# Orbix Code Generation Toolkit Programmer's Guide

**IONA Technologies PLC**
**February 1999**

**M 2 2 4 4**

SunSoft, Inc.

2550 Garcia Avenue

Mountain View, California 94043

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 4 OrbixWeb Genies: Library Reference

# Appendices

# Preface

The Orbix Code Generation Toolkit is a flexible development tool that increases programmer productivity by automating many repetitive coding tasks. Out of the box, it is immediately useful for programmers who use Orbix, IONA Technologies' implementations of the Object Management Group's (OMG) Common Object Request Broker Architecture (CORBA).

The Orbix Code Generation Toolkit contains an IDL parser called *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 libraries of useful commands so that you can develop your own genies.

## Audience

This guide is intended for programmers of Orbix or OrbixWeb based applications or anyone who uses the CORBA Interface Definition Language (IDL). Before reading this guide, you should be familiar with the overview of the CORBA IDL as presented in the chapter "Introduction to CORBA IDL" in the *Orbix C++ Programmer's Guide* or the *OrbixWeb Programmer's Guide*. This guide also assumes a familiarity with C++ or Java.

# Organization of this Guide

This guide is divided into five parts:

## Part 1 Using the Orbix Code Generation 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 bundled with the product and details the ready-to-run genies that allow to produce C++ and Java code straight out of the box.

## Part 2 Developing Genies with the Orbix Code Generation Toolkit

This section of the guide takes an in depth look at the Orbix Code Generation Toolkit and teaches you how to develop your own genies so that you can, if you want, expand on the usefulness of the Orbix Code Generation Toolkit in a way tailored to your own specific needs.

## Part 3 Orbix C++ Genies: Library Reference

This section of the guide provides a comprehensive reference to the library commands that you can use in your genies when you want to produce C++ code from CORBA IDL files.

## Part 4 OrbixWeb Genies: Library Reference

This section of the guide provides a comprehensive reference to the library commands that you can use in your genies when you want to produce Java code from CORBA 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.<br><br>Constant width paragraphs represent code examples or information a system displays on the screen. For example:<br><br>`#include <iostream.h>` |
| **`Constant width (bold)`** | Constant width (courier font) in bold text represents portions of code from Tcl bilingual files. See "What are Bilingual Files?" on page 76. |
| *Italic* | Italic words in normal text represent *emphasis* and *new terms*.<br><br>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:<br><br>`% 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 95 command prompt. |

| | |
|---|---|
| ...<br>.<br>.<br>. | 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 IDLgen on both UNIX and Windows platforms.

# Part 1

# Using the Orbix Code Generation Toolkit

# 1

# Overview of the Code Generation Toolkit

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

You can use the Orbix Code Generation Toolkit to write your own code generation scripts, or genies. These can, for example, generate C++ or Java code from an IDL file, or translate an IDL file into another format, such as HTML, RTF or LaTeX.

Several ready-to-run genies are bundled with the Orbix Code Generation Toolkit. These bundled genies can, among other things, generate code from an IDL file for the Orbix products and so can dramatically reduce development time.

## Architecture

As shown in the Figure 1.2, an IDL compiler typically contains three sub-components. A *parser* processes an input IDL file and constructs an in-memory representation called a *parse tree*. A *back-end* then traverses this parse tree and generates, say, C++ or Java stub-code.

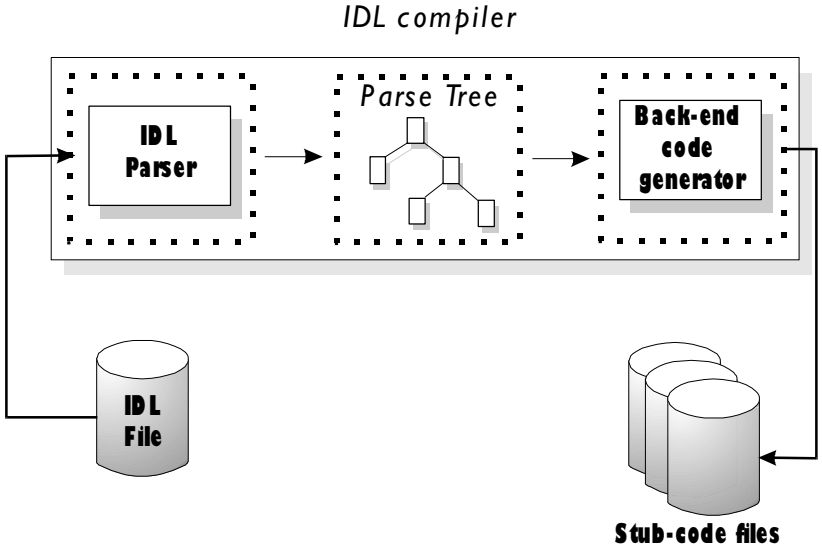


**Figure 1.1:** *Standard IDL Compiler Components*

## IDLgen and Genies

At the heart of the Orbix Code Generation Toolkit is an executable called *IDLgen* which employs a variation of this architecture. The IDLgen executable uses an IDL parser and parse tree, but instead of having a back-end which generates stub-code, the back-end is an interpreter for a scripting language called Tcl. The core Tcl interpreter provides the normal features of a language such as flow-control statements, variables and procedures.

As shown in Figure 1.1, the Tcl interpreter inside IDLgen has been extended so that the IDL parser and parse tree can be manipulated with Tcl commands. This makes it possible to implement a customized back-end as a Tcl script. These customized back-ends are called *genies* (which is short for "code **gen**eration scripts").

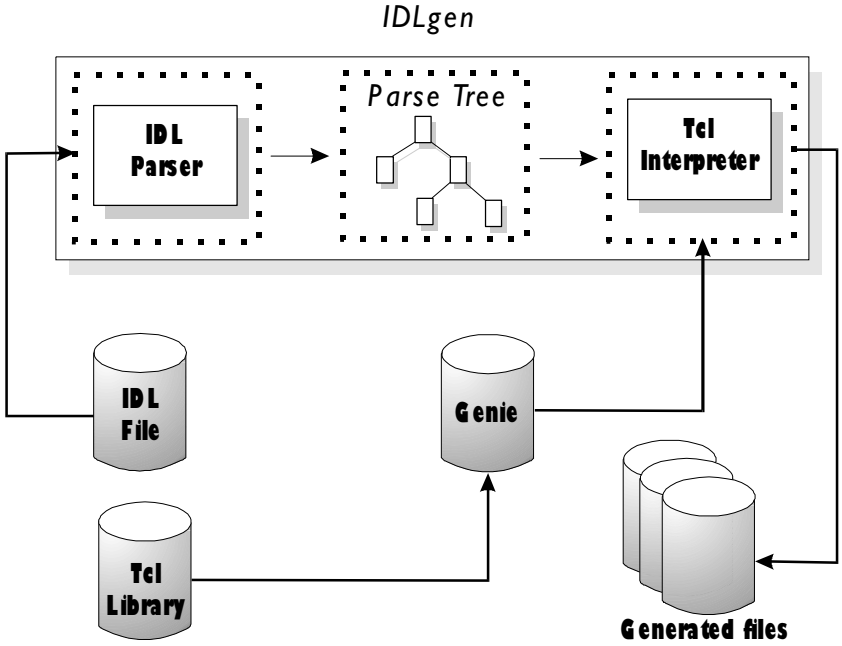


**Figure 1.2:** *Code Generation Toolkit Components*

## 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

IDLgen comes with a number of bundled genies that can be fed into IDLgen 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++".

**Note:** The bundled genies can be used straight away and a user does not need to know anything about Tcl or programming in Tcl to use them.

# Approaches to Using the Code Generation Toolkit

The Code Generation Toolkit is a powerful addition to any CORBA programmer'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.

There are therefore two approaches to using Code Generation Toolkit: development and non-development. This guide is split into two corresponding parts:

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

The second part is only applicable if you wish to extend the bundled genies or write your own.

# 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` or `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;
        sequence<tree>  children;
};

typedef sequence< sequence<long> > longSeqSeq;

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

**Line 1**

**Line 2**

**Line 3**

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.*

IDLgen comes with a collection of genies that can accomplish a number of diverse 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 with IDLgen

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 installation directory of IDLgen.
4. The demos directory under the installation directory of IDLgen.

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_print.tcl   idl2html.tcl
cpp_op.tcl      cpp_random.tcl  stats.tcl
```

The -list option is useful if you cannot remember the name of an genie you want to run. You can also pass a filter to the list command, to filter the available genies. The filtering parameter ensures that only the genies whose names contain the given text are shown.

To show all the genies whose names contain the text cpp use the list command in this way:

```
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
```

The equivalent in silent mode is:

```
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` then IDLgen defaults to verbose mode. If this is set to `no` then IDLgen defaults to silent mode.

# What are the Bundled Genies?

The genies that are bundled with IDLgen 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 Applications**

| | |
|---|---|
| `cpp_genie.tcl` | Generates code for Orbix from an IDL file. |
| `cpp_op.tcl` | Generates code for new operations from an IDL interface. |
| `cpp_random.tcl` | Creates a number of functions that generate random values for all the types present in an IDL file. |
| `cpp_print.tcl` | Creates a number of 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. |

**OrbixWeb Specific Applications**

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

This chapter describes the demo genies. Chapter 3, "Ready-to-use Genies for Orbix C++" discusses the Orbix-specific genies. Chapter 5"Ready-to-use Genies for OrbixWeb"discusses the OrbixWeb-specific genies.

For a full reference to all the genies please refer to Appendix A on page 251. This describes all the configuration and command line options that are available.

# Demonstration Genies

Two demonstration genies are shipped with IDLgen:

- `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 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++

*The 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 which eliminates this task. If you have an IDL file that defines the interfaces for your new project then the C++ genie can generate a demonstration, client-server genie that contains all the starting point code that you are likely to need for your project. In just a few seconds, the C++ genie can give your project a kickstart, and make you productive immediately.

## Generating a Complete 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 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.cpp
idlgen: creating bank_i.h
idlgen: creating bank_i.cpp
idlgen: creating smart_account.h
idlgen: creating smart_account.cpp
idlgen: creating smart_bank.h
idlgen: creating smart_bank.cpp
idlgen: creating loader.h
idlgen: creating loader.cpp
idlgen: creating server.cpp
idlgen: creating client.cpp
idlgen: creating call_funcs.h
idlgen: creating call_funcs.cpp
idlgen: creating it_print_funcs.h
idlgen: creating it_print_funcs.cpp
idlgen: creating it_random_funcs.h
idlgen: creating it_random_funcs.cpp
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.

To compile this application, ensure there is an Orbix daemon running and issue the following commands:

```
% make[1]
% make putit
% client server_hostname
```

1. If you are using Microsoft Windows use `nmake` instead of `make`.

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.
- Programmers can examine the generated code to see examples of how to initialize and pass parameters, and how to perform memory management of various IDL data types.
- A starting point for a programmer's own 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 Orbix 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 application in this way:

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

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

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 and perform memory management). Specifying the name of an interface (`account` in the above example) causes the genie to consider only that interface when generating code.

The above command generates files `account_i.h` and `account_i.cpp` that provide a skeleton class called `account_i` 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:

```
// 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())
   ...
};
```

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 cpp_genie.tcl bank.idl -all -nosmart
        -noloader -nomakefile -noclient -noserver
```

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 C++ genie generates a complete application by generating different files, such as a client mainline (`client.cpp`), server mainline (`server.cpp`), 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 3.1 describes the various command line options:

| Command line option | Purpose |
|---|---|
| `-interface` | Generates the classes that implement the interfaces in the IDL. |
| `-smart` | Generates smart proxy classes. |
| `-loader` | Generates a single loader class for all the interfaces in an IDL. |
| `-server` | Generate a simple server mainline. |
| `-client` | Generate a simple client application. |
| `-incomplete` | Generates skeletal clients and servers. |
| `-makefile` | Generates a Makefile that can build the server and client applications. |

**Table: 3.1:** *C++ Genie Command Line Options*

These command line arguments are detailed in the following sections.

## -interface: Classes that Implement Interfaces

You can generate the classes that implement the interfaces in an IDL file by 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 interface `account` that appears in the `bank.idl` file. The `account` interface is implemented by a class of the same name but suffixed by `_i`.The suffix is specified by the `default.cpp.impl_class_suffix` setting in the `idlgen.cfg` configuration file. The `account_i` 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 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, an 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 (which is 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 `-nothis` command-line option can be used to suppress the generation of the `_this()` operation.

Another related matter is how the constructors of a class that implements an interface 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, then 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.

Again, consider the interface `account` 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.cpp` file:

```
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.cpp` 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 Mainline

Use the `-server` option to generate a simple server mainline:

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

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

```
// C++
int main(int argc, char **argv)
{
   loader          *srvLoader;
   account_var     obj1;
   bank_var        obj2;

   CORBA::Orbix.setDiagnostics(1);

   try {
      CORBA::Orbix.impl_is_ready("bankSrv", 0);
   } catch(CORBA::SystemException &ex) {
      cerr << "impl_is_ready() failed" << endl
           << ex << endl;
      exit(1);
   }

   obj1 = account_i::create("account-1");
   obj2 = bank_i::create("bank-1");

   try {
      CORBA::Orbix.processEvents();
   } catch(CORBA::SystemException &ex) {
      cerr << "processEvents() failed" << endl
```

```
            << ex << endl;
      exit(1);
   }

   return 0;
};
```

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 = new loader(1);
obj1 = account_i::create("account-1",srvLoader);
obj2 = bank_i::create("bank-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.cpp` with a simple `main()`. The client first binds to all of the objects in the server (one bind per interface that appears in the IDL file). It then calls every operation and attribute `get()` and `set()` method with random values for parameters.

The client source file is of the form:

```
// C++
int main(int argc, char **argv)
{
   account_var     obj1;
   bank_var        obj2;

   parse_cmd_line_args(argc, argv);

   CORBA::Orbix.setDiagnostics(1);

   try {
      obj1 = account::_bind(
                  "marker-1:bankSrv", host);
      obj2 = bank::_bind("
                  marker-2:bankSrv", host);
```

```
    } catch(CORBA::SystemException &ex) {
        cerr    << "_bind() failed" << endl
        << ex << endl;
        exit(1);
    }

    call_account_get_balance(obj1);
    call_account_makeLodgement(obj1);
    call_account_makeWithdrawal(obj1);
    call_bank_newAccount(obj2);
    call_bank_deleteAccount(obj2);

    return 0;
}
```

## -incomplete: Skeletal Clients and Servers

If the `-client` option is specified then, by default, the C++ genie generates a file called `call_funcs.cpp` which contains functions to invoke all the operations and attributes of objects in the server. These functions assign random values to the parameters of operations. They also print out the values of parameters that they send (and those that are received back as `out` parameters). Utility functions to assign random values to IDL type are generated in the file `it_random_funcs.cpp`, and utility functions to print the values of IDL type are generated in the file `it_print_funcs.cpp`.

Likewise, if the `-interface` option is specified then, by default, the C++ genie generates bodies of operations and attributes which print the values of `in` and `out` parameters, and also assign random values for the `out` parameters.

These bodies of the generated server-side operations and the client-side calling functions mean that the C++ genie can produce a complete application which can be compiled and run straight away. This is very useful for quickly producing a demo or proof-of-concept prototype. However, it also serves another useful purpose: the generated code provides a working example of how to initialize parameters (albeit with random values), invoke operations, throw and catch exceptions, and perform memory management.

If you do not want the C++ genie to generate the bodies of operations, attributes or the client-side calling functions then you can use the `-incomplete` command-line option.

### -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.

## A Few Other Options

There are a number of other miscellaneous command line arguments that may come in useful. These are:

| Command line argument | Purpose |
|---|---|
| `-(no)var` | The default behavior in the generated code is to use `_var` types whenever possible. This can be turned off by using the `-novar` option. Using the `_var` types make memory management easier. |
| `-(no)any` | By default, the C++ genie does not generate code to support the use of `any` or `TypeCode` for user-defined types. This support can be turned on by using the `-any` command-line option. |

**Table: 3.2:** *Miscellaneous Command Line Arguments*

**Note:** For a full list of the command line options for the Orbix C++ Genie please refer to the Appendix, under the section "User's Reference" on page 251.

# Generating Signatures of Individual Operations with cpp_op.tcl

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 in this way:

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

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

As this example shows, you can use wildcards to specify the names of operations or attributes. If you do not explicitly specify any operations or attributes then 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.

# Creating Print Functions for IDL Types with cpp_print.tcl

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.cpp
```

The names of the generated files are always `it_print_funcs.{h,cpp}`, 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 these 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 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 function for the `EmployeeDetailsSeq` type has the following signature:

```
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:

```
#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.

# Creating Random Functions for IDL Types with cpp_random.tcl

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.cpp
```

The names of the generated files are always `it_random_funcs.{h, cpp}`, 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:

```
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:

```
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:

```
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:

```
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.

# Creating Equality Functions for IDL Types with cpp_equal.tcl

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:

```
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.cpp
```

The names of the generated files are always `it_equal_funcs.{h,cpp}`, 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:

```
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 on page 251.

# 4

# 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++" 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 4.1.
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 **Advanced** if you want to set some advanced options.
7. Select **Next** to continue the process.

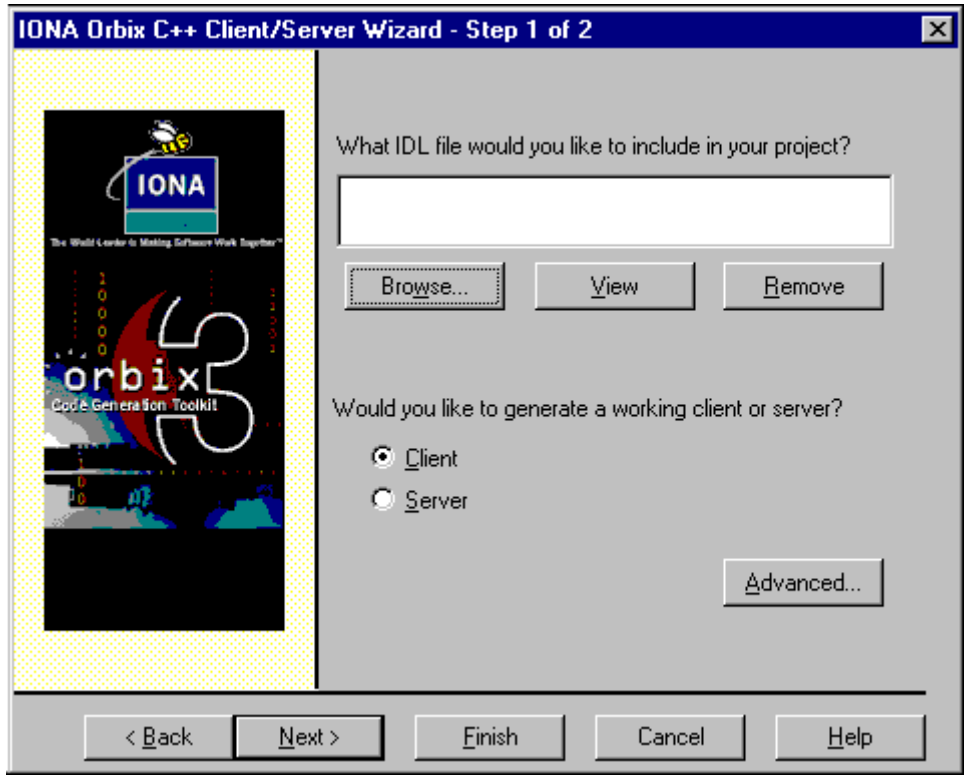


**Figure 4.1:** *Client/Server Wizard: Browsing for IDL Files*

## Advanced Code Generation Options

When you select **Advanced** from the window shown in Figure 4.1, the **Advanced Code Generation Options** window is displayed, as shown in Figure 4.2:

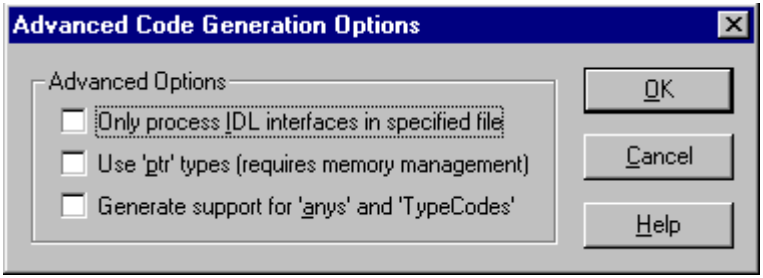


**Figure 4.2:** *Advanced Code Options Window*

You can set three options, which are described in Table 4.1:

| Advanced Code Option | Effect |
|---|---|
| `Only process IDL interfaces in specified file` | By default, the wizard generates code for IDL interfaces fo 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: 4.1:** *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 the `_var` type. The `_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 constructs utilize the any CORBA data type, then you should turn this switch on to instruct the Orbix IDL compiler to generate helper types and methods to allow you to insert and extract your complex types into and out of an any. |

**Table: 4.1:** *Advanced Code Options*

## Generating Client Code

When you generate an Orbix C++ client, you are presented with the window shown in Figure 4.3. You can choose to generate Smart Proxy code for your client by selecting **Generate Smart Proxies**.

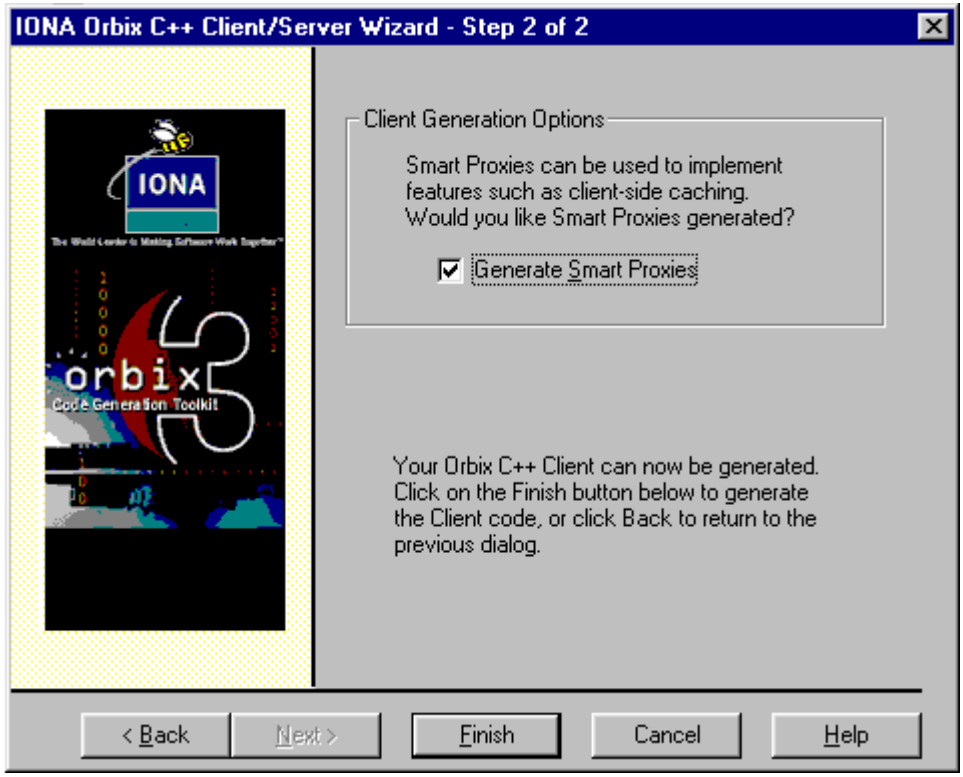


**Figure 4.3:** *Client Generation Options*

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 4.4. You have several options to tailor the generated server code.

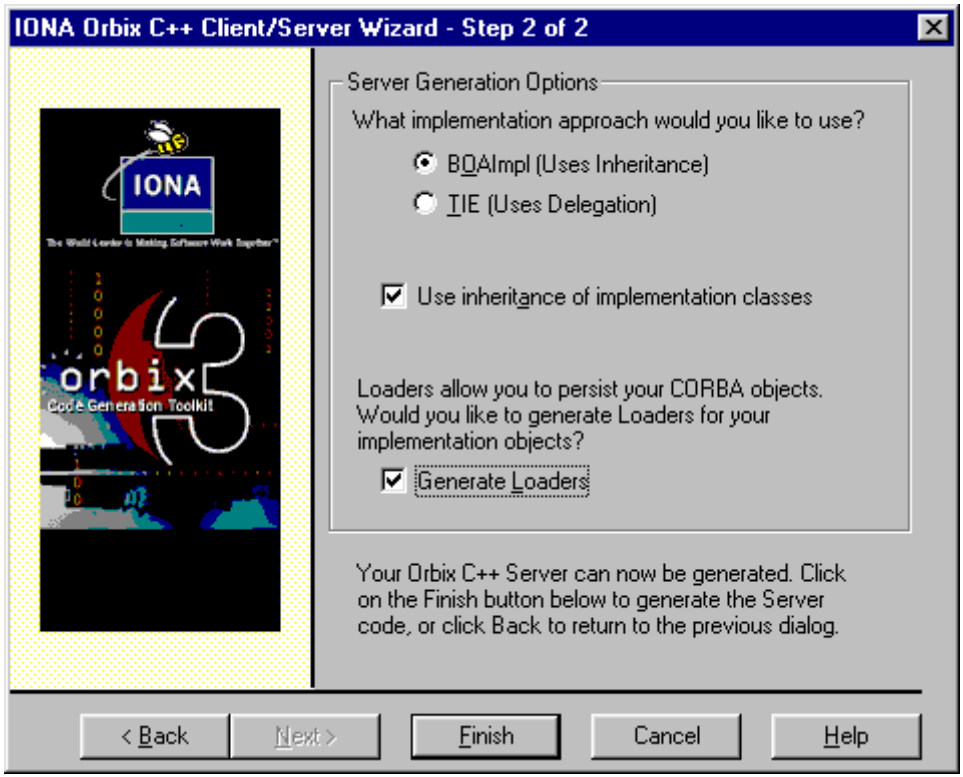


**Figure 4.4:** *Server Generation Options*

By default, your implementation objects will use the **BOAImpl** approach to associate them with their 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 interface `CheckingAccount` (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 very useful for serializing (reading or writing) your objects to and from files or a database. The loader that is generated simply delegates the "Loading" and "Saving" process to the actual 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

Once you have made your decisions about how to build your client or server code files, you can sit back and watch the Wizard generate your files, as shown in Figure 4.5:

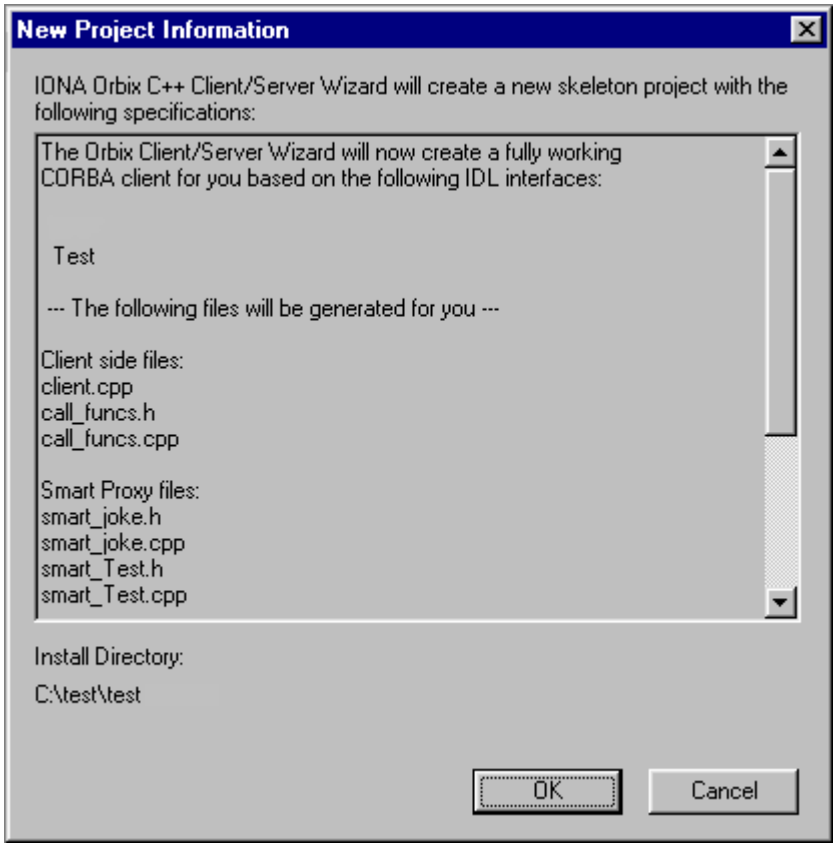


**Figure 4.5:** *Building a C++ Project with the Wizard*

Once this process is complete, your C++ files are available as a Visual Studio project, where you can work with them as you choose, or just go ahead and build a final application by selecting **Build→Build.**

# 5

# Ready-to-use Genies for OrbixWeb

*The Code Generation Toolkit is packaged with several genies for use with IONA's product OrbixWeb which maps CORBA 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 Complete 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: myPackage\
idlgen: creating accountImpl.java
idlgen: creating bankImpl.java
idlgen: creating Smartaccount.java
idlgen: creating Smartbank.java
idlgen: creating loader.java
idlgen: creating server.java
idlgen: creating client.java
idlgen: creating accountCaller.java
idlgen: creating bankCaller.java
idlgen: creating PrintFuncs.java
idlgen: creating RandomFuncs.java
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.

To compile this application, ensure there is an OrbixWeb daemon running and issue the following commands:

```
make[1]
make putit
client server_hostname
```

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:

1. If you are running IDLgen on Windows use `nmake` instead of `make`, or use the automatically generated batch file.

- Demonstrating or testing an OrbixWeb client/server application for a particular interface or interfaces.
- Programmers can examine the generated code to see examples of how to initialize and pass parameters.
- A starting point for a programmer's own 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 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 above 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 extends accountImplBase
{
...
   public void makeLodgement(
      float f,
         throw(org.omg.CORBA.SystemException);

   public void makeWithdrawal(
      float f,
         throw(org.omg.CORBA.SystemException);

   public org.omg.CORBA.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 mainline (`client.java`), server mainline (`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 5.1 summarizes the Java genie command-line arguments:

| Command line argument | Purpose |
|---|---|
| `-interface` | Generates the classes that implement the interfaces in the IDL. |
| `-smart` | Generates smart proxy classes. |
| `-loader` | Generates a single loader class for all the interfaces in an IDL. |
| `-server` | Generates a simple server mainline. |
| `-client` | Generates a simple client application. |
| `-incomplete` | Generates skeletal clients and servers. |
| `-makefile` | Generates a makefile that can build the server and client applications. |
| `-batch` | Generates a batch file for compiling the Java code under Windows. Use this option if `nmake` is unavailable. |
| `-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: 5.1:** *Java Genie Command Line Arguments*

These command line arguments are detailed in the following sections.

## -interface: Classes that Implement Interfaces

You can generate the classes that implement the interfaces in an IDL file by 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`.

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` operation (which is described in the next paragraph). The `_this` operation makes it possible for a TIE object to pass itself as a parameter to an IDL operation.

---

**Note:** The `-nothis` command-line option can be used to suppress the generation of the `_this` operation.

---

Another related matter is how the constructors of a class that implements an interface 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, then the algorithm used in the implementation of this operation 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` 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 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 package `finance`.

## -smart: Smart Proxies

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

```
idlgen java_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.

Again, consider the interface `account` 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()
      { ... };

public void makeLodgement(
      float f)
      throws org.omg.CORBA.SystemException;
      { ... };

public void makeWithdrawal(
      float f)
      throws org.omg.CORBA.SystemException;
      { ... };

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 SmartaccountFactory extends ProxyFactory
{
public SmartaccountFactory(
      org.omg.CORBA.Boolean  factoryDiagnostics,
      org.omg.CORBA.Boolean  proxyDiagnostics);
```

```
public void New(
      org.omg.CORBA.portable.Delegate d);
};
```

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 `Smartaccount.java` file.

```
SmartaccountFactory saf = new SmartaccountFactory(TRUE,TRUE);
```

## -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
class loader extends IE.Iona.OrbixWeb.CORBA.LoaderClass
{
public loader(org.omg.CORBA.Boolean printDiagnostics);

public org.omg.CORBA.Object load(
      String                interface,
      String                marker,
      boolean               isLocalBind );

public void save(
      org.omg.CORBA.Object      obj,
      org.omg.CORBA.saveReason  reason );

public void record(
      org.omg.CORBA.Object   obj,
      String                 marker );

   public org.omg.CORBA.Boolean rename(
      org.omg.CORBA.Object   obj,
      String                 marker );
};
```

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.java` file and uses a `TRUE` value when creating the loader, thereby enabling diagnostic messages. You can alter this if required.

The `load` operation 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 Mainline

Use the `-server` option to generate a simple server mainline:

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

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

```
// Java
int main(String args[])
{
   loader srvLoader = null;
   account obj1 = null;
   bank bj2 = null;

   orbRef = org.omg.CORBA.ORB.init(this, null);

   try {
      OrbRef.impl_is_ready("bankSrv", 0);
   } catch(org.omg.CORBA.SystemException ex) {
      System.err.println("impl_is_ready() failed");
      ex.printStackTrace(System.err);
      System.exit(1);
   }

   obj1 = accountImpl.create("account-1");
   obj2 = bankImpl.create("bank-1");

   try {
      OrbRef.impl_is_ready("bankSrv", 0);
   } catch(org.omg.CORBA.SystemException ex) {
```

```
      System.err.println("impl_is_ready() failed");
      ex.printStackTrace(System.err);
      System.exit(1);
   }

   return 0;
};
```

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

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

The server code would have included the following lines:

```
// Java
loader srvLoader = new loader(TRUE);
obj1 = accountImpl.create("account-1",srvLoader);
obj2 = bankImpl.create("bank-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 first binds to all of the objects in the server (one bind per interface that appears in the IDL file). It then calls every operation and attribute `get` and `set` method with random values for parameters.

The client source file is of the for

```
// Java
int main(String args[])
{
   account obj1 = null;
   bank obj2 = null;

   parse_cmd_line_args(args);

   OrbRef = org.omg.CORBA.ORB.init(this,null);

   try {
      obj1 = accountHelper._bind(
                  "marker-1:bankSrv", host);
      obj2 = bankHelper._bind("
```

```
                marker-2:bankSrv", host);
    } catch(org.omg.CORBA.SystemException ex) {
       System.err.println("_bind() failed");
       ex.printStackTrace(System.err);
       System.exit(1);;
    }

    accountCaller.get_balance(obj1);
    accountCaller.makeLodgement(obj1);
    accountCaller.makeWithdrawal(obj1);
    accountCaller.newAccount(obj2);
    accountCaller.deleteAccount(obj2);

    return 0;
}
```

## -incomplete: Skeletal Clients and Servers

If the `-client` option is specified then, by default, the Java genie generates a file called `call_funcs.java` which contains functions to invoke all the operations and attributes of objects in the server. These functions assign random values to the parameters of operations. They also print out the values of parameters that they send (and those that are received back as `out` parameters). Utility functions to assign random values to IDL type are generated in the file `RandomFuncs.java`, and utility functions to print the values of IDL type are generated in the file `PrintFuncs.java`.

Likewise, if the `-interface` option is specified then, by default, the Java genies generate bodies of operations and attributes which print the values of `in` and `out` parameters, and also assign random values for the `out` parameters.

These bodies of the generated server-side operations and the client-side calling functions mean that the Java genie can produce a complete application which can be compiled and run straight away. This is very useful for quickly producing a demo or proof-of-concept prototype. However, it also serves another useful purpose: the generated code provides a working example of how to initialize parameters (albeit with random values), invoke operations, `throw` and `catch` exceptions, and perform memory management.

If you do not want the Java genie to generate the bodies of operations, attributes or the client-side calling functions then you can use the `-incomplete` command-line option.

### -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.

### -batch: Batch File

This option generates a batch file for compiling the Java code under Windows. Use this option if `nmake` is unavailable.

## A Few Other Options

There are a number of other miscellaneous command line arguments that may come in useful. These are shown in Table 5.2:

| Command line argument | Purpose |
|---|---|
| `-(no)any` | By default, the Java genie does not generate code to support the use of `any` or `TypeCode` for user-defined types. This support can be turned on by using the `-any` command-line option. |
| `-jp` | This option allows you to specify the package name for your generated Java classes. The default name used is `noPackage`. |

**Table: 5.2:** *Miscellaneous Command Line Arguments*

**Note:** For a full list of the command line options for the Java genie please refer to the Appendix, under the section "User's Reference" on page 251.

# Creating Print Functions for IDL Types with java_print.tcl

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

```
idlgen java_print.tcl foo.idl -jp "myPackage"

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 `myPackage.Print<type Name>`, and the print method is simply called `<Type Name>`. 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 function for the `EmployeeDetailsSeq` type is placed in a class `myPackage.PrintEmployeeDetailsSeq`. The method itself has the following signature:

```
void EmployeeDetailsSeq(PrintStream &ut,
        EmployeeDetailsSeq seq,
        int indent);
```

This function takes three parameters. The first parameter is the `stream` 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, stringand 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 using the print functions is shown below:

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

# Creating Random Functions for IDL Types with java_random.tcl

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 -jp "myPackage"

idlgen: creating RandomFuncs.java
```

The names 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<type Name>`, and the print method is simply called `Random<Type Name>`. The functions generated for small IDL types (`long`, `short`, `enum`, and so on) return the random value. Thus, you can write code as follows:

```
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:

```
void set_seed( long new_seed);
long get_seed();
long get_rand(long range);
void reset_recursive_limits();
```

`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.

IDL allows the declaration of recursive types. For example:

```
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. `resetrecursive.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 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`
- `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 251.

# Part 2

# Developing Genies with the Orbix Code Generation Toolkit

# 6

# Writing a Genie

*Earlier chapters discussed how to run the bundled genies. However, You can do more with the Orbix Code Generation Toolkit than simply run the genies. You can modify the bundled genies or write your own to suit your own needs when developing CORBA systems.*

As described earlier in this guide, the IDLgen interpreter is extension of Tcl. Genies are Tcl scripts that use these extensions in parallel with the basic Tcl commands and features. These extensions allow programmers to easily parse IDL files and generate corresponding code of whatever specification they require.

This chapter uses several examples to illustrate how to write a genie. Some of the examples produce C++ code: however, the same principles apply for other output, such as Java.

## Prerequisites for Developing Genies

To develop your own genies you must have a good grasp of the following:

- The OMG IDL.
- Writing scripts in the Tcl language[1].

1. There are several good guides to the Tcl language available. The first part of *Tcl and the Tk Toolkit by John K. Ousterhout* provides an excellent introduction to the language.

If you wish to write your own genie which generates, for example, C++, then you should have a good knowledge of C++ and be familiar with the IDL to C++ mapping specification.

# Some Simple Examples

As IDLgen is a Tcl interpreter you can give it a Tcl script to interpret and it processes it in the same way as any other Tcl interpreter[2]. Tcl script files are fed into it and IDLgen outputs any results to the screen or to a file.

IDLgen can only interpret Tcl commands stored in a script file. IDLgen does not have an interactive mode where a user can interactively type commands in.

## Hello World

Consider this simple Tcl script:

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

Running this through IDLgen gives the following result:

```
idlgen hello.tcl

Hello, World
```

## Hello World with Command Line Arguments

IDLgen adheres to the Tcl conventions for command-line argument support. This is demonstrated in this script:

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

2. While IDLgen is a Tcl interpreter, it does not have any of the common Tcl extensions built in, such as Tk or Expect. You cannot use IDLgen to execute a Tk or Expect script.

Running this through IDLgen gives us the following results:

```
idlgen arguments.tcl Fred Joe Mary

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

# Some Extensions Provided by IDLgen

IDLgen adds some new commands to the Tcl language. These new commands support such tasks as:

- Parsing IDL files.
- Writing text into output files.
- Mapping IDL constructs onto programming language constructs.

This section will introduce you to some of the building blocks that make up these extensions, in preparation for writing an application that tackles the more complex areas of IDL parsing and code generation. Appendix B, "Command Library Reference" provides a reference to these Tcl commands.

## Using Commands in Other Libraries

Standard Tcl has a command called `source`. The `source` command is very much like the `#include` compiler directive used in C++ and allows a Tcl script to use commands 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 shown (the C++ equivalent is given for comparison):

```
# Tcl
source foobar.tcl

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

The source command has one limitation compared to its C++ equivalent. It has no search path for locating files. This has the obvious disadvantage of forcing the coder to specify full directory paths for other Tcl scripts.

IDLgen provides an enhanced version of the source command that allows a file to be sourced using a search path[3]. This command is called smart_source.

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

*smart_source* 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 discussed in the section "Embedding Text in Your Application" on page 74.
- It has a pragma once directive. This prevents repeated sourcing of library files and aids in overriding Tcl commands. This is covered later on in the guide, in the section "Re-implementing IDLgen Commands" on page 125.

## Writing to a File from Your Genie

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

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

Both of these defaults 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 in the following way:

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

3. The search path is given in the idlgen.genie_search_path item in the idlgen.cfg configuration file. For more details please refer to "General Configuration Options" on page 251.

However, this syntax is a little too verbose to be useful. As genies regularly need to create output in the form of a text file, IDLgen provides some utility functions for creating and writing files that provide a more concise syntax for writing text to a file.

These utility functions are located in the script `std/output.tcl`, so to use them you must `smart_source` them into your application. Here is a example, using these alternative output 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 "        ${class_name}() {\n"[4]
output "          cout << \"$class_name CTOR\";\n"
output "         }\n"
output "};\n"
close_output_file
```

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

```
idlgen codegen.tcl

idlgen: creating example.h
```

The contents of this file are:

```
class testClass : public baseClass
{
     public:
         testClass() {
            cout << "testClass CTOR";
         }
};
```

4. There are brackets placed around the `class_name` variable so that the Tcl interpreter does not assume `$class_name()` is an array.

The three commands used to create a file are listed in Table 6.1.

| Command | Result |
|---|---|
| `open_output_file` *`filename`* | Opens the specified file for writing. If the file does not exist then it is created. If the file exists it is over written. |
| `output` *`string`* | Appends the given string to the file currently open. |
| `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 on page 72 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 still allowing the flexibility of embedding Tcl commands and variables.

The Tcl language provides two ways that can be used to quote a large block of text, such as in our coding example.

The first approach is to quote the text inside braces ("`{ }`")which 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 gives us a new `example.h`:

```
class $class_name : public virtual $base_name
{
   public:
      ${class_name}() {
         cout << "$class_name CTOR";
      }
};
```

This example is easier to read but it does not allow you to substitute variables.

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

```
# 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 IDLgen results in this `example.h`:

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

This is much better than using braces as the variables are substituted correctly, but the use of quote marks is hampered because the Tcl programmer must remember that quote marks in the generated code must be prefixed with an escape character:

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

This can be difficult for the programmer.

The best solution is to have the C++ code in exactly the same form as you intend it to appear in the generated file and still have the ability to escape back for variables and nested commands. Luckily, in IDLgen you can do this, with *bilingual files*.

## What are Bilingual Files?

A *bilingual file* contains a mixture of two languages; one language is Tcl and the other is plain text. A preprocessor in IDLgen translates the plain text into `output` commands.

This is our example as a bilingual Tcl script.

```
# 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
```

As you can see from this example, plain text areas in bilingual scripts are marked by using *escape sequences*. These escape sequences are shown in Table 6.2:

| 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*

If you compare this to the same example without the use of a bilingual file, then it should be obvious that the bilingual version is easier to read.

It is much easier to write genies with bilingual files, particularly if you have a syntax-highlighting text editor which 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 the rest of this guide to help you distinguish text blocks.

**Note:** Bilingual files normally have the extension `.bi`. This is not required, but it is a convention used by all the genies bundled with the Orbix Code Generation Toolkit.

## Using Bilingual Files

Although bilingual files are a great benefit, there are a few things to watch out for. For instance, if you want to print the @ symbol inside a textual block use this technique:

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

```
...***]
```

Similarly, if you want to print `[***` or `***]` in a file then print it in two parts (to avoid it being treated as an escape sequence).

Comments can also be a problem as the bilingual file preprocessor does not understand them. You cannot do this:

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

So instead, use an `if` statement to disable the plain-text block:

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

A final point to note involves debugging. Debugging a bilingual file can sometimes be a little awkward. IDLgen 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.

This is where the `bi2tcl` utility can be useful. This utility takes a bilingual file and replaces the embedded text with `output` commands, generating a new, but semantically equivalent script. This can be useful for debugging purposes as it easier to understand the run-time interpreter error messages if line numbers tie together.

If you run the bilingual example used earlier 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 the equivalent but slightly lengthy:

```
# 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 sizes, so 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 with the original bilingual script.

# 7

# Processing an IDL File

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

This chapter describes how IDLgen 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, "IDL Parser Reference" provides a reference to the commands discussed in this chapter.

## IDL Files and IDLgen

The IDL parsing extension provided by IDLgen 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 parsed information is stored in an internal format called a *parse tree*. The contents of this parse tree can then be manipulated by an genie.

Consider this 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. Processing 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 first is the name of the IDL file and the second (which is optional) is a list of preprocessor directives that are passed to the IDL preprocessor.

Here is how you can use this command to process the IDL file `finance.idl`.

```
# 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:** During the parsing process, if any warning or error messages are generated they are printed to standard error. If the parsing fails the `idlgen_parse_idl_file` command returns false.

## Structure of the Parse Tree

Once an IDL file has been 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, with each node in the tree representing an IDL construct. For instance, parsing the finance IDL file forms a tree that looks like Figure 7.1.

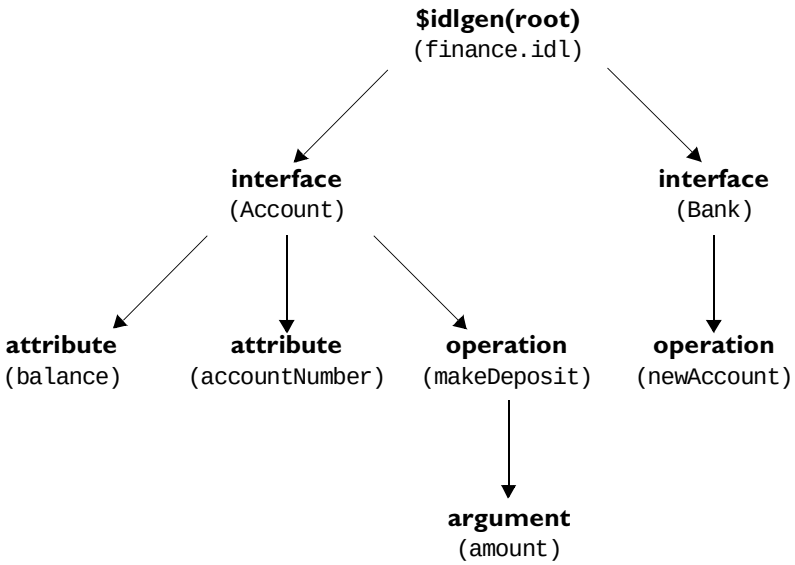


**Figure 7.1:** *The Finance IDL File's Parse Tree*

A genie can invoke operations on a node to obtain information about the corresponding IDL construct or to transverse to other parts of the tree that are related to the node the operation was performed on.

Assume that you have traversed the parse tree (how to do this is explained later in the chapter), and have located the node representing the `balance` attribute. You can find out the information associated with this node by invoking operations on that node:

```
# Tcl
```

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

> float
```

The operation used here is `type`, which returns a node that represents the type of the attribute. This `type` operation is specific to `attribute` nodes and `l_name` (which obtains the local name) is an operation that is common to all nodes.

**Note:** It is important to note that the parse tree also has all the contents of all the IDL files from `#include` statements as well as the ones from the parsed file.

You can use the node operation `is_in_main_file` to find out whether or not a construct came from the original file:

```
# 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"
}

> Account is in the main file
```

## Nodes of the Parse Tree

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 different type of node provides a number of operations. Some of these operation, like the local name of the node, are common across all the types of node:

```
# Tcl
puts [$operation_node l_name]

> newAccount
```

Some operations are specific to a particular type of node. For instance, 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]
```

```
> Account
```

All the different types of node are arranged into an inheritance hierarchy as shown in Figure 7.2:

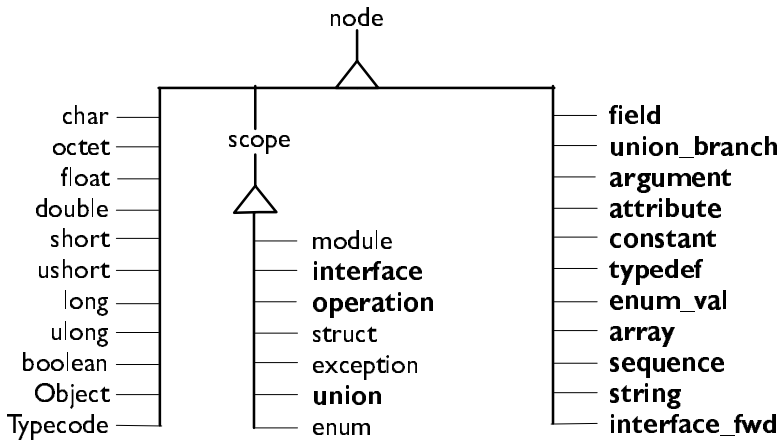


**Figure 7.2:** *Inheritance Hierarchy for Node Types*

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*.

## The Abstract Node *node*

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

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

Here is a pseudo-code definition of the abstract class node[1]:

```
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()
}
```

This abstract node supplies operations that allow you to find out such things as:

- What is the name of the node? `l_name()`.
- Which IDL file does this node appear in? `file()`.

All types of node inherit directly or indirectly from this abstract node. For instance, the node type that represents an argument of an operation inherits from node. It supplies some additional operations on top of the ones the abstract node supplies to allow the programmer to determine the type of the argument and what the direction modifier is (in, inout or out).

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

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

1. This is a partial definition of the abstract class node. Its complete definition can be found in Appendix C, "IDL Parser Reference".

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);
};
```

This handle to the amount argument has been placed in a variable called argument_node. To obtain information about the argument, the Tcl script could use any of the operations 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]'"

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 Abstract Node *scope*

The other abstract node is the scope node. The scope node represents constructs that have *scoping behavior* — constructs that can contain other constructs nested inside them. The operations provided by the scope node are the ones that aid in traversing the parse tree.

For instance, 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 abstract node node.

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

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

The pseudo-code definition of the interface node is:

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

In a number of the previous examples, an operation was performed on a node but no details were given about how that node was located. To locate this node, a search operation 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]
```

```
idlgen lookup.tcl

balance
Account::balance
```

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

## Locating Nodes with *contents* and *rcontents*

There are two more `scope` defined operations that can be used to locate nodes in the parse tree. These two operations 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` operation:

```
# 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]
}

idlgen contents.tcl

Account
Bank
```

This operation allows you to specify what type of constructs you wish 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).

There is a recursive version of this operation that allows a deeper search to be made. It does this by extending the search so that it recurses into other scoping constructs.

Here is an example of the `rcontents` operation:

```
# 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]"
}

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 Pseudo Node *all*

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

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

It is now very easy to write an genie that can visit (almost) every node in the parse tree[2]:

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

2. This example genie will visit *most* of the nodes in the parse tree. However, it will not visit any *hidden* nodes. The section "Visiting Hidden Nodes" on page 94 discusses how to access the hidden nodes in the parse tree.

```
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 wish to traverse.

## Nodes Representing Built-in IDL Types

Nodes that represent the built-in IDL types can be accessed with the `lookup` operation 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"
}

idlgen basic_types.tcl

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

For convenience, IDLgen 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 initialised
set param_type [$param_node type]
puts "Parameter type is [$param_type s_name]"

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];
```

The above segment of IDL apparently defines a sequence called longSeq and an array called longArray. However, a close reading of the CORBA specification reveals that all sequences and array types are anonymous. So the above segment of IDL actually defines a typedef (called longSeq) for an anonymous sequence, and another typedef (called longArray) for an anonymous array.

Here is a pseudo-code definition of the class typedef:

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

The base_type operation returns the node representing the typedef's underlying type. In the case of:

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

The base_type operation returns the node representing the anonymous sequence.

When writing IDLgen scripts, sometimes you may 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 tedious coding task, IDLgen defines an operation called `true_base_type` in the base class node. For most node types, this operation simply returns the node directly. However, for `typedef` nodes, this operation strips away all the layers of typedef, and returns the underlying type. Thus, the above example could be rewritten more concisely as:

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

## Visiting Hidden Nodes

As mentioned on page 90, using the `all` pseudo type 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 type does not really represent all types. However, it does represent all types that most scripts want to explicitly process. However, it is possible to visit these hidden nodes explicitly. For example, the following snippet of code processes all the nodes in the parse tree, including anonymous sequences or arrays and the built-in types.

```
# 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 abstract node `node` or from the abstract scoping node `scope`. The examples given have only covered a small number of the IDL constructs that are available. The different types of nodes are arranged into an inheritance hierarchy A full reference guide, which lists all of the node types and available operations, can be found in Appendix C, "IDL Parser Reference".

# 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 matched 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 IDLgrep 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
```

Theses results are a little more verbose than a normal `grep`.

So what does the search genie actually need to do? It must examine the whole parse tree and look for constructs that match the wild card criteria. It can examine all the possible constructs in the IDL file, but let us restrict this genie to search for the `interface`, `operation`, `exception`, and `attribute` constructs only.

This is a first attempt at writing `idlgrep`:

```
# 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]
}
```

```
idlgen idlgrep.tcl
```

```
Account
Bank
```

This is not exactly what the genie should do. Using the `contents` operation 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 not really what is required as the search goes no further than the root scope.

## Refining the Search

The Tcl code on page 95 can manually move through the parse tree by using further calls to `contents` but the `rcontents` operation is a more concise solution. The types of constructs the genie is looking for only appear in `module` and `interface` scopes, so the genie only needs to search those scopes.

This information is passed to the `rcontents` command in this 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]"
}

idlgen idlgrep.tcl

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

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

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

## Completing the Basic Search Tool

You can code this in a neater way by using a further feature of the `rcontents` operation (this feature is also provided by `contents`). By passing an additional parameter to `rcontents` command the resulting list of nodes can be filtered in-line. This parameter is the name of a function which 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`.

So to complete the basics of the `grep` style genie, this additional parameter is added to the `rcontents` command as well as providing the wild card and IDL file as 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 ""
   }
}
```

Running the finished genie on the `finance.idl` file gives the following results:

```
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
```

This example may seem a little contrived, but the principles and techniques it embodies are often used in other, more practical genies. What it shows is that a small genie can achieve a lot. The next few chapters extend the ideas shown here and allow better genies to be developed. For instance, `idlgrep.tcl` could be easily improved by allowing the user to specify more than one IDL file on the command line or allow further search options to be defined in a configuration file. The commands to allow the programmer to achieve such tasks are discussed in the next chapter.

# Recursive Descent Traversal

The main method of traversal over the 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 can be applicable to certain 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 as 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
}
```

Running this application through IDLgen, provides the following result:

```
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 an 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 kind of genie is perfect for the recursive descent mechanism. Look at the key 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` 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. In our genie this procedure does this:

```
# 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 procedure called process_interface for each interface (and so on):

```
# 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 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 the programmer to be in control when it comes to the traversal of the parse tree.

# Processing User-defined Types

The command `idlgen_list_builtin_types` returns a list of all the built-in IDL types. IDLgen 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 which 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` then user-defined exceptions are *not* included in the list of user-defined types that are returned. An example of the usage of this command is as follows:

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

Another utility command provided by IDLgen is:

`idlgen_list_all_types` *`exception`*

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

# Recursive Structs and Unions

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

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

Some genies may have to do special-case processing for recursive types. IDLgen provides the following utility functions to aid this task:

| Function | Operation |
|---|---|
| `idlgen_is_recursive_type` *`type`* | Returns:<br>1: if *`type`* is a recursive type.<br>0: if *`type`* is not recursive.<br>For example, this command returns 1 for both the `tree` and `widget` types. |
| `idlgen_is_recursive_member` *`member`* | Returns:<br>1: if *`member`* (a field of a `struct` or a branch of a `union`) has a recursive type.<br>0: if *`member`* does not have a recursive type.<br>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 which 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. |

**Table: 7.1:** *Utility Functions for Special-Case Processing*

# 8

# Configuring your Genies

*Genies are as a rule, at their best when they are made flexible through the use of preferences and user-defined options.*

There are two related mechanisms that allow a user of IDLgen and genies to specify their preferences and options. These two mechanisms are:

- Command-line arguments processing.
- Configuration file parsing.

This chapter discusses these two topics and describes how to make your genies flexible through configuration. Appendix B, "Command Library Reference" provides a reference to the commands discussed in this chapter.

## Command Line Arguments

Most useful command-line programs take command-line arguments. As IDLgen is predominately a command-line application, your genies will invariably use command-line arguments as well. IDLgen supplies functionality to parse command-line arguments easily.

### Enhancing IDLgrep

Although the `idlgrep.tcl` application (which was described in the section "Searching an IDL File with IDLgrep" on page 95) used command-line options it assumed that the IDL file was the first parameter and the wild card was the

second. Instead of hard coding these settings, it would have been better to use a more intelligent approach to command-line processing that did not make assumptions about argument ordering. 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 by using the built-in command `idlgen_getarg` 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.

Here is an example of what the short piece of code with do 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 user wishes to see all of the command-line options available 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 variable passed as the first parameter is a data structure which describes which command-line arguments are being searched for. The next three parameters are variable names that are assigned values by the idlgen_getarg command, as described in Table 8.1.

| idlgen_getarg 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*. |

**Table: 8.1:** *idlgen_getarg Arguments*

| idlgen_getarg Arguments | Purpose |
|---|---|
| `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 `smart_source` to access the `idlgen_getarg` command as it is a built-in command.

## Searching for Command Line Arguments

This first parameter to the `idlgen_getarg` command is a data structure which describes the syntax of the command-line arguments to search for. In the example for the IDLgrep application, this first parameter was 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 lists. Each sub-list is used to specify the search criteria for a type of command-line parameter.

The first element of this sub-list is a regular expression which specifies the format of the command-line arguments. In the example, 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 the 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 the 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 then the symbol is identical to the first element but if the regular expression does contain wild cards then 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

Here is an another example of `idlgen_getarg` looping through some command-line arguments:

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

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

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

```
}
puts "Extension is $extension"
```

Running this application with appropriate command-line arguments gives:

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

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

This 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
```

## IDLgrep with Command Line Arguments

To finish the IDL grep 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
}
```

Here is the full listing for the `grep_file` procedure:

```
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 procedure, parse_cmd_line_args, which processes the command-line arguments which are common to most genies. In particular, it processes the following command-line arguments: -I, -D, -v, -s, -dir, -h and IDL files. 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 procedure returns the name of the specified IDL file through the idl_file parameter, and preprocessor options through the options parameter. However, if parse_cmd_line_args encounters the -h option or any unrecognized option, or if there is no IDL file specified on the command-line then 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 then 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 then you can use std/args.tcl "as is" in your genie. If, however, your genie requires some additional command-line arguments then you can copy std/args.tcl and modify the copy so that it can process additional command-line arguments. In this way, std/args.tcl provides a useful starting point for command-line processing in your genies.

# Using Configuration Files

IDLgen and the bundled genies use information in a configuration to enhance the range of options and preferences offered to a user. Some such configurable options are:

- The search path for the `smart_source` command.
- Whether a user prefers the TIE or BOA approach when implementing an interface.
- Which file extensions to use when generating C++ or Java files.

IDLgen's core settings and preferences are stored in a standard configuration file which, 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, "Configuration File Grammar".

A comment can appear anywhere and lasts to the end of the line:

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

Use the = symbol to assign a string value to a name. Use a semi-colon to terminate the assignment:

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

Use the + symbol to concatenate strings together. In this example the `host` configuration item would have the value `amachine.iona.com`:

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

Use the = symbol to assign a list to a name and put the items of the item inside matching `[` and `]` symbols:

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

Use the + symbol to concatenate lists together. 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 instance, allow configuration items of the same name to be stored in different scopes. In this 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 command `idlgen_parse_config_file` 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.

Here 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`).

This example 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 once 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";
};
```

Running this application through IDLgen gives the following results:

```
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 please refer to the Appendix section "Configuration File API" on page 262.

## The Standard Configuration File

When IDLgen starts, it reads a configuration file specified by the `IDLGEN_CONFIG_FILE` environment variable. 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`.

## IDLgrep with Configuration Files

Consider a new requirement to enhance IDLgrep once more to allow the user to specify which IDL constructs they wish the search to include. The user may also wish to specify which constructs to search into recursively. It would be time consuming for the user to specify these details on the command-line, so it is better to have these settings stored in the standard configuration file.

Assume that the standard configuration file has the following scoped entries in it:

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

The following code from the `grep_file`[1] procedure must be replaced:

```
# Tcl
```

1. The full listing of this procedure can be found on page 111.

```
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 IDLgen with the new variation of IDLgrep gives us this 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 user a much more flexible application that can be tailored to meet their further needs. A small shortcoming of this application is that it assumes there is an entry in the in the configuration file. This is a bad assumption and so the code needs to be improved. This will ensure a more robust solution.

Here is the improved version that employs some more of the configuration file operations:

```
# 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 operation `type` allows a programmer to determine whether the configuration item exists or not and if it does exist, whether it is a list entry or just a string entry. The improved code checks which is the case and 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

# Further Development Issues

*This chapter details further development facets of IDLgen that will help you write genies with more speed and efficiency.*

This chapter described the following topics in detail:

- Global variable arrays used in IDLgen.
- Re-implementing IDLgen commands.
- A recommended programming style for genies.

## Global Arrays

Commonly accessed information must be readily available and quick to access, or else even coding simple things can become difficult. IDLgen employs a number of global array variables to store such common information with the added benefit that this approach aids in the reduction of name space pollution.

Some of these global variables have already been touched upon in previous chapters. For example,`$idlgen(root)` was mentioned in "Structure of the Parse Tree" on page 83 and is used to hold the results of parsing an IDL file.

---

**Note:** When using array variables make sure you do not place spaces inside the parentheses, otherwise Tcl will treat it as a different array index than 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 refer to the Chapter "Processing an IDL File" on page 81.

### $idlgen(cfg)

This variable represents all the configuration settings from IDLgen's standard configuration file `idlgen.cfg`:

```
# Tcl
set version [$idlgen(cfg) get_string orbix.version_number]
```

For more information please refer to the section "Using Configuration Files" on page 113.

### $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>"

idlgen globalvars.tcl

Usage: idlgen globalvars.tcl -f <file>
```

## The $pref Array

It is best to avoid embedding coding preferences in a script which will be re-used many times by different users. Genies usually consist of numerous procedures, so you should keep individual procedures flexible. Passing numerous parameters to each procedure is impractical so it is better to have a global repository of coding preferences which can be examined by procedures.

IDLgen provides a number of mechanisms to support genie preference:

- Command line arguments.
- Configuration files.

Configuration files can be, in coding terms, 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 IDLgen. 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 these 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(trousers,colour)
```

IDLgen automatically creates preference array values for all the `default` scoped entries in the standard configuration file using this command:

```
# Tcl
idlgen_set_preferences $idlgen(cfg)
```

**Note:** This command assumes that all names in 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, then 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 have been processed explicitly by a programmer:

```
# 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(trousers,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 procedures in the `std/output.tcl` library examine the entries described in Table 9.1:

| $pref(...) Array Entry | Purpose |
|---|---|
| `$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) then 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` then diagnostic messages such as `idlgen: creating foo_i.h` are written to standard output whenever a genie generates an output file.<br>If this entry has the value `0` then no such 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`.<br>The default value of this entry is `1`. |

**Table: 9.1: $pref(...) Array Entries**

## The $cache Array

If a procedure is called frequently, caching its result can speed up a genie. Caching the results of frequently called procedures can speed up genies by up to twenty per cent. Many of the commands supplied with IDLgen perform caching. This mechanism is useful for speeding up your own genies.

Consider this simple 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 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 IDLgen Commands

Consider a genie which uses a particular Tcl procedure extensively, but you must now alter the its behavior. The genie uses this 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 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 procedure while still allowing the process to be reversed if required. The new implementation of the 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` procedure:

```
# Tcl
proc say_hello {message} {
   puts "Hello '$message'"
}
```

If an genie used `say_hello` from the original script it can use the original procedure's functionality:

```
# Tcl
smart_source "original.tcl"
say_hello Tony

idlgen application.tcl

Tony
```

However, to override the procedure the programmer only needs to smart_source the new procedure instead:

```
# Tcl
smart_source override.tcl
say_hello Tony

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 command to be re-instated.

```
# Tcl
smart_source "override.tcl"
smart_source "original.tcl" ;# Oops
say_hello Tony

idlgen application.tcl

Tony
```

Smart source provides a mechanism to prevent this. This mechanism is accomplished is 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 procedure. Here is the original implementation with the new pragma directive added:

```
# Tcl
smart_source pragma once
proc say_hello {message} {
   puts $message
}
```

Here 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 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

idlgen application.tcl

Hello 'Tony'
```

## More Output

IDLgen provides an alterative set of commands for the ones found in the script `std/output.tcl`. This alternative set of 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 command are slower because they write a temporary file and run a `diff` with the target file. This is typically10% slower than the equivalent commands in `std/output.tcl`.

# Miscellaneous Utility Commands

The following sections discuss miscellaneous utility commands provided by IDLgen.

## 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 which 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 are adding a CORBA wrapper to an existing Widget-based application. Such a script usually:

- Contains procedures that generate data-type conversion functions for user-defined type such as `structs`, `unions`, `sequences`, and so on.
- Copies (to the output files) pre-written functions that perform data-type conversion for built-in IDL types such as `short`, `long`, `string`, and so on.

You can ensure that pre-written code is copied to an output file by taking advantage of IDLgen'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 then this approach becomes unwieldy because the script can contain more lines of embedded text than lines of Tcl code, which leads to an excessive amount of scrolling in program editors when developing genies.

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 then 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:

- An 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 an 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 an genie that enables you to put a CORBA wrapper around an arbitrary Widget-based genie then 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 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. If your organization has such a policy then every genie you write must contain code to copy this copyright notice into every generated file. You could use IDLgen's bilingual capability to embed the copyright notice inside a text block in all your genies. However, this has several drawbacks. Firstly, copying this copyright notice into all your genies is a boring, repetitive task. Secondly, if your organizations legal department requests that the copyright notice be updated then you will have to manually edit all your genies in order to update the text.

A better approach is to store the copyright notice in a well-known place, such as a configuration file, and have your genies invoke a utility function which formats the text as a comment and then writes it to the generated file. The `idlgen_gen_comment_block` command is provided for this purpose. Let us 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 IDLgen 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 "//" "-"
```

then 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 to 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 then 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 the requirement to generate a separator (for example, a comma) between each item of a list requires substantially more code. Furthermore, if empty lists require special-case logic then additional code is required to handle them.

IDLgen 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 then *empty_str* is returned. Otherwise:

1. idlgen_process_list 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, along with *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 onto the result, which is then returned.

The *start_str, sep_str, end_str* and *empty_str* parameters have a default value of "". Thus, you need 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 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 then 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) so acc is not properly aligned. Using a relatively large value for *pad_len*, such as 32, minimizes the likelihood of misalignment occurring. However, IDL 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, say, 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 parameter and variable declarations.

# Recommended Programming Style

The bundled genies share a common programming style. In the following sections, we highlight some aspects of this programming style and explain how adopting the same style will 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 procedures in other files. This helps to keep the genie code modular.
- If the mainline script of your genie is called `foo.tcl` then any associated files should be in a sub-directory called `foo`. This helps to prevent (file)name-space pollution. 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`.
- 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 `$pref`. For example, the `args.tcl` procedures in the `cpp_genie.tcl` genie uses the entry `$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 then place the 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 Procedures

The following code illustrates several recommendations for organizing the 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 (if it helps, think of ACME as being DCE, DCOM, OSP, RogueWave, Oracle, ObjectStore or some other product with which you are familiar). 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` then `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 may 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 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:

- 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, say 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 an genie can access the API of a library with just one smart_source command, regardless of how feature rich that library is.

- Lack of encapsulation: any genie that wants to access the API of idl2acme.tcl must be aware of the names of files in the idl2acme directory. If the names of these files ever change it will break other genies that make use of them.

Both of these problems can be solved with the following convention. 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 an 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 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 `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 just the latter three 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 the full and minimal libraries is that the minimal library is likely to contain only a small amount of code (say, fifty lines 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 IDLgen 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 3

# Orbix C++ Genies: Library Reference

# 10

# The C++ Development Library

*The Orbix Code Generation Toolkit comes with a rich C++ development library that makes it easy to create code generation applications that map IDL onto C++ code.*

The file `std/cpp_boa_lib.tcl` is a library of Tcl procedures that map IDL constructs into their C++ counterparts. The term `boa` in the name of the library indicates that this library is for the IDL-to-C++ mapping defined by the CORBA specification of the BOA (Basic Object Adaptor).

## Naming Conventions in API procedures

Abbreviations are commonly used in the names of procedures defined in the `std/cpp_boa_lib.tcl` library. The following table lists these abbreviations and their meanings:

| Abbreviation | Meaning |
|---|---|
| `clt` | Client. |
| `srv` | Server. |
| `var` | Variable. |
| `var_decl` | Variable declaration. |

**Table: 10.1:** *Abbreviations Used in Procedure Names*

| Abbreviation | Meaning |
|---|---|
| is_var | Discussed below. |
| gen_ | Discussed below. |
| par (or param) | Parameter. |
| ref | Reference. |
| stmt | Statement. |
| mem | Memory. |
| op | Operation. |
| attr_acc | An attribute's accessor. |
| attr_mod | An attribute's modifier. |
| sig | Signature. |
| _cc | A .cc or .cpp file. |
| _h | A .h file. |

**Table: 10.1:** *Abbreviations Used in Procedure Names*

The names of all the procedures in `std/cpp_boa_lib.tcl` start with `cpp_`, which implies "C++".

As an example, the following statement assigns the C++ signature of an operation (for use in a `.h` file) to variable `foo`:

```
set foo [cpp_op_sig_h $op]
```

## Naming Conventions for "is_var"

The 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 are using these `_var` types. For this reason, some of the procedures 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_"

Some procedures contain the term `gen_` in their names. Such procedures *generate* output. For example, `cpp_gen_op_sig_h` outputs the C++ signature of an operation for use in a header file. Procedures whose names do not contain `gen_` *return* a value (which you can use as a parameter to the `output` command if you wish).

Some procedures whose names do not contain `gen_` also have `gen_` counterparts. The reason for providing both forms of a procedure is to offer flexibility in how you can write scripts. In particular, the procedures 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 of textual blocks. Some examples can help to illustrate this.

The following segment of code prints the C++ signatures of all the operations of an interface for use in a `.h` file:

```
foreach op [$inter contents {operation}] {
   output "\t[cpp_op_sig_h $op];\n"
}
```

Note that the `output` statement uses a TAB character (`\t`) to indent the signature of the operation, and also follows the signature with a semicolon and newline character. The printing of all this white space and syntactic baggage is automated by the `gen_` counterpart of this procedure, so the above code snippet could be rewritten in the following, slightly more concise format:

```
foreach op [$inter contents {operation}] {
   cpp_gen_op_sig_h $op
}
```

The cpp_gen_ procedures 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 procedure and then enter another textual block to print more text. For example:

```
[***
//--------
// Function: ...
//--------
***]
cpp_gen_op_sig_cc $op
[***
{
   ... // body of the operation
}
***]
```

The use of procedures 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:

```
[***
//--------
// Function: ...
//--------
@[cpp_op_sig_cc $op]@
{
   ... // body of the operation
}
***]
```

## Indentation

Consistent indentation is important for code clarity. However, there are no universally accepted rules for indentation: some programmers use two spaces for each level of indentation, while other programmers use four or eight spaces, or a TAB character.

If the procedures in `std/cpp_boa_lib.tcl` obeyed a particular indentation policy, say, four spaces for each level of indentation, then this would suit C++ programmers who use the same indentation policy in their applications. It would, however, be frustrating for people who prefer a different indentation policy. To avoid this problem, the white space to be used for one level of indentation is held in the variable `$pref(cpp,indent)`. Any procedure in `std/cpp_boa_lib.tcl` that needs to print some indentation uses the string specified in `$pref(cpp,indent)`.

The default value for `$pref(cpp,indent)` is `\t`, but you can change it to a different value such as four or eight spaces.

Some procedures take a parameter called `ind_lev`. This parameter is an integer that specifies the indentation level at which output should be generated. To illustrates this, consider the following code:

```
# Tcl
set name "foo"
set type [$idlgen(root) lookup "string"]
set is_var 1
for {set ind_lev 0) ($ind_lev < 3) (incr ind_lev} {
   cpp_gen_var_decl $name $type $is_var $ind_lev
}
```

The `cpp_gen_var_decl` procedure declares a variable of the specified `name` and `type`, at the specified level of indentation. The output from the above code would look something like:

```
CORBA::String_var      foo;
   CORBA::String_var      foo;
      CORBA::String_var      foo;
```

## $pref(cpp,...) Entries

Some entries in the `$pref(...)` array are used to specify various user preferences for the generation of C++ code. All of these entries are given default values by `std/cpp_boa_lib.tcl`, but the default values can be over-ridden by corresponding entries in the `idlgen.cfg` file or by explicit assignment in a Tcl script.

| $pref(...) Array Entry | Purpose |
|---|---|
| `$pref(cpp,h_file_ext)` | Specifies the filename extension to be on header files. Its default value is `.h`. |
| `$pref(cpp,cc_file_ext)` | Specifies the filename extension to be on code files. Its default value is `.cc`. |
| `$pref(cpp,indent)` | Specifies the amount of white space to be used for one level of indentation. Its default value is `\t`. |
| `$pref(cpp,impl_class_suffix)` | Specifies the suffix that is used to obtain the name of a class that implements an IDL interface. Its default value is `_i`. |
| `$pref(cpp,smart_proxy_prefix)` | Specifies the prefix that is used to obtain the name of a smart proxy class for an IDL interface. Its default value is `smart_`. |
| `$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 `1`. It should be set to `0` only if generating C++ code for an old C++ compiler that does not support exceptions. |

**Table: 10.2:** *$pref(...) Array Entries*

| $pref(...) Array Entry | Purpose |
|---|---|
| `$pref(cpp,want_named_env)` | A boolean value that specifies whether the `CORBA::Environment` parameter at the end of the C++ signature of an operation or attribute should have a name or should be an anonymous parameter. If you are generating C++ bodies of operations/attributes that do *not* access the `CORBA::Environment` parameter then setting `$pref(cpp,want_named_env)` to `0` prevents C++ compilers from warning that this parameter is not used. Its default value is `0`. |
| `$pref(cpp,env_param_name)` | Specifies the name of the `CORBA::Environment` parameter in the C++ signatures of operations and attributes. This name is used only if `$pref(cpp,want_named_env)` is `0`. Its default value is `_env`. |
| `$pref(cpp,attr_mod_param_name)` | Specifies the name of the parameter in the C++ signature of an attribute's modifier operation. Its default value is `_new_value`. |
| `$pref(cpp,ret_param_name)` | Specifies the name of the variable that is to be used to hold the return value from a non-`void` operation call. Its default value is `_result`. |
| `$pref(cpp,max_padding_for_types)` | This is used to pad out C++ type names when declaring variables or parameters. This padding helps to ensure that the names of variables/parameters are vertically aligned, which makes code easier to read. Its default value is 32. |

**Table: 10.2:** *$pref(...) Array Entries*

As the rules for mapping IDL to C++ code are clearly defined it is easy to encapsulate these rules into procedures in the Tcl language. These procedures and those specific to Orbix code generation comprise the C++ development library for IDLgen and make genies easy to develop.

# Identifiers and Keywords

It is illegal to use the underscore character as the first character when choosing operation names in IDL because the generated code for the interface can contain class functions that conflict with the operations defined at the `CORBA::Object` level. When writing or generating code from the IDL interface make sure that the IDL identifiers do not conflict with the built in language keywords.

Consider this unusual, but valid, interface:

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

The interface maps to a C++ class with a class method defined as:

```
// C++
char* _for( const CORBA::Long _while );
```

There are a number of procedures that help map the IDL data types to their language equivalents.

## cpp_l_name

This command is used as follows:

`cpp_l_name` *node*

This procedure returns the C++ mapping of a node's local name. This is usually the node's local name itself, but is prefixed with an underscore if the local name conflicts with a C++ keyword. Also, if the node happens to be a built-in type then the result is the C++ mapping of the type.

For example, consider this IDL:

```
// IDL
```

```
interface for {
   exception new {};
   void while( in octet goto );
};
```

If each construct (node) of this IDL is run through the `cpp_l_name` procedure the returned value is as follows:

```
for         _for
new         _new
while       _while
```

The `cpp_l_name` procedure also maps the built-in IDL data types to the corresponding C++ `typedefs`. For example the basic data types map as follows:

```
short               CORBA::Short
long                CORBA::Long
unsigned short      CORBA::UShort
unsigned long       CORBA::ULong
float               CORBA::Float
double              CORBA::Double
char                CORBA::Char
boolean             CORBA::Boolean
octet               CORBA::Octet
Object              CORBA::Object
```

## cpp_s_name

This command is used as follows:

```
cpp_s_name node
```

This procedure is similar to the cpp_l_name procedure. The difference is that it returns the-fully scoped name rather than the local name. If the IDL on page 150 in run through cpp_s_name the result is as follows:

```
for         _for
new         _for::_new
while       _for::_while
```

Built in IDL types are mapped as they are in the cpp_l_name procedure.

## cpp_typecode_s_name

This command is used as follows:

```
cpp_typecode_s_name type
```

This procedure returns the fully-scoped C++ name of the typecode for the specified type. Typecodes are usually formed by prefixing the name of the type with _tc_, but there are some exceptions. In particular, the typecodes for the built-in types (long, short and so on) are defined inside the CORBA module.

Examples of the fully-scoped names of C++ typecodes for IDL types:

```
cow             _tc_cow
farm::cow       farm::_tc_cow
long            CORBA::_tc_long
```

## cpp_typecode_l_name

This command is used as follows:

`cpp_typecode_l_name` *type*

This procedure returns the local C++ name of the `typecode` for the specified `type`. `Typecodes` are usually formed by prefixing the name of the type with `_tc_`, but there are some exceptions. In particular, the typecodes for the built-in types (`long, short` and so on) are defined inside the CORBA module.

Examples of the local names of C++ typecodes for IDL types:

| | |
|---|---|
| `cow` | `tc_cow` |
| `farm::cow` | `_tc_cow` |
| `long` | `CORBA::_tc_long` |

# General Purpose Procedures

There are also a number of general purpose procedures that can be used to help write code generation applications.

## cpp_is_fixed_size

This command is used as follows:

`cpp_is_fixed_size` *type*

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 or not the specified `type` is fixed size.

This command is called internally from other commands in the `std/cpp_boa_lib.tcl` library. However, it is unlikely that you will need to make use of it directly in your own applications.

## cpp_is_var_size

This command is used as follows:

`cpp_is_var_size` *type*

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 or not the specified `type` is variable size.

This command is called internally from other commands in the std/`cpp_boa_lib.tcl` library. However, it is unlikely that you will need to make use of it directly in your own applications.

## cpp_is_keyword

This command is used as follows:

cpp_is_keyword *name*

This command returns a boolean value that indicates whether or not the specified `name` is a C++ keyword. For example:

```
# Tcl
cpp_is_keyword "new"; # returns 1
cpp_is_keyword "cow"; # returns 0
```

This command is called internally from other commands in the `std/cpp_boa_lib.tcl` library. However, it is unlikely that you will need to make use of it directly in your own applications.

## cpp_assign_stmt

This command is used as follow:

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

This command returns a C++ statement that assigns the specified `value` to the variable of the specified `name` and `type`. 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 reason why the

ind_lev (indentation level) parameter is required is that 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 1) that specifies whether or not an extra scope (that is, a pair of braces ("{}") should surround the for loop. This extra level of scoping makes the generated code look ugly, but it works around a scoping-related bug in some C++ compilers.

There is a gen_ counterpart to the cpp_assign_stmt command:

cpp_gen_assign_stmt *type name value ind_lev ?scope?*

The following example illustrates the use of this gen_ 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_gen_assign_stmt $type $name $value $ind_lev 0
}
 [***
} // some_func()
***]
```

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

```
// 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_gen_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.

The for loop used in the assignment to my_long_array declares the index variable i1 inside the body of the for loop. The C++ specification states that variable i1 is visible only inside the body of the for loop. However, some C++ compilers do not correctly scope the visibility of such index variables which can result in i1 being visible outside body of the for loop. If the Tcl script generated assignment statements for several array variables then the (buggy) C++ compiler would complain of the variable i1 being declared multiple times. To work around this problem in some compilers, the cpp_assign_stmt command (and its gen_ counterpart) takes a boolean parameter called scope. If this is set to 1 (which is its default value) then an extra pair of braces ("{}") are placed around the for-loop. This extra level of scoping limits the visibility of index variables used in the for-loops and thus prevents redeclaration of variable i1 error messages from buggy compilers.

## cpp_indent

This command is used as follows:

cpp_indent *number*

This command returns a string which is the string $pref(cpp,indent) concatenated with itself the specified number of times. For example:

```
#Tcl
puts "[cpp_indent 1]One"
puts "[cpp_indent 2]Two"
puts "[cpp_indent 3]Three"
```

This produces output in the following form:

```
One
    Two
        Three
```

## cpp_nil_pointer

This command is used as follows:

`cpp_nil_pointer` *`type`*

This command returns a C++ expression that is a nil pointer (or a nil object reference) for the specified `type`. It should be used *only* for types that might be heap-allocated, that is, `struct`, `exception`, `union`, `sequence`, `array`, `string`, `Object`, `interface` or `TypeCode`. If used for any other types, for example, a `long`, then this command throws an exception.

This procedure can be used to initialise pointer variables. Note that there will rarely be a need to use this command if you make use of `_var` types in your applications.

## cpp_sanity_check_idl

Unfortunately, the mapping of IDL-to-C++ has some loopholes. For example, consider the following type:

```
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 (emphasised in the above example) is mapped to C++. This means that each CORBA vendor can map this anonymous type in a proprietary manner. The net effect of loopholes like these in the mapping from IDL to C++ is that use of these anonymous types can hinder portability of C++ code. This portability problem can be overcome by using extra typedef declarations in IDL files. For example, the above snippet of IDL could be rewritten as shown below:

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

The `cpp_sanity_check_idl` command traverses the parse tree looking for unnecessary anonymous types which cause portability problems in C++. If it finds any then it prints out a message which warns the user of the portability problems in the IDL file. An example of using this command is shown below:

```
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
```

# Interfaces

One of the major encapsulating constructs in IDL is the *interface.* This maps to an appropriately named class in C++. There are a number of procedures that aid in generating code for interfaces.

## cpp_impl_class

This command is used as follows:

`cpp_impl_class` *`interface_node`*

This procedure returns the name of a C++ class that can be used to implement a given IDL interface. The class name is constructed by getting the fully scoped name of the IDL interface and replacing all occurrences of :: with _ (that is, flattening the namespace). It also appends `$pref(cpp,impl_class_suffix)` on to the end. The default value for this is `_i`.

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

For example, the following interface definitions result in the generation of the corresponding C++ code.

```
// IDL                        // C++
interface cow {               class cow_i {
   ...                           public:
};                                  cow_i();
                              };
```

```
// IDL
module farm {
   interface cow {
      ...
   };
};
```

```
// C++
class farm_cow_i {
   public:
      farm_cow_i();
};
```

## cpp_tie_class

This command is used as follows:

`cpp_tie_class` *`interface_node`*

This procedure returns the name of the TIE macro corresponding to the given IDL interface.

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

DEF_@[cpp_tie_class $inter]@(@$class@)
***]
```

For example, the following interface definitions result in the generation of the corresponding C++ code.

```
// IDL
interface cow {
   ...
};
```

```
// C++
class cow_i {
   public:
      cow_i();
};
DEF_TIE_cow(cow_i)
```

```
// IDL
module farm{
   interface cow {
      ...
   };
};
```

```
// C++
class farm_cow_i {
   public:
      farm_cow_i();
};
DEF_TIE_farm_cow(farm_cow_i)
```

## cpp_boa_class_s_name

This command is used as follows:

`cpp_boa_class_s_name` *`interface_node`*

This command returns the fully scoped name of the BOA class that can be used to implement an IDL interface.

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

***]
```

For example, 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:
      cow_i();
};
```

## cpp_boa_class_l_name

This command is used as follows:

`cpp_boa_class_l_class` *`interface_node`*

This command returns the local name of the BOA class that can be used to implement an IDL interface.

Note that this command is rarely used; the `cpp_boa_class_s_name` is normally used instead.

## cpp_smart_proxy_class

This command is used as follows:

`cpp_smart_proxy_class` *`interface_node`*

This procedure returns the name of a C++ class that can be used when constructing a smart proxy class for a given IDL interface. The class name is constructed by obtaining the fully-scoped name of the IDL interface and replacing all occurrences of `::` with `_` (that is, flattening the namespace). It also prefixes the interface name with `$pref(cpp,impl_class_suffix)`. The default value for this is `smart_` .

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

For example, the following interface definitions result in the generation of the corresponding C++ code.

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

```
// IDL                  // C++
module farm {           class smart_farm_cow : virtual public
   interface cow {      farm::cow {
      ...                  public:
   };                         smart_farm_cow();
};                      };
```

# Signatures of Operations

The following set of commands is used to obtain the C++ signature of an operation:

```
cpp_op_sig_h      operation_node
cpp_gen_op_sig_h  operation_node
cpp_op_sig_cc     operation_node ?class_name?
cpp_gen_op_sig_cc operation_node ?class_name?
```

The gen_ variants of these commands generate output, while the *non-* gen_ variants return the C++ signature as a string (which you can then `output` if you want).

The _h variants of these commands return/generate C++ signatures suitable for use in a header (`.h`) file, while the _cc variants return/generate C++ signatures suitable for use in an implementation (`.cc`) file. The _cc variants take an optional `class_name` parameter. If this parameter is not specified it is calculated as:

```
set class_name [cpp_impl_class [$op defined_in]]
```

This is usually the desired class name. However, the ability to use an alternative class name is provided in case you want to generate signatures of operations for, as an example, smart proxies or some other support class.

# Signatures of Attributes

The following set of commands are used to obtain the C++ signatures of accessor and modifier functions for attributes:

```
cpp_attr_acc_sig_h      attribute_node
cpp_gen_attr_acc_sig_h  attribute_node
cpp_attr_mod_sig_h      attribute_node
cpp_gen_attr_mod_sig_h  attribute_node
cpp_attr_acc_sig_cc     attribute_node ?class_name?
cpp_gen_attr_acc_sig_cc attribute_node ?class_name?
cpp_attr_mod_sig_cc     attribute_node ?class_name?
cpp_gen_attr_mod_sig_cc attribute_node ?class_name?
```

You can determine which command to use by the different elements in the command name. Table 10.3 describes the different name elements:

| Element in command name | Command use |
|---|---|
| acc | Variants are for attribute *accessor* functions. |
| mod | Variants are for attribute *modifier* functions. |
| gen_ | Variants of these commands *generate* output. |
| *non-* gen_ | Variants *return* the C++ signature as a string (which you can then output if you want). |
| _h | Variants of these commands return/ generate C++ signatures suitable for use in a header (.h) file. |

**Table: 10.3:** *Attribute Signature Commands*

| Element in command name | Command use |
|---|---|
| _cc | Variants return/generate C++ signatures suitable for use in an implementation (.cc) file. The _cc variants take an optional class_name parameter. If this parameter is not specified then it is calculated as:<br>`set class_name [cpp_impl_class [$op defined_in]]`<br>This is usually the desired class name. However, the ability to use an alternative class name is provided in case you want to generate attribute signatures for, as an example, smart proxies or some other support class. |

**Table: 10.3:** *Attribute Signature Commands*

# Types and Signatures of Parameters

Previous sections have discussed commands that can be used to generate C++ signatures of IDL operations and attribute accessor/modifier functions. However, sometimes you may want more control over the construction of an operation's signature. In order to do this, you need to be able to determine the *type* or *signature* of individual parameters. The following commands are provided for this purpose:

```
cpp_param_type op_or_arg
cpp_param_type type dir
cpp_param_sig  op_or_arg
cpp_param_sig  name type dir
```

The cpp_param_type command returns the C++ type of a parameter of the specified type and direction (dir). For example, the following snippet of Tcl prints out const char *:

```
# Tcl
set type [$idlgen(root) lookup "string"]
set dir "in"
```

```
puts "[cpp_param_type $type $dir]"
```

The `cpp_param_sig` command returns the C++ signature of a parameter of the specified `name`, `type` and direction (`dir`). The signature is composed of the C++ type and the parameter's name. For example, consider the following snippet of Tcl code:

```
# Tcl
set type [$idlgen(root) lookup "string"]
set dir "in"
puts "[cpp_param_sig "foo" $type $dir]"
```

The output generated from the above code is:

```
const char * foo;
```

There is some whitespace padding between the parameter's type and name. The amount of padding is determined by `$pref(cpp,max_padding_for_types)`. This padding is used to ensure vertical alignment of parameter names. You can use an `argument` node or an `operation` node (the latter indicating the operation's return type) instead of specifying `type` and `dir` separately.

## Client-Side Processing of Parameters

The `std/cpp_boa_lib.tcl` library provides commands to manipulate client-side variables that are used as parameters to, or the return value of, an operation call. The commands provided are as follows:

```
cpp_clt_par_decl         arg_or_op is_var
cpp_clt_par_ref          arg_or_op is_var
cpp_clt_free_mem_stmt    arg_or_op is_var
cpp_clt_need_to_free_mem arg_or_op is_var
```

Some of the above commands have `gen_` counterparts:

```
cpp_gen_clt_par_decl       arg_or_op is_var ind_lev
cpp_gen_clt_free_mem_stmt  arg_or_op is_var ind_lev
```

In all of the above commands, the `arg_or_op` parameter can be either an `argument` node or an `operation` node in the parse tree. If `arg_or_op` is an `argument` node then the above commands apply to a parameter of an operation call. Conversely, if `arg_or_op` is an `operation` node then the above commands apply to the return value of an operation call.

The cpp_clt_par_decl command returns a C++ declaration of a variable that can be used as a parameter to (or return value of) an operation. 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 then it must be heap allocated, and the is_var parameter determines whether the parameter will be declared as a _var (smart pointer) type or as a raw pointer.

The cpp_clt_par_ref command returns a reference to the value of the specified parameter (or return value) of an operation. The returned reference is either of the form foo or *foo, depending on how the parameter was declared by the cpp_clt_par_decl command.

The cpp_clt_free_mem_stmt command returns a C++ statement that frees the memory associated with the specified parameter (or return value) of an operation. If there is no need to free memory for the parameter (for example, if is_var is 1 or the parameter's type/direction does not require any memory management) then this command returns an empty string.

The cpp_clt_need_to_free_mem command returns a boolean indicating whether or not there is any need to free the memory of the specified parameter (or return value) of an operation.

The examples in the following subsections illustrate the use of these commands. In each of the examples, the following IDL is assumed:

```
// 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);
};
```

## Declaring Variables to Hold Parameters and the Return Value

The Tcl script below illustrates how to declare C++ variables to be used as parameters to (and 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
}
```

**Line 1** `cpp_gen_clt_par_decl $arg $is_var $ind_lev`

**Line 2** `cpp_gen_clt_par_decl $op $is_var $ind_lev`

Notice how the command `cpp_gen_clt_par_decl` is used to declare variables for both parameters (line 1) and the return value (line 2). The above Tcl code produces the following C++:

```
//--------
// Declare parameters for operation
//--------
widget p_widget;
```

**Line 3** `char * p_string;`

**Line 4** `longSeq* p_longSeq;`

`long_array p_long_array;`

**Line 5** `longSeq* _result;`

The name of the C++ variable declared for holding the return value (line 5) is determined by `$pref(cpp,ret_param_name)`. Its default value is `_result`. The C++ variables declared in lines 3, 4 and 5 are raw pointers. This is because in the calls to `cpp_gen_clt_par_decl`, the `is_var` parameter had the value `0` (false). If `is_var` was `1` (true) then the variables declared at lines 3, 4 and 5 would have been `_var` types.

## Initializing Input Parameters

The Tcl script below illustrates how to initialize `in` and `inout` parameters:

```
# Tcl
[***
   //--------
   // Initialize "in" and "inout" parameters
   //--------
***]
```

**Line 1**
```
foreach arg [$op args {in inout}] {
   set type [$arg type]
```

**Line 2**
```
   set arg_ref [cpp_clt_par_ref $arg $is_var]
   set value "other_[$type s_uname]"
```

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

The `foreach`-loop (line 1) iterates over all the `in` and `inout` parameters. The command `cpp_clt_par_ref` (line 2) is used to obtain a reference to a parameter, and this reference can then be used to initialize the parameter with the `cpp_gen_assign_stmt` command (line 3). The above Tcl code produces the following C++:

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

## Invoking an IDL Operation

Continuing on the example, the Tcl script below illustrates how to invoke an IDL operation, passing parameters and assigning the return value to a variable:

```
# Tcl
```

**Line 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"
```

**Line 2**
```
set call_args [idlgen_process_list $arg_list \
                        cpp_l_name $start_str $sep_str]
```

```
[***
   //--------
   // Invoke the operation
   //--------
   try {
Line 3        @$ret_assign@obj->@$op_name@(@$call_args@);
   } catch(CORBA::Exception &ex) {
      ... // handle the exception
   }
***]
```

The above Tcl code produces the following C++:

```
//--------
// Invoke the operation
//--------
try {
Line 4        _result = obj->op(
Line 5        p_widget,
Line 6        p_string,
Line 7        p_longSeq,
Line 8        p_long_array);
} catch(CORBA::Exception &ex) {
   ... // handle the exception
}
```

Two points are worth noting about the Tcl script that produced the above output:

- The text `_result =` (line 4 of the C++ code) is produced by the command `[cpp_ret_assign $op]` (at lines 1 and 3 in the Tcl script). If the operation invoked does not have a return type then `[cpp_ret_assign $op]` returns an empty string.
- You can format the parameters to an operation call (lines 5-8 in the C++ code) with the command `idlgen_process_list` (used at lines 2 and 3 in the Tcl script). This command is discussed in "Idlgen_process_list" on page 131.

## Processing Output Parameters and the Return Value

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

```
# Tcl
[***
   //--------
   // Process the returned parameters
   //--------
***]
```

**Line 1**

```
foreach arg [$op args {out inout}] {
   set type [$arg type]
   set name [cpp_l_name $arg]
```

**Line 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"} {
```

**Line 3**

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

The `foreach`-loop at line 1 iterates over all the `out` and `inout` parameters. Notice how the `cpp_clt_par_ref` command can be used to obtain references to both parameters (line 2) and the return value (line 3). The above Tcl code produces the following C++:

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

## Memory Management

The Tcl script below illustrates how to free memory associated with the parameters and return value of an operation call:

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

**Line 1**

**Line 2** `cpp_gen_clt_free_mem_stmt $op $is_var $ind_lev`

Notice how the command `cpp_gen_clt_free_mem_stmt` is used to free memory both for parameters (line 1) and the return value (line 2). The above Tcl code produces the following C++:

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

It can be seen in the produced output that 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. Also, if the `is_var` parameter had the value `1` (indicating that the parameters and `_result` variable had been declared as `_var` types, that is, smart pointers) then no memory-freeing statements would have been generated.

## Processing Implicit Parameters to Attributes

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

`cpp_clt_par_decl` *`arg_or_op is_var`*

This command is used to declare a client-side variable to be used as a parameter to (or return value of) an operation. Similar functionality is needed to declare a client-side variable to be used as the implicit parameter to (or return value of) an attribute. This additional functionality is obtained by implementing the `cpp_clt_par_decl` command in a way that allows it to be invoked with either two arguments (as indicated above) or with four arguments, as shown below:

```
cpp_clt_par_decl name type dir is_var
```

In this case, the argument `arg_or_op` has been replaced with three arguments that specify the attribute's `name`, `type` and direction (`dir`). The `dir` argument should be `in` or `return`, for an attribute's modifier and accessor, respectively. This convention of replacing `arg_or_op` with three arguments is also used in the other commands for the client-side processing of parameters. Thus, the full collection of commands for processing the implicit parameter/return value for an attribute is:

```
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
```

It also applies to the `gen_` counterparts:

```
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
```

## Server-Side Processing of Parameters

The `std/cpp_boa_lib.tcl` library provides the following commands to process parameters (and the return value) inside the body of an operation:

```
cpp_srv_ret_decl op ?alloc_mem?
cpp_srv_par_alloc arg_or_op
cpp_srv_par_ref arg_or_op
cpp_srv_free_mem_stmt arg_or_op
cpp_srv_need_to_free_mem rg_or_op
```

Some of the above commands have `gen_` counterparts:

```
cpp_gen_srv_ret_decl op ind_lev ?alloc_mem?
cpp_gen_srv_par_alloc arg_or_op ind_lev
cpp_gen_srv_free_mem_stmt arg_or_op ind_lev
```

In all of the above commands, the arg_or_op parameter can be either an argument node or an operation node in the parse tree. If arg_or_op is an argument node then the above commands apply to a parameter of an operation call. Conversely, if arg_or_op is an operation node then the above commands apply to the return value of an operation.

The cpp_srv_ret_decl 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 then this command returns an empty string. Assuming that the operation does have a return value, if alloc_mem is 1 then the variable declaration also allocates memory to hold the return value, if necessary. If alloc_mem is 0 then 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.

The cpp_srv_par_alloc 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 then this command returns an empty string.

The cpp_srv_par_ref 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.

The cpp_srv_free_mem_stmt command returns a C++ statement that frees the memory associated with the specified parameter (or return value) of an operation. If there is no need to free memory for the parameter then this command returns an empty string. Note that 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 may be necessary to release the previous value of the parameter.
- If the body of the operation decides to throw an exception *after* having allocated memory 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 had previously been allocated. Note that if the exception is thrown before having allocated memory for the out parameters and the return value, then no memory management is necessary.

The cpp_srv_need_to_free_mem command returns a boolean indicating whether or not there is any need to free the memory of the specified parameter (or return value) of an operation.

The examples in the following subsections illustrate the use of these commands. In each of the examples, the following IDL is assumed (this is the same IDL used previously in "Client-Side Processing of Parameters" on page 165):

```
// 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);
};
```

## Declaring the Return Value and Allocating Memory for Parameters

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

**Line 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
}
```

**Line 2** `cpp_gen_srv_par_alloc $op $ind_lev`

The output of the above Tcl is as follows:

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

Note that the declaration of the `_result` variable (line 1) is separated from the allocation of memory for it (line 2). This gives us an 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 then change line 1 of the Tcl script so that it passes `1` as the value of the `alloc_mem` parameter, and then delete line 2 of the Tcl script because it is no longer needed. If you make these changes then the declaration of `_result` is changed to:

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

## Initializing Output Parameters and the Return Value

The next Tcl script iterates over all the `inout` and `out` parameters, and the return value, to assign values to them. Comments follow after the script:

```
# 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 [cpp_srv_par_ref $arg]
      set name2   "other_[$type s_uname]"
      if {[$arg direction] == "inout"} {
      cpp_gen_srv_free_mem_stmt $arg $ind_lev
      }
      cpp_gen_assign_stmt $type $arg_ref $name2 \
                          $ind_lev 0
}
if {[$ret_type l_name] != "void"} {
      set ret_ref [cpp_srv_par_ref $op]
      set name2   "other_[$ret_type s_uname]"
      cpp_gen_assign_stmt $ret_type $ret_ref \
                            $name2 $ind_lev 0
}
```

**Line 1**

**Line 2**

**Line 3**

**Line 4**

**Line 5**

The command `cpp_srv_par_ref` (lines 1 and 4) can be used to obtain a reference to both 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 as to initialize it with the `cpp_gen_assign_stmt` command (lines 3 and 5).

It is sometimes necessary to first free the old value associated with an `inout` parameter before assigning it a new value. This can be achieved by using the `cpp_gen_srv_free_mem_stmt` command (line 2). However, note that this should be done only for `inout` parameters; hence the use of an `if` statement around this command.

The output generated by the above Tcl code is as follows:

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

## Memory Management when Throwing Exceptions

If an operation decides to throw an exception after it has already allocated memory for out parameters and the return value, then some memory management duties must be carried out before throwing the exception. These duties are illustrated 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}] {
     set free_mem_stmt [cpp_srv_free_mem_stmt $arg]
     if {$free_mem_stmt != ""} {
        set name [cpp_l_name $arg]
        set type [$arg type]
[***
        @$free_mem_stmt@;
        @$name@ = @[cpp_nil_pointer $type]@;
***]
     }
}
cpp_gen_srv_free_mem_stmt $op 2
[***
        throw some_exception;
     }
***]
```

**Line 1**

**Line 2**

**Line 3**

The above script illustrates how the cpp_srv_free_mem_stmt and cpp_gen_srv_free_mem_stmt commands (lines 1 and 3, respectively) can be used to 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 output of this Tcl script is as follows:

```
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;
}
```

## Processing Implicit Parameters to Attributes

Recall that the cpp_srv_par_alloc command is defined as follows:

cpp_srv_par_alloc *arg_or_op*

This command is used to allocate memory, if necessary, for an out parameter or return value of an operation. Similar functionality is needed to allocate memory for the return value of the accessor function for an attribute. This additional functionality is obtained by implementing the cpp_srv_par_alloc command in a way which allows it to be invoked with either one argument (as indicated above) or with three arguments, as shown below:

cpp_srv_par_alloc *name type dir*

In this case, the argument arg_or_op has been replaced with three arguments that specify the attribute's name, type and direction (dir). The dir argument should be return for an attribute's *accessor*. This convention of replacing arg_or_op with several arguments is also used in the other commands for the server-side processing of parameters. Thus, the full collection of commands for processing the implicit parameter/return value for an attribute is:

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

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 dir ind_lev
cpp_gen_srv_free_mem_stmt name type dir ind_lev
```

## Processing Instance Variables and Local Variables

Previous sections have discussed how to process variables used for parameters and the return value of an operation call. However, not all variables are used as parameters. For example, a C++ class that implements an IDL interface may contain some *instance variables* that are not used as parameters; or the body of an operation may 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` command returns a C++ variable declaration with the specified `name` and `type`. 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 then it must be heap allocated, and the `is_var` parameter determines whether the variable will be declared as a `_var` (smart pointer) type or as a raw pointer. Note that all variables declared via `cpp_var_decl` are references and hence can be used directly with `cpp_assign_stmt`.

The `cpp_var_free_mem_stmt` 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 (for example, if `is_var` is `1` or the variable's type does not require any memory management) then this command returns an empty string.

The `cpp_var_need_to_free_mem` command returns a boolean indicating whether or not there is any need to free memory with a variable of the specified `type`.

There are also some gen_ counterparts to some of the above commands:

```
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 illustrates the use of these gen_ 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]"
      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]"
      cpp_gen_assign_stmt $type $name $value $ind_lev 0
}
[***

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

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

```
// 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()
```

Note that the `cpp_gen_var_free_mem_stmt` command generated memory-freeing statements only for the `my_string` variable. This is because the other variables were stack-allocated and hence did not require their memory to be freed. Modifying the Tcl code so that `is_var` is set to `1` would change the type of `my_string` from `char *` to `CORBA::String_var` and would have suppressed the memory-freeing statement for that variable.

# Processing Unions

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 commands `cpp_branch_case_s_label` and `cpp_branch_case_l_label` are provided to help with this task. However, sometimes you may want to process an IDL `union` using a different C++ construct, such as an `if-then-else` statement. The slightly lower-level commands `cpp_branch_s_label` and `cpp_branch_l_label` are provided to help with this task.

## cpp_branch_case_s_label

This command is used as follows:

`cpp_branch_case_s_label` *`union_branch`*

This command returns a string in the form `case label` where `label` is the (fully-scoped) label for that branch of the union, or the string `default` if it is the default branch in the union. As an example, 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 generates a C++ `switch` statement to process the union:

```
# Tcl
set union [$idlgen(root) lookup "m::foo"]
[***
void some_func()
{
      switch(u._d()) {
***]
foreach branch [$union contents {union_branch}] {
      set name [cpp_l_name $branch]
```

```
      set case_label [cpp_branch_case_s_label $branch]
[***
      @$case_label@:
         ... // process u.@$name@()
         break;
***]
}; # foreach
[***
      };
} // some_func()
***]
```

The code generated from the above Tcl is as follows:

```
// 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 command `cpp_branch_case_s_label` works irrespective of the type of the union's discriminant. For example, if the discriminant is, say, type `long` then this command will return a string of the form `case 42` (where `42` is the value of the case label), or if the discriminant is type `char` then this command will return a string of the form `case 'a'`.

## cpp_branch_case_l_label

This command is used as follows:

`cpp_branch_case_l_label` *`union_branch`*

It works almost identically to `cpp_branch_case_s_label` except that it produces the non-scoped label of the union's case. For example, instead of returning `case m::red`, it returns `case red`.

## cpp_branch_s_label

This command is used as follows:

`cpp_branch_s_label` *`union_branch`*

It works almost identically to `cpp_branch_case_s_label` except that the `case` prefix is not included in the returned value.

Note that the command `cpp_branch_case_s_label` is slightly easier to use if you are generating a C++ switch statement to process a union. The command `cpp_branch_s_label` could, however, be used if you wanted to generate a C++ `if-then-else` statement to process a union.

## cpp_branch_l_label

This command is used as follows:

`cpp_branch_l_label` *`union_branch`*

It works almost identically to `cpp_branch_s_label` except that it produces the non-scoped value of the union's case. For example, instead of returning `m::red`, it returns `red`.

# Processing Arrays

Arrays are usually processed in C++ by 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 of type `long_array`. C++ code to perform an element-wise copy from `bar` into `foo` might be written as follows:

```
// C++
void some_func()
{
```

**Line 1** `   CORBA::ULong                    i1;`

**Line 1** `   CORBA::ULong                    i2;`

**Line 2** `   for (i1 = 0; i1 < 5; i1 ++) {`

**Line 2** `      for (i2 = 0; i2 < 7; i2 ++) {`

**Line 3** `         foo[i1][i2] = bar[i1][i2];`

**Line 4** `      }`

**Line 4** `   }`

```
}
```

In order to write a Tcl script to generate the above C++ code, you need Tcl commands that *declare* the index variables (lines marked 1), generate the *header* of the `for`-loop (lines marked 2), provide the *index* for each element of the array (`[i1][i2]` in the above example, as used in line 3), and generate the *footer* of the `for`-loop (lines marked 4). The following commands provide exactly 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
```

In each of these commands, the following conventions hold:

- `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.

As a concrete example, the following Tcl script generates the for-loop shown previously:

```
# 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]@
}
***]
```

**Line 5** `set indent [cpp_indent [$a num_dims]]`

**Line 3** `set index [cpp_array_elem_index $a "i"]`

**Line 1** `@[cpp_array_decl_index_vars $a "i" 1]@`

**Line 2** `@[cpp_array_for_loop_header $a "i" 1]@`

**Line 3** `@$indent@foo@$index@ = bar@$index@;`

**Line 4** `@[cpp_array_for_loop_footer $a 1]@`

The amount of indentation to be used *inside* the body of the for-loop (at line 3) is calculated by using the number of dimensions in the array as a parameter to the cpp_indent command (line 5).

The cpp_array_for_loop_header command takes a boolean parameter called decl, which has a default value of 0. If decl has the value 1 then the index variables will be 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 does not need to use 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 Anys

The commands to process type any are split into two categories:

- Those used to insert a value into an any.
- Those used to extract a value from an any.

## Inserting Values into an Any

Use the cpp_any_insert_stmt command to generate 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 as follows:

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

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

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

## Extracting Values from an Any

The following commands are provided to help you write Tcl scripts that extract values from an any:

```
cpp_any_extract_var_decl type name
cpp_any_extract_var_ref type name
cpp_any_extract_stmt type any_name name
```

The cpp_any_extract_var_decl command is used to declare a variable into which values from an any will be extracted. The parameters to this command are the variable's type and name. Note that if the value to be extracted from the any is a small value (such as a short, long, boolean, and so on), then the variable is declared as a normal variable of the specified type. However, if the value is of a larger type (struct, sequence, and so on) then the variable is declared as a pointer to the specified type.

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

The cpp_any_extract_var_ref command is used to extract a value of the specified type from the any called any_name into the variable called name.

The following example illustrates the use of 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 types `widget` (a `struct`), `boolean` and `long_array` 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);
}
```

# 11

# Other Tcl Libraries for C++ Utility Functions

*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++". 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:

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

The minimal API defines the following command:

`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 then use the following command:

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

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

```
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.cc`, respectively. The *`full_any`* parameter to `gen_cpp_print_func_cc` is explained below.

The Orbix runtime system has built-in TypeCodes for the basic IDL types such as `long`, `short`, `string`, and so on. However, by default, the Orbix IDL compiler does *not* generate TypeCodes for user-defined IDL types. Without these TypeCodes, 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 TypeCodes, 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 TypeCodes.

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 TypeCodes 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 usage of the `cpp_print` library.

```
# Tcl
smart_source "std/sbs_output.tcl"
smart_source "std/cpp_boa_lib.tcl"
* smart_source "cpp_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_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_variable_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:

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

The minimal API defines the following commands:

```
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 then use the following command:

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

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

```
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.cc`, 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_print/lib-full.tcl"
```

```
+       smart_source "cpp_random/lib-full.tcl"
```

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

#--------
# Generate it_print_funcs.{h,cc}
#--------
```

```
*       gen_cpp_print_funcs_h
```

```
*       gen_cpp_print_funcs_cc 1

#--------
# Generate it_random_funcs.{h,cc}
```

```
            #--------
+           gen_cpp_random_funcs_h
+           gen_cpp_random_funcs_cc 1

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

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

The minimal API defines the following commands:

```
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 (in)equality.

## Example of Use

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

```
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:

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

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

```
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.cc, 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 4

# OrbixWeb Genies: Library Reference

# 12

# The Java Development Library

*The Orbix Code Generation Toolkit comes with a rich Java development library that makes it easy to create genies to map IDL onto Java code.*

The file `std/java_boa_lib.tcl` is a library of Tcl procedures that map IDL constructs into their Java counterparts.

## Naming Conventions in API procedures

Abbreviations are commonly used in the names of procedures defined in the `std/java_boa_lib.tcl` library. The following table lists these abbreviations and their meanings:

| Abbreviation | Meaning |
|---|---|
| `clt` | Client. |
| `srv` | Server. |
| `gen_` | Discussed below. |
| `par (or param)` | Parameter. |
| `ref` | Reference. |

**Table: 12.1:** *Abbreviations Used in Procedure Names*

| Abbreviation | Meaning |
|---|---|
| stmt | Statement. |
| mem | Memory. |
| op | Operation. |
| attr_acc | An attribute's accessor. |
| attr_mod | An attribute's modifier. |
| sig | Signature. |

**Table: 12.1:** *Abbreviations Used in Procedure Names*

The names of all the procedures in `std/java_boa_lib.tcl` start with `java_`, which implies Java.

As an example, the following statement assigns the Java signature of an operation to variable `foo`:

```
set foo [java_op_sig $op]
```

## Naming Conventions for "gen_"

Some procedures contain `gen_` in their names. Such procedures *generate* output. For example, `java_gen_op_sig` outputs the Java signature of an operation. Procedures whose names do not contain `gen_` *return* a value (which you can use as a parameter to the `output` command if you wish).

Some procedures whose names do not contain `gen_` also have `gen_` counterparts. The reason for providing both forms of a procedure is to offer flexibility in how you can write genies. In particular, the procedures 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 of textual blocks. Some examples can help to illustrate this.

The following segment of code prints the Java signatures of all the operations of an interface:

```
foreach op [$inter contents {operation}] {
   output "\t[java_op_sig $op];\n"
}
```

Note that the output statement uses a TAB character ('\t') to indent the signature of the operation, and also follows the signature with a semicolon and newline character. The printing of all this white space and syntactic baggage is automated by the gen_ counterpart of this procedure, so the above code snippet could be rewritten in the following, slightly more concise format:

```
foreach op [$inter contents {operation}] {
   java_gen_op_sig $op
}
```

The java_gen_ procedures tend to be useful inside foreach loops to, for example, declare operation signatures or variables. However, when generating the bodies of operations in .java 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 procedure and then enter another textual block to print more text. For example:

```
[***
//--------
// Function: ...
//--------
***]
java_gen_op_sig $op
[***
{
   ... // body of the operation
}
***]
```

The use of *non*-gen_ procedures 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:

```
[***
//--------
// Function: ...
//--------
@[java_op_sig $op]@
{
   ... // body of the operation
}
***]
```

## Indentation

Consistent indentation is important for code clarity. However, there are no universally accepted rules for indentation: some programmers use two spaces for each level of indentation, while other programmers use four or eight spaces, or perhaps use a TAB character.

If the procedures in `std/java_boa_lib.tcl` obeyed a particular indentation policy, say, four spaces for each level of indentation, then this would suit Java programmers who use the same indentation policy in their genies. It would, however, be frustrating for people who prefer a different indentation policy. To avoid this problem, the amount of white space to be used for one level of indentation is held in the variable `$pref(java,indent)`. Any procedure in `std/java_boa_lib.tcl` which needs to print some indentation uses the string specified in `$pref(java,indent)`.

The default value for `$pref(java,indent)` is three spaces, but you can change it to be a different value such as four or eight spaces.

Some procedures take a parameter called `ind_lev`. This parameter is an integer which specifies the indentation level at which output should be generated. To illustrates this, consider the following code:

```
# Tcl
set name "foo"
set type [$idlgen(root) lookup "string"]
for {set ind_lev 0) ($ind_lev < 3) (incr ind_lev} {
   java_param_decl $name $type $ind_lev
}
```

The `java_param_decl` procedure declares a variable of the specified `name` and `type`, at the specified level of indentation. The output from the above code would look something like:

```
String foo;
   String foo;
      String foo;
```

## $pref(java,...) Entries

Some entries in the `$pref(...)` array are used to specify various user preferences for the generation of Java code. All of these entries are given default values by `std/java_boa_lib.tcl`, but the default values can be over-ridden by corresponding entries in the `idlgen.cfg` file or by explicit assignment in a genie.

| $pref(...) Array Entry | Purpose |
|---|---|
| `$pref(java,java_file_ext)` | Specifies the filename extension to be on Java files. Its default value is `.java`. |
| `$pref(java,indent)` | Specifies the amount of white space to be used for one level of indentation. Its default value is two spaces. |
| `$pref(java,impl_class_suffix)` | Specifies the suffix that is used to obtain the name of a class that implements an IDL interface. Its default value is `Impl`. |
| `$pref(java,smart_proxy_prefix)` | Specifies the prefix that is used to obtain the name of a smart proxy class for an IDL interface. Its default value is `Smart`. |
| `$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 `1`. |
| `$pref(java,attr_mod_param_name)` | Specifies the name of the parameter in the Java signature of an attribute's modifier operation. Its default value is `_new_value` |
| `$pref(java,ret_param_name)` | Specifies the name of variable which is to be used to hold the return value from a non-`void` operation call. Its default value is `_result`. |

| $pref(...) Array Entry | Purpose |
|---|---|
| `$pref(java,max_padding_for_types)` | This is used to pad out Java type names when declaring variable or parameters. This padding helps to ensure that the names of variables/parameters are vertically aligned, which makes code easier to read. Its default value is 32. |

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

The rules for mapping IDL to Java code are clearly specified so it is easy to encapsulate these rules into procedures in the Tcl language. These procedures and those specific to OrbixWeb code generation comprise the Java development library for IDLgen and make genies easy to develop.

# Identifiers and Keywords

You cannot use the underscore character as the first character when choosing operation names in IDL, because the generated code for the interface can contain class functions that conflict with the operations defined at the `CORBA::Object` level. When writing or generating code from the IDL interface make sure that the IDL identifiers do not conflict with the built-in language keywords.

Consider this unusual, but valid, interface:

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

The interface maps to a Java class with a class method defined as:

```
// Java
String for( int while );
```

There are a number of procedures that help in mapping the IDL data types to their language equivalents.

## java_l_name

This command is used as follows:

`java_l_name` *node*

This procedure returns the Java mapping of a node's local name. This is usually the node's local name itself, but it is prefixed with an underscore if the local name conflicts with a Java keyword. Also, if the node happens to be a built-in type then the result is the Java mapping of the type.

For example, consider this IDL:

```
// IDL
interface for {
   exception new {};
   void while( in octet goto );
};
```

If each construct (node) of this IDL is run through the `java_l_name` procedure the returned value is as follows:

```
for         _for
new         _new
while       _while
```

The `java_l_name` procedure also maps the built in IDL data types to the corresponding Java types. For example, the basic data types map as follows:

| | |
|---|---|
| `short` | `short` |
| `ushort` | `short` |
| `long` | `int` |
| `ulong` | `int` |
| `longlong` | `long` |
| `ulonglong` | `long` |
| `float` | `float` |
| `double` | `double` |
| `boolean` | `boolean` |
| `char` | `char` |
| `octet` | `byte` |

| | |
|---|---|
| string | java.lang.String |
| wstring_t | java.lang.String |
| any | org.omg.CORBA.Any |
| wchar_t | char |
| void | void |
| Object | org.omg.CORBA.Object |
| TypeCode | org.omg.CORBA.TypeCode |
| Principal | java.lang.String |
| NamedValue | org.omg.CORBA.NamedValue |

## java_s_name

This command is used as follows:

java_s_name *node*

This procedure performs the same kind of functionality as the java_l_name procedure. The difference is that it returns the fully-scoped name rather than the local name. If the IDL on page 207 in run through java_s_name the result is as follows:

```
for         _for
new         _for._new
while       _for._while
```

So for example consider this IDL:

```
//IDL
module outer{
   interface inner{...
   };
};
```

This maps as follows:

| | |
|---|---|
| java_s_name node | outer.inner |
| java_l_name node | inner |

Built-in IDL types are mapped as they are in the java_l_name procedure.

## java_typecode_s_name

This command is used as follows:

`java_typecode_s_name` *type*

This procedure returns the fully-scoped Java name of the `typecode` for the specified `type`. `Typecodes` are usually formed by suffixing the name of the type with `Helper.type()`, but there are some exceptions. In particular, the `typecodes` for the built-in types (`long`, `short`, and so on) are defined inside the CORBA module.

Examples of the fully-scoped names of Java `typecodes` for IDL types:

| | |
|---|---|
| `cow` | `cowHelper.type()` |
| `farm::cow` | `farm.cowHelper.type()` |
| `long` | `ORB.init().get_primitive_tc(org.omg.C ORBA.TCKind._tk_long)` |

## java_typecode_l_name

This command is used as follows:

`java_typecode_l_name` *type*

This procedure returns the local Java name of the typecode for the specified `type`. Typecodes are usually formed by suffixing the name of the type with `Helper.type()`, but there are some exceptions. In particular, the typecodes for the built-in types (`long`, `short` and so on) are defined inside the CORBA module.

Examples of the local names of Java typecodes for IDL types:

| | |
|---|---|
| `cow` | `cowHelper.type()` |
| `farm::cow` | `farm.cowHelper.type()` |
| `long` | `org.omg.CORBA.ORB.init().get_primitive _tc(org.omg.CORBA.TCKind._tk_long)` |

# General Purpose Procedures

There are also a number of general purpose procedures that can be used to help write genies.

## java_is_keyword

This command is used as follows:

`java_is_keyword` *`name`*

This command returns a boolean value that indicates whether or not the specified `name` is a Java keyword. For example:

```
# Tcl
java_is_keyword "new"; # returns 1
java_is_keyword "cow"; # returns 0
```

This command is called internally from other commands in the `std/java_boa_lib.tcl` library. However, it is unlikely that you will need to make use of it directly in your own genies.

## java_assign_stmt

This command is used as follows:

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

This command returns a Java statement that assigns the specified `value` to the variable of the specified `name` and `type`. 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 reason why the `ind_lev` (indentation level) parameter is required is that the returned `for` loop spans several lines of code and these lines of code need to be indented consistently. The `scope` parameter is unused.

There is a `gen_` counterpart to the `java_assign_stmt` command:

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

The following example illustrates the use of this `gen_` command:

```
#Tcl
```

```
set ind_lev 1
[***
void some_func()
{
***]
foreach type $type_list {
      set name  "my_[$type l_name]"
      set value "other_[$type l_name]"
      java_gen_assign_stmt $type $name $value $ind_lev 0
}
 [***
} // some_func()
***]
```

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

```
// Java
void some_func()
{
      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()
```

## java_indent

This command is used as follows:

`java_indent` *`number`*

This command returns a string which is the string `$pref(java,indent)` concatenated with itself the specified `number` of times. For example:

```
#Tcl
puts "[java_indent 1]One"
puts "[java_indent 2]Two"
puts "[java_indent 3]Three"
```

This produces output in the following form:

```
One
   Two
```

```
Three
```

## java_nil_pointer

This command is used as follows:

`java_nil_pointer` *type*

This command returns a Java expression that is a nil pointer (or a nil object reference) for the specified `type`.

It should be used *only* for types that might be heap-allocated, that is, `struct`, `exception`, `union`, `sequence`, `array`, `string`, `Object`, `interface` or `TypeCode`. If used for any other types, for example, a `long`, then this command throws an exception.

# Interfaces

One of the major encapsulating constructs in IDL is the *interface.* This maps to an appropriately named class in Java. There are a number of procedures that aid in generating code for interfaces.

## java_impl_class

This command is used as follows:

`java_impl_class` *interface_node*

This procedure returns the name of a Java class that can be used to implement the specified IDL interface. The class name is constructed by getting the fully scoped name of the IDL interface and replacing all occurrences of :: with . (that is flattening the namespace). It also appends `$pref(java,impl_class_suffix)` on to the end. The default value for this is `Impl`.

```
# Tcl
set class [java_impl_class $inter]
set base [java_boa_class_s_name $inter]
[***
package @[java_package_name $inter]@;
class @$class@ extends @$base@ {
   public @$class@(){
```

```
... };
};
***]
```

For example, the following interface definitions result in the generation of the corresponding Java code.

```
// IDL                    // Java
interface cow {           class cowImpl extends cowImplBase{
   ...                       public cowImpl(){
};                        ... };
                          };
```

```
// IDL                    // Java
module farm {             package farm
   interface cow {        class cowImpl extends cowImplBase{
      ...                    public cowImpl(){
   };                     ... };
};                        };
```

## java_tie_class

This command is used as follows:

`java_tie_class` *`interface_node`*

This procedure returns the name of the Java class corresponding to the given IDL interface using the TIE approach.

```
# Tcl
set class [java_impl_class $inter]
[***
package @$[java_package_name $inter]@
class @$class@ implements @[java_tie_class]@{
   public @$class@(){
   ...};
};
***]
```

For example, the following interface definitions result in the generation of the corresponding Java code.

```
// IDL                    //Java
interface cow {           class cowImpl implements
   ...                    cowOperations{
};                           public cowImpl(){
                          ...};
                          };
```

```
// IDL                    // Java
module farm{              package farm
   interface cow {        class cowImpl implements
      ...                 cowOperations{
   };                        public cowImpl(){
};                        ... };
                          };
```

## java_boa_class_s_name

This command is used as follows:

`java_boa_class_s_name` *`interface_node`*

This command is exactly the same as `java_tie_class` except that this procedure returns the fully scoped name of the BOA class that can be used to implement an IDL interface. For example:

```
# Tcl
set class [java_impl_class if_node]
[***
class @$class@ extends @[java_boa_class_s_name if_node]@ {
   public @$class@(){
... };
};

***]
```

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

```
// IDL                    // Java
interface cow {           class cowImpl extends
   ...                    cowImplBase {
};                           public cowImpl(){
                          ... };
                          };
```

## java_boa_class_l_name

This command is used as follows:

`java_boa_class_l_class` *`interface_node`*

This procedure returns the local name of the BOA class that can be used to implement an IDL interface.

Note that this command is rarely used; the `java_boa_class_s_name` is normally used instead.

## java_smart_proxy_class

This command is used as follows:

`java_smart_proxy_class` *`interface_node`*

This procedure returns the name of a Java class that can be used when constructing a smart proxy class for a given IDL interface. The class name is constructed by obtaining the fully-scoped name of the IDL interface and replacing all occurrences of `::` with `.` (that is, flattening the namespace). It also prefixes the interface name with `$pref(java,impl_class_suffix)`. The default entry for this is `Smart`.

```
# 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@(){
```

```
... };
};
***]
```

For example, the following interface definitions result in the generation of the corresponding Java code.

```
// IDL                  // Java
interface cow {         class Smartcow extends cow {
   ...                     public Smartcow(){
};                      ... };
                        };


// IDL                  // Java
module farm {           package farm
   interface cow {      class Smartcow extends cow {
      ...                  public Smartcow(){
   };                   ... };
};                      };
```

# Signatures of Operations

The following commands are used to obtain the Java signature of an operation:

`java_op_sig` *`operation_node ?class_name?`*
`java_gen_op_sig` *`operation_node ?class_name?`*

The `gen_` variant generates output, while the *non*-`gen_` variant returns the Java signature as a string (which you can then `output` if you want).

These commands take an optional `class_name` parameter. If this parameter is not specified, it is calculated as:

```
set class_name [java_impl_class [$op defined_in]]
```

This is usually the desired class name. However, the ability to use an alternative class name is provided in case you want to generate signatures of operations for, as an example, smart proxies or some other support class.

# Signatures of Attributes

The following set of commands is used to obtain the Java signatures of accessor and modifier functions for attributes:

```
java_attr_acc_sig_ attribute_node ?class_name?
java_gen_attr_acc_sig attribute_node ?class_name?
java_attr_mod_sig attribute_node ?class_name?
java_gen_attr_mod_sig attribute_node ?class_name?
```

You can determine which command to use by the different elements in the command name. Table 12.3 describes the different name elements:

| Element in command name | Command use |
|---|---|
| acc | Variants are for attribute *accessor* functions. |
| mod | Variants are for attribute *modifier* functions. |
| gen_ | Variants of these commands *generate* output. |
| *non*-gen_ | Variants *return* the Java signature as a string (which you can then `output` if you want). |

**Table: 12.3:** *Attribute Signature Commands*

# Types and Signatures of Parameters

Previous sections have discussed commands that can be used to generate Java signatures of IDL operations and attribute accessor/modifier functions. However, sometimes you may want more control over the construction of an operation's signature. In order to do this, you need to be able to determine the *type* or *signature* of individual parameters. The following commands are provided for this purpose:

```
java_param_type op_or_arg
java_param_type type dir
java_param_sig  arg
java_param_sig  name type dir
```

The `java_param_type` command returns the Java type of a parameter of the specified `type` and direction (`dir`). For example, the following snippet of Tcl prints out `String`:

```
# Tcl
set type [$idlgen(root) lookup "string"]
set dir "in"
puts "[java_param_type $type $dir]"
```

The `java_param_sig` command returns the Java signature of a parameter of the specified `name, type` and direction (`dir`). The signature is composed of the Java type and the parameter's name. For example, consider the following snippet of Tcl code:

```
# Tcl
set type [$idlgen(root) lookup "string"]
set dir in"
puts "[java_param_sig “foo” $type $dir]"
```

The output generated from the above code is:

```
String foo;
```

There is some white-space padding between the parameter's type and name. The amount of padding is determined by `$pref(java,max_padding_for_types)`. This padding is used to ensure vertical alignment of parameter names.

You can use an `argument` node or an `operation` node (the latter indicating the operation's return type) in these commands instead of specifying `type` and `dir` separately.

## Client-Side Processing of Parameters

The std/java_boa_lib.tcl library provides commands to manipulate client-side variables that are used as parameters to, or the return value of, an operation call. The commands provided are as follows:

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

Some of the above commands have gen_ counterparts:

```
java_gen_clt_par_decl arg_or_op is_var ind_lev
```

In all of the above commands, the arg_or_op parameter can be either an argument node or an operation node in the parse tree. If arg_or_op is an argument node then the above commands apply to a parameter of an operation call. Conversely, if arg_or_op is an operation node then the above commands apply to the return value of an operation call.

The java_clt_par_decl command returns a Java declaration of a variable that can be used as a parameter to (or return value of) an operation. However, if the parameter is of a out or inout type the parameter is declared as a Holder type.

The java_clt_par_ref command returns a reference to the value of the specified parameter (or return value) of an operation.

The examples in the following subsections illustrate the use of these commands. In each of the examples, the following IDL is assumed:

```
// 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_outlongSeq,
      out long_array p_long_array,
      in longSeq    p_inlongSeq;
```

## Declaring Variables to Hold Parameters and the 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
set op         [$idlgen(root) lookup "foo::op"]
set ind_lev    1
set arg_list   [$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
```

**Line 1**

**Line 2**

Notice how the command `java_gen_clt_par_decl` is used to declare variables for both parameters (line 1) and the return value (line 2). The above Tcl code produces the following Java:

```
//--------
// Declare parameters for operation
//--------
widget              p_widget;
StringHolder        p_string;
longSeqHolder       p_outlongSeq;
long_arrayHolder    p_long_array;
longSeqHolder       _result;
int[]               p_outlongseq
```

The name of the Java variable declared for holding the return value is determined by `$pref(java,ret_param_name)`. Its default value is `_result`.

## Initializing Input Parameters

The Tcl script below illustrates how to initialize `in` and `inout` parameters:

```
# Tcl
[***
```

```
   //--------
   // Initialize "in" and "inout" parameters
   //--------
***]
```

**Line 1**

```
foreach arg [$op args {in inout}] {
set type [$arg type]
```

**Line 2**

```
set arg_ref [java_clt_par_ref $arg $is_var]
set value "other_[$type s_uname]"
```

**Line 3**

```
java_gen_assign_stmt $type $arg_ref $value $ind_lev 0
}
```

The `foreach`-loop (line 1) iterates over all the `in` and `inout` parameters. The command `java_clt_par_ref` (line 2) is used to obtain a reference to a parameter, and this reference can then be used to initialize the parameter with the `java_gen_assign_stmt` command (line 3). The above Tcl code produces the following Java:

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

## Invoking an IDL Operation

Continuing on the example, the Tcl script below illustrates how to invoke an IDL operation, passing parameters and assigning the return value to a variable:

```
# Tcl
```

**Line 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"
```

**Line 2**

```
set call_args [idlgen_process_list $arg_list \
java_l_name $start_str $sep_str]
[***
//--------
// Invoke the operation
//--------
try {
```

**Line 3**
```
   @$ret_assign@obj.@$op_name@(@$call_args@);
} catch(org.omg.CORBA.Exception ex) {
   ... // handle the exception
}
***]
```

The above Tcl code produces the following Java:

```
//--------
// Invoke the operation
//--------
try {
```

**Line 4**
```
   _result = obj.op(
```

**Line 5**
```
   p_widget,
```

**Line 6**
```
   p_string,
```

**Line 7**
```
   p_longSeq,
```

**Line 8**
```
   p_long_array);
} catch(org.omg.CORBA.Exception ex) {
   ... // handle the exception
}
```

Two points are worth noting about the Tcl script that produced the above output:

- The text `_result =` (line 4 of the Java code) is produced by the command `[java_ret_assign $op]` (at lines 1 and 3 in the Tcl script). If the operation invoked does not have a return type then `[java_ret_assign $op]` returns an empty string.
- You can format the parameters to an operation call (lines 5-8 in the Java code) with the command `idlgen_process_list` (used at lines 2 and 3 in the Tcl script). This command is discussed in "Idlgen_process_list" on page 131.

## Processing Output Parameters and the Return Value

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

```
# Tcl
[***
   //--------
```

```
   // Process the returned parameters
   //--------
***]
```

**Line 1**

```
foreach arg [$op args {out inout}] {
set type [$arg type]
set name [java_l_name $arg]
```

**Line 2**

```
set arg_ref [java_clt_par_ref $arg]
[***
   process_@[$type s_uname]@(@$arg_ref@);
***]
}
set ret_type [$op return_type]
if {[$ret_type l_name] != "void"} {
```

**Line 3**

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

The `foreach` loop at line 1 iterates over all the `out` and `inout` parameters. Notice how the `java_clt_par_ref` command can be used to obtain references to both parameters (line 2) and the return value (line 3). The above Tcl code produces the following Java:

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

## Processing Implicit Parameters to Attributes

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

`java_clt_par_decl` *`arg_or_op`*

This command is used to declare a client-side variable to be used as a parameter to (or return value of) an operation. Similar functionality is needed to declare a client-side variable to be used as the implicit parameter to (or return value of) an

attribute. This additional functionality is obtained by implementing the java_clt_par_decl command in a way that allows it to be invoked with either two arguments (as indicated above) or with four arguments, as shown below:

```
java_clt_par_decl name type dir is_var
```

In this case, the argument arg_or_op has been replaced with three arguments that specify the attribute's name, type and direction (dir). The dir argument should be in or return, for an attribute's modifier and accessor, respectively. This convention of replacing arg_or_op with three arguments is also used in the other commands for the client-side processing of parameters. Thus, the full collection of commands for processing the implicit parameter/return value for an attribute is:

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

It also applies to the gen_ counterparts:

```
java_gen_clt_par_decl name type dir ind_lev
```

## Server-Side Processing of Parameters

The std/java_boa_lib.tcl library provides the following commands to process parameters (and the return value) inside the body of an operation:

```
java_srv_ret_decl op
java_srv_par_alloc arg_or_op
java_srv_par_ref arg_or_op
```

Some of the above commands have gen_ counterparts:

```
java_gen_srv_ret_decl op ind_lev ?alloc_mem?
java_gen_srv_par_alloc arg_or_op ind_lev
```

In all of the above commands, the arg_or_op parameter can be either an argument node or an operation node in the parse tree. If arg_or_op is an argument node then the above commands apply to a parameter of an operation call. Conversely, if arg_or_op is an operation node then the above commands apply to the return value of an operation.

The java_srv_ret_decl command returns a Java declaration of a variable that holds the return value of an operation. If the operation does not have a return value then this command returns an empty string.

The java_srv_par_alloc command returns a Java statement to create a Holder class for an out parameter (or return value), if needed. If there is no need to allocate memory then this command returns an empty string.

The java_srv_par_ref command returns a reference to the value of the specified parameter (or return value) of an operation.

The examples in the following subsections illustrate the use of these commands. In each of the examples, the following IDL is assumed (this is the same IDL used previously in "Client-Side Processing of Parameters" on page 219):

```
// 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);
};
```

## Declaring the Return Value and Creating Memory for Parameters

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 ind_lev    1
set arg_list   [$op contents {argument}]
if {[$ret_type l_name] != "void"} {
[***
   //--------
   // Declare a variable to hold the return value.
   //--------
   @[java_srv_ret_decl $op 0]@;
```

**Line 1**

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

**Line 2** `java_gen_srv_par_alloc $op $ind_lev`

The output of the above Tcl is as follows:

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

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

Note that the declaration of the `_result` variable is separated from the creation of memory for it. If you prefer to allocate memory for the `_result` variable in its declaration then the declaration of `_result` is changed to:

```
int[] _result = new longSeq;
```

## Initializing Output Parameters and the Return Value

The next Tcl script iterates over all the `inout` and `out` parameters, and the return value, to assign values to them. Comments follow after the script:

```
# 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]"
   if {[$arg direction] == "inout"} {

   java_gen_assign_stmt $type $arg_ref $name2 \
$ind_lev 0
}
if {[$ret_type l_name] != "void"} {
   set ret_ref [java_srv_par_ref $op]
   set name2   "other_[$ret_type s_uname]"
   java_gen_assign_stmt $ret_type $ret_ref \
                              $name2 $ind_lev 0
}
```

**Line 1**

**Line 2**

The command `java_srv_par_ref` (lines 1 and 2) can be used to obtain a reference to both parameters and the return value.

The output generated by the above Tcl code is as follows:

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

## Processing Implicit Parameters to Attributes

Recall that the java_srv_par_alloc command is defined as follows:

```
java_srv_par_alloc arg_or_op
```

This command is used to allocate memory, if necessary, for an out parameter or return value of an operation. Similar functionality is needed to allocate memory for the return value of the accessor function for an attribute. This additional functionality is obtained by implementing the java_srv_par_alloc command in a way which allows it to be invoked with either one argument (as indicated above) or with three arguments, as shown below:

```
java_srv_par_alloc name type dir
```

In this case, the argument arg_or_op has been replaced with three arguments that specify the attribute's name, type and direction (dir). The dir argument should be return for an attribute's *accessor*. This convention of replacing arg_or_op with several arguments is also used in the other commands for the server-side processing of parameters. Thus, the full collection of commands for processing the implicit parameter/return value for an attribute is:

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

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 dir ind_lev
```

## Processing Instance Variables and Local Variables

Previous sections have discussed how to process variables used for parameters and the return value of an operation call. However, not all variables are used as parameters. For example, a Java class that implements an IDL interface may contain some *instance variables* that are not used as parameters; or the body of an operation may 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 dir`*

The `java_var_decl` command returns a Java variable declaration with the specified `name` and `type`.

There is also a `gen_` counterparts to the above command:

`java_gen_var_decl` *`name type dir ind_lev`*

The following example illustrates the use of these `gen_` commands:

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

      // Initialize variables
***]
foreach type $type_list {
      set name  "my_[$type l_name]"
      set value "other_[$type l_name]"
      java_gen_assign_stmt $type $name $value $ind_lev 0
}
[***
} // some_func()
***]
```

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

```
// Java
void some_method()
{
      // Declare variables
      String my_string
      widget my_widget;
      int[] my_long_array;

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

# Processing Unions

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 commands java_branch_case_s_label and java_branch_case_l_label are provided to help with this task. However, sometimes you may want to process an IDL union using a different Java construct, such as an if-then-else statement. The slightly lower-level commands java_branch_s_label and java_branch_l_label are provided to help with this task.

## java_branch_case_s_label

This command is used as follows:

java_branch_case_s_label *union_branch*

This command returns a string in the form "case label" where label is the (fully-scoped) label for that branch of the union, or the string "default" if it is the default branch in the union. As an example, 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 generates a Java `switch` statement to process the union:

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

The code generated from the above Tcl is as follows:

```
// Java
void some_method()
{
switch(u._discriminator()) {
case m.red().value():
      ... // process u.a()
      break;
case m.green().value():
      ... // process u.b()
      break;
default:
```

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

The command `java_branch_case_s_label` works for any type of the union's discriminant. For example, if the discriminant is, say, type `long` then this command will return a string of the form "`case 42`" (where `42` is the value of the case label), or if the discriminant is type `char` then this command returns a string of the form `"case 'a'"`.

## java_branch_case_l_label

This command is used as follows:

`java_branch_case_s_label` *`union_branch`*

It works almost identically to `java_branch_case_s_label` except that it produces the non-scoped label of the union's case. For example, instead of returning `case m.foo.red().value()`, it returns `case foo.red().value()`.

## java_branch_s_label

This command is used as follows:

`java_branch_s_label` *`union_branch`*

It works almost identically to `java_branch_case_s_label` except that the `case` prefix is not included in the returned value.

Note that the command `java_branch_case_s_label` is slightly easier to use if you are generating a Java switch statement to process a union. The command `java_branch_s_label` could, however, be used if you wanted to generate a Java `if-then-else` statement to process a union.

## java_branch_l_label

This command is used as follows:

`java_branch_l_label` *`union_branch`*

It works almost identically to `java_branch_s_label` except that it produces the non-scoped value of the union's case. For example, instead of returning `m.red`, it returns `red`.

# Processing Arrays

Arrays are usually processed in Java by 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 of type `long_array`. Java code to perform an element-wise copy from `bar` into `foo` might be written as follows:

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

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

In order to write a Tcl script to generate the above Java code, you need Tcl commands that *declare* the index variables (lines marked 1), generate the *header* of the `for` loop (lines marked 2), provide the *index* for each element of the array

([i1][i2] in the above example, as used in line 3), and generate the *footer* of the for loop (lines marked 4). The following commands provide exactly these capabilities:

1. java_array_decl_index_vars *arr pre ind_lev*
2. java_array_for_loop_header *arr pre ind_lev ?decl?*
3. java_array_elem_index *arr pre*
4. java_array_for_loop_footer *arr indent*

In each of these commands, the following conventions hold:

- 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.

As a concrete example, the following Tcl script generates the for loop shown previously:

```
# Tcl
set typedef  [$idlgen(root) lookup "long_array"]
set a        [$typedef true_base_type]
```

**Line 5** `set indent   [java_indent [$a num_dims]]`

**Line 3** `set index    [java_array_elem_index $a "i"]`

```
[***
void some_method()
{
```

**Line 1** `   @[java_array_decl_index_vars $a "i" 1]@`

**Line 2** `   @[java_array_for_loop_header $a "i" 1]@`

**Line 3** `      @$indent@foo@$index@ = bar@$index@;`

**Line 4** `      @[java_array_for_loop_footer $a 1]@`

```
}
***]
```

The amount of indentation to be used *inside* the body of the for-loop (at line 3) is calculated by using the number of dimensions in the array as a parameter to the java_indent command (line 5).

The java_array_for_loop_header command takes a boolean parameter called decl which has a default value of 0. If decl has the value 1, the index variables will be declared inside the header of the for-loop. Thus, 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 does not need to use 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_func()
{
   @[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 Anys

The commands to process type any are split into two categories:

- Those used to insert a value into an any.
- Those used to extract a value from an any.

## Inserting Values into an Any

Use the java_any_insert_stmt command to generate code that inserts a value into an any:

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. An example of its use is as follows:

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

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

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

## Extracting Values from an Any

The following commands are provided to help you write Tcl scripts that extract values from an any:

```
java_any_extract_var_decl type name
java_any_extract_var_ref  type name
java_any_extract_stmt type any_name name
```

The java_any_extract_var_decl command is used to declare a variable into which values from an any will be extracted. The parameters to this command are the variable's type and its name.

The java_any_extract_var_ref command returns a reference to the value in the specified variable (called name and of the specified type).

The java_any_extract_stmt command is used to extract a value of the specified type from the any called any_name into the variable called name.

The following example illustrates the use of these commands:

```
# Tcl
[***
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){
   Systemout.println("Error: extract from any.");
```

```
   e.printStackTrace();
};
***]
```

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

```
// Java
try {

   widget                my_widget;
   org.omg.CORBA.Boolean  my_boolean;
   int[]                 my_long_array;

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

   my_boolean = AnyHelper.extractBoolean(an_any);
   process_boolean(my_boolean);

   my_long_array = long_arrayHelper.extract(an_any);
   process_long_array(my_long_array);
}
catch( exception e){
   Systemout.println("Error: extract from any.");
   e.printStackTrace();
};
```

# 13

# 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++". 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:

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

The minimal API defines the following command:

`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:

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

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

`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                          //Java
module outer{                  idlgen.outer.inner.Printmystruct
   interface inner{
      struct mystruct{
         ...
      }
   }
}
```

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_variable_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:

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

The minimal API defines the following commands:

`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:

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

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

`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                          //Java
module outer{                  idlgen.outer.inner.Randommystruct
   interface inner{
      struct mystruct{
         ...
      }
   }
}
```

## 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_variable_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:

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

The minimal API defines the following commands:

```
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 (in)equality.

## Example of Use

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

```
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:

```
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.

# Appendices

# Appendix A
# User's Reference

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

## General Configuration Options

Table 13.1 describes the general purpose configuration options available in standard configuration file `idlgen.cfg`.

| Configuration Option | Description |
|---|---|
| `idlgen.install_root` | Set this to the installation directory IDLgen is installed in. |
| `idlgen.genie_search_path` | Search order used by the `smart_source` command. |
| `idlgen.tmp_dir` | Directory that IDLgen should use when creating temporary files. |
| `default.all.want_diagnostics` | Setting for diagnostics:<br>1: Genies print diagnostic messages.<br>0: Genies stay silent. |
| `default.orbix.install_root` | Set this to be the installation directory where Orbix has been installed. |

**Table: 13.1:** *Configuration File Options*

| Configuration Option | Description |
|---|---|
| `default.orbix.version_number` | Set this to your version of Orbix. Supported values are 2.2, 2.3, and 3.0. |
| `default.orbix.is_multi_threaded` | Setting:<br>1: if you have multi-threaded Orbix.<br>0: if you have single-threaded Orbix.<br>Note: Orbix is multi-threaded on most platforms. |
| `default.html.file_ext` | File extension preferred by your web browser (".html" for most platforms). |

**Table: 13.1:** *Configuration File Options*

## Configuration Options for C++ Genies

Table 13.2 describes the configuration options specific to C++ genies in the standard configuration file `idlgen.cfg`:

| Configuration Option | Purpose |
|---|---|
| `idlgen.preprocessor` | 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. |

**Table: 13.2:** *Configuration File Options for C++ Genies*

| Configuration Option | Purpose |
|---|---|
| `default.cpp_genie.want_boa` | Sets the approach used when writing C++ classes that implement IDL interfaces:<br>1: Use the BOA approach.<br>0: Use the TIE approach. |
| `default.cpp_genie.want_this` | Do you want the generated C++ class to have a `_this()` function? |
| `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. This is 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.want_throw` | This allows you to set `throw` clauses on the C++ signatures of IDL operations and attributes. Setting:<br>1: Your C++ compiler supports exceptions.<br>0: Your C++ compiler does not support exceptions. |

**Table: 13.2:** *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:<br>1: Named.<br>0: Anonymous. |

**Table: 13.2:** *Configuration File Options for C++ Genies*

## Configuration Options for Java Genies

Table 13.2 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:\jdk1.1". |
| `default.java.java_file_ext` | File extension preferred by your Java compiler. |
| `default.java.java_class_ext` | Class name extension preferred by your Java compiler. |
| `default.java.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.impl_class_suffix` | Suffix for your Java classes that implement IDL interfaces. |

**Table: 13.3:** *Configuration File Options for Java Genies*

| Configuration Option | Purpose |
|---|---|
| default.java.smart_proxy_prefix | Prefix for your Java classes that implement smart proxies for IDL interfaces. |
| 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_throw_sys_except | This allows you to set throw clauses on the Java signatures of IDL operations and attributes. Setting:<br>1: Your Java compiler supports exceptions.<br>0: 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. |
| 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: 13.3:** *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. |
| `-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. |
| `-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. |
| -merge | Generate "marker merge" comments (not default). This allows any amendments to this IDL interface to be reflected in the code by the use of the Orbix Code Amend script. |
| -(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. |

## 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. |

## 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. |

## 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. |

# 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 your 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, IDLgen executes the statement:

```
idlgen_set_preferences $idlgen(cfg)
```

As such, `default` scoped entries in the IDLgen 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 your 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 13.1:

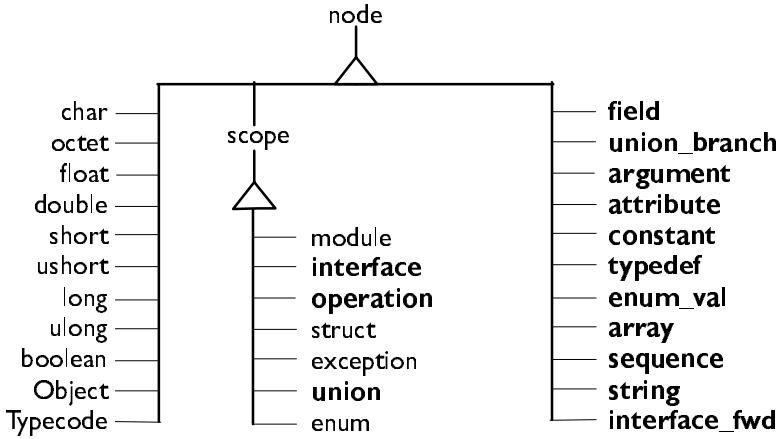


**Figure 13.1:** *Inheritance Hierarchy for Node Types*

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 92. |
| 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

# Index

## D



## E



## F



## G



## H



## I

## J



## L

## T



## U



## V



## W