_is_keyword string
```

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_boa_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

```
cpp_is_var_size type
```

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_boa_lib.tcl` library.

See Also

```
cpp_is_fixed_size
cpp_is_keyword
```

cpp_l_name

cpp_l_name *node*

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 _ (underscore) 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:

```
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

```
cpp_nil_pointer type
```

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`.

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_op_sig_h

```
cpp_op_sig_h op
cpp_gen_op_sig_h op
```

This command generates the signature of an operation for inclusion in `.h` files.

Parameters

op	An operation 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_boa_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,
    CORBA::Environment &_env=CORBA::IT_chooseDefaultEnv())
    throw(CORBA::SystemException)

    virtual void makeDeposit(
        CORBA::Float                    amount,
        CORBA::Environment &_env=CORBA::IT_chooseDefaultEnv())
        throw(CORBA::SystemException);
```

See Also

cpp_gen_op_sig_h
cpp_op_sig_cc

cpp_op_sig_cc

```
cpp_op_sig_cc op ?class?
cpp_gen_op_sig_cc op ?class?
```

This command generates the signature of the operation for inclusion in `.cxx` files.

Parameters

op	An operation 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_boa_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
Account_i::makeDeposit(
    CORBA::Float                    amount,
    CORBA::Environment &)
    throw(CORBA::SystemException)

void
Account_i::makeDeposit(
    CORBA::Float                    amount,
    CORBA::Environment &)
    throw(CORBA::SystemException)
```

See Also

cpp_op_sig_h
cpp_gen_op_sig_cc

cpp_param_sig

cpp_param_sig *name type direction*
cpp_param_sig *arg*

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

`cpp_param_type` *`type direction`*
`cpp_param_type` *`arg`*
`cpp_param_type` *`op`*

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_ret_assign

`cpp_ret_assign` *op*

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.

See Also

```
cpp_assign_stmt
cpp_gen_assign_stmt
```

cpp_s_name

`cpp_s_name` *node*

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

`cpp_s_uname` *node*

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:

//IDL	//C++
foo	foo
m::foo	m_foo
m::for	m_for
unsigned long	unsigned_long
sequence<foo>	anon_sequence_foo

See Also cpp_l_name
cpp_s_name

cpp_sanity_check_idl

cpp_sanity_check_idl

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_boa_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_smart_proxy_class

`cpp_smart_proxy_class` *`interface`*

This command returns a C++ identifier that can be used as the name of a smart proxy class for 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 prefixing `$pref(cpp, smart_proxy_prefix)`, which has the default value `smart_`.

Examples Consider the following Tcl script:

```
# Tcl
...
# Node $inter is already initialized...
set sproxyc [cpp_smart_proxy_class $inter]
set proxyc [cpp_s_name $inter]
[***
class @$sproxyc@ :public virtual @$proxyc@ {
    public:
        @$sproxyc@();
};
***]
```

The following interface definitions result in the generation of the corresponding C++ code.

`//IDL`
`interface Cow {`
`  //...`
`};`
	`// C++`
	`class smart_Cow :public virtual Cow`
	`{`
	`  public:`
	`  smart_Cow();`
	`};`

`//IDL`
`module Farm {`
`  interface Cow {`
`  //...`
`  };`
`};`
	`// C++`
	`class smart_Farm_Cow :public virtual`
	`Farm::Cow {`
	`  public:`
	`  smart_Farm_Cow();`
	`};`

cpp_srv_free_mem_stmt

`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`*

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:

```
# 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

`cpp_srv_need_to_free_mem` *`type direction`*
`cpp_srv_need_to_free_mem` *`arg`*
`cpp_srv_need_to_free_mem` *`op`*

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

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*

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.

```
# 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}] {
        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

```
cpp_srv_par_ref name type direction
cpp_srv_par_ref arg
cpp_srv_par_ref op
```

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:

```
# 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

```
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?
```

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.

```
# 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_tie_class

`cpp_tie_class` *`interface`*

This command returns the name of the BOA tie macro 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_s_name $inter]@_ptr the_tie =
        new @[cpp_tie_class $inter]@(@$class@)(tied_object);
***]
```

If `$inter` is set to the node representing the IDL interface `Cow`, the Tcl code produces the following output:

```
// C++
    Farm_Cow_i* tied_object = new Farm_Cow_i();
    Farm::Cow_ptr the_tie =
        new TIE_Farm_Cow(Farm_Cow_i)(tied_object);
```

See Also `cpp_boa_class_s_name`
`cpp_boa_class_l_name`

cpp_typecode_l_name

`cpp_typecode_l_name` *type*

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:

```
// IDL          // C++
cow             _tc_cow
farm::cow       _tc_cow
long            CORBA::_tc_long
```

See Also cpp_typecode_s_name

cpp_typecode_s_name

cpp_typecode_s_name *type*

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:

```
// IDL          // C++
cow             _tc_cow
farm::cow       farm::_tc_cow
long            CORBA::_tc_long
```

See Also cpp_typecode_l_name

cpp_var_decl

cpp_var_decl *name type is_var*
cpp_gen_var_decl *name type is_var ind_lev*

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

```
cpp_var_free_mem_stmt name type is_var
cpp_gen_var_free_mem_stmt name type is_var
```

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

cpp_var_need_to_free_mem *type is_var*

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

13

Other C++ Utility Libraries

This chapter describes some further Tcl libraries available for use in your genies.

The stand-alone genies `cpp_print.tcl`, `cpp_random.tcl` and `cpp_equal.tcl` are discussed in Chapter 3 "Ready-to-Use Genies for Orbix C++ Edition". Aside from being available as stand-alone genies, `cpp_print.tcl`, `cpp_random.tcl` and `cpp_equal.tcl` also provide libraries of Tcl commands that can be called from within other genies. This chapter discusses the APIs of these libraries.

Tcl API of cpp_print

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

```
# Tcl
smart_source "cpp_boa_print/lib-min.tcl"
```

The minimal API defines the following command:

```
# Tcl
cpp_print_func_name type
```

This command returns the name of the print function for the specified *`type`*.

If you want access to the full API of the cpp_print library, use the following command:

```
# Tcl
smart_source "cpp_boa_print/lib-full.tcl"
```

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

```
# Tcl
gen_cpp_print_func_h
gen_cpp_print_func_cc full_any
```

These commands generate the files it_print_funcs.h and it_print_funcs.cxx, respectively. The *full_any* parameter to gen_cpp_print_func_cc is explained below.

The Orbix runtime system has built-in type codes for the basic IDL types such as long, short, string, and so on. However, by default, the Orbix IDL compiler does *not* generate type codes for user-defined IDL types. Without these type codes, you cannot insert a user-defined type into an any. This is not usually a problem because most CORBA applications do not use either TypeCode or any, and by *not* generating these extra type codes, the IDL compiler reduces unnecessary code for most applications. If you want to write an genie that does insert user-defined IDL types into an any, you must specify the -A command-line option to the IDL compiler so that it will generate the necessary type codes.

Among the functions generated by gen_cpp_print_func_cc are IT_print_any() and IT_print_TypeCode(). When generating these functions, gen_cpp_print_func_cc generates code that uses type codes of user-defined IDL types only if the -A option is to be given to the IDL compiler. The *full_any* parameter must be 1 if the -A option is to be given to the IDL compiler. Otherwise, *full_any* should have the value 0.

Example of Use

The following script illustrates how to use all the API commands of the cpp_print library. Lines marked with * are relevant to the use of the cpp_print library.

```
# Tcl
smart_source "std/sbs_output.tcl"
smart_source "std/cpp_boa_lib.tcl"
*  smart_source "cpp_boa_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 }

#--------
# Generate it_print_funcs.{h,cxx}
#--------
*  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 $pref(cpp,cc_file_ext)
open_output_file "example_func$cc_file_ext"

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

void example_func()
{
      //--------
      // 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

      //--------
      // 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

[***
} // end of example_func()
***]
close_output_file
```

The source code of the C++ genie provides a larger example of the use of the cpp_print library.

Tcl API of cpp_random

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

```
# Tcl
smart_source "cpp_boa_random/lib-min.tcl"
```

The minimal API defines the following commands:

```
# Tcl
cpp_random_assign_stmt type name
cpp_gen_random_assign_stmt type name ind_lev
```

The cpp_random_assign_stmt command returns a string representing a C++ statement that assigns a random value to the variable with the specified type and name. The command cpp_gen_random_assign_stmt outputs the statement at the indentation level specified by ind_lev.

If you want access to the full API of the cpp_random library, use the following command:

```
# Tcl
smart_source "cpp_boa_random/lib-full.tcl"
```

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

```
# Tcl
gen_cpp_random_func_h
gen_cpp_random_func_cc full_any
```

These commands generates the files it_random_funcs.h and it_random_funcs.cxx, respectively. The *full_any* parameter to gen_cpp_print_func_cc must have the value 1 if the -A command-line option is to be given to the IDL compiler. Otherwise, *full_any* should be 0.

Example of Use

The following script illustrates how to use all the API commands of the cpp_random library. This example is an extension of the example shown in the section "TCL API of cpp_print". Lines marked with + are relevant to the use of the cpp_random library, while lines marked with * are relevant to the use of the cpp_print library.

```
              # Tcl
              smart_source "std/sbs_output.tcl"
              smart_source "std/cpp_boa_lib.tcl"
*             smart_source "cpp_boa_print/lib-full.tcl"
+             smart_source "cpp_boa_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,cxx}
              #--------
*             gen_cpp_print_funcs_h
*             gen_cpp_print_funcs_cc 1

              #--------
              # Generate it_random_funcs.{h,cxx}
              #--------
+             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 $pref(cpp,cc_file_ext)
              open_output_file "example_func$cc_file_ext"

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

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

            [***

                  //--------
                  // Assign random values to each variable
                  //--------
            ***]
            foreach type $type_list {
                  set name my_[$type s_uname]
            [***
               @[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

[***
} // end of example_func()
***]
close_output_file
```

The source-code of the C++ genie provides a larger example of the use of the cpp_random library.

Tcl API of cpp_equal

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

```
# Tcl
smart_source "cpp_boa_equal/lib-min.tcl"
```

The minimal API defines the following commands:

```
# Tcl
cpp_equal_expr type name1 name2
cpp_not_equal_expr type name1 name2
```

These commands return a string representing a C++ boolean expression that tests the two specified variables *name1* and *name2* of the same *type* for equality.

Example of Use

An example of the use of cpp_equal_expr and cpp_not_equal_expr is as follows:

```
# Tcl
foreach type [idlgen_list_all_types "exception"] {
      set name1 "my_[$type s_uname]_1";
      set name2 "my_[$type s_uname]_2";
[***
      if (@[cpp_equal_expr $type $name1 $name2]@) {
         cout << "values are equal" << endl;
      }
***]
```

```
}; # foreach type
```

Full API of cpp_equal

If you want access to the *full* API of the cpp_equal library then use the following command:

```
# Tcl
smart_source "cpp_boa_equal/lib-full.tcl"
```

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

```
# Tcl
gen_cpp_equal_func_h
gen_cpp_equal_func_cc full_any
```

These commands generates the files it_equal_funcs.h and it_equal_funcs.cxx, respectively. The *full_any* parameter to gen_cpp_equal_func_cc should be 1 if the -A command-line option is to be given to the IDL compiler. Otherwise, *full_any* should be 0.

Part IV

Java Genies Library Reference

14

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

The abbreviations shown in Table 14.1 are used in the names of commands defined in the `std/java_boa_lib.tcl` library.

Abbreviation	Meaning
`clt`	Client
`srv`	Server
`var`	Variable
`var_decl`	Variable declaration
`gen_`	See "Naming Conventions for gen_" on page 312
`par/param`	Parameter
`ref`	Reference

Table: 14.1: *Abbreviations Used in Command Names.*

Abbreviation	Meaning
stmt	Statement
op	Operation
attr_acc	An IDL attribute's accessor
attr_mod	An IDL attribute's modifier
sig	Signature

Table: 14.1: *Abbreviations Used in Command Names.*

Command names in std/java_boa_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.

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_boa_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 14.2. 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.

$pref(...) Array Entry	Purpose
`$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,smart_proxy_prefix)`	Specifies the prefix that is added to an IDL interface to give the name of a smart proxy class. Its default value is `Smart`.
`$pref(java,attr_mod_param_name)`	Specifies the name of the parameter in the Java signature of an attribute's modifier operation. Its default value is `_new_value`.
`$pref(java,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.

Table: 14.2: *$pref(java,...) Array Entries*

$pref(...) Array Entry	Purpose
`$pref(java,want_throw)`	A boolean value that specifies whether or not the Java signatures of operations and attributes should have a `throw` clause. Its default value is `true`.

Table: 14.2: *$pref(java,...) Array Entries*

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_l_name type
java_typecode_s_name type
```

General Purpose Commands

```
java_assign_stmt type name value ?dir? ?scope?
java_indent number
java_is_keyword name
```

Servant/Implementation Classes

```
java_boa_class_s_name interface_node
java_impl_class interface_node
java_tie_class 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 ?dir? ?scope?
java_clt_par_decl name type dir
java_clt_par_ref arg_or_op
java_gen_clt_par_decl arg_or_op ind_lev
java_ret_assign op
```

Invoking Attributes

```
java_clt_par_decl name type dir
java_clt_par_ref name type dir
java_gen_clt_par_decl name type dir 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 ?dir?
```

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
```

java_boa_lib Commands

This section gives detailed descriptions of the Tcl commands in the `java_boa_lib` library.

java_any_extract_stmt

`java_any_extract_stmt` *`type any_name var_name`*

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_boa_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();
};
```

See Also

```
java_any_insert_stmt
java_any_extract_var_decl
java_any_extract_var_ref
```

java_any_extract_var_decl

`java_any_extract_var_decl` *`type name`*

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

```
java_any_extract_var_ref type name
```

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

`java_any_insert_stmt` *`type any_name value`*

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_boa_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

```
java_array_decl_index_vars array prefix ind_lev
java_gen_array_decl_index_vars array prefix ind_lev
```

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:

```
# Tcl
smart_source "std/output.tcl"
smart_source "std/java_boa_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 previous 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

`java_array_elem_index` *`array prefix`*

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

```
java_array_for_loop_footer array ind_lev
java_gen_array_for_loop_footer array ind_lev
```

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

```
java_array_for_loop_header array prefix ind_lev ?declare?
java_gen_array_for_loop_header array prefix ind_lev ?declare?
```

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

```
java_assign_stmt type name value ?direction? ?scope?
java_gen_assign_stmt type name value ind_lev ?direction? ?scope?
```

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	(Optional) The parameter passing mode—one of `in`, `inout`, `out`, or `return`.
scope	(Optional) Only affects array assignments. If equal to `1`, the lines of code that make an array assignment are enclosed in curly braces. Otherwise the braces are omitted. The default is `1`.

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_boa_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

`java_assign_stmt_array` *`type name value ind_lev ?scope?`*

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_boa_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

`java_attr_acc_sig` *`attribute`*
`java_gen_attr_acc_sig` *`attribute`*

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_boa_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

```
java_attr_mod_sig attribute
java_gen_attr_mod_sig attribute
```

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_boa_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_boa_class_l_name

`java_boa_class_l_name` *`interface`*

This command returns the local name of the BOA 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_boa_class_l_name $inter]@
{
    //...
};
***]
```

The following interface definitions results in the generation of the corresponding Java code:.

`// IDL`
`interface Cow {`
`    //...`
`};`
	`// Java`
	`public class NoPackage.CowImpl`
	`    extends CowImplBase`
	`{`
	`    //...`
	`};`

`// IDL`
`module Farm {`
`    interface Cow{`
`    //...`
`    };`
`};`
	`// Java`
	`public class NoPackage.Farm.CowImpl`
	`    extends CowImplBase`
	`{`
	`    //...`
	`};`

See Also `java_boa_class_s_name`

java_boa_class_s_name

`java_boa_class_s_name` *`interface`*

This command returns the fully scoped name of the BOA 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_boa_class_s_name $inter]@
{
    //...
};
***]
```

The following interface definitions results in the generation of the corresponding Java code:.

`// IDL`
`interface Cow {`
`    //...`
`};`
	`// Java`
	`public class NoPackage.CowImpl`
	`    extends NoPackage.CowImplBase`
	`{`
	`    //...`
	`};`

`// IDL`
`module Farm {`
`    interface Cow{`
`    //...`
`    };`
`};`
	`// Java`
	`public class NoPackage.Farm.CowImpl`
	`    extends NoPackage.Farm.CowImplBase`
	`{`
	`    //...`
	`};`

See Also `java_boa_class_l_name`

java_branch_case_l_label

`java_branch_case_l_label` *`union_branch`*

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
case red
case green
case default
```

See Also `java_branch_case_s_label`
`java_branch_l_label`
`java_branch_s_label`

java_branch_case_s_label

`java_branch_case_s_label` *`union_branch`*

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

`java_branch_l_label` *`union_branch`*

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

`java_branch_s_label` *`union_branch`*

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

```
java_clt_par_decl name type direction
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
```

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

```
java_clt_par_ref name type direction
java_clt_par_ref arg
java_clt_par_ref op
```

This command returns *name*`.value`, if the parameter *direction* is `inout` or `out` (as is appropriate for `Holder` types). Otherwise it returns *name*.

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`.
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:

```
# 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 $arg]
        set value "other_[$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`*

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

`java_gen_array_for_loop_footer` *`array ind_lev`*

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

`java_gen_array_for_loop_header` *`array prefix ind_lev ?declare?`*

This command is a variant of `java_array_for_loop_header` that prints its result directly to the current output.

java_gen_assign_stmt

`java_gen_assign_stmt` *`type name value ind_lev ?dir? ?scope?`*

This command is a variant of `java_assign_stmt` that prints its result directly to the current output.

java_gen_attr_acc_sig

`java_gen_attr_acc_sig` *`attribute`*

This command is a variant of `java_attr_acc_sig` that prints its result directly to the current output.

java_gen_attr_mod_sig

`java_gen_attr_mod_sig` *`attribute`*

This command is a variant of `java_attr_mod_sig` that prints its result directly to the current output.

java_gen_clt_par_decl

`java_gen_clt_par_decl` *`name type direction ind_lev`*
`java_gen_clt_par_decl` *`arg_or_op ind_lev`*

This command is a variant of `java_clt_par_decl` that prints its result directly to the current output.

java_gen_op_sig

`java_gen_op_sig` *`op`*

This command is a variant of `java_op_sig` that prints its result directly to the current output.

java_gen_srv_par_alloc

```
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
```

This command is a variant of `java_srv_par_alloc` that prints its result directly to the current output.

java_gen_srv_ret_decl

```
java_gen_srv_ret_decl name type ind_lev
```

This command is a variant of `java_srv_ret_decl` that prints its result directly to the current output.

java_gen_var_decl

```
java_gen_var_decl name type direction ind_lev
```

This command is a variant of `java_var_decl` that prints its result directly to the current output.

java_helper_name

```
java_helper_name type
```

This command returns the scoped name of the `Helper` class associated with *type*.

Parameters

type	A type node of the parse tree.

Notes 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();
};
```

Examples of Java identifiers returned by `[java_helper_name $type]` are given in Table 14.3:

Java Name of `$type`	Output from `java_helper_name` Command
`NoPackage.Widget`	`NoPackage.WidgetHelper`
`NoPackage.StringAlias`	`NoPackage.StringAliasHelper`
`NoPackage.Foo`	`NoPackage.FooHelper`

Table: 14.3: *Helper Classes for User-Defined Types*

See Also `java_holder_name`

java_holder_name

`java_holder_name` *type*

This command returns the scoped name of the `Holder` class associated with *type*.

Parameters

type	A type node of the parse tree.

Examples Given the following IDL:

```
//IDL
struct Widget {
    short s;
};

typedef string StringAlias;

interface Foo {
    void dummy();
};
```

Examples of Java identifiers returned by `[java_holder_name $type]` are given in Table 14.4:

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`

Table: 14.4: *Holder Classes for User-Defined Types*

See Also `java_helper_name`

java_impl_class

`java_impl_class` *`interface`*

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.

`//IDL`
`interface Cow {`
`    //...`
`};`
	`// Java`
	`public class NoPackage.CowImpl {`
	`    //...`
	`};`

`//IDL`
`module Farm {`
`    interface Cow {`
`        //...`
` };`
`};`
	`// Java`
	`public class NoPackage.Farm.CowImpl`
	`{`
	`    //...`
	`};`

java_indent

`java_indent` *`ind_lev`*

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

`java_is_basic_type` *`type`*

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

`java_is_keyword` *`string`*

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_boa_lib.tcl` library.

Examples For example:

```
# Tcl
java_is_keyword “new”; # returns 1
java_is_keyword “cow”; # returns 0
```

java_list_recursive_member_types

`java_list_recursive_member_types`

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_boa_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

`java_l_name` *node*

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:

```
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

```
java_op_sig op
java_gen_op_sig op
```

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_boa_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

`java_package_name` *`node`*

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

`java_param_sig` *`name type direction`*
`java_param_sig` *`arg`*

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

```
java_param_type type direction
java_param_type arg
java_param_type op
```

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_tie_class

`java_tie_class` *`interface`*

This command returns the local name of the BOA 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
smart_source "std/output.tcl"
smart_source "std/java_boa_lib.tcl"

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

open_output_file "tie_class.java"

set inter [$idlgen(root) lookup "Cow"]

set class [java_impl_class $inter]
[***
    @$class@ tied_object = new @$class@();
    @[java_s_name $inter]@ the_tie = new @[java_tie_class
$inter]@(tied_object);
***]

close_output_file
```

If `$inter` is set to the node representing the IDL interface, `Cow`, the Tcl code produces the following output:

```
// Java
    NoPackage.CowImpl tied_object = new NoPackage.CowImpl();
    NoPackage.Cow the_tie = new _tie_Cow(tied_object);
```

See Also `java_scoped_tie_class`

java_scoped_tie_class

`java_scoped_tie_class` *`interface`*

This command returns the scoped name of the BOA 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
smart_source "std/output.tcl"
smart_source "std/java_boa_lib.tcl"

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

open_output_file "tie_class.java"

set inter [$idlgen(root) lookup "Cow"]

set class [java_impl_class $inter]
[***
    @$class@ tied_object = new @$class@();
    @[java_s_name $inter]@ the_tie
        = new @[java_scoped_tie_class $inter]@(tied_object);
***]

close_output_file
```

If `$inter` is set to the node representing the IDL interface, `Cow`, the Tcl code produces the following output:

```
// Java
    NoPackage.CowImpl tied_object = new NoPackage.CowImpl();
    NoPackage.Cow the_tie = new NoPackage._tie_Cow(tied_object);
```

See Also `java_tie_class`

java_ret_assign

`java_ret_assign` *op*

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

`java_s_name` *node*

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

`java_s_uname` *node*

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`:

```
//IDL                 //Java
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

`java_sequence_elem_index` *`seq prefix`*

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

`java_sequence_for_loop_footer` *`seq ind_lev`*

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

`java_sequence_for_loop_header` *`seq prefix ind_lev ?declare?`*

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_smart_proxy_class

`java_smart_proxy_class` *`interface`*

This command returns a Java identifier that can be used as the name of a smart proxy class for 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 prefixing `$pref(java,smart_proxy_prefix)`, which has the default value `Smart`.

Examples Consider the following Tcl script:

```
# Tcl
set sproxyc    [java_smart_proxy_class $inter]
set proxyc     [java_s_name $inter]
[***
package @[java_package_name $inter]@;
class @$sproxyc@ extends @$proxyc@ {
    public @$sproxyc@() {
        //...
    };
};
***]
```

The following interface definitions result in the generation of the corresponding Java code.

`//IDL`
`interface Cow {`
`  //...`
`};`
	`// Java`
	`package NoPackage;`
	`class SmartCow extends Cow {`
	`  public SmartCow() {`
	`  //...`
	`  };`
	`};`

`//IDL`
`module Farm {`
`  interface Cow {`
`  //...`
`  };`
`};`
	`// Java`
	`package NoPackage.Farm;`
	`class SmartCow extends Farm.Cow{`
	`  public SmartCow(){`
	`  //...`
	`  };`
	`};`

java_srv_par_alloc

```
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
```

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_boa_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

```
java_srv_par_ref name type direction
java_srv_par_ref arg
java_srv_par_ref op
```

This command returns *name*.value, if the *direction* parameter is inout or out (as is appropriate for Holder types). Otherwise it returns *name*.

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

```
java_srv_ret_decl name type
java_gen_srv_ret_decl name type ind_lev
```

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

`java_typecode_l_name` *type*

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:

`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

`java_typecode_s_name` *`type`*

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:

`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

`java_user_defined_type` *`type`*

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

```
java_var_decl name type direction
java_gen_var_decl name type direction ind_lev
```

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`

15

Other Tcl Libraries for Java Utility Functions

This chapter describes some further Tcl libraries available for use in your genies.

The stand-alone genies `java_print.tcl`, `java_random.tcl` and `java_equal.tcl` are discussed in Chapter 3 "Ready-to-Use Genies for Orbix C++ Edition". Aside from being available as stand-alone genies, `java_print.tcl`, `java_random.tcl` and `java_equal.tcl` also provide libraries of Tcl commands that can be called from within other genies. This chapter discusses the APIs of these libraries.

Tcl API of java_print

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

```
# Tcl
smart_source "java_print/lib-min.tcl"
```

The minimal API defines the following command:

```
# Tcl
java_print_func_name type
```

This command returns the name of the print function for the specified *`type`*.

If you want access to the full API of the java_print library then use the following command:

```
# Tcl
smart_source "java_print/lib-full.tcl"
```

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

```
# Tcl
gen_java_print_func full_any
```

This command generates several files.

gen_java_print_func generates the class PrintFuncs.Java in the package Idlgen. All the print functions, such as printany() and printTypeCode(), for the IDL basic types are members of the PrintFuncs.Java class.

In addition to the PrintFuncs.Java class, another Java class is generated for each of the IDL types in your source IDL file. This class is called Print<type name> and contains a method with the same name as the IDL type name. This class is contained in the package Idlgen.<type package name>. For example, the following IDL produces corresponding Java print class:

```
//IDL
module outer{
   interface inner{
      struct mystruct{
         ...
      }
   }
}
```

```
//Java
idlgen.outer.inner.Printmystruct
```

When generating PrintFuncs.Java, gen_java_print_func generates code that uses TypeCodes of user-defined IDL types only if the -A option is to be given to the IDL compiler.

Example of Use

The following script illustrates how to use all the API commands of the java_print library. Lines marked with * are relevant to the usage of the java_print library.

```
# Tcl
smart_source "std/sbs_output.tcl"
smart_source "std/java_boa_lib.tcl"
*   smart_source "java_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 }

#--------
# Generate it_print_funcs.{h,cc}
#--------
*   gen_java_print_funcs 1

#--------
# Generate a file which contains
# calls to the print functions
#--------
set java_file_ext $pref(java,java_file_ext)
open_output_file "example_func$java_file_ext"

set type_list [idlgen_list_all_types "exception"]
[***
package @[java_package_name ""]@
public class Example{
   public static void func() {
      //--------
      // Declare variables of each type
      //--------
***]
foreach type $type_list {
      set name my_[$type s_uname]
[***
```

```
      @[java_var_decl $name $type 1]@;
***]
}; # foreach type

[***

      ... //Initialize variables

      //--------
      // Print out the value of each variable
      //--------
***]
foreach type $type_list {
*     set print_func [java_print_func_name $type]
*     set name my_[$type s_uname]
[***
      System.out.println("@$name@ =");
      @$print_func@(cout, @$name@, 1);

***]
}; # foreach type

[***
   } // end of func()
} //end of class
***]
close_output_file
```

The source code of the Java genie provides a larger example of the use of the `java_print` library.

Tcl API of java_random

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

```
# Tcl
smart_source "java_random/lib-min.tcl"
```

The minimal API defines the following commands:

```
# Tcl
java_random_assign_stmt type name
java_gen_random_assign_stmt type name ind_lev
```

The java_random_assign_stmt command returns a string representing a C++ statement that assigns a random value to the variable with the specified type and name. The command java_gen_random_assign_stmt outputs the statement at the indentation level specified by ind_lev.

If you want access to the full API of the java_random library then use the following command:

```
# Tcl
smart_source "java_random/lib-full.tcl"
```

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

```
# Tcl
gen_java_random_func full_any
```

gen_java_random_func generates the class RandomFuncs.Java in the package Idlgen. All the random functions, such as randomany() and randomTypeCode(), for the IDL basic types are members of the RandomFuncs.Java class.

In addition to the RandomFuncs.Java class, another Java class is generated for each of the IDL types in your source IDL file. This class is called Random<type name> and contains a method with the same name as the IDL type name. This class is contained in the package Idlgen.<type package name>. For example, the following IDL produces corresponding Java print class:

```
//IDL
module outer{
   interface inner{
      struct mystruct{
         ...
      }
   }
}
```

```
//Java
idlgen.outer.inner.Randommystruct
```

Example of Use

The following script illustrates how to use all the API commands of the java_random library. This example is an extension of the example shown in the section "TCL API of java_print". Lines marked with "+" are relevant to the use of the java_random library, while lines marked with "*" are relevant to the use of the java_print library.

```
# Tcl
smart_source "std/sbs_output.tcl"
smart_source "std/java_boa_lib.tcl"
*   smart_source "java_print/lib-full.tcl"
+   smart_source "java_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 PrintFuncs.Java
#--------
*   gen_java_print_funcs 1

#--------
# Generate RandomFuncs.Java
#--------
+   gen_java_random_funcs 1

#--------
# Generate a file which contains
# calls to the print and random functions
#--------
set java_file_ext $pref(java,java_file_ext)
open_output_file "Example$java_file_ext"

set type_list [idlgen_list_all_types "exception"]
[***
```

```
package @[java_package_name ""]@
public class Example{
      public static void func(){

void Example()
{
      //--------
      // Declare variables of each type
      //--------
***]
foreach type $type_list {
      set name my_[$type s_uname]
[***
+     @[java_var_decl $name $type 1]@;
***]
}; # foreach type

[***

      //--------
      // Assign random values to each variable
      //--------
***]
foreach type $type_list {
      set name my_[$type s_uname]
[***
   @[java_random_assign_stmt $type $name]@;
***]
}; # foreach type

[***

      //--------
      // Print out the value of each variable
      //--------
***]
foreach type $type_list {
*     set print_func [java_print_func_name $type]
   set name my_[$type s_uname]
[***
      System.out.println("@$name@ =");
```

```
*                 @$print_func@(cout, @$name@, 1);

***]
}; # foreach type

[***
   } // end of Example()
}
***]
close_output_file
```

The source-code of the C++ genie provides a larger example of the use of the `java_random` library.

Tcl API of java_equal

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

```
# Tcl
smart_source "java_equal/lib-min.tcl"
```

The minimal API defines the following commands:

```
# Tcl
java_equal_expr type name1 name2
java_not_equal_expr type name1 name2
```

These commands return a string representing a Java Boolean expression that tests the two specified variables *name1* and *name2* of the same *type* for equality.

Example of Use

An example of the use of `java_equal_expr` and `java_not_equal_expr` is as follows:

```
# Tcl
foreach type [idlgen_list_all_types "exception"] {
      set name1 "my_[$type s_uname]_1";
      set name2 "my_[$type s_uname]_2";
[***
      if (@[java_equal_expr $type $name1 $name2]@) {
         System.out.println("values are equal");
      }
***]
}; # foreach type
```

Equality Functions

Unlike cpp_print and cpp_random there is no full cpp_equal API. The equality functions used by IDLgen are implemented in a pre-written class called EqualFuncs. This Java class uses Java Reflection (Java's Runtime Type Information System) to perform the comparisons. For example, any two CORBA objects can be compared by calling:

```
// Java
IT_is_eq_object(Object obj1, Object obj2);
```

The methods in this class can only be used for CORBA types as they make assumptions about classes based on the way the IDL compiler generates code.

As the equality functions use Java Reflection they cannot distinguish between the mappings of certain IDL types, for example:

```
//IDL
typedef sequence <long> apples;
typedef sequence <unsigned long> oranges;
```

Both the above typedefs map to a Java `int[]`, so if the Java instance of `apples` and `oranges` contain the same number of elements and the same values the equality functions return `TRUE`. It is the responsibility of the programmer to ensure that the parameters to the equality functions are of the same type.

Appendix A
User's Reference

This appendix presents reference material about all the configuration and usage details for Orbix Code Generation Toolkit and for the genies provided with the Orbix Code Generation Toolkit.

General Configuration Options

Table 15.1 describes the general purpose configuration options available in standard configuration file `idlgen.cfg`.

Configuration Option	Description
`idlgen.install_root`	The IDLgen installation directory.
`idlgen.genie_search_path`	Search order used by the `smart_source` command.
`idlgen.config_dir`	The IDLgen configuration directory.
`idlgen.tmp_dir`	Directory that Orbix Code Generation Toolkit should use when creating temporary files.
`idlgen.builtin_types`	A list of the basic IDL types supported by the genies. In general, this list might be a subset of all the types understood by the IDL parsing engine. See release notes for details.

Table: 15.1: *Configuration File Options*

Configuration Option	Description
default.all.want_diagnostics	Setting for diagnostics: yes: Genies print diagnostic messages. no: Genies stay silent.
default.all.copyright	The copyright notice that appears at the top of all generated files.
default.orbix.install_root	The Orbix C++ Edition installation directory.
default.orbix.version_number	The version of Orbix C++ Edition. Supported values are 2.2, 2.3, 3.0 and 3.3.
default.orbix.is_multi_threaded	Set equal to: yes: for multi-threaded Orbix. no: for single-threaded Orbix. Note: Orbix is multi-threaded on most platforms.
default.orbix_web.install_root	The Orbix Java Edition installation directory.
default.orbix_web.version_number	The version of Orbix Java Edition. Supported values are 2.2, 2.3, 3.0 and 3.3.
default.orbix_web.is_multi_threaded	Set equal to: yes: for multi-threaded Orbix. no: for single-threaded Orbix. Note: Orbix is multi-threaded on most platforms.

Table: 15.1: *Configuration File Options*

Configuration Option	Description
`default.html.file_ext`	File extension preferred by your web browser (`.html` for most platforms).

Table: 15.1: *Configuration File Options*

Configuration Options for C++ Genies

Table 15.2 describes the configuration options specific to C++ genies in the standard configuration file `idlgen.cfg`:

Configuration Option	Purpose
`idlgen.preprocessor.cmd`	Location of a C++ preprocessor. You should not have to change this entry.
`idlgen.preprocessor.args`	Arguments to pass to the preprocessor. You should not have to change this entry.
`idlgen.preprocessor.suppress_lines`	Used internally by `idlgen`. You should not have to change this entry.
`default.cpp_genie.want_boa`	Sets the approach used when writing C++ classes that implement IDL interfaces: yes: Use the BOA approach. no: Use the TIE approach.
`default.cpp_genie.want_this`	Do you want the generated C++ class to have a `_this()` function?

Table: 15.2: *Configuration File Options for C++ Genies*

Configuration Option	Purpose
`default.cpp.compiler`	The C++ compiler that is used in generated makefiles.
`default.cpp.cc_file_ext`	File extension preferred by your C++ compiler (for example, `.cc`, `.cpp`, `.cxx`, 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.smart_proxy_prefix`	Prefix for your C++ classes that implement smart proxies for IDL interfaces.
`default.cpp.server_timeout`	Timeout (milliseconds) passed to `impl_is_ready()` in the generated `server.cxx` file. A value of `-1` represents infinity.
`default.cpp.want_throw`	This allows you to set `throw` clauses on the C++ signatures of IDL operations and attributes. Setting: yes: Your C++ compiler supports exceptions. no: Your C++ compiler does not support exceptions.

Table: 15.2: *Configuration File Options for C++ Genies*

Configuration Option	Purpose
`default.cpp.want_named_env`	This allows the `CORBA::Environment` parameter at the end of operation and attribute signatures to be named. Setting: yes: Named. no: Anonymous.
`default.cpp.max_padding_for_types`	The width (in characters) of the field occupied type names when declaring parameters and variables. The use of padding vertically aligns parameter and variable names.

Table: 15.2: *Configuration File Options for C++ Genies*

Configuration Options for Java Genies

Table 15.2 describes the configuration options specific to Java genies in the standard configuration file `idlgen.cfg`:

Configuration Option	Purpose
`default.java.java_install_dir`	Location of Java compiler. For example: `D:\jdk1.2`.
`default.java.version_number`	Version of the Java compiler referenced by `java_install_dir`. For example: `1.1`, `1.2` or `1.3`.
`default.java.java_file_ext`	File extension preferred by your Java compiler.

Table: 15.3: *Configuration File Options for Java Genies*

Configuration Option	Purpose
`default.java.java_class_ext`	Class name extension preferred by your Java compiler.
`default.java.package_name`	Default top-level package name for classes generated by a Java genie.
`default.java.printpackage_name`	Default package name for generated print utility classes.
`default.java.randompackage_name`	Default package name for generated random value utility classes.
`default.java.equalpackage_name`	Default package name for generated equality testing utility classes.
`default.java.loader_class_name`	Local name of loader class generated by `java_genie.tcl`.
`default.java.serialized_file_ext`	File extension for loaders.
`default.java.serialized_dir`	Directory to store serialized files.
`default.java.server_name`	Default server name.
`default.java.server_timeout`	Timeout (milliseconds) passed to `impl_is_ready()` in the generated `server.java` file. A value of `-1` represents infinity.
`default.java.impl_class_suffix`	Suffix for your Java classes that implement IDL interfaces.
`default.java.smart_proxy_prefix`	Prefix for your Java classes that implement smart proxies for IDL interfaces.

Table: 15.3: *Configuration File Options for Java Genies*

Configuration Option	Purpose
`default.java.smart_proxy_factory_suffix`	Suffix for your Java classes that implement smart proxy factories for 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.want_javadoc_comments`	Controls the generation of JavaDoc comments in generated code: yes: Generate JavaDoc comments. no: Do not generate JavaDoc comments.
`default.java.want_throw_sys_except`	This allows you to set `throw` clauses on the Java signatures of IDL operations and attributes. Setting: yes: Your Java compiler supports exceptions. no: Your Java compiler does not support exceptions.
`default.java.impl_is_ready_timeout`	The timeout value to pass to `impl_is_ready`.
`default.java.final`	Generate final classes and methods.

Table: 15.3: *Configuration File Options for Java Genies*

Configuration Option	Purpose
`default.java.nohangup`	Set to `true` if you want the server to remain alive while a client is connected.
`default.java.appendLog`	Set to `true` if you want the server logs to be appended.

Table: 15.3: *Configuration File Options for Java Genies*

Command Line Usage

This section summarizes the command-line arguments used by the genies bundled with the Orbix Code Generation Toolkit.

stats

```
idlgen stats.tcl [options] [file.idl]+
```

The command line options are:

`-I<directory>`	Passed to preprocessor.
`-D<name>[=value]`	Passed to preprocessor.
`-h`	Prints a help message.
`-include`	Count statistics for `files in #include` statement too.

idl2html

```
idlgen idl2html.tcl [options] [file.idl]+
```

The command line options are:

`-I<directory>`	Passed to preprocessor.
`-D<name>[=value]`	Passed to preprocessor.

`-h`	Prints help message.
`-v`	Verbose mode (default).
`-s`	Silent mode.

Orbix C++ Genies

cpp_genie

```
idlgen cpp_genie.tcl [options] file.idl [interface wildcard]*
```

The command line options are:

`-I<directory>`	Passed to preprocessor.
`-D<name>[=value]`	Passed to preprocessor.
`-h`	Prints help message.
`-v`	Verbose mode (default).
`-s`	Silent mode.
`-dir <directory>`	Put generated files in the specified directory.
`-include`	Process interfaces in files in `#include` statement too.
`-boa`	Use the BOA approach.
`-tie`	Use the TIE approach (opposite of `-boa` option).
`-(no)interface`	Generate implementation of IDL interfaces.
`-(no)smart`	Generate smart proxies for IDL interfaces.
`-(no)loader`	Generate skeleton loader classes.
`-(no)client`	Generate skeleton client class.
`-(no)server`	Generate skeleton server class.
`-(no)makefile`	Generate a Makefile to build all the generated files.

`-file`	(Default.) Distribute object references using the file system. Mutually incompatible with `-ns` and `-bind`.
`-ns`	Distribute object refernces using the CORBA Naming Service. Mutually incompatible with `-file` and `-bind`.
`-bind`	(Deprecated.) Use `_bind()` to create object references on the client side. Mutually incompatible with `-ns` and `-file`.
`-iorloc <directory>`	Specifies the directory where stringified object reference files are stored. This option is used in combination with the `-file` option. Default is `.` (current directory).
`-all`	Shorthand for specifying all of the options: `-interface`, `-client`, `-server`, `-makefile`, `-loader`, and `-smart`.
`-(no)var`	Use `_var` types in the generated code. This is the default.
`-(no)any`	Generate support for `any` and `TypeCode`. The default is not to support these types.
`-(in)complete`	Generate complete applications. This is the default. If incomplete applications are chosen, the client application does not invoke any operations and the server application does not return random values.
`-(no)inherit`	Use inheritance of implementation classes (default).
`-(no)_this`	Generate operation `_this()` in implementation class.

cpp_op

```
idlgen cpp_op.tcl [options] file.idl [operation or attribute
wildcard]*
```

The command line options are:

`-I<directory>`	Passed to preprocessor.
`-D<name>[=value]`	Passed to preprocessor.
`-h`	Prints help message.
`-v`	Verbose mode (default).
`-s`	Silent mode.
`-o file`	Writes the output to the specified file.
`-include`	Process operations and attributes in files in `#include` statements too.
`-(no)var`	Use `_var` types in generated code (default).
`-(in)complete`	Generate bodies of operations and attributes (default).

cpp_print

```
idlgen cpp_print.tcl [options] file.idl
```

The command line options are:

`-I<directory>`	Passed to preprocessor.
`-D<name>[=value]`	Passed to preprocessor.
`-h`	Prints help message.
`-(no)any`	Generate code to support `any` and `TypeCode`. The default is not to generate print functions for these types.
`-dir <directory>`	Put generated files in the specified directory.

cpp_random

```
idlgen cpp_random.tcl [options] file.idl
```

The command line options are:

`-I<directory>`	Passed to preprocessor.
`-D<name>[=value]`	Passed to preprocessor.
`-h`	Prints help message.
`-(no)any`	Generate code to support any and TypeCode. The default is not to generate random functions for these types.
`-dir <directory>`	Put generated files in the specified directory.

cpp_equal

```
idlgen cpp_equal.tcl [options] file.idl
```

The command line options are:

`-I<directory>`	Passed to preprocessor.
`-D<name>[=value]`	Passed to preprocessor.
`-h`	Prints help message.
`-(no)any`	Generate code to support any and TypeCode. The default is not to generate equal functions for these types.
`-dir <directory>`	Put generated files in the specified directory.

java_genie

```
idlgen java_genie.tcl [options] file.idl [interface wildcard]*
```

The command line options are:

`-I<directory>`	Passed to preprocessor.
`-D<name>[=value]`	Passed to preprocessor.
`-h`	Prints help message.
`-v`	Verbose mode (default).
`-s`	Silent mode.
`-jP <package_name>`	Package into which generated files are placed.
`-dir <directory>`	Put generated files in the specified directory.
`-include`	Process interfaces in files in `#include` statement too.
`-boa`	Use the BOA approach.
`-tie`	Use the TIE approach (opposite of `-boa` option).
`-(no)interface`	Generate implementation of IDL interfaces.
`-(no)smart`	Generate smart proxies for IDL interfaces.
`-(no)loader`	Generate skeleton loader classes.
`-(no)client`	Generate skeleton client class.
`-(no)server`	Generate skeleton server class.
`-(no)makefile`	Generate a Makefile to build all the generated files.
`-ns`	Distribute object references using the CORBA Naming Service. Mutually incompatible with `-bind`.

-bind	(Deprecated.) Use _bind() to create object references on the client side. Mutually incompatible with -ns.
-all	Shorthand for specifying all of the options: -interface, -client, -server, -makefile, -loader, and -smart.
-(no)var	Use _var types in the generated code. This is the default.
-(no)any	Generate support for any and TypeCode. The default is not to support these types.
-(in)complete	Generate complete applications. This is the default. If incomplete applications are chosen, the client application does not invoke any operations and the server application does not return random values.
-(no)inherit	Use inheritance of implementation classes (default).
-(no)_this	Generate operation _this() in implementation class.
-target=<OS>	Use in combination with the -makefile switch to generate a makefile for a specific platform. The supported <OS> values are "windows" and "unix". Default is your installation platform.

java_print

```
idlgen java_print.tcl [options] file.idl
```

The command line options are:

-I<directory>	Passed to preprocessor.
-D<name>[=value]	Passed to preprocessor.
-h	Prints help message.

`-(no)any`	Generate code to support `any` and `TypeCode`. The default is not to generate print functions for these types.
`-dir <directory>`	Put generated files in the specified directory.

java_random

```
idlgen java_random.tcl [options] file.idl
```

The command line options are:

`-I<directory>`	Passed to preprocessor.
`-D<name>[=value]`	Passed to preprocessor.
`-h`	Prints help message.
`-dir <directory>`	Put generated files in the specified directory.

Appendix B
Command Library Reference

This appendix presents reference material on all the commands that the Code Generation Toolkit provides in addition to of the standard Tcl interpreter.

File Output API

The following commands provide support for file output.

Location	`std/output.tcl`	For normal output.
	`std/sbs_output.tcl`	For 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.

Example
```
open_output_file "my_code.cpp"
```

See Also `close_output_file`
`output`

close_output_file

Synopsis. `close_output_file`

Description. Closes the currently opened file.

Notes. Throws an exception if there is no currently opened file.

Example. `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 opened file.

Example `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 operations associated with configuration files. These commands are discussed in Chapter 8, "Configuring Genies".

Conventions A pseudo-code notation is used for the operation definitions of the configuration file variable that results in parsing a configuration file:

```
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 the command throws an exception, the text of which indicates the problem. If parsing is successful this command returns a handle to a Tcl object which 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 )
}
```

Notes None.

Example

```
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.

Notes None.

Example
```
$my_cfg_file destroy
```

See Also `idlgen_parse_config_file`

$cfg filename

Synopsis `$cfg filename`

Description Returns the name of the configuration file which was parsed.

Notes None.

Example
```
$my_cfg_file filename
> mycfg.cfg
```

See Also `idlgen_parse_config_file`

list_names

Synopsis `$cfg list_names`

Description Returns a list which 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.

Example
```
puts "[$my_cfg_file filename] contains the following entries..."

foreach name [$my_cfg_file list_names] {
   puts "\t$name"
}
> 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`.

Example

```
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 then 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 was supplied.
- The entry for name exists but is of type `list`.

Example

```
puts [$my_cfg get_string "foo_bar"]
> 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 then 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 was supplied.
- The entry for name exists but is of type `string`.

Example

```
foreach item [$my_cfg get_list my_list] { puts $item }
> 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 entry `name` already exists it is overridden. The updated configuration settings are not written back to the file.

Example

```
$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 overridden. The updated configuration settings are not written back to the file.

Example

```
$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 containing `is_` or `want_` have boolean values. If such an entry has a value other than `0` or `1`, an exception is thrown.

During initialization, Orbix Code Generation Toolkit executes the statement:

`idlgen_set_preferences $idlgen(cfg)`

As such, `default` scoped entries in the Orbix Code Generation Toolkit configuration file is always copied into the `$pref` array.

Example

```
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 Chapter 8, "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 describing which command-line parameters you wish to extract.
`argument (out)`	The command-line argument that was matched on this run of the command.
`parameter (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}
   ...
   ...
}
```

Example

```
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

Appendix C
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.

Location Built-in commands.

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 then error messages are written to standard error and `idlgen_parse_idl_file` returns `0`.

Example

```
# Tcl
if { [idlgen_parse_idl_file "bank.idl" {-DDEBUG}]}{
      puts "parsing succeeded"
} else {
      puts "parsing failed"
}
```

See Also IDL Parse Tree Nodes.

IDL Parse Tree Nodes

This section lists and describes all the possible node types that can be created from parsing an IDL file.

Conventions This section uses the following typographical conventions:

1. 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)
   }
   ```

   Abstract classes are in *italics*.

2. 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`.

   ```
   // IDL                          # Tcl
   module Finance {                puts [$module l_name]
         interface Account {       > Finance
            ...
         };
   };
   ```

Table of Node Types

All the different types of nodes are arranged into an inheritance hierarchy as shown in Figure 15.1:

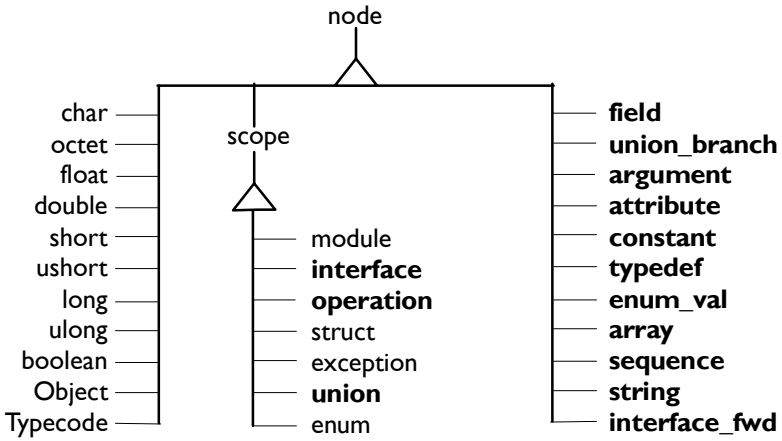

Figure 15.1: *Inheritance Hierarchy for Node Types*

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*.

node

Synopsis. 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.

Definition

```
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()
}
```

`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 "::" replaces with and 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 the discussion in "Typedefs and Anonymous Types" on page 103.
`file`	IDL file which 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`.

Example

```
// 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

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`.

Definition

```
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 = "")
}
```

Methods

`lookup` *`name`*

Get a handle to the named node.

`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 return either true or false. The parameter is the handle to a located node, the function can then return true if it wants that node in the results list or false if it is to be excluded.

`rcontents` *`node_types scope_types [func]`*

Exactly the same as `contents` but also recursively traverses any contained scopes as specified in the *`scope_types`* list. The pseudo-type `all` can be used as a value for the `constructs_wanted` and `recurse_into` parameters of the `contents` and `rcontents` operations.

Example

```
// 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

Synopsis All the built-in IDL types (long, short, string, and so on) are represented by types which inherit from node and do not define any additional operations.

Definition

```
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 Object : node {}
class TypeCode : node {}
class NamedValue : node {}
class Principal : node {}
```

Example

```
// IDL
interface bank {
     void findAccount( in long accNumber, inout branch brchObj );
};
```

```
# Tcl
puts [$long_type l_name]            > long
```

argument

Synopsis An individual argument to an operation.

Definition

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

`type`	The data type of the argument.
`direction`	The passing direction of the argument: `in`, `out` or `inout`.

Example

```
// 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 An anonymous array type.

Definition

```
class array : node {
   node             elem_type()
   list<integer>    dims()
}
```

`elem_type`	The data type of the array.
`dims`	The dimensions of the array.

Example

```
// IDL
module finance {
      typedef long longArray[10][20];
};
```

```
 # Tcl
 set type [$array base_type]
 puts [$type l_name]                 > long
 puts [$array dims]                  > 10 20
 puts [$array l_name]                > longArray
```

attribute

Synopsis An attribute.

Definition

```
class attribute : node {
   boolean     is_readonly()
   node        type()
}
```

`is_readonly`	Determines whether the attribute is read only or not.
`type`	The type of the attribute.

Example

```
// 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
```

constant

Synopsis. A const.

Definition.

```
class constant : node {
   string     value()
   node       type()
}
```

Description

value	The value of the constant.
type	The data type of the constant.

Example

```
// 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 A single entry in an enumeration.

Definition

```
class enum_val : node {
   string     value()
   string     type()
}
```

value	The value of the enumerated entry.
type	A name given to the whole enumeration.

Example

```
// 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 The enumeration.

Definition

```
class enum : scope {
}
```

Example

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

 # Tcl
 puts [$enum s_name]                 > colour
```

exception

Synopsis An exception.

Definition

```
class exception : scope {
}
```

Example

```
// IDL
module finance{
      exception noFunds {
         string   reason;
         float    amountExceeded;
      };
};

 # Tcl
 puts [$exception l_name]            > noFunds
```

field

Synopsis. A field is an item inside an exception or structure.

Definition

```
class field : node {
   node     type()
}
```

`type` The type of the field.

Example

```
// IDL
struct cardNumber {
      long binNumber;
      long accountNumber;
};
```

```
# Tcl
set type [$field type]
puts [$type l_name]                 > long
puts [$field l_name]                > binNumber
```

interface

Synopsis An interface.

Definition

```
class interface : scope {
   list<node>    inherits()
   list<node>    ancestors()
   list<node>    acontents(
                       list<string> constructs_wanted
                       function filter_func = "" )
}
```

Description

`inherits`	The list of interfaces this one derives from.
`ancestors`	The list of all the interfaces that are ancestors of this one.
`acontents`	Like the normal `scope::contents` command but searches ancestor interfaces as well.

Notes An interface is an ancestor of itself.

Example

```
// IDL
module finance {
   interface bank {
      ...
   };
};
```

```
# Tcl
puts [$interface l_name]            > bank
```

interface_fwd

Synopsis A forward declaration of an interface.

Definition

```
class interface : node {
   node     full_definition()
}
```

`full_definition` The actual interface.

Example

```
// IDL
interface bank;
...
interface bank {
   account findAccount( in string accountNumber );
};
```

```
# Tcl
set interface [$interface_fwd full_definition]
set operation [$interface lookup "findAccount"]
puts [ $operation l_name ]                    > findAccount
```

module

Synopsis A module.

Definition

```
class module : scope {
}
```

Example

```
// IDL
module finance {
   interface bank {
      ...
   };
};
```

```
# Tcl
puts [$module l_name]              > finance
```

operation

Synopsis An interface operation.

Definition

```
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 = "")
}
```

return_type	The return type of the operation.
is_oneway	Determines whether the operation is a oneway or not.
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 which 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.

Example

```
// 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 An anonymous sequence.

Definition

```
class sequence : node {
   node        elem_type()
   integer     max_size()
}
```

`elem_type` The type of the sequence.

`max_size` The maximum size, if the sequence is bounded. Otherwise the value is 0.

Example

```
// 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>
```

string

Synopsis A bounded or unbounded string data type.

Definition

```
class string : node {
   integer    max_size()
}
```

`max_size` The maximum size if the string is bounded. Otherwise the value is 0.

Example

```
// 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. A structure.

Definition.
```
class struct : scope {
}
```

Example
```
// IDL
module finance {
   struct branchCode
   {
      string   cateogory;
      long     zoneCode;
   };
};
```

```
 # Tcl
 puts [$structure s_name]            > finance::branchCode
```

typedef

Synopsis A type definition.

Definition
```
class typedef : node {
   node     base_type()
}
```
base_type The data type of the typedef.

Example
```
// IDL
module finance
{
   typedef sequence<account, 100> bankAccounts;
};
```

```
 # Tcl
 set $sequence [$typedef base_type]
 puts [$sequence max_size]           > 100
 puts [$typedef l_name]              > bankAccounts
```

union

Synopsis. A union.

Definition

```
class union : scope {
   node     disc_type()
}
```

`disc_type` The data type of the discriminant.

Example

```
// 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 A single branch in a union.

Definition

```
class union_branch : node {
   string         l_label()
   string         s_label()
   string         s_label_list()
   string         type()
}
```

`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.

Example

```
// 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
```

Appendix D
Configuration File Grammar

This appendix summarizes the syntax of the 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.

Index

Symbols

A

D

E

F

G

M

N

O

P

R

S

T

U

V

W