Babel
Users' Guide

Tammy Dahlgren
Thomas Epperly
Scott Kohn
Gary Kumfert

Disclaimer

This work was performed under the auspices of the U.S. Department of Energy by University of California Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48.

Release Information

Babel Users' Guide (this document)	UCRL-MA-145991
Babel Source Code (associated software)	UCRL-CODE-2002-054

Babel
Users' Guide

TAMMY DAHLGREN
THOMAS EPPERLY
SCOTT KOHN
GARY KUMFERT

Center For Applied Scientific Computing
Lawrence Livermore National Laboratory
P.O. Box 808
Livermore, California, USA

January 14, 2003

Preface

Babel in a Nutshell

Babel is a tool that enables software written in different languages to communicate. It accomplishes this task by using an Interface Definition Language (IDL) similar to COM and CORBA, but specifically tuned for scientific applications. By expressing software interfaces (sometimes called APIs) in SIDL, Babel can generate the appropriate glue code to make software language interoperable. Features unique to SIDL are:

- Dynamic multi-dimensional arrays
- Complex numbers (e.g. $2 + 3i$)
- In-process optimizations
- Special directives for large-scale parallel distributed programming (future)
- Syntax for specifying interface behavior (future)

Babel enables true object-oriented techniques even in non object-oriented languages. The object model that SIDL supports is similar to Java and Objective C where a class can extend at most one class, but implement many interfaces. In C++ speak, an interface is simply a class of all pure-virtual methods. Furthermore, if library developers want object-oriented features but are required to be 100% ANSI C compliant, Babel can meet those constraints. Although the Babel code generator is implemented in Java, the runtime libraries and generated files for C bindings are 100% ANSI C compliant.

Babel is the basis for a component framework, but it is not a complete framework by itself. We've added a tiny CCA-0.5 compliant framework in our examples/ directory, called *Decaf*. Decaf demonstrates how Babel can be used to implement a component framework, but Decaf itself has just enough to build a "Hello World" component and connect it to a "Printf" component.

SIDL is also a useful tool in itself for code development teams. SIDL can only express the public API and nothing about its implementation. Discussions between groups who restrict themselves to SIDL can safely avoid implementation issues that would detract from the interface design. Furthermore, SIDL is simple, clean, and powerful enough for Computer Scientists, Math Programmers, and Application Scientists to debate APIs proficiently using only email.

Scope of this Manual

This document is a tutorial on how to use the Babel tools to generate and use component software. The Babel tools were designed specifically for scientific applications, therefore most of the examples and exercises here also deal with scientific applications.

This manual assumes the reader is a programmer who is proficient in two or more of the following languages: FORTRAN 77, FORTRAN 90, C, C++, Java, or Python. Furthermore, this manual assumes the reader is familiar with the SPMD[1] programming model that pervades the scientific computing community. Knowledge of and experience with MPI programming is helpful, but not strictly required.

[1] Single Program Multiple Data

Babel source is available free of charge on the web. Developed by the Components Project at the Lawrence Livermore National Laboratory Center for Applied Scientific Computing (CASC), it is licensed under the Lesser GNU Public License (LGPL). See the source distribution for details.

The Babel distribution is published on Alexandria along with software components available for use with Babel. Alexandria is a software component repository that is also built by the Components Project at CASC. You can access Alexandria on the web from the following URL:

`http://www-casc.llnl.gov`

Readers may also be interested in viewing the Components Project home page at

`http://www.llnl.gov/CASC/components`

Conventions

The following typographic conventions are used throughout this manual.

Italic	is used for file and command names. It is also used to highlight comments in examples and to define terms the first time they appear in a document.
`Constant Width`	is used in examples to show the text that is generated, and in regular text to show operators, variables, and the output from commands or programs.
`Constant Slanted`	is used for displaying for SIDL source code. We use a separate font to distinguish SIDL code from generated code.
`Constant Bold`	is used to show user's modifications to generated code and in examples to show user's actual input at a terminal.
Sans Serif Slanted	is used in examples to show variables for which a context-specific substitution should be made. The variable *filename*, for example, would be replaced by the actual filename.

Additionally, we use specific blocks of text as sidebars to call the readers attention to particular information. Here's one kind.

> **Rationale:** *Often when listing restrictions or requirements, we find it helpful to also explain and document the rationale behind a design decision. In time, the context in which the rationale was based may become irrelevant, making the rationale blocks very useful for understanding when to change a decision.*

Here's another kind of text-block you'll see.

Occasionally, we'll also put notes to ourselves about things to fix or add. As the document matures, you'll hopefully see less of these. We could prune them from the distributed document entirely, but opted to keep them, so you know what we're working on.

We Appreciate Your Feedback

We have tested and verified the information in this manual. Nonetheless, features may have changed or oversights may exist. Please contact us with any issues, corrections, or suggestions for future versions of this manual through snail mail at:

Components Project
Center for Applied Scientific Computing
Lawrence Livermore National Laboratory
P.O. Box 808, L–661
Livermore, CA 94551

or through email to:

`components@llnl.gov`

To find out more about Babel, feel free to subscribe to one or more of the associated distribution lists given below.

- `babel-users@llnl.gov` is an open discussion forum about Babel. Anyone can subscribe or send email to this list.
- `babel-announce@llnl.gov` is a moderated email forum to which anyone can subscribe (though no-one can post). This is a low-volume alternative for people who want to know about releases and major announcements.

To subscribe, simply send email to `majordomo@lists.llnl.gov` with the appropriate line(s):

`subscribe babel-users` *[email-address]*
`subscribe babel-announce` *[email-address]*

where you can explicitly state your email address in *email-address* or, if you leave *email-address* blank, majordomo will use your email ReplyTo: field.

Acknowledgments

Project Alumni

- Melvina Blackgoat
- Nathan Dykman
- Brent Smolinski

Alpha Testers

- Andy Cleary
- Jeff Painter
- Cal Ribbens

Shared Software

Babel depends on its share of third-party software. Some is redistributed with the code, some not. They all deserve mention (and some require it).

- **JavaCC** is used to generate the SIDL Parser. This is owned by Sun Microsystems with WebGain as the "caretaker" through a partnership with Sun. JavaCC can be downloaded and used in most situations at no cost (`http:///www.webgain.com/products/java_cc`). There are liscencing restrictions on redistribution, modifying, etc.

- **gnu.getopt** is an implementation of GNU Getopt in Java and is distributed with Babel as a JAR file. It can be downloaded (along with sourcecode) from either the GNU website `http://www.gnu.org/software/java/packages.` or the author's website `http://www.urbanophile.com/arenn/hacking/download.html` The following is the copyright notice for gnu.getopt:

```
/**************************************************************************
/* Getopt.java -- Java port of GNU getopt from glibc 2.0.6
/*
/* Copyright (c) 1987-1997 Free Software Foundation, Inc.
/* Java Port Copyright (c) 1998 by Aaron M. Renn (arenn@urbanophile.com)
/*
/* This program is free software; you can redistribute it and/or modify
/* it under the terms of the GNU Library General Public License as published
/* by  the Free Software Foundation; either version 2 of the License or
/* (at your option) any later version.
/*
/* This program is distributed in the hope that it will be useful, but
/* WITHOUT ANY WARRANTY; without even the implied warranty of
/* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
/* GNU Library General Public License for more details.
/*
/* You should have received a copy of the GNU Library General Public License
/* along with this program; see the file COPYING.LIB.  If not, write to
/* the Free Software Foundation Inc., 59 Temple Place - Suite 330,
/* Boston, MA  02111-1307 USA
/**************************************************************************/
```

The text for the GNU Library GPL is available at `http://www.gnu.org/copyleft/library.html`.

Contents

Chapter 1

Introduction

__NOTE:__ This document applies to Babel 0.8.0.
It, like the software it documents, is still a work in progress.

– The Babel Development Team

Contents

1.1 Babel Addresses a Problem

Babel was conceived, designed, and built to solve a problem; namely, how scientific software libraries can be made equally accessible from all of the standard languages. The goal is language interoperability. The vision goes far beyond calling BLAS[1] implemented in FORTRAN 77 from a C program. At its heart, Babel lets programmers use their tool of choice in developing complete applications using components implemented in one or more distinct programming languages.

For instance, let us say that an application scientist is running a sophisticated C++ code from a Python scripting environment. This can already be easily accomplished with technologies like SWIG. Now let's say that the simulation is showing some erratic behavior and the application scientist wants to extend the `ConvergenceCheck` class to also report some information to a log file. Let's also assume that this application scientist doesn't want to write a new C++ class much less rewrite the current application. What this individual wants to do is derive and utilize a new class in Python from the C++ `ConvergenceCheck` class. Thus, the C++ simulation code will now have to invoke a method on a class implemented in Python, which then dispatches back to the C++ base class after doing its additional logging. This is an example of a capability that Babel provides that is outside the scope of SWIG.

Figure 1.1 lists many of the primary languages that are of interest to scientific simulation software developers and users. The good news is that there is a path from each language to every other; meaning that calling from one to

[1] BLAS: Basic Linear Algebra Subroutines

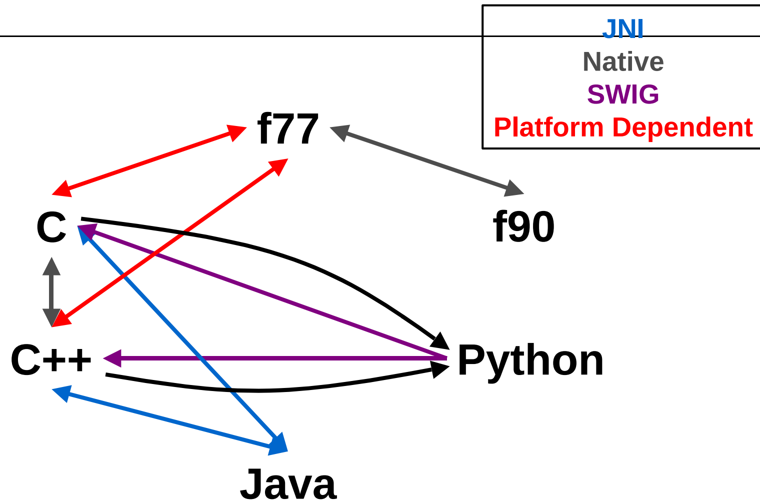

Figure 1.1: Language Interoperability Using Current Technology.

another is possible. However, the technologies to get from one language to another vary widely and are fraught with pitfalls.

Babel works by providing the technology to define and support the multi-language interoperation of a common subset of functionality. See Fig. 1.2 to see a graphical representation of the supported languages. It is important to note that this common subset is *far* from a lowest common denominator solution. In fact, to support this common set Babel actually adds functionality when it is lacking in the host language.

The common subset of functionality is expressed in the Scientific Interface Definition Language (SIDL) (pronounced “SIGH-dull”). SIDL is similar to COM and CORBA IDLs, but was designed with an emphasis on scientific computing. Specifically, SIDL supports dynamic multi-dimensional arrays, has a built-in complex numbers, and will acquire a set of directives to aid in the description of massively parallel distributed objects.

1.2 SIDL: The Interoperable Feature Set

When it comes to deciding what programming idioms to support across all languages and which ones to reject, SIDL strikes a careful balance between minimalism and completeness. It is not a lowest common denominator solution. SIDL is minimal to keep the learning curve as low as possible. It is complete so developers do not feel constrained in how to express their solutions.

SIDL is object-oriented. Its object model closely resembles that of Java and Objective C. In this model there is single inheritance of implementation and multiple inheritance of interfaces. It supports the typical notions of virtual, static, and final methods. SIDL also provides a basic set of features by defining and implementing the basic types for interfaces, classes and exception. All types implicitly inherit from these basic types.

SIDL has a complete set of fundamental data types, from booleans to double precision complex numbers. It also supports more sophisticated types such as enumerations, strings, objects, and dynamic multi-dimensional arrays[2][3].

SIDL is still a work in progress. Of particular research interest are directives that will be added for parallel distributed object interaction and features to specify behavioral semantics associated with the interfaces.

[2] Arrays of enums are not yet supported.
[3] Some language bindings may not be mature enough to fully support all types.

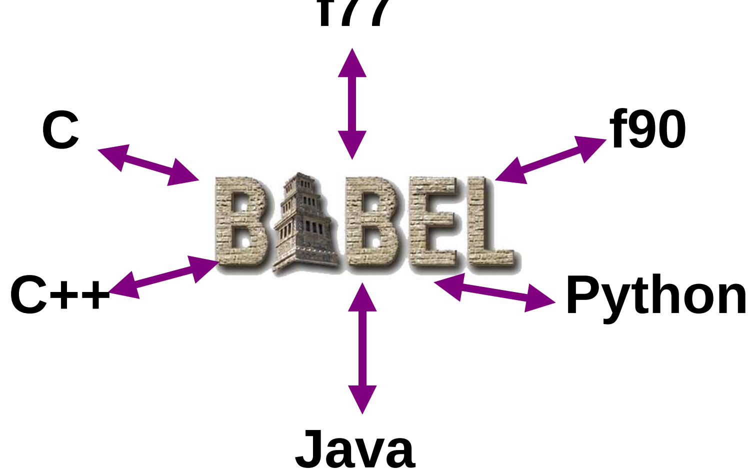

Figure 1.2: Language Interoperability Using Babel.

1.3 How Babel is Used

Babel has two types of customers: *developer* and *user*. The developer implements a library that will be used by one or more users. Since one goal of the developer is to increase their customer base, the developer writes a SIDL file that effectively publishes the interface to their software in a platform and language neutral manner. The user, on the other hand, may not care or even know that they are interacting with a library through Babel.

Babel provides some features that benefits user and developer alike. The most important aspect to note here is that all Babel objects are reference counted. This feature is critical to encapsulate the memory allocation library (e.g. C's malloc/free or C++'s new/delete) used in the implementation of the object. Users never need concern themselves with when to free up a resource, they only declare when they're done with their reference to that resource. Developers are free to use different memory allocation subsystems in different parts of their code if need be.

1.4 Deployment of Babel Enabled Libraries

At this point, there is no standard — or even recommended — model for deploying Babel enabled libraries. Below are a few examples of how our developer-customers are currently packaging their code.

One mode expects users to have Babel installed on their system. In this mode, developers simply include a SIDL file and their corresponding implementation files. The user in this case must build the software, call Babel to generate the client bindings in the language of choice, and link it all together into a final application.

Another mode tries to hide Babel as much as possible. In this mode, the developer pre-generates many different client language bindings and distributes them along with their code and the sources for the Babel runtime library. Then the user has a "batteries included" package that's ready to run out of the box. The user may not even be aware that Babel has been used unless they pay careful attention to how the package was built.

A third mode distributes only the SIDL file and the precompiled shared library files. This is not an open-source solution, though users still need to build the language bindings to access the shared library.

1.5 Beyond Babel's Scope

The language interoperability problem is a large one, and though the Babel tool addresses much of it, there is still a lot that is beyond the scope of our tool. Babel is at its heart a code generator and a runtime library.

Reverse engineering is not supported. That is, there is no support for inspecting or modifying compiled code. In addition, scanning existing software to generate SIDL wrappers is not supported. There are other groups who are pursuing a C++ to SIDL converter. Since SIDL contains different information than what is in a C++ header file, however, such a converter cannot be fully automated without additional help.

Library compatibility is limited. Since Python and Java dynamically load libraries into their virtual machines, using these languages requires the ability to build shared libraries. In general, building shared libraries (particularly from C++) is difficult and error prone. This is compounded by the fact that compiler vendors have no standard way of doing this, and many tools that help building shared libraries don't support C++. One can build a legitimate shared library that still won't work because there are unresolved symbols, or the library was loaded in the wrong mode.

Compiler compatibility is limited. Since the C++ standard does not specify a binary interface and uses a lot of hashing in their symbol tables, there have been no attempts to get libraries from dissimilar C++ compilers to work together. Similarly, although we support FORTRAN 77 and FORTRAN 90, all libraries of Fortran code must be compiled with the same compiler. . . again because of the lack of a standard binary interface.

Despite the aforementioned limitations, Babel does facilitate the development of language interoperable software. However, issues of robust packaging, building, and deployment of language interoperable software still loom on the horizon.

Chapter 2

Hello World Tutorial

This tutorial will guide you through the process of writing the classic "Hello World!" example using the Babel tools. In the process, you will learn how to write a SIDL (Scientifi c Interface Defi nition Language) interface description fi le, generate the library implementation in C++, and write a C main program to call the library. You will also learn how to write a Makefi le to compile and link the library and program.

Contents

2.1 Writing the SIDL File

We will write the "Hello World!" program in a directory called hello/ and place the client library in a subdirectory hello/lib/:

```
% mkdir hello
% cd hello
% mkdir lib
```

The first step is to write a SIDL file. Recall that SIDL is an interface definition language (IDL) that describes the calling interface for a scientific library. It is used by the Babel tools to generate glue code that hooks together different programming languages. A complete description of SIDL can be found in Chapter 5.

For this particular application, we will write a SIDL file that contains a class World in a package Hello. Method getMsg() in class World returns a string containing the traditional computer greeting. Using your favorite text editor, create a file called hello.sidl in the hello/ directory containing the following:

```
package Hello version 1.0 {
  class World {
    string getMsg();
  }
}
```

The package statement provides a scope (or namespace) for class World, which contains only one method, getMsg(). The version clause of the statement identifies this as version 1.0 of the Hello package.

2.2 Writing the Implementation

We will write the implementation in the lib/ subdirectory of hello/. The first step is to run the Babel shell script to generate the library implementation code for the SIDL file. We will implement the library in C++. The command to generate the Babel library code (assuming Babel is in your PATH) is:

```
% babel -sC++ -olib ../hello.sidl
```

In this Babel command, the "`-sC++`" flag, or its long form "`--server=C++`", indicates that we wish to generate C++ bindings for an implementation[1]. The "`-olib`" flag, or its long form "`--output-dir=lib`", defines the root directory of where the generated code should be placed.

This command will generate a large number of C and C++ header and source files. It is often surprising to newcomers just how much code is generated by Babel. Rest assured, each file has a purpose and there is a lot of important things being done as efficiently as possible under the hood.

Files are named after the fully-qualified class-name. For instance, a package *`Hello`* and class *`World`* would have a fully qualified name (in SIDL) as *`Hello.World`*. This corresponds to file names beginning with `Hello_World`[2]. For each class, there will be files with `_IOR`, `_skel` or `_impl` appended after the fully qualified name. *IOR files* are always in ANSI C (source and headers), containing Babel's Internal Object Representation. *Impl files* contain the actual implementation, and can be in any language that Babel supports, in this case, they're C++ files. Impl files are the only files that a developer need look at or touch after generating code from the SIDL source. *Skel files* perform translations between the IORs and the Impls. In some cases (like Fortran) the Skels are split into a few files: some in C, some in the Impl language. In the case of C++, the Skels are pure C++ code wrapped in `extern "C"` {} declarations. If the file is neither an IOR, Skel, nor Impl, then it is likely a *Stub*. Stubs are the proxy classes of Babel, performing translations between the caller language and the IOR. Finally, the file babel.make is a Makefile fragment that will simplify writing the Makefile necessary to compile the library. You may ignore the babel.make file if you wish.

The only files that should be modified by the developer (that's you since you're implementing Hello World) are the "Impls", which are in this case files ending with `_Impl.hh` or `_Impl.cc` Babel generates these implementation files as a starting point for developers. These files will contain the implementation of the Hello library. Every implementation file contains many pairs of comment "splicer" lines such as the following:

```
std::string
Hello::World_impl::getMsg()
throw ()
{
    // DO-NOT-DELETE splicer.begin(Hello.World.getMsg)
    // Insert code here...
    // DO-NOT-DELETE splicer.end(Hello.World.getMsg)
}
```

Any modifications between these splicer lines will be saved after subsequent invocations of the Babel tool. Any changes outside the splicer lines will be lost. This splicer feature was developed to make it easy to do incremental development using Babel. By keeping your edits within the splicer blocks, you can add new methods to the hello.sidl file and rerun Babel without the loss of your previous method implementations.

For our hello application, the implementation is trivial. Add the following return statement between the splicer lines in the `lib/Hello_World_Impl.cc` file:

```
std::string
Hello::World_impl::getMsg()
throw ()
{
```

[1] You can also try the "`--help`" flag to list all of the Babel command-line options.
[2] Note: dots are converted to underscores for file naming.

```
        // DO-NOT-DELETE splicer.begin(Hello.World.getMsg)
        return std::string("Hello World!");
        // DO-NOT-DELETE splicer.end(Hello.World.getMsg)
    }
```

To keep the Makefile simple, we will use some GNU Make features. This Makefile may not work with other make implementations. The GNU gcc and g++ compilers are used in this example. The following Makefile in the lib/ subdirectory will compile the library files and create a shared library named `libhello.so`:

```
.cc.o:
    g++ -fPIC -I$(HOME)/babel/include -c $<
.c.o:
    gcc -fPIC -I$(HOME)/babel/include -c $<

include babel.make
OBJS = ${IMPLSRCS:.cc=.o} ${IORSRCS:.c=.o} \
       ${SKELSRCS:.cc=.o} ${STUBSRCS:.cc=.o}

libhello.so: ${OBJS}
    g++ -shared -o $@ ${OBJS}

clean:
    ${RM} *.o libhello.so
```

You do not necessarily need to create a shared library for this example; you may generate a standard static library (e.g., libhello.a). However, in general, you must generate a shared library if you will be calling your library from Python or Java. To create the shared library archive libhello.so, simply execute make as follows:

```
% cd lib/
% make libhello.so
```

2.3 Writing the Client

We will write the client in the main hello/ subdirectory. The main program will be written in C. File hello.c is as follows:

```
#include <stdio.h>
#include "Hello_World.h"

int main(int argc, char** argv)
{
  Hello_World h = Hello_World__create();
  char* msg = Hello_World_getMsg(h);
  printf("%s\n", msg);
  Hello_World_deleteRef(h);
  free(msg);
}
```

This code creates the Hello_World object, calls the getMsg() method, prints the ubiquitous saying, decrements the reference count for the object, and frees the message string.

There are a few details worth noting here. The C bindings generate function names by combining packages, classes, and method names with underscores (e.g. `Hello_World_getMsg()`. Whenever you see double underscores in Babel generated symbols, they indicate something built-in to (and sometimes specific to) the language binding. The _create() method is built-in to every instantiable class defined in SIDL, triggering the creation of Babel internal data structures as well as the constructor of the actual object implementation.

To generate the C glue code necessary to call the library, we run the Babel tool again, this time specifying C as the target language:

```
% babel -cC hello.sidl
```

The "-cC" flag, or its equivalent long-form "--client=C", tells the Babel code generator to create only the C stub calling code, not the entire library implementation. The library libhello.so already contains the necessary IOR, skeleton, and implementation object files. We compile the hello program using the following GNU Make Makefile:

```
.c.o:
    gcc -I$(HOME)/babel/include -Ilib -c $<

include babel.make
OBJS = hello.o ${STUBSRCS:.c=.o}

hello: ${OBJS}
    gcc ${OBJS} -o $@ \
      -Rlib -Llib -lhello \
      -R$(HOME)/babel/lib -L$(HOME)/babel/lib -lsidl

clean:
    ${RM} *.o hello
```

Note that the "-R" flags tell the dynamic library loader where to find the hello and sidl shared libraries. You could achieve the same behavior through environment variables such as LD_LIBRARY_PATH. On some machines and compilers (notably linux-gcc-3.0) the -R flag is no longer supported, so you will have to modify the appropriate environment variable to find the shared library.

Finally, we make the executable and run it:

```
% make hello
% ./hello
Hello World
```

2.4 Final Remarks

Congratulations! You are now ready to develop a parallel scalable linear solver package.

The preceding process may seem to be the most complicated way to write the world's simplest program but, of course, the same process will also work for significantly more complex applications. "Hello World" is small enough to experiment with in the language of your choice. Parallel, multithreaded, scientific simulation codes are another matter entirely.

Chapter 3

Basics: Interfaces, Classes and Arrays

In this section, you will learn more about SIDL, and how Babel works to generate language independent software packages. At the end of this chapter, most developers will have enough information to begin to reformulate their software in SIDL.

Contents

3.1 SIDL Files

SIDL files are language and platform independent interface specifications for objects and their methods. Babel reads these SIDL files to generate appropriate code to support language interoperable software.

3.1.1 Comments and Doc-Comments

SIDL has the same commenting style as C++/Java and even has a special documentation comment (so called *doc-comment*) similar to those used in Javadoc. One can embed comments anywhere in their SIDL file. Documentation comments should immediately preceed the class, interface, or method with which they are associated. Babel replicates documentation comments in the files it generates.

```
/*
 *  1. This is a multi-line comment.
 *
 */

//  2. This comment fits entirely on a single line.

/*  3. This comment can fill less than a line. */

/** 4. This is a documentation comment. */

/**
 *  5. Documentation comments can span
 *     multiple lines without the beginning
 *     space-asterisk-space combinations
 *     getting in the way.
 */
```

Consider the above SIDL file fragment.

1. This comment is a regular multi-line comment that is delimited by a slash-star , star-slash (“`/*`”, “`*/`”) pair.

2. This is a single-line comment that starts with a double slash “`//`” and continues to the end of the line.

3. This comment is the same as # 1 except that it is completely contained on a single line. It can be embedded in the middle of a line anywhere a space naturally occurs.

4. This is a documentation comment. In keeping with Javadoc, Doc++, and other tools, it is delimited by slash-star-star and star-slash (“`/**`”, “`*/`”) combinations. Documentation comments are important because their contents are preserved by Babel in the corresponding generated files. Doc-comments must directly precede the interface, class, or method that they document.

5. This is a multi-line variant of a doc-comment. Note that initial asterisks on a line are assumed to be for human readers only and are discarded by Babel when it reads in the text. The multi-line doc-comment is the preferred way of documenting SIDL.

3.1.2 Packages, Versions, and Import

This section needs to be brought up-to-date now that we have revised versioning (which includes `import` and `require` statements) and introduced re-entrant package support.

WARNING:

This section explains issues that are commonly misunderstood and misused. Please read this whole section carefully.

SIDL has both a packaging and versioning mechanism built in. Packages are essentially named scopes, serving a similar function as Java packages or C++ namespaces. Versions are decimal separated integer values where it is assumed larger numbers imply more recent versions.

The outtermost SIDL package has a version number assigned to it. By default that version is 0. All classes and interfaces in that package get that same version number. If subpackages are specified, they can have their own version number assigned.

```
package mypkg {

}
```

This SIDL file represents the minimum needed for each and every SIDL file. The package statement defines a scope where all classes within the package must reside. Since no version clause is included, the version number defaults to 0.

Packages can be nested. This is shown in the example below. The version numbers assigned to all the types is determined by the package, or subpackage, in which it resides. In the design of the SIDL file, remember that some languages get very long function names from excessively nested packages or excessively long package names.

```
package mypkg version 1.0 {

  package thisIsAReallyLongPackageName {
  }

  package this version 0.6 {
    package is {
      package a {
        package really {
          package deeply version 0.4 {
            package nested {
              package packageName version 0.1 {
              }
            }
          }
        }
      }
    }
  }
}
```

Advanced:

There is a bug/feature in Babel which allows sub-packages to be broken into separate files, but you'd still have to run Babel on all the files at the same time. Here's how it works.

First define the outtermost package in a file.

```
package mypkg version 2.0 {

}
```

Then define a sub-package in a second file.

```
package mypkg.subpkg version 2.0 {

}
```

Note that both files begin with the identical version statement. Now as long as you run Babel on both SIDL files at the same time (with the outtermost one first on the commandline), all is fine.

This works because the package statement takes a scoped identifier as an argument. As long as Babel knows that a package *mypkg* exists, it can handle a new package called *subpkg*. Version statements require an identifier for the outtermost package. Since packages cannot have dots "." in their names, the only dots in version statements should appear at the numbers, not the package names.

Running the second file without the first will (and should) generate an error since the enclosing package was not declared. Use of this bug/feature should be used judiciously.

External types can be expressed in one of two ways. The fully scoped external type can be used anywhere in the class description. Alternatively, an *import* statement can be used to put the type in the local package-space. Below is a sample SIDL file, that should help bring all of these concepts together.

```
require pkgA version 1.0; // restrict pkgA to 1.0

import pkgB;      // import pkgB.B to my space

require pkgC version 2.0; // restrict pkgC to version 2.0

package mypkg version 2.0 {
  class foo {
    setA( A ); // imported from pkgA, must be pkgA.A-v1.0
    setB( B ); // imported from pkgB, must be pkgB.B, no version restriction
    setC( pkgC.C ); // must be pkgC.C-v2.0
    setD( pkgD.D ); // no version restriction
  }
}
```

3.1.3 XML Repositories

Even though SIDL is the primary input format for Babel, it is not the only format Babel understands. For type repositories (similar in function to include directories for C/C++ headers) the preferred language to articulate types is XML.

Babel has the capabilities to convert SIDL files into XML files adhering to the SIDL.DTD. The XML files in these repositories can be included in subsequent runs quickly since all the external references were resolved by Babel during their creation. A SIDL file may refer to unresolved types.

Similarly, Babel can convert XML files that adhere to the SIDL.DTD to SIDL files.

3.2 Babel Command Line

The entire Babel code generator is written in Java and compiled into a jar file. For convenience, a small script called **`babel`** is provided that *should* set the appropriate environment variables and invoke the Java Virtual Machine on the jar file. To test that the script and jar file are working together properly, simply type **`babel --help`**.

3.2.1 Command Line Options

Babel requires exactly one of the following mutually exclusive arguments on the command line.

- **`--help`** : Print options to stdout.
- **`--version`** : Print version of Babel.
- **`--xml`** : Generate XML equivalent of SIDL file. Deprecated; see text option.
- **`--text=`*form*** : Generate text equivalent (”sidl” or ”xml”) of associated package(s).
- **`--client=`*lang*** : Generate client, or proxy, classes to access library.
- **`--server=`*lang*** : Generate the server and client classes to implement the library.
- **`--parse-check`** : Check the SIDL file only.
- **`--generate-sidl-stdlib`** : Regenerate the Babel runtime library.

By far, the three most common uses of Babel will be to generate the Client-side proxies, Server-side implementations, and XML associated with the SIDL file. The last option is essentially used internally when the Babel runtime library is being developed.

Additionally, there are a few supplemental arguments that complete the picture.

- **`--output-directory=`*dir*** : Specifies the root directory associated with the generated files. The default setting is the current working directory.
- **`--generate-subdirs`** : Generates files in a directory tree matching the packaging scope of the SIDL file. This is on by default for languages that have this requirement, such as Java and Python, but off by default for languages that have no such requirement. Hence, code generation for only the latter languages (e.g. C, C++, F77, F90) is effected by this option.
- **`--repository-path=`*path*** : Specifies a semicolon separated list of directories, or URLs[1] to search for XML Type descriptions. The need for these XML types is to resolve references in the SIDL file. This option can be used multiple times on the same command line. If appropriate, the Babel script adds the default repository path to the command line before dispatching to the Java Virtual Machine.
- **`--no-default-repository`** : Prohibits the use of the default repository in resolving symbols.
- **`--suppress-timestamp`** : Suppresses the insertion of meta-information that could result in generated files that would otherwise not differ from prior executions on the same, unchanged input file. Typically Babel inserts meta-information such as creation time into files it generates. Although this information is useful, it does result in the creation of excessive changes when using version control systems.
- **`--exclude=`*regex*** : This options can be used multiple times. Each time you add a regular expression that will be used to exclude symbols from code generation. No code or XML will be generated for any symbol matching the user provided regular expression. This command line option requires version 1.4.0 or later of the Java runtime environment.

[1] URLs have colons in them, so this path has to be semi-colon separated, even though UNIX paths are traditionally colon separated.

Table 3.1: Command Line Arguments.

Short Form	Long Form	Notes
-h	**--help**	
-v	**--version**	
-t*form*	**--text=*form***	
-x	**--xml**	
-c*lang*	**--client=*lang***	
-s*lang*	**--server=*lang***	
-p	**--parse-check**	
	--generate-sidl-stdlib	
-o*dir*	**--output-directory=*dir***	
-g	**--generate-subdirs**	
-R*path*	**--output-directory=*path***	
-e*regex*	**--exclude=*regex***	
	--no-default-repository	
	--suppress-timestamp	

3.2.2 Long and Short Forms

So far, we've shown only the long forms of command line arguments, starting with two hyphens "--". There are also short forms for many of the more frequently used commands. See Table 3.1 for details.

3.2.3 Examples

To create a new XML version of a SIDL file, you are free to use the following deprecated command:

```
% babel -x -omydepot mystuff.sidl
```

However, it would be better if you got used to using the new version:

```
% babel -tXML -omydepot mystuff.sidl
```

To exclude code generation for types whose name begins with "MPI.", use the following command:

```
% babel -sC++ --exclude='^MPI\.'  mystuff.sidl
```

Now suppose a developer wants to implement a library in C++ that corresponds to these types in the SIDL file.

```
% babel -sC++ mystuff.sidl
```

Alternatively, the developer could also create C++ implementation files based on the XML repository. In this case, a list of symbols to be implemented would need to be specified. Assuming that all of the types are in a package called "mystuff", the following command can be issued:

```
% babel -sC++ -Rmydepot mystuff
```

Now suppose a second developer wants to extend this software. A second SIDL file is created then the implementation files in FORTRAN 90 are generated with the following command:

```
% babel -sf90 -Rmydepot newstuff.sidl
```

Table 3.2: ALL SIDL Types And Their Bindings.

SIDL type	size (bits)	C Binding	C++ Binding	F77 Binding	F90 Binding
`bool`	1	int	bool	LOGICAL	LOGICAL
`char`	8	char	char	CHARACTER*1	CHARACTER (LEN=1)
`int`	32	int32_t	int32_t	INTEGER*4	INTEGER (SELECTED_INT_KIND(9))
`long`	64	int64_t	int64_t	INTEGER*8	INTEGER (SELECTED_INT_KIND(18)
`float`	32	fbat	fbat	REAL	REAL (SELECTED_REAL_KIND(8,38))
`double`	64	double	double	DOUBLE PRECISION	REAL (SELECTED_REAL_KIND(17,30
`fcomplex`	64	struct	std::complex<fbat>	COMPLEX	COMPLEX (SELECTED_REAL_KIND(
`dcomplex`	128	struct	std::complex<double>	DOUBLE COMPLEX	COMPLEX (SELECTED_REAL_KIND(
`opaque`	64	void*	void*	INTEGER*8	INTEGER (SELECTED_INT_KIND(18)
`string`	varies	char*	std::string	CHARACTER*(*)	CHARACTER*(*)
`enum`	32	enum	enum	INTEGER	INTEGER (SELECTED_INT_KIND(9))
`interface`	varies	struct	class	INTEGER*8	INTEGER (SELECTED_INT_KIND(18)
`class`	varies	struct	class	INTEGER*8	INTEGER (SELECTED_INT_KIND(18)
`array<Type,Dim>`	varies	struct	template class	INTEGER*8	INTEGER (SELECTED_INT_KIND(18)

Finally, a user can download both SIDL files and create their Python bindings to use both libraries with the following command:

```
% babel -cPython -Rhttp://localhost/mystuff/mydepot;http://www.otherhost.com/newstuff
mystuff newstuff
```

To generate SIDL files for each package based on the XML stored in the repository, the following command command is used:

```
% babel -tSIDL -Rhttp://localhost/mystuff/mydepot;http://www.otherhost.com/newstuff
mystuff newstuff
```

3.3 Fundamental Types

Table 3.2 briefly shows the different data types that are supported in Babel and some of the language specific bindings to that SIDL type. The "S" in SIDL stands for "Scientific." This emphasis is reflected in the fundametal support for complex numbers (`fcomplex` and `dcomplex`) and dynamic multidimensional arrays (`array<Type,Dim>`).

C++ developers looking at the SIDL syntax for arrays, might think that SIDL is a templated IDL, but this is not so. Although the syntax for SIDL arrays looks like a template, it is specific only to the array type. Developers cannot create templated classes or methods in SIDL.

> **Rationale:** *Although C++ templates are a very powerful programming mechanism, they apply only to C++. For Babel to implement similar hashing routines, method names in languages other than C++ would become prohibitively (thousands of characters) long. Moreover, this C++ template hashing mechanism is compiler specific so while C++ is very good at hiding the expanded template names (unless there is an error to report) we would have to add babel C++ bindings on a compiler by compiler basis.*

Discussion of the various types is broken up into sections. Numeric types such as `bool`, `char`, `int`, `long`, `float`, `double`, `fcomplex`, and `dcomplex` are discussed in SubSection 3.3.1. Discussion of `string`s is found in SubSection 3.3.2. A brief justification for the `opaque` type is in SubSection 3.3.4. Information about enumerated types is presented in SubSection 3.3.4 which concludes our discussion of fundamental types and this section. Information about extended types such as Interfaces and Classes (Section 3.4) and Arrays (Section 3.5 follow thereafter.

3.3.1 Numeric Types

The SIDL types *bool*, *char*, *int*, *long*, *float*, *double*, *fcomplex*, and *dcomplex* are the smallest and easiest data types to transfer between languages transparently. They all have a fixed size and can just as resonably be copied as passed by reference.

Most languages natively support all of these data types (though perhaps less so with complex types). There are a few notable exceptions that may be of interest.

ANSI C does not define the size of `int` and `long`, only that the latter be at least as big as the former. As of the C99 standard, there are types `int32_t` and `int64_t` that are signed integers that explicitly support a fixed number of bits. Most compilers already have these symbols defined appropriately in `sys/types.h` (pre C99 standard) or `inttypes.h`.

Python defines its `int` and `long` to be equivalent to C, and therefore suffers the same platform dependent integer size problem with less flexibility for a workaround. It is not uncommon for regression tests involving longs and Python to fail on certain platforms. Python 2.2 has a patch to make SIDL long support better.

3.3.2 Strings

Strings are an interesting datatype because they are fundamental to many pieces of software, but represented differently by practically every single programming language. Strings can have a high overhead to support language interoperability because there is invariably so much copying involved.

FORTRAN 77 and 90 support for strings is limited to a predetermined buffer size. Since the results of a string assignment into that buffer in FORTRAN does not propagate the length of the string, trailing whitespace is always trimmed for any string begin passed out from a FORTRAN implementation.

3.3.3 Opaque

The *opaque* type is dangerous, and rarely useful. However, there are particular times when an opaque type is the only way to solve a problem. When a SIDL file uses an *opaque* type, Babel guarantees only bits will be relayed exactly between caller and callee. If there is a need to pass more information than an opaque provides, than the developer can simply pass a pointer to that information.

Use of a *opaque* carries a heavy penalty. When Babel matures enough to support distributed computing, any method calls with *opaque* in the argument list (or return type) will be restricted to in-process calls only.

> **Rationale:** *Since `opaque` is typically used for a pointer to memory, this sequence of bits has no meaning outside of its own process space.*

3.3.4 Enumerations

An enumeration is typically used in programming languages to specify a limited range of states and deal with them by names instead of hard-coded values. For language interoperability purposes — especially to support this concept on languages with no native support — we've had to create specific rules for the integer values associated with enumerated types.

```
package enumSample version 1.0 {

  // undefined integer values
  enum color {
    red, orange, yellow, green, blue, violet
  };

  // completely defined integer values
  enum car {
    porsche = 911,
    ford = 150,
```

```
    mercedes = 550
  };

  // partially defined integer value
  enum number {
    notZero,
    notOne,
    zero=0,
    one=1,
    negOne=-1,
    notNeg
  };
}
```

Above is a sample of enumerations taken directly from our regression tests. It defines a package *enumSample* that contains three enumerations. C/C++ developers will find the syntax very familiar. When defining an enumeration, the actual integer values assigned can be undefined, completely defined, or partially defined.

SIDL defines the following rules for adding integer values to enumerated states that don't have a value explicitly defined.

1. Error if two states are explicitly assigned the same value
2. Assign all explicit values to their named state.
3. Assign smallest unused non-negative value to first unassigned state in enumeration.
4. Repeat 3 until all states have assigned (unique) values.

To verify the application of these rules, the *enumSample.number* enumeration will have the following values assigned to its states: *NotZero*=2, *NotOne*=3, *zero*=0; *one*=1, *negOne*=-1, *notNeg*=4.

3.4 Objects

One of the strategies that SIDL uses to enforce language interoperability is to define an object model that it supports across all language bindings. This enables real object-oriented programming in non OO languages such as C and FORTRAN 77. This also means that the inheritance mechanisms inside real OO languages may be circumvented.

Contrary to newer scripting languages such as Python and Ruby, not everything in SIDL is an object. Only classes (abstract or not) and interfaces are objects. Everything else (e.g. arrays, enums, strings, ints) is something other than an object and therefore outside the scope of this Section.

3.4.1 Babel's Object Model

SIDL defines three types of objects: interfaces, classes, and abstract classes. A SIDL *interface* is akin to a Java interface or a C++ pure abstract base class. It is an object that defines methods (aka member functions), but carries no implementation of those methods. A *class* by comparision is always concrete; meaning that there is an implementation for each of its methods and it can be instantiated. An *abstract class* falls somewhere between an *interface* and a *class*. It has at least one method unimplemented, so it cannot be instantiated, but it also may have several methods that are implemented and these implementations can be inherited.

SIDL supports multiple inheritance of interfaces and single inheritance of implementation. This is a strategy found in other OO languages such as Java and ObjectiveC. The words to distinguish these two forms of inheritance are *extends* and *implements*. Interfaces can extend multiple interfaces, but they cannot implement anything. Classes can extend at most one other class (abstract or not), but can implement multiple interfaces.

We display a small SIDL file below and finish this SubSection with a discussion of its details.

```
package object version 1.0 {

  interface A {
    void display();
    void printMe();
  }

  abstract class B implements A {
    void display();
  }

  class C extends B {
    void printMe();
  }

  class D implements-all A {
  }
}
```

`object.A` is an interface that has two methods `display()` and `print()`. Both of these methods take no arguments and return no value. (We will discuss arguments and return values in the next section.) Since `object.A` is an interface, there is no implementation associated with it, and Babel will not generate any implementation code associated with it.

`object.B` is an abstract class that inherits from `object.A`. Since it redeclares the `display()` method, Babel will generate the appropriate code for an implementation of this method only. It will not generate code for the other inherited method `print()` (since it wasn't declared in the SIDL file) and it will not generate constructors/destructors since the class is abstract.

`object.C` is a class that extends the abstract class `object.B` it then lists only the unimplemented method `print()`, implying that it will use the implementation of `display()` it inherited from its parent.

`object.D` is also a class that uses the `implements-all` directive. This is identical to using `implements` and then listing all the methods declared in the interface. The `implements-all` directive was added to SIDL as a convenience construct and to save excessive typing in the SIDL file. By virtue of the `implements-all` directive, `object.D` will provide its own implementation of all of `object.A`'s methods, namely `display()` and `print()`.

3.4.2 Methods on Objects

Methods in SIDL are virtual by default. This means that the actual binding of a method invokation to an actual implementation is determined at runtime, based on the concrete type of the object.

SIDL currently defines three modifiers to methods that change their default behavior.

- `final` : Final methods are the opposite of virtual. While they may still be inherited by child classes, they cannot be overridden.
- `static` : Static methods do not depend on an instance. In non-OO languages, this means that the typical first argument of an instance is removed. In OO languages, these are mapped directly to an Java or C++ static method.
- `oneway` : reserved for future use.

3.4.3 Parameter Passing

Each parameter in a method call obeys the following syntax

```
[ (modifier) ] (mode) (type) (name)
```

Where `(mode)` is one of *in*, *out*, or *inout*; `(type)` is any SIDL recognized type; and `(name)` is any non-reserved word. The `(modifier)` is optional, and currently unimplemented. SIDL currently reserves the word *copy* for future use as an parameter modifier, and may add others in the future[2].

For new users, the parameter's mode (e.g. *in*, *out*, or *inout*) is perhaps the most troublesome. On the surface, it's easy to explain that *in* parameters are passed into the code, *out* parameters come out, and *inout* parameters do both. However, there are some deeper issues that users need to be aware of.

1. *in* does not mean `const`.
2. *inout* may destroy the input instance and replace it with a completely new one.
3. Types created on the stack should never be passed as an *inout* argument, since the implementation may want to destroy it.

Need a lot more here. There are subtle issues that really need to be discussed. FIX ME!

3.4.4 Method Overloading

Method overloading is the object-oriented practice of defining more than one method with the same name in a class. Doing so allows the convenient reuse of a method name when, for example, the underlying implementations differ based on the types of the arguments. Actually, support for overloaded methods typically relies on the signature of each method to ensure uniqueness. In this case, the signature consists of the method name along with the number, types, and ordering of its arguments.

Since Babel supports languages that do not support method overloading, a mechanism for generating unique names was needed. These are typically generated by compilers based on hashing the argument types into the method name. However, developers often manually address this with far fewer characters than would be used by a compiler. Consequently, it was determined it would be more efficient to leave the task of identifying the unique name to the developer. Therefore, Babel allows the specification of the base, or short, method name along with an optional method name extension as illustrated in the SIDL file below for the `getValue` method.

```
package Overload version 1.0 {

  class Sample {
    int      getValue ( );
    int      getValue[Int]( in int v );
    double   getValue[Double]( in double v );
  }
}
```

Thus, the full method name is the concatenation of the short name followed by the name extension. When generating code for supported languages, Babel makes use of either the short or full method name as appropriate for the language(s) involved. For those that support method overloading, such as C++ and Java, Babel relies only on the short method name, thus ignoring the extension. For the rest, like C, Fortran, and Python, Babel must make use of the full name to ensure methods are uniquely identified.

In the example above, the first method specification takes no arguments so has no name extension. This is acceptable because there are no potentially conflicting methods at this point for any programming language supported by Babel. The second method, with the user-defined name extension of `Int`, takes a single int argument, resulting in the unique method name `getValueInt`. The last method, with a user-defined name extension of `Double`, takes a single double argument, resulting in the unique method name of `getValueDouble`. Examples of calling overloaded methods from Babel-supported languages can be found in the respective language binding chapters.

[2] Babel is still pre-1.0 after all!

3.4.5 SIDL Base Classes

There are a small collection of objects that are defined by the SIDL runtime library. Some of these objects are implicitly inherited by objects and classes.

All classes that do not explictly extend another class implicitly extend *SIDL.BaseClass*. All interfaces that do not explicitly extend another interface implicitly extend *SIDL.BaseInterface*. Furthermore, *SIDL.BaseClass* implements *SIDL.BaseInterface*. This means that all classes can be cast to a *SIDL.BaseClass* and all objects can be cast to *SIDL.BaseInterface*

All exceptions must explicitly inherit from *SIDL.BaseException*. If a method in SIDL claims to throw an object that does not inherit from *SIDL.BaseException*, this is an error and will be reported by Babel.

3.5 Arrays

One of the features that separates SIDL and BABEL from Microsoft's COM/DCOM and the OMG's CORBA is support for multi-dimensional arrays. SIDL is designed to serve the high performance computing community, so we anticipate that both SIDL object developers and object clients may require direct access to the underlying array data structure to try to optimize instruction pipelining or cache performance. The purpose of this document is to describe the functional API to the SIDL array data structure and the underlying data structures. This presentation will focus on the C API for arrays because it is the basis for the other language APIs, so they will likely reflect its idiosyncrasies.

SIDL arrays can be "row-major" or "column-major", really. They are not parallel array classes, and not particularly sophisticated, but they are very, very general. These are meant to generalize the array types built into many languages, not to provide a general array component that everyone will use. It is expected for parallel array libraries to build on top of the array type presented into SIDL.

3.5.1 SIDL Language Features

As of release 0.6.5, interface defintions can specify that an array argument or return value must have a particular ordering for a method. The type `array<int, 2, row-major>` indicates a dense,[3] two-dimensional array of 32 bit integers in row-major order; and likewise, the type `array<int, 2, column-major>` indicates an dense array in column-major order. Some numerical routines can only provide high performance with a particular type of array. The ordering is part of the interface definition to give clients the information they need to use the underlying code efficiently. The ordering specification is optional.

For one-dimensional arrays, specifying `row-major` or `column-major` allows you to specify that the array must be dense, that is stride 1. Otherwise, for one-dimensional arrays row-major and column-major are identical.

If you pass an array into a method and the array does not have the specified ordering, the skeleton code will make a copy of the array with the required ordering and pass the copy to the method. This copying is necessary for correctness, but it will cause a decrease in performance. The implementor of the method can count on an incoming array to have the required ordering.

For `out` parameters and return values, an ordering specification means that the method promises to return an array with the specified ordering. The implementation should create the `out` arrays with the proper ordering; because if it does not, the skeleton code will have to copy the outgoing array into a new array with the required ordering.

For `inout` parameters, an ordering specification means the ordering specification will be enforced by the skeleton code for the incoming and outgoing array value.

At the time of writing this, the ordering constraints are enforced for Python implementation because Python uses Numeric Python arrays, so BABEL cannot control the array ordering as fully. The Python skeletons do force outgoing arrays (i.e., arrays passed back from Python) to have the required ordering.

[3] meaning nonstrided

3.5.2 Independent and borrowed arrays

There are two main kinds of arrays: independent and borrowed. The independent arrays owns and manages its data. It allocates space for the array elements when the array is created, and it deallocates that space when the array is finally destroyed.

The borrowed array does not own or manage its data. It borrows its array element data from another source that it cannot manage, and it only allocates space for the index bounds and stride information. The rationale for borrowed arrays is to allow data from another source to temporarily appear as a SIDL array without requiring data be copied.

If you `slice` an independent array, the resulting array is also considered independent even though it borrows data from the original independent array. The resulting array can still manage its data by retaining a reference to the original array; hence, its element data cannot disappear until the resulting array is destroyed. If you `slice` a borrowed array, the resulting array is also borrowed because like its original array, it doesn't manage the underlying data.

3.5.3 The Life of an Array

The existence of borrowed arrays causes the arrays to deviate from the normal reference counting pattern. Arrays are reference counted. An array's resources are reclaimed when the reference count goes to zero. However, a borrowed array's array element data will disappear whenever the source of the borrowed data determines that it should regardless of the reference count in corresponding the SIDL array. This behavior means that developers should consider any SIDL array that they did not create themselves, for example incoming arguments to methods, as potential borrowed arrays. When a method wants to keep a copy of an array that might be a borrowed array, it should use the `smartCopy` method documented below.

Here are some rules of thumb about the use of borrowed arrays:

- The creator of a borrowed array should guarantee that the data for the borrowed array will exist through the duration of any method calls using the borrowed array.
- Methods should not return a borrowed array as a return value or `out` parameter unless the method can guarantee that the array element data will be available until the process shuts down.
- There is a neglible performance cost when using `smartCopy` when the array is not borrowed, and there is a huge correctness benefit when the array is borrowed.

3.5.4 The Language Bindings

The C++ binding for array provides access to the C API in case you need to take the gloves off and revel in the data directly. But the C++ binding also provides a templated wrapper class to provide a more natural look and feel for C++ programmers.

The Python binding for arrays involves copying SIDL arrays to/from Numeric Python arrays. Arrays in Python don't have the SIDL methods available. They just have the Numeric Python API available.

The FORTRAN 77 API mimics the C API; all the C functions have been FORTRANified and have _f appended to their names. Similarly, the FORTRAN 90 API appends _m to each of the C function names.

3.5.5 The Array API

In the following presention, we use the SIDL *`int`* type; however, everything in this section applies to all types except where noted. The basic types are in the SIDL namespace. Table 3.3 shows the prefix for SIDL base types and the actual value type held by the array...

For arrays of interfaces or classes, the name of the array function prefix is derived from the fully qualified type name. For example, for the type *`SIDL.BaseClass`*, the array functions all begin with `SIDL_BaseClass`. For *`SIDL.BaseInterface`*, they all begin with `SIDL_BaseInterface`.

When you add an object or interface to an array, the reference count of the element being overwritten is decremented, and the reference count of the element being added is incremented. When you get an object or interface from an array, the caller owns the returned reference.

Table 3.3: SIDL types to array function prefixes

SIDL type	Array function prefix	Value type
bool	SIDL_bool	SIDL_bool
char	SIDL_char	char
dcomplex	SIDL_dcomplex	struct SIDL_dcomplex
double	SIDL_double	double
fcomplex	SIDL_fcomplex	struct SIDL_fcomplex
float	SIDL_float	float
int	SIDL_int	int32_t
long	SIDL_long	int64_t
opaque	SIDL_opaque	void *
string	SIDL_string	char *

For arrays of strings when you add a string to any array, the array will store a copy of the string. When you retrieve a string from an array, you will receive a copy of the string. You should `SIDL_String_free` the returned string when you are done with it.

When you create an array of interfaces, classes, or strings, all elements of the array are initialized to NULL. Other arrays are not initialized. When an array of interfaces, classes, or strings is destroyed, it releases any held references in the case of objects or interfaces. In the case of strings, it frees any non-NULL pointers.

The name of the data structure that holds the array if int is `struct SIDL_int__array`. For some types, the data structure is an opaque type, and for others, it is defined in a public C header file.

Here are the functions one-by-one:

```
/* C */
struct SIDL_int__array*
SIDL_int__array_createCol(int32_t       dimen,
                          const int32_t lower[],
                          const int32_t upper[]);
//
// C++
static SIDL::array<int>
SIDL::array<int>::createCol(int32_t       dimen,
                            const int32_t lower[],
                            const int32_t upper[]);
C
C FORTRAN 77
       subroutine SIDL_int__array_createCol_f(dimen, lower, upper, result)
       integer*4 dimen
       integer*4 lower(dimen), upper(dimen)
       integer*8 result
!
! FORTRAN 90
subroutine SIDL_int__array_createCol_m(dimen, lower, upper, result)
  integer (selected_int_kind(9)) :: dimen
  integer (selected_int_kind(9)) :: lower(dimen), upper(dimen)
  integer (selected_int_kind(18)) :: result
```

This method creates a column-major, multi-dimensional array in a contiguous block of memory. `dimen` should be strictly greater than zero, and `lower` and `upper` should have `dimen` elements. `lower[i]` must be less than or equal to `upper[i]-1` for $i \geq 0$ and $i < \texttt{dimen}$. If this function fails for some reason, it returns `NULL`. `lower[i]` specifies the smallest valid index for dimension `i`, and `upper[i]` specifies the largest. Note this definition is somewhat un-C like where the upper bound is often one past the end. In SIDL, the size of dimension `i` is `1 + upper[i] - lower[i]`.

The function makes copies of the information provided by `dimen`, `lower`, and `upper`, so the caller is not obliged to maintain those values after the function call.

For FORTRAN, the new array is returned in the last parameter, `result`. A zero value in `result` indicates that the operation failed.

```
/* C */
struct SIDL_int__array*
SIDL_int__array_createRow(int32_t       dimen,
                          const int32_t lower[],
                          const int32_t upper[]);
//
// C++
static SIDL::array<int>
SIDL::array<int>::createRow(int32_t       dimen,
                            const int32_t lower[],
                            const int32_t upper[]);
C
C FORTRAN 77
       subroutine SIDL_int__array_createRow_f(dimen, lower, upper, result)
       integer*4 dimen
       integer*4 lower(dimen), upper(dimen)
       integer*8 result
!
! FORTRAN 90
subroutine SIDL_int__array_createRow_m(dimen, lower, upper, result)
  integer (selected_int_kind(9)) :: dimen
  integer (selected_int_kind(9)) :: lower(dimen), upper(dimen)
  integer (selected_int_kind(18)) :: result
```

This method creates a row-major, multi-dimensional array in a contingguous block of memory. Other than the difference in the ordering of the array elements, this method is identical to `createCol`.

```
/* C */
struct SIDL_int__array*
SIDL_int__array_create1d(int32_t len);

// C++
static SIDL::array<int>
SIDL::array<int>::create1d(int32_t len);

C FORTRAN 77
       subroutine SIDL_int__array_create1d_f(len, result)
       integer*4 len
       integer*8 result

! FORTRAN 90
subroutine SIDL_int__array_create1d_m(len, result)
  integer (selected_int_kind(9)) :: len
  integer (selected_int_kind(18)) :: result
```

This method creates a dense, one-dimensional vector of ints with a lower index of 0 and an upper index of $len - 1$. This is defined primarily as a convenience for C and C++ programmers. If $len \leq 0$, this routine returns NULL.

```
/* C */
struct SIDL_int__array*
```

```
SIDL_int__array_create2dCol(int32_t m, int32_t n);

// C++
static SIDL::array<int>
SIDL::array<int>::create2dCol(int32_t m, int32_t n);

C FORTRAN 77
       subroutine SIDL_int__array_create2dCol_f(m, n, result)
       integer*4 m, n
       integer*8 result

! FORTRAN 90
subroutine SIDL_int__array_create2dCol_m(m, n, result)
  integer (selected_int_kind(9)) :: m, n
  integer (selected_int_kind(18)) :: result
```

This method creates a dense, column-major, two-dimensional array of ints with a lower index of $(0, 0)$ and an upper index of $(m-1, n-1)$. If $m \leq 0$ or $n \leq 0$, this method returns NULL. This is defined primarily as a convenience for C and C++ programmers.

```
/* C */
struct SIDL_int__array*
SIDL_int__array_create2dRow(int32_t m, int32_t n);

// C++
static SIDL::array<int>
SIDL::array<int>::create2dRow(int32_t m, int32_t n);

C FORTRAN 77
       subroutine SIDL_int__array_create2dRow_f(m, n, result)
       integer*4 m, n
       integer*8 result

! FORTRAN 90
subroutine SIDL_int__array_create2dRow_m(m, n, result)
  integer (selected_int_kind(9)) :: m, n
  integer (selected_int_kind(18)) :: result
```

This method creates a dense, row-major, two-dimensional array of ints with a lower index of $(0, 0)$ and an upper index of $(m-1, n-1)$. If $m \leq 0$ or $n \leq 0$, this method returns NULL. This is defined primarily as a convenience for C and C++ programmers.

```
/* C */
struct SIDL_int__array *
SIDL_int__array_slice(struct SIDL_int__array *src,
                      int32_t       dimen,
                      const int32_t numElem[],
                      const int32_t *srcStart,
                      const int32_t *srcStride,
                      const int32_t *newStart);
//
// C++
array<int>
SIDL::array<int>::slice(int dimen,
                        const int32_t newElem[],
                        const int32_t *srcStart = 0,
                        const int32_t *srcStride = 0,
```

```
                          const int32_t *newStart = 0);
C
C FORTRAN 77
       subroutine SIDL_int__array_slice_f(src, dimen, numElem, srcStart,
      $                       srcStride, newStart)
       integer*8 src, result
       integer*4 dimen
       integer*4 numElem(srcDimen), srcStart(srcDimen)
       integer*4 srcStride(srcDimen),  newStart(dimen)
!
! FORTRAN 90
subroutine SIDL_int__array_slice_m(src, dimen, numElem, srcStart, srcStride, &
                                   newStart)
  integer (selected_int_kind(18)) :: src, result
  integer (selected_int_kind(9)) :: dimen
  integer (selected_int_kind(9)) :: numElem(srcDimen), srcStart(srcDimen)
  integer (selected_int_kind(9)) :: srcStride(srcDimen),  newStart(dimen)
```

This method will create a sub-array of another array. The resulting array shares data with the original array. The new array can be of the same dimension or potentially less than the original array. If you are removing a dimension, indicate the dimensions to remove by setting `numElem[i]` to zero for any dimension `i` that should go away in the new array. The meaning of each argument is covered below.

src the array to be created will be a subset of this array. If this argument is NULL, NULL will be returned. The returned array borrows data from `src`, so modifying one array modifies both. In C++, the `this` pointer takes the place of `src`.

dimen this argument must be greater than zero and less than or equal to the dimension of `src`. An illegal value will cause a NULL return value.

numElem this specifies how many elements from src should be in the new array in each dimension. A zero entry indicates that the dimension should not appear in the new array. This argument should be an array with an entry for each dimension of `src`. If $srcStart[i] + numElem[i] srcStride[i] > upper[i]$ or $srcStart[i] + numElem[i] srcStride[i] < lower[i]$ for src, NULL will be returned.

srcStart this parameter specifies which element of `src` will be the first element of the new array. If this argument is NULL, the first element of `src` will be the first element of the new array. If non-NULL, this argument provides the coordinates of an element of `src`, so it must have an entry for each dimension of `src`. If $srcStart[i] < lower[i]$ or $srcStart[i] > upper[i]$ for `src`, NULL will be returned.

srcStride this argument lets you specify the stride between elements of `src` for each dimension. For example with a stride of 2, you could create a sub-array with only the odd or even elements of `src`. If this argument is NULL, the stride is taken to be one in each dimension. If non-NULL, this argument should be an array with an entry for each dimension of `src`.

newLower this argument is like the `lower` argument in a create method. It sets the coordinates for the first element in the new array. If this argument is NULL, the values indicated by `srcStart` will be used. If non-NULL, this should be an array with `dimen` elements.

Assuming the method is successful and the return value is named newArray, `src[srcStart]` refers to the same underlying element as `newArray[newStart]`.

If `src` is not a borrowed array (i.e., it manages its own data), the returned array can manage its by keeping a reference to `src`. It is not considered a borrowed array for purposes of `smartCopy`.

```
/* C */
struct SIDL_int__array*
```

```
SIDL_int__array_borrow(int32_t*      firstElement,
                       int32_t       dimen,
                       const int32_t lower[],
                       const int32_t upper[],
                       const int32_t stride[]);
//
// C++
void
SIDL::array<int>::borrow(int32_t*      firstElement,
                         int32_t       dimen,
                         const int32_t lower[],
                         const int32_t upper[],
                         const int32_t stride[]);
C
C FORTRAN 77
       subroutine SIDL_int__array_borrow_f(firstElement, dimen, lower, upper,
     $     stride, result)
       integer*4 firstElement(), dimen, lower(dimen), upper(dimen)
       integer*4 stride(dimen)
       integer*8 result
!
! FORTRAN 90
subroutine SIDL_int__array_borrow_m(firstElement, dimen, lower, upper, stride, &
                                    result)
  integer (selected_int_kind(9)) :: firstElement(), dimen
  integer (selected_int_kind(9)) :: lower(dimen), upper(dimen), stride(dimen)
  integer (selected_int_kind(18)) :: result
```

This method creates a proxy SIDL multi-dimensional array using data provided by a third party. In some cases, this routine can be used to avoid making a copy of the array data. `dimen`, `lower`, and `upper` have the same meaning and constraints as in `SIDL_int__array_createCol`. The `firstElement` argument should be a pointer to the first element of the array; in this context, the first element is the one whose index is `lower`.

`stride[i]` specifies the signed offset from one element in dimension `i` to the next element in dimension `i`. For a one dimensional array, the first element has the address `firstElement`, the second element has the address `firstElement + stride[0]`, the third element has the address `firstElement + 2 * stride[0]`, etc. The algorithm for determining the address of the element in a multi-dimensional array whose index is in array `ind[]` is as follows:

```
int32_t* addr = firstElement;
for(int i = 0; i < dimen; ++i) {
  addr += (ind[i] - lower[i])*stride[i];
}
/* now addr is the address of element ind */
```

Note elements of stride need not be positive.

The function makes copies of the information provided by `dimen`, `lower`, `upper`, and `stride`. The type of `firstElement` is changed depending on the array value type (see Table 3.3).

```
/* C */
struct SIDL_int__array*
SIDL_int__array_smartCopy(struct SIDL_int__array *array);

// C++
void
SIDL::array<int>::smartCopy();
```

```
C FORTRAN 77
        subroutine SIDL_int__array_smartCopy_f(array, result)
        integer*8 array, result

! FORTRAN 90
subroutine SIDL_int__array_smartCopy_m(array, result)
  integer (selected_int_kind(18)) :: array, result
```

This method will copy a borrowed array or increment the reference count of an array that is able to manage its own data. This method is useful when you want to keep a copy of an incoming array. The C++ method operates on `this`.

```
/* C */
void
SIDL_int__array_addRef(struct SIDL_int__array* array);

// C++
void
SIDL::array<int>::addRef() throw ( NullIORException );

C FORTRAN 77
        subroutine SIDL_int__array_addRef_f(array)
        integer*8 array

! FORTRAN 90
subroutine SIDL_int__array_addRef_m(array)
  integer (selected_int_kind(18)) :: array
```

This increments the reference count by one. In C++, this method should be avoided because the C++ wrapper class manages the reference count for you.

```
/* C */
void
SIDL_int__array_deleteRef(struct SIDL_int__array* array);

// C++
void
SIDL::array<int>::deleteRef() throw ( NullIORException );

C FORTRAN 77
        subroutine SIDL_int__array_deleteRef_f(array)
        integer*8 array

! FORTRAN 90
subroutine SIDL_int__array_deleteRef_m(array)
  integer (selected_int_kind(18)) :: array
```

This decreases the reference count by one. If this reduces the reference count to zero, the resources associated with the array are reclaimed. In C++, this method should be avoided because the C++ wrapper class manages the reference count for you.

```
/* C */
int32_t
SIDL_int__array_get1(const struct SIDL_int__array* array,
                     int32_t                       i1);

// C++
int32_t
```

```
SIDL::array<int>::get(int32_t i1);

C FORTRAN 77
       subroutine SIDL_int__array_get1_f(array, i1, result)
       integer*8 array
       integer*4 i1, result

! FORTRAN 90
subroutine SIDL_int__array_get1_m(array, i1, result)
  integer (selected_int_kind(18)) :: array
  integer (selected_int_kind(9)) :: i1, result
```

This method returns the element with index `i1` for a one dimensional array. The return type of this method is the value type for the SIDL type being held (see Table 3.3). This method must only be called for one dimensional arrays. For objects and interfaces, the client owns the returned reference (i.e., the client is obliged to call `deleteRef()` when they are done with the reference unless it is `NULL`). For arrays of strings, the client owns the returned string (i.e., the client is obliged to call free on the returned pointer unless it is `NULL`). There is no reliable way to determine from the return value cases when i1 is out of bounds.

```
/* C */
int32_t
SIDL_int__array_get2(const struct SIDL_int__array* array,
                     int32_t                        i1,
                     int32_t                        i2);

// C++
int32_t
SIDL::array<int>::get(int32_t i1, int32_t i2);

C FORTRAN 77
       subroutine SIDL_int__array_get2_f(array, i1, i2, result)
       integer*8 array
       integer*4 i1, i2, result

! FORTRAN 90
subroutine SIDL_int__array_get2_m(array, i1, i2, result)
  integer (selected_int_kind(18)) :: array
  integer (selected_int_kind(9)) :: i1, i2, result
```

This method returns the element with indices (`i1`, `i2`) for a two dimensional array. The return type of this method is the value type for the SIDL type being held (see Table 3.3. This method must only be called for two dimensional arrays. For objects and interfaces, the client owns the returned reference (i.e., the client is obliged to call `deleteRef` when they are done with the reference unless it is `NULL`). For arrays of strings, the client owns the returned string (i.e., the client is obliged to call free on the returned pointer unless it is `NULL`). There is no reliable way to determine from the return value cases when `i1`, `i2` are out of bounds.

```
/* C */
int32_t
SIDL_int__array_get3(const struct SIDL_int__array* array,
                     int32_t                        i1,
                     int32_t                        i2,
                     int32_t                        i3);
// C++
int32_t
SIDL::array<int>::get(int32_t i1, int32_t i2, int32_t i3);
```

```
C FORTRAN 77
       subroutine SIDL_int__array_get3_f(array, i1, i2, i3, result)
       integer*8 array
       integer*4 i1, i2, i3, result

! FORTRAN 90
subroutine SIDL_int__array_get3_m(array, i1, i2, i3, result)
  integer (selected_int_kind(18)) :: array
  integer (selected_int_kind(18)) :: i1, i2, i3, result
```

This method returns the element with indices (`i1`, `i2`, `i3`) for a three dimensional array. The return type of this method is the value type for the SIDL type being held (see Table 3.3). This method must only be called for three dimensional arrays. For objects and interfaces, the client owns the returned reference (i.e., the client is obliged to call `deleteRef()` when they are done with the reference unless it is `NULL`). For arrays of strings, the client owns the returned string (i.e., the client is obliged to call `free()` on the returned pointer unless it is `NULL`). There is no reliable way to determine from the return value cases when i1, i2, i3 are out of bounds.

```
/* C */
int32_t
SIDL_int__array_get4(const struct SIDL_int__array* array,
                     int32_t                        i1,
                     int32_t                        i2,
                     int32_t                        i3,
                     int32_t                        i4);
// C++
int32_t
SIDL::array<int>::get(int32_t i1, int32_t i2, int32_t i3, int32_t i4);

C FORTRAN 77
       subroutine SIDL_int__array_get4_f(array, i1, i2, i3, i4, result)
       integer*8 array
       integer*4 i1, i2, i3, i4, result

! FORTRAN 90
subroutine SIDL_int__array_get4_m(array, i1, i2, i3, i4, result)
  integer (selected_int_kind(18)) :: array
  integer (selected_int_kind(9)) :: i1, i2, i3, i4, result
```

This method returns the element with indices(`i1`, `i2`, `i3`, `i4`) for a four dimensional array. The return type of this method is the value type for the SIDL type being held (see Table 3.3). This method must only be called for four dimensional arrays. For objects and interfaces, the client owns the returned reference (i.e., the client is obliged to call `deleteRef()` when they are done with the reference unless it is `NULL`). For arrays of strings, the client owns the returned string (i.e., the client is obliged to call `free()` on the returned pointer unless it is `NULL`). There is no reliable way to determine from the return value cases when `i1`, `i2`, `i3`, or `i4` are out of bounds.

```
/* C */
int32_t
SIDL_int__array_get(const struct SIDL_int__array* array,
                    const int32_t                 indices[]);

// C++
int32_t
SIDL::array<int>::get(const int32_t indices[]);

C FORTRAN 77
       subroutine SIDL_int__array_get_f(array, indices, result)
```

```
       integer*8 array
       integer*4 indices(), result

! FORTRAN 90
subroutine SIDL_int__array_get_m(array, indices, result)
  integer (selected_int_kind(18)) :: array
  integer (selected_int_kind(9)) :: indices(), result
```

This method returns the element whose index is indices for an array of any dimension. The return type of this method is the value type for the SIDL type being held (see Table 3.3). This method can be called for any positively dimensioned array. For objects and interfaces, the client owns the returned reference (i.e., the client is obliged to call `deleteRef()` when they are done with the reference unless it is `NULL`). For arrays of strings, the client owns the returned string (i.e., the client is obliged to call `free()` on the returned pointer unless it is `NULL`). There is no reliable way to determine from the return value cases when indices has an element out of bounds.

```
/* C */
int32_t
SIDL_int__array_set2(const struct SIDL_int__array* array,
                     int32_t                       i1,
                     int32_t                       value));

// C++
int32_t
SIDL::array<int>::set(int32_t value, int32_t i1);

C FORTRAN 77
       subroutine SIDL_int__array_set1_f(array, i1, value)
       integer*8 array
       integer*4 i1, value

! FORTRAN 90
subroutine SIDL_int__array_set1_m(array, i1, value)
  integer (selected_int_kind(18)) :: array
  integer (selected_int_kind(9)) :: i1, value
```

This method sets the value in index `i1` of a one dimensional array to value. The type of the argument value is the value type for the SIDL type being held (see Table 3.3). This method must only be called for one dimensional arrays. For arrays of objects and interfaces, the array will make its own reference by calling `addRef()` on value, so the client retains its reference to value. For arrays of strings, the array will make a copy of the string, so the client retains ownership of the value pointer.

```
/* C */
int32_t
SIDL_int__array_set2(const struct SIDL_int__array* array,
                     int32_t                       i1,
                     int32_t                       i2,
                     int32_t                       value));

// C++
int32_t
SIDL::array<int>::set(int32_t value, int32_t i1, int32_t i2);

C FORTRAN 77
       subroutine SIDL_int__array_set2_f(array, i1, i2, value)
       integer*8 array
       integer*4 i1, i2, value
```

```
! FORTRAN 90
subroutine SIDL_int__array_set2_m(array, i1, i2, value)
  integer (selected_int_kind(18)) :: array
  integer (selected_int_kind(9)) :: i1, i2, value
```

This method sets the value in index (`i1`, `i2`) of a two dimensional array to value. The type of the argument value is the value type for the SIDL type being held (see table 3.3). This method must only be called for two dimensional arrays. For arrays of objects and interfaces, the array will make its own reference by calling `addRef()` on value, so the client retains its reference to value. For arrays of strings, the array will make a copy of the string, so the client retains ownership of the value pointer.

```
/* C */
int32_t
SIDL_int__array_set3(const struct SIDL_int__array* array,
                     int32_t                       i1,
                     int32_t                       i2,
                     int32_t                       i3,
                     int32_t                       value));

// C++
int32_t
SIDL::array<int>::set(int32_t value, int32_t i1, int32_t i2, int32_t i3);

C FORTRAN 77
       subroutine SIDL_int__array_set3_f(array, i1, i2, i3, value)
       integer*8 array
       integer*4 i1, i2, i3, value

! FORTRAN 90
subroutine SIDL_int__array_set3_m(array, i1, i2, i3, value)
  integer (selected_int_kind(18)) :: array
  integer (selected_int_kind(9)) :: i1, i2, i3, value
```

This method sets the value in index (`i1`, `i2`, `i3`) of a three dimensional array to value. The type of the argument value is the value type for the SIDL type being held (see table 3.3). This method must only be called for three dimensional arrays. For arrays of objects and interfaces, the array will make its own reference by calling `addRef()` on value, so the client retains its reference to value. For arrays of strings, the array will make a copy of the string, so the client retains ownership of the value pointer.

```
/* C */
int32_t
SIDL_int__array_set4(const struct SIDL_int__array* array,
                     int32_t                       i1,
                     int32_t                       i2,
                     int32_t                       i3,
                     int32_t                       i4,
                     int32_t                       value));
//
// C++
int32_t
SIDL::array<int>::set(int32_t value, int32_t i1, int32_t i2,
                     int32_t i3, int32_t i4);
C
C FORTRAN 77
       subroutine SIDL_int__array_set4_f(array, i1, i2, i3, i4, value)
       integer*8 array
```

```
       integer*4 i1, i2, i3, i4, value
!
! FORTRAN 90
subroutine SIDL_int__array_set4_m(array, i1, i2, i3, i4, value)
  integer (selected_int_kind(18)) :: array
  integer (selected_int_kind(9)) :: i1, i2, i3, i4, value
```

This method sets the value in index (i1, i2, i3, i4) of a four dimensional array to value. The type of the argument value is the value type for the SIDL type being held (see table 3.3). This method must only be called for four dimensional arrays. For arrays of objects and interfaces, the array will make its own reference by calling `addRef()` on value, so the client retains its reference to value. For arrays of strings, the array will make a copy of the string, so the client retains ownership of the value pointer.

```
/* C */
void
SIDL_int__array_set(struct SIDL_int__array* array,
                    const int32_t           indices[],
                    int32_t                 value);

// C++
void
SIDL::array<int>::set(int32_t value, const int32_t indices[]);

C FORTRAN 77
       subroutine SIDL_int__array_set_f(array, indices, value)
       integer*8 array
       integer*4 indices()

! FORTRAN 90
subroutine SIDL_int__array_set_m(array, indices, value)
  integer (selected_int_kind(18)) :: array
  integer (selected_int_kind(9)) :: indices()
```

This method sets the value in index indices for an array of any dimension to value. The type of the argument value is the value type for the SIDL type being held (see table 3.3). For arrays of objects and interfaces, the array will make its own reference by calling `addRef()` on value, so the client retains its reference to value. For arrays of strings, the array will make a copy of the string, so the client retains ownership of the value pointer.

```
/* C */
int32_t
SIDL_int__array_dimen(const struct SIDL_int__array *array);

// C++
int32_t
SIDL::array<int>::dimen() const;

C FORTRAN 77
       subroutine SIDL_int__array_dimen_f(array, result)
       integer*8 array
       integer*4 result

! FORTRAN 90
subroutine SIDL_int__array_dimen_m(array, result)
  integer (selected_int_kind(18)) :: array
  integer (selected_int_kind(9)) :: result
```

This method returns the dimension of the array.

```
/* C */
int32_t
SIDL_int__array_lower(const struct SIDL_int__array *array, int32_t ind);

// C++
int32_t
SIDL::array<int>::lower(int32_t ind) const;

C FORTRAN 77
       subroutine SIDL_int__array_lower_f(array, ind, result)
       integer*8 array
       integer*4 ind, result

! FORTRAN 90
subroutine SIDL_int__array_lower_m(array, ind, result)
  integer (selected_int_kind(18)) :: array
  integer (selected_int_kind(9)) :: ind, result
```

This method returns the lower bound on the index for dimension `ind` of array.

```
/* C */
int32_t
SIDL_int__array_upper(const struct SIDL_int__array *array, int32_t ind);

// C++
int32_t
SIDL::array<int>::upper(int32_t ind) const;

C FORTRAN 77
       subroutine SIDL_int__array_upper_f(array, ind, result)
       integer*8 array
       integer*4 ind, result

! FORTRAN 90
subroutine SIDL_int__array_upper_m(array, ind, result)
  integer (selected_int_kind(18)) :: array
  integer (selected_int_kind(9)) :: ind, result
```

This method returns the upper bound on the index for dimension ind of array. If the upper bound is greater than or equal to the lower bound, the upper bound is a valid index (i.e., it is not one past the end).

```
/* C */
int32_t
SIDL_int__array_stride(const struct SIDL_int__array *array, int32_t ind);

// C++
int32_t
SIDL::array<int>::stride(int32_t ind) const;

C FORTRAN 77
       subroutine SIDL_int__array_stride_f(array, ind, result)
       integer*8 array
       integer*4 ind, result

! FORTRAN 90
subroutine SIDL_int__array_stride_m(array, ind, result)
  integer (selected_int_kind(18)) :: array
  integer (selected_int_kind(9)) :: ind, result
```

This method returns the stride for a particular dimension. This stride indicates how much to add to a pointer to get for the current element this the particular dimension to the next.

```
/* C */
SIDL_bool
SIDL_int__array_isColumnOrder(const struct SIDL_int__array *array);

// C++
bool
SIDL::array<int>::isColumnOrder() const;

C FORTRAN 77
       subroutine SIDL_int__array_isColumnOrder_f(array, result)
       integer*8 array
       logical   result

! FORTRAN 90
subroutine SIDL_int__array_isColumnOrder_m(array, result)
  integer (selected_int_kind(18)) :: array
  logical :: result
```

This method returns a true value if and only if `array` is dense, column-major ordered array. It does not modify the array at all.

```
/* C */
SIDL_bool
SIDL_int__array_isRowOrder(const struct SIDL_int__array *array);

// C++
bool
SIDL::array<int>::isRowOrder() const;

C FORTRAN 77
       subroutine SIDL_int__array_isRowOrder_f(array, result)
       integer*8 array
       logical   result

! FORTRAN 90
subroutine SIDL_int__array_isRowOrder_m(array, result)
  integer (selected_int_kind(18)) :: array
  logical :: result
```

This method returns a true value if and only if `array` is dense, row-major ordered array. It does not modify the array at all.

```
/* C */
void
SIDL_int__array_copy(const struct SIDL_int__array *src,
                           struct SIDL_int__array *dest);

// C++
void
SIDL::array<int>::copy(const SIDL::array<int> &src);

C FORTRAN 77
       subroutine SIDL_int__array_copy_f(array, dest)
       integer*8 array, dest
```

```
! FORTRAN 90
subroutine SIDL_int__array_copy_m(array, dest)
  integer (selected_int_kind(18)) :: array, dest
```

This method copies the contents of `src` to `dest`. For the copy to take place, both arrays must exist and be of the same dimension. This method will not modify `dest`'s size, index bounds, or stride; only the array element values of `dest` may be changed by this function. No part of `src` is changed by this method.

If `dest` has different index bounds than `src`, this method only copies the elements where the two arrays overlap. If `dest` and `src` have no indices in common, nothing is copied. For example, if `src` is a 1-d array with elements 0-5 and dest is a 1-d array with element 2-3, this function will copy element 2 and 3 from `src` to `dest`. If `dest` had elements 4-10, this method could copy elements 4 and 5.

```
/* C */
struct SIDL_int__array *
SIDL_int__array_ensure(const struct SIDL_int__array *src,
                       int32_t                       dimen,
                       int                           ordering);

// C++
void
SIDL::array<int>::ensure(int32_t dimen, int ordering);

C FORTRAN 77
       subroutine SIDL_int__array_ensure_f(src, dimen, ordering, result)
       integer*8 src, result
       integer*4 dimen, ordering

! FORTRAN 90
subroutine SIDL_int__array_ensure_m(src, dimen, ordering, result)
   integer (selected_int_kind(18)) :: src, result
   integer (selected_int_kind(9)) :: dimen, ordering
```

This method is used to obtain a matrix with a guaranteed ordering and dimension from an array with uncertain properties. If the incoming array has the required ordering and dimension, its reference count is incremented, and it is returned. If it doesn't, a copy with the correct ordering is created and returned. In either case, the caller knows that the returned matrix (if not NULL) has the desired properties.

This method is used internally to enforce the array ordering constraints in SIDL. Clients can use it in similar ways.

The ordering parameter should be one of the constants defined in `enum SIDL_array_ordering` (e.g. `SIDL_general_order`, `SIDL_column_major_order`, or `SIDL_row_major_order`). If you pass in `SIDL_general_order`, this routine will only check the dimension of the matrix.

```
/* C */
int32_t *
SIDL_int__array_first(const struct SIDL_int__array *src);

// C++...Is there an equivalent here?

C FORTRAN 77
      subroutine SIDL_int__array_access_f(array, ref, lower, upper,
     $   stride, index)
      integer*8 array
      integer*4 lower(), upper(), stride(), index
      integer*4 ref()

! FORTRAN 90
```

```
subroutine SIDL_int__array_access_m(array, ref, lower, upper, stride, index)
  integer (selected_int_kind(18)) :: array
  integer (selected_int_kind(9)) :: lower(), upper(), stride(), index
  integer (selected_int_kind(9)) :: ref()
```

This method provides direct access to the element data. Using this pointer and the stride information, you can perform your own array accesses without function calls. This method isn't available for arrays of strings, interface and objects because of memory/reference management issues.

The FORTRAN versions of the method return the lower, upper and stride information in three arrays, each with enough elements to hold an entry for each dimension of `array`. Because FORTRAN 77 does not have pointers, you must pass in a reference array, `array`. Upon exit, `ref(index)` is the first element of the array. The type of `ref` depends on the type of the array.

While calling the FORTRAN direct access routines, there is a possibility of an alignment error between your reference pointer, `ref`. The problem is more likely with arrays of `double` or `dcomplex`; although, it could occur with any type on some future platform. If `index` is zero on return, an alignment error occured. If an alignment error occurs, you may be able to solve it by recompiling your FORTRAN files with flags to force doubles to be aligned on 8 byte boundaries. For example, the `-malign-double` flag for g77 forces doubles to be aligned on 64-bit boundaries. An alignment error occurs when `(char *)ref` minus `(char *)SIDL_int__array_first(array)` is not integer divisible by `sizeof(datatype)` where `ref` refers to the address of the reference array.

Here is an example FORTRAN 77 subroutine to output each element of a 1-dimensional array of doubles using the direct access routine. The FORTRAN 90 version of the code is very similar except for the type mappings, of course.

```
C This subroutine will print each element of an array of doubles
      subroutine print_array(dblarray)
      implicit none
      integer*8 dblarray
      real*8 refarray(1)
      integer*4 lower(1), upper(1), stride(1), index, dimen, i
      if (dblarray .ne. 0) then
         call SIDL_double__array_dimen_f(dblarray, dimen)
         if (dimen .eq. 1) then
            call SIDL_double__array_access_f(dblarray, refarray,
     $           lower, upper, stride, index)
            if (index .ne. 0) then
               do i = lower(1), upper(1)
                  write(*,*) refarray(index + (i-lower(1))*stride(1))
               enddo
            else
               write(*,*) 'Alignment error occured'
            endif
         endif
      endif
      end
```

For a 2-dimensional array, the loop and array access is

```
      do i = lower(1), upper(1)
         do j = lower(2), upper(2)
            write(*,*) refarray(index+(i-lower(1))*stride(1)+
     $           (j - lower(2))*stride(2))
         enddo
      enddo
```

Suppose you are wrapping a legacy FORTRAN application and you need to pass a SIDL array to a FORTRAN subroutine. Further suppose there is a FORTRAN 77 and FORTRAN 90 version of the subroutine. For example, the FORTRAN 77 subroutine has a signature such as:

```
      subroutine TriedAndTrue(x, n)
      integer n
      real*8 x(n)
C insert wonderful, efficient, debugged code here
      end
```

The FORTRAN 90 subroutine has basically the same signature as follows:

```
subroutine TriedAndTrue(x, n)
  integer (selected_int_kind(9))     :: n
  real (selected_real_kind(17, 308)) :: x(n)

  ! insert wonderful, efficient, debugged code here
end subroutine TriedAndTrue
```

Here is one way to wrap this method using SIDL. First of all, the SIDL method definition specifies that the array must be a 1-dimensional, column-major ordered array. This forces the incoming array to be a dense column.

```
    static void TriedAndTrue(inout array<double,1,column-major> arg);
```

Given that method definition in a class named Class and a package named Pkg, the implementation of the wrapper should look something like the following for FORTRAN 77:

```
        subroutine Pkg_Class_TriedAndTrue_fi(arg)
        implicit none
        integer*8 arg
C       DO-NOT-DELETE splicer.begin(Pkg.Class.TriedAndTrue)
        real*8 refarray(1)
        integer*4 lower(1), upper(1), stride(1), index
        integer n
        call SIDL_double__array_access_f(arg, refarray,
     $       lower, upper, stride, index)
        if (index .ne. 0) then
c we can assume stride(1) = 1 because of column-major specification
           n = 1 + upper(1) - lower(1)
           call TriedAndTrue(refarray(index), n)
        else
           write(*,*) 'ERROR: array alignment'
        endif
C       DO-NOT-DELETE splicer.end(Pkg.Class.TriedAndTrue)
        end
```

Similarly, it should look something like the following for FORTRAN 90, where the include statements are required at the top of the Impl file to ensure proper handling of subroutine names that have automatically been mangled by the Babel compiler:

```
#include "Pkg_Class_fAbbrev.h"
#include "SIDL_BaseClass_fAbbrev.h"
#include "SIDL_BaseInterface_fAbbrev.h"
! DO-NOT-DELETE splicer.begin(_miscellaneous_code_start)
#include "SIDL_double_fAbbrev.h"
! DO-NOT-DELETE splicer.end(_miscellaneous_code_start)
```

```
.
.
.

subroutine Pkg_Class_TriedAndTrue_mi(arg)
  ! DO-NOT-DELETE splicer.begin(Pkg.Class.TriedAndTrue.use)
  ! Insert use statements here...
  ! DO-NOT-DELETE splicer.end(Pkg.Class.TriedAndTrue.use)
  implicit none
  integer (selected_int_kind(18)) :: arg

! DO-NOT-DELETE splicer.begin(Pkg.Class.TriedAndTrue)
  real (selected_real_kind(17,308)) :: refarray(1)
  integer (selected_int_kind(8))    :: lower(1), upper(1), stride(1), index
  integer (selected_int_kind(8))    :: n
  call SIDL_double__array_access_m(arg, refarray, lower, upper, stride, index)
  if (index .ne. 0) then
    ! We can assume stride(1) = 1 because of column-major specification
    n = 1 + upper(1) - lower(1)
    call TriedAndTrue(refarray(index), n)
  else
    write(*,*) 'ERROR: array alignment'
  endif
! DO-NOT-DELETE splicer.end(Pkg.Class.TriedAndTrue)
end subroutine Pkg_Class_TriedAndTrue_mi
```

3.5.6 The C Macro API

For all the SIDL basic types except string, there is a C macro API for those who fear the function overhead of the C function API. When efficiency is not a concern, I recommend using the function API, but the C macro API is preferrable to the direct access to the data structure. The macro API is not available for arrays of strings, interfaces or objects because the issues associated with memory and object reference management.

The macro API is very similar to the function API; however, a single set of macros applies to all the supported array types. The macro names are independent of the type of array you're accessing.

```
SIDLArrayDim(array)
```

Return the dimension of array.

```
SIDLLower(array,ind)
```

Return the lower bound on dimension ind.

```
SIDLUpper(array,ind)
```

Return the upper bound on dimension ind.

```
SIDLStride(array,ind)
```

Return the stride for dimension ind. The stride is the offset between elements in a particular dimension. It can be positive or negative. It is in terms of number of value types (i.e., it's 1 means contiguous regardless of what data type).

```
SIDLArrayElem1(array, ind1)
SIDLArrayElem2(array, ind1, ind2)
SIDLArrayElem3(array, ind1, ind2, ind3)
SIDLArrayElem4(array, ind1, ind2, ind3, ind4)
```

Provide access to one, two, three and four dimensional array elements. This macro can appear on the left hand side of an assignment or on the right hand side in an expression. These macros blindly assume that the dimension and indices are correct.

3.5.7 The C Data Structure

If even the macro interface is not fast enough for you, you can access the internal data structure for all the basic types except string. You cannot access the internal data structure for arrays of strings, interfaces and objects.

The basic form of the C data structure for type XXXX is:

```
struct SIDL_XXXX__array {
  <value type for XXXX>   *d_firstElement;
  int32_t                 *d_lower;
  int32_t                 *d_upper;
  int32_t                 *d_stride;
  int32_t                  d_dimen;
  SIDL_bool                d_borrowed;
};
```

The string "<value type for XXXX>" should be replaced by something like `SIDL_bool`for an array of *bool*, `int32_t` for any array of *int*, `double` for an array of *double*, `int64_t` for an array of *long*, etc. (See Table 3.3)

- `d_dimen` tells the dimension of the multi-dimensional array. `d_lower`, `d_upper`, and `d_stride` each point to arrays of `d_dimen int32_t`'s. `d_lower[i]` provides the lower bound for the index in dimension `i`, and `d_upper[i]` provides the upper bound for the index in dimension `i`. Both the lower and upper bounds are valid index values; the upper bound is not one past the end.
- `d_borrowed` is true if the array does not managed the data that `d_firstElement` points too, and it is false otherwise. This mainly influences the behavior of the destructor.
  Clients should not modify `d_lower`, `d_upper`, `d_stride`, `d_dimen`, `d_borrowed` or (in the case of pointers) the values to which they point.
- `d_stride[i]` determines how elements are packed in dimension `i`. A value of 1 means that to get from element `j` to `j+1` in dimension `i`, you add one to the data pointer. Negative values for `d_stride` can be used to express a transposed matrix. The definition also allows either column or row major ordering for the data, and it also allows treating a subsection of an array as an array.

The data structure was inspired by the data structure used by Numeric Python; although, in Numeric Python, the stride is in terms of bytes. In SIDL, the stride is in terms of number of objects. One can convert to the Numeric Python view of things by multiplying the stride by the sizeof the value type.

3.6 Summary

Now that we've stepped through the general SIDL language, it is necessary to review the chapter and fill in some finer details. We discuss SIDL's reserved words in SubSection 3.6.1. Other words and constructs that are problematic in particular language bindings are discussed in SubSection 3.6.2.

3.6.1 Reserved Words

Table 3.4 lists all the words that are part of the SIDL grammar and cannot be used as a package, enum, interface, class, or argument name.

3.6.2 Suggested Things To Avoid

Since SIDL maps onto many other languages there are a great number of words and constructs that are harmless in SIDL, but cause great trouble in generated language bindings. We list known problems in Table 3.5.

Table 3.4: SIDL Reserved Words

reserved word	role	more info
abstract	optional modifier for class	
array	datatype	
bool	builtin datatype	
char	builtin datatype	
class	user defined datatype	
copy	(future) argument modifer	
dcomplex	builtin datatype	
double	builtin datatype	
enum	user defined datatype	
extends	inheritance mode	
fcomplex	builtin datatype	
final	method modifier	
float	builtin datatype	
implements	inheritance mode	
implements-all	inheritance mode	
import	bring other packages into current scope	
in	argument mode	
inout	argument mode	
int	builtin datatype	
interface	user defined datatype	
local	(future) method modifier	
long	builtin datatype	
oneway	(future) method modifier	
opaque	builtin datatype	
out	argument mode	
package	scoping construct	
static	method modifier	
string	builtin datatype	
throws	exception declaration	
version	assign version number to package	
void	declares method as not returning a type	

word	C	C++	Java	Python	FORTRAN	word	C	C++	Java	Python	FORTRAN
abstract			X			lambda				X	
and		X		X		long	X	X	X		
and_eq		X				mutable		X			
asm	X	X				namespace		X			
assert				X		native			X		
auto	X	X				new		X	X		
bitand		X				not		X		X	
bitor		X				not_eq		X			
bool		X				null			X		
boolean			X			operator		X			
break	X	X	X	X		or		X		X	
case	X	X	X			or_eq		X			
catch		X	X			package			X		
char	X	X	X			pass				X	
class		X	X			print				X	
compl		X				private		X	X		
const	X	X	X			protected		X	X		
const_cast		X				public		X	X		
continue	X	X	X	X		raise				X	
def				X		register	X	X			
default	X	X	X			reinterpret_cast		X			
del				X		return	X	X	X	X	
delete		X				short	X	X	X		
do	X	X	X			signed	X	X			
double	X	X	X			sizeof	X	X			
dynamic_cast		X				static	X	X	X		
elif				X		static_cast		X			
else	X	X	X	X		strictfp			X		
enum	X	X				struct	X	X			
except				X		super			X		
exec				X		switch	X	X	X		
explicit		X				synchronized			X		
export		X				template		X			
extends			X			this		X	X		
extern	X	X				throw		X	X		
false		X	X			throws			X		
final			X			transient			X		
finally			X	X		true		X	X		
fbat	X	X	X			try		X	X	X	
for	X	X	X	X		typedef	X	X			
friend		X				typeid		X			
from				X		typename		X			
global				X		union	X	X			
goto	X	X	X			unsigned	X	X			
if	X	X	X	X		using		X			
implements			X			virtual		X			
import			X			void	X	X	X		
inline		X				volatile	X	X	X		
instanceof			X			wchar_t		X			
int	X	X	X			while	X	X	X	X	
interface			X			xor		X			
is				X		xor_eq		X			

Table 3.5: Other words/constructs to avoid

- Reserved words in C/C++/Fortran/Java/Python. This is a long list and we'll probably compile it here sometime.
- Methods with same name as class (this is a constructor in C++).
- Packages, Classes, Interfaces, Methods or Arguments that differ only by case. Not all languages are case sensitive.

Chapter 4

Advanced Topics

In this section, we will discuss dynamic vs. static linkage, exception handling, and go deeper into how Babel works. At the end of this chapter, most developers will have enough information to be able to determine if a bug they experience is in their code or with Babel.

Contents

4.1 Dynamic vs. Static Linkage

Most UNIX users are very comfortable with statically linked libraries (e.g. `libXXX.a`). Most are aware of "shared object files" in UNIX (with the form `libXXX.so`) — but few actually build them. Even fewer still are familiar with dynamically linked libraries — called DLL's in Microsoft (after the common `.dll` suffix) which involve actually selecting and loading dynamic libraries (based on their string name) at runtime. This section serves as a quick overview of how Babel handles both static and dynamic libraries, including runtime loading.

4.1.1 Linkers and Position Independent Code (PIC)

In a static library, the linker simply copies needed compilation units from the library to the executable. The static library can subsequently be deleted with no adverse affects to the executable. This also causes common libraries to be duplicated in every executable that links against it, and for the resulting executables to be quite large.

In a shared library, the linker simply inserts in the executable enough information to find the library and load it when the executable is envoked. This typically happens before the program ever gets to `main()`. This keeps executables small and allows commonly used libraries to be reused without copying, but it also means that the executable can fail if the library is renamed, moved, deleted, or even if the user's environment changes sufficiently.

A necessary (but not sufficient) condition for shared libraries to work is that all the compilation units (`*.o`) contained must be explicitly compiled as *position independent code*(PIC). Position independent code has an added level of indirection in critical areas since details (such as addresses to jump to in subroutine calls) are not known until runtime. Even though shared libraries are very useful, PIC causes a small but measurable degradation in performance, making static linked libraries with non-PIC code a viable option for performance-critical situations.

A dynamic-linked library is a shared library with one added feature, it can be loaded explicitly by the user at runtime by passing the string name into `dlopen()`. Dynamic-linked libraries (DLL's) also require compilation as PIC, though many compilers (including GCC) have special commands for each[1].

4.1.2 Compiler Flags for Babel

Babel generated code depends critically on `babel_config.h` to correctly define a lot of platform specific details. One detail that changes too frequently to encode in `babel_config.h` is whether or not the software is being compiled s position independent code (PIC). This detail is commonly added to the compilation instruction using the flags (e.g. `-fPIC -DPIC`[2]). The first flag tells the compiler to generate position independent code. The second defines the preprocessor macro `PIC`. Looking now at `babel_config.h`, we see that either `SIDL_DYNAMIC_LIBRARY` or `SIDL_STATIC_LIBRARY` are defined depending on whether or not `PIC` is defined.

Babel tends to focus on static libraries and dynamic linked libraries; not worrying much about shared libraries. The main reason is that for every last drop of performance, people would want static libraries. To support Java and Python (and the CCA model) dynamic loading is required. There's no real benefit to doing shared libraries that can't be dynamically loaded, so in developing Babel, we focus on the other two linkage situations.

4.2 Exceptions

Insert section here.

---

[1] `-fpic` for SO's, `-fPIC` for DLL's

[2] The actual command to the compiler varies, `-fPIC` is understood by GCC

Chapter 5

SIDL

This section will discuss the SIDL grammar in exhaustive detail.

Contents

Insert chapter here

FIX ME!

Chapter 6

C++ Bindings

This chapter provides the minimum information necessary for people who are familiar with object-oriented/component oriented software development and the SIDL (Scientific Interface Definition Language) to implement classes in C++ or use classes implemented by someone else from a C++ driver.

Contents

Unlike C or FORTRAN 77, there is no runtime library created for a particular C++ compiler at installation. Instead, when you generate C++ from SIDL, you will find Stubs (aka proxy classes) generated for SIDL base classes and will have to compile and link them into your application.

That said, if you switch to a different compiler after installation, there may be some values set in `babel_config.h` that become invalid. This can be overcome by copying the header file, making the necessary changes, and placing the modified header file earlier in the include path than the original one.

6.1 SIDL C++ Header Suffix

The first thing that C++ users will notice is that C++ headers have a ".hh" suffix to distinguish them from C's ".h" suffix. This convention was born out of necessity to distinguish both differing header files and their include guards.

6.2 SIDL's Main C++ Header File

All C++ code generated by Babel `#include`'s a file called "`SIDL_cxx.hh`". This file includes `babel_config.h`, the C header file that defines configuration information. Finally, `SIDL_cxx.hh` defines some C++ classes in the SIDL namespace such as

- `SIDL::StubBase` [implementation detail] Common base class for all C++ stubs (proxy classes)
- template<T,U,V> SIDL::array_mixin [implementation detail] Common base class for all C++ array classes.

Table 6.1: SIDL to C++ Type Mappings

SIDL TYPE	C++ TYPE
int	int32_t
long	int64_t
float	float
double	double
bool	bool
char	char
string	std::string
fcomplex	SIDL::fcomplex
dcomplex	SIDL::dcomplex
enum	enum
opaque	SIDL::opaque
interface	class
class	class
array	SIDL::array (template specialization)

- typedefs for `SIDL::fcomplex`, `SIDL::dcomplex`, and `SIDL::opaque` (usually `std::complex`, `std::complex` and `void*`, respectively)
- `template<T> SIDL::array` Template array type for SIDL arrays.
- template specializations [implementation detail] specialization of arrays of all SIDL types are defined in this file.

6.3 Basic SIDL to C++ Type Mappings

The basic types in SIDL are mapped into C++ according to Table 6.1.

6.4 Calling Methods from C++

Since C++ is an object-oriented language, there is a lot less programmer overhead in using SIDL from the C++ perspective than from non-OO languages such as C or FORTRAN 77.

These proxy classes (we call ”stubs”) serve as the firewall between the application in C++ and Babel's internal workings. As one would expect, the proxy classes maintain minimal state so that, unlike C or FORTRAN 77, there is no special context argument added to non-static member functions.

Below are examples using standard classes. The first is an example of creating an object of the base class and its association to the base interface.

```
SIDL::BaseClass object = SIDL::BaseClass::_create();
SIDL::BaseInterface interface = object;
```

Here is an example call to the addSearchPath in the SIDL.Loader class:

```
std::string s("/try/looking/here");
SIDL::Loader::addSearchPath( s );
```

Adapted from the BABEL regression tests, the following is an example of a package called ExceptionTest that has a class named Fib with a method declared in SIDL as follows:

```
int getFib(in int n, in int max_depth, in int max_value, in int depth)
  throws NegativeValueException, FibException;
```

Table 6.2: SIDL Features Mapped onto C++

SIDL Feature	C++ Implementation
packages	C++ namespaces (no name transformations)
version numbers	ignored
interface	C++ class (called "stub", serves as a proxy to the implementation)
class	C++ class (called "stub", serves as a proxy to the implementation)
methods	C++ member functions; uses base method name when overloading; no name mangling; NOTE: Member functions beginning with a leading underscore "_" may be Babel internals, or specific to C++ binding.
static methods	Static C++ member functions; uses base method name when overloading; no name mangling; even works for dynamically loaded object's exceptions thrown and caught using C++ exception handling.
reference counting	SIDL C++ stubs can be treated as smart-pointers. Constructors, destructors, and operators are overloaded so that explicit calls to `addRef()` or `deleteRef()` are rarely needed.
casting	Assignment operators are overloaded to handle safe casting up and down the inheritance hierarchy. User should never call `dynamic_cast<>()` on a SIDL object since the stubs inheritance hierarchy does not follow the SIDL inheritance hierarchy. Attempted downcasts using assignment should be checked by a call to (`_is_nil()`, or `_not_nil()`).
instance creation	Use static member function "`_create`". The default constructor for a C++ stub creates the equivalent of a NULL pointer. Works only with non-abstract classes.

The corresponding C++ code fragment to use this method is:

```
ExceptionTest::Fib fib = ExceptionTest::Fib::_create();
try {
  int result = fib.getFib( 4, 100, 32000, 0 );
  cout << "Result of fib.getFib() = " << result << endl;
} catch ( ExceptionTest::NegativeValueException  e ) {
  // ...
} catch ( ExceptionTest::FibException e ) {
  // ...
}
```

Examples of calls to SIDL overloaded methods are based on the `overload_sample.sidl` file shown in Section 3.4.4. Recall that the file describes three versions of the `getValue` method. The first takes no arguments, the second takes an integer argument, and the third takes a boolean. Each is called in the code snippet below:

```
bool b1, bresult;
int  i1, iresult, nresult;

Overload::Sample t  = Overload::Sample::_create();

nresult = t.getValue();
bresult = t.getValue(b1);
iresult = t.getValue(i1);
```

To create the C++ stubs from a SIDL file, invoke BABEL as follows:

```
% babel --client=C++ file.sidl
```

```
% babel -cC++ file.sidl
```

This will create a babel.make file, some C headers and sources, and many C++ headers and sources. Files ending in ".c" or ".h" are in C, files ending in ".cc" or ".hh" are C++.

You will need to compile and link the files together to use the C++ stubs.

6.5 Implementing SIDL Classes in C++

Much of the information from the previous section is pertinent to implementing a SIDL class in C++. The types of the arguments are as indicated in Table 6.1. Your implementation can call other SIDL methods, in which case follow the rules for client calls.

To create the implementation, you must first have a valid SIDL file, then invoke Babel as follows:

```
% babel --server=C++ file.sidl
```

or use the shorter arguments

```
% babel -sC++ file.sidl
```

This will create a makefile fragment called babel.make, several C headers and source files, and numerous C++ header and source files. To create a working implementation, the only files that need to be hand-edited are the C++ "Impl" files (header and source files that end in `_Impl.hh` or `_Impl.cc`). All your additions to this file should be made between code splicer pairs. Code splicing is a technique Babel uses to preserve hand-edited code between multiple invocations of Babel. This allows a developer to refine their SIDL file without ruining all their previous implementations. Code between splicer pairs will be retained by subsequent invocations of BABEL; code outside splicer pairs is not.

Here is an example of a code splicer pair in C++. In this example, you would replace the line "// Insert code here... " with your implementation.

```
void MyPackage::MyClass::myMethod() {
    // DO-NOT-DELETE splicer.begin(MyPackage.MyClass.myMethod)
    // Insert code here...
    // DO-NOT-DELETE splicer.end(MyPackage.MyClass.myMethod)
}
```

It is important to understand where and why splicer blocks occur. Splicer blocks appear at the beginning and end of each Impl header and source file; for developers to add `#include`'s and other miscellaneous items respectively. In the headers, there is a splicer block that allows a user to make the impl class inherit from some other class. From SIDL's point of view this is private inheritance — meaning that it is useful for inheriting implementation details, but they can't be automatically exposed to the SIDL method dispatch mechanism. There is a splicer block inside the class definition for developers to add any data members the wish to the class. In the source files, splicer blocks appear in each method implementation. There are two implicit methods (i.e. methods that did not appear in the SIDL file) that must also be implemented. The _ctor method is a constructor function that is run whenever an object is created. The _dtor method is a destructor function that is run whenever an object is destroyed. If the object has no state, these functions are typically empty.

6.6 Accessing SIDL Arrays From C++

Although it is feasible to expose the underlying C array API to create, destroy and access array elements and meta-data, the C++ bindings provide a `SIDL::array<T>` template mechanism that is more in keeping with C++ idioms.

For SIDL built-in types, template specializations of `SIDL::array<T>` are defined in `SIDL_cxx.hh`. For SIDL interface and classes, the array template is again specialized in the corresponding stub header. The reason for the extensive use of template specialization is an effort to hide the detail that the array implementation is really templated on three terms: the type of the C struct that represents the array internally, the internal representation of each item in the array, and the C++ representation of each item in the array. (See `array_mixin` in `SIDL_cxx.hh` for grungy implementation details.)

An example is given below.

```
int32_t len = 10; // array length=10
int32_t dim = 1;  // one dimensional
int32_t lower[1] = {0}; // zero offset
int32_t upper[1] = {len-1};
int32_t prime = nextPrime(0);

// create a SIDL array of primes.
SIDL::array<int> a = SIDL::array<int>::createRow(dim, lower, upper);
for( int i=0; i<len; ++i ) {
   prime = nextPrime( prime );
   a.set(i, v);
}
```

Of course, the example above is only one way to create an array. The list of member functions for all C++ array classes is:

```
// constructors
array ( array_ior_t * src ); // internal
array () ;                   // empty

// destructor
~array() ;

// creation
static array<x>
createRow( int32_t dimen, const int32_t lower[],
           const int32_t upper[]);
static array<x>
createCol( int32_t dimen, const int32_t lower[],
           const int32_t upper[]);
static array<x>
create1d( int32_t len);
static array<x>
create2dCol( int32_t m, int32_t n);
static array<x>
create2dRow( int32_t m, int32_t n);
array<x>
slice( int32_t dimen, const int32_t numElem[],
       const int32_t *srcStart = 0,
       const int32_t *srcStride = 0,
       const int32_t *newStart = 0);

void borrow( item_ior_t * first_element, int32_t dimen,
        const int32_t lower[], const int32_t upper[],
```

```
            const int32_t stride[]);

    void addRef();
    void deleteRef();

    // get/set
    item_cxx_wrapper_t get(int32_t i);
    item_cxx_wrapper_t get(int32_t i1, int32_t i2);
    item_cxx_wrapper_t get(int32_t i1, int32_t i2, int32_t i3);
    item_cxx_wrapper_t get(int32_t i1, int32_t i2, int32_t i3, int32_t i4);
    item_cxx_wrapper_t get(const int32_t *indices);

    void set(int32_t i, item_cxx_wrapper_t elem);
    void set(int32_t i1, int32_t i2, item_cxx_wrapper_t elem);
    void set(int32_t i1, int32_t i2, int32_t i3,
     item_cxx_wrapper_t elem);
    void set(int32_t i1, int32_t i2, int32_t i3, int32_t i4,
     item_cxx_wrapper_t elem);
    void set(const int32_t *indices, item_cxx_wrapper_t elem);

    // other accessors
    int32_t dimen() const;

    int32_t lower( int32_t dim ) const;

    int32_t upper( int32_t dim ) const;

    int32_t stride( int32_t dim ) const;

    bool _is_nil() const;

    bool _not_nil() const;

    // get a const pointer to the actual array ior
    const array_ior_t* _get_ior() const { return d_array; }

    // get a non-const pointer to the actual array ior
    array_ior_t* _get_ior() { return d_array;}
```

where

- `array_ior_t` is the type of the C struct that represents the array internally,
- `item_ior_t` is the internal representation of each item in the array,
- `item_cxx_wrapper_t` is the C++ representation of each item in the array

Chapter 7

C Bindings

This chapter provides an introduction to the C bindings for SIDL. It includes some of the information needed to implement classes in C or use interfaces and objects from C.

Contents

Because BABEL uses C for its IOR, the C bindings are very similar to the IOR. All of the objects in the SIDL namespace (e.g. SIDL::BaseClass, SIDL::BaseException, etc.) are implemented in C, so clients can develop solely with a C compiler if necessary. Of course, the intent of BABEL is to interface many languages.

7.1 Header files

If you would like to use type `X.Y.Z` from C, you should `#include "X_Y_Z.h"`. If you would like to include the header files for a whole package `X.Y`, you can `#include "X_Y.h"`. For example, you can include all the types in the `SIDL` namespace with `#include "SIDL.h"`.

Each client side header file will ensure that `SIDL_header.h` is included. `SIDL_header.h` defines:

- `struct SIDL_dcomplex` for the SIDL dcomplex type with parts named `real` and `imaginary`.
- `struct SIDL_fcomplex` for the SIDL fcomplex type with parts named `real` and `imaginary`.
- `int32_t` and `int64_t` for the SIDL int and long types.
- a typedef for `SIDL_bool` for the SIDL bool type.
- and preprocessor symbols `TRUE` and `FALSE`.

It also includes the function prototypes for the multi-dimensional array APIs for the basic SIDL types. In general, clients don't need to worry about including `SIDL_header.h` because the BABEL generated header files will include it for you.

Table 7.1: SIDL to C Type Mappings

SIDL TYPE	C TYPE
int	int32_t
long	int64_t
float	float
double	double
bool	typedef SIDL_bool
char	char
string	char *
fcomplex	struct SIDL_fcomplex
dcomplex	struct SIDL_dcomplex
enum	enum
opaque	void *
interface	typedef
class	typedef
array	struct *

7.2 SIDL to C Type Mappings

The basic types in SIDL are mapped into C according to Table 7.1.

7.3 Mapping for classes, interfaces and arrays

Because C doesn't have builtin mechanisms for protecting the global namespace, the C mapping attempts to avoid namespace collisions by using struct and method names that incorporate all the naming information from the package, class and method names. For a type `Z` in package `X.Y`, the name of the type that C clients use for an object reference is `X_Y_Z`. `X_Y_Z` is defined as follows in the `X_Y_Z.h` header file:

```
struct X_Y_Z__object;
struct X_Y_Z__array;
typedef struct X_Y_Z__object* X_Y_Z;
```

This code fragment also shows that `struct X_Y_Z__array` is used for a multi-dimensional array of `X.Y.Z` objects. Here are some additional concrete examples of the object and interface reference types derived by the C mapping:

```
/**
 * Symbol "SIDL.BaseClass" (version 0.5.1)
 *
 * Every class implicitly inherits from <code>BaseClass</code>.  This
 * class implements the methods in <code>BaseInterface</code>.
 */
struct SIDL_BaseClass__object;
struct SIDL_BaseClass__array;
typedef struct SIDL_BaseClass__object* SIDL_BaseClass;

/**
 * Symbol "SIDL.BaseInterface" (version 0.5.1)
 *
 * Every interface in <code>SIDL</code> implicitly inherits
 * from <code>BaseInterface</code>, and it is implemented
 * by <code>BaseClass</code> below.
 */
```

```
struct SIDL_BaseInterface__object;
struct SIDL_BaseInterface__array;
typedef struct SIDL_BaseInterface__object* SIDL_BaseInterface;
```

7.4 Calling SIDL methods from C

The names of the C functions used to call SIDL methods are a concatenation of the package name, the class or interface name and the method name(s) with the period characters changed to underscores. If the method is specified as being overloaded (i.e., has a name extension) , the full method name is the concatenation of the short name and the extension. For non-static methods, the object or interface pointer is passed as the first parameter before any of the formal parameters. This parameter operates like an `in` parameter.

For methods that throw exceptions, there is an extra `out` argument of type `SIDL.BaseException` that holds the thrown exception in cases when an exception is thrown. When an exception is thrown, the caller should ignore the value of `out` parameters and the function's return value.

Here are the C bindings for the critical `addRef` and `deleteRef` methods from `SIDL.BaseInterface`. I mention these methods in particular because C clients must manage object reference counts themselves.

```
void
SIDL_BaseInterface_addRef(
  SIDL_BaseInterface self);

void
SIDL_BaseInterface_deleteRef(
  SIDL_BaseInterface self);
```

You can call these same methods from the `SIDL.BaseClass` bindings. In fact, every C binding for an interface or class will have entries for `addRef` and `deleteRef`.

```
void
SIDL_BaseClass_addRef(
  SIDL_BaseClass self);

void
SIDL_BaseClass_deleteRef(
  SIDL_BaseClass self);
```

The following SIDL method taken from the BABEL regression tests demonstrates how exceptions are handled.

```
int getFib(in int n, in int max_depth, in int max_value, in int depth)
  throws NegativeValueException, FibException;
```

Here is the C binding for this method:

```
int32_t
ExceptionTest_Fib_getFib(
  ExceptionTest_Fib self,
  int32_t n,
  int32_t max_depth,
  int32_t max_value,
  int32_t depth,
  SIDL_BaseException *_ex);
```

Here is an example of how to perform exception handling in C using a package of macros defined in `SIDL_Exception.h`.

```
#include "SIDL_Exception.h"
/* ...numerous lines deleted... */
  x = ExceptionTest_Fib_getFib(f, 10, 1, 100, 0, &_ex);
  if (SIDL_CATCH(_ex, "ExceptionTest.TooDeepException")) {
    traceback(_ex);
    SIDL_CLEAR(_ex);
  }
  else if (SIDL_CATCH(_ex, "ExceptionTest.TooBigException")) {
    traceback(_ex);
    SIDL_CLEAR(_ex);
  }
  else if (_ex == NULL) {
    return FALSE;
  }
  SIDL_CHECK(_ex);
  return TRUE;

  EXIT:;
    traceback(_ex);
    SIDL_CLEAR(_ex);
    return FALSE;
```

Examples of calls to SIDL overloaded methods are based on the `overload_sample.sidl` file shown in Section 3.4.4. Recall that the file describes three versions of the `getValue` method. The first takes no arguments, the second takes an integer argument, and the third takes a boolean. Each is called in the code snippet below:

```
int b1, i1, iresult, nresult;

Overload_Sample t  = Overload_Sample__create ();

nresult = Overload_Sample_getValue(t);
iresult = Overload_Sample_getValueInt(t, i1);
bresult = Overload_Sample_getValueBool(t, b1);
```

7.5 Implicitly defined methods

The C binding for interfaces and classes includes two methods for perform type casts. The methods are named `_cast` and `_cast2`. The leading underscore prevents these builtin methods from conflicting with a user method because user methods cannot begin with an underscore. Neither of these methods increases the reference count of the underlying object — this is contrary to standard methods that always return new reference counts. Here are the signatures for `_cast` and `_cast2` from `SIDL.BaseClass`.

```
SIDL_BaseClass
SIDL_BaseClass__cast(
  void* obj);

void*
SIDL_BaseClass__cast2(
  void* obj,
  const char* type);
```

The `_cast` method attempts to cast a SIDL interface or object pointer to a pointer to `SIDL.BaseClass`. The `_cast2` method attempts to cast a SIDL interface or object pointer to a pointer to an interface or object pointer of the type named `type`. In the case of `_cast2`, the client is responsible for casting the return value into the proper pointer type. Both methods are `NULL` safe. A `NULL` return value indicates that the cast failed or that `obj` was `NULL`.

Non-abstract classes have an additional implicit method called `_create` to create new instances of the class. Interfaces and abstract classes do not have this method because you cannot instantiate them. The `_create` method returns a new reference that the client must manage. Here is an example of its signature.

```
/**
 * Constructor function for the class.
 */
SIDL_BaseClass
SIDL_BaseClass__create(void);
```

7.6 Invoking BABEL to generate C bindings

To create C stubs (i.e. code to support C clients to a set of SIDL classes or interfaces), you should invoke BABEL as follows:

```
% babel --client=C file.sidl
```

or more cryptically

```
% babel -cC file.sidl
```

This will create more files than you can shake a stick at. The files ending in `_IOR.h` and `_IOR.c` are the Internal Object Representation. The files ending with `_Stub.c` are the C stubs — the interface between a C client and the IOR. The remaining header files have external C API that C clients may use.

To use the C stubs, you must compile the stub files whose file names end with `_Stub.c` and link them against the SIDL runtime library and a backend implementation.

7.7 Invoking BABEL to generate C implementations

To implement a set of SIDL classes in C, you should invoke BABEL as follows:

```
% babel --server=C file.sidl
```

or use the cryptic short form

```
% babel -sC file.sidl
```

Chapter 8

Python Bindings

Contents

8.1 How to Create a SIDL Object in Python

(once you've built the Python extension module)

You need to import the extension module and then calling a method to create an instance. If you have a class whose fully qualified name is x.y.z, you would say:

```
>>> import x.y.z
>>> obj = x.y.z.z()
```

The last part of the class name is repeated. You can also use **`from x.y.z import *`** if you prefer; although, you must guarantee that there are no namespace collisions.

In some cases, the Python extension module may be name `zmodule.so` instead of simply `z.so`. This might tempt you to say **`import x.y.zmodule`** instead of just **`import x.y.z`**; resist this temptation!

8.2 How to Cast SIDL Objects in Python

Let's say you have an object `obj`, and you would like to see if it is an instance of a SIDL class or interface whose fully qualified name is `x.y.z`. Here is how you do it.

```
>>> import x.y.z
>>> zobj = x.y.z.z(obj)
```

Of course, you don't need the import if you know that `x.y.z` has already been imported. If `zobj` is not equal to `None`, the cast was successful.

8.3 How to Call Methods from Python

Once you have created an object, you call methods on it using normal Python method calls. The arguments to the method only include the `in` and `inout` arguments, and the return value of the Python method includes the SIDL return value and the `inout` and `out` parameters. Hopefully, this will seem natural to Python programmers. For the following example, the object `obj` has a method `passeverywhere` with the following SIDL declaration:

```
double passeverywhere( in double d1, out double d2, inout double d3 );
```

You can see the Python calling signature with **`print obj.passeverywhere.__doc__`**. Here is what that shows for this example:

```
$ python
>>> import Args.Cdouble
>>> obj = Args.Cdouble.Cdouble()
>>> print obj.passeverywhere.__doc__
passeverywhere(in double d1,
               inout double d3)
RETURNS
   (double _return,
    out double d2,
    inout double d3)
```

In the method documentation, the SIDL method's return value is called `_return`; and unless the method is `void`, the return value always appears first. The fact that `_return` starts with an underbar should alert you to the fact that it is not a parameter because parameter names cannot start with an underbar. The document comments from the SIDL file (i.e. comments enclosed in `/** */` comments) appear below the BABEL generated signature documentation.

Static methods of a class are available in the Python `x.y.z` namespace assuming you use the **`import x.y.z`** command. Static methods have documentation just like class methods.

Examples of calls to SIDL overloaded methods are based on the `overload_sample.sidl` file shown in Section 3.4.4. Recall that the file describes three versions of the `getValue` method. The first takes no arguments, the second takes an integer argument, and the third takes a boolean. Each is called in the code snippet below:

```
b1 = 1
i1 = 1

t = Overload.Sample.Sample()

nresult = t.getValue()
iresult = t.getValueInt(i1)
bresult = t.getValueBool(b1)
```

8.4 Building Python Extension Modules

SIDL creates a `Setup.in` file that can be used to build the Python extension modules that you create. Copy `Makefile.pre.in` from your Python distribution into the directory containing `Setup.in`. There are three make variables you need to set when your building your Python extension module.

- `EXTRAFLAGS` — This is for the user to pass in additional compilation flags.

- `SIDLLIBDIR` — This should be a path (absolute or relative) to the directory where the SIDL runtime library file (i.e. the shared library/dynamic link library) resides.
- `SIDLPYHDRS` — This should be a path (absolute or relative) to the top directory in which the basic SIDL Python extensions are installed.
- `SIDLHDRS` This should be a path (absolute or relative) to the directory where the SIDL C header files are installed.

Here is a hypothetical example:

```
% make -f Makefile.pre.in SIDLLIBDIR=/usr/local/lib \
SIDLPYHDRS=/usr/local/include SIDLHDRS=/usr/local/include/SIDL
% make SIDLLIBDIR=/usr/local/lib/libsidl.so \
SIDLPYHDR=/usr/local/include SIDLHDRS=/usr/local/include/SIDL boot
```

It is unlikely that any installation actually uses those settings.

8.5 Setting up to Run Python

Here I assume that you've installed BABEL in directories rooted at `$PREFIX`. You need to have `$PREFIX/python` in your `PYTHONPATH` environment variable in addition to the directory where you are doing your work.

On many systems, you will need `$PREFIX/lib` in your `LD_LIBRARY_PATH` (or whatever system setting controls which directories are searched for shared libraries/dynamic link libraries).

You will likely need to use `SIDL_DLL_PATH` (a semicolon separated path) to provide the path to the directory that holds the shared library/dynamic link library containing the implementation of the SIDL objects.

8.6 Notes

The Python binding for SIDL long uses Python's unlimited precision integer data type, so it will not behave exactly like a 64 bit integer (i.e. there is no overflow). For Python versions before 2.2, your code needs to guarantee that a Python unlimited precision integer is used whenever a SIDL long is needed. For example, if you want to call a method whose SIDL signature is `bool isPrime(long num)`, calling `isPrime(1)` will fails; but calling `isPrime(1L)` will work fine.

The Python binding for an array of SIDL longs uses an array of 32 bit integers. This problem comes about because Numeric Python does provide an array of 64 bit integers.

What does this error message mean?

```
>>> import x.y.Zmodule
Traceback (innermost last):
File "<stdin>", line 1, in ?
ImportError:  dynamic module does not define init function (initZmodule)
```

Is the name of your SIDL interface/class `x.y.Z` or `x.y.Zmodule`, if it's the former, you should say **`import x.y.Z`**. If this isn't the problem, submit a bug report for BABEL. It might be informative to look at the symbol of the shared library/dynamic link library using a tool like nm. I suppose it's also worth checking the PYTHONPATH environment variable to make sure it's pointing to the right place.

```
>>> import x.y.Z
Fatal Python error:  Cannot load implementation for SIDL class x.y.Z
Abort (core dumped)
```

This means that the Python stub code (the code that links Python to SIDL's independent object representation (IOR)) failed in its attempt to load the shared library or dynamic link library containing the implementation of SIDL class `x.y.Z`. Make sure the environment variable `SIDL_DLL_PATH` lists all the directories where the shared libraries/dynamic link libraries for your SIDL objects/interfaces are stored. `SIDL_DLL_PATH` is a semicolon separated list of directories where SIDL client stubs will search for shared libraries required for SIDL classes and interfaces. Make sure the directory in which the SIDL runtime resides is in the `LD_LIBRARY_PATH` (or whatever your machine's mechanism for locating shared library files is).

```
>>> import x.y.Z
Fatal Python error:  Cannot load implementation for SIDL interface x.y.Z
Abort (core dumped)
```

This is the same problem for an interface as described immediately above for a class.

8.7 How to Implement SIDL Objects in Python

To build server side Python, you must have Python compiled as a shared library or dynamically link library. The standard Python build only builds the necessary shared library on a few platforms — none of which are target platforms for BABEL. Some Linux distributions include a Python shared library, and it is possible to make a Python shared library on Solaris. The Python shared library should contain the objects from libpythonx.y.a where x.y is your Python version. Making a shared library is different on each platform, so it is not covered here.

To implement an object in Python, first you must run BABEL to create the Python server side bindings.

```
% babel --server=python file.sidl
```

This creates the IOR, Python skeleton (pSkel), and Python launch (pLaunch) files in your currect directory, and it will create tree of subdirectories based on the package hierarchy found in file.sidl. The IOR, pSkel and pLaunch files must be compiled and place in a shared library (in most cases).

The tree of subdirectories created by BABEL includes Python implementation files whose name ends with `_Impl.py` and Python extension modules for the Python client side binding (`_Module.h` and `_Module.c`). The extension modules need to be compiled as above in section 8.4, and you need to fill in the implementations in the `_Impl.py` files.

BABEL generates the outline of the implementation. It creates a class definition and empty methods for you to fill in the each `_Impl.py` file. If you put your code between the comments as indicated, your code will be preserved if you rerun BABEL. Any changes out side the comment blocks will be lost if you rerun BABEL. Here is an example of a method implementation:

```
def passeverywhere(self, d1, d3):
  #
  # SIDL EXPECTED INCOMING TYPES
  # ============================
  # double d1
  # double d3
  #

  #
  # SIDL EXPECTED RETURN VALUE(s)
  # =============================
  # (_return, d2, d3)
  # double _return
  # double d2
  # double d3
  #
```

```
# DO-NOT-DELETE splicer.begin(passeverywhere)
if (d1 == 3.14):
  retval = 3.14
else:
  retval = 0
return (retval, 3.14, -d3)
# DO-NOT-DELETE splicer.end(passeverywhere)
```

BABEL generated everything except the code that appears between the `splicer.begin` and `splicer.end` comments.

Chapter 9

Java Bindings

Contents

Chapter 10

FORTRAN 77 Bindings

The intent of this chapter is to provide the minimum information necessary for people who are familiar with object-oriented/component oriented software development and the SIDL (Scientifi c Interface Defi nition Language) to implement classes in FORTRAN 77 or use classes implemented by someone else from a FORTRAN 77 driver.

Contents

10.1 Basic Types

For pointer types, such as opaque, interface, class, and array, a 64-bit integer is used, so FORTRAN 77 code will be portable between systems with a 32 bit address space and systems with a 64 bit address space. On a 32 bit system, the upper 32 bits of these quantities are ignored. Systems with more than 64-bit pointers aren't currently supported.

Generally, clients should treat opaque, interface, class and array values as black boxes. However, there is one value that is special. A value of zero for any of these quantities indicates that the value does not refer to an object. Zero is the

Table 10.1: SIDL to FORTRAN 77 type mapping

SIDL TYPE	FORTRAN 77 TYPE
int	INTEGER*4
long	INTEGER*8
float	REAL
double	DOUBLE PRECISION
bool	LOGICAL
char	CHARACTER*1
string	CHARACTER*(*)
fcomplex	COMPLEX
dcomplex	DOUBLE COMPLEX
enum	INTEGER
opaque	INTEGER*8

FORTRAN 77 equivalent of NULL . Any nonzero value is or should be a valid object reference. Developers writing in FORTRAN 77 should initialize values to be passed as in or inout parameters to zero or a valid object reference.

When mapping the SIDL string type into FORTRAN 77, some capability was sacrificed to make it possible to use normal looking FORTRAN 77 string handling. One difference is that all FORTRAN 77 strings have a limited fixed size. When implementing a subroutine with an out parameter, the size of the string is limited to 1024 characters.

Enumerated types are just integer values. The constants are defined in an includable file assuming your FORTRAN 77 compiler supports some form of including.

10.2 Calling Methods From FORTRAN 77

All SIDL methods are implemented as FORTRAN 77 subroutines regardless of whether they have a return value or not. For object methods, the object or interface pointer is passed as the first argument to the subroutine before all the formally declared arguments. The exception is static methods, where the object or interface pointer does not appear in the argument list at all.

When a method has a return value, a variable to hold the return value should be passed as an argument following the formally declared arguments. When a method can throw an exception (i.e., its SIDL definition has a throws clause), a variable of type `INTEGER*8` should be passed to hold a *`SIDL.BaseException`* pointer if an exception is thrown. The exception argument appears after the return value when both occur in a method. After the call, the client should test this argument. If it is non-zero, an exception was thrown by the method, and the method should respond appropriately. When an exception is thrown, the value of all other arguments is undefined.

The name of the subroutine that FORTRAN 77 clients should call is derived from the fully qualified name of the class and the name(s) of the method. If the method is specified as overloaded (i.e., has a name extension), the method's full name will be used. That is, the concatenation of the short name and the name extension will be used for a unique method name. Hence, to determine the subroutine name for FORTRAN 77, take the fully qualified name, replace all the periods with underscores, append an underscore, append the short method name, append the method name extension (if any) and then append "_f".

For example, to call the `deleteRef()` method on a *`SIDL.BaseInterface`* interface, you would write:

```
      integer*8 interface1, interface2
      logical areSame
C     code to initialize interface1 & interface 2 here
      call SIDL_BaseInterface_deleteRef_f(interface1)
```

To call the isSame method on a SIDL.BaseInterface , you would write:

```
      call SIDL_BaseInterface_queryInt_f(interface1, 'My.Interface.Name', interface2)
```

To call the queryInt method on a SIDL.BaseInterface, you would write:

```
      call SIDL_BaseInterface_queryInt_f(interface1, 'My.Interface.Name', interface2)
```

Examples of calls to SIDL overloaded methods are based on the `overload_sample.sidl` file shown in Section 3.4.4. Recall that the file describes three versions of the `getValue` method. The first takes no arguments, the second takes an integer argument, and the third takes a boolean. Each is called in the code snippet below:

```
      integer*8 t
      logical b1, bretval
      integer*4 i1, iretval

      call Overload_Sample__create_f (t)

      call Overload_Sample_getValue_f (t, iretval)
      call Overload_Sample_getValueInt_f (t, i1, iretval)
      call Overload_Sample_getValueBool_f (t, b1, bretval)
```

For interfaces and classes, there is an implicit method called `_cast()`. Its return type is opaque, and it has one formal argument, a string in addition to the implicit object/interface reference. The `_cast()` method attempts to cast the object/interface to the named type. It is similar to the queryInt method in *SIDL.BaseInterface* except it does not increment the reference count of the return object or interface, and it may return an object or an interface pointer. The `queryInt()` method always returns an interface pointer.

For non-abstract classes, there is an implicit method called `_create()`. It creates and returns an instance of the class.

Here are examples of the use of these two methods:

```
        integer*8 object, interface
        call SIDL_BaseClass__create_f(object)
        call SIDL_BaseClass__cast_f(object, 'SIDL.BaseInterface', interface)
```

Please note the presence of two underscores between BaseClass and create and between BaseClass and cast; the extra underscore is there because the first character of the method name is an underscore.

Here is an example call to the `addSearchPath()` in the *SIDL.Loader* class:

```
        call SIDL_Loader_addSearchPath_f('/try/looking/here')
```

Your FORTRAN 77 must manage any object references created by the calls you make.

Here is another example adapted from the BABEL regression tests. Package ExceptionTest has a class named Fib with a method declared in SIDL as follows:

```
int getFib(in int n, in int max_depth, in int max_value, in int depth)
    throws NegativeValueException, FibException;
```

Here is the outline of a FORTRAN 77 code fragment to use this method.

```
        integer*8 fib, except, except2
         integer*4 index, maxdepth, maxval, depth, result
         call ExceptionTest_Fib__create_f(fib)
         index = 4
         maxdepth = 100
         maxvalue = 32000
         depth = 0
         call ExceptionTest_getFib_f(fib, index, maxdepth,
       $       maxvalue, depth, result, except)
         if (except .ne. 0) then
           call SIDL_BaseException__cast_f(except,
       $       'ExceptionTest.FibException', except2)
           if (except2 .ne. 0) then
 c            do something here with the FibException
           else
             call SIDL_BaseException__cast_f(exception,
       $        'ExceptionTest.NegativeValueException',
       $         except2)
 c           do something here with the NegativeValueException
           endif
           call SIDL_BaseException_deleteRef_f(except)
         else
           write (*,*) 'getFib for ', index, ' returned ', result
         endif
         call ExceptionTest_Fib_deleteRef_f(fib)
```

Here is how you should invoke BABEL to create the FORTRAN 77 stubs for an IDL file.

```
% babel --client=f77 file.sidl
```


This will create a babel.make file, numerous C headers, numerous C source files, and some FORTRAN 77 files. The files ending in `_fStub.c` are the FORTRAN 77 stubs that allow FORTRAN 77 to call a SIDL method.

You will need to compile and link the files ending in `_fStub.c` into your application (i.e. `STUBSRCS` in `babel.make`). Normally, the IOR files (`_IOR.c`) are linked together with the implementation file, so you probably don't need to compile them.

If you have some *enum*'s defined in your SIDL file, BABEL will generate FORTRAN 77 include files in the style of DEC FORTRAN (Compaq FORTRAN? (now HP Fortran???)) `%INCLUDE`. These files are named by taking the fully qualified name of the *enum*, changing the periods to underscores, and appending `.inc`. Here is an example of a generated include file.

```
C       File:          enums_car.inc
C       Symbol:        enums.car-v1.0
C       Symbol Type:   enumeration
C       Babel Version: 0.5.0
C       Description:   Automatically generated; changes will be lost
C
C       babel-version = 0.5.0
C       source-line   = 25
C
        integer porsche
        parameter (porsche = 911)
        integer ford
        parameter (ford = 150)
        integer mercedes
parameter (mercedes = 550)
```

10.3 Implementing Classes in FORTRAN 77

Much of the information from the previous section is pertinent to implementing a SIDL class in FORTRAN 77. The types of the arguments are as indicated in Table 10.1. Your implementation can call other SIDL methods in which case follow the rules for client calls.

You should invoke BABEL:

```
% babel --server=f77 file.sidl
```

This will create a babel.make, numerous C headers, numerous C source files and some FORTRAN 77 source files. Your job is to fill in the FORTRAN 77 source files with the implementation of the methods. The files you need to edit all end with `_Impl.f`. All your changes to the file should be made between code splicer pairs. Code between splicer pairs will be retained by subsequent invocations of BABEL; code outside splicer pairs is not. Here is an example of a code splicer pair. In this example, you would replace the line "C Insert extra code here... " with your lines of code.

```
C       DO-NOT-DELETE splicer.begin(_miscellaneous_code_start)
C       Insert extra code here...
C       DO-NOT-DELETE splicer.end(_miscellaneous_code_start)
```

Each _Impl.f file contains numerous empty subroutines. Each subroutine that you must implement is partially implemented. The `SUBROUTINE` statement is written, and the types of the arguments have been declared. You must provide the body of each subroutine that implements the expected behavior of the method.

There are two implicit methods (i.e. methods that did not appear in the SIDL file) that must also be implemented. The `_ctor()` method is a constructor function that is run whenever an object is created. The `_dtor()` method is a destructor function that is run whenever an object is destroyed. If the object has no state, these functions are typically empty.

The SIDL IOR keeps a pointer (i.e. C void *) for each object that is intended to hold a pointer to the object's internal data. The FORTRAN 77 skeleton provides two functions that the FORTRAN 77 developer will need to use to access the private pointer. The name of the function is derived from the fully qualified type name as follows. Replace periods with underscores and append `__get_data_f` or `__set_data_f`. The first argument is the object pointer (i.e. self), and the second argument is an opaque . These arguments are 64 bit integers in FORTRAN 77, but the number of bits stored by the IOR is determined by the sizeof(void *).

BABEL/SIDL does not provide a mechanism for FORTRAN 77 to allocate memory to use for the private data pointer.

10.4 Accessing SIDL Arrays From FORTRAN 77

The normal SIDL C function API is available from FORTRAN 77 to create, destroy and access array elements and meta-data. The function name from FORTRAN has `_f` appended.

For SIDL types dcomplex, double, fcomplex , float, int and long, SIDL provides a method to get direct access to the array elements. For the other types, you must use the functional API to access array elements.

For type X, there is a FORTRAN 77 function called `SIDL_X__array_access_f` to provide a method to get direct access. An example is given below. Of course, this will not work if your FORTRAN 77 compiler does array bounds checking.

```
        integer*4 lower(1), upper(1), stride(1), i, index(1)
        integer*4 value, refindex, refarray(1), modval
        integer*8 nextprime, tmp
        lower(1) = 0
        value = 0
        upper(1) = len - 1
        call SIDL_int__array_create_f(1, lower, upper, retval)
        call SIDL_int__array_access_f(retval, refarray, lower,
     $       upper, stride, refindex)
        do i = 0, len - 1
           tmp = value
           value = nextprime(tmp)
           modval = mod(i, 3)
           if (modval .eq. 0) then
              call SIDL_int__array_set1_f(retval, i, value)
           else
              if (modval .eq. 1) then
                 index(1) = i
                 call SIDL_int__array_set_f(retval, index, value)
              else
C this is equivalent to the SIDL_int__array_set_f(retval, index, value)
                 refarray(refindex + stride(1)*(i - lower(1))) =
     $                value
              endif
           endif
        enddo
```

To access a two dimensional array, the expression referring to element i, j is

```
refarray(refindex + stride(1) * (i - lower(1)) + stride(2) * (j - lower(2))
```

To access a three dimensional array, the expression referring to element i, j k is

```
refarray(refindex + stride(1) * (i - lower(1)) + stride(2) * (j - lower(2))
```

You can call things like LINPACK or BLAS if you want, but you should check the stride to make sure the array is packed as you need it to be. `stride(i)` indicates the distance between elements in dimension `i`. A value of 1 means elements are packed densely in dimension `i`. Negative stride values are possible, and when an array is a slice of another array, there may be no dimension with a stride of 1.

For a *dcomplex* array, the reference array should a FORTRAN array of `REAL*8` instead of a FORTRAN array of double complex to avoid potential alignment problems. For a *fcomplex* array, the reference array is a `COMPLEX*8` because we don't anticipate an alignment problem in this case.

10.5 FORTRAN 77 objects with state

If you need to implement a FORTRAN 77 class with state, you can use SIDL arrays to store the state information. This is certainly not the only way to implement a FORTRAN 77 class with state, but it's one that will work wherever BABEL works. For example, if you have a class whose state requires three boolean variables and two double precision variables, your constructor might look something like the following:

```
      subroutine example_withState__ctor_fi(self)
      implicit none
      integer*8 self
C     DO-NOT-DELETE splicer.begin(example.withState._ctor)
      integer*8 statearray, logarray, dblarray
      call SIDL_opaque__array_create1d_f(2, statearray)
      call SIDL_bool__array_create1d_f(3, logarray)
      call SIDL_double__array_create1d_f(2, dblarray)
      if ((statearray .ne. 0) .and. (logarray .ne. 0) .and.
     $      (dblarray .ne. 0)) then
         call SIDL_opaque__array_set1_f(statearray, 0, logarray)
         call SIDL_opaque__array_set1_f(statearray, 1, dblarray)
      else
C        a real implementation would not leak memory like this one
         statearray = 0
      endif
      call example_withState__set_data_f(self, statearray)
C     DO-NOT-DELETE splicer.end(example.withState._ctor)
      end
```

Of course, it is up to your application make the association between elements of the arrays and particular state variables. For example, you could say that element 0 of the double array is the kinematic viscosity and element 1 could be the airspeed velocity of an unladden swallow. Element 0 of the boolean array could specify African (true) or European (false). The destructor for this class could look something like this:

```
      subroutine example_withState__dtor_fi(self)
      implicit none
      integer*8 self
C     DO-NOT-DELETE splicer.begin(example.withState._dtor)
      integer*8 statearray, logarray, dblarray
      call example_withState__get_data_f(self, statearray)
      if (statearray .ne. 0) then
         call SIDL_opaque__array_get1_f(statearray, 0, logarray)
         call SIDL_opaque__array_get1_f(statearray, 1, dblarray)
         call SIDL_bool__array_deleteRef_f(logarray)
         call SIDL_double__array_deleteRef_f(dblarray)
         call SIDL_opaque__array_deleteRef_f(statearray)
C     the following two lines are not strictly necessary
         statearray = 0
         call example_withState__set_data_f(self, statearray)
```

```
      endif
C       DO-NOT-DELETE splicer.end(example.withState._dtor)
        end
```

In this example, an accessor function for the airspeed velocity of an unladden swallow could be implemented as follows:

```
        subroutine example_withState_getAirspeedVelocity_fi(
     $     self, velocity)
        implicit none
        integer*8 self
        real*8 velocity
C       DO-NOT-DELETE splicer.begin(example.withState.getAirspeedVelocity)
        integer*8 statearray, dblarray
        call example_withState__get_data_f(self, statearray)
        if (statearray .ne. 0) then
           call SIDL_opaque__array_get1_f(statearray, 1, dblarray)
           call SIDL_double__array_get1_f(dblarray, 1, velocity)
        endif
C       DO-NOT-DELETE splicer.end(example.withState.getAirspeedVelocity)
        end
```

Chapter 11

FORTRAN 90 Bindings

The intent of this chapter is to provide the minimum information necessary for people who are familiar with object-oriented/component oriented software development and the SIDL (Scientific Interface Definition Language) to implement classes in FORTRAN 90 or use classes implemented by someone else from a FORTRAN 90 driver. For ease of use, it is patterned after the chapter on FORTRAN 77 bindings. Further, the initial support described below is very similar to that provided for FORTRAN 77.

Contents

11.1 Basic Types

For pointer types, such as opaque, interface, class, and array, the equivalent of a SIDL double is used. That is, the intermediate object reference assumes a 64-bit integer is used to enable portability between systems with a 32 bit

Table 11.1: SIDL to FORTRAN 90 type mapping

SIDL TYPE	FORTRAN 90 TYPE
int	INTEGER (SELECTED_INT_KIND(9))
long	INTEGER (SELECTED_INT_KIND(18))
float	REAL (SELECTED_REAL_KIND(6,37))
double	REAL (SELECTED_REAL_KIND(15, 307))
bool	LOGICAL
char	CHARACTER (LEN=1)
string	CHARACTER*(*)
fcomplex	COMPLEX (SELECTED_REAL_KIND(6, 37))
dcomplex	COMPLEX (SELECTED_REAL_KIND(15, 307))
enum	INTEGER (SELECTED_INT_KIND(9))
opaque	INTEGER (SELECTED_INT_KIND(18))

address space and those with a 64 bit address space. On a 32 bit system, the upper 32 bits of these quantities are ignored. Systems with more than 64-bit pointers aren't currently supported.

Generally, clients should treat opaque, interface, class and array values as black boxes. However, there is one value that is special. A value of zero for any of these quantities indicates that the value does not refer to an object. Zero is the equivalent of NULL. Any nonzero value is or should be a valid object reference. Developers writing in FORTRAN 90 should initialize values to be passed as in or inout parameters to zero or a valid object reference.

The SIDL string type mapping is currently identical to that of the FORTRAN 77 mapping. That is, all FORTRAN 90 strings have a limited fixed size. When implementing a subroutine with an out parameter, the size of the string is limited to 1024 characters.

Enumerated types are just integer values. The constants are defined in an includable file assuming your FORTRAN 90 compiler supports some form of including.

11.2 Calling Methods From FORTRAN 90

All SIDL methods are implemented as FORTRAN 90 subroutines regardless of whether they have a return value or not. For object methods, the object or interface pointer is passed as the first argument to the subroutine before all the formally declared arguments. The exception is static methods, where the object or interface pointer does not appear in the argument list at all.

When a method has a return value, a variable to hold the return value should be passed as an argument following the formally declared arguments. When a method can throw an exception (i.e., its SIDL definition has a throws clause), a variable of type `INTEGER (SELECTED_INT_KIND(18))` should be passed to hold a *`SIDL.BaseException`* pointer if an exception is thrown. The exception argument appears after the return value when both occur in a method. After the call, the client should test this argument. If it is non-zero, an exception was thrown by the method, and the method should respond appropriately. When an exception is thrown, the value of all other arguments is undefined.

The name of the subroutine that FORTRAN 90 clients should call is derived from the fully qualified name of the class and the name(s) of the method. If the method is specified as overloaded (i.e., has a name extension), the method's full name will be used. That is, the concatenation of the short name and the name extension will be used for a unique method name. Hence, to determine the subroutine name for FORTRAN 90, take the fully qualified name, replace all the periods with underscores, append an underscore, append the short method name, append the method name extension (if any) and then append "_m".

For example, to call the `deleteRef()` method on a *`SIDL.BaseInterface`* interface, you would write:

```
integer (selected_int_kind(18)) :: interface1, interface2
logical                         :: areSame
!
! code to initialize interface1 & interface 2 here
!
call SIDL_BaseInterface_deleteRef_m(interface1)
```

To call the isSame method on a *`SIDL.BaseInterface`*, you would write:

```
call SIDL_BaseInterface_queryInt_(interface1, 'My.Interface.Name', interface2)
```

To call the queryInt method on a *`SIDL.BaseInterface`*, you would write:

```
call SIDL_BaseInterface_queryInt_m(interface1, 'My.Interface.Name', interface2)
```

Examples of calls to SIDL overloaded methods are based on the `overload_sample.sidl` file shown in Section 3.4.4. Recall that the file describes three versions of the `getValue` method. The first takes no arguments, the second takes an integer argument, and the third takes a boolean. Each is called in the code snippet below:

```
integer (selected_int_kind(18)) :: t
logical                         :: b1, bretval
integer (selected_int_kind(9))  :: i1, iretval
```

```
call Overload_Sample__create_m (t)

call Overload_Sample_getValue_m (t, iretval)
call Overload_Sample_getValueInt_m (t, i1, iretval)
call Overload_Sample_getValueBool_m (t, b1, bretval)
```

For interfaces and classes, there is an implicit method called `_cast()`. Its return type is opaque, and it has one formal argument, a string in addition to the implicit object/interface reference. The `_cast()` method attempts to cast the object/interface to the named type. It is similar to the queryInt method in *SIDL.BaseInterface* except it does not increment the reference count of the return object or interface, and it may return an object or an interface pointer. The `queryInt()` method always returns an interface pointer.

For non-abstract classes, there is an implicit method called `_create()`. It creates and returns an instance of the class.

Here are examples of the use of these two methods:

```
integer (selected_int_kind(18)) :: object, interface
call SIDL_BaseClass__create_m(object)
call SIDL_BaseClass__cast_m(object, 'SIDL.BaseInterface', interface)
```

Please note the presence of two underscores between BaseClass and create and between BaseClass and cast; the extra underscore is there because the first character of the method name is an underscore.

Here is an example call to the `addSearchPath()` in the *SIDL.Loader* class:

```
call SIDL_Loader_addSearchPath_m('/try/looking/here')
```

Your FORTRAN 90 must manage any object references created by the calls you make.

Here is another example adapted from the BABEL regression tests. Package ExceptionTest has a class named Fib with a method declared in SIDL as follows:

```
int getFib(in int n, in int max_depth, in int max_value, in int depth)
    throws NegativeValueException, FibException;
```

Here is the outline of a FORTRAN 90 code fragment to use this method.

```
  integer (selected_int_kind(18)) :: fib, except, except2
  integer (selected_int_kind(9))  :: index, maxdepth, maxval, depth, result
  call ExceptionTest_Fib__create_f(fib)

  index    = 4
  maxdepth = 100
  maxvalue = 32000
  depth    = 0
  call ExceptionTest_getFib_m(fib, index, maxdepth, maxvalue, depth, result, &
                              except)
  if (except .ne. 0) then
    call SIDL_BaseException__cast_m(except, 'ExceptionTest.FibException', &
                                    except2)
    if (except2 .ne. 0) then
!      do something here with the FibException
    else
      call SIDL_BaseException__cast_m(exception, &
                                      'ExceptionTest.NegativeValueException', &
                                      except2)
!     do something here with the NegativeValueException
```

```
    endif
    call SIDL_BaseException_deleteRef_m(except)
  else
    write (*,*) 'getFib for ', index, ' returned ', result
  endif
  call ExceptionTest_Fib_deleteRef_m(fib)
```

Here is how you should invoke BABEL to create the FORTRAN 90 stubs for an IDL file.

```
% babel --client=f90 file.sidl
```

This will create a babel.make file, numerous C headers, numerous C source files, and some FORTRAN 90 files. The files ending in `_fStub.c` are the FORTRAN 90 stubs that allow FORTRAN 90 to call a SIDL method.

You will need to compile and link the files ending in `_fStub.c` into your application (i.e. `STUBSRCS` in `babel.make`). Normally, the IOR files (`_IOR.c`) are linked together with the implementation file, so you probably don't need to compile them.

If you have some *enum*'s defined in your SIDL file, BABEL will generate FORTRAN 90 include files in the style of DEC FORTRAN (Compaq FORTRAN? (now HP Fortran???)) `%INCLUDE`. These files are named by taking the fully qualified name of the *enum*, changing the periods to underscores, and appending `.inc` . Here is an example of a generated include file.

```
! File:          enums_car.inc
! Symbol:        enums.car-v1.0
! Symbol Type:   enumeration
! Babel Version: 0.8
! Description:   Automatically generated; changes will be lost
!
! babel-version = 0.8
! source-line   = 25
!
  integer (selected_int_kind(9)) :: porsche
  parameter (porsche = 911)
  integer (selected_int_kind(9)) :: ford
  parameter (ford = 150)
  integer (selected_int_kind(9)) :: mercedes
  parameter (mercedes = 550)
```

11.3 Implementing Classes in FORTRAN 90

Much of the information from the previous section is pertinent to implementing a SIDL class in FORTRAN 90. The types of the arguments are as indicated in Table 11.1. Your implementation can call other SIDL methods in which case follow the rules for client calls.

You should invoke BABEL:

```
% babel --server=f90 file.sidl
```

This will create a babel.make, numerous C headers, numerous C source files and some FORTRAN 90 source files. Your job is to fill in the FORTRAN 90 source files with the implementation of the methods. The files you need to edit all end with `_Impl.F90`. All your changes to the file should be made between code splicer pairs. Code between splicer pairs is retained by subsequent invocations of BABEL; code outside splicer pairs is not.

Here is an example of the standard code splicer pairs in generated FORTRAN 90 code. You would replace the comment "Insert extra code here... " associated with the "miscellaneous code start" splicer pair with code needed for your implementation such as additional abbreviation file(s) and any local, or private, subroutines. For the subroutine's "use" splicer pair, you would replace the "Insert use statements here..." comment with any use statements that are needed by the subroutine. Finally, you would add the implementation between the subroutine body's splicer pairs in the place of the "Insert the implementation here..." comment.

```
! DO-NOT-DELETE splicer.begin(_miscellaneous_code_start)
! Insert extra code here...
! DO-NOT-DELETE splicer.end(_miscellaneous_code_start)

.
.
.

subroutine Pkg_Class_name_mi(args)
  ! DO-NOT-DELETE splicer.begin(Pkg.Class.name.use)
  ! Insert use statements here...
  ! DO-NOT_DELETE splicer.end(Pkg.Class.name.use)
  implicit none
  integer (selected_int_kind(18)) :: arg

! DO-NOT-DELETE splicer.begin(Pkg.Class.name)
! Insert the implementation here...
! DO-NOT-DELETE splicer.end(Pkg.Class.name)
```

Each _Impl.F90 file contains numerous partially implemented subroutines. The `SUBROUTINE` and `END SUBROUTINE` statements have been generated and the types of the arguments declared. As mentioned above, you must provide any needed use statements and the body of each subroutine to implement the expected behavior of the method.

There are two implicit methods (i.e. methods that did not appear in the SIDL file) that must also be implemented if the object is to have state. The `_ctor()` method is a constructor function that is run whenever an object is created. The `_dtor()` method is a destructor function that is run whenever an object is destroyed. If there is not state then these functions are typically empty.

The SIDL IOR keeps a pointer (i.e. C `void *`) for each object that is intended to hold a pointer to the object's internal data. The FORTRAN 90 skeleton provides two functions that the FORTRAN 90 developer will need to use to access the private pointer. The name of the function is derived from the fully qualified type name by replacing periods with underscores and appending `__get_data_m` or `__set_data_m`. The first argument is the object pointer (i.e. self), and the second is an opaque. The number of bits used in FORTRAN 90 depends upon the size of `SELECTED_INT_KIND(18)`, while the number of bits stored by the IOR is determined by the sizeof(`void *`).

BABEL/SIDL does not provide a mechanism for FORTRAN 90 to allocate memory to use for the private data pointer.

11.4 Accessing SIDL Arrays From FORTRAN 90

The normal SIDL C function API is available from FORTRAN 90 to create, destroy, and access array elements and meta-data. The function name from FORTRAN 90 has `_m` appended.

For SIDL types dcomplex, double, fcomplex , float, int, and long, SIDL provides a method to get direct access to the array elements. For the other types, you must use the functional API to access array elements.

For type X, for example, there is a FORTRAN 90 function called `SIDL_X__array_access_m` to provide a method to get direct access. This is illustrated in the example below. Of course, this will not work if your FORTRAN 90 compiler does array bounds checking.

```
integer (selected_int_kind(9))  :: lower(1), upper(1), stride(1), i, index(1)
integer (selected_int_kind(9))  :: value, refindex, refarray(1), modval
integer (selected_int_kind(18)) :: nextprime, tmp
lower(1) = 0
```

```
value    = 0
upper(1) = len - 1
call SIDL_int__array_create_m(1, lower, upper, retval)
call SIDL_int__array_access_m(retval, refarray, lower, upper, stride, &
                              refindex)
do i = 0, len - 1
  tmp    = value
  value  = nextprime(tmp)
  modval = mod(i, 3)
  if (modval .eq. 0) then
    call SIDL_int__array_set1_m(retval, i, value)
  else
    if (modval .eq. 1) then
       index(1) = i
       call SIDL_int__array_set_m(retval, index, value)
    else
      !
      ! this is equivalent to the SIDL_int__array_set_m(retval, index, value)
      !
      refarray(refindex + stride(1)*(i - lower(1))) = value
    endif
  endif
enddo
```

To access a two dimensional array, the expression referring to element i, j is

```
refarray(refindex + stride(1) * (i - lower(1)) + stride(2) * (j - lower(2))
```

To access a three dimensional array, the expression referring to element i, j, k is

```
refarray(refindex + stride(1) * (i - lower(1)) + stride(2) * (j - lower(2))
```

You can call things like LINPACK or BLAS if you want, but you should check the stride to make sure the array is packed as needed. You can check `stride(i)`, which indicates the distance between elements in dimension `i`. A value of 1 means elements are packed densely in dimension `i`. Negative stride values are possible, and when an array is a slice of another array, there may be no dimension with a stride of 1.

For a *dcomplex* array, the reference array should be a FORTRAN array of `REAL (SELECTED_REAL_KIND(15, 307))` instead of a FORTRAN array of double complex to avoid potential alignment problems. For a *fcomplex* array, the reference array is a `COMPLEX (SELECTED_REAL_KIND(15, 307))` because we don't anticipate an alignment problem in this case.

11.5 FORTRAN 90 objects with state

If you need to implement a FORTRAN 90 class with state, you can use SIDL arrays to store the state information. This is certainly not the only way to implement a FORTRAN 90 class with state, but it's one that will work wherever BABEL works. For example, if you have a class whose state requires three boolean variables and two double precision variables, your constructor might look something like the following:

```
subroutine example_withState__ctor_mi(self)
  ! DO-NOT-DELETE splicer.begin(example.withState._ctor.use)
  ! Insert use statements here...
  ! DO-NOT-DELETE splicer.end(example.withState._ctor.use)
  implicit none
  integer (selected_int_kind(18)) :: self
```

```
  ! DO-NOT-DELETE splicer.begin(example.withState._ctor)
    integer*8 statearray, logarray, dblarray
    call SIDL_opaque__array_create1d_m(2, statearray)
    call SIDL_bool__array_create1d_m(3, logarray)
    call SIDL_double__array_create1d_m(2, dblarray)
    if ((statearray .ne. 0) .and. (logarray .ne. 0) .and. (dblarray .ne. 0)) then
      call SIDL_opaque__array_set1_m(statearray, 0, logarray)
      call SIDL_opaque__array_set1_m(statearray, 1, dblarray)
    else
      ! a real implementation would not leak memory like this one
      statearray = 0
    endif
    call example_withState__set_data_m(self, statearray)
  ! DO-NOT-DELETE splicer.end(example.withState._ctor)
  end subroutine example_withState__ctor_mi
```

Of course, it is up to your application to make the association between elements of the arrays and particular state variables. For example, you could say that element 0 of the double array is the kinematic viscosity and element 1 could be the airspeed velocity of an unladden swallow. Element 0 of the boolean array could specify African (true) or European (false). The destructor for this class could look something like this:

```
  subroutine example_withState__dtor_mi(self)
    ! DO-NOT-DELETE splicer.begin(example.withState._dtor.use)
    ! Insert use statements here...
    ! DO-NOT-DELETE splicer.end(example.withState._dtor.use)
    implicit none
    integer (selected_int_kind(18)) :: self

  ! DO-NOT-DELETE splicer.begin(example.withState._dtor)
    integer (selected_int_kind(18)) :: statearray, logarray, dblarray
    call example_withState__get_data_m(self, statearray)
    if (statearray .ne. 0) then
       call SIDL_opaque__array_get1_m(statearray, 0, logarray)
       call SIDL_opaque__array_get1_m(statearray, 1, dblarray)
       call SIDL_bool__array_deleteRef_m(logarray)
       call SIDL_double__array_deleteRef_m(dblarray)
       call SIDL_opaque__array_deleteRef_m(statearray)
       ! the following two lines are not strictly necessary
       statearray = 0
       call example_withState__set_data_m(self, statearray)
    endif
  ! DO-NOT-DELETE splicer.end(example.withState._dtor)
  end subroutine example_withState__dtor_mi
```

In this example, an accessor function for the airspeed velocity of an unladden swallow could be implemented as follows:

```
  subroutine example_withState_getAirspeedVelocity_mi(self, velocity)
    ! DO-NOT-DELETE splicer.begin(example.withState.getAirspeedVelocity.use)
    ! Insert use statements here...
    ! DO-NOT-DELETE splicer.end(example.withState.getAirspeedVelocity.use)
    implicit none
    integer (selected_int_kind(18)) :: self
    real (selected_real_kind(15, 307)) :: velocity

  ! DO-NOT-DELETE splicer.begin(example.withState.getAirspeedVelocity)
    integer (selected_int_kind(18)) :: statearray, dblarray
```

```
  call example_withState__get_data_m(self, statearray)
  if (statearray .ne. 0) then
     call SIDL_opaque__array_get1_m(statearray, 1, dblarray)
     call SIDL_double__array_get1_m(dblarray, 1, velocity)
  endif
! DO-NOT-DELETE splicer.end(example.withState.getAirspeedVelocity)
end subroutine example_withState_getAirspeedVelocity_mi
```

Chapter 12

SIDL Bindings

Contents

Chapter 13

SIDL Runtime Library

Contents

Chapter 14

Building and Deploying Software

Contents

Chapter 15

Troubleshooting

We've included some tips here, but perhaps the most useful is feedback from our users.

Contents

15.1 Compilation Consistency is Key

– Steve Smith, 24 September 2001 Basically "be consistent" is the biggest lesson we found.

When compiling C++ codes, you may have conflicts if you use different compile options. Under KCC we found -no_exceptions caused problems if parts were compiled with/without the flag. There are most likely other compile flags which turn features on/off which would cause similar problems. This caused a core dump immediately when core file was loaded. This is somewhat obvious but if you are linking together several different codes from a variety of developers you need to examine the compile flags very carefully. This problem is probably more likely with C++ due to the greater number of code generation options (e.g. RTTI, exceptions etc).

A much more subtle problem occurred when we had a C shared library which called functions in a C++ shared library. We initially used gcc to create the C shared library and KCC to create the C++ shared library. The application would core dump when a dynamic cast was attempted. This was solved by using the "cc" compiler wrapper that is part of the KCC distribution (which uses the native "cc"). So you need to be aware of not only what is in your .so and how it is compiled but all the .so's that you are using.

If you have several versions of a library, say during a debugging process, make sure you are using the correct versions of things. The "ldd" command is very useful for making sure you getting the shared libraries that you think you should be linking to. Along these lines, keep your LD_LIBRARY_PATH as simple as possible when debugging.

In retrospect this does not look like a large number of problems, but figuring out the second problem took a long time since I focused on how the C++ library was being created rather than where the real problem was being introduced. It wasn't until after I had exhausted a long list of other potential conflicts that I started messing with the C library compilation.

Bibliography

[1] Babel homepage. `http://www.llnl.gov/CASC/components/babel.html`.

[2] CCAFE homepage. `http://www.cca-forum.org/~baallan/ccafe`.

[3] Common Component Architecture (CCA) Forum homepage. `http://www.cca-forum.org`.

[4] Tammy Dahlgren, Tom Epperly, Scott Kohn, and Gary Kumfert. *Babel User's Guide*. CASC, Lawrence Livermore National Laboratory, version 0.7.0 edition, May 2002.

[5] Guy Eddon and Henry Eddon. *Inside Distributed COM*. Microsoft Press, Redmond, WA, 1998.

[6] Eric Eide, Jay Lepreau, and James L. Simister. Flexible and optimized IDL compilation for distributed applications. In *Proceedings of the Fourth Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers*, 1998.

[7] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*, July 1996. Available at `http://java.sun.com`.

[8] Michi Hennig and Steve Vinoski. *Advanced CORBA Programming with C++*. Professional Computing. Addison-Wesley, 1999.

[9] International Organization for Standardization, Geneva. *ISO/IEC 14882 Standard for the C++ Programming Language*, 1998.

[10] Bill Janssen, Mike Spreitzer, Dan Larner, and Chris Jacobi. *ILU Reference Manual*. Xerox Corporation, November 1997. Available at `ftp://ftp.parc.xerox.com/pub/ilu/ilu.html`.

[11] Scott Meyers. *More Effective C++: 35 New Ways to Improve your Programs and Designs*. Professional Computing. Addision-Wesley, 1996.

[12] Scott Meyers. *Effective C++: 50 Specific Ways to Improve your Programs and Designs*. Professional Computing. Addision-Wesley, 2 edition, 1998.

[13] Microsoft Corporation. *Component Object Model Specification (Version 0.9)*, October 1995. See `http://www.microsoft.com/oledev/olecom/title.html`.

[14] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, February 1998. Available at `http://www.omg.org/corba`.

[15] SciDAC: Scientific Discovery through Advanced Computing. `http://www.science.doe.gov/scidac`.

[16] SCIRun homepage. `http://www.sci.utah.edu`.

[17] John Shirley, Wei Hu, and David Magid. *Guide to Writing DCE Applications*. O'Reilly & Associates, Inc., Sebastopol, CA, 1994.

[18] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 3 edition, 1997.

[19] U. S. Department of Energy (DOE) homepage. `http://www.energy.gov`.

[20] Norm Walsh. *DocBook*. O'Reilly, 2000.

[21] XCAT homepage. `http://www.extreme.indiana.edu/xcat`.