Using ILU with ANSI C: A Tutorial

Bill Janssen <janssen@parc.xerox.com>

Formatted 7 May 1996.

Copyright © 1995 Xerox Corporation
All Rights Reserved.

0.1 Introduction

This document is a tutorial on how to use the ILU system with the programming language ANSI C, both as a way of developing software libraries, and as a way of building distributed systems. In an extended example, we'll build an ILU module that implements a simple four-function calculator, capable of addition, subtraction, multiplication, and division. It will signal an error if the user attempts to divide by zero. The example demonstrates how to specify the interface for the module; how to implement the module in ANSI C; how to use that implementation as a simple library; how to provide the module as a remote service; how to write a client of that remote service; and how to use subtyping to extend an object type and provide different versions of a module.

Each of the programs and files referenced in this tutorial is available as a complete program in a separate appendix to this document; parts of programs are quoted in the text of the tutorial.

0.2 Specifying the Interface

Our first task is to specify more exactly what it is we're trying to provide. A typical four-function calculator lets a user enter a value, then press an operation key, either +, -, /, or *, then enter another number, then press = to actually have the operation happen. There's usually a CLEAR button to press to reset the state of the calculator. We want to provide something like that.

We'll recast this a bit more formally as the *interface* of our module; that is, the way the module will appear to clients of its functionality. The interface typically describes a number of function calls which can be made into the module, listing their arguments and return types, and describing their effects. ILU uses *object-oriented* interfaces, in which the functions in the interface are grouped into sets, each of which applies to an *object type*. These functions are called *methods*.

For example, we can think of the calculator as an object type, with several methods: Add, Subtract, Multiply, Divide, Clear, etc. ILU provides a standard notation to write this down with, called *ISL* (which stands for "Interface Specification Language"). ISL is a declarative language which can be processed by computer programs. It allows you to define object types (with methods), other non-object types, exceptions, and constants.

The interface for our calculator would be written in ISL as:

```
INTERFACE Tutorial;

EXCEPTION DivideByZero;

TYPE Calculator = OBJECT
  METHODS
    SetValue (v : REAL),
    GetValue () : REAL,
    Add (v : REAL),
    Subtract (v : REAL),
    Multiply (v : REAL),
    Divide (v : REAL) RAISES DivideByZero END
  END;
```

This defines an interface `Tutorial`, an exception `DivideByZero`, and an object type `Calculator`. Let's consider these one by one.

The interface, `Tutorial`, is a way of grouping a number of type and exception definitions. This is important to prevent collisions between names defined by one group and names defined by another group. For example, suppose two different people had defined two different object types, with different methods, but both called `Calculator`! It would be impossible to tell which calculator was meant. By defining the `Calculator` object type within the scope of the `Tutorial` interface, this confusion can be avoided.

The exception, `DivideByZero`, is a formal name for a particular kind of error, division by zero. Exceptions in ILU can specify an *exception-value type*, as well, which means that real errors of that kind have a value of the exception-value type associated with them. This allows the error to contain useful information about why it might have come about. However, `DivideByZero` is a simple exception, and has no exception-value type defined. We should note that the full name of this exception is `Tutorial.DivideByZero`, but for this tutorial we'll simply call our exceptions and types by their short name.

The object type, `Calculator` (again, really `Tutorial.Calculator`), is a set of six methods. Two of those methods, `SetValue` and `GetValue`, allow us to enter a number into the calculator object, and "read" the number. Note that `SetValue` takes a single argument, *v*, of type `REAL`. `REAL` is a built-in ISL type, denoting a 64-bit floating point number. Built-in ISL types are things like `INTEGER` (32-bit signed integer), `BYTE` (8-bit unsigned byte), and `CHARACTER` (16-bit Unicode character). Other more complicated types are built up from these simple types using ISL *type constructors*, such as `SEQUENCE OF`, `RECORD`, or `ARRAY OF`.

Note also that `SetValue` does not return a value, and neither do `Add`, `Subtract`, `Multiply`, or `Divide`. Rather, when you want to see what the current value of the calculator is, you must call `GetValue`, a method which has no arguments, but which returns a `REAL` value, which is the value of the calculator object. This is an arbitrary decision on our part; we could have written the interface differently, say as

```
TYPE NotOurCalculator = OBJECT
  METHODS
    SetValue () : REAL,
    Add (v : REAL) : REAL,
    Subtract (v : REAL) : REAL,
    Multiply (v : REAL) : REAL,
    Divide (v : REAL) : REAL RAISES DivideByZero END
  END;
```

– but we didn't.

Our list of methods on `Calculator` is bracketed by the two keywords `METHODS` and `END`, and the elements are separated from each other by commas. This is pretty standard in ISL: elements of a list are separated by commas; the keyword `END` is used when an explicit list-end marker is needed (but not when it's not necessary, as in the list of arguments to a method); the list often begins with some keyword, like `METHODS`. The *raises clause* (the list of exceptions which a method might raise) of the method `Divide` provides another example of a list, this time with only one member, introduced by the keyword `RAISES`.

Another standard feature of ISL is separating a name, like `v`, from a type, like `REAL`, with a colon character. For example, constants are defined with syntax like

```
CONSTANT Zero : INTEGER = 0;
```

Definitions, of interface, types, constants, and exceptions, are terminated with a semicolon.

We should expand our interface a bit by adding more documentation on what our methods actually do. We can do this with the *docstring* feature of ISL, which allows the user to add arbitrary text to object type definitions and method definitions. Using this, we can write

```
INTERFACE Tutorial;

EXCEPTION DivideByZero
  "this error is signalled if the client of the Calculator calls
the Divide method with a value of 0";

TYPE Calculator = OBJECT
  COLLECTIBLE
  DOCUMENTATION "4-function calculator"
  METHODS
    SetValue (v : REAL) "Set the value of the calculator to `v'",
    GetValue () : REAL  "Return the value of the calculator",
    Add (v : REAL)      "Adds `v' to the calculator's value",
    Subtract (v : REAL) "Subtracts `v' from the calculator's value",
    Multiply (v : REAL) "Multiplies the calculator's value by `v'",
    Divide (v : REAL) RAISES DivideByZero END
      "Divides the calculator's value by `v'"
  END;
```

Note that we can use the `DOCUMENTATION` keyword on object types to add documentation about the object type, and can simply add documentation strings to the end of exception and method

definitions. Documentation strings cannot currently be used for non-object types. We also use the `COLLECTIBLE` keyword to mark the object type as participating in a distributed 'garbage-collection' protocol; this is not discussed in this tutorial, but is covered in the Python tutorial.

ILU provides a program, `islscan`, which can be used to check the syntax of an ISL specification. `islscan` parses the specification and summarizes it to standard output:

```
% islscan Tutorial.isl
Interface "Tutorial", imports "ilu"
  {defined on line 1
   of file /tmp/tutorial/Tutorial.isl (Fri Jan 27 09:41:12 1995)}

Types:
  real                        {<built-in>, referenced on 10 11 12 13 14 15}

Classes:
  Calculator                  {defined on line 17}
    methods:
      SetValue (v : real);                              {defined 10, id 1}
        "Set the value of the calculator to 'v'"
      GetValue () : real;                               {defined 11, id 2}
        "Return the value of the calculator"
      Add (v : real);                                   {defined 12, id 3}
        "Adds 'v' to the calculator's value"
      Subtract (v : real);                              {defined 13, id 4}
        "Subtracts 'v' from the calculator's value"
      Multiply (v : real);                              {defined 14, id 5}
        "Multiplies the calculator's value by 'v'"
      Divide (v : real) {DivideByZero};                 {defined 16, id 6}
        "Divides the calculator's value by 'v'"
    documentation:
      "4-function calculator"
    unique id:  ilu:cigqcW09P1FF98gYV0hf5XxGf15

Exceptions:
  DivideByZero                {defined on line 5, refs 15}
%
```

`islscan` simply lists the types defined in the interface, separating out object types (which it calls "classes"), the exceptions, and the constants. Note that for the `Calculator` object type, it also lists something called its *unique id*. This is a 160-bit number (expressed in base 64) that ILU assigns automatically to every type, as a way of distinguishing them. While it might interesting to know that it exists (:-), the ILU user never has know what it is; `islscan` supplies it for the convenience of the ILU implementors, who sometimes do have to know it.

0.3 Implementing the True Module

After we've defined an interface, we then need to supply an implementation of our module. Implementations can be done in any language supported by ILU. Which language you choose often depends on what sort of operations have to be performed in implementing the specific functions of the module. Different languages have specific advantages and disadvantages in different areas. Another consideration is whether you wish to use the implementation mainly as a library, in which case it should probably be done in the same language as the rest of your applications, or mainly as a remote service, in which case the specific implementation language is less important.

We'll demonstrate an implementation of the `Calculator` object type in ANSI C, which is perhaps the most primitive of all the ILU-supported languages. This is just a matter of writing 6 C functions, corresponding to the 6 methods defined on the `Tutorial.Calculator` type. Before we do that, though, we'll explain how the names and signatures of the C functions are arrived at.

0.3.1 What the Interface Looks Like in ANSI C

For every programming language supported by ILU, there is a standard *mapping* defined from ISL to that programming language. This mapping defines what ISL type names, exception names, method names, and so on look like in that programming language.

The mapping for ANSI C is straightforward. For type names, such as `Tutorial.Calculator`, the C name of the ISL type `Interface.Name` is `Interface_Name`. That is, the period is replaced with an underscore. So the name of our `Calculator` type in C would be `Tutorial_Calculator`, which is really just a `typedef` for the standard ILU-C type `ILU_C_Object`, the type which all ILU object types have in C.

The C mapping for a method name such as `SetValue`, is just the method name appended to the C name of the of the object type: `Tutorial_Calculator_SetValue`. The return type of this C function is whatever is specified in the ISL specification for the method, or `void` if no type is specified. The arguments for the C are the same as specified in the ISL; their types are the C types corresponding to the ISL types, *except* that two extra arguments are added to each C version of an ISL method. The first extra argument is added at the beginning of the parameter list; it supports the object-oriented paradigm used in ILU, and is an *instance* of the object type on which the method is defined. An instance is simply a value of that type. The second extra argument is added at the end of the parameter list; it is a value of type `CORBA_Environment *`, which is used to pass meta-information such as the identity of the caller into the function, and to pass information about

exception conditions back to the caller. Thus the C method corresponding to our ISL `SetValue` would have the prototype signature

```
   void
Tutorial_Calculator_SetValue (
   Tutorial_Calculator,
   CORBA_double v,
   CORBA_Environment *
);
```

Note that the ISL type `REAL` is mapped to the C type `CORBA_double`, which is just another name for `double`.

Similarly, the signatures for the other methods, in C, are

```
   CORBA_double
Tutorial_Calculator_GetValue (
   Tutorial_Calculator,
   CORBA_Environment *
);

   void
Tutorial_Calculator_Add (
   Tutorial_Calculator,
   CORBA_double v,
   CORBA_Environment *
);

   void
Tutorial_Calculator_Subtract (
   Tutorial_Calculator,
   CORBA_double v,
   CORBA_Environment *
);

   void
Tutorial_Calculator_Multiply (
   Tutorial_Calculator,
   CORBA_double v,
   CORBA_Environment *
);

   void
Tutorial_Calculator_Divide (
   Tutorial_Calculator,
```

```
    CORBA_double v,
    CORBA_Environment *
);
```

Note that even though the `Divide` method can raise an exception, the signature looks like those of the other methods. This is because the `CORBA_Environment *` parameter is used, in C, to signal exceptions back to the caller. Exceptions are represented in C by a value of the standard ILU-C type `ILU_C_ExceptionCode`. The mapping of exception names is similar to the mapping used for types, except that exception names are prefixed with the characters `"ex_"`. So the exception `Tutorial.DivideByZero` would have the name `ex_Tutorial_DivideByZero`, in C.

There is one further refinement of the C mapping we have to know before we can proceed with the implementation. Object systems in programming languages typically differentiate between two different kinds of procedure calls, sometimes called *generic function*s and *method*s. The generic function represents the abstract form of the procedure call, the one that a user of a routine would call. However, in an object system, invocation of a generic function might cause one of several different actual subroutines to be called, because each object type might implement the generic function separately. These implementations are called methods. This means that in a language like C, there will at least two names used for each ILU procedure, the name by which a user of the procedure calls it (the name of the generic function), and the name by which the implementor of the procedure defines it (the name of the method). The C procedure names we've been using so far has been the generic function names. The method name is the same as the generic function name, but with the prefix `server_`, as in `server_`*Interface*`_`*Type*`_`*Method*. ILU generates code that matches an invocation of a function named with its generic function name to execution of the function named with the true name, or method name. Since we are making an implementation of the `Tutorial` module, we'll use the method names for each of the functions we write. Just to confuse things a bit more, I should mention that in the ILU world, we call the generic function a *surrogate method*, and the method a *true method*!

One way to see what all the C names for an interface look like is to run the program `c-stubber`. This program reads an ISL file, and generates the necessary C code to support that interface in C. One of the files generated is '*Interface*`.h`', which contains the definitions of all the C types for that interface, along with prototypes for both the generic functions and methods.

```
% c-stubber Tutorial.isl
header file interface Tutorial to ./Tutorial.h...
common code for interface Tutorial to ./Tutorial-common.c...
code for surrogate stubs of interface Tutorial to ./Tutorial-surrogate.c...
code for true stubs of interface Tutorial to ./Tutorial-true.c...
%
```

0.3.2 A Faulty Implementation

Let's consider a simple implementation of our six true methods:

```
/* faulty-impl.c */
[ The first thing we need to do is to include the generated header
  file, which describes the types and methods used by the Tutorial
  interface. ]

#include <Tutorial.h>

[ We'll then define a static variable of type "CORBA_double" to hold the
  value of the calculator object, and call it "the_Value". ]

static CORBA_double the_Value = 0.0;

[ Now to implement the method, we simply take the true prototype
  and add whatever code is necessary to actually perform the operation. ]

  void
server_Tutorial_Calculator_SetValue (
  Tutorial_Calculator self,
  CORBA_double v,
  CORBA_Environment *env)
{
  the_Value = v;
}

   CORBA_double
server_Tutorial_Calculator_GetValue (
   Tutorial_Calculator self,
   CORBA_Environment *env)
{
  return (the_Value);
}

  void
server_Tutorial_Calculator_Add (
  Tutorial_Calculator self,
  CORBA_double v,
  CORBA_Environment *env)
{
  the_Value += v;
}
```

```
  void
server_Tutorial_Calculator_Subtract (
  Tutorial_Calculator self,
  CORBA_double v,
  CORBA_Environment *env)
{
  the_Value -= v;
}

  void
server_Tutorial_Calculator_Multiply (
  Tutorial_Calculator self,
  CORBA_double v,
  CORBA_Environment *env)
{
  the_Value *= v;
}
```

[The Divide method gets a little trickier. We have to compare the value "v" to zero, which for floating point values actually means comparing it to some epsilon to see whether it is less than that epsilon, and then if it is "zero" we need to signal an error, by "raising" the `DivideByZero` *exception. The way of raising exceptions in ILU C is rather clumsy, so we'll define a macro to make it look prettier. We also define some macros to make testing the value of "v" a bit prettier.]*

```
#define ABS(x)  (((x)<0)?(-(x)):(x))
#define SOME_EPSILON    0.000000001     /* zero, practically speaking */

#define RAISE(env,exception) { (env)->returnCode=(exception);\
                               (env)->_major=CORBA_USER_EXCEPTION; }

  void
server_Tutorial_Calculator_Divide (
  Tutorial_Calculator self,
  CORBA_double v,
  CORBA_Environment *env)
{
  if (ABS(v) < SOME_EPSILON)
    RAISE(env, ex_Tutorial_DivideByZero)
  else
    the_Value /= v;
}
```

The problem with this implementation is that all instances of the `Calculator` type share the same value. This doesn't seem to mirror the way real 4-function calculators work. They have

individual values that can differ for different calculators, instead of one shared value. We need to provide a way for each calculator object to have its own state, its own value.

0.3.3 Fixing the Implementation

To provide instance-specific state for each instance of the calculator type, we can begin by observing that the calculator object is provided to each true method as the *self* parameter. We could thus perform the binding of some local state to each instance with a hash table, which would map values of `Tutorial_Calculator` to values of `CORBA_double` (or more probably, to values of `CORBA_double *`), so that each calculator object would be manipulating its own separate value.

A simpler way, however, is to use the *user data field* of each `ILU_C_Object`, a slot of type `void *` which is reserved for use by the true implementation of the object type. This field can be initialized by the user with a value when a true instance of an object type is created. We'll specify that the user data field of instances of the type `Tutorial_Calculator` contain a pointer to a `malloc`'ed `CORBA_double` value. We can help to enforce that constraint by adding a utility function to our code, that creates an instance of `Tutorial_Calculator` by `malloc`'ing a value of `CORBA_double`, then calls the automatically generated function `Tutorial_Calculator__CreateTrue` to actually create the instance, passing the pointer to the `malloc`'ed value as an argument.

```
[ We define a function which creates a new instance of a Calculator object. ]

  Tutorial_Calculator
Create_Tutorial_Calculator ()
{
  CORBA_double *the_value = (CORBA_double *) malloc(sizeof(CORBA_double));

  *the_value = 0.0;     /* zero out our value */

[ The function Tutorial_Calculator__CreateTrue is automatically
  generated into the file "Tutorial-true.c" by the c-stubber.
  It takes three arguments, INSTANCE-HANDLE, SERVER, and
  USER-DATA-FIELD, and returns a new instance of Tutorial_Calculator.
  We don't care about what the INSTANCE-HANDLE and SERVER of
  Calculator instances are, so we'll pass ILU_NIL (which is another name
  for NULL) for the first two arguments, which will cause ILU
  to choose reasonable default values for us. ]

  return (Tutorial_Calculator__CreateTrue (ILU_NIL, ILU_NIL, the_value));
}
```

We also modify each of our six true methods to use the user data field, which is available through the `void * data` field of any `ILU_C_Object` value. For example, the `Add` method now looks like:

```
  void
server_Tutorial_Calculator_Add (
  Tutorial_Calculator self,
  CORBA_double v,
  CORBA_Environment *env)
{

[ The user data field is available as the field "void *data" of
  the object instance "self", so we'll just add "v" to it. ]

  *((CORBA_double *) (self->data)) += v;
}
```

The complete implementation can be found in Appendix B [Calculator-impl.c], page 34.

0.3.4 Making a Library

Our implementation can now be combined with the code generated by the `c-stubber` program to form a library. Assuming that our implementation resides in a file called '`Calculator-impl.c`', we compile each of the C source files, and combine them in a library (assuming that ILU has been installed under '`/usr/local/ilu/`'):

```
% make
/usr/local/ilu/bin/c-stubber  Tutorial.isl
header file interface Tutorial to ./Tutorial.h...
common code for interface Tutorial to ./Tutorial-common.c...
code for surrogate stubs of interface Tutorial to ./Tutorial-surrogate.c...
code for true stubs of interface Tutorial to ./Tutorial-true.c...
rm -f Tutorial-common.o
cc -c -I. -I/usr/local/ilu/include Tutorial-common.c
rm -f Tutorial-surrogate.o
cc -c -I. -I/usr/local/ilu/include Tutorial-surrogate.c
rm -f Tutorial-true.o
cc -c -I. -I/usr/local/ilu/include Tutorial-true.c
rm -f Calculator-impl.o
cc -c -I. -I/usr/local/ilu/include Calculator-impl.c
rm -f libTutorial.a
ar clq libTutorial.a  Tutorial-true.o \
                      Tutorial-surrogate.o \
                      Tutorial-common.o \
                      Calculator-impl.o
ranlib libTutorial.a
%
```

The exact paths used, and the exact form of the `cc` and `ar` commands will differ from system to system. On some systems, the `ranlib` command is not necessary.

0.4 Using the True Module as a Library

If an implementation of a module (sometimes called the *true module*) is available as a library, how would it be used? Taking our 4-function calculator as an example, suppose that a programmer wanted to write a C program which used its functionality; how would that functionality be seen from that application? It would appear as an object of the `Calculator` type; that is, a value of the C type `Tutorial_Calculator`, which would be used as the first argument to any of the six methods that the library exports. These methods would be called by their *generic function names*, rather than their true names, because this is a *use* of the module rather than an *implementation* of the module.

Another question that immediately comes up is, "how do I get my hands on an instance of the `Calculator` object to begin with?" Remember that we added the function `Create_Tutorial_Calculator`, which will return an instance of a calculator object.

So, a very simple program to use the `Tutorial` module might be the following:

```
/* simple1.c */
[ A simple program that demonstrates the use of the
   Tutorial true module as a library. ]

#include <stdio.h>      /* for NULL */
#include <stdlib.h>     /* for atof */

[ Include the header file for the Tutorial interface, so that our
   types and methods will be defined. ]

#include <Tutorial.h>

[ We should also define a prototype for the Create function
   exported from the implementation of the Tutorial module. ]

extern Tutorial_Calculator Create_Tutorial_Calculator(void);

[ A simple program:
   1) make an instance of Tutorial.Calculator
   2) add all the arguments by invoking the Add method
   3) print the resultant value. ]
```

[continued on following page]

```
int main (int argc, char **argv)
{
  Tutorial_Calculator c;
  CORBA_double v;
  char **arg;
  CORBA_Environment env;

[ Initialize the Tutorial module. ]

  Tutorial__InitializeServer();

[ Create an instance of a Tutorial.Calculator object. ]

  if ((c = Create_Tutorial_Calculator()) == NULL)
    {
      fprintf (stderr, "Couldn't create calculator!\n");
      exit(1);
    }

[ Clear the calculator before using it. ]

  Tutorial_Calculator_SetValue (c, 0.0, &env);

[ Now loop over the arguments, adding each in turn. ]

  for (arg = ++argv;  *arg != NULL;  arg++)
    {
      v = atof (*arg);
      Tutorial_Calculator_Add (c, v, &env);
    }

[ And print the result. ]

  v = Tutorial_Calculator_GetValue (c, &env);
  printf ("the sum is %.5e\n", v);

  exit (0);
}
```

This program would be compiled and run as follows:

```
% cc -o simple1 -I. -I/usr/local/ilu/include simple1.c libTutorial.a \
 /usr/local/ilu/lib/{libilu-c,libilu}.a
% ./simple1 34.9 45.23111 12
the sum is 9.21311e+01
%
```

This is a completely self-contained use of the `Tutorial` implementation; when a method is called, it is the true method that is invoked. The use of ILU in this program adds some overhead in terms of included code, but has almost the same performance as a version of this program that does not use ILU.

0.4.1 Checking for Exceptions

Suppose, instead of the `Add` method, we'd called the `Divide` method. In that case, we might have had to *handle* a `DivideByZero` exception; that is, notice the exception and do something sensible. We do this by testing the `env` parameter. We can test this parameter in a number of ways; the simplest is to use the macro `ILU_C_SUCCESSFUL`, which evaluates to 1 if no exception was raised, or 0 if some exception occurred. We could then use the macro `ILU_C_EXCEPTION_ID` to retrieve the name of the exception and print it. For example, here's a fragment of C code that checks for the `DivideByZero` exception:

```
/* from simple2.c */
...
Tutorial_Calculator_Divide (c, v, &env);
if (! ILU_C_SUCCESSFUL(&env))
  {
    fprintf (stderr, "Divide signalled exception <%s>.\n",
             ILU_C_EXCEPTION_ID(&env));
    exit(1);
  }
...
```

And here's an example of what we get when it runs:

```
% ./simple2 12345 6 7 8 9
the quotient is 4.08234e+00
% ./simple2 12345 6 0 8 9
Divide signalled exception <Tutorial: DivideByZero>.
%
```

Actually, every method may return an exception, as there are a number of standard system exceptions which may be signalled even by methods which have no declared exceptions. So we should check every method to see if it succeeded, even simple ones like `GetValue`.

0.5 Providing the True Module as a Network Service

Now let's see what's involved in providing the calculator functionality as a network service. Basically, there are three things to look at:

- providing a "factory" to build calculator objects;
- publishing the name of the factory; and
- writing a server program.

0.5.1 Using Factories to Build Objects

When one program uses code from another address space, it has to get its hands on an instance of an ILU object, to be able to call methods. In our library application, we simply made a call into the true module, to create an instance of the calculator object. In the networked world, we need to do the same kind of thing, but this time the call into the true module has to be a method on an object type. In short, we need to have some object type which exports a method something like

```
CreateCalculator () : Calculator
```

There are several ways to provide this. The standard way of doing it is to add an object type to our `Tutorial` interface, which contains this method. This kind of object type is sometimes called a *factory*, because it exists only in order to build instances of other object types. We'll add the following type definition to our '`Tutorial.isl`':

```
TYPE Factory = OBJECT
  METHODS
    CreateCalculator () : Calculator
  END;
```

Then we need to provide an implementation of the `Factory` object type, just as we did with the `Calculator` type:

```
/* Factory-impl.c */

[ Include the Tutorial header file, to get all the defined
  types and function prototypes. ]

#include <Tutorial.h>

[ Code for the Factory object type. ]

extern Tutorial_Calculator Create_Tutorial_Calculator(void);

  Tutorial_Calculator
server_Tutorial_Factory_CreateCalculator (
  Tutorial_Factory self,
  CORBA_Environment *env)
{
  return (Create_Tutorial_Calculator());
}
```

Now, to provide other programs a way of creating calculator objects, we'll just create just one instance of `Tutorial.Factory`, and let programs call the `CreateCalculator` method on that at will, to obtain new calculator objects.

0.5.2 Publishing a Well-Known Instance

The question then arises, how does a program that wants to use the `Factory` object get its hands on that one well-known instance? The answer is to use the *simple binding* system built into ILU. Simple binding allows a program acting as a "server" to *publish* the location of a well-known object, and allows programs acting as "clients" of that server to look up the location, given the object's name.

The name of an ILU object instance has two parts, which are the *instance handle* of the object, and the name of its *kernel server*, called the *server ID*. (The kernel server is a data structure maintained by the ILU kernel which takes care of all communication between different address spaces.) These two combined must form a universally unique ID for the object. Usually you can simply let the ILU system choose names for your objects automatically, in which case it takes care to choose names which will never conflict with names in use by others. However, for objects which we wish to publish, we need to specify what the name of an object will be, so that users of the well-known object can find it.

When working with the C programming language, this act of explicitly specifying an object name is divided into two steps. First, we create a kernel server with a specified server ID. Secondly, we create an instance of an object on this new server, with a specified instance handle. Together, the server ID and the instance handle form the name of the instance.

For instance, we might use a server ID of `Tutorial.`*domain*, where *domain* is your Internet domain (typically something like *department*`.`*company*`.com`, or *department*`.`*univerity*`.edu`). This serves to distinguish your server from other servers on the net. Then we can use a simple instance handle, like `theFactory`. The name, or *object ID*, of this object would then be the pair (`Tutorial.`*domain*, `theFactory`), where *domain* would vary from place to place. Note that this implies that only one instance of this object is going to exist in the whole domain. If you have many people using different versions of this object in your domain, you should introduce more qualifiers in the server ID so that your kernel server can be distinguished from that run by others.

0.5.3 The Server Program

Given this information, we can now write a complete program that will serve as a provider of calculator objects to other programs. It will create a single `Factory` instance with a well-known name, publish that instance, then sit tight servicing methods invoked on its objects. Here's what it looks like:

```
/* server.c */

#include <stdio.h>      /* for stderr and NULL */

[ Include the Tutorial header file, to get all the defined
   types and function prototypes. ]

#include <Tutorial.h>

int main (int ac, char **av)
{
  Tutorial_Factory theFactory;
  ilu_Server theServer;
  int stop;
[ Continued on next page... ]
```

```
[ This program is to be called with one argument, the server ID
  to use ("Tutorial.foo.something.com" or something like that.) ]

  if (ac < 2)
    {
      fprintf (stderr, "Usage:  server SERVER-ID\n");
      exit(1);
    }
[ In any server program, we have to initialize each interface
  that we're providing types from, by calling the InitializeServer
  method on that interface.  In this case, it's just the
  Tutorial interface. ]

  Tutorial__InitializeServer();

[ We make a "kernel server", using the server ID that was
  passed in on the command line, the default "object table",
  the default protocol for data pickling and message packets,
  the default transport system for getting data back and forth,
  and we make this kernel server the default server for the
  program. ]

  theServer = ILU_C_InitializeServer (
                    av[1],     /* the server ID */
                    NULL,      /* use no object table */
                    NULL,      /* use default protocol */
                    NULL,      /* use default transport */
                    ilu_TRUE   /* establish as default server */
                    );

[ Now that we have a server, we create an instance of the
  Factory object type, with the instance handle of "Factory",
  by calling the automatically generated procedure
  "Tutorial_Factory__CreateTrue()". ]

  theFactory = Tutorial_Factory__CreateTrue ("theFactory",
                                             /* instance handle */
                                             theServer,
                                             /* server to use */
                                             NULL
                                             /* no user data */
                                             );

[ Now make the Factory object "well-known" by publishing it.
  PublishObject will return NULL if it can't publish the
  object; otherwise it will return a pointer to a string,
  which is the "publish proof". ]

  if (ILU_C_PublishObject(theFactory) == NULL)
```

```
    {
      fprintf (stderr, "Can't publish theFactory object.\n");
      exit(1);
    }
  else
    {
[ Now we print the string binding handle (the object's name plus
  its location) of the new instance. ]

      printf ("Factory instance published.\n");
      printf ("Its SBH is \"%s\".\n", ILU_C_SBHOfObject(theFactory));

[ ilu_RunMainLoop() is an event dispatching loop.  It may
  be exited by invoking ilu_ExitMainLoop() passing the same
  "int *" value  ilu_RunMainLoop was invoked with. ]

      ilu_RunMainLoop (&stop);
    }
}
```

When we run this program, we'll see something like:

```
% ./server Tutorial.dept.company.com
Factory instance published.
Its SBH is "ilu:Tutorial.dept.company.com/theFactory;@somegibberish".
%
```

This indicates that the object known as `theFactory@Tutorial.`*dept.company.com* is being exported in a particular way, which is encoded in the *somegibberish* part of the string binding handle. Your specific numbers may vary, but it should look similar.

0.6 Using the Network Service

Given that someone has exported a module as a network service, by publishing the location of a well-known instance of an object type, a potential client of that module can then use the module by binding to that well-known instance. It does this by calling the standard ILU routine `ILU_C_LookupObject`, which takes the name and type of an instance, and attempts to find that instance on the net.

So, in our first example, we could replace the call to `Create_Tutorial_Calculator` with a routine that calls `ILU_C_LookupObject` to find the factory, then creates an instance of a `Calculator`. The full code of the revised example, '`simple3.c`', is available as Appendix D [simple3.c], page 39, but here's what the new code for obtaining an instance of a `Calculator` looks like:

```
...
  static Tutorial_Calculator
Get_Tutorial_Calculator (char *factObjSID, char *factObjIH)
{
  Tutorial_Factory f;
  Tutorial_Calculator c;
  CORBA_Environment env;

[ We have to call ILU_C_LookupObject() with the object ID of the factory
  object (the SID and IH pair), and the "type" of the object we're looking
  for, which is always available as "TYPENAME__MSType". ]

  f = ILU_C_LookupObject (factObjSID, factObjIH, Tutorial_Factory__MSType);

  if (f == NULL) {
      fprintf (stderr, "Couldn't find Factory object <%s %s>.\n",
               factObjSID, factObjIH);
      return (NULL);  };

[ Now call the CreateCalculator method on the factory, and check the result. ]

  c = Tutorial_Factory_CreateCalculator (f, &env);
  if (! ILU_C_SUCCESSFUL(&env)) {
      fprintf (stderr,
               "Call to CreateCalculator failed with exception <%s>.\n",
               ILU_C_EXCEPTION_ID(&env));
      return (NULL);  };

[ Return the calculator object... ]

  return (c);
}
```

We then link and use the `simple3` program:

```
% cc -o simple3 simple3.o libTutorial.a ${ILUHOME}/lib/libilu-c.a \
    ${ILUHOME}/lib/libilu.a
% ./simple3 Tutorial.dept.company.com theFactory 1 2 3 4 5 6
the sum is 2.10000e+01
%
```

0.7 Subtyping and Other ISL Types

ILU ISL contains support for a number of types other than object types and `REAL`. The primitive ISL types include 16, 32, and 64 bit signed and unsigned integers, bytes, 8 and 16 bit characters, a boolean type, and 32, 64, and 128 bit floating point types. A number of type constructors allow specification of arrays, sequences, records, unions, and enumerations, as well as object types. The ISL `OPTIONAL` type constructor provides an implicit union of some type with `NULL`, which is useful for building recursive data structures such as linked lists or binary trees.

To illustrate some of these types, we'll extend the `Tutorial.Calculator` type. Many real-world desktop calculators include a *register tape*, a printed listing of all the operations that have been performed, with a display of what the value of the calculator was after each operation. We'll add a register tape to `Tutorial.Calculator`.

We could do it by adding a new method to `Tutorial.Calculator`, called `GetTape`. Unfortunately, this would break our existing code, because it would change the `Tutorial.Calculator` object type, and existing compiled clients wouldn't be able to recognize the new object type. Instead, we'll extend the object type by *subtyping*; that is, by creating a new object type which uses `Tutorial.Calculator` as a *supertype*, but adds new methods of its own. This subtype will actually have two types; both its own new type, and `Tutorial.Calculator`. We'll also define a subtype of the `Tutorial.Factory` type, to allow us to create new instances of the new `Calculator` subtype. Finally, we'll define a new module interface for the new types, so that we don't have to modify the `Tutorial` interface.

First, let's define the necessary type to represent the operations performed on the calculator:

```
INTERFACE Tutorial2 IMPORTS Tutorial END;

TYPE OpType = ENUMERATION
    SetValue, Add, Subtract, Multiply, Divide END;

TYPE Operation = RECORD
    op : OpType,
    value : REAL,
    accumulator : REAL
  END;

TYPE RegisterTape = SEQUENCE OF Operation;
```

The enumerated type `OpType` defines an abstract type with five possible values. The type `Operation` defines a record type (in C, a struct) with 3 fields: the *op* field, which tells us which of the five possible calculator operations was performed, the *value* field, which tells us the value of the operand for the operation, and the *accumulator* field, which tells us what the value of the calculator was after the operation had been performed. Finally, the `Operation` type is a simple sequence, or list, of `Operation`. Note that `Tutorial2` *imports* `Tutorial`; that is, it allows the use of the `Tutorial` types, exceptions, and constants, in the specifications in `Tutorial2`.

Now we define the new object types (in the same file):

```
TYPE TapeCalculator = OBJECT COLLECTIBLE
  SUPERTYPES Tutorial.Calculator END
  DOCUMENTATION "4 function calculator with register tape"
  METHODS
    GetTape () : RegisterTape
  END;

TYPE Factory = OBJECT SUPERTYPES Tutorial.Factory END
  METHODS
    CreateTapeCalculator () : TapeCalculator
  END;
```

The `SUPERTYPES` attribute of an object type may take multiple object type names, so ISL supports multiple inheritance. The `Tutorial2.TapeCalculator` type will now support the six methods of `Tutorial.Calculator`, as well as its own method, `GetTape`.

We then need to provide an implementation for `Tutorial2.TapeCalculator` (see Appendix F [TapeCalculator-impl.c], page 47 for the actual code). We use a data structure for the state of each calculator object:

```
struct calc_state {
  CORBA_double the_value;
  Tutorial2_RegisterTape *tape;
};
```

and modify each method on the `TapeCalculator` object to record its invocation.

We also provide an implementation for `Tutorial2.Factory`:

```
/* Factory2-impl.c */

/* Include the Tutorial header file, to get all the defined
 * types and function prototypes.
 */

#include <Tutorial2.h>

/* Code for the Factory object type */

extern Tutorial2_TapeCalculator Create_Tutorial2_TapeCalculator(void);

  Tutorial_Calculator
server_Tutorial2_Factory_CreateCalculator (
  Tutorial2_Factory f,
  CORBA_Environment *env)
{
  return ((Tutorial_Calculator) Create_Tutorial2_TapeCalculator());
}

  Tutorial2_TapeCalculator
server_Tutorial2_Factory_CreateTapeCalculator (
  Tutorial2_Factory f,
  CORBA_Environment *env)
{
  return (Create_Tutorial2_TapeCalculator());
}
```

Note that both the `Tutorial2.Factory.CreateCalculator` and `Tutorial2.Factory.CreateTapeCalculator` methods create and return instances of `Tutorial2.TapeCalculator`. This is valid, because instances of `Tutorial2.TapeCalculator` are also instances of `Tutorial.Calculator`.

Both '`TapeCalculator-impl.o`' and '`Factory2-impl.o`' are added to our library, along with the generated files from running `c-stubber` on '`Tutorial2.isl`':

```
% make
/usr/local/ilu/bin/c-stubber  Tutorial.isl
header file interface Tutorial to ./Tutorial.h...
common code for interface Tutorial to ./Tutorial-common.c...
code for surrogate stubs of interface Tutorial to ./Tutorial-surrogate.c...
code for true stubs of interface Tutorial to ./Tutorial-true.c...
rm -f Tutorial-common.o
cc -c -I. -I/usr/local/ilu/include Tutorial-common.c
rm -f Tutorial-surrogate.o
cc -c -I. -I/usr/local/ilu/include Tutorial-surrogate.c
rm -f Tutorial-true.o
cc -c -I. -I/usr/local/ilu/include Tutorial-true.c
/usr/local/ilu/bin/c-stubber  Tutorial2.isl
header file interface Tutorial2 to ./Tutorial2.h...
common code for interface Tutorial2 to ./Tutorial2-common.c...
code for surrogate stubs of interface Tutorial2 to ./Tutorial2-surrogate.c...
code for true stubs of interface Tutorial2 to ./Tutorial2-true.c...
rm -f Tutorial2-common.o
cc -c -I. -I/usr/local/ilu/include Tutorial2-common.c
rm -f Tutorial2-surrogate.o
cc -c -I. -I/usr/local/ilu/include Tutorial2-surrogate.c
rm -f Tutorial2-true.o
cc -c -I. -I/usr/local/ilu/include Tutorial2-true.c
rm -f Calculator-impl.o
cc -c -I. -I/usr/local/ilu/include Calculator-impl.c
rm -f TapeCalculator-impl.o
cc -c -I. -I/usr/local/ilu/include TapeCalculator-impl.c
rm -f Factory-impl.o
cc -c -I. -I/usr/local/ilu/include Factory-impl.c
rm -f Factory2-impl.o
cc -c -I. -I/usr/local/ilu/include Factory2-impl.c
rm -f libTutorial.a
ar clq libTutorial.a  Tutorial-true.o \
                      Tutorial-surrogate.o \
                      Tutorial-common.o \
                      Calculator-impl.o \
                      Factory-impl.o \
                      Tutorial2-true.o \
                      Tutorial2-surrogate.o \
                      Tutorial2-common.o \
                      TapeCalculator-impl.o \
                      Factory2-impl.o
ar: filename Tutorial-surrogate.o truncated to Tutorial-surrog
ar: filename Tutorial-common.o truncated to Tutorial-common
ar: filename Calculator-impl.o truncated to Calculator-impl
ar: filename Tutorial2-surrogate.o truncated to Tutorial2-surro
ar: filename Tutorial2-common.o truncated to Tutorial2-commo
ar: filename TapeCalculator-impl.o truncated to TapeCalculator-
ranlib libTutorial.a
%
```

Now we modify '`server.c`' to create an instance of `Tutorial2.Factory`, instead of `Tutorial.Factory`, and to initialize the `Tutorial2` true-side code (See Appendix C [server2.c], page 37 for the actual code).

Finally, see Appendix E [simple4.c], page 42, for an example of a client of the `TapeCalculator` type.

Note that one nice result of this approach to versioning is that old clients, which know nothing about the new `TapeCalculator` class, or about the whole `Tutorial2` interface in general, will continue to function, since every instance of `Tutorial2.TapeCalculator` is also an instance of `Tutorial.Calculator`, and every instance of `Tutorial2.Factory` is also an instance of `Tutorial.Factory`.

0.8 Makefiles and Imakefiles

ILU includes support for a kind of macro-ized make system called called `imake`, that's distributed with the X Window System. With `imake`, most of the specific cruft of program options and switches is hidden behind macros. ILU provides some specific macros to do ILU-ish things, like running the `c-stubber`. To use `imake`, you put your rules in a file called '`Imakefile`', then run the program `ilumkmf` to create the real '`Makefile`'. Once you have the '`Makefile`', you can just use `make`.

An '`Imakefile`' for `Tutorial` and `Tutorial2` would look like:

```
NormalObjectRule()
DependTarget()

InterfaceTarget(Tutorial.isl)
ILUCTarget(Tutorial.h Tutorial-true.c Tutorial-surrogate.c Tutorial-common.c,
Tutorial.isl)
ObjectTarget(Tutorial-common.o)
ObjectTarget(Tutorial-surrogate.o)
ObjectTarget(Tutorial-true.o)

ObjectTarget(Calculator-impl.o)
Calculator-impl.o : Tutorial.h Calculator-impl.c

ObjectTarget(Factory-impl.o)
Factory-impl.o : Tutorial.h Factory-impl.c

InterfaceTarget(Tutorial2.isl)
ILUCTarget(Tutorial2.h Tutorial2-true.c Tutorial2-surrogate.c Tutorial2-common.c,
Tutorial2.isl)
ObjectTarget(Tutorial2-common.o)
ObjectTarget(Tutorial2-surrogate.o)
ObjectTarget(Tutorial2-true.o)

ObjectTarget(TapeCalculator-impl.o)
TapeCalculator-impl.o : Tutorial.h Tutorial2.h TapeCalculator-impl.c

ObjectTarget(Factory2-impl.o)
Factory2-impl.o : Tutorial.h Factory2-impl.c

LibraryTarget (libTutorial.a, Tutorial-true.o Tutorial-surrogate.o Tutorial-
common.o Calculator-impl.o Factory-impl.o Tutorial2-true.o Tutorial2-surrogate.o
Tutorial2-common.o TapeCalculator-impl.o Factory2-impl.o)

simple1.o : Tutorial.h simple1.c
ILUCProgramTarget(simple1, simple1.o, libTutorial.a,)

simple2.o : Tutorial.h simple2.c
```

```
ILUCProgramTarget(simple2, simple2.o, libTutorial.a,)

simple3.o : Tutorial.h simple2.c
ILUCProgramTarget(simple3, simple3.o, libTutorial.a,)

simple4.o : Tutorial.h simple4.c
ILUCProgramTarget(simple4, simple4.o, libTutorial.a,)

server.o : Tutorial.h server.c
ILUCProgramTarget(server, server.o, libTutorial.a,)

server2.o : Tutorial.h Tutorial2.h server2.c
ILUCProgramTarget(server2, server2.o, libTutorial.a,)
```

The two macros `NormalObjectRule()` and `DependTarget()` are required before the other macros. The macro `InterfaceTarget()` marks a file as being an ISL file. The macro `ObjectTarget()` specifies that the indicated object file should be produced. The macro `ILUCTarget()` indicates that running the `c-stubber` on the second argument will produce the first argument's files. The macro `LibraryTarget()` specifies that the library named by the first argument is composed of the object files named by the second argument. The macro `ILUCProgramTarget()` specifies the components necessary to build the image named by the first argument; the second argument lists all object files on which it is dependent, the third argument lists all libraries on which it is dependent, and the fourth lists all libraries and object files which it should be linked against, but which it is not dependent on.

Notice that normal `make` dependency rules can also be used in an '`Imakefile`'. The '`Imakefile`' is passed through the C preprocessor to expand the macros, so it is also possible to use features of `cpp` in the '`Imakefile`'.

0.9 Using OMG IDL with ILU

ILU also supports the use of the interface definition language OMG IDL, defined by the Object Management Group (OMG) for their Common Object Request Broker Architecture (CORBA). OMG IDL uses a C++-like syntax, so it may be easier for C and C++ programmers to use than ILU ISL. Unfortunately, CORBA doesn't include some of the concepts in ILU, such as garbage collection for transient objects, or `OPTIONAL` types, so not every ILU interface can be expressed in OMG IDL, but many of them can. For example, here is the OMG IDL version of the `Tutorial` interface:

```
module Tutorial {

  exception DivideByZero {};

  interface Calculator {
    // Set the value of the calculator to 'v'
    void SetValue (in double v);
    // Return the value of the calculator
    double GetValue ();
    // Adds 'v' to the calculator's value
    void Add (in double v);
    // Subtracts 'v' from the calculator's value
    void Subtract (in double v);
    // Multiplies the calculator's value by 'v'
    void Multiply (in double v);
    // Divides the calculator's value by 'v'
    void Divide (in double v) raises (DivideByZero);
  };

  interface Factory {
    // Create and return an instance of a Calculator object
    Calculator CreateCalculator();
  };
};
```

This can be used with the `c-stubber`:

```
% c-stubber Tutorial.idl
header file for interface Tutorial to ./Tutorial.h...
common code for interface Tutorial to ./Tutorial-common.c...
code for surrogate stubs of interface Tutorial to ./Tutorial-surrogate.c...
code for true stubs of interface Tutorial to ./Tutorial-true.c...
%
```

This will be a bit slower than running the `c-stubber` on the equivalent ISL file, as the program works by converting the OMG IDL into ISL, then compiling from the ISL description. OMG IDL interfaces can be checked by running the OMG IDL-to-ILU ISL converter, `idl2isl`, directly:

```
% idl2isl Tutorial.idl
INTERFACE Tutorial;

EXCEPTION DivideByZero;
TYPE Calculator = OBJECT OPTIONAL
        METHODS
                SetValue (v : REAL),
                GetValue () : REAL,
                Add (v : REAL),
                Subtract (v : REAL),
                Multiply (v : REAL),
                Divide (v : REAL)
                        RAISES DivideByZero END
        END;
TYPE Factory = OBJECT OPTIONAL
        METHODS
                CreateCalculator () : Calculator
        END;
%
```

You will notice that the ISL interface generated by `idl2isl` is a bit different, in that the object type modifier `OPTIONAL` is used in the description of the `Calculator` and `Factory` types. This is because CORBA has the notion that any object type instance passed as a parameter or return value (or field in an array, or element of a sequence, etc.) may be `NULL`, instead of being a valid instance pointer. Thus, when working with OMG IDL descriptions of your interfaces, it is necessary to check the return type of methods like `Tutorial.Factory.CreateCalculator` to see that a valid object reference has been returned, before using the object. ISL allows you to have these CORBA-style objects, by using the `OPTIONAL` modifier in the declaration of an object type, but it also allows object pointers which can't be `NULL`. By default ILU object instances may not be `NULL`.

The OMG IDL version of `Tutorial2` can be found in [Tutorial2.idl], page 52.

Appendix A General ILU Info

The Inter-Language Unification system (ILU) is a multi-language object interface system. The object interfaces provided by ILU hide implementation distinctions between different languages, between different address spaces, and between operating system types. ILU can be used to build multi-lingual object-oriented libraries ("class libraries") with well-specified language-independent interfaces. It can also be used to implement distributed systems. It can also be used to define and document interfaces between the modules of non-distributed programs.

The 2.0 release of ILU contains support for the programming languages ANSI C, C++, Modula-3, Python, and Common Lisp. It has been installed on many flavors of UNIX, including SPARC machines running SunOS 4.1.3 and Solaris 2, SGI MIPS machines running IRIX 5.2, Intel 486 machines running Linux 1.1.78, DEC Alpha machines with OSF/1, IBM RS/6000 machines running AIX, and HP machines running HP/UX. It runs on Microsoft Windows 3.1, Windows 95, and Windows NT environments. It supports both threaded and non-threaded operation. Since one of the implementation goals of ILU is to maximize compatibility with existing open standards, ILU provides support for use of the OMG CORBA IDL interface description language, and can be thought of as a CORBA ORB system (though with omissions from and extensions to the CORBA spec). As another result, ILU includes a self-contained implementation of ONC RPC.

ILU is available free from `ftp://ftp.parc.xerox.com/pub/ilu/ilu.html`.

Appendix B Calculator-impl.c

```
/* The first thing we need to do is to include the generated header
 * file, which describes the types and methods used by the Tutorial
 * interface
 */

#include <Tutorial.h>

/* We define a function which creates a new instance of a Calculator
 * object.
 */

  Tutorial_Calculator
Create_Tutorial_Calculator ()
{
  CORBA_double *the_value = (CORBA_double *) malloc(sizeof(CORBA_double));

  *the_value = 0.0;     /* zero out our value */

  /* The function "Tutorial_Calculator__CreateTrue" is automatically
   * generated into the file "Tutorial-true.c" by the c-stubber.
   * It takes three arguments, INSTANCE-HANDLE, SERVER, and
   * USER-DATA-FIELD, and returns a new instance of Tutorial_Calculator.
   * We don't care about what the INSTANCE-HANDLE and SERVER of Calculator
   * instances are, so we'll pass ILU_NIL (which is another name
   * for NULL) for the first two arguments, which will cause ILU
   * to choose reasonable default values for us.
   */

  return (Tutorial_Calculator__CreateTrue (ILU_NIL, ILU_NIL, the_value));
}

/* Now to implement the method, we simply take the true prototype
 * and add whatever code is necessary to actually perform the operation.
 */

  void
server_Tutorial_Calculator_SetValue (
  Tutorial_Calculator self,
  CORBA_double v,
  CORBA_Environment *env)
{
  /* The user data field is available as the field "void *data" of
   * any object instance, so we'll just set it to be "v".
   */

  *((CORBA_double *) (self->data)) = v;
}
```

```
   CORBA_double
server_Tutorial_Calculator_GetValue (
   Tutorial_Calculator self,
   CORBA_Environment *env)
{
  return (*((CORBA_double *) (self->data)));
}

  void
server_Tutorial_Calculator_Add (
  Tutorial_Calculator self,
  CORBA_double v,
  CORBA_Environment *env)
{
  *((CORBA_double *) (self->data)) += v;
}

  void
server_Tutorial_Calculator_Subtract (
  Tutorial_Calculator self,
  CORBA_double v,
  CORBA_Environment *env)
{
  *((CORBA_double *) (self->data)) -= v;
}

  void
server_Tutorial_Calculator_Multiply (
  Tutorial_Calculator self,
  CORBA_double v,
  CORBA_Environment *env)
{
  *((CORBA_double *) (self->data)) *= v;
}

/* The Divide method gets a little trickier.  We have to compare the
 * value "v" to zero, which for floating point values actually means
 * comparing it to some epsilon to see whether it is less than that
 * epsilon, and then if it is "zero" we need to signal an error, by
 * "raising" the "DivideByZero" exception.  The way of raising exceptions
 * in ILU C is rather clumsy, so we'll define a macro to make it look
 * prettier.  We also define some macros to make testing the value
 * of "v" a  bit prettier.
 */

#define ABS(x)  (((x)<0)?(-(x)):(x))
#define SOME_EPSILON    0.000000001     /* zero, practically speaking */

#define RAISE(env,exception) { (env)->returnCode=(exception);\
```

```
                                (env)->_major=CORBA_USER_EXCEPTION; }

  void
server_Tutorial_Calculator_Divide (
  Tutorial_Calculator self,
  CORBA_double v,
  CORBA_Environment *env)
{
  if (ABS(v) < SOME_EPSILON)
    RAISE(env, ex_Tutorial_DivideByZero)
  else
    *((CORBA_double *) (self->data)) /= v;
}
```

Appendix C server2.c

```
/* server2.c */

#include <stdio.h>      /* for stderr and NULL */

/* Include the Tutorial2 header file, to get all the defined
 * types and function prototypes.  Note that Tutorial2.h will
 * #include Tutorial.h for us.
 */

#include <Tutorial2.h>

int main (int ac, char **av)
{
  Tutorial2_Factory theFactory;
  ilu_Server theServer;
  int stop;

  /* this program is to be called with one argument, the server ID
     to use ("Tutorial.foo.something.com" or something like that.)
  */

  if (ac < 2)
    {
     fprintf (stderr, "Usage:  server SERVER-ID\n");
     exit(1);
    }

  /* In any server program, we have to initialize each interface
     that we're providing types from, by calling the InitializeServer
     method on that interface.  In this case, it's both the
     Tutorial and Tutorial2 interfaces.
  */

  Tutorial__InitializeServer();
  Tutorial2__InitializeServer();

  /* We make a "kernel server", using the server ID that was
     passed in on the command line, the default "object table",
     the default protocol for data pickling and message packets,
     the default transport system for getting data back and forth,
     and we make this kernel server the default server for the
     program.
  */

  theServer = ILU_C_InitializeServer(av[1],
                                     /* the server ID */
                                     NULL,
```

```
                                        /* use std object table */
                                        NULL,
                                        /* use default protocol */
                                        NULL,
                                        /* use default transport */
                                        ilu_TRUE
                                        /* establish as default server */
                                        );

    /* Now that we have a server, we create an instance of the
       Factory object type, with the instance handle of "Factory",
       by calling the automatically generated procedure
       "Tutorial_Factory__CreateTrue()".
    */

    theFactory = Tutorial2_Factory__CreateTrue ("theFactory",
                                                /* instance handle */
                                                theServer,
                                                /* server to use */
                                                NULL
                                                /* no user data */
                                                );

    /* Now make the Factory object "well-known" by publishing it.
       PublishObject will return NULL if it can't publish the
       object; otherwise it will return a pointer to a string,
       which is the "publish proof".
    */

    if (ILU_C_PublishObject(theFactory) == NULL)
      {
        fprintf (stderr, "Can't publish theFactory object.\n");
        exit(1);
      }
    else
      {
        /* Now we print the string binding handle (the object's name plus
           its location) of the new instance.
        */

        printf ("Factory instance published.\n");
        printf ("Its SBH is \"%s\".\n", ILU_C_SBHOfObject(theFactory));

        /* ilu_RunMainLoop() is an event dispatching loop.  It may
           be exited by invoking ilu_ExitMainLoop() passing the same
           int * RunMainLoop was invoked with.  */

        ilu_RunMainLoop (&stop);
      }
}
```

Appendix D simple3.c

```
/* simple3.c */

#include <stdio.h> /* for NULL */
#include <stdlib.h> /* for atof */

/* Include the header file for the Tutorial interface, so that our
 * types and methods will be defined.
 */

#include <Tutorial.h>

/* We define a new routine, "Get_Tutorial_Calculator", which
 * finds the tutorial factory, then creates a new Calculator
 * object for us.
 */

  static Tutorial_Calculator
Get_Tutorial_Calculator (char *sid, char *ih)
{
  Tutorial_Factory f;
  Tutorial_Calculator c;
  ILU_C_ENVIRONMENT env;

  /* We have to call ILU_C_LookupObject() with the object ID of
   * the factory object, and the ``type'' of the object we're looking
   * for, which is always available as "TYPENAME__MSType".
   */

  f = ILU_C_LookupObject (sid, ih, Tutorial_Factory__MSType);

  if (f == NULL)
    {
      fprintf (stderr, "Couldn't find Factory object <%s %s>.\n",
               sid, ih);
      return (NULL);
    }

  /* Now call the CreateCalculator method on the factory, and check
   * the result...
   */

  c = Tutorial_Factory_CreateCalculator (f, &env);
  if (! ILU_C_SUCCESSFUL(&env))
    {
      fprintf (stderr, "Call to CreateCalculator failed with exception <%s>.\n",
       ILU_C_EXCEPTION_ID(&env));
      return (NULL);
```

```
    }

  /* And return the calculator */

  return (c);
}

/* A simple program:
 *  1)  make an instance of Tutorial.Calculator
 *  2)  add all the arguments by invoking the Add method
 *  3)  print the resultant value.
 */

int main (int argc, char **argv)
{
  Tutorial_Calculator c;
  CORBA_double v;
  char *sid, *ih;
  char **arg;
  ILU_C_ENVIRONMENT env;

  if (argc < 4)
    {
      fprintf (stderr, "Usage:  %s FACTORY-OBJECT-SID FACTORY-OBJECT-IH NUM-
BER [NUMBER...]\n",
       argv[0]);
      exit(1);
    }

  Tutorial__Initialize();

  sid = *++argv;
  ih = *++argv;
  if ((c = Get_Tutorial_Calculator(sid, ih)) == NULL)
    {
      fprintf (stderr, "Couldn't create calculator!\n");
      exit(1);
    }

  /* clear the calculator before using it */

  Tutorial_Calculator_SetValue (c, 0.0, &env);
  if (! ILU_C_SUCCESSFUL(&env))
    {
      fprintf (stderr, "SetValue signalled <%s>.\n",
       ILU_C_EXCEPTION_ID(&env));
      exit(1);
    }

  /* now loop over the arguments, adding each in turn */
```

```
  for (arg = ++argv;  *arg != NULL;  arg++)
    {
      v = atof (*arg);
      Tutorial_Calculator_Add (c, v, &env);
      if (! ILU_C_SUCCESSFUL(&env))
{
  fprintf (stderr, "Add signalled <%s>.\n",
   ILU_C_EXCEPTION_ID(&env));
  exit(1);
}
    }

  /* and print the result */

  v = Tutorial_Calculator_GetValue (c, &env);
  if (! ILU_C_SUCCESSFUL(&env))
    {
      fprintf (stderr, "GetValue signalled <%s>.\n",
       ILU_C_EXCEPTION_ID(&env));
      exit(1);
    }
  else
    printf ("the sum is %.5e\n", v);

  exit (0);
}
```

Appendix E simple4.c

```
/* simple4.c */

#include <stdio.h> /* for NULL */
#include <stdlib.h> /* for atof */

/* Include the header file for the Tutorial interface, so that our
 * types and methods will be defined.
 */

#include <Tutorial2.h>

/* We define a new routine, "Get_Tutorial_Calculator", which
 * finds the tutorial factory, then creates a new Calculator
 * object for us.
 */

  static Tutorial_Calculator
Get_Tutorial_Calculator (char *sid, char *ih)
{
  Tutorial_Factory f;
  Tutorial2_TapeCalculator c;
  ILU_C_ENVIRONMENT env;

  /* We have to call ILU_C_LookupObject() with the object ID of
   * the factory object, and the ``type'' of the object we're looking
   * for, which is always available as "TYPENAME__MSType".
   */

  f = ILU_C_LookupObject (sid, ih, Tutorial2_Factory__MSType);

  if (f == NULL)
    {
      fprintf (stderr, "Couldn't find Factory object <%s>.\n",
       factoryObjectID);
      return (NULL);
    }

  /* Now call the CreateCalculator method on the factory, and check
   * the result...
   */

  c = Tutorial2_Factory_CreateTapeCalculator (f, &env);
  if (! ILU_C_SUCCESSFUL(&env))
    {
      fprintf (stderr, "Call to CreateCalculator failed with exception <%s>.\n",
       ILU_C_EXCEPTION_ID(&env));
      return (NULL);
```

```
    }
  else
    printf ("Got calculator object \"%s\" of type \"%s\".\n",
    ILU_C_SBHOfObject(c), ILU_C_ClassName(c));

  /* And return the calculator */

  return ((Tutorial_Calculator) c);
}

/* A routine to print an operation from a Tutorial2.RegisterTape */

static char *NameOfOp (Tutorial2_OpType ot)
{
  static struct _ops { char *name; Tutorial2_OpType ot; } ops[] = {
    { "Add", Tutorial2_Add },
    { "Subtract", Tutorial2_Subtract },
    { "Divide", Tutorial2_Divide },
    { "Multiply", Tutorial2_Multiply },
    { "SetValue", Tutorial2_SetValue } };
  int opslen = sizeof(ops)/sizeof(struct _ops);
  int i;
  char *result;

  for (i = 0;  i < opslen;  i++)
    if (ops[i].ot == ot)
      return (ops[i].name);
  return ILU_NIL;
}

static void PrintOperation (Tutorial2_Operation *op, void *userArg)
{
  char *opname = NameOfOp (op->op);

  if (opname == ILU_NIL)
    printf ("** Unrecognized operation, op = %d!\n", op->op);
  else
    printf ("  %s(%.5f) => %.5f\n", opname, op->value, op->accumulator);
}

int main (int argc, char **argv)
{
  Tutorial_Calculator c;
  ILU_C_ENVIRONMENT e;
  char *line, *sid, *ih;
  char buf[1000];
  CORBA_double val;
  CORBA_double newval = 0.0;

  if (argc < 3)
```

```
    {
      fprintf (stderr, "Usage:  %s FACTORY-OBJECT-SID FACTORY-OBJECT-IH\n",
       argv[0]);
      exit(1);
    }

  Tutorial2__Initialize();

  sid = *++argv;
  ih = *++argv;
  if ((c = Get_Tutorial_Calculator(sid, ih)) == NULL)
    {
      fprintf (stderr, "Couldn't create calculator!\n");
      exit(1);
    }

  Tutorial_Calculator_SetValue(c, 0.0, &e);
  do {
    printf ("%.5f\n> ", newval);
    fflush(stdout);

    *buf = 'q';
    line = gets(buf);

    switch (buf[0]) {

    case '+':
      val = atof(buf+1);
      if (!((Tutorial_Calculator_Add(c, val, &e), ILU_C_SUCCESSFUL(&e))
    && (newval = Tutorial_Calculator_GetValue(c, &e),
ILU_C_SUCCESSFUL(&e))))
{
  fprintf (stderr, "Operation <%s> signals error <%s>.\n",
   buf, ILU_C_EXCEPTION_ID(&e));
}
      break;

    case '-':
      val = atof(buf+1);
      if (!((Tutorial_Calculator_Subtract(c, val, &e), ILU_C_SUCCESSFUL(&e))
    && (newval = Tutorial_Calculator_GetValue(c, &e),
ILU_C_SUCCESSFUL(&e))))
{
  fprintf (stderr, "Operation <%s> signals error <%s>.\n",
   buf, ILU_C_EXCEPTION_ID(&e));
}
      break;

    case '*':
      val = atof(buf+1);
```

```
      if (!((Tutorial_Calculator_Multiply(c, val, &e), ILU_C_SUCCESSFUL(&e))
    && (newval = Tutorial_Calculator_GetValue(c, &e),
ILU_C_SUCCESSFUL(&e))))
{
  fprintf (stderr, "Operation <%s> signals error <%s>.\n",
   buf, ILU_C_EXCEPTION_ID(&e));
}
      break;

    case '/':
      val = atof(buf+1);
      if (!((Tutorial_Calculator_Divide(c, val, &e), ILU_C_SUCCESSFUL(&e))
    && (newval = Tutorial_Calculator_GetValue(c, &e),
ILU_C_SUCCESSFUL(&e))))
{
  fprintf (stderr, "Operation <%s> signals error <%s>.\n",
   buf, ILU_C_EXCEPTION_ID(&e));
}
      break;

    case 'c':
      if (!(Tutorial_Calculator_SetValue(c, 0.0, &e), ILU_C_SUCCESSFUL(&e)))
{
  fprintf (stderr, "Operation <%s> signals error <%s>.\n",
   buf, ILU_C_EXCEPTION_ID(&e));
}
      newval = 0.0;
      break;

    case 'q':
      line = NULL;
      break;

    case 't':
      /* get the register tape and print it out */
      {
Tutorial2_TapeCalculator tc = (Tutorial2_TapeCalculator) c;
Tutorial2_RegisterTape *rt;

rt = Tutorial2_TapeCalculator_GetTape (c, &e);
if (! ILU_C_SUCCESSFUL(&e))
  {
    fprintf (stderr, "Operation <%s> signals error <%s>.\n",
     buf, ILU_C_EXCEPTION_ID(&e));
  }
else
  {
    Tutorial2_RegisterTape_Every (rt, PrintOperation, ILU_NIL);
    Tutorial2_RegisterTape__Free (rt);
  }
```

```
        }
        break;

      default:
        fprintf (stderr, "Invalid operation <%s>\n", buf);
        fprintf (stderr, "Valid ops are +, -, *, /, tape, clear, quit\n");
      };

  } while (line != NULL);

  return (0);
}
```

Appendix F TapeCalculator-impl.c

```
/* TapeCalculator-impl.c */

/* The first thing we need to do is to include the generated header
 * file, which describes the types and methods used by the Tutorial
 * interface
 */

#include <Tutorial2.h>

/* Now we define a struct type to represent the internal state of
 * our calculator objects
 */

struct calc_state {
  CORBA_double the_value;
  Tutorial2_RegisterTape *tape;
};

/* We define a function which creates a new instance of a Calculator
 * object.
 */

  Tutorial2_TapeCalculator
Create_Tutorial2_TapeCalculator ()
{
  struct calc_state *the_state = (struct calc_state *)
    ilu_must_malloc(sizeof(struct calc_state));

  /* zero out our value */
  the_state->the_value = 0.0;
  the_state->tape = Tutorial2_RegisterTape_Create(0, ILU_NIL);

  /* The function "Tutorial2_TapeCalculator__CreateTrue" is automatically
   * generated into the file "Tutorial-true.c" by the c-stubber.
   * It takes three arguments, INSTANCE-HANDLE, SERVER, and
   * USER-DATA-FIELD, and returns a new instance of Tutorial2_TapeCalculator.
   * We don't care about what the INSTANCE-HANDLE and SERVER of Calculator
   * instances are, so we'll pass ILU_NIL (which is another name
   * for NULL) for the first two arguments, which will cause ILU
   * to choose reasonable default values for us.
   */

  return (Tutorial2_TapeCalculator__CreateTrue (ILU_NIL,
            ILU_NIL, the_state));
}

  static Tutorial2_Operation
```

```
AddOp (
  Tutorial2_RegisterTape *tape,
  Tutorial2_OpType t,
  CORBA_double value,
  CORBA_double acc)
{
  Tutorial2_Operation n;

  n.op = t;
  n.value = value;
  n.accumulator = acc;
  Tutorial2_RegisterTape_Append (tape, &n);
}

  static Tutorial2_RegisterTape *
CopyTape (
  Tutorial2_RegisterTape *orig)
{
  unsigned long len = orig->_length;
  Tutorial2_Operation *p = orig->_buffer;
  Tutorial2_Operation *newops = ILU_NIL;

  if (len > 0)
    {
      newops = (Tutorial2_Operation *)
        ilu_must_malloc(sizeof(struct Tutorial2_Operation) * orig->_length);

      while (len-- > 0)
        newops[len] = (orig->_buffer)[len];
    }

  return (Tutorial2_RegisterTape_Create (orig->_length, newops));
}

/* Now to implement the method, we simply take the true prototype
 * and add whatever code is necessary to actually perform the operation.
 */

  void
server_Tutorial2_TapeCalculator_SetValue (
  Tutorial2_TapeCalculator self,
  CORBA_double v,
  CORBA_Environment *env)
{
  /* The user data field is available as the field "void *data" of
   * any object instance, so we'll just set it to be "v".
   */

  struct calc_state *s = ((struct calc_state *)(self->data));
```

```
  s->the_value = v;
  AddOp (s->tape, Tutorial2_SetValue, v, s->the_value);
}

   CORBA_double
server_Tutorial2_TapeCalculator_GetValue (
   Tutorial2_TapeCalculator self,
   CORBA_Environment *env)
{
  return (((struct calc_state *) (self->data))->the_value);
}

  void
server_Tutorial2_TapeCalculator_Add (
  Tutorial2_TapeCalculator self,
  CORBA_double v,
  CORBA_Environment *env)
{
  struct calc_state *s = ((struct calc_state *)(self->data));

  s->the_value += v;
  AddOp (s->tape, Tutorial2_Add, v, s->the_value);
}

  void
server_Tutorial2_TapeCalculator_Subtract (
  Tutorial2_TapeCalculator self,
  CORBA_double v,
  CORBA_Environment *env)
{
  struct calc_state *s = ((struct calc_state *)(self->data));

  s->the_value -= v;
  AddOp (s->tape, Tutorial2_Subtract, v, s->the_value);
}

  void
server_Tutorial2_TapeCalculator_Multiply (
  Tutorial2_TapeCalculator self,
  CORBA_double v,
  CORBA_Environment *env)
{
  struct calc_state *s = ((struct calc_state *)(self->data));

  s->the_value *= v;
  AddOp (s->tape, Tutorial2_Multiply, v, s->the_value);
}

/* The Divide method gets a little trickier.  We have to compare the
 * value "v" to zero, which for floating point values actually means
```

```
 * comparing it to some epsilon to see whether it is less than that
 * epsilon, and then if it is "zero" we need to signal an error, by
 * "raising" the "DivideByZero" exception.  The way of raising exceptions
 * in ILU C is rather clumsy, so we'll define a macro to make it look
 * prettier.  We also define some macros to make testing the value
 * of "v" a  bit prettier.
 */

#define ABS(x)  (((x)<0)?(-(x)):(x))
#define SOME_EPSILON    0.000000001     /* zero, practically speaking */

#define RAISE(env,exception) { (env)->returnCode=(exception);\
                               (env)->_major=CORBA_USER_EXCEPTION; }

  void
server_Tutorial2_TapeCalculator_Divide (
  Tutorial2_TapeCalculator self,
  CORBA_double v,
  CORBA_Environment *env)
{
  if (ABS(v) < SOME_EPSILON)
    RAISE(env, ex_Tutorial_DivideByZero)
  else
    {
      struct calc_state *s = ((struct calc_state *)(self->data));

      s->the_value /= v;
      AddOp (s->tape, Tutorial2_Divide, v, s->the_value);
    }
}

  Tutorial2_RegisterTape *
server_Tutorial2_TapeCalculator_GetTape (
  Tutorial2_TapeCalculator self,
  CORBA_Environment *env)
{
  return (CopyTape(((struct calc_state *)(self->data))->tape));
}
```

Appendix G Tutorial2.idl

```
#include "Tutorial.idl"

module Tutorial2 {

  enum OpType { SetValue, Add, Subtract, Multiply, Divide };

  struct Operation {
    OpType op;
    double value;
    double accumulator;
  };

  typedef sequence<Operation> RegisterTape;

  // A four function calculator with a register tape
  interface TapeCalculator : Tutorial::Calculator {

    RegisterTape GetTape ();
  };

  // A factory that produces TapeCalculators
  interface Factory : Tutorial::Factory {

    TapeCalculator CreateTapeCalculator ();
  };
};
```

Index of Concepts

D

docstring 4

E

exception-value type 3

F

factory 17

G

generic function 8

H

handle 16

I

import 25
instance 6
instance handle 18
interface 2
ISL 2

K

kernel server 18

M

mapping 6
method 8
methods 2

O

object ID 19
object type 2
object-oriented 2

P

publish 18

R

raises clause 4
register tape 24

S

server ID 18
simple binding 18
subtyping 24
supertype 24
surrogate method 8

T

true method 8
true module 14
type constructors 3

U

unique id 5
user data field 11