What is CORBA?

CORBA (Common Object Request Broker Architecture) is a distributed object-oriented client/server platform.

It includes:

- an object-oriented Remote Procedure Call (RPC) mechanism
- object services (such as the Naming or Trading Service)
- language mappings for different programming languages
- interoperability protocols
- programming guidelines and patterns

CORBA replaces ad-hoc special-purpose mechanisms (such as socket communication) with an open, standardized, scalable, and portable platform.

The Object Management Group (OMG)

The OMG was formed in 1989 to create specifications for open distributed computing.

Its mission is to

> *"… establish industry guidelines and object management specifications to provide a common framework for distributed application development."*

The OMG is the world's largest software consortium with more than 800 member organizations.

Specifications published by the OMG are free of charge. Vendors of CORBA technology do not pay a royalty to the OMG.

Specifications are developed by consensus of interested submitters.

What is Client/Server Computing?

A client/server computing system has the following characteristics:

- A number of clients and servers cooperate to carry out a computational task.
- Servers are passive entities that offer a service and wait for requests from clients to perform that service.
- Clients are active entities that obtain service from servers.
- Clients and servers usually run as processes on different machines (but may run on a single machine or even within a single process).
- Object-oriented client/server computing adds OO features to the basic distribution idea: interfaces, messages, inheritance, and polymorphism.

Advantages and Disadvantages of CORBA

Some advantages:

- vendor-neutral and open standard, portable, wide variety of implementations, hardware platforms, operating systems, languages
- takes the grunt work out of distributed programming

Some disadvantages:

- no reference implementation
- specified by consensus and compromise
- not perfect
- can shoot yourself in the foot and blow the whole leg off…

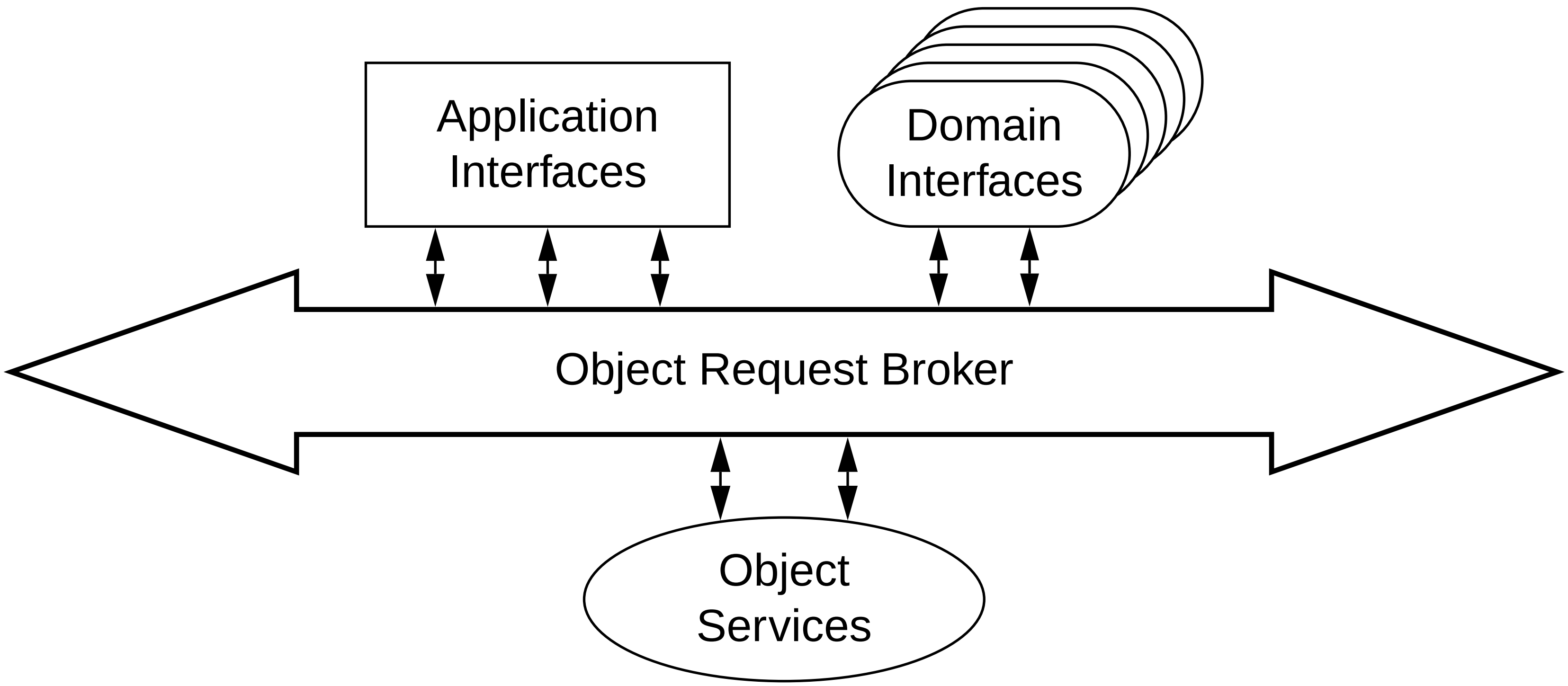
Still, it's the best thing going!

Heterogeneity

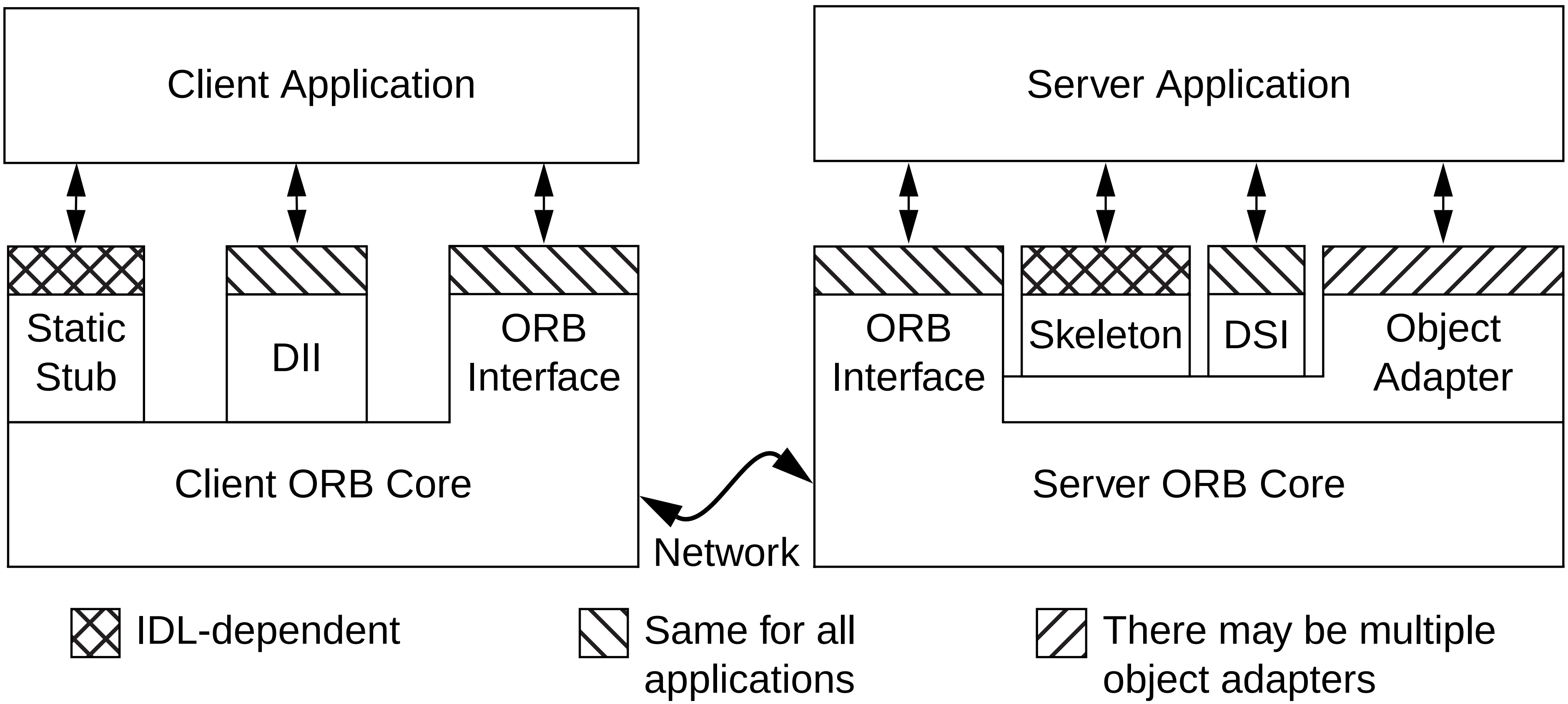
CORBA can deal with homogeneous and heterogeneous environments. The main characteristics to support heterogeneous systems are:

- location transparency
- server transparency
- language independence
- implementation independence
- architecture independence
- operating system independence
- protocol independence
- transport independence

The Object Management Architecture (OMA)



Core Components of an ORB



Request Invocation

Clients invoke requests (send messages) to objects via an object reference. The object reference (IOR) identifies the target object.

When a request is sent by a client, the ORB:

- locates the target object
- activates the server if it is not running
- transmits arguments for the request to the server
- activates the target object (servant) in the server if it is not instantiated
- waits for the request to complete
- returns the results of the request to the client or returns an exception if the request failed

Object Reference Semantics

An object reference is similar to a C++ class instance pointer, but can denote an object in a remote address space.

- Every object reference identifies exactly one object instance.
- Several different references can denote the same object.
- References can be nil (point nowhere).
- References can dangle (like C++ pointers that point at deleted instances).
- References are opaque.
- References are strongly typed.
- References support late binding.
- References can be persistent.

Introduction

IDL specifications separate language-independent interfaces from language-specific implementations.

IDL establishes the interface contract between client and server.

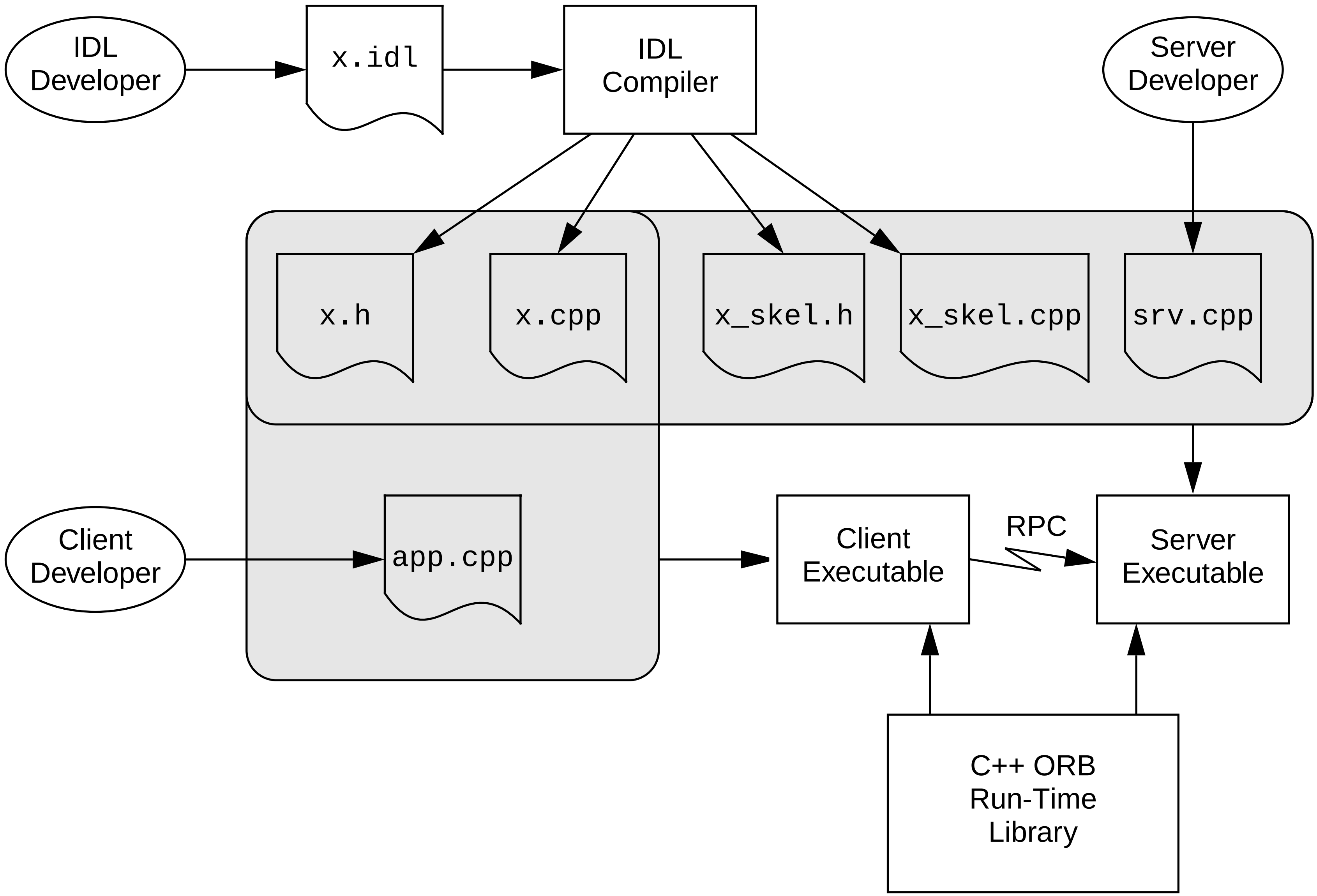
Language-independent IDL specifications are compiled by an IDL compiler into APIs for a specific implementation language.

IDL is purely declarative. You can neither write executable statements in IDL nor say anything about object state.

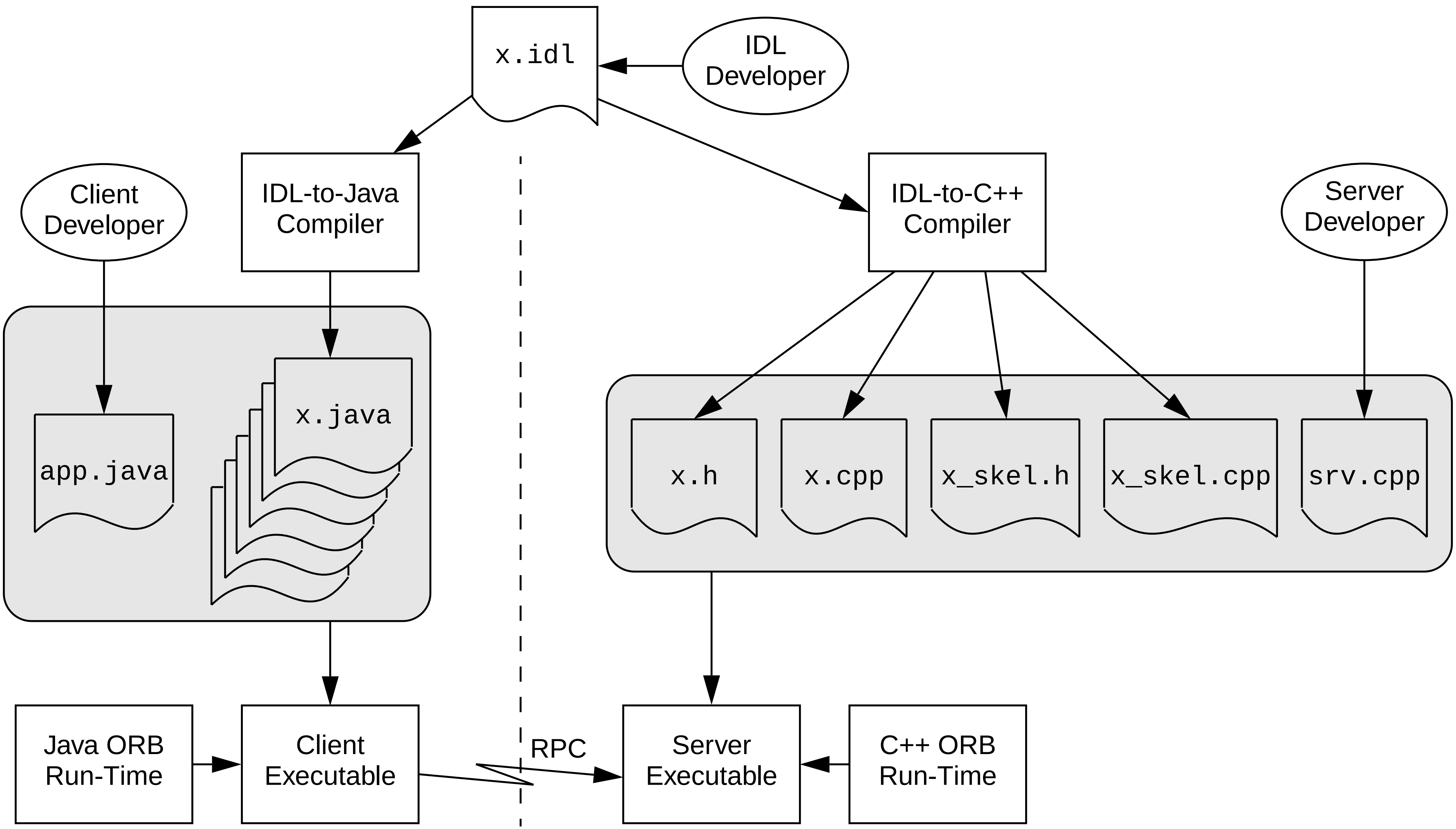
IDL specifications are analogous to C++ type and abstract class definitions. They define types and interfaces that client and server agree on for data exchange.

You can exchange data between client and server only if the data's types are defined in IDL.

IDL Compilation (C++ Language)



IDL Compilation (Mixed Languages)



IDL Source Files

The CORBA specification imposes a number of rules on IDL source files:

- IDL source files must end in a `.idl` extension.
- IDL is a free-form language. You can use white space freely to format your specification. Indentation is not lexically significant.
- IDL source files are preprocessed by the C++ preprocessor. You can use **`#include`**, macro definitions, etc.
- Definitions can appear in any order, but you must follow the “define before use” rule.

Comments and Keywords

- IDL permits both C++-style and C-style comments:

```
/*
 * A C-style comment
 */

// A C++-style comment
```

- IDL keywords are in lower case (e.g. **`interface`**), except for the keywords **`TRUE`**, **`FALSE`**, **`Object`**, and **`ValueBase`**, which must be spelled as shown.

Identifiers

- IDL identifiers can contain letters, digits, and underscores. For example:

  `Thermometer`, **`nominal_temp`**

- IDL identifiers must start with a letter. A leading underscore is permitted but ignored. The following identifiers are treated as identical:

  `set_temp`, **`_set_temp`**

- Identifiers are case-insensitive, so **`max`** and **`MAX`** are the same identifier, but you must use consistent capitalization. For example, once you have named a construct **`max`**, you must continue to refer to that construct as **`max`** (and not as **`Max`** or **`MAX`**).

- Try to avoid identifiers that are likely to be keywords in programming languages, such as **`class`** or **`package`**.

Built-In Types

IDL provides a number of integer and floating-point types:

Type	Size	Range
`short`	≥ 16 bits	-2^{15} to $2^{15}-1$
`unsigned short`	≥ 16 bits	0 to $2^{16}-1$
`long`	≥ 32 bits	-2^{31} to $2^{31}-1$
`unsigned long`	≥ 32 bits	0 to $2^{32}-1$
`long long`	≥ 64 bits	-2^{63} to $2^{63}-1$
`unsigned long long`	≥ 64 bits	0 to $2^{64}-1$
`float`	≥ 32 bits	IEEE single precision
`double`	≥ 64 bits	IEEE double precision
`long double`	≥ 79 bits	IEEE extended precision

Types **`long long`**, **`unsigned long long`**, and **`long double`** may not be supported on all platforms.

Built-In Types (cont.)

CORBA 2.1 added type **`fixed`** to IDL:

```
typedef fixed<9,2>  AssetValue;     // up to 9,999,999.99
                                    // accurate to 0.01
typedef fixed<9,4>  InterestRate;   // up to 99,999.9999,
                                    // accurate to 0.0001
typedef fixed<31,0> BigInt;         // up to 10^31 - 1
```

Fixed-point types have up to 31 decimal digits.

Fixed-point types are not subject to the imprecision of floating-point types.

Calculations are carried out internally with 62-digit precision.

Fixed-point types are useful mainly for monetary calculations.

Fixed-point types are not supported by older ORBs.

Built-In Types (cont.)

IDL provides two character types, **`char`** and **`wchar`**.

- **`char`** is an 8-bit character, **`wchar`** is a wide (2- to 6-byte) character.
- The default codeset for **`char`** is ISO Latin-1 (a superset of ASCII), the codeset for **`wchar`** is 16-bit Unicode.

IDL provides two string types, **`string`** and **`wstring`**.

- A **`string`** can contain any character except NUL (the character with value zero). A **`wstring`** can contain any character except a character with all bits zero.
- Strings and wide strings can be unbounded or bounded:

```
typedef string      City;           // Unbounded
typedef string<3>   Abbreviation;   // Bounded
typedef wstring     Stadt;          // Unbounded
typedef wstring<3>  Abkuerzung;     // Bounded
```

Built-In Types (cont.)

- IDL type **octet** provides an 8-bit type that is guaranteed not to be tampered with in transit. (All other types are subject to translation, such as codeset translation or byte swapping.)

  Type **octet** is useful for transmission of binary data.

- IDL type **boolean** provides a type with values **TRUE** and **FALSE**.
- IDL type **any** provides a universal container type.
  - A value of type **any** can hold a value of any type, such as **boolean**, **double**, or a user-defined type.
  - Values of type **any** are type safe: you cannot extract a value as the wrong type.
  - Type **any** provides introspection: given an **any** containing a value of unknown type, you can ask for the type of the contained value.

Type Definitions

You can use **typedef** to create a new name for a type or to rename an existing type:

```
typedef short       YearType;
typedef short       TempType;
typedef TempType    TemperatureType;    // Bad style
```

You should give each application-specific type a name once and then use that type name consistently.

Judicious use of **typedef** can make your specification easier to understand and more self-documenting.

Avoid needless aliasing, such as **TempType** and **TemperatureType**. It is confusing and can cause problems in language mappings that use strict rules about type equivalence.

Enumerations

You can define enumerated types in IDL:

```
enum Color { red, green, blue, black, mauve, orange };
```

- The type **`Color`** becomes a named type in its own right. (You do not need a **`typedef`** to name it.)
- A type name (such as **`Color`**) is mandatory. (There are no anonymous enumerated types.)
- The enumerators enter the enclosing naming scope and must be unique in that scope:

```
enum InteriorColor { white, beige, grey };
enum ExteriorColor { yellow, beige, green }; // Error!
```

- You cannot control the ordinal values of enumerators:

```
enum Wrong { red = 0, blue = 8 };    // Illegal!
```

```
typedef enum Direction { up, down } DirectionType;  // Bad style!
```

Structures

You can define structures containing one or more members of arbitrary type (including user-defined complex types):

```
struct TimeOfDay {
    short   hour;   // 0 - 23
    short   minute; // 0 - 59
    short   second; // 0 - 59
};
```

- A structure must have at least one member.
- The structure name is mandatory. (There are no anonymous structures.)
- Member names must be unique with the structure.
- Structures form naming scopes.
- Avoid use of **`typedef`** for structures.

```
typedef struct TimeOfDay {
    short   hour;   // 0 - 23
    short   minute; // 0 - 59
    short   second; // 0 - 59
} DayTime;                                  // Bad style!
```

```
struct Outer {
    struct FirstNested {
        long    first;
        long    second;
    } first;

    struct SecondNested {
        long    first;
        long    second;
    } second;
};
```

```
struct FirstNested {
    long    first;
    long    second;
};

struct SecondNested {
    long    first;
    long    second;
};

struct Outer {
    FirstNested     first;
    SecondNested    second;
};
```

Unions

IDL supports discriminated unions with arbitrary member type:

```
union ColorCount switch (Color) {
case red:
case green:
case blue:
    unsigned long   num_in_stock;
case black:
    float           discount;
default:
    string          order_details;
};
```

- A union must have at least one member.
- The type name is mandatory. (There are no anonymous unions.)
- Unions form naming scopes with unique member names.

```
typedef union DateTime switch (boolean) {
case FALSE:
    Date    d;
case TRUE:
    Time    t;
} DateAndTime;                                  // Bad style!
```

```
union BadUnion switch (boolean) {
case FALSE:
    string  member_1;
case TRUE:
    float   member_2;
default:
    octet   member_3;       // Error!
};
```

```
union AgeOpt switch (boolean) {
case TRUE:
    unsigned short age;
};
```

Guidelines for Unions

A few guidelines to make life with unions easier:

- Do not use **`char`** as a discriminator type.
- Do not use unions to simulate type casting.
- Avoid using multiple **`case`** labels for a single union member.
- Avoid using the **`default`** case.
- Use unions sparingly. Often, they are abused to create operations that are like a Swiss army knife.

```
union U switch (char) {
case '~':
    long    long_member;
//...
};
```

```
enum InfoKind { text, numeric, none };

union Info switch (InfoKind) {
case text:
    string  description;
case numeric:
    long    index;
};

interface Order {
    void set_details(in Info details);
};
```

```
interface Order {
    void set_text_details(in string details);
    void set_details_index(in long index);
    void clear_details();
};
```

Arrays

IDL supports single- and multi-dimensional arrays of any element type:

```
typedef Color   ColorVector[10];
typedef string  IdTable[10][20];
```

You must use a **`typedef`** to define array types. The following is illegal:

```
Color ColorVector[10];  // Syntax error!
```

You must specify all array dimensions. Open arrays are not supported:

```
typedef string OpenTable[][20]; // Syntax error!
```

Be careful when passing array *indexes* between address spaces.

```
typedef string      IdVector[20];
typedef IdVector    IdTable[10];
```

Sequences

Sequences are variable-length vectors of elements of the same type.

Sequences can be unbounded (grow to any length) or bounded (limited to a maximum number of elements):

```
typedef sequence<Color>      Colors;
typedef sequence<long, 100> Numbers;
```

The sequence bound must be a non-zero, positive integer constant expression.

You must use a **`typedef`** to define sequence types.

The element type can be any other type, including a sequence type:

```
typedef sequence<Node>          ListOfNodes;
typedef sequence<ListOfNodes>   TreeOfNodes;
```

Sequences can be empty.

```
typedef sequence< sequence<Node> > TreeOfNodes; // Deprecated!
```

Sequences or Arrays?

Sequences and arrays are similar, so here are a few rules of thumb for when to use which:

- If you have a list of things with fixed number of elements, all of which exist at all times, use an array.
- If you require a collection of a varying number of things, use a sequence.
- Use arrays of **char** to model fixed-length strings.
- Use sequences to implement sparse arrays.
- You must use sequences to implement recursive data structures.

```
typedef char ZIPCode[5];
```

```
typedef float   RowType[100];
typedef RowType SquareMatrix[100];

interface MatrixProcessor {
    SquareMatrix invert(in SquareMatrix m);
    // ...
};
```

```
struct CellType {
    float           value;
    unsigned long   col_num;
};
typedef sequence<CellType, 100> RowType;

struct RowInfo{
    RowType         row_vals;
    unsigned long   row_num;
};
typedef sequence<RowInfo, 100> SquareMatrix;
```

Recursive Types

IDL does not have pointers, but still supports recursive data types:

```
struct Node {
    long             value;
    sequence<Node>   children;
};
```

- Recursion is possible only for structures and unions.
- Recursion can be achieved only via a sequence member. The element type of the sequence must be an enclosing structure or union.
- Recursion can span more than one enclosing level.
- Mutual recursion is not supported by IDL.

```
enum OpType {
    OP_AND, OP_OR, OP_NOT,
    OP_BITAND, OP_BITOR, OP_BITXOR, OP_BITNOT
};

enum NodeKind { LEAF_NODE, UNARY_NODE, BINARY_NODE };

union Node switch (NodeKind) {
case LEAF_NODE:
    long    value;
case UNARY_NODE:
    struct UnaryOp {
        OpType              op;
        sequence<Node, 1>   child;
    } u_op;
case BINARY_NODE:
    struct BinaryOp {
        OpType              op;
        sequence<Node, 2>   children;
    } bin_op;
};
```

```
// ...
case BINARY_NODE:
    struct BinaryOp {
        OpType  op;
        Node    children[2];    // Illegal!
    } bin_op;
// ...
```

```
struct TwoLevelRecursive {
    string id;
    struct Nested {
        long                        value;
        sequence<TwoLevelRecursive> children;   // OK
    } data;
};
```

Constants and Literals

You can define a constant of any built-in type (except **any**) or of an enumerated type:

```
const long       FAMOUS_CONST = 42;
const double     SQRT_2 = 1.1414213;
const char       FIRST = 'a';
const string     GREETING = "Gooday, mate!";
const octet      LSB_MASK = 0x01;

typedef fixed<6,4> ExchangeRate;
const ExchangeRate UNITY = 1.0D;

enum Color { ultra_violent, burned_hombre, infra_dead };
const Color      NICEST_COLOR = ultra_violent;
```

Constants must be initialized by a literal or a constant expression.

```
const unsigned short    A = 1;
const long              B = -0234;  // Octal 234, decimal 156
const long long         C = +0x234; // Hexadecimal 234, decimal 564
```

```
const double      A = 3.7e-12;  // integer, fraction, & exponent
const float       B = -2.71;    // integer part and fraction part
const double      C = .88;      // fraction part only
const long double D = 12.;      // integer part only
const double      E = .3E8;     // fraction part and exponent
const double      F = 2E11;     // integer part and exponent
```

```
const fixed f1 = 99D;            // fixed<2,0>
const fixed f2 = -02.71d;        // fixed<3,2>
const fixed f3 = +009270.00D;    // fixed <4,0>
const fixed f4 = 00.009D;        // fixed <4,3>
```

```
const char c1 = 'c';         // the character c
const char c2 = '\007';      // ASCII BEL, octal escape
const char c3 = '\x41';      // ASCII A, hex escape
const char c4 = '\n';        // newline
const char c5 = '\t';        // tab
const char c6 = '\v';        // vertical tab
const char c7 = '\b';        // backspace
const char c8 = '\r';        // carriage return
const char c9 = '\f';        // form feed
const char c10 = '\a';       // alert
const char c11 = '\\';       // backslash
const char c12 = '\?';       // question mark
const char c13 = '\'';       // single quote
```

```
const wchar X = L'X';             // 'X' as a wide character
const wchar OMEGA = L'\u03a9';  // Unicode universal character name
```

```
const string S1 = "Quote: \"";       // string with double quote
const string S2 = "hello world";    // simple string
const string S3 = "hello" " world"; // concatenate
const string S4 = "\xA" "B";        // two characters   \
                                       ('\xA' and 'B'), \
                                       not the single   \
                                       character '\xAB'
const string<5> BS = "Hello";       // Bounded string constant
```

```
const wstring LAST_WORDS = L"My God, it's full of stars!";
const wstring<8> O = L"Omega: \u3A9";
```

```
const boolean CONTRADICTION = FALSE;    // Bad idea...
const boolean TAUTOLOGY     = TRUE;     // Just as bad...
```

```
const octet O1 = 0;
const octet O2 = 0xff;
```

```
enum Color { red, green, blue };

const FavoriteColor = green;
const OtherColor    = ::blue;
```

Constant Expressions

IDL defines the usual arithmetic and bitwise operators for constant expressions:

Operator Type	IDL Operators
Arithmetic	+ - * / %
Bitwise	& | ^ << >> ~

The bitwise operators require integral operands. (IDL guarantees two's complement representation for integral types.)

The operators do *not* have exactly the same semantics as in C++:

- Arithmetic operators do not support mixed-mode arithmetic.
- The `>>` operator *always* performs a logical shift.

```
const short MIN_TEMP = -10;
const short MAX_TEMP = 35;
const short DFLT_TEMP = (MAX_TEMP + MIN_TEMP) / 2;

const float TWO_PIES = 3.14 * 2.0;  // Cannot use 3.14 * 2 here!

const fixed YEARLY_RATE = 0.1875D;
const fixed MONTHLY_RATE = YEARLY_RATE / 12D;   // Cannot use 12 here!
```

```
const long ALL_ONES = -1;                   // 0xffffffff
const long LHW_MASK = ALL_ONES << 16;   // 0xffff0000
const long RHW_MASK = ALL_ONES >> 16;   // 0x0000ffff, guaranteed
```

Interfaces

Interfaces, like C++ class definitions, define object types:

```
interface Thermometer {
    string  get_location();
    void    set_location(in string loc);
};
```

- Invoking an operation on an instance of an interface sends an RPC call to the server that implements the instance.
- Interfaces define a *public* interface. There is no private or protected section for interfaces.
- Interfaces do not have members. Members store state, but state is an implementation (not interface) concern.
- Interfaces define the smallest and only granularity of distribution: for something to be accessible remotely, it must have an interface.

Interface Syntax

You can nest exception, constant, attribute, operation, and type definitions in the scope of an interface definition:

```
interface Haystack {
    exception NotFound { unsigned long num_straws_searched; };

    const unsigned long MAX_SIZE = 1000000;

    readonly attribute unsigned long num_straws;

    typedef long    Needle;
    typedef string  Straw;

    void    add(in Straw s);
    boolean remove(in Straw s);
    boolean find(in Needle n) raises(NotFound);
};
```

Interface Semantics

Interfaces are types and can be used as parameter types (or as a member for data structures). For example:

```
interface FeedShed {
    void    add(in Haystack s);
    void    eat(in Haystack s);
};
```

- The parameters of type **`Haystack`** are object reference parameters.
- Passing an object always passes it by reference.
- The object stays where it is, and the reference is passed by value.
- Invocations on the reference send an RPC call to the server.
- CORBA defines a dedicated nil reference, which indicates no object (points nowhere).

Operation Syntax

Every operation definition has:

- an operation name
- a return type (**`void`** if none)
- zero or more parameter definitions

Optionally, an operation definition may have:

- a **`raises`** expression
- a **`oneway`** operation attribute
- a **`context`** clause

You cannot overload operations because operation names must be unique within the enclosing interface.

```
interface Simple {
    void op();
};
```

```
interface Simple {
    op();    // Error, missing return type
};
```

Operation Example

Here is an interface that illustrates operations with parameters:

```
interface NumberCruncher {
    double square_root(in double operand);
    void   square_root2(in double operand, out double result);
    void   square_root3(inout double op_res);
};
```

- Parameters are qualified with a directional attribute: **`in`**, **`inout`**, or **`out`**.
  - **`in`**: The parameter is sent from the client to the server.
  - **`out`**: The parameter is returned from the server to the client.
  - **`inout`**: The parameter is sent from the client to the server, possibly modified by the server, and returned to the client (overwriting the initial value).

```
interface NumberCruncher {
    double square_root(in double operand);
    void   square_root(in double operand, out double result); // Error
    void   square_root(inout double op_res);                   // Error
};
```

```
boolean get_next(out SomeType next_value);
```

```
while (get_next(value)) {
    // Process value
}
```

User Exceptions

A **`raises`** clause indicates the exceptions that an operation may raise:

```
exception Failed {};
exception RangeError {
    unsigned long   min_val;
    unsigned long   max_val;
};

interface Unreliable {
    void can_fail() raises(Failed);
    void can_also_fail(in long l) raises(Failed, RangeError);
};
```

- Exceptions are like structures but are allowed to have no members.
- Exceptions cannot be nested or be part of inheritance hierarchies.
- Exceptions cannot be members of other data types.

```
exception E1 {};
exception E2 {
    long    value;
    E1      exc;    // Illegal!
};

struct S {
    E1      exc;    // Illegal!
};
```

Using Exceptions Effectively

A few rules of thumb for how to use exceptions:

- Use exceptions only for *exceptional* circumstances.
- Make sure that exceptions carry *useful* information.
- Make sure that exceptions carry *precise* information.
- Make sure that exceptions carry *complete* information.

Sticking to these rules make the resulting APIs easier to use and understand and results in better quality code.

System Exceptions

CORBA defines 35 system exceptions. (The list is occasionally extended.)

- Any operation can raise a system exception.
- System exceptions must not appear in a **`raises`** clause.
- All system exceptions have the same exception body:

```
enum completion_status {
    COMPLETED_YES, COMPLETED_NO, COMPLETED_MAYBE
};

exception <SystemExceptionName> {
    unsigned long       minor;
    completion_status   completed;
};
```

```
enum completion_status {
        COMPLETED_YES, COMPLETED_NO, COMPLETED_MAYBE
};

#define SYSEX(NAME) exception NAME {                    \
                        unsigned long      minor;      \
                        completion_status   completed;  \
                    }

SYSEX(BAD_CONTEXT);              // error processing context object
SYSEX(BAD_INV_ORDER);            // routine invocations out of order
SYSEX(BAD_OPERATION);            // invalid operation
SYSEX(BAD_PARAM);                // an invalid parameter was passed
SYSEX(BAD_TYPECODE);             // bad typecode
SYSEX(CODESET_INCOMPATIBLE);     // incompatible codeset
SYSEX(COMM_FAILURE);             // communication failure
SYSEX(DATA_CONVERSION);          // data conversion error
SYSEX(FREE_MEM);                 // cannot free memory
SYSEX(IMP_LIMIT);                // violated implementation limit
SYSEX(INITIALIZE);               // ORB initialization failure
SYSEX(INTERNAL);                 // ORB internal error
SYSEX(INTF_REPOS);               // interface repository unavailable
SYSEX(INVALID_TRANSACTION);      // invalid TP context passed
SYSEX(INV_FLAG);                 // invalid flag was specified
SYSEX(INV_IDENT);                // invalid identifier syntax
SYSEX(INV_OBJREF);               // invalid object reference
```

```
SYSEX(INV_POLICY);                 // invalid policy override
SYSEX(MARSHAL);                    // error marshaling param/result
SYSEX(NO_IMPLEMENT);               // implementation unavailable
SYSEX(NO_MEMORY);                  // memory allocation failure
SYSEX(NO_PERMISSION);              // no permission for operation
SYSEX(NO_RESOURCES);               // out of resources for request
SYSEX(NO_RESPONSE);                // response not yet available
SYSEX(OBJECT_NOT_EXIST);           // no such object
SYSEX(OBJ_ADAPTER);                // object adapter failure
SYSEX(PERSIST_STORE);              // persistent storage failure
SYSEX(REBIND);                     // rebind needed
SYSEX(TIMEOUT);                    // operation timed out
SYSEX(TRANSACTION_MODE);           // invalid transaction mode
SYSEX(TRANSACTION_REQUIRED);       // operation needs transaction
SYSEX(TRANSACTION_UNAVAILABLE);    // no transaction
SYSEX(TRANSACTION_ROLLEDBACK);     // operation was a no-op
SYSEX(TRANSIENT);                  // transient error, try again later
SYSEX(UNKNOWN);                    // the unknown exception
```

Oneway Operations

IDL permits operations to be declared as **oneway**:

```
interface Events {
    oneway void send(in EventData data);
};
```

The following rules apply to **oneway** operations:

- The must have return type **void**.
- They must not have any **inout** or **out** parameters.
- They must not have a **raises** clause.

Oneway operations provide “best effort” send-and-forget semantics.

Oneway operations may not be delivered, may be dispatched synchronously or asynchronously, and may block.

Oneway is non-portable in CORBA 2.3 and earlier ORBs.

Contexts

Operations can optionally define a **`context`** clause. For example:

```
interface Poor {
    void doit() context("USER", "GROUP", "X*");
};
```

This instructs the client to send the values of the CORBA context variables **USER** and **GROUP** with the call, as well as the value of all context variables beginning with **X**.

Contexts are similar in concept to UNIX environment variables.

Contexts shoot a big hole through the type system!

Many ORBs do not support contexts correctly, so we suggest you avoid them.

Attributes

An interface can contain one or more attributes of arbitrary type:

```
interface Thermostat {
    readonly attribute short temperature;
             attribute short nominal_temp;
};
```

Attributes implicitly define a *pair* of operations: one to send a value and one to fetch a value.

Read-only attributes define a single operation to fetch a value.

Attributes are *not* state or member variables. They are simply a notational short-hand for operations.

Attributes cannot have a **raises** clause, cannot be **oneway**, and cannot have a **context** clause.

If you use attributes, they should be **readonly**.

```
interface Thermostat {
    short get_temperature();
    void  set_nominal_temp(in short temp);
    short get_nominal_temp();
};
```

Modules

IDL modules provide a scoping construct similar to C++ namespaces:

```
module M {
    // ...
    module L {      // Modules can be nested
        // ...
        interface I { /* ... */ };
        // ...
    };
    // ...
};
```

Modules are useful to prevent name clashes at the global scope.

Modules can contain any IDL construct, including other modules.

Modules can be reopened and so permit incremental definition.

```
module A {
    // Some definitions here...
};
module B {
    // Some other definitions here...
};
module A {
    // Reopen module A and add to it...
};
```

Forward Declarations

IDL permits forward declarations of interfaces so they can be mutually dependent:

```
interface Husband;  // Forward declaration

interface Wife {
    Husband get_spouse();
};

interface Husband {
    Wife get_spouse();
};
```

The identifier in a forward declaration must a be a simple (non-qualified) name:

```
interface MyModule::SomeInterface;  // Syntax error!
```

```
module Males {
    interface Husband;
};

module Females {
    interface Wife {
        Males::Husband get_spouse();
    };
};

module Males {
    interface Husband {
        Females::Wife get_spouse();
    };
};
```

Inheritance

IDL permits interface inheritance:

```
interface Thermometer {
    typedef short TempType;
    readonly attribute TempType temperature;
};

interface Thermostat : Thermometer {
    void set_nominal_temp(in TempType t);
};
```

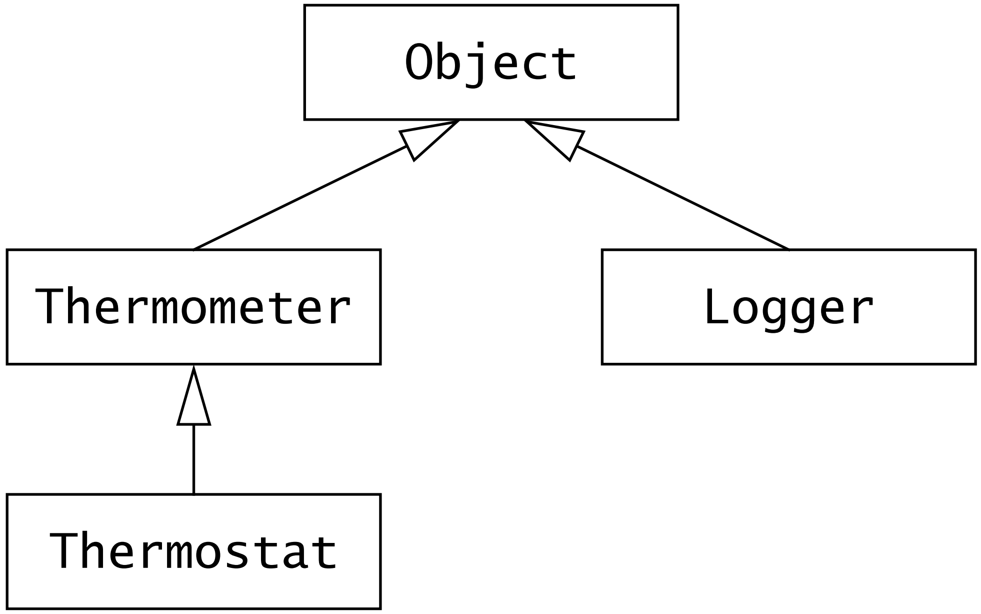
You can pass a derived interface where a base interface is expected:

```
interface Logger {
    long add(in Thermometer t, in unsigned short interval);
    void remove(in long id);
};
```

At run time, you can pass a **Thermometer** or a **Thermostat** to **add**.

Inheritance from Object

All IDL interfaces implicitly inherit from type **`Object`**:



You must not explicitly inherit from type **`Object`**.

Because all interfaces inherit from **`Object`**, you can pass any interface type as type **`Object`**. You can determine the actual type of an interface at run time with a safe down-cast.

```
interface Generic {
    void    accept(in KeyType key, Object o);
    Object  lookup(in KeyType key);
};
```

```
interface Wrong : Object { /*...*/ };    // Error
```

Inheritance Redefinition Rules

You can redefine types, constants, and exceptions in the derived interface:

```
interface Thermometer {
    typedef long    IDType;
    const IDType    TID = 5;
    exception       TempOutOfRange {};
};

interface Thermostat : Thermometer {
    typedef string  IDType;                             // Yuk!
    const IDType    TID= "Thermostat";                  // Aargh!
    exception       TempOutOfRange { long temp; };      // Ick!
};
```

While legal, this is too terrible to even contemplate. Do not do this!

Inheritance Limitations

You cannot override attributes or operations in a derived interface:

```
interface Thermometer {
    attribute long  temperature;
    void            initialize();
};

interface Thermostat : Thermometer {
    attribute long  temperature;        // Error!
    void            initialize();       // Error!
};
```

It is understood that a **`Thermostat`** already has an inherited **`temperature`** attribute and **`initialize`** operation and you are not allowed to restate this.

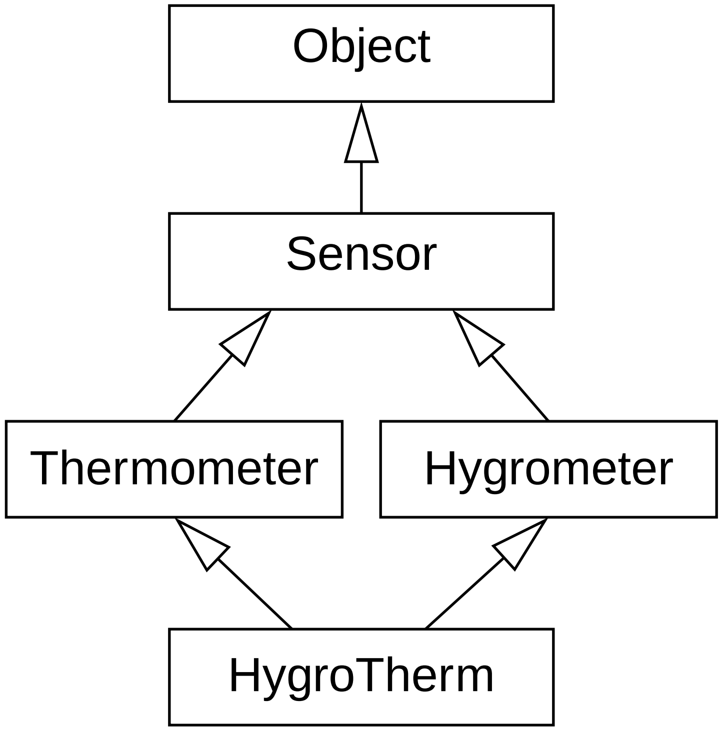
Overriding is a meaningless concept for *interface* inheritance!

```
interface Thermometer {
    attribute string    my_id;
              string    get_id();
              void      set_id(in string s);
};

interface Thermostat : Thermometer {
    attribute double    my_id;                      // Illegal!
              double    get_id();                   // Illegal!
              void      set_id(in double d);        // Illegal!
};
```

Multiple Inheritance

Multiple inheritance, including inheritance of the same base interface multiple times, is supported:

```
interface Sensor {
    // ...
};

interface Thermometer : Sensor {
    // ...
};

interface Hygrometer : Sensor {
    // ...
};

interface HygroTherm : Thermometer, Hygrometer {
    // ...
};
```



Scope Rules for Multiple Inheritance

You cannot inherit the same attribute or operation from more than one base interface:

```
interface Thermometer {
    attribute string    model;
              void      reset();
};

interface Hygrometer {
    attribute string    model;
              string    reset();
};

interface HygroTherm : Thermometer, Hygrometer { // Illegal!
    // ...
};
```

Scope Rules for Multiple Inheritance (cont.)

You can multiply inherit ambiguous types, but you must qualify them explicitly at the point of use:

```
interface Thermometer {
    typedef string<16> ModelType;
};

interface Hygrometer {
    typedef string<32> ModelType;
};

interface HygroTherm : Thermometer, Hygrometer {
    attribute ModelType             model;      // Error!
    attribute Hygrometer::ModelType model;      // OK
};
```

IDL Scope Resolution

The following IDL constructs establish naming scopes:

- modules, interfaces, structures, unions, exceptions, parameter lists

Within a naming scope, names must be unique.

To resolve a name, the compiler searches:

1. the current scope
2. if the current scope is an interface, the base interfaces toward the root
3. enclosing scopes of the current scope
4. the global scope

Names not qualified as being part of the global scope with a leading `::` operator are introduced into the current scope when first used.

```
struct Bad {
    short   temperature;
    long    Temperature;    // Error!
    Temp    temp;           // Error!
};

typedef string SomeType;

interface AlsoBad {
    void op1(in SomeType t, in double t);   // Error!
    void op2(in Temp temp);                 // Error!
    void op3(in sometype t);                // Error!
};
```

```
module CCS {
    typedef short   TempType;
    const TempType  MAX_TEMP = 99;      // Max_TEMP is a short

    interface Thermostat {
        typedef long    TempType;       // OK

        TempType        temperature();  // Returns a long
        CCS::TempType   nominal_temp(); // Returns a short
    };
};
```

```
module Sensors {
    typedef short   TempType;
    typedef string  AssetType;

    interface Thermometer {
        typedef long    TempType;

        TempType    temperature();      // Returns a long
        AssetType   asset_num();        // Returns a string
    };
};

module Controllers {
    typedef double  TempType;

    interface Thermostat : Sensors::Thermometer {
        TempType    nominal_temp();     // Returns a long
        AssetType   my_asset_num();     // Error!
    };
};
```

```
typedef string ModelType;

module CCS {
    typedef short   TempType;
    typedef string  AssetType;
};

module Controllers {
    typedef CCS::TempType       Temperature; // Introduces CCS _only_
    typedef string              ccs;         // Error!
    typedef long                TempType;    // OK
    typedef ::CCS::AssetType    AssetType;   // OK
};

struct Values {
    ::ModelType ModelType;  // OK
    ::modelType ModelType2; // Error!
};
```

```
interface Sensor {
    enum DeviceType { READ_ONLY, READ_WRITE };
};

interface Thermometer : Sensor {
    union ThermData switch (DeviceType) {
    case Sensor::READ_ONLY:
        unsigned long   read_addr;
    case READ_WRITE:                        // Error!
        unsigned long   write_addr;

    };
};
```

Nesting Restrictions

You cannot use directly nested constructs with the same name:

```
module M {
    module X {
        module M { /* ... */ }; // OK
    };
    module M { /* ... */ };     // Error!
};

struct S {
    long    s;                  // Error!
};

interface I {
    void I();                   // Error!
};
```

Anonymous Types

Anonymous types are currently legal in IDL, but CORBA 3.0 will deprecate them.

Anonymous types cause problems for language mappings because they create types without well-defined names.

You should avoid anonymous types in your IDL definitions.

For recursive structures and unions, they cannot be avoided in CORBA 2.3 and earlier versions.

CORBA 2.4 provides a forward declaration for structures and unions, so anonymous types can be avoided completely.

If you name all your types, you will never have a problem!

```
const string<5> GREETING = "Hello";      // Deprecated

interface Deprecated {
    readonly attribute wstring<5> name;  // Deprecated
    wstring<5> op(in wstring<5> param);  // Deprecated
};
typedef sequence<wstring<5> >   WS5Seq;  // Deprecated
typedef wstring<5>              Name[4]; // Deprecated

struct Foo {
    wstring<5> member;                   // Deprecated
};

// Similar for unions and exceptions...
```

```
typedef string<5>   GreetingType;
typedef wstring<5>  ShortWName;

const GreetingType GREETING = "Hello";

interface OK {
    readonly attribute ShortWName name;
    ShortWName op(in ShortWName param);
};
typedef sequence<ShortWName>    WS5Seq;
typedef ShortWName              Name[4];

struct Foo {
    ShortWName member;
};
```

```
interface Account {
    fixed<10,2> balance;    // Deprecated
};
```

```
typedef fixed<10,2> BalanceType;

interface Account {
    BalanceType balance;
};
```

```
exception E {
    long            array_mem[10];  // Deprecated
    sequence<long>  seq_mem;        // Deprecated
    string<5>       bstring_mem;    // Deprecated
};
```

```
typedef long             LongArray[10];
typedef sequence<long>  LongSeq;
typedef string<5>       ShortString;

exception E {
    LongArray   array_mem;
    LongSeq     seq_mem;
    ShortString string_mem;
};
```

```
typedef sequence<sequence<long> > NumberTree;
typedef fixed<10,2>        FixedArray[10];
```

```
typedef sequence<long>            ListOfNumbers;
typedef sequence<ListOfNumbers> NumberTree;
typedef fixed<10,2>               Fixed_10_2;
typedef Fixed_10_2                FixedArray[10];
```

```
struct Node {
    long                val;
    sequence<Node,2>    children;   // Anonymous member type
};
```

```
typedef struct                  Node;       // Forward declaration
typedef sequence<Node,2>    ChildSeq;

struct Node {
    long        val;
    ChildSeq    children;   // Avoids anonymous type
};
```

Repository IDs

The IDL compiler generates a repository ID for each identifier:

```
module M {                          // IDL:M:1.0
    typedef short T;                // IDL:M/T:1.0
    interface I {                   // IDL:M/I:1.0
        attribute T a;              // IDL:M/I/a:1.0
    };
};
```

The repository ID uniquely identifies each IDL type.

You must ensure that repository IDs are unique.

If you have two IDL specifications with the same repository IDs but different meaning, CORBA's type system is destroyed!

Controlling Repository ID Prefixes

You should routinely add a **`#pragma prefix`** to your IDL definitions:

```
#pragma prefix "acme.com"

module M {                              // IDL:acme.com/M:1.0
    typedef short T;                    // IDL:acme.com/M/T:1.0
    interface I {                       // IDL:acme.com/M/I:1.0
        attribute T a;                  // IDL:acme.com/M/I/a:1.0
    };
};
```

`#pragma prefix` adds the specified prefix to each repository ID.

Use of a prefix makes name clashes with other repository IDs unlikely.

```
module acme_com {          // IDL:acme_com:1.0
    module M {             // IDL:acme_com/M:1.0
        // ...
    };
};
```

Predefined IDL

The CORBA specification defines a number of IDL types in the **CORBA** module.

For example, the definition for **TypeCode** (a type that describes types) and the definitions for the Interface Repository (IFR) are in the **CORBA** module.

To use such predefined types in your IDL, you must include **orb.idl**:

```
#include <orb.idl>  // Get access to CORBA module

interface TypeRepository {
    CORBA::TypeCode lookup(in string name);     // OK
    // ...
};
```

Using the IDL Compiler

The IDL compiler is called **`idl`**. By default, for a file called **`x.idl`**, it produces:

`x.h`	client-side header file
`x.cpp`	client-side (stub) source file
`x_skel.h`	server-side header file
`x_skel.cpp`	server-side (skeleton) source file

Major options:

`-D`*`<name>`*`[=`*`<val>`*`]`	define preprocessor symbol *`<name>`* [with value *`<val>`*]
`-U`*`<name>`*	undefine preprocessor symbol
`-I`*`<dir>`*	add directory to include search path
`-E`	run the preprocessor only
`--c-suffix `*`<s>`*	change default `cpp` extension to *`<s>`*
`--h-suffix `*`<s>`*	change default h extension to *`<s>`*
`--impl`	generate implementation files

```
$ idl x.idl y.idl
```

Topics Not Covered Here

There are a few parts of IDL we did not cover in this unit:

- **`#pragma ID`**

  This pragma allows you to selectively change the repository ID for a particular type.

- **`#pragma version`**

  This pragma allows you to change the version number for **`IDL:`** format repository IDs.

- Objects By Value (OBV)

  OBV provides a hybrid of structures with inheritance and objects that are passed by value instead of by reference.

  OBV is large and complex and covered in a separate unit.

The Climate Control System

The climate control system consists of:

- Thermometers

  Thermometers are remote sensing devices that report the temperature at various location.

- Thermostats

  Thermostats are like thermometers but also permit a desired temperature to be selected.

- A single control station

  A control station permits an operator to monitor all devices and to change the temperature in various parts of a building remotely.

The devices in the system use a proprietary instrument control protocol. We need a CORBA interface to this system.

Thermometers

Thermometers are simple devices that report the temperature and come with a small amount of non-volatile memory that stores additional information:

- Asset number (read-only)

  Each thermometer has a unique asset number, assigned when it is manufactured. This number also serves as the device's network address.

- Model (read-only)

  Each thermometer can report its model (such as “Sens-A-Temp”).

- Location (read/write)

  Each thermometer has non-volatile RAM that can be set to indicate the device's location (such as “Annealing Oven 27” or “Room 414").

Thermostats

Thermostats are like thermometers:

- They can report the temperature, and have an asset number, model, and location.
- The asset numbers of thermometers and thermostats do not overlap. (No thermostat has the same asset number as a thermometer).
- Thermostats permit remote setting of a nominal temperature.
- The CCS attempts to keep the actual temperature as close as possible to the nominal temperature.
- The nominal temperature has a lower and upper limit to which it can be set.
- Different thermostats have different temperature limits (depending on the model).

The Monitoring Station

The monitoring station provides central control of the system. It can:

- read the attributes of any device
- list the devices that are connected to the system
- locate devices by asset number, location, or model
- update a number of thermostats as a group by increasing or decreasing the current temperature setting relative to the current setting

```
#pragma prefix "acme.com"

module CCS {
    typedef unsigned long   AssetType;
    typedef string          ModelType;
    typedef short           TempType;
    typedef string          LocType;

    interface Thermometer {
        readonly attribute ModelType    model;
        readonly attribute AssetType    asset_num;
        readonly attribute TempType     temperature;
                 attribute LocType      location;
    };

    interface Thermostat : Thermometer {
        struct BtData {
            TempType    requested;
            TempType    min_permitted;
            TempType    max_permitted;
            string      error_msg;
        };
        exception BadTemp { BtData details; };

        TempType    get_nominal();
        TempType    set_nominal(in TempType new_temp)
```

```
                    raises(BadTemp);
};

interface Controller {
    typedef sequence<Thermometer>   ThermometerSeq;
    typedef sequence<Thermostat>    ThermostatSeq;

    enum SearchCriterion { ASSET, LOCATION, MODEL };

    union KeyType switch(SearchCriterion) {
    case ASSET:
        AssetType   asset_num;
    case LOCATION:
        LocType     loc;
    case MODEL:
        ModelType   model_desc;
    };

    struct SearchType {
        KeyType     key;
        Thermometer device;
    };
    typedef sequence<SearchType>    SearchSeq;

    struct ErrorDetails {
        Thermostat          tmstat_ref;
```

```
            Thermostat::BtData  info;
        };
        typedef sequence<ErrorDetails>  ErrSeq;

        exception EChange {
            ErrSeq  errors;
        };

        ThermometerSeq  list();
        void            find(inout SearchSeq slist);
        void            change(
                            in ThermostatSeq tlist, in short delta
                        ) raises(EChange);
    };
};
```

Introduction

The basic C++ mappings defines how IDL types are represented in C++. It covers:

- mapping for identifiers
- scoping rules
- mapping for built-in types
- mapping for constructed types
- memory management rules

For each IDL construct, the compiler generates a definition into the client-side header file, and an implementation into the client-side stub file.

General definitions (in the **`CORBA`** module) are imported with

```
#include <OB/CORBA.h>
```

Mapping for Identifiers

- IDL identifiers map to corresponding C++ identifiers:

  ```
  enum Color { red, green, blue };
  ```

  The generated C++ contains:

  ```
  enum Color { red, green, blue };
  ```

- IDL identifiers may clash with C++ keywords:

  ```
  enum class { if, this, while, else };
  ```

  Such identifiers are mapped with a **_cxx_** prefix:

  ```
  enum _cxx_class {
      _cxx_if, _cxx_this, _cxx_while, _cxx_else
  };
  ```

  You should avoid using IDL identifiers that are likely to be keywords in one or more programming languages.

Scoping Rules

IDL scopes are preserved in the mapped C++. For example:

```
interface I {
    typedef long L;
};
```

As in IDL, you can refer to the corresponding constructs as **`I`** or **`::I`** and as **`I::L`** or **`::I::L`**.

The specific kind of C++ scope a particular IDL scope maps to depends on the specific IDL construct.

Mapping for Modules

IDL modules map to C++ namespaces:

```
module Outer {
    // More definitions here...
    module Inner {
        // ...
    };
};
```

This maps to:

```
namespace Outer {
    // More definitions here...
    namespace Inner {
        // ...
    }
}
```

```
module M1 {
    // Some M1 definitions here...
};
module M2 {
    // M2 definitions here...
};
module M1 {
    // More M1 definitions here...
};
```

```
namespace M1 {
    // Some M1 definitions here...
}
namespace M2 {
    // M2 definitions here...
}
namespace M1 {
    // More M1 definitions here...
}
```

Mapping for Built-In Types

IDL built-in types map to type definitions in the **CORBA** namespace:

IDL	C++
`short`	`CORBA::Short`
`unsigned short`	`CORBA::UShort`
`long`	`CORBA::Long`
`unsigned long`	`CORBA::ULong`
`long long`	`CORBA::LongLong`
`unsigned long long`	`CORBA::ULongLong`
`float`	`CORBA::Float`
`double`	`CORBA::Double`
`long double`	`CORBA::LongDouble`

The type definitions are used to hide architecture-dependent size differences.

You should use these type names for portable code.

Mapping for Built-In Types (cont.)

IDL	C++
char	CORBA::Char
wchar	CORBA::WChar
string	char *
wstring	CORBA::WChar *
boolean	CORBA::Boolean
octet	CORBA::Octet
fixed<n,m>	CORBA::Fixed
any	CORBA::Any

CORBA::Fixed and **CORBA::Any** are C++ classes. The remaining types map to C++ native types.

WChar and integer types may not be distinguishable for overloading.

Boolean, **Char**, and **Octet** may all use the same underlying character type.

Overloading on Built-In Types

- Do not overload functions solely on **`CORBA::Char`**, **`CORBA::Boolean`**, and **`CORBA::Octet`**. They may all use the same underlying type.

  ORBacus maps **`CORBA::Boolean`** to **`bool`**, **`CORBA::Char`** to **`char`**, and **`CORBA::Octet`** to **`unsigned char`**.

- Do not overload functions solely on **`CORBA::WChar`** and one of the integer types. With older C++ compilers, **`wchar_t`** may be indistinguishable from an integer type for overloading.

  If you are working exclusively with standard C++ compilers, `wchar_t` is a type in its own right and so does not cause problems.

```
void foo(CORBA::Char param)     { /* ... */ };
void foo(CORBA::Boolean param)  { /* ... */ };  // !!!
void foo(CORBA::Octet param)    { /* ... */ };  // !!!
void foo(CORBA::Short param)    { /* ... */ };
void foo(CORBA::Long param)     { /* ... */ };
void foo(CORBA::WChar param)    { /* ... */ };  // !!!
```

Memory Allocation for Strings

For dynamic allocation of strings, you *must* use the provided functions:

```
namespace CORBA {
    // ...
    char *  string_alloc(ULong len);
    char *  string_dup(const char *);
    void    string_free(char *);
    WChar * wstring_alloc(ULong len);
    WChar * wstring_dup(const WChar *);
    void    wstring_free(WChar *);
    // ...
};
```

Calling (`w`)**`string_alloc(n)`** allocates **`n+1`** characters!

These functions are necessary for environments with non-uniform memory architectures (such as Windows).

```
char * p = CORBA::string_alloc(5);  // Allocates 6 bytes
strcpy(p, "Hello");                 // OK, "Hello" fits
```

```
char * p = CORBA::string_dup("Hello");
```

Mapping for Constants

Constants map to corresponding constant definitions in C++:

```
const long   ANSWER = 42;
const string NAME = "Deep Thought";
```

This maps to:

```
const CORBA::Long  ANSWER = 42;
const char * const NAME = "Deep Thought";
```

Global constants and constants that are nested in namespaces (IDL modules) are initialized in the header file.

Constants that are defined inside interfaces *may* be initialized in the header file if:

- they are of integral or enumerated type
- the target compiler complies with standard C++

```
interface I {
    const long   ANSWER = 42;
    const string NAME = "Deep Thought";
};
```

```
class I /* ... */ {
    static const CORBA::Long  ANSWER = 42;
    static const char * const NAME;          // "Deep Thought"
};
```

```
class I /* ... */ {
    static const CORBA::Long  ANSWER;   // 42
    static const char * const NAME;     // "Deep Thought"
};
```

```
char * wisdom_array[I::ANSWER];     // Compile-time error
```

```
char ** wisdom_array = new char *[I::ANSWER];   // OK
```

Variable-Length Types

The following types are variable-length:

- strings and wide strings (bounded or unbounded)
- object references
- type **any**
- sequences (bounded or unbounded)

Structures, unions, and arrays can be fixed- or variable-length:

- They are fixed-length if they (recursively) contain only fixed-length members or elements.
- They are variable-length if they (recursively) contain variable-length members or elements.

Variable-length values require the sender to dynamically allocate the value and the receiver to deallocate it.

Example: String Allocation

The callee allocates the string and returns it to the caller:

```
char * getstring()
{
    return CORBA::string_dup(some_message); // Pass ownership
}
```

The caller takes ownership of the string and must deallocate it:

```
{
char * p = getstring(); // Caller becomes responsible
// Use p...
CORBA::string_free(p);  // OK, caller deallocates
}
```

All variable-length types follow this basic pattern.

Whenever a variable-length value is passed from server to client, the server allocates, and the client must deallocate.

_var Types

_var types are smart pointer classes you can use to make memory leaks unlikely.

A **_var** type is initialized with a pointer to dynamic memory. When a **_var** type goes out of scope, its destructor deallocates the memory.

```
class String_var {
public:
    String_var(char * p) { _ptr = p; }
    ~String_var() { CORBA::string_free(_ptr); }
    // etc...
private:
    char * _ptr;
};
```

H	e	l	l	o	\0

The *only* purpose of **_var** types is to “catch” a dynamically-allocated value and deallocate it later. You need not (but should) use them.

```
{
    CORBA::String_var sv(getstring());
    // Use sv...

} // No explicit deallocation required here.
```

C++ Mapping Levels

The IDL compiler generates a pair of types for every variable-length and user-defined complex type, resulting in a low- and high-level mapping:

IDL Type	C++ Type	C++ _var Type
`string`	`char *`	`CORBA::String_var`
`any`	`CORBA::Any`	`CORBA::Any_var`
`interface foo`	`foo_ptr`	`class foo_var`
`struct foo`	`struct foo`	`class foo_var`
`union foo`	`class foo`	`class foo_var`
`typedef sequence<X> foo;`	`class foo`	`class foo_var`
`typedef X foo[10];`	`typedef X foo[10];`	`class foo_var`

- The low level does not use **`_var`** types and you must deal with memory management explicitly.
- The high level provides **`_var`** types as a convenience layer to make memory management less error-prone.

The String_var Class

```
class String_var {
public:
                    String_var();
                    String_var(char * p);
                    String_var(const char * p);
                    String_var(const String_var & s);
                    ~String_var();

    String_var &    operator=(char * p);
    String_var &    operator=(const char * p);
    String_var &    operator=(const String_var & s);

                    operator char *();
                    operator const char *() const;
                    operator char * &();
    // ...
};
```

```
CORBA::String_var s;
cout << "s = \"" << s << "\"" << endl;  // Crash imminent!
```

```
const char * message = "Hello";
// ...

{
    CORBA::String_var s(message);   // Makes a deep copy
    // ...
}   // ~String_var() deallocates its own copy only.

cout << message << endl;            // OK
```

```
CORBA::String_var target;
target = CORBA::string_dup("Hello");    // target takes ownership

CORBA::String_var source;
source = CORBA::string_dup("World");    // source takes ownership

target = source;    // Deallocates "Hello" and takes
                    // ownership of deep copy of "World".
```

```
CORBA::String_var s = CORBA::string_dup("Hello");
cout << "Length of \"" << s << "\" is " << strlen(s) << endl;
```

The `String_var` Class (cont.)

```
class String_var {
public:
    // ...

    char &               operator[](ULong index);
    char                 operator[](ULong index) const;

    const char *         in() const;
    char * &             inout();
    char * &             out();
    char *               _retn();
};

ostream & operator<<(ostream, const CORBA::String_var);
istream & operator>>(istream, CORBA::String_var &);
```

```
CORBA::String_var s = CORBA::string_dup("Hello");
cout << s[4] << endl;
```

```
CORBA::String_var s = ...;
cout << strlen(s) << endl;  // Bad compiler can't handle this...
```

```
CORBA::String_var s = ...;
cout << strlen(s.in()) << endl; // Force explicit conversion
```

```
void read_string(char * & s)
{
    // Read a line of text from a file...
    s = CORBA::string_dup(line_of_text);
}
```

```
char * s;
read_string(s);
cout << s << endl;
CORBA::string_free(s);  // Must deallocate here!
read_string(s);
cout << s << endl;
CORBA::string_free(s);  // Must deallocate here!
```

```
CORBA::String_var s;
read_string(s.out());
cout << s << endl;
read_string(s.out());    // No leak here.
cout << s << endl;
```

`String_var`: Summary

Keep the following rules in mind when using `String_var`:

- Always initialize a **`String_var`** with a dynamically-allocated string or a **`const char *`**.
- Assignment or construction from a **`const char *`** makes a deep copy.
- Assignment or construction from a **`char *`** transfers ownership.
- Assignment or construction from a **`String_var`** makes a deep copy.
- Assignment of a **`String_var`** to a pointer makes a shallow copy.
- The destructor of a **`String_var`** deallocates memory for the string
- Be careful when using string literals with **`String_var`**.

```
{
    CORBA::String_var s1;
    cout << s1 << endl;                         // Bad news!

    char message[] = "Hello";
    CORBA::String_var s2 = message;             // Bad news!

    CORBA::String_var s3 = strdup("Hello"); // Bad news!
}
```

```
const char message[] = "Hello";
{
    CORBA::String_var s = message;  // OK, deep copy
}
cout << message << endl;            // Fine
```

```
{
    char * p = CORBA::string_dup("Hello");
    CORBA::String_var s = p;        // s takes ownership
    // Do not deallocate p here!
    // ...
} // OK, s deallocates the string
```

```
String_var s1 = CORBA::string_dup("Hello");
String_var s2 = s1;
cout << s1 << endl; // Prints "Hello"
cout << s2 << endl; // Prints "Hello"
s1[0] = 'h';
s1[4] = 'O';
cout << s1 << endl; // Prints "hellO"
cout << s2 << endl; // Prints "Hello"
```

```
char * p;
{
    CORBA::String_var s = CORBA::string_dup("Hello");
    p = s;  // Fine, p points at memory owned by s
}
cout << p << endl;  // Disaster!
```

```
char * p = CORBA::string_dup("Hello");
char * q = p;   // Both p and q point at the same string

CORBA::String_var s1 = p;   // Fine, s1 takes ownership
// ...
CORBA::String_var s2 = q;   // Very bad news indeed!
```

```
CORBA::String_var s = "Hello";  // No problem with standard C++,
                                // but a complete disaster with
                                // older compilers!
```

```
CORBA::String_var s = (const char *)"Hello";        // Fine
```

```
CORBA::String_var s = CORBA::string_dup("Hello");   // Fine too
```

Mapping for Fixed-length Structures

IDL structures map to C++ classes with public members. For example:

```
struct Details {
    double          weight;
    unsigned long   count;
};
```

This maps to:

```
struct Details {
    CORBA::double   weight;
    CORBA::ULong    count;
};
```

The generated structure may have additional member functions. If so, they are internal to the mapping and you must not use them.

```
Details d;
d.weight = 8.5;
d.count = 12;
```

```
Details d = { 8.5, 12 };
```

Mapping for Variable-Length Structures

Variable-length structures map to C++ classes with public data members. Members of variable-length type manage their own memory.

```
struct Fraction {
    double  numeric;
    string  alphabetic;
};
```

This maps to:

```
struct Fraction {
    CORBA::Double       numeric;
    OB::StrForStruct    alphabetic; // vendor-specific
};
```

String members behave like a **`String_var`** that is initialized to the empty string.

Never use internal types, such as **`StrForStruct`**, in your code!

```
{
    Fraction f;
    f.numeric = 1.0/3.0;
    f.alphabetic = CORBA::string_dup("one third");
} // No memory leak here
```

```
{
    struct Fraction f1, f2, f3;

    f1.numeric = .5;
    f1.alphabetic = CORBA::string_dup("one half");
    f2.numeric = .25;
    f2.alphabetic = CORBA::string_dup("one quarter");
    f3.numeric = .125;
    f3.alphabetic = CORBA::string_dup("one eighth");

    f2 = f1;                             // Deep assignment
    f3.alphabetic = f1.alphabetic;       // Deep assignment
    f3.numeric = 1.0;
    f3.alphabetic[3] = '\0';             // Does not affect f1 or f2
    f1.alphabetic[0] = 'O';              // Does not affect f2 or f3
    f1.alphabetic[4] = 'H';              // Does not affect f2 or f3
} // Everything deallocated OK here
```

Mapping for Unbounded Sequences

Each IDL sequence type maps to a distinct C++ class.

An unbounded sequence grows and shrinks at the tail (like variable-length vectors).

A **`length`** accessor function returns the number of elements.

A **`length`** modifier function permits changing the number of elements.

Sequences provide an overloaded subscript operator (**`[]`**).

Access to sequence elements is via the subscript operator with indexes from **`0`** to **`length() - 1`**.

You cannot grow a sequence by using the subscript operator. Instead, you must explicitly increase the sequence length using the **`length`** modifier.

Accessing elements beyond the current length is illegal.

Mapping for Unbounded Sequences (cont.)

```
typedef sequence<string> StrSeq;
```

This maps to:

```
class StrSeq {
public:
                    StrSeq();
                    StrSeq(CORBA::ULong max);
                    StrSeq(const StrSeq &);
                    ~StrSeq();
    StrSeq &        operator=(const StrSeq &);

    CORBA::ULong    length() const;
    void            length(CORBA::ULong newlen);
    CORBA:ULong     maximum();

    OB::StrForSeq   operator[](CORBA::ULong idx);
    const char *    operator[](CORBA::ULong idx) const;
    // ...
};
```

Example: Using a String Sequence

```
StrSeq s(5);                                // Maximum constructor
assert(s.length() == 0);                    // Sequences start off empty

s.length(4);                                // Create four empty strings
assert(s[0] && *s[0] == '\0');              // New strings are empty

for (CORBA::ULong i = 0; i < 4; ++i)
    s[i] = CORBA::string_dup(argv[i]);      // Assume argv has four elmts

s.length(2);                                // Lop off last two elements
assert(s.length() == 2);

for (CORBA::ULong i = 2; i < 8; ++i) {      // Assume argv has eight elmts
    s.length(i + 1);                        // Grow by one element
    s[i] = CORBA::string_dup(argv[i]);      // Last three iterations may
                                            // cause reallocation
}
for (CORBA::ULong i = 0; i < 8; ++i)
    cout << s[i] << endl;                   // Show elements
```

Using Complex Element Types

If a sequence contains complex elements, such as structures, the usual deep copy semantics apply:

- Assignment or copying of sequences makes a deep copy.
- Assignment or copying of sequence elements makes a deep copy.
- Extending a sequence constructs the elements using their default constructor.
- Truncating a sequence (recursively) releases memory for the truncated elements.
- Destroying a sequence (recursively) releases memory for the sequence elements and the sequence.

```
struct Fraction {
    double  numeric;
    string  alphabetic;
};
typedef sequence<Fraction> FractSeq;
```

```
FractSeq fs1;
fs1.length(1);
fs1[0].numeric = 1.0;
fs1[0].alphabetic = CORBA::string_dup("One");
FractSeq fs2 = fs1;     // Deep copy
assert(fs2.length() == 1);
fs2.length(2);
fs2[1] = fs1[0];        // Deep copy
```

Mapping for Bounded Sequences

Bounded sequences have a hard-wired maximum:

```
typedef sequence<string,5> StrSeq;
```

This maps to:

```
class StrSeq {
public:
                    StrSeq();
                    StrSeq(const StrSeq &);
                    ~StrSeq();
    StrSeq &        operator=(const StrSeq &);

    CORBA::ULong    length() const;
    void            length(CORBA::ULong newlen);
    CORBA:ULong     maximum();
    OB::StrForSeq   operator[](CORBA::ULong idx);
    const char *    operator[](CORBA::ULong idx) const;
    // ...
};
```

Rules for Safe Use of Sequences

Keep the following in mind when using sequences:

- Never point at sequence elements or keep references to them.

  If the sequence relocates in memory, the pointers or references will dangle.

- Never subscript beyond the current length.

  The behavior is undefined if you read or write an element beyond the current length. Most likely, you will corrupt memory.

- Do not assume that sequence elements are adjacent in memory.

  Never perform pointer arithmetic on pointers to sequence elements. The results are undefined.

Mapping for Arrays

IDL arrays map to C++ arrays. For example:

```
typedef string  NameList[10];
typedef long    ScoreTable[3][2];
```

This maps to:

```
typedef OB::StrForStruct    NameList[10];
typedef OB::StrForStruct    NameList_slice;

typedef CORBA::Long         ScoreTable[3][2];
typedef CORBA::Long         ScoreTable_slice[2];
```

The slice type of an array is the element type of an array or, for a multi-dimensional array, the element type of the outermost dimension.

This means that an `<array>_slice *` is of type “pointer to element”.

```
NameList nl;                                    // Ten empty strings
for (int i = 0; i < 10 && i < argc; ++i)
    nl[i] = CORBA::string_dup(argv[i]);
nl[0] = nl[1];                                  // Deep copy

ScoreTable st;                                  // Six undefined scores
st[0][0] = 99;                                  // Initialize one score
```

Array Assignment and Allocation

For each array, the compiler generates functions to allocate, allocate and copy, deallocate, and assign arrays:

```
NameList_slice *    NameList_alloc();
NameList_slice *    NameList_dup(const NameList_slice *);
void                NameList_free(NameList_slice *);
void                NameList_copy(
                        const NameList_slice *  from,
                        NameList_slice *        to
                    );

ScoreTable_slice *  ScoreTable_alloc();
ScoreTable_slice *  ScoreTable_dup(const ScoreTable_slice *);
void                ScoreTable_free(ScoreTable_slice *);
void                ScoreTable_copy(
                        const ScoreTable_slice *    from,
                        ScoreTable_Slice *          to
                    );
```

```
// Allocate and initialize an array
//
NameList_slice * nlp = NameList_alloc();
for (int i = 0; i < sizeof(nlp) / sizeof(*nlp); ++i)
    nlp[i] = CORBA::string_dup("some name");

// Create copy of nlp
//
NameList_slice * nlp2 = NameList_dup(nlp);

// Clean up
//
NameList_free(nlp);
NameList_free(nlp2);
```

```
typedef string TwoNames[2];
typedef string FiveNames[5];
```

```
FiveNames fn;
// Initialize fn...
TwoNames_slice * tnp = FiveNames_dup(fn);   // Bad news!
```

```
FiveNames first_five, last_five;
// Initialize...

// The last will be the first...
FiveNames_copy(last_five, first_five);
```

Mapping for Unions

IDL unions map to C++ classes of the same name:

- For each union member, the class has a modifier and accessor function with the name of the member.
- If a union member is of complex type, a third overloaded member function permits in-place modification of the active member.
- Every union has an overloaded member function **_d** which is used to get and set the discriminator value.
- The default constructor of a union performs no application-visible initialization.
- You activate a union member *only* by initializing it with its modifier function.

```
union U switch (char) {
case 'L':
    long    long_mem;
case 'c':
case 'C':
    char    char_mem;
default:
    string  string_mem;
};
```

```
class U {
public:
                    U();
                    U(const U &);
                    ~U();
    U &             operator=(const U &);

    CORBA::Char     _d();
    void            _d(CORBA::Char);

    CORBA::Long     long_mem() const;
    void            long_mem(CORBA::Long);
    CORBA::Char     char_mem() const;
    void            char_mem(CORBA::Char);
    const char *    string_mem() const;
    void            string_mem(char *);
    void            string_mem(const char *);
    void            string_mem(const CORBA::String_var &);
};
```

```
U my_u;                            // my_u is not initialized
my_u.long_mem(99);                 // Activate long_mem
assert(my_u._d() == 'L');          // Verify discriminator
assert(my_u.long_mem() == 99);     // Verify value
```

```
// Deactivate long_mem, activate char_mem
//
my_u.char_mem('X');
assert(my_u.char_mem() == 'X');

// The discriminator is now either 'C' or 'c',
// but we don't know which...
//
assert(my_u._d() == 'c' || my_u._d() == 'C');

my_u._d('C');   // Now the discriminator is definitely 'C'
```

```
my_u.char_mem('Z');      // Activate/assign char_mem
assert(my_u._d() == 'c' || my_u._d() == 'C');

my_u._d('C');   // OK
my_u._d('c');   // OK too, doesn't change active member
my_u._d('X');   // Undefined behavior, would activate string_mem
```

```
// Activate string_mem
//
my_u.string_mem(CORBA::string_dup("Hello"));

// Discriminator value is now anything except 'c', 'C', or 'L'
//
assert(my_u._d() != 'c' && my_u._d() != 'C' && my_u._d() != 'L');

// Now the discriminator has the value 'A'
//
my_u._d('A');   // OK, consistent with active member
```

```
if (my_u._d() != 'c' && my_u._d() != 'C' && my_u._d() != 'L') {
    // string_mem is active
    CORBA::String_var s = my_u.string_mem();
    cout << "member is " << s << endl;
} // s will deallocate the string
```

Mapping for Unions (cont.)

It is easiest to use a **switch** statement to access the correct member:

```
switch (my_u._d()) {
case 'L':
    cout << "long_mem: " << my_u.long_mem() << endl;
    break;
case 'c':
case 'C':
    cout << "char_mem: " << my_u.char_mem() << endl;
    break;
default:
    cout << "string_mem: "
         << my_u.string_mem() << endl;
    break;
}
```

Mapping for Unions (cont.)

A union without a **`default`** label has an extra member function called **`_default`**:

```
union AgeOpt switch (boolean) {
case TRUE:
    unsigned short age;
};
```

The generated class contains:

```
class AgeOpt {
public:
    // ...
    void _default();    // Sets discriminator to FALSE
};
```

`_default` picks a discriminator value that is not used by any of the explicit case labels of the union.

```
AgeOpt age; // Nothing is initialized
age._default(); // Sets discriminator to FALSE
assert(age._d() == 0);
```

```
AgeOpt age; // Nothing is initialized
age._d(0);  // Illegal!
```

Mapping for Unions (cont.)

Unions with members that are sequences, structures, unions, a fixed-point type or of type **any** contain a referent function:

```
typedef sequence<long> LongSeq;
union U switch (long) {
case 0:
    LongSeq ls;
};
```

The generated C++ contains:

```
class U {
public:
    const LongSeq & ls() const;             // Accessor
    void            ls(const LongSeq &);    // Modifier
    LongSeq &       ls();                   // Referent

    // Other member functions here...
};
```

```
LongSeq ls;                    // Empty sequence
U my_u;                        // Uninitialized union
my_u.ls(ls);                   // Activate sequence member
LongSeq & lsr = my_u.ls();  // Get reference to sequence member
lsr.length(max);               // Create max elements

// Fill the sequence inside the union,
// instead of filling the sequence first
// and then having to copy it into the
// union member.
//
for (int i = 0; i < max; ++i)
    lsr[i] = i;
```

Using Unions Safely

A few rules for using unions safely:

- Avoid multiple **`case`** labels for a single member.
- Avoid the **`default`** label.
- Never access a union member that is inconsistent with the discriminator value.
- Only set the discriminator value if a member is already active and only set it to a value that is consistent with that member.
- To deactivate all members, use **`_default`**.
- Do not assume that union members will overlay each other memory.
- Members are activated by their copy constructor.
- Do not rely on side effects from the destructor.

Mapping for `typedef`

IDL **`typedef`** maps to a corresponding C++ **`typedef`**.

Note that aliases are preserved:

```
typedef short   TempType;
typedef string  LocType;
typedef LocType LocationType;
```

The corresponding C++ is:

```
typedef CORBA::Short           TempType;

typedef char *                 LocType;
typedef CORBA::String_var      LocType_var;

typedef LocType                LocationType;
typedef LocType_var            LocationType_var;
```

Type any: Concepts

A value of type **any** contains a pair of values internally:

- a **TypeCode** that describes the type of the value in the **any**
- the actual value

`CORBA::TypeCode` Describing the Value
Actual Value

The **TypeCode** inside an **any** is used to enforce type safety. Extraction of a value succeeds only if it is extracted as the correct type.

During marshaling, the **TypeCode** precedes the value on the wire, so the receiving end knows how to interpret the bit pattern that constitutes the value.

Applications of Type any

Type any is useful if you cannot determine the types you will have to use at compile time. This permits generic interfaces:

```
interface ValueStore {
    void    put(in string value_name, in any value);
    any     get(in string value_name);
};
```

You can also use this to implement variable-length parameter lists:

```
struct NamedValue {
    string  name;
    any     value;
};
typedef sequence<NamedValue> ParamList;

interface Foo {
    void op(in ParamList pl);
};
```

Mapping for Type any

IDL **any** maps to a class **CORBA::Any**:

```
class Any {
public:
            Any();
            Any(const Any &);
            ~Any();
    Any &   operator=(const Any &);

    // ...
};
```

The constructor constructs an **Any** containing no value.

The usual deep copy semantics apply to the copy constructor and the assignment operator.

Mapping for Type any (cont.)

Built-in types are inserted using overloaded **<<=** operators in the CORBA namespace:

```
namespace CORBA {
    // ...
    void operator<<=(CORBA::Any &, Short);
    void operator<<=(CORBA::Any &, UShort);
    void operator<<=(CORBA::Any &, Long);
    void operator<<=(CORBA::Any &, ULong);
    void operator<<=(CORBA::Any &, LongLong);
    // More insertion operators for other types here...
    // ...
};
```

Each insertion operator inserts the value and sets the type code of the **Any** as a side effect.

Note that string insertion makes a deep copy.

```
CORBA::Any a;
CORBA::UShort us = 99;
a <<= us;                // Insert 99 as an unsigned short
a <<= "Hello";           // Insert deep copy of "Hello"
a <<= (CORBA::Double)3; // Deallocate "Hello", insert 3.0
```

```
a <<= 99;                  // Dubious!
a <<= (CORBA::Short)99; // Much better
```

Mapping for Type any (cont.)

Extraction uses overloaded **>>=** operators:

```
namespace CORBA {
    // ...
    Boolean operator>>=(const CORBA::Any &, Short &);
    Boolean operator>>=(const CORBA::Any &, UShort &);
    Boolean operator>>=(const CORBA::Any &, Long &);
    Boolean operator>>=(const CORBA::Any &, ULong &);
    Boolean operator>>=(const CORBA::Any &, LongLong &);
    // More extraction operators for other types here...
    // ...
};
```

Each operator returns true if the extraction succeeds.

Extraction succeeds only if the type code in the **Any** matches the type as which a value is being extracted.

```
CORBA::Any a;
a <<= (CORBA::Long)99;

CORBA::Long long_val;
CORBA::ULong ulong_val;

if (a >>= long_val)            // This must succeed
    assert(long_val == 99); // We know that we put 99 in there...
if (a >>= ulong_val)
    abort();                // Badly broken ORB!
```

Mapping for Type any (cont.)

Insertion and extraction of char, boolean, and octet require use of a helper type:

```
CORBA::Any a;
a <<= CORBA::Any::from_boolean(0);  // Insert false
a <<= CORBA::Any::from_char(0);     // Insert NUL
a <<= CORBA::Any::from_octet(0);    // Insert zero byte

CORBA::Boolean  b;
CORBA::Char     c;
CORBA::Octet    o;
if (a >>= CORBA::Any::to_boolean(b)) {
    cout << "Boolean: " << b << endl;
} else if (a >>= CORBA::Any::to_char(c)) {
    cout << "Char: '\\" << setw(3) << setfill('0') << oct
         << (unsigned)c << "\\'" << endl;
} else if (a >>= CORBA::Any::to_octet(o)) {
    cout << "Octet: " <<
}
```

```
CORBA::Any a;
CORBA::Char c = 'X';
a <<= CORBA::Any::from_boolean(c);  // Oops!
// ...
a >>= CORBA::Any::to_octet(c);      // Oops!
```

Mapping for Type any (cont.)

Insertion of a string makes a deep copy and sets the type code to indicate an *unbounded* string:

```
CORBA::Any a;
a <<= "Hello";  // Deep copy, inserts unbounded string
```

Extraction of strings is by *constant* pointer:

```
const char * msg;
if (a >>= msg) {
    cout << "Message was: \"" << msg << "\"" << endl;

// Do NOT deallocate the string here!
```

Extraction of strings (as for all other types extracted by pointer) is shallow. (The **Any** continues to own the string after extraction.)

Do not dereference the pointer once the **Any** goes out of scope!

```
const char * p = "Hello";
CORBA::Any a;
a <<= p;                            // Deep copy
a <<= (char *)p;                    // Deep copy too
```

```
CORBA::Any a;
a <<= "Hello";
const char * p;
a >>= p;            // Extract string

cout << "Any contents: \"" << p << "\"" << endl;
a <<= (CORBA::Double)3.14;
cout << "Any contents: \"" << p << "\"" << endl; // Big trouble!
```

Mapping for Type any (cont.)

To insert and extract bounded strings, you must use helper functions:

```
CORBA::Any a;
a <<= CORBA::Any::from_string("Hello", 10);

char * msg;
a >>= CORBA::Any::to_string(msg, 10);
cout << "Message: \"" << msg << "\"" << endl;
```

The bound for extraction must match the bound for insertion.

Do not insert a string with a bound that is less than the string length.

A bound value of zero indicates an unbounded string.

Consuming insertion can be achieved with an additional parameter:

```
CORBA::Any a;
char * p = CORBA::string_dup("Hello");
a <<= CORBA::Any::from_string(p, 0, 1); // a takes ownership
```

```
CORBA::Any a;
a <<= CORBA::Any::from_string("Hello", 3);  // Undefined!
```

Mapping for Type any (cont.)

The IDL compiler generates overloaded operators for each user-defined type:

```
CORBA::Any a;
Color c = blue;     // Assume enumerated type Color is defined
a <<= c;

Color c2;
int ok = (a >>= c2);
assert(ok && c2 == blue);
```

This also works for aliases of simple types, such as **`TempType`**.

Mapping for Type any (cont.)

For structures, unions, and sequences, the compiler generates overloaded insertion and extraction operators:

```
CORBA::Any a;
CCS::Thermostat::BtData btd = ...;  // Structure
a <<= btd;                        // Deep copy

CCS::Thermostat::BtData * btdp      // *Pointer* to struct
    = new CCS::Thermostat::BtData;
a <<= btdp;                       // Consuming insertion
```

- Insertion of a structure makes a deep copy.
- Insertion of a pointer is a consuming insertion.

Extraction is always by pointer to constant data:

```
const CCS::Thermostat::BtData * p;
a >>= p;                                // Shallow extraction
```

```
const CCS::Thermostat::BtData * btdp;
if (a >>= btdp) {
    // It's a BtData structure...
    CCS::Thermostat::BtData copy = *btdp;   // Make copy
    copy.error_msg = another_message;
}
```

Mapping for Type any (cont.)

Arrays are inserted and extracted using generated helper classes called ***<array>*_forany**.

```
typedef long arr10[10]; // IDL
```

Insertion and extraction use the **arr10_forany** helper class:

```
CORBA::Any a;
arr10 aten = ...;
a <<= arr10_forany(aten);
// ...

arr10_forany aten_array;
if (a >>= aten_array) {
    cout << "First element: " << aten_array[0] << endl;
}
```

Insertion makes a deep copy, extraction is shallow.

```
arr10 aten;                         // IDL: typedef long arr10[10];
arr20 atwenty;                      // IDL: typedef long arr20[20];

a <<= arr20_forany(aten);           // Bad news!
a >>= arr10_forany(atwenty);        // Bad news!
```

Using _var Types

The mapping creates a **`_var`** type for every user-defined complex type. For variable-length types, a **`_var`** type behaves like a **`String_var`**:

- Assignment of a pointer to a **`_var`** transfers ownership of memory.
- Assignment of **`_var`** types to each other makes a deep copy.
- Assignment of a **`_var`** to a pointer makes a shallow copy.
- The destructor deallocates the underlying value.
- An overloaded **`->`** operator delegates to the underlying value.
- **`_var`** types have user-defined conversion operators so you can pass a **`_var`** where the underlying value is expected.

As for strings, **`_var`** types are simply smart pointers to help with memory management.

```
struct Person {
    string  name;
    string  birth_date;
};
```

```
{
    Person_var pv = new Person;
    pv->name = CORBA::string_dup("Michi Henning");
    pv->birth_date = CORBA::string_dup("16 Feb 1960");
} // ~Person_var() deallocates here
```

Mapping for Variable-Length _var Types

For a variable-length structure, union, or sequence **T**, the **T_var** type is:

```
class T_var {
public:
                T_var();
                T_var(T *);
                T_var(const T_var &);
                ~T_var();
    T_var &     operator=(T *);
    T_var &     operator=(const T_var &);
    T *         operator->();
    const T *   operator->() const;
                operator T &();
                operator const T &() const;
                // Other members here...
private:
    T * _ptr;
};
```

Example: Simple Use of _var Types

```
// IDL: typedef sequence<string> NameSeq;

NameSeq_var ns;                                 // Default constructor
ns = new NameSeq;                               // ns assumes ownership
ns->length(1);                                  // Create one empty string
ns[0] = CORBA::string_dup("Bjarne");            // Explicit copy

NameSeq_var ns2(ns);                            // Deep copy constructor
ns2[0] = CORBA::string_dup("Stan");             // Deallocates "Bjarne"

NameSeq_var ns3;                                // Default constructor
ns3 = ns2;                                      // Deep assignment
ns3[0] = CORBA::string_dup("Andrew");           // Deallocates "Stan"

cout << ns[0] << endl;                          // "Bjarne"
cout << ns2[0] << endl;                         // "Stan"
cout << ns3[0] << endl;                         // "Andrew"
```

```
{
    NameSeq_var nsv = get_names();  // Assume get_names returns
                                    // a pointer to a dynamically
                                    // allocated sequence...
    // Use nsv...

} // No need to deallocate anything here
```

Mapping for Fixed-Length _var Types

_var types for fixed-length underlying types is almost identical to **_var** type for variable-length underlying types:

- As usual, the pointer constructor adopts the underlying instance.
- An additional constructor from a **T** value deep-copies the value.
- An additional assignment operator from a **T** deep-assigns the value.

The net effect is that **_var** types for both fixed-length and variable-length underlying types provide intuitive deep copy semantics.

Fixed-length **_var** types are provided for consistency with variable-length **_var** types.

_var types hide the memory management difference between fixed-length and variable-length types for operation invocations.

```
// IDL: struct Point { double x; double y; };

Point origin = { 0.0, 0.0 };
Point_var pv1 = origin;            // Deep copy
Point_var pv2 = new Point;         // pv2 takes ownership
pv2 = pv1;                         // Deep assignment
pv1->x = 99.0;                     // Does not affect pv2 or origin
pv2->x = 3.14;                     // Does not affect pv1, or origin
cout << pv1->x << endl;            // 99.0
cout << pv2->x << endl;            // 3.14
cout << origin->x << endl;         // 0.0
```

Dealing with Broken Compilers

Compilers occasionally have problems applying the parameter matching rules correctly when you pass a **`_var`** type to a function.

Both fixed- and variable-length types have additional member functions to get around such problems:

- **`in`**: passes a **`_var`** as an **`in`** parameter
- **`inout`**: passes a **`_var`** as an **`inout`** parameter
- **`out`**: passes a **`_var`** as an **`out`** parameter

Variable-length **`_var`** types have a **`_retn`** member function that return a pointer to the underlying value and transfer ownership.

Fixed-length **`_var`** types have a **`_retn`** member function that returns the underlying value itself. No transfer of ownership takes place in this case.

```
void get_vals(FLT & p1, VLT * & p2);
```

```
FLT_var p1;
VLT_var p2;
get_vals(p1, p2);                    // This may not compile,
get_vals(p1.out(), p2.out());   // but this will.
```

Introduction

The client-side C++ mapping covers:

- Mapping for interfaces and object references
- Mapping for operation invocations and parameter passing rules
- Exception handling
- ORB initialization

This unit also covers how to compile and link a client into a working binary program.

Object References

To make an invocation on an object, the client must have an object reference.

An object reference encapsulates:

- a network address that identifies the server process
- a unique identifier (placed into the reference by the server) that identifies which object in the server a request is for

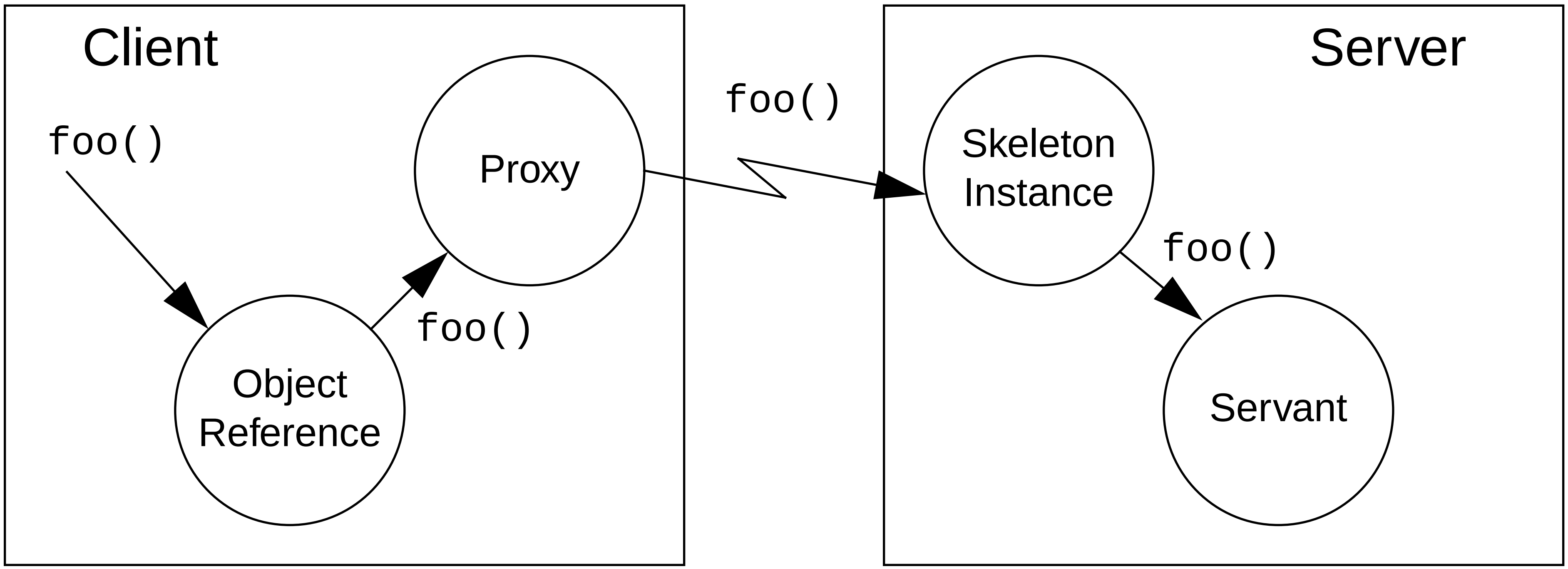
Object references are opaque to the client. Clients cannot instantiate references directly. (The ORB does this for the client.)

Each object reference denotes exactly one object but an object may have more than one reference.

You can think of references as C++ pointers that can point into another address space.

Client-Side Proxies

A client-side invocation on an object reference is forwarded by the reference to a client-side proxy object:



The proxy acts as a local ambassador for the remote object.

Clients control the life cycle of the proxy indirectly via the reference.

Mapping for Interfaces

Interfaces map to abstract base classes:

```
interface MyObject {
    long get_value();
};
```

This generates the following proxy class:

```
class MyObject : public virtual CORBA::Object {
public:
    CORBA::Long get_value();
    // ...
};
```

- For each IDL operation, the class contains a member function.
- The proxy class inherits from **`CORBA::Object`** (possibly indirectly, if the interface is a derived interface).

Never instantiate the proxy class directly!

```
MyObject myobj;      // Cannot instantiate a proxy directly
MyObject * mop;      // Cannot declare a pointer to a proxy
void f(MyObject &); // Cannot declare a reference to a proxy
```

Mapping for Object References

For each interface, the compiler generates two object reference types:

- ***<interface>*_ptr**

  A **_ptr** reference is an unmanaged type that requires you to allocate and deallocate resources explicitly.

- ***<interface>*_var**

  A **_var** reference is a smart type that deallocates resources automatically (similar to **String_var** and other **_var** types).

With either type of reference, you use **->** to call an operation:

```
MyObject_ptr mop = ...;              // Get _ptr reference...
CORBA::Long v = mop->get_value();    // Get value from object

MyObject_var mov = ...;              // Get _var reference...
v = mov->get_value();                // Get value from object
```

```
class MyObject : public virtual CORBA::Object {
    // ...
};

typedef MyObject * MyObject_ptr;

class MyObject_var {
public:
    // ...
private:
    MyObject_ptr _ptr;
};
```

Life Cycle of Object References

Object references can be created and destroyed.

- Clients cannot create references (except for nil references).
- Clients can make a copy of an existing reference.
- Clients can destroy a reference.

The ORB uses the life cycle of references to track when it can reclaim the resources (memory and network connection) associated with a proxy.

Proxies are reference counted. The reference count tracks the number of references that point to a proxy.

Destruction of the last reference to a proxy also destroys the proxy.

Reference Life Cycle Operations

To destroy a reference, you call **`release`** in the **`CORBA`** namespace:

```
namespace CORBA {
    // ...
    void release(Object_ptr);
};
```

Every proxy contains a static **`_duplicate`** member function:

```
class MyObject : public virtual CORBA::Object {
public:
    static MyObject_ptr _duplicate(MyObject_ptr);
    // ...
};
```

`_duplicate` returns a copy of the reference passed as the argument.

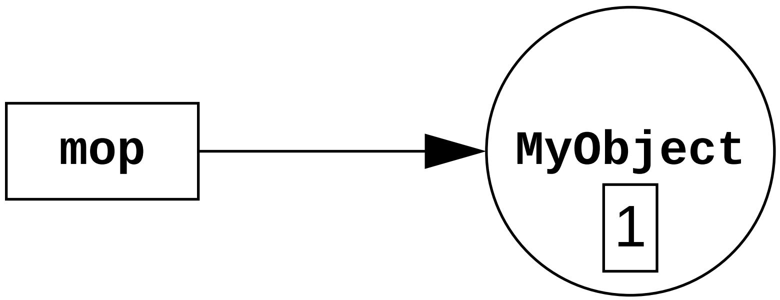
The copy of the reference is indistinguishable from the original.

Object Reference Counts

When the ORB returns a reference to the client, its proxy is always instantiated with a reference count of 1:

```
MyObject_ptr mop = ...;     // Get reference from somewhere...
```

This creates the following situation:



The client invokes operations on the proxy via the reference, for example:

```
CORBA::Long v = mop->get_value();
```

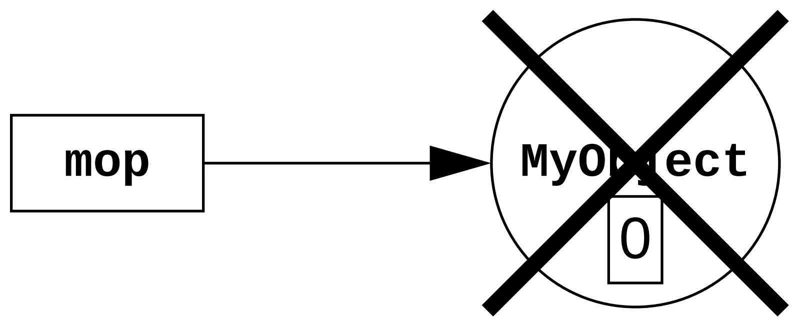
The proxy is kept alive in the client while its reference count is non-zero.

Object Reference Counts (cont.)

The client is responsible for informing the ORB when it no longer wants to use a reference by calling **`release`**:

```
MyObject_ptr mop = ...;                    // Get reference
CORBA::Long v = mop->get_value();          // Use reference
// ...
CORBA::release(mop);                       // No longer interested
                                           // in this object
```

`release` decrements the reference count:



Dropping the reference count to zero causes deallocation.

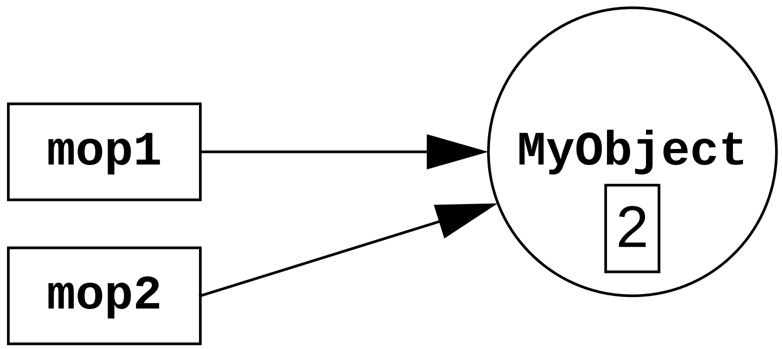
```
MyObject_ptr mop = ...;                // Get reference...
CORBA::Long v = mop->get_value();   // Get a value
CORBA::release(mop);                   // Done with object
v = mop->get_value();                  // Disaster!!!
```

Object Reference Counts (cont.)

`_duplicate` makes a (conceptual) copy of a proxy by incrementing the reference count:

```
MyObject_ptr mop1 = ...;                            // Get ref...
MyObject_ptr mop2 = MyObject::_duplicate(mop1); // Make copy
```

The proxy now looks like:



The client must release each of **`mop1`** and **`mop2`** exactly once to get rid of the proxy.

```
MyObject_ptr mop1 = ...;
MyObject_ptr mop2 = MyObject::_duplicate(mop1);

// Use mop1 and mop2...

CORBA::release(mop1);       // Could release mop2 here
CORBA::release(mop2);       // Could release mop1 here

// Can't use either mop1 or mop2 from here on
```

Scope of Object References

`_duplicate` and **`release`** exist purely to manage resources in the local address space.

If a client calls **`release`** on a reference, the server has no idea that this has happened.

Conversely, if the server calls **`release`** on one of its references, the client has no idea that this has happened.

Calling release *has no effect whatsoever* on anything but the local address space.

You *cannot* implement destruction of objects by calling **`release`** in the client. Instead, you must add an explicit **`destroy`** operation.

Nil References

Every proxy class contains a static **`_nil`** member function. **`_nil`** creates a nil reference:

```
class MyObject : public virtual CORBA::Object {
public:
    static MyObject_ptr _nil();
    // ...
};
```

You can duplicate a nil reference like any other reference.

You can (but need not) release a nil reference.

Do not invoke an operation on a nil reference:

```
MyObject_ptr nil_obj = MyObject::_nil();    // Create nil ref
nil_obj->get_value();                       // Disaster!!!
```

You can test whether a reference is nil by calling **`CORBA::is_nil`**.

```
MyObject_ptr mop = ...;      // Get reference
if (!CORBA::is_nil(mop)) {
    // OK, not nil, we can make a call
    cout << "Value is: " << mop->get_value() << endl;
} else {
    // We got a nil reference, better not use it!
    cout << "Cannot call via nil reference" << endl;
}
```

References and Inheritance

Proxy classes mirror the IDL inheritance structure.

```
interface Thermometer { /* ... */ };
interface Thermostat : Thermometer { /* ... */ };
```

The generated proxy classes reflect the same hierarchy:

```
class CORBA::Object { /* ... */ };
typedef CORBA::Object * Object_ptr;

class Thermometer : public virtual CORBA::Object { /* ... */ };
typedef Thermometer * Thermometer_ptr;

class Thermostat : public virtual Thermometer { /* ... */ };
typedef Thermostat * Thermostat_ptr;
```

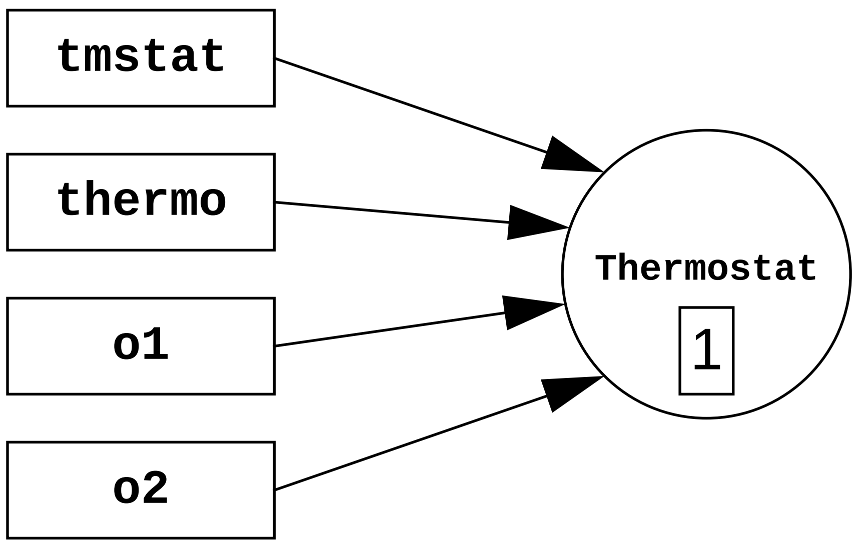
It follows that object references to a derived interface are compatible with object references to a base interface.

Implicit Widening of `_ptr` References

The following is legal code for the CCS:

```
CCS::Thermostat_ptr tmstat = ...;         // Get Thermostat ref
CCS::Thermometer_ptr thermo = tmstat;     // OK, widens
CORBA::Object_ptr o1 = tmstat;            // OK too
CORBA::Object_Ptr o2 = tmstat;            // OK too
```

After these assignments, we have the following situation:



```
CCS::TempType t;
t = tmstat->get_nominal();  // OK
t = thermo->get_nominal();  // Compile-time error
t = o1->get_nominal();      // Compile-time error
```

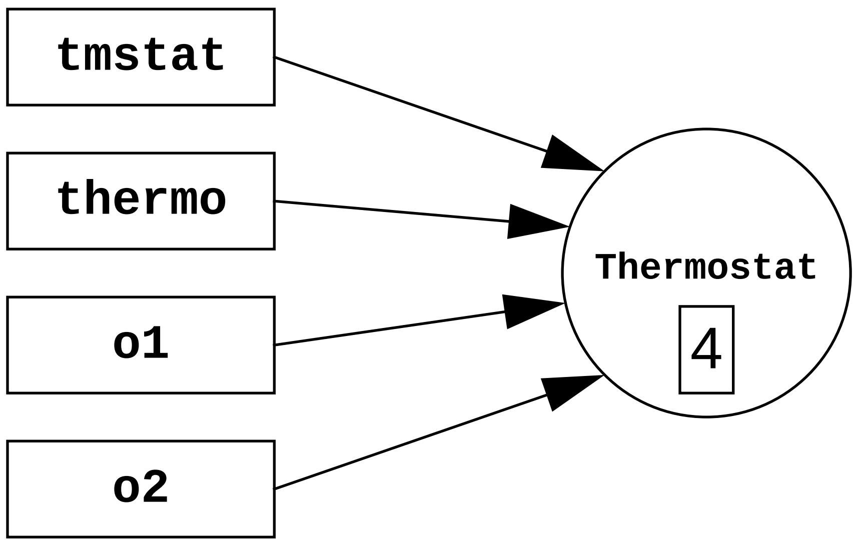
```
CORBA::release(thermo); // or CORBA::release(tmstat)
                        // or CORBA::release(o1)
                        // or CORBA::release(o2)
// Can't use any of the four references from here on...
```

Widening with `_duplicate`

You can also explicitly make duplicates during widening:

```
CCS::Thermostat_ptr tmstat = ...;   // Get reference
CCS::Thermometer_ptr thermo
                        = CCS::Thermometer::_duplicate(tmstat);
CORBA::Object_ptr o1 = CCS::Thermometer::_duplicate(thermo);
CORBA::Object_ptr o2 = CORBA::Object::_duplicate(thermo);
```

The reference count now is 4:



Narrowing Conversion

The compiler generates a static **`_narrow`** member function for each proxy that works like a C++ dynamic cast:

```
class Thermometer : public virtual CORBA::Object {
public:
    // ...
    static Thermometer_ptr _narrow(CORBA::Object_ptr);
};

class Thermostat : public virtual Thermometer {
public:
    // ...
    static Thermostat_ptr _narrow(CORBA::Object_ptr);
};
```

`_narrow` returns a non-nil reference if the argument is of the expected type, nil otherwise. **`_narrow`** implicitly calls **`_duplicate`**!

```
CCS::Thermometer_ptr thermo = ...;  // Get reference
CCS::Thermostat_ptr tstat = thermo; // Compile-time error
```

```
CCS::Thermometer_ptr thermo = ...;                  // Get ref
CCS::Thermostat_ptr tstat1
    = (CCS::Thermostat_ptr)thermo;                  // NO!
CCS::Thermostat_ptr tstat2
    = dynamic_cast<CCS::Thermostat_ptr>(thermo);    // NO!
```

```
CCS::Thermometer_ptr thermo = ...;      // Get reference

// Try down-cast
CCS::Thermostat_ptr tmstat = CCS::Thermostat::_narrow(thermo);
if (CORBA::is_nil(tmstat)) {
    // thermo isn't a Thermostat
} else {
    // thermo is-a Thermostat
    cout << "Nominal temp: " << tmstat->nominal_temp() << endl;
}
CORBA::release(tmstat);             // _narrow calls _duplicate!
```

Illegal Uses of References

The following are illegal and have undefined behavior:

- comparison of references for equality or inequality
- applying relational operators to references
- applying arithmetic operators to references
- conversion of references to and from **`void *`**
- Down-casts other than with **`_narrow`**
- Testing for nil other than with **`CORBA::is_nil`**

Some of these may happen to work and may even do the right thing, but they are still illegal!

```
CORBA::Object_ptr o1 = ...;
CORBA::Object_ptr o2 = ...;
if (o1 == o2)                  // Undefined behavior!
    ...;
if (o1 != o2)                  // Undefined behavior!
    ...;
```

```
CORBA::Object_ptr o1 = ...;
CORBA::Object_ptr o2 = ...;
if (o1 < o2)                    // Undefined behavior!
    ...;                        // <, <=, >, and >= have no meaning
```

```
CORBA::Object_ptr o1 = ...;
CORBA::Object_ptr o2;
o2 = o1 + 5;                    // Meaningless!
ptrdiff_t diff = o2 - o1;   // Meaningless!
```

```
CORBA::Object_ptr o = ...;
void * v = (void *)o;       // Meaningless!
```

```
CCS::Thermostat_ptr tmstat = ...;   // Get reference
CORBA::Object_ptr o = tmstat;       // OK
CCS::Thermostat_ptr tmstat2;

tmstat2 = dynamic_cast<CCS::Thermostat_ptr>(o);         // Bad!
tmstat2 = static_cast<CCS::Thermostat_ptr>(o);          // Bad!
tmstat2 = reinterpret_cast<CCS::Thermostat_ptr>(o);     // Bad!
tmstat2 = (CCS::Thermostat_ptr)o;                       // Bad!

tmstat2 = CCS::thermostat::_narrow(o);                  // OK
```

```
if (tmstat) ...                             // Illegal!
if (tmstat != 0) ...                        // Illegal!
if (tmstat != CCS::Thermostat::_nil()) ...  // Illegal!

if (!CORBA::is_nil(tmstat)) ...             // OK
```

Pseudo Objects and the ORB Interface

The **CORBA** module contains an interface **ORB**:

```
module CORBA {
    interface ORB {      // PIDL
        // ...
    };
    // ...
};
```

The ORB interface is used to initialize the ORB run time and to get access to initial object references.

The **PIDL** comment indicates Pseudo-IDL. PIDL interfaces are implemented as library objects and used to access the ORB run time.

PIDL objects are not fully-fledged objects because they cannot be accessed remotely.

ORB Initialization and Finalization

The ORB interface contains an initialization and finalization operation. You must call these to initialize and clean up the ORB:

```
CORBA::ORB_ptr orb; // Global for convenience

int
main(int argc, char * argv[])
{
    try {
        orb = CORBA::ORB_init(argc, argv);
    } catch (...) {
        cerr << "Cannot initialize ORB" << endl;
        return 1;
    }
    // Use ORB...
    orb->destroy();           // Must destroy!
    CORBA::release(orb);      // Clean up
    return 0;
}
```

```
namespace CORBA {
    // ...
    ORB_ptr ORB_init(
                int &            argc,
                char **          argv,
                const char *     orb_identifier = ""
            );
    // ...
};
```

Stringified References

You can convert an object reference into a string and back:

```
interface ORB {
    string  object_to_string(in Object obj);
    Object  string_to_object(in string str);
    // ...
};
```

Stringified references can be used for bootstrapping:

- The server creates an object and writes its stringified reference to a file.
- The client reads the file and uses the reference to access the object.

While simple, there are drawbacks to this idea. A Naming Service does the same job better.

Stringified references are also useful to store references in databases.

```
#include <OB/CORBA.h>   // Import CORBA module
#include "CCS.h"        // Import CCS system (IDL-generated)

CORBA::ORB_ptr orb;     // Global for convenience

int
main(int argc, char * argv[])
{
    // Initialize the ORB
    orb = CORBA::ORB_init(argc, argv);

    // Get controller reference from argv
    // and convert to object.
    CORBA::Object_ptr obj = orb->string_to_object(argv[1]);
    if (CORBA::is_nil(obj)) {
        cerr << "Nil controller reference" << endl;
        abort();
    }

    // Try to narrow to CCS::Controller.
    CCS::Controller_ptr ctrl;
    ctrl = CCS::Controller::_narrow(obj);
    if (CORBA::is_nil(ctrl)) {
        cerr << "Wrong type for controller ref." << endl;
        abort();
```

```
    }

    // Use controller...

    CORBA::release(ctrl);   // Clean up
    CORBA::release(obj);    // Ditto...
    orb->destroy();         // Must destroy before leaving main()
    CORBA::release(orb);    // Ditto...

    return 0;
}
```

```
#include <OB/CORBA.h>   // Import CORBA module
#include "CCS.h"        // Import CCS system (IDL-generated)

CORBA::ORB_ptr orb = CORBA::ORB::_nil();    // Nil initialized

int
main(int argc, char * argv[])
{
    CCS::Controller_ptr ctrl = CCS::Controller::_nil();

    try {
        // Initialize the ORB
        orb = CORBA::ORB_init(argc, argv);

        // Get controller reference from argv
        // and convert to object.
        CORBA::Object_ptr obj = orb->string_to_object(argv[1]);
        if (CORBA::is_nil(obj)) {
            cerr << "Nil controller reference" << endl;
            abort();
        }

        // Try to narrow to CCS::Controller.
        ctrl = CCS::Controller::_narrow(obj);
        if (CORBA::is_nil(ctrl)) {
```

```
            cerr << "Wrong type for controller ref." << endl;
            abort();
        }

        // Use controller...

    } catch (...) {
        CORBA::release(ctrl);   // Clean up
        CORBA::release(obj);    // Ditto...
        orb->destroy();         //
 Must destroy before leaving main()
        CORBA::release(orb);    // Ditto...
    }

    return 0;
}
```

Stringified References (cont.)

Stringified references are interoperable and can be exchanged among clients and servers using different ORBs.

Nil references can be stringified.

You must treat stringified references as opaque:

- *Never* compare stringified references to determine whether they point at the same object.
- Do not use stringified references as database keys

The *only* things you can legally do with stringified references are:

- obtain them from **`object_to_string`**
- store them for later retrieval
- convert them back to a reference with **`string_to_object`**

Stringified References (cont.)

You can use a URL to denote a file containing a stringified reference:

- **`file:///usr/local/CCS/ctrl.ref`** (UNIX)
- **`file://c:\winnt\Program%20Files\CCS\ctrl.ref`** (NT)

`string_to_object` accepts such URLs as a valid IOR strings and reads the stringified reference from the specified file.

This mechanism is specific to ORBacus!

The Object Interface

The **CORBA** module contains the **Object** interface.

All references provide this interface (because all interfaces inherit from **Object**):

```
interface Object {                                  // PIDL
    Object          duplicate()
    void            release();
    boolean         is_nil();
    boolean         non_existent();
    boolean         is_equivalent(in Object other_object);
    unsigned long   hash(unsigned long max);
    boolean         is_a(in string repository_id);
    // ...
};
```

Note that **Object** is defined in PIDL.

```
class Object {
public:
    // ...
    Boolean _non_existent();
    Boolean _is_equivalent(Object_ptr other_object);
    ULong   _hash(ULong max);
    Boolean _is_a(const char * repository_id);
    // ...
};
```

```
CORBA::Object_ptr obj = ...;        // Get a reference
if (!CORBA::is_nil(obj) {
    if (obj->_is_a("IDL:acme.com/CCS/Controller:1.0")) {
        // It's a controller
    } else {
        // It's something else
    }
} else {
    // Got a nil reference
}
```

Object Reference Equivalence

`is_equivalent` tests if two object references are identical:

- if they are equivalent, the two references denote the same object
- if they are not equivalent, the two references may or may not denote the same object

`is_equivalent` test object *reference* identity, not *object* identity!

Because a single object may have several different references, a false return from **`is_equivalent`** does *not* indicate that the reference denote different objects!

`is_equivalent` is a local operation (never goes out on the wire).

```
CORBA::Object_ptr o1 = ...; // Get reference
CORBA::Object_ptr o2 = ...; // Another one...

if (o1->_is_equivalent(o2)) {
    // o1 and o2 denote the same object
} else {
    // o1 and o2 may or may not denote the same
    // object, who knows...
}
```

Providing Object Equivalence Testing

If you require *object* identity, you must supply it yourself:

```
interface Identity {
    typedef whatever UniqueID;
    UniqueID id();
};
```

You can use this interface as a base interface for your objects.

Clients can invoke the **id** operation to obtain a unique ID for each object.

Two objects are identical if their IDs are identical.

Note that the asset number in the CCS serves as object identity.

_var References

_var references are used to make memory leaks less likely.

Like other **_var** types, **_var** references make deep copies and release their reference in the destructor:

```
{
    CORBA::Object_var obj = orb->string_to_object(...);
    CCS::Controller_var ctrl = CCS::Controller::_narrow(obj);
    // ...
} // No need to release anything here...
```

Use **_var** references to “catch” object references returned from invocations.

_var references are extremely useful for exception safety!

```
class Thermometer_var {
public:
                      Thermometer_var();
                      Thermometer_var(Thermometer_ptr &);
                      Thermometer_var(const Thermometer_var &);
                      ~Thermometer_var();

    Thermometer_var & operator=(Thermometer_ptr &);
    Thermometer_var & operator=(const Thermometer_var &);
                      operator Thermometer_ptr &();
    Thermometer_ptr   operator->() const;

    Thermometer_ptr   in() const;
    Thermometer_ptr & inout();
    Thermometer_ptr & out();
    Thermometer_ptr   _retn();
private:
    Thermometer_ptr _ptr;
};
```

```
#include <OB/CORBA.h>   // Import CORBA module
#include "CCS.hh"       // Import CCS system (IDL-generated)

CORBA::ORB_var orb;     // Global for convenience

int
main(int argc, char * argv[])
{
    // Initialize the ORB
    orb = CORBA::ORB_init(argc, argv);

    // Get controller reference from argv
    // and convert to object.
    CORBA::Object_var obj = orb->string_to_object(argv[1]);
    if (CORBA::is_nil(obj)) {
        cerr << "Nil controller reference" << endl;
        return 1;
    }

    // Try to narrow to CCS::Controller.
    CCS::Controller_var ctrl = CCS::Controller::_narrow(obj);
    if (CORBA::is_nil(ctrl)) {
        cerr << "Wrong type for controller ref." << endl;
        return 1;
    }
```

```
    // Use controller...

    // No need to release anything here (but
    // ORB::destroy() is still necessary).
    orb->destroy();

    return 0;
}
```

_var References and Widening

_var references do not mirror the IDL inheritance hierachy:

```
class Thermometer : public virtual CORBA::Object { /* ... */ };
class Thermostat : public virtual Thermometer { /* ... */ };

typedef Thermometer * Thermometer_ptr;
typedef Thermostat * Thermostat_ptr;

class Thermometer_var { /* ... */ };    // No inheritance!
class Thermostat_var { /* ... */ };     // No inheritance!
```

Implicit widening on **_var** references therefore does not compile:

```
Thermostat_var tstat = ...;
Thermometer_var thermo = tstat;     // Compile-time error
CORBA::Object_var obj = tstat;      // Compile-time error
```

You can use **_duplicate** to widen a reference explicitly:

```
Thermostat_var tstat = ...;
Thermometer_var thermo = Thermometer::_duplicate(tstat);
```

```
Thermostat_var tstat = ...;

Thermometer_var thermo;
thermo = Thermometer::_duplicate(tstat);
thermo = Thermostat::_duplicate(tstat);

CORBA::Object_var obj;
obj = Thermostat::_duplicate(tstat);
obj = Thermometer::_duplicate(tstat);
obj = CORBA::Object::_duplicate(tstat);
```

References Nested in Complex Types

References that are members of structures, unions, or exceptions, or elements of sequences or arrays behave like **`_var`** references:

```
struct DevicePair {
    Thermometer mem1;
    Object      mem2;
};
```

The same rules as for strings apply:

```
Thermometer_var thermo = ...;
Thermostat_var tstat = ...;

DevicePair dp;
dp.mem1 = thermo;                             // Deep assignment
dp.mem2 = Object::_duplicate(thermo);         // Deep assignment
DevicePair dp2 = dp;                          // Deep copy
dp2.mem2 = orb->string_to_object(argv[1]);   // No leak here
```

Mapping for Operations

Operations on IDL interfaces map to proxy member functions with the same name.

If you have a `_var` or `_ptr` reference to a proxy instance, you invoke the member function via the reference's `->` operator.

The proxy member function sends the request to the remote object and blocks until the reply arrives.

The proxy unmarshals the results and returns.

The net effect is a synchronous procedure call.

```
interface Example {
    void        send(in char c);
    oneway void put(in char c);
    long        get_long();
    string      id_to_name(in string id);
};
```

```
class Example : public virtual CORBA::Object {
public:
    // ...
    void        send(CORBA::Char c);
    void        put(CORBA::Char c);
    CORBA::Long get_long();
    char *      id_to_name(const char * id);
    // ...
};
```

Mapping for Attributes

IDL attributes map to a pair of member functions, one to read the value and one to write it.

`readonly` attributes only have an accessor and no modifier.

```
interface Thermometer {
    readonly attribute unsigned long    asset_num;
             attribute string           location;
};
```

The proxy contains:

```
class Thermometer : public virtual CORBA::Object {
public:
  CORBA::ULong asset_num();            // Accessor
  char *       location();             // Accessor
  void         location(const char *); // Modifier
};
```

```
CCS::Thermometer_var t = ...;   // Get reference
CORBA::ULong anum = t->asset_num();
cout << "Asset number is " << anum << endl;
```

```
CCS::Thermometer_var t = ...;
t->location("Room 414");
```

Parameter Passing

The rules for parameter passing depend on the type and direction:

- Simple **`in`** parameters are passed by value.
- Complex **`in`** parameters are passed by constant reference.
- **`inout`** parameters are passed by reference.
- Fixed-length **`out`** parameters are passed by reference.
- Variable-length **`out`** parameters are dynamically allocated.
- Fixed-length return values are passed by value.
- Variable-length return values are dynamically allocated.
- Fixed-length array return values are dynamically allocated.

Note: Variable-length values that travel from server to client are dynamically allocated. Everything else can be allocated on the stack.

Parameter Passing (cont.)

Consider an operation that passes a **`char`** parameter in all possible directions:

```
interface Foo {
    char op(in char p_in, inout char p_inout, out char p_out);
};
```

The proxy signature is:

```
CORBA::Char op(
                CORBA::Char p_in,
                CORBA::Char & p_inout,
                CORBA::Char & p_out
            );
```

This signature is no different than for any normal C++ function that passes parameters in the same directions.

```
Foo_var fv = ...;   // Get reference

CORBA::Char inout_val;
CORBA::Char out_val;
CORBA::Char ret_val;

inout_val = 'A';
ret_val = fv->op('X', inout_val, out_val);

cout << "ret_val: " << ret_val << endl;
cout << "inout_val: " << inout_val << endl;
cout << "out_val: " << out_val << endl;
```

Parameter Passing (cont.)

Fixed-length unions and structures are passed by value or by reference:

```
struct F {
    char    c;
    short   s;
};

interface Foo {
    F op(in F p_in, inout F p_inout, out F p_out);
};
```

The proxy signature is:

```
typedef F & F_out;
F op(
        const F &   p_in,
        F &         p_inout,
        F_out       p_out
  );
```

```
Foo_var fv = ...;   // Get reference...

F in_val = { 'A', 1 };
F inout_val = { 'B', 2 };
F out_val;
F ret_val;

ret_val = fv->op(in_val, inout_val, out_val);

// in_val is unchanged, inout_val may have been changed,
// and out_val and ret_val are filled in by the operation.
```

Parameter Passing (cont.)

Fixed-length arrays are passed by pointer to an array slice:

```
typedef short SA[2];

interface Foo {
    SA op(in SA p_in, inout SA p_inout, out SA p_out);
};
```

The proxy signature is:

```
typedef SA_slice * SA_out;
SA_slice * op(
                const SA    p_in,
                SA_slice *  p_inout,
                SA_out      p_out
           );
```

```
Foo_var fv = ...;          // Get reference...

SA in_val = { 1, 2 };
SA inout_val = { 3, 4 };
SA out_val;
SA_slice * ret_val;        // Note pointer to an array slice

ret_val = op(in_val, inout_val, out_val);

// in_val is unchanged, inout_val may have been changed,
// out_val now contains values, and ret_val points
// to a dynamically allocated instance.

SA_free(ret_val);          // Must free here!
```

Parameter Passing (cont.)

Strings are passed as pointers.

```
interface Foo {
    string op(
            in string       p_in,
            inout string    p_inout,
            out             string p_out
    );
};
```

The proxy signature is:

```
char * op(
            const char *        p_in,
            char * &            p_inout,
            CORBA::String_out   p_out
       );
```

`String_out` is a class that behaves (almost) like a **`char * &`**.

```
Foo_ref fv = ...;        // Get reference...

// inout strings *must* be dynamically allocated
char * inout_val = CORBA::string_dup("World");

char * out_val;
char * ret_val;

ret_val = fv->op("Hello", inout_val, out_val);

cout << "inout_val: \"" << inout_val << "\"" << endl;
cout << "out_val: \"" << out_val << "\"" << endl;
cout << "ret_val: \"" << ret_val << "\"" << endl;

// Clean up...
CORBA::string_free(inout_val);
CORBA::string_free(out_val);
CORBA::string_free(ret_val);
```

```
Foo_ref fv = ...;        // Get reference...

CORBA::String_var inout_val = CORBA::string_dup("World");

CORBA::String_var out_val;
CORBA::String_var ret_val;

ret_val = fv->op("Hello", inout_val, out_val);

cout << "inout_val: \"" << inout_val << "\"" << endl;
cout << "out_val: \"" << out_val << "\"" << endl;
cout << "ret_val: \"" << ret_val << "\"" << endl;

// No deallocation necessary, the String_vars take care of it...
```

```
CORBA::String_var in_val, inout_val, out_val, ret_val;

ret_val = fv->op(in_val, inout_val, out_val);   // Wrong!
```

Parameter Passing (cont.)

```
interface Foo {
    void get_name(out string name);
};
```

The following code leaks memory:

```
char * name;
fv->get_name(name);
// Should have called CORBA::string_free(name) here!
fv->get_name(name);                                     // Leak!
```

The following code does not:

```
CORBA::String_var name;
fv->get_name(name);
fv->get_name(name);                             // Fine
```

The **String_out** type takes care of deallocating the first result before passing the parameter to the stub.

```
class String_out {
public:
    String_out(char * & s): _sref(s) { _sref = 0; }
    String_out(String_var & sv): _sref(sv._sref) {
        string_free(_sref);
        _sref = 0;
    }
    // More member functions here...
private:
    char * & _sref;
};
```

```
void get_name(CORBA::String_out name);
```

Parameter Passing (cont.)

Sequences and variable-length structures and unions are dynamically allocated if they are an **out** parameter or the return value.

```
typedef sequence<octet> OctSeq;
interface Foo {
    OctSeq op(
                in OctSeq       p_in,
                inout OctSeq    p_inout,
                out OctSeq      p_out
            );
};
```

The proxy signature is:

```
typedef OctSeq & OctSeq_out;
OctSeq * op(  const OctSeq & p_in,
              OctSeq &       p_inout,
              OctSeq_out     p_out);
```

```
Foo_var fv = ...;        // Get reference...

OctSeq in_val;
OctSeq inout_val
OctSeq * out_val;        // *Pointer* to OctSeq
OctSeq * ret_val;        // *Pointer* to OctSeq

in_val.length(1);
in_val[0] = 99;
inout_val.length(2);
inout_val[0] = 5;
inout_val[1] = 6;

ret_val = fv->op(in_val, inout_val, out_val);

// inout_val may have been modified, out_val and
// ret_val point to now initialized sequences

delete out_val;          // Must deallocate here!
delete ret_val;          // Must deallocate here!
```

```
Foo_var fv = ...;        // Get reference...

OctSeq in_val;
OctSeq inout_val;
OctSeq_var out_val;      // _var type
OctSeq_var ret_val;      // _var type

in_val.length(1);
in_val[0] = 99;
inout_val.length(2);
inout_val[0] = 5;
inout_val[1] = 6;

ret_val = fv->op(in_val, inout_val, out_val);

// out_val and ret_val will eventually deallocate...
```

```
Foo_var fv = ...;        // Get reference...

OctSeq_var in_val(new OctSeq);
OctSeq_var inout_val(new OctSeq);
OctSeq_var out_val;
OctSeq_var ret_val;

in_val->length(1);
in_val[0] = 99;
inout_val->length(2);
inout_val[0] = 5;
inout_val[1] = 6;

ret_val = fv->op(in_val, inout_val, out_val);

// ...
```

Parameter Passing (cont.)

Variable-length **out** arrays and return values are dynamically allocated.

```
struct Employee {
    string  name;
    long    number;
};
typedef Employee EA[2];

interface Foo {
    EA op(in EA p_in, inout EA p_inout, out EA p_out);
};
```

The proxy signature is:

```
EA_slice * op(
                const EA    p_in,
                EA_slice *  p_inout,
                EA_out      p_out
           );
```

```
Foo_var fv = ...;        // Get reference...

EA in_val;
in_val[0].name = CORBA::string_dup("Michi");
in_val[0].number = 1;
in_val[1].name = CORBA::string_dup("Steve");
in_val[1].number = 2;

EA inout_val;
inout_val[0].name = CORBA::string_dup("Bjarne");
inout_val[0].number = 3;
inout_val[1].name = CORBA::string_dup("Stan");
inout_val[1].number = 4;

EA_slice * out_val;      // Note pointer to slice
EA_slice * ret_val;      // Note pointer to slice

ret_val = fv->op(in_val, inout_val, out_val);
// in_val is unchanged, inout_val may have been changed,
// out_val and ret_val point at dynamically allocated arrays

EA_free(out_val);        // Must free here!
EA_free(ret_val);        // Must free here!
```

```
Foo_var fv = ...;        // Get reference...

EA in_val;
// Initialize in_val...

EA inout_val;
// Initialize inout_val...

EA_var out_val;     // _var type
EA_var ret_val;     // _var type

ret_val = fv->op(in_val, inout_val, out_val);
// in_val is unchanged, inout_val may have been changed,
// out_val and ret_val point at dynamically allocated arrays

// No need for explicit deallocation here...
```

Parameter Passing (cont.)

Object reference **out** parameters and return values are duplicated.

```
interface Thermometer { /* ... */ };

interface Foo {
    Thermometer op(
                    in Thermometer      p_in,
                    inout Thermometer   p_inout,
                    out Thermometer     p_out
                );
};
```

The proxy signature is:

```
Thermometer_ptr op(
                    Thermometer_ptr     p_in,
                    Thermometer_ptr &   p_inout,
                    Thermometer_out     p_out
                );
```

```
Foo_var fv = ...;                           // Get reference...

Thermometer_ptr in_val = ...;               // Initialize in param
Thermometer_ptr inout_val = ...;            // Initialize inout param
Thermometer_ptr out_val;
Thermometer_ptr ret_val;

ret_val = fv->op(in_val, inout_val, out_val);

CORBA::release(in_val);
CORBA::release(inout_val);
CORBA::release(out_val);
CORBA::release(ret_val);
```

```
Foo_var fv = ...;                    // Get reference...

Thermometer_var in_val = ...;        // Initialize in param
Thermometer_var inout_val = ...;     // Initialize inout param
Thermometer_var out_val;
Thermometer_var ret_val;

ret_val = fv->op(in_val, inout_val, out_val);

// No releases necessary here.
```

Parameter Passing: Pitfalls

Stick to the following rules when invoking operations:

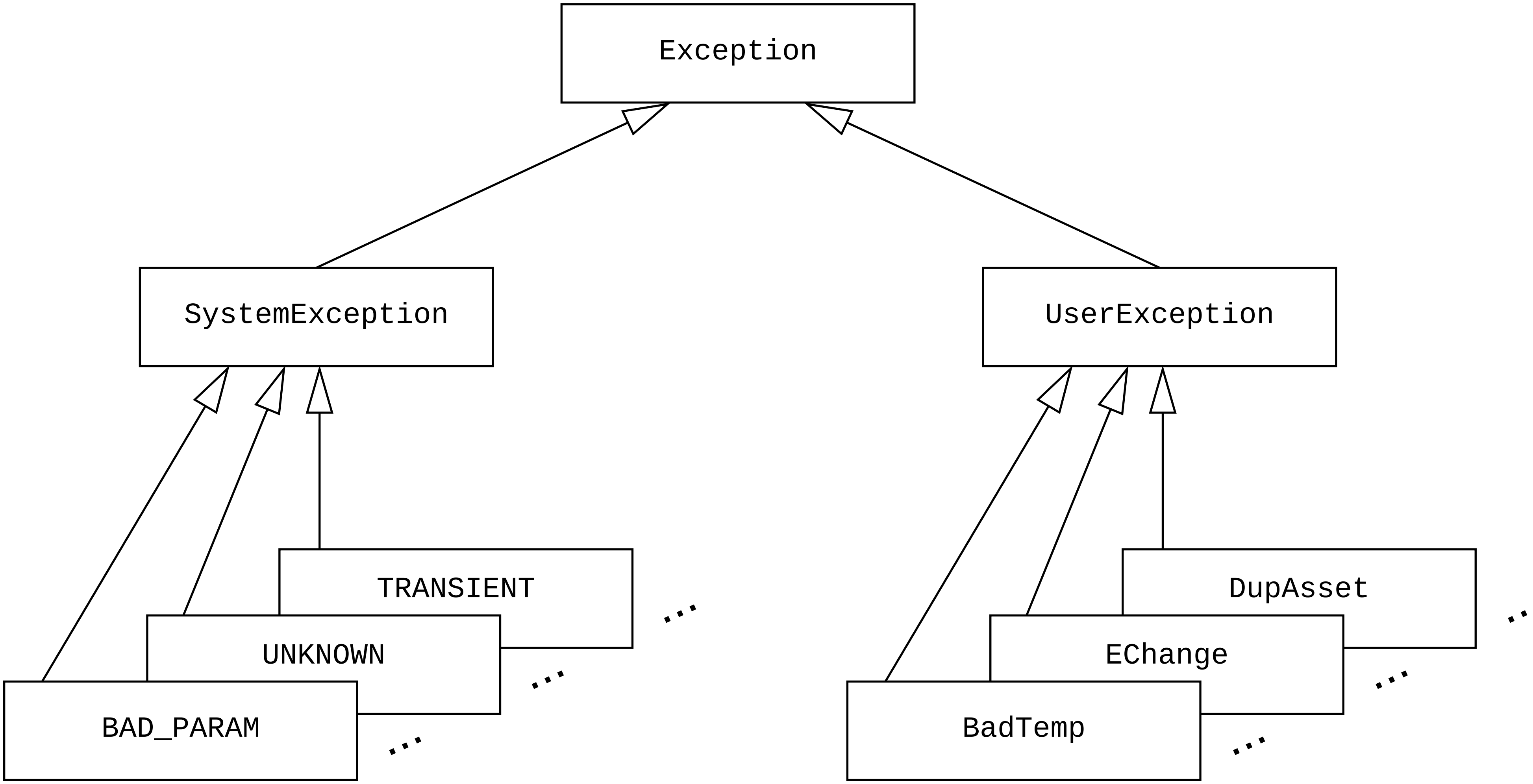
- Always initialize **`in`** and **`inout`** parameters**.**
- Do not pass **`in`** or **`inout`** null pointers.
- Deallocate **`out`** parameters that are passed by pointer.
- Deallocate variable-length return values.
- Do not ignore the return value from an operation that returns a variable-length value.
- Use _**`var`** types to make life easier for yourself.

```
// Assume IDL:
// interface Foo {
//      string get_name();
//      void   set_name(in string n);
// };

Foo_var fv = ...;

cout << fv->get_name() << endl;          // Leak!

CORBA::String_var s = fv->get_name();   // Better
cout << s << endl;

// Or use:
cout << CORBA::String_var(fv->get_name()) << endl;
```

```
Foo_var fv1 = ...;
Foo_var fv2 = ...;

fv1->set_name(fv2->get_name());         // Leak!

CORBA::String_var tmp = fv2->get_name();
fv1->set_name(tmp);                     // Better!

// Or use:
fv1->set_name(CORBA::String_var(fv2->get_name()));
```

Mapping for Exceptions

IDL exceptions map to a hierarchy of C++ classes:



Mapping for Exceptions (cont.)

The exception hierarchy looks like this:

```
namespace CORBA {
    class Exception {                                           // Abstract
    public:
        virtual ~Exception();
        virtual const char * _name() const;
        virtual const char * _rep_id() const;
        virtual void         _raise() = 0;
    };
    class SystemException : public Exception {                  // Abstract
        // ...
    };
    class UserException : public Exception {                    // Abstract
        // ...
    };
    class UNKNOWN : public SystemException { /* ... */ };  // Concrete
    class FREE_MEM : public SystemException { /* ... */ }; // Concrete
    // etc...
};
```

```
CCS::Thermostat_var ts = ...;
CCS::TempType temp = ...;

try {
    ts->set_nominal(temp);
} catch (const CCS::Thermostat::BadTemp &) {
    // New temperature out of range
} catch (const CORBA::UserException & e) {
    // Some other user exception was raised
    cerr << "User exception: " << e << endl;
} catch (const CORBA::OBJECT_NOT_EXIST &) {
    cerr << "No such thermostat" << endl;
} catch (const CORBA::SystemException & e) {
    // Some other system exception
    cerr << "System exception: " << e << endl;
} catch (...) {
    // Some non-CORBA exception -- should never happen
}
```

Mapping for System Exceptions

All system exceptions derive from the **`SystemException`** base class:

```
class SystemException : public Exception {
public:
                        SystemException();
                        SystemException(const SystemException &);
                        SystemException(
                            ULong                   minor,
                            CompletionStatus        status
                        );
                        ~SystemException();
    SystemException &   operator=(const SystemException);

    ULong               minor() const;
    void                minor(ULong);

    CompletionStatus    completed() const;
    void                completed(CompletionStatus);
    // ...
};
```

Semantics of System Exceptions

The standard defines semantics for system exceptions. For some exceptions, the semantics are defined only in broad terms (such as **INTERNAL** or **NO_MEMORY**).

The most commonly encountered system exceptions are:

OBJECT_NOT_EXIST, **TRANSIENT**, **BAD_PARAM**, **COMM_FAILURE**, **IMP_LIMIT**, **NO_MEMORY**, **UNKNOWN**, **NO_PERMISSION**, and **NO_RESOURCES**.

The specification defines minor codes for some exceptions to provide more detailed information on a specific error. Standard minor codes are allocated in the range **0x4f4d0000–0x4f4d0fff**.

Vendors can allocate a block of minor code values for their own use. For ORBacus-specific minor codes, the allocated range is **0x4f4f0000–0x4f4f0fff**.

Mapping for User Exceptions

User exceptions map to a class with public members:

```
exception Boom {
    string  msg;
    short   errno;
};
```

This generates:

```
class Boom : public CORBA::UserException {
                        Boom();
                        Boom(const char*, CORBA::Short);
                        Boom(const Boom &);
                        Boom& operator=(const Boom &);
                        ~Boom();

    OB::StrForStruct    msg;
    CORBA::Short        errno;
};
```

```
if (something_failed)
    throw Boom("Something failed", 99);
```

```
try {
    some_ref->some_op();
} catch (const Boom & b) {
    cerr << "Boom: " << b.msg << " (" << b.errno << ")" << endl;
}
```

Compiling and Linking

To create a client executable, you must compile the application code and the stub code. Typical compile commands are:

```
c++ -I. -I/opt/OB/include -c client.cc
c++ -I. -I/opt/OB/include -c MyIDL.cpp
```

The exact flags and compile command vary with the platform.

To link the client, you must link against `libOB`:

```
c++ -o client client.o MyIDL.o -L/opt/OB/lib -lOB
```

If you are using JThreads/C++, you also need to link against the JTC library and the native threads library. For example:

```
c++ -o client client.o MyIDL.o -L/opt/OB/lib \
-lOB -lJTC -lpthread
```

```
#include <OB/CORBA.h>
#include <OB/JTC.h>
```

```
// Get thermostat reference from argv[1]
// and convert to object.
CORBA::Object_var obj = orb->string_to_object(argv[1]);
if (CORBA::is_nil(obj)) {
    cerr << "Nil thermostat reference" << endl;
    throw 0;
}

// Try to narrow to CCS::Thermostat.
CCS::Thermostat_var tmstat;
try {
    tmstat = CCS::Thermostat::_narrow(obj);
} catch (const CORBA::SystemException & se) {
    cerr << "Cannot narrow thermostat reference: "
     << se << endl;
    throw 0;
}
if (CORBA::is_nil(tmstat)) {
    cerr << "Wrong type for thermostat ref." << endl;
    throw 0;
}
```

```
// Show the details for a thermometer or thermostat.

static ostream &
operator<<(ostream & os, CCS::Thermometer_ptr t)
{
    // Check for nil.
    if (CORBA::is_nil(t)) {
        os << "Cannot show state for nil reference." << endl;
        return os;
    }

    // Try to narrow and print what kind of device it is.
    CCS::Thermostat_var tmstat = CCS::Thermostat::_narrow(t);
    os << (CORBA::is_nil(tmstat)?"Thermometer:":"Thermostat:")
       << endl;

    // Show attribute values.
    CCS::ModelType_var model = t->model();
    CCS::LocType_var location = t->location();
    os << "\tAsset number: " << t->asset_num() << endl;
    os << "\tModel       : " << model << endl;
    os << "\tLocation    : " << location << endl;
    os << "\tTemperature : " << t->temperature() << endl;

    // If device is a thermostat, show nominal temperature.
```

```
    if (!CORBA::is_nil(tmstat))
        os << "\tNominal temp: " << tmstat->get_nominal() << endl;
    return os;
}
```

```
// Show details of thermostat
cout << tmstat << endl;
```

```
// Change the temperature of the thermostat to a valid
// temperature.
cout << "Changing nominal temperature" << endl;
CCS::TempType old_temp = tmstat->set_nominal(60);
cout << "Nominal temperature is now 60, previously "
     << old_temp << endl << endl;

cout << "Retrieving new nominal temperature" << endl;
cout << "Nominal temperature is now "
     << tmstat->get_nominal() << endl << endl;
```

```
// Show the information in a BtData struct.

static ostream &
operator<<(ostream & os, const CCS::Thermostat::BtData & btd)
{
    os << "CCS::Thermostat::BtData details:" << endl;
    os << "\trequested    : " << btd.requested << endl;
    os << "\tmin_permitted: " << btd.min_permitted << endl;
    os << "\tmax_permitted: " << btd.max_permitted << endl;
    os << "\terror_msg    : " << btd.error_msg << endl;
    return os;
}
```

```
// Change the temperature to an illegal value and
// show the details of the exception that is thrown.
cout << "Setting nominal temperature out of range" << endl;
bool got_exception = false;
try {
    tmstat->set_nominal(10000);
} catch (const CCS::Thermostat::BadTemp & e) {
    got_exception = true;
    cout << "Got BadTemp exception: " << endl;
    cout << e.details << endl;
}
if (!got_exception)
    assert("Did not get exception");
```

```
// Get controller reference from argv[2]
// and convert to object.
obj = orb->string_to_object(argv[2]);
if (CORBA::is_nil(obj)) {
    cerr << "Nil controller reference" << endl;
    throw 0;
}

// Try to narrow to CCS::Controller.
CCS::Controller_var ctrl;
try {
    ctrl = CCS::Controller::_narrow(obj);
} catch (const CORBA::SystemException & se) {
    cerr << "Cannot narrow controller reference: "
         << se << endl;
    throw 0;
}
if (CORBA::is_nil(ctrl)) {
    cerr << "Wrong type for controller ref." << endl;
    throw 0;
}
```

```
// Get list of devices
CCS::Controller::ThermometerSeq_var list = ctrl->list();

// Show number of devices.
CORBA::ULong len = list->length();
cout << "Controller has " << len << " device";
if (len != 1)
    cout << "s";
cout << "." << endl;

// Show details for each device.
CORBA::ULong i;
for (i = 0; i < len; ++i)
    cout << list[i].in();
cout << endl;
```

```
// Look for device in Rooms Earth and HAL.
cout << "Looking for devices in Earth and HAL." << endl;
CCS::Controller::SearchSeq ss;
ss.length(2);
ss[0].key.loc(CORBA::string_dup("Earth"));
ss[1].key.loc(CORBA::string_dup("HAL"));
ctrl->find(ss);

// Show the devices found in that room.
for (i = 0; i < ss.length(); ++i)
    cout << ss[i].device.in();      // Overloaded <<
cout << endl;
```

```
// Loop over the sequence of records in an EChange exception and
// show the details of each record.

static ostream &
operator<<(ostream & os, const CCS::Controller::EChange & ec)
{
    CORBA::ULong i;
    for (i = 0; i < ec.errors.length(); ++i) {
        os << "Change failed:" << endl;
        os << ec.errors[i].tmstat_ref.in(); // Overloaded <<
        os << ec.errors[i].info << endl;    // Overloaded <<
    }
    return os;
}
```

```
// Increase the temperature of all thermostats
// by 40 degrees. First, make a new list (tss)
// containing only thermostats.
cout << "Increasing thermostats by 40 degrees." << endl;
CCS::Thermostat_var ts;
CCS::Controller::ThermostatSeq tss;
for (i = 0; i < list->length(); ++i) {
    ts = CCS::Thermostat::_narrow(list[i]);
    if (CORBA::is_nil(ts))
        continue;                           // Skip thermometers
    len = tss.length();
    tss.length(len + 1);
    tss[len] = ts;
}

// Try to change all thermostats.
try {
    ctrl->change(tss, 40);
} catch (const CCS::Controller::EChange & ec) {
    cerr << ec;                             // Overloaded <<
}
```

```
#include <OB/CORBA.h>
#include <assert.h>

#if defined(HAVE_STD_IOSTREAM)
using namespace std;
#endif

#include    "CCS.h"

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

// Show the details for a thermometer or thermostat.

static ostream &
operator<<(ostream & os, CCS::Thermometer_ptr t)
{
    // Check for nil.
    if (CORBA::is_nil(t)) {
        os << "Cannot show state for nil reference." << endl;
        return os;
    }

    // Try to narrow and print what kind of device it is.
    CCS::Thermostat_var tmstat = CCS::Thermostat::_narrow(t);
    os << (CORBA::is_nil(tmstat)?"Thermometer:":"Thermostat:")
```

```
        << endl;

    // Show attribute values.
    CCS::ModelType_var model = t->model();
    CCS::LocType_var location = t->location();
    os << "\tAsset number: " << t->asset_num() << endl;
    os << "\tModel       : " << model << endl;
    os << "\tLocation    : " << location << endl;
    os << "\tTemperature : " << t->temperature() << endl;

    // If device is a thermostat, show nominal temperature.
    if (!CORBA::is_nil(tmstat)) {
        os << "\tNominal temp: "
        << tmstat->get_nominal() << endl;
    }
    return os;
}

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

// Show the information in a BtData struct.

static ostream &
operator<<(ostream & os, const CCS::Thermostat::BtData & btd)
{
```

```
    os << "CCS::Thermostat::BtData details:" << endl;
    os << "\trequested    : " << btd.requested << endl;
    os << "\tmin_permitted: " << btd.min_permitted << endl;
    os << "\tmax_permitted: " << btd.max_permitted << endl;
    os << "\terror_msg    : " << btd.error_msg << endl;
    return os;
}

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

// Loop over the sequence of records in an EChange exception and
// show the details of each record.

static ostream &
operator<<(ostream & os, const CCS::Controller::EChange & ec)
{
    CORBA::ULong i;
    for (i = 0; i < ec.errors.length(); ++i) {
        os << "Change failed:" << endl;
        os << ec.errors[i].tmstat_ref.in(); // Overloaded <<
        os << ec.errors[i].info << endl;    // Overloaded <<
    }
    return os;
}
```

```
//-------------------------------------------------------------

int
main(int argc, char * argv[])
{
    int status = 0;
    CORBA::ORB_var orb;

    try {
        // Initialize ORB and check arguments.
        orb = CORBA::ORB_init(argc, argv);
        if (argc != 3) {
            cerr << "Usage: client IOR IOR" << endl;
            throw 0;
        }

        // Get thermostat reference from argv[1]
        // and convert to object.
        CORBA::Object_var obj = orb->string_to_object(argv[1]);
        if (CORBA::is_nil(obj)) {
            cerr << "Nil thermostat reference" << endl;
            throw 0;
        }

        // Try to narrow to CCS::Thermostat.
```

```
CCS::Thermostat_var tmstat;
try {
    tmstat = CCS::Thermostat::_narrow(obj);
} catch (const CORBA::SystemException & se) {
    cerr << "Cannot narrow thermostat reference: "
         << se << endl;
    throw 0;
}
if (CORBA::is_nil(tmstat)) {
    cerr << "Wrong type for thermostat ref." << endl;
    throw 0;
}

// Show details of thermostat
cout << tmstat << endl;

// Change the temperature of the thermostat to a valid
// temperature.
cout << "Changing nominal temperature" << endl;
CCS::TempType old_temp = tmstat->set_nominal(60);
cout << "Nominal temperature is now 60, previously "
     << old_temp << endl << endl;

cout << "Retrieving new nominal temperature" << endl;
cout << "Nominal temperature is now "
```

```
     << tmstat->get_nominal() << endl << endl;

// Change the temperature to an illegal value and
// show the details of the exception that is thrown.
cout << "Setting nominal temperature out of range"
 << endl;
bool got_exception = false;
try {
    tmstat->set_nominal(10000);
} catch (const CCS::Thermostat::BadTemp & e) {
    got_exception = true;
    cout << "Got BadTemp exception: " << endl;
    cout << e.details << endl;
}
if (!got_exception)
    assert("Did not get exception");

// Get controller reference from argv[2]
// and convert to object.
obj = orb->string_to_object(argv[2]);
if (CORBA::is_nil(obj)) {
    cerr << "Nil controller reference" << endl;
    throw 0;
}
```

```
// Try to narrow to CCS::Controller.
CCS::Controller_var ctrl;
try {
    ctrl = CCS::Controller::_narrow(obj);
} catch (const CORBA::SystemException & se) {
    cerr << "Cannot narrow controller reference: "
         << se << endl;
    throw 0;
}
if (CORBA::is_nil(ctrl)) {
    cerr << "Wrong type for controller ref." << endl;
    throw 0;
}

// Get list of devices
CCS::Controller::ThermometerSeq_var list = ctrl->list();

// Show number of devices.
CORBA::ULong len = list->length();
cout << "Controller has " << len << " device";
if (len != 1)
    cout << "s";
cout << "." << endl;

// Show details for each device.
```

```
CORBA::ULong i;
for (i = 0; i < len; ++i)
    cout << list[i].in();
cout << endl;

// Look for device in Rooms Earth and HAL.
cout << "Looking for devices in Earth and HAL." << endl;
CCS::Controller::SearchSeq ss;
ss.length(2);
ss[0].key.loc(CORBA::string_dup("Earth"));
ss[1].key.loc(CORBA::string_dup("HAL"));
ctrl->find(ss);

// Show the devices found in that room.
for (i = 0; i < ss.length(); ++i)
    cout << ss[i].device.in();      // Overloaded <<
cout << endl;

// Increase the temperature of all thermostats
// by 40 degrees. First, make a new list (tss)
// containing only thermostats.
cout << "Increasing thermostats by 40 degrees." << endl;
CCS::Thermostat_var ts;
CCS::Controller::ThermostatSeq tss;
for (i = 0; i < list->length(); ++i) {
```

```
        ts = CCS::Thermostat::_narrow(list[i]);
        if (CORBA::is_nil(ts))
            continue;                   // Skip thermometers
        len = tss.length();
        tss.length(len + 1);
        tss[len] = ts;
    }

    // Try to change all thermostats.
    try {
        ctrl->change(tss, 40);
    } catch (const CCS::Controller::EChange & ec) {
        cerr << ec;                     // Overloaded <<
    }
}
catch (const CORBA::Exception& ex) {
    cerr << "Uncaught CORBA exception: " << ex << endl;
    status = 1;
} catch (...) {
    status = 1;
}

if (!CORBA::is_nil(orb)) {
    try {
        orb -> destroy();
```

```
        } catch (const CORBA::Exception& ex) {
            cerr << ex << endl;
            status = 1;
        }
    }

    return status;
}
```

Introduction

The server-side C++ mapping is a superset of the client-side mapping.

Writing a server requires additional constructs to:

- connect servants to skeleton classes
- receive and return parameters correctly
- create object references for objects
- initialize the server-side run time
- run an event loop to accept incoming requests

The server-side mapping is easy to learn because most of it follows from the client-side mapping.

Mapping for Interfaces

On the server side, a skeleton class provides the counterpart to the client-side proxy class.

Skeletons provides an up-call interface from the ORB networking layer into the application code.

The skeleton class contains pure virtual functions for IDL operations.

Skeleton classes have the name of the IDL interface with a **`POA_`** prefix. For example:

- **`::MyObject`** has the skeleton **`::POA_MyObject`**
- **`CCS::Thermometer`** has the skeleton class **`POA_CCS::Thermometer`**.

Note that modules map to namespaces as for the client side, and that the **`POA_`** prefix applies only to the outermost scope (whether module or interface).

Skeleton Classes

The skeleton class for an interface contains a pure virtual function for each IDL operation:

```
interface AgeExample {
    unsigned short  get_age();
};
```

The skeleton class contains:

```
class POA_AgeExample :
    public virtual PortableServer::ServantBase {
public:
    virtual CORBA::UShort
                get_age() throw(CORBA::SystemException) = 0;
    // ...
};
```

Servant Classes

Servant classes are derived from their skeleton:

```
#include "Age_skel.h"   // IDL file is called "Age.idl".
                        // Header file names are ORB-specific!

class AgeExample_impl : public virtual POA_AgeExample {
public:
    // Inherited IDL operation
    virtual CORBA::UShort
                get_age() throw(CORBA::SystemException);
    // You can add other members here...
private:
    AgeExample_impl(const AgeExample_impl &); // Forbidden
    void operator=(const AgeExample_impl &);  // Forbidden
    // You can add other members here...
};
```

Operation Implementation

The implementation of a servant's virtual functions provides the behavior of an operation:

```
CORBA::UShort
AgeExample_impl::
get_age() throw(CORBA::SystemException)
{
    return 16;
}
```

Typically, the implementation of an operation will access private member variables that store the state of an object, or perform a database access to retrieve or update the state.

Once a servant's function is invoked, your code is in control and can therefore do whatever is appropriate for your implementation.

Attribute Implementation

As for the client side, attributes map to an accessor and modifier function (or just an accessor for **readonly** attributes):

```
interface Part {
    readonly attribute long asset_num;
             attribute long price;
};
```

The skeleton code contains:

```
class POA_Part : public virtual PortableServer::ServantBase {
public:
    virtual CORBA::Long asset_num() throw(CORBA::SystemException) = 0;
    virtual CORBA::Long price() throw(CORBA::SystemException) = 0;
    virtual void price(CORBA::Long) throw(CORBA::SystemException) = 0;
    // ...
};
```

Servant Activation and Reference Creation

Every skeleton class contains a function called **`_this`**:

```
class POA_AgeExample :
    public virtual PortableServer::ServantBase {
public:
    AgeExample_ptr _this();
    // ...
};
```

To create a CORBA object, you instantiate the servant and call **`_this`**:

```
AgeExample_impl age_servant;              // Create servant
AgeExample_var av = age_servant._this(); // Create reference
```

- Instantiating the servant has no effect on the ORB.
- Calling **`_this`** activates the servant and returns its reference.

`_this` implicitly calls **`_duplicate`**, so you must eventually release the returned reference.

Server Initialization

A server must initialize the ORB run time before it can accept requests. To initialize the server, follow the following steps:

1. Call **`ORB_init`** to initialize the run time.
2. Get a reference to the Root POA.
3. Instantiate one or more servants.
4. Activate each servant.
5. Make references to your objects available to clients.
6. Activate the Root POA's POA manager.
7. Start a dispatch loop.

```
#include <iostream.h>
#include <OB/CORBA.h>
#include "Age_skel.h"

// Servant class definition here...

int
main(int argc, char * argv[])
{
    // Initialize ORB
    CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

    // Get reference to Root POA
    CORBA::Object_var obj =
        orb->resolve_initial_references("RootPOA");
    PortableServer::POA_var poa =
        PortableServer::POA::_narrow(obj);

    // Create an object
    AgeExample_impl age_servant;

    // Write its stringified reference to stdout
    AgeExample_var aev = age_servant._this();
    CORBA::String_var str = orb->object_to_string(aev);
    cout << str << endl;
```

```
    // Activate POA manager
    PortableServer::POAManager_var mgr = poa->the_POAManager();
    mgr->activate();

    // Accept requests
    orb->run();
}
```

Parameter Passing

The parameter passing rules for the server side follow those for the client side.

If the client is expected to deallocate a parameter it receives from the server, the server must allocate that parameter:

- Variable-length **`out`** parameters and return values are allocated by the server.
- String and object reference **`inout`** parameters are allocated by the client; the server code must reallocate object references to change them and may reallocate **`inout`** strings or modify their contents in place.
- Everything else is passed by value or by reference.

Parameter Passing (cont.)

Consider an operation that passes a **char** parameter in all possible directions:

```
interface Foo {
    char op(in char p_in, inout char p_inout, out char p_out);
};
```

The skeleton signature is:

```
virtual CORBA::Char op(
                        CORBA::Char     p_in,
                        CORBA::Char &   p_inout,
                        CORBA::Char_out p_out
                    ) throw(CORBA::SystemException) = 0;
```

Parameters are passed by value or by reference, as for the client side.

(**Char_out** is a **typedef** for **Char &**.)

```
CORBA::Char
Foo_impl::
op(
    CORBA::Char     p_in,
    CORBA::Char &   p_inout,
    CORBA::Char_out p_out
) throw(CORBA::SystemException)
{
    // Use p_in, it's initialized already
    cout << p_in << endl;

    // Change p_inout
    p_inout = 'A';

    // Set p_out
    p_out = 'Z';

    // Return a value
    return 'B';
}
```

Parameter Passing (cont.)

Fixed-length unions and structures are passed by value or by reference:

```
struct F {
    char    c;
    short   s;
};

interface Foo {
    F op(in F p_in, inout F p_inout, out F p_out);
};
```

The skeleton signature is:

```
typedef F & F_out;
virtual F op(
                const F &   p_in,
                F &         p_inout,
                F_out       p_out
          ) throw(CORBA::SystemException) = 0;
```

```
F Foo_impl::
op(const F &    p_in,
   F &          p_inout,
   F_out        p_out
) throw(CORBA::SystemException)
{
    // Use incoming values of p_in and p_inout (not shown)

    // Modify p_inout
    p_inout.c = 'A';
    p_inout.s = 1;

    // Initialize p_out
    p_out.c = 'Z';
    p_out.s = 99;

    // Create and initialize return value
    F result = { 'Q', 55 };
    return result;
}
```

Parameter Passing (cont.)

Fixed-length arrays are passed by pointer to an array slice:

```
typedef short SA[2];

interface Foo {
    SA op(in SA p_in, inout SA p_inout, out SA p_out);
};
```

The skeleton signature is:

```
typedef SA_slice * SA_out;
virtual SA_slice * op(
                        const SA    p_in,
                        SA_slice *  p_inout,
                        SA_out      p_out
                    ) throw(CORBA::SystemException) = 0;
```

```
SA_slice * Foo_impl::
op(const SA     p_in,
   SA_slice *   p_inout,
   SA_out       p_out
) throw(CORBA::SystemException)
{
    const size_t arr_len = sizeof(p_in) / sizeof(*p_in);

    // Use incoming values of p_in and p_inout (not shown)

    // Modify p_inout
    for (CORBA::ULong i = 0; i < arr_len; ++i)
        p_inout[i] = i;

    // Initialize p_out
    for (CORBA::ULong i = 0; i < arr_len; ++i)
        p_out[i] = i * i;

    // Create and initialize return value.
    SA_slice * result = SA_alloc();        // Dynamic allocation!
    for (CORBA::ULong i = 0; i < arr_len; ++i)
        result[i] = i * i * i;

    return result;
}
```

Parameter Passing (cont.)

Strings are passed as pointers.

```
interface Foo {
    string op(
                in string       p_in,
                inout string    p_inout,
                out             string p_out
    );
};
```

The skeleton signature is:

```
virtual char * op(
                    const char *         p_in,
                    char * &             p_inout,
                    CORBA::String_out    p_out
                ) throw(CORBA::SystemException) = 0;
```

```
char * Foo_impl::
op( const char *        p_in,
    char * &            p_inout,
    CORBA::String_out   p_out
) throw(CORBA::SystemException)
{
    // Use incoming value
    cout << p_in << endl;

    // Change p_inout
    size_t len = strlen(p_inout);
    for (i = 0; i < len; ++i)
        to_lower(p_inout[i]);

    // Create and initialize p_out
    p_out = CORBA::string_dup("Hello");

    // Create and initialize return value
    return CORBA::string_dup("World");
}
```

```
char * Foo_impl::
op( const char *        p_in,
    char * &            p_inout,
    CORBA::String_out   p_out
) throw(CORBA::SystemException)
{
    // ...

    // Change p_inout
    *p_inout = '\0';                  // Shorten by writing NUL

    // OR:

    CORBA::string_free(p_inout);
    p_inout = CORBA::string_dup("");  // Shorten by reallocation

    // ...
}
```

```
char * Foo_impl::
op( const char *          p_in,
    char * &              p_inout,
    CORBA::String_out     p_out
) throw(CORBA::SystemException)
{
    // ...

    // Lengthen inout string by reallocation
    CORBA::string_free(p_inout);
    p_inout = CORBA::string_dup(longer_string);

    // ...
}
```

Parameter Passing (cont.)

Sequences and variable-length structures and unions are dynamically allocated if they are an **out** parameter or the return value.

```
typedef sequence<octet> OctSeq;
interface Foo {
    OctSeq op(
                in OctSeq    p_in,
                inout OctSeq p_inout,
                out OctSeq   p_out
           );
};
```

The skeleton signature is:

```
typedef OctSeq & OctSeq_out;
virtual OctSeq * op(const OctSeq & p_in,
                    OctSeq &       p_inout,
                    OctSeq_out     p_out
                  ) throw(CORBA::SystemException) = 0;
```

```
OctSeq *
Foo_impl::
op( const OctSeq &  p_in,
    OctSeq &        p_inout,
    OctSeq_out      p_out
) throw(CORBA::SystemException)
{
    // Use incoming values of p_in and p_inout (not shown)

    // Modify p_inout
    CORBA::ULong len = p_inout.length();
    p_inout.length(++len);
    for (CORBA::ULong i = 0; i < len; ++len)
        p_inout[i] = i % 256;

    // Create and initialize p_out
    p_out = new OctSeq;
    p_out->length(1);
    (*p_out)[0] = 0;

    // Create and initialize return value
    OctSeq * p = new OctSeq;
    p->length(2);
    (*p)[0] = 0;
    (*p)[1] = 1;
```

```
    some_func();                                // Potential leak here!

    return p;
}
```

```
OctSeq *
Foo_impl::
op( const OctSeq &  p_in,
    OctSeq &        p_inout,
    OctSeq_out      p_out
) throw(CORBA::SystemException)
{
    // ...

    // Create and initialize return value
    OctSeq_var p = new OctSeq;
    p->length(2);
    p[0] = 0;
    p[1] = 1;

    some_func();                    // No leak here

    return p._retn();
}
```

Parameter Passing (cont.)

```
struct Employee {
    string  name;
    long    number;
};
typedef Employee EA[2];

interface Foo {
    EA op(in EA p_in, inout EA p_inout, out EA p_out);
};
```

The skeleton signature is:

```
virtual EA_slice * op(
                        const EA     p_in,
                        EA_slice *   p_inout,
                        EA_out       p_out
                   ) throw(CORBA::SystemException) = 0;
```

```
EA_slice *
Foo_impl::
op( const EA    p_in,
    EA_slice *  p_inout,
    EA_out      p_out
) throw(CORBA::SystemException)
{
    size_t arr_len = sizeof(p_in) / sizeof(*p_in);

    // Use p_in and initial value of p_inout (not shown)

    // Modify p_inout
    p_inout[0] = p_inout[1];
    p_inout[1].name = CORBA::string_dup("Michi");
    p_inout[1].number = 1;

    // Create and initialize p_out
    p_out = EA_alloc();
    for (CORBA::ULong i = 0; i < arr_len; ++i) {
        p_out[i].name = CORBA::string_dup("Sam");
        p_out[i].number = i;
    }

    // Create and initialize return value
    EA_slice * result = EA_alloc();
```

```
    for (CORBA::ULong i = 0; i < arr_len; ++i) {
        result[i].name = CORBA::string_dup("Max");
        result[i].number = i;
    }

    return result;
}
```

Parameter Passing (cont.)

Object reference **out** parameters and return values are duplicated.

```
interface Thermometer { /* ... */ };

interface Foo {
    Thermometer op(
                    in Thermometer      p_in,
                    inout Thermometer   p_inout,
                    out Thermometer     p_out
                );
};
```

The skeleton signature is:

```
virtual Thermometer_ptr op(
                        Thermometer_ptr     p_in,
                        Thermometer_ptr &   p_inout,
                        Thermometer_out     p_out
                    ) throw(CORBA::SystemException) = 0;
```

```
Thermometer_ptr
Foo_impl::
op( Thermometer_ptr       p_in,
    Thermometer_ptr &     p_inout,
    Thermometer_out       p_out
) throw(CORBA::SystemException)
{
    // Use p_in
    if (!CORBA::is_nil(p_in))
        cout << p_in->temperature() << endl;

    // Use p_inout
    if (!CORBA::is_nil(p_inout))
        cout << p_inout->temperature() << endl;

    // Modify p_inout
    CORBA::release(p_inout);
    p_inout = Thermometer::_duplicate(p_in);

    // Initialize p_out
    p_out = Thermostat::_narrow(p_in);

    // Create return value
    return _this();
}
```

Throwing Exceptions

The exception mapping is identical for client and server. To throw an exception, instantiate the appropriate exception class and throw it.

You can either instantiate an exception as part of the throw statement, or you can instantiate the exception first, assign to the exception members, and then throw the exception.

You should always make your implementation exception safe in that, if it throws an exception, no durable state changes remain.

Avoid throwing system exceptions and use user exceptions instead.

If you must use a system exception, set the **`CompletionStatus`** appropriately.

```
CCS::TempType
Thermostat_impl::
set_nominal(CCS::TempType new_temp)
throw(CORBA::SystemException, CCS::Thermostat::BadTemp)
{
    if (new_temp > max_temp || new_temp < min_temp) {
        CCS::Thermostat::BtData btd;
        btd.requested = new_temp;
        btd.min_temp = min_temp;
        btd.max_temp = max_temp;
        throw CCS::Thermostat::BadTemp(btd);
    }
    // Remember previous nominal temperature and
    // set new nominal temperature...
    return previous_temp;
}
```

```
CCS::TempType
Thermostat_impl::
set_nominal(CCS::TempType new_temp)
throw(CORBA::SystemException, CCS::Thermostat::BadTemp)
{
    if (new_temp < min_temp || new_temp > max_temp) {
        CCS::Thermostat::BadTemp bt;
        bt.details.requested = new_temp;
        bt.details.min_temp = min_temp;
        bt.details.max_temp = max_temp;
        throw bt;
    }
    // Remember previous nominal temperature and
    // set new nominal temperature...
    return previous_temp;
}
```

```
exception ErrorReport {
    string          file_name;
    unsigned long   line_num;
    string          reason;
};
```

```
throw ErrorReport("foo.cc", 597, "Syntax error");
```

```
if (input_parameter_unacceptable)
    throw CORBA::BAD_PARAM();
```

```
if (db_connection_broke_after_partial_update)
    throw CORBA::PERSIST_STORE(CORBA::COMPLETED_YES, 0);
```

Exception Pitfalls

- If you throw an exception and have allocated memory to a variable-length **`out`** parameter, you must deallocate that memory first.

  Use **`_var`** types to prevent such memory leaks.

- Do not throw user exceptions that are not part an operation’s **`raises`** clause.

  Use a **`try`** block around calls to other operations that may throw user exceptions.

```
interface Example {
    void get_name(out string name);
};
```

```
void
Example_impl::
op(CORBA::String_out name)
{
    name = CORBA::string_dup("Otto");

    // Do some database access or whatever...
    if (database_access_failed)
        throw CORBA::PERSIST_STORE();   // Bad news!
}
```

```
void
Example_impl::
op(CORBA::String_out name)
{
    CORBA::String_var name_tmp = CORBA::string_dup("Otto");

    // Do some database access or whatever...
    if (database_access_failed)
        throw CORBA::PERSIST_STORE();   // OK, no leak
    name = name_tmp._retn();            // Transfer ownership
}
```

```
interface EmployeeFinder {
    struct Details { /* ... */ };
    exception BadEmployee { /* ... */ };
    Details get_details(in string name) raises(BadEmployee);
    // ...
};

interface ReportGenerator {
    exception BadDepartment { /* ... */ };
    void show_employees(in string department) raises(BadDepartment);
    // ...
};
```

```
void
ReportGenerator_impl::
show_employees(const char * department)
throw(CORBA::SystemException, ReportGenerator::BadDepartment)
{
    EmployeeFinder_var ef = ...;

    // Locate department and get list of employee names...
    for (each emp_name in list) {
        Details_var d = ef->get_details(emp_name);  // Dubious!
        // Show employee's details...
    }
}
```

```
void
ReportGenerator_impl::
show_employees(const char * department)
throw(CORBA::SystemException, ReportGenerator::BadDepartment)
{
    try {
        EmployeeFinder_var ef = ...;

        // Locate department and get list of employee names...
        for (each emp_name in list) {
            try {
                Details_var d = ef->get_details(emp_name);
                // Show employee's details...
            } catch (const EmployeeFinder::BadEmployee &) {
                // Ignore bad employee and try next one...
            }
        }
    } catch (const CORBA::Exception &) {
        // Other CORBA exceptions are dealt with higher up.
        throw;
    } catch (...) { // This really is an assertion failure
                    // because it indicates a bug in the code
```

```
        write_error_log_report();
        throw CORBA::INTERNAL();
    }
}
```

Tie Classes

The C++ mapping offers an alternative way to implement servants.

A tie class replaces inheritance from the skeleton with delegation:

```
class Thermometer_impl {    // NO inheritance here!
public:
    // Usual CORBA operation implementation here...
};

// ...

Thermometer_impl * servantp = new Thermometer_impl;
POA_CCS::Thermometer_tie<Thermometer_impl> tie(servantp);
CCS::Thermometer_var = tie._this();
```

The tie instance delegates each call to the implementation instance, so the implementation instance does not have to inherit from a skeleton.

The IDL compiler generates ties with the **`--tie`** option.

Clean Server Shutdown

The **ORB** object contains **shutdown** and **destroy** operations that permit clean server shutdown:

```
interface ORB {
    void shutdown(in boolean wait_for_completion);
    void destroy();
    // ...
};
```

- With a false parameter, **shutdown** initiates ORB shutdown and returns immediately.
- With a true parameter, **shutdown** initiates ORB shutdown and does not return until shutdown is complete.

ORB shutdown stops accepting new requests, allows requests in progress to complete, and destroys all object adapters.

You *must* call **destroy** before leaving **main**!

```
#include <iostream.h>
#include <OB/CORBA.h>
#include "Age.h"

// Servant class definition here...

CORBA::ORB_var orb;     // Global, OK!

int
main(int argc, char * argv[])
{
    int status = 0;     // Return value from main()

    try {
        // Initialize ORB
        orb = CORBA::ORB_init(argc, argv);

        // Get Root POA, etc., and initialize application...

        // Accept requests
        orb->run();         // orb->shutdown(false) may be called
                            // from elsewhere, such as another
                            // thread, a signal handler, or as
                            // part of an operation.
    } catch (...) {
```

```
        status = 1;
    }

    // Don't invoke CORBA operations from here on, it won't work!

    if (!CORBA::is_nil(orb)) {  // If we created an ORB...
        try {
            orb->destroy();     // Wait for shutdown to complete
                                // and destroy ORB
        } catch (const CORBA::Exception &) {
            status = 1;
        }
    }
    // Do application-specific cleanup here...

    return status;
}
```

Handling Signals (UNIX)

To react to signals and terminate cleanly, call **`shutdown`** from within the signal handler:

```
extern "C"
void handler(int)
{
    try {
        if (!CORBA::is_nil(orb))
            orb->shutdown(false);
    } catch (...) {
        // Can't throw here...
    }
}
```

You can install the signal handler on entry to **`main`**.

You should handle at least **`SIGINT`**, **`SIGHUP`**, and **`SIGTERM`**.

Do not call **`shutdown(true)`** or **`destroy`** from within a signal handler!

```
struct sigaction sa;              // New signal state

sa.sa_handler = handler;          // Set handler function
sigfillset(&sa.sa_mask);          // Mask all other signals
                                  // while handler runs
sa.sa_flags = 0 | SA_RESTART;     // Restart interrupted syscalls

if (sigaction(SIGINT, &sa, (struct sigaction *)0) == -1)
    abort();
if (sigaction(SIGHUP, &sa, (struct sigaction *)0) == -1)
    abort();
if (sigaction(SIGTERM, &sa, (struct sigaction *)0) == -1)
    abort();

// Initialize ORB, etc...
```

```
extern "C"
void handler(int)
{
    // Ignore further signals
    struct sigaction ignore;
    ignore.sa_handler = SIG_IGN;
    sigemptyset(&ignore.sa_mask);
    ignore.sa_flags = 0;
    if (sigaction(SIGINT, &ignore, (struct sigaction *)0) == -1)
        abort();
    if (sigaction(SIGTERM, &ignore, (struct sigaction *)0) == -1)
        abort();
    if (sigaction(SIGHUP, &ignore, (struct sigaction *)0) == -1)
        abort();

    // Terminate event loop
    try {
        if (!CORBA::is_nil(orb))
            orb->shutdown(false);
    } catch (...) {
        // Can't throw here...
    }
}
```

Handling Signals (Windows)

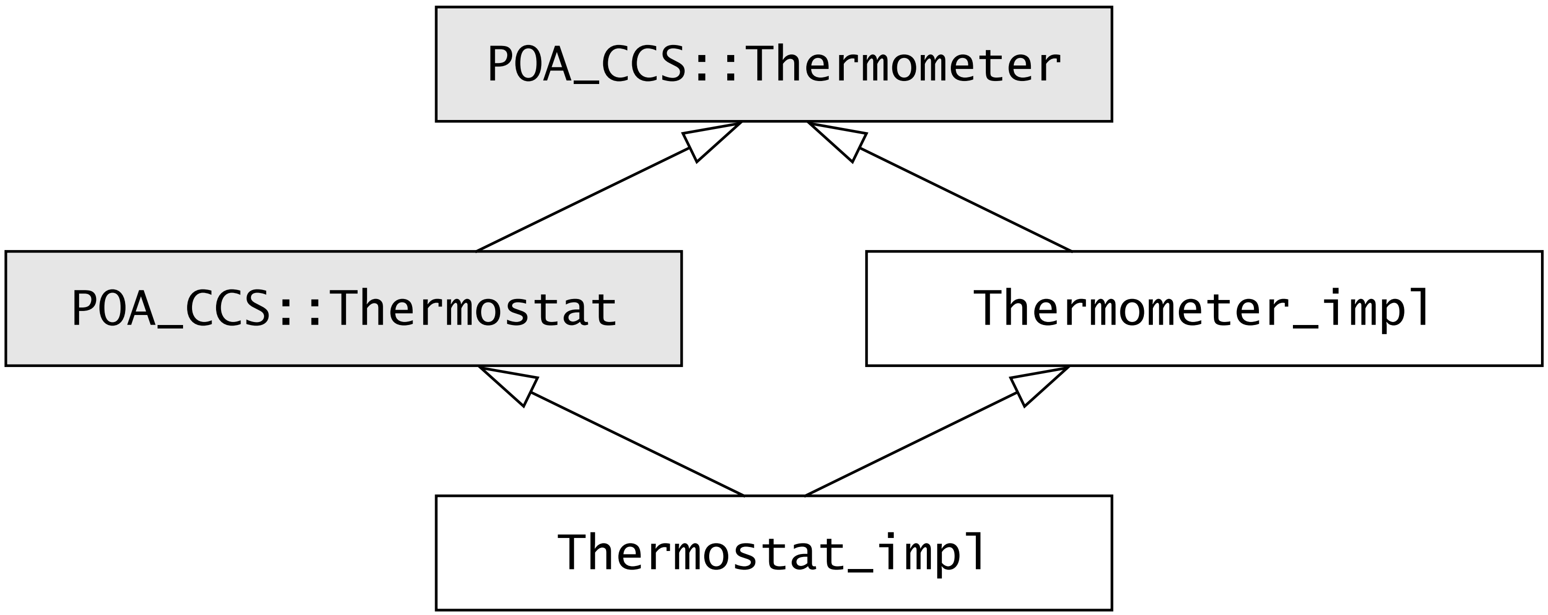
For Windows, use the following signal handler:

```
BOOL
handler(DWORD)
{
    // Inform JTC of presence of new thread
    JTCAdoptCurrentThread adopt;

    // Terminate event loop
    try {
        if (!CORBA::is_nil(orb))
            orb->shutdown(false);
    } catch (...) {
        // Can't throw here...
    }
    return TRUE;
}
```

```
BOOL rc = SetConsoleCtrlHandler((PHANDLER_ROUTINE)handler, TRUE);
if (!rc) {
    // Could not install handler
    abort();
}
```

Implementation Inheritance

If you are implementing base and derived interfaces in the same server, you can use implementation inheritance:

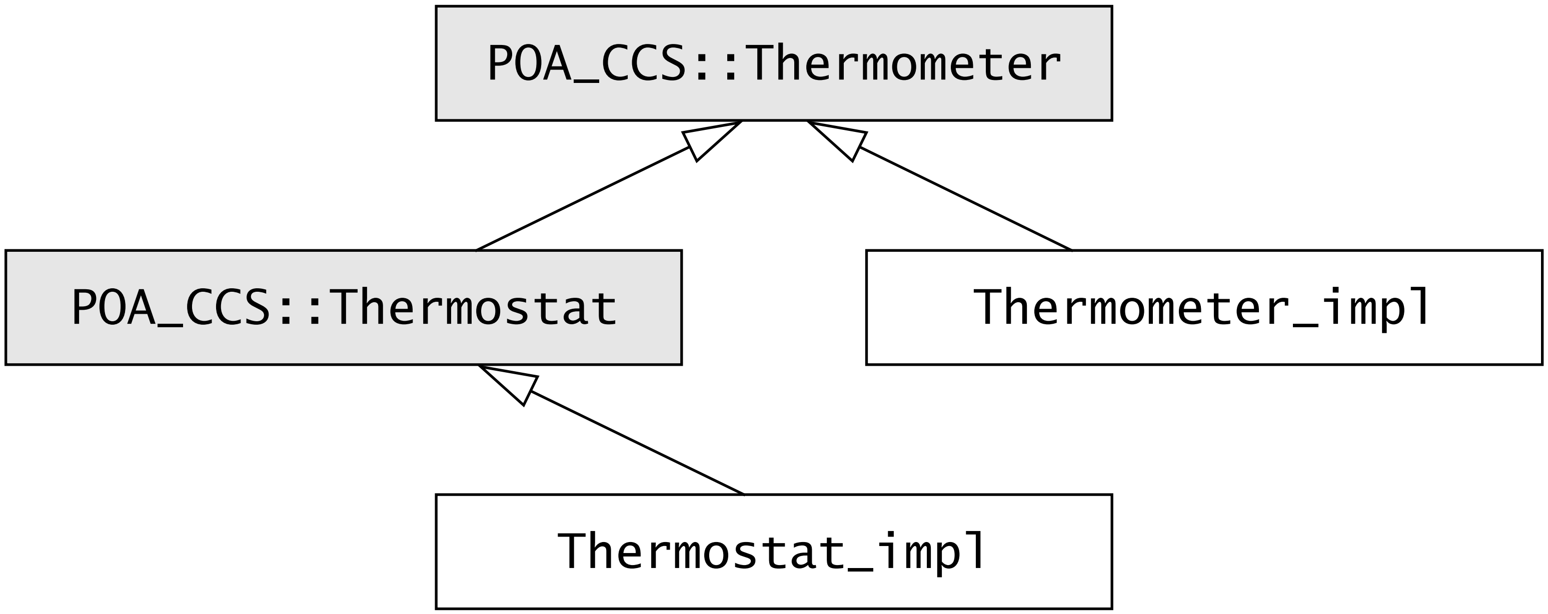


`Thermometer_impl` implements pure virtual functions inherited from **`POA_CCS::Thermometer`**, and **`Thermostat_impl`** implements pure virtual functions inherited from **`POA_CSS::Thermostat`**.

```
class Thermometer_impl : public virtual POA_CCS::Thermometer {
// ...
};

class Thermostat_impl : public virtual POA_CCS::Thermostat,
                        public virtual Thermometer_impl {
// ...
}
```

Interface Inheritance

You can choose to use interface inheritance:



`Thermometer_impl` implements five virtual functions, and **`Thermostat_impl`** implements seven virtual functions.

```
class Thermometer_impl : public virtual POA_CCS::Thermometer {
// ...
};

class Thermostat_impl : public virtual POA_CCS::Thermostat {
// ...
}
```

Compiling and Linking

To create a server executable, you must compile the application code, skeleton code, and the stub code. Typical compile commands are:

```
c++ -I. -I/opt/OB/include -c server.cc
c++ -I. -I/opt/OB/include -c MyIDL_skel.cpp
c++ -I. -I/opt/OB/include -c MyIDL.cpp
```

The exact flags and compile command vary with the platform.

To link the server, you must link against `libOB`:

```
c++ -o server server.o MyIDL_skel.o MyIDL.o -L/opt/OB/lib -lOB
```

If you are using JThreads/C++, you also need to link against the JTC library and the native threads library. For example:

```
c++ -o server server.o MyIDL_skel.o MyIDL.o -L/opt/OB/lib \
-lOB -lJTC -lpthread
```

```cpp
class Controller_impl : public virtual POA_CCS::Controller {
public:
    // ...

    // Helper functions to allow thermometers and
    // thermostats to add themselves to the m_assets map
    // and to remove themselves again.
    void add_impl(CCS::AssetType anum, Thermometer_impl * tip);
    void remove_impl(CCS::AssetType anum);

private:
    // Map of known servants
    typedef map<CCS::AssetType, Thermometer_impl *> AssetMap;
    AssetMap m_assets;

    // ...
};
```

```
AssetMap::iterator where;
where = m_assets.find(28);
if (where != m_assets.end())
    // Found it, where points at map entry
```

```
AssetMap::iterator where;   // Iterator for asset map
where = find_if(
            m_assets.begin(), m_assets.end(),
            StrFinder(CCS::Controller::LOCATION, "some_string")
        );
if (where != m_assets.end())
    // Found it...
```

```
// Helper function to read the location from a device.

CCS::LocType
Thermometer_impl::
get_loc()
{
    char buf[32];
    if (ICP_get(m_anum, "location", buf, sizeof(buf)) != 0)
        abort();
    return CORBA::string_dup(buf);
}

// IDL location attribute accessor.

CCS::LocType
Thermometer_impl::
location() throw(CORBA::SystemException)
{
    return get_loc();
}
```

```
// Helper function to set a thermostat's nominal temperature.

CCS::TempType
Thermostat_impl::
set_nominal_temp(CCS::TempType new_temp)
throw(CORBA::SystemException, CCS::Thermostat::BadTemp)
{
    short old_temp;

    // We need to return the previous nominal temperature,
    // so we first read the current nominal temperature before
    // changing it.
    if (ICP_get(
        m_anum, "nominal_temp", &old_temp, sizeof(old_temp)
       ) != 0) {
        abort();
    }

    // Now set the nominal temperature to the new value.
    if (ICP_set(m_anum, "nominal_temp", &new_temp) != 0) {

        // If ICP_set() failed, read this thermostat's
        // minimum and maximum so we can initialize the
        // BadTemp exception.
        CCS::Thermostat::BtData btd;
```

```
        ICP_get(
            m_anum, "MIN_TEMP",
            &btd.min_permitted, sizeof(btd.min_permitted)
        );
        ICP_get(
            m_anum, "MAX_TEMP",
            &btd.max_permitted, sizeof(btd.max_permitted)
        );
        btd.requested = new_temp;
        btd.error_msg = CORBA::string_dup(
            new_temp > btd.max_permitted ? "Too hot" : "Too cold"
        );
        throw CCS::Thermostat::BadTemp(btd);
    }
    return old_temp;
}
```

```
// IDL list operation.

CCS::Controller::ThermometerSeq *
Controller_impl::
list() throw(CORBA::SystemException)
{
    // Create a new thermometer sequence. Because we know
    // the number of elements we will put onto the sequence,
    // we use the maximum constructor.
    CCS::Controller::ThermometerSeq_var listv
        = new CCS::Controller::ThermometerSeq(m_assets.size());
    listv->length(m_assets.size());

    // Loop over the m_assets map and create a
    // reference for each device.
    CORBA::ULong count = 0;
    AssetMap::iterator i;
    for (i = m_assets.begin(); i != m_assets.end(); ++i)
        listv[count++] = i->second->_this();
    return listv._retn();
}
```

```
// IDL change operation.

void
Controller_impl::
change(
    const CCS::Controller::ThermostatSeq &  tlist,
    CORBA::Short                            delta
) throw(CORBA::SystemException, CCS::Controller::EChange)
{
    CCS::Controller::EChange ec;    // Just in case we need it

    // We cannot add a delta value to a thermostat's temperature
    // directly, so for each thermostat, we read the nominal
    // temperature, add the delta value to it, and write
    // it back again.
    CORBA::ULong i;
    for (i = 0; i < tlist.length(); ++i) {
        if (CORBA::is_nil(tlist[i]))
            continue;                           // Skip nil references

        // Read nominal temp and update it.
        CCS::TempType tnom = tlist[i]->get_nominal();
        tnom += delta;
        try {
            tlist[i]->set_nominal(tnom);
```

```
        }
        catch (const CCS::Thermostat::BadTemp & bt) {
            // If the update failed because the temperature
            // is out of range, we add the thermostat's info
            // to the errors sequence.
            CORBA::ULong len = ec.errors.length();
            ec.errors.length(len + 1);
            ec.errors[len].tmstat_ref = tlist[i];
            ec.errors[len].info = bt.details;
        }
    }

    // If we encountered errors in the above loop,
    // we will have added elements to the errors sequence.
    if (ec.errors.length() != 0)
        throw ec;
}
```

```
// IDL find operation

void
Controller_impl::
find(CCS::Controller::SearchSeq & slist)
throw(CORBA::SystemException)
{
    // Loop over input list and look up each device.
    CORBA::ULong listlen = slist.length();
    CORBA::ULong i;
    for (i = 0; i < listlen; ++i) {

        AssetMap::iterator where;   // Iterator for asset map
        int num_found = 0;          // Num matched per iteration

        // Assume we will not find a matching device.
        slist[i].device = CCS::Thermometer::_nil();

        // Work out whether we are searching by asset,
        // model, or location.
        CCS::Controller::SearchCriterion sc = slist[i].key._d();
        if (sc == CCS::Controller::ASSET) {
            // Search for matching asset number.
            where = m_assets.find(slist[i].key.asset_num());
            if (where != m_assets.end())
```

```
            slist[i].device = where->second->_this();
    } else {
        // Search for model or location string.
        const char * search_str;
        if (sc == CCS::Controller::LOCATION)
            search_str = slist[i].key.loc();
        else
            search_str = slist[i].key.model_desc();

        // Find first matching device (if any).
        where = find_if(
                    m_assets.begin(), m_assets.end(),
                    StrFinder(sc, search_str)
                );

        // While there are matches...
        while (where != m_assets.end()) {
            if (num_found == 0) {
                // First match overwrites reference
                // in search record.
                slist[i].device = where->second->_this();
            } else {
                // Each further match appends a new
                // element to the search sequence.
                CORBA::ULong len = slist.length();
```

```
                    slist.length(len + 1);
                    slist[len].key = slist[i].key;
                    slist[len].device = where->second->_this();
                }
                ++num_found;

                // Find next matching device with this key.
                where = find_if(
                            ++where, m_assets.end(),
                            StrFinder(sc, search_str)
                        );
            }
        }
    }
}
```

```
#ifndef server_HH_
#define server_HH_

#include <map>

#ifdef HAVE_STDLIB_H
#   include <stdlib.h>
#endif

#include "CCS_skel.h"

#ifdef _MSC_VER
using namespace std;
#endif

class Controller_impl;

class Thermometer_impl : public virtual POA_CCS::Thermometer {
public:
    // IDL attributes
    virtual CCS::ModelType  model()
                                throw(CORBA::SystemException);
    virtual CCS::AssetType  asset_num()
                                throw(CORBA::SystemException);
    virtual CCS::TempType   temperature()
```

```
                                    throw(CORBA::SystemException);
    virtual CCS::LocType    location()
                                    throw(CORBA::SystemException);
    virtual void            location(const char * loc)
                                    throw(CORBA::SystemException);

    // Constructor and destructor
    Thermometer_impl(CCS::AssetType anum, const char * location);
    virtual ~Thermometer_impl();

    static Controller_impl *    m_ctrl; // My controller

protected:
    const CCS::AssetType        m_anum; // My asset number

private:
    // Helper functions
    CCS::ModelType  get_model();
    CCS::TempType   get_temp();
    CCS::LocType    get_loc();
    void            set_loc(const char * new_loc);

    // Copy and assignment not supported
    Thermometer_impl(const Thermometer_impl &);
    void operator=(const Thermometer_impl &);
```

```
};

class Thermostat_impl :
    public virtual POA_CCS::Thermostat,
    public virtual Thermometer_impl {
public:
    // IDL operations
    virtual CCS::TempType   get_nominal()
                                throw(CORBA::SystemException);
    virtual CCS::TempType   set_nominal(
                                CCS::TempType new_temp
                            ) throw(
                                CORBA::SystemException,
                                CCS::Thermostat::BadTemp
                            );

    // Constructor and destructor
    Thermostat_impl(
        CCS::AssetType  anum,
        const char *    location,
        CCS::TempType   nominal_temp
    );
    virtual ~Thermostat_impl() {}

private:
```

```
    // Helper functions
    CCS::TempType   get_nominal_temp();
    CCS::TempType   set_nominal_temp(CCS::TempType new_temp)
                        throw(CCS::Thermostat::BadTemp);

    // Copy and assignment not supported
    Thermostat_impl(const Thermostat_impl &);
    void operator=(const Thermostat_impl &);
};

class Controller_impl : public virtual POA_CCS::Controller {
public:
    // IDL operations
    virtual CCS::Controller::ThermometerSeq *
                list() throw(CORBA::SystemException);
    virtual void
                find(CCS::Controller::SearchSeq & slist)
                    throw(CORBA::SystemException);
    virtual void
                change(
                    const CCS::Controller::ThermostatSeq & tlist,
                    CORBA::Short                           delta
                ) throw(
                    CORBA::SystemException,
                    CCS::Controller::EChange
```

```
            );

    // Constructor and destructor
    Controller_impl() {}
    virtual ~Controller_impl() {}

    // Helper functions to allow thermometers and
    // thermostats to add themselves to the m_assets map
    // and to remove themselves again.
    void add_impl(CCS::AssetType anum, Thermometer_impl * tip);
    void remove_impl(CCS::AssetType anum);

private:
    // Map of known servants
    typedef map<CCS::AssetType, Thermometer_impl *> AssetMap;
    AssetMap m_assets;

    // Copy and assignment not supported
    Controller_impl(const Controller_impl &);
    void operator=(const Controller_impl &);

    // Function object for the find_if algorithm to search for
    // devices by location and model string.
    class StrFinder {
    public:
```

```
    StrFinder(
        CCS::Controller::SearchCriterion    sc,
        const char *                        str
    ) : m_sc(sc), m_str(str) {}
    bool operator()(
        pair<const CCS::AssetType, Thermometer_impl *> & p
    ) const
    {
        switch (m_sc) {
        case CCS::Controller::LOCATION:
            return strcmp(p.second->location(), m_str) == 0;
            break;
        case CCS::Controller::MODEL:
            return strcmp(p.second->model(), m_str) == 0;
            break;
        default:
            abort();    // Precondition violation
        }
        return 0;       // Stops compiler warning
    }
private:
    CCS::Controller::SearchCriterion    m_sc;
    const char *                        m_str;
```

```
    };
};

#endif
```

```
#include    <OB/CORBA.h>

#include    <algorithm>
#include    <signal.h>
#include    <string>
#include    <fstream>

#if defined(HAVE_STD_IOSTREAM) || defined(HAVE_STD_STL)
using namespace std;
#endif

#include    "icp.h"
#include    "server.h"

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

// Helper function to write a stringified reference to a file.

void
write_ref(const char * filename, const char * reference)
{
    ofstream file(filename);
    if (!file) {
        string msg("Error opening ");
        msg += filename;
```

```
        throw msg.c_str();
    }
    file << reference << endl;
    if (!file) {
        string msg("Error writing ");
        msg += filename;
        throw msg.c_str();
    }
    file.close();
    if (!file) {
        string msg("Error closing ");
        msg += filename;
        throw msg.c_str();
    }
}

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

Controller_impl * Thermometer_impl::m_ctrl; // static member

// Helper function to read the model string from a device.

CCS::ModelType
Thermometer_impl::
get_model()
```

```
{
    char buf[32];
    if (ICP_get(m_anum, "model", buf, sizeof(buf)) != 0)
        abort();
    return CORBA::string_dup(buf);
}

// Helper function to read the temperature from a device.

CCS::TempType
Thermometer_impl::
get_temp()
{
    short temp;
    if (ICP_get(m_anum, "temperature", &temp, sizeof(temp)) != 0)
        abort();
    return temp;
}

// Helper function to read the location from a device.

CCS::LocType
Thermometer_impl::
get_loc()
{
```

```
    char buf[32];
    if (ICP_get(m_anum, "location", buf, sizeof(buf)) != 0)
        abort();
    return CORBA::string_dup(buf);
}

// Helper function to set the location of a device.

void
Thermometer_impl::
set_loc(const char * loc)
{
    if (ICP_set(m_anum, "location", loc) != 0)
        abort();
}

// Constructor.

Thermometer_impl::
Thermometer_impl(
    CCS::AssetType      anum,
    const char *        location
) : m_anum(anum)
{
    if (ICP_online(anum) != 0)    // Mark device as on-line
```

```
        abort();
    set_loc(location);            // Set its location
    m_ctrl->add_impl(anum, this); // Add self to controller's map
}

// Destructor.

Thermometer_impl::
~Thermometer_impl()
{
    try {
        m_ctrl->remove_impl(m_anum); // Remove self from map
        ICP_offline(m_anum);         // Mark device as off-line
    } catch (...) {
        abort();        // Prevent exceptions from escaping
    }
}

// IDL model attribute.

CCS::ModelType
Thermometer_impl::
model() throw(CORBA::SystemException)
{
    return get_model();
```

```
}

// IDL asset_num attribute.

CCS::AssetType
Thermometer_impl::
asset_num() throw(CORBA::SystemException)
{
    return m_anum;
}

// IDL temperature attribute.

CCS::TempType
Thermometer_impl::
temperature() throw(CORBA::SystemException)
{
    return get_temp();
}

// IDL location attribute accessor.

CCS::LocType
Thermometer_impl::
location() throw(CORBA::SystemException)
```

```
{
    return get_loc();
}

// IDL location attribute modifier.

void
Thermometer_impl::
location(const char * loc) throw(CORBA::SystemException)
{
    set_loc(loc);
}

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

// Helper function to get a thermostat's nominal temperature.

CCS::TempType
Thermostat_impl::
get_nominal_temp()
{
    short temp;
    if (ICP_get(m_anum, "nominal_temp", &temp,sizeof(temp)) != 0)
        abort();
    return temp;
```

```
}

// Helper function to set a thermostat's nominal temperature.

CCS::TempType
Thermostat_impl::
set_nominal_temp(CCS::TempType new_temp)
throw(CCS::Thermostat::BadTemp)
{
    short old_temp;

    // We need to return the previous nominal temperature,
    // so we first read the current nominal temperature before
    // changing it.
    if (ICP_get(
        m_anum, "nominal_temp", &old_temp, sizeof(old_temp)
       ) != 0) {
        abort();
    }

    // Now set the nominal temperature to the new value.
    if (ICP_set(m_anum, "nominal_temp", &new_temp) != 0) {

        // If ICP_set() failed, read this thermostat's
        // minimum and maximum so we can initialize the
```

```
        // BadTemp exception.
        CCS::Thermostat::BtData btd;
        ICP_get(
            m_anum, "MIN_TEMP",
            &btd.min_permitted, sizeof(btd.min_permitted)
        );
        ICP_get(
            m_anum, "MAX_TEMP",
            &btd.max_permitted, sizeof(btd.max_permitted)
        );
        btd.requested = new_temp;
        btd.error_msg = CORBA::string_dup(
            new_temp > btd.max_permitted ? "Too hot" : "Too cold"
        );
        throw CCS::Thermostat::BadTemp(btd);
    }
    return old_temp;
}

// Constructor.

Thermostat_impl::
Thermostat_impl(
    CCS::AssetType      anum,
    const char *        location,
```

```
    CCS::TempType       nominal_temp
) : Thermometer_impl(anum, location)
{
    // Base Thermometer_impl constructor does most of the
    // work, so we need only set the nominal temperature here.
    set_nominal_temp(nominal_temp);
}

// IDL get_nominal operation.

CCS::TempType
Thermostat_impl::
get_nominal() throw(CORBA::SystemException)
{
    return get_nominal_temp();
}

// IDL set_nominal operation.

CCS::TempType
Thermostat_impl::
set_nominal(CCS::TempType new_temp)
throw(CORBA::SystemException, CCS::Thermostat::BadTemp)
{
    return set_nominal_temp(new_temp);
```

```
}

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

// Helper function for thermometers and thermostats to
// add themselves to the m_assets map.

void
Controller_impl::
add_impl(CCS::AssetType anum, Thermometer_impl * tip)
{
    m_assets[anum] = tip;
}

// Helper function for thermometers and thermostats to
// remove themselves from the m_assets map.

void
Controller_impl::
remove_impl(CCS::AssetType anum)
{
    m_assets.erase(anum);
}

// IDL list operation.
```

```
CCS::Controller::ThermometerSeq *
Controller_impl::
list() throw(CORBA::SystemException)
{
    // Create a new thermometer sequence. Because we know
    // the number of elements we will put onto the sequence,
    // we use the maximum constructor.
    CCS::Controller::ThermometerSeq_var listv
        = new CCS::Controller::ThermometerSeq(m_assets.size());
    listv->length(m_assets.size());

    // Loop over the m_assets map and create a
    // reference for each device.
    CORBA::ULong count = 0;
    AssetMap::iterator i;
    for (i = m_assets.begin(); i != m_assets.end(); ++i)
        listv[count++] = i->second->_this();
    return listv._retn();
}

// IDL change operation.

void
Controller_impl::
```

```
change(
    const CCS::Controller::ThermostatSeq &  tlist,
    CORBA::Short                            delta
) throw(CORBA::SystemException, CCS::Controller::EChange)
{
    CCS::Controller::EChange ec;    // Just in case we need it

    // We cannot add a delta value to a thermostat's temperature
    // directly, so for each thermostat, we read the nominal
    // temperature, add the delta value to it, and write
    // it back again.
    CORBA::ULong i;
    for (i = 0; i < tlist.length(); ++i) {
        if (CORBA::is_nil(tlist[i]))
            continue;                            // Skip nil references

        // Read nominal temp and update it.
        CCS::TempType tnom = tlist[i]->get_nominal();
        tnom += delta;
        try {
            tlist[i]->set_nominal(tnom);
        }
        catch (const CCS::Thermostat::BadTemp & bt) {
            // If the update failed because the temperature
            // is out of range, we add the thermostat's info
```

```
                // to the errors sequence.
                CORBA::ULong len = ec.errors.length();
                ec.errors.length(len + 1);
                ec.errors[len].tmstat_ref = tlist[i];
                ec.errors[len].info = bt.details;
            }
        }

    // If we encountered errors in the above loop,
    // we will have added elements to the errors sequence.
    if (ec.errors.length() != 0)
        throw ec;
}

// IDL find operation

void
Controller_impl::
find(CCS::Controller::SearchSeq & slist)
throw(CORBA::SystemException)
{
    // Loop over input list and look up each device.
    CORBA::ULong listlen = slist.length();
    CORBA::ULong i;
    for (i = 0; i < listlen; ++i) {
```

```
AssetMap::iterator where;   // Iterator for asset map
int num_found = 0;          // Num matched per iteration

// Assume we will not find a matching device.
slist[i].device = CCS::Thermometer::_nil();

// Work out whether we are searching by asset,
// model, or location.
CCS::Controller::SearchCriterion sc = slist[i].key._d();
if (sc == CCS::Controller::ASSET) {
    // Search for matching asset number.
    where = m_assets.find(slist[i].key.asset_num());
    if (where != m_assets.end())
        slist[i].device = where->second->_this();
} else {
    // Search for model or location string.
    const char * search_str;
    if (sc == CCS::Controller::LOCATION)
        search_str = slist[i].key.loc();
    else
        search_str = slist[i].key.model_desc();

    // Find first matching device (if any).
    where = find_if(
```

```
                m_assets.begin(), m_assets.end(),
                StrFinder(sc, search_str)
            );

    // While there are matches...
    while (where != m_assets.end()) {
        if (num_found == 0) {
            // First match overwrites reference
            // in search record.
            slist[i].device = where->second->_this();
        } else {
            // Each further match appends a new
            // element to the search sequence.
            CORBA::ULong len = slist.length();
            slist.length(len + 1);
            slist[len].key = slist[i].key;
            slist[len].device = where->second->_this();
        }
        ++num_found;

        // Find next matching device with this key.
        where = find_if(
                    ++where, m_assets.end(),
                    StrFinder(sc, search_str)
                );
```

```
            }
        }
    }
}

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

void
run(CORBA::ORB_ptr orb)
{
    // Get reference to Root POA.
    CORBA::Object_var obj
        = orb->resolve_initial_references("RootPOA");
    PortableServer::POA_var poa
        = PortableServer::POA::_narrow(obj);

    // Create a controller and set static m_ctrl member
    // for thermostats and thermometers.
    Controller_impl ctrl_servant;
    Thermometer_impl::m_ctrl = &ctrl_servant;

    // Write controller stringified reference to ctrl.ref.
    CCS::Controller_var ctrl = ctrl_servant._this();
    CORBA::String_var str = orb->object_to_string(ctrl);
    write_ref("ctrl.ref", str);
```

```
    // Create a few devices. (Thermometers have odd asset
    // numbers, thermostats have even asset numbers.)
    Thermometer_impl thermo1(2029, "Deep Thought");
    Thermometer_impl thermo2(8053, "HAL");
    Thermometer_impl thermo3(1027, "ENIAC");

    Thermostat_impl tmstat1(3032, "Colossus", 68);
    Thermostat_impl tmstat2(4026, "ENIAC", 60);
    Thermostat_impl tmstat3(4088, "ENIAC", 50);
    Thermostat_impl tmstat4(8042, "HAL", 40);

    // Write a thermostat reference to tmstat.ref.
    CCS::Thermostat_var tmstat = tmstat1._this();
    str = orb->object_to_string(tmstat);
    write_ref("tmstat.ref", str);

    // Activate POA manager
    PortableServer::POAManager_var mgr = poa->the_POAManager();
    mgr->activate();

    // Accept requests
    orb->run();
}
```

```
//-----------------------------------------------------------------

static CORBA::ORB_var orb; // Global, so handler can see it.

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

#ifdef WIN32
BOOL
handler(DWORD)
{
    // Inform JTC of presence of new thread
    JTCAdoptCurrentThread adopt;

    // Terminate event loop
    try {
        if (!CORBA::is_nil(orb))
            orb->shutdown(false);
    } catch (...) {
        // Can't throw here...
    }
    return TRUE;
}
#else
extern "C"
void handler(int)
```

```
{
    // Ignore further signals
    struct sigaction ignore;
    ignore.sa_handler = SIG_IGN;
    sigemptyset(&ignore.sa_mask);
    ignore.sa_flags = 0;
    if (sigaction(SIGINT, &ignore, (struct sigaction *)0) == -1)
        abort();
    if (sigaction(SIGTERM, &ignore, (struct sigaction *)0) == -1)
        abort();
    if (sigaction(SIGHUP, &ignore, (struct sigaction *)0) == -1)
        abort();

    // Terminate event loop
    try {
        if (!CORBA::is_nil(orb))
            orb->shutdown(false);
    } catch (...) {
        // Can't throw here...
    }
}
#endif

//----------------------------------------------------------------
```

```
int
main(int argc, char* argv[])
{
    // Install signal handler for cleanup
#ifdef WIN32
    BOOL rc = SetConsoleCtrlHandler((PHANDLER_ROUTINE)handler, TRU
E);
    if (!rc)
        abort();
#else
    struct sigaction sa;                // New signal state
    sa.sa_handler = handler;            // Set handler function
    sigfillset(&sa.sa_mask);            // Mask all other signals
                                        // while handler runs
    sa.sa_flags = 0 | SA_RESTART;       //
 Restart interrupted syscalls

    if (sigaction(SIGINT, &sa, (struct sigaction *)0) == -1)
        abort();
    if (sigaction(SIGHUP, &sa, (struct sigaction *)0) == -1)
        abort();
    if (sigaction(SIGTERM, &sa, (struct sigaction *)0) == -1)
        abort();
#endif
```

```
    // Initialize the ORB and start working...
    int status = 0;
    try {
        orb = CORBA::ORB_init(argc, argv);
        run(orb);
    } catch (const CORBA::Exception & ex) {
        cerr << "Uncaught CORBA exception: " << ex << endl;
        status = 1;
    } catch (...) {
        cerr << "Uncaught non-CORBA exception" << endl;
        status = 1;
    }

    // Destroy the ORB.
    if (!CORBA::is_nil(orb)) {
        try {
            orb->destroy();
        } catch (const CORBA::Exception & ex) {
            cerr << "Cannot destroy ORB: " << ex << endl;
            status = 1;
        }
    }

    return status;
}
```

Interface Overview

The Portable Object Adapter provides a number of core interfaces, all part of the **`PortableServer`** module:

- **`POA`**
- **`POAManager`**
- **`Servant`**
- POA Policies (seven interfaces)
- Servant Managers (three interfaces)
- **`POACurrent`**
- **`AdapterActivator`**

Of these, the first five are used regularly in almost every server; **`POACurrent`** and **`AdapterActivator`** support advanced or unusual implementation techniques.

```
module PortableServer {
    native Servant;
    // ...
};
```

Functions of a POA

Each POA forms a namespace for servants.

All servants that use the same POA share common implementation characteristics, determined by the POA's policies. (The Root POA has a fixed set of policies.)

Each servant has exactly one POA, but many servants may share the same POA.

The POA tracks the relationship between object references, object IDs, and servants (and so is intimately involved in their life cycle).

POAs map between object references and the associated object ID and servants and map an incoming request onto the correct servant that incarnates the corresponding CORBA object.

Functions of a POA Manager

A POA manager is associated with a POA when the POA is created. Thereafter, the POA manager for a POA cannot be changed.

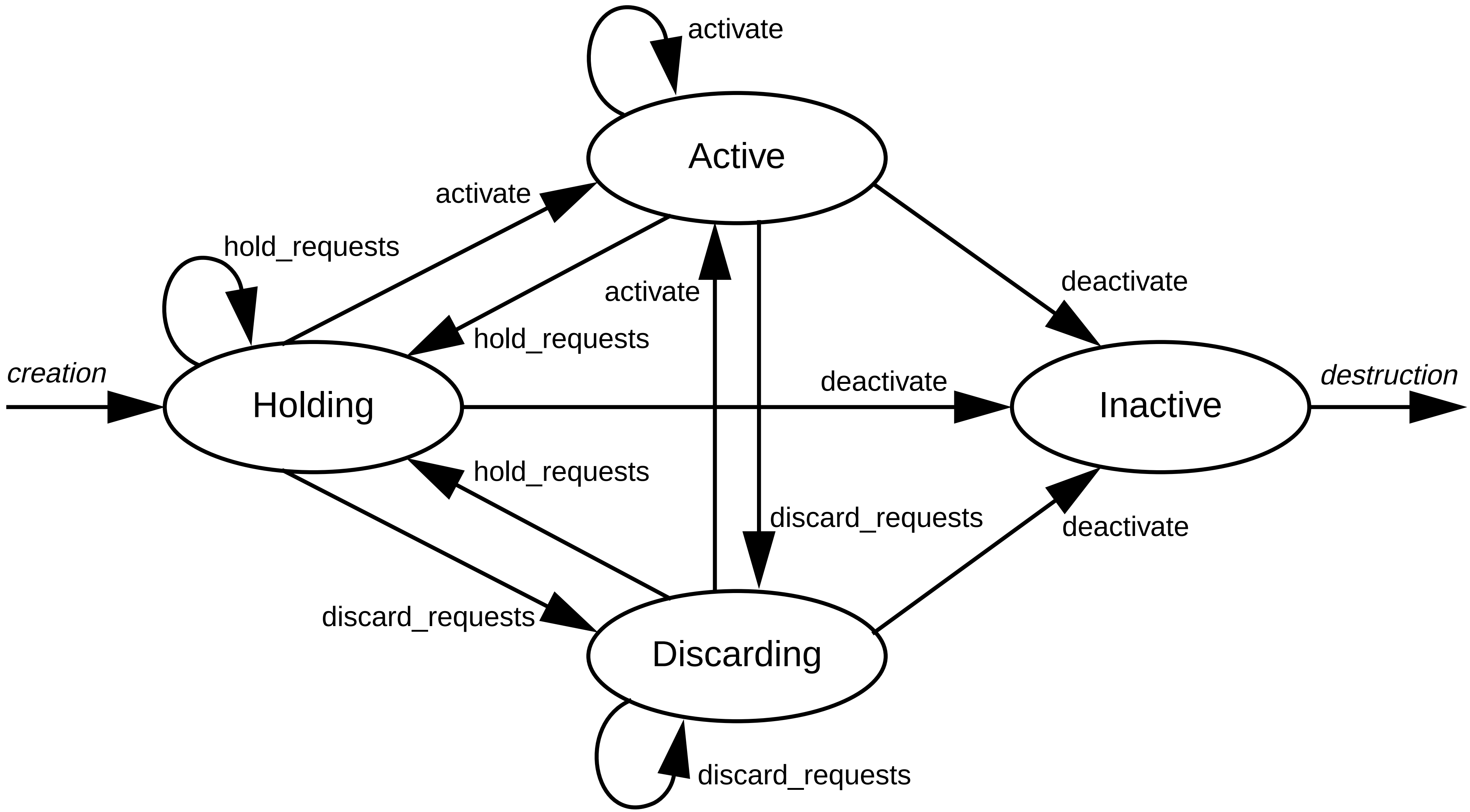
A POA manager controls the flow of requests into one or more POAs.

A POA manager is associated with a POA when the POA is created. Thereafter, the POA manager for a POA cannot be changed.

A POA manager is in one of four possible states:

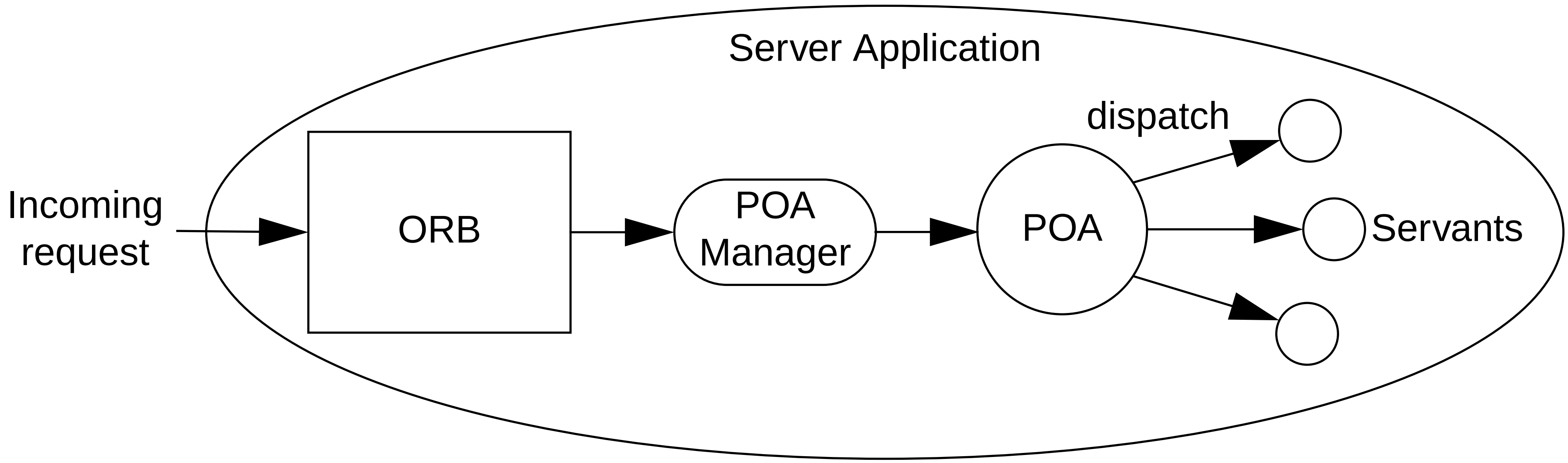
- **Active**: Requests are processed normally
- **Holding**: Requests are queued
- **Discarding**: Requests are rejected with `TRANSIENT`
- **Inactive**: Requests are rejected; clients may be redirected to a different server instance to try again.

POA Manager State Transitions



```
module PortableServer {
    // ...
    interface POAManager {
        exception AdapterInactive {};

        enum State { HOLDING, ACTIVE, DISCARDING, INACTIVE };

        State   get_state();
        void    activate() raises(AdapterInactive);
        void    hold_requests(in boolean wait_for_completion)
                    raises(AdapterInactive);
        void    discard_requests(in boolean wait_for_completion)
                    raises(AdapterInactive);
        void    deactivate(
                    in boolean etherealize_objects,
                    in boolean wait_for_completion
                ) raises(AdapterInactive);
    };
};
```

Request Flow

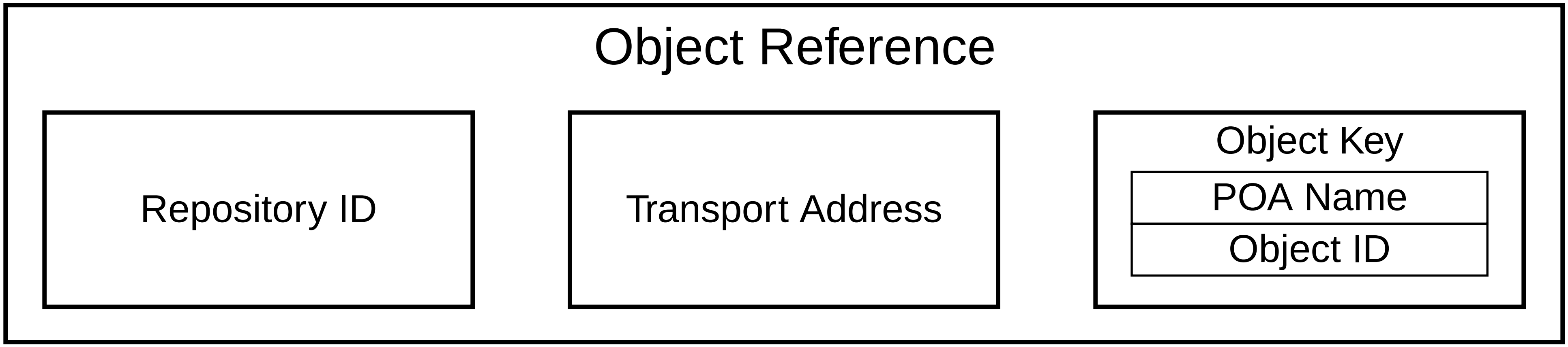


Conceptually, incoming requests are directed toward a particular ORB.

If the ORB is accepting requests, it passes the request to a POA manager.

The POA manager (if it is in the active state) passes the request to a POA, which in turn passes it to the correct servant.

Contents of an Object Reference



Conceptually, an object reference contains the following information:

- Repository ID (optional, identifies interface type)
- Addressing information (identifies a transport endpoint)
- Object key (identifies POA and object within the POA)

The object key is in a proprietary format, specific to each ORB.

Policies

Each POA is associated with a set of seven policies when it is created.

Policies control implementation characteristics of object references and servants.

The **CORBA** module provides a **Policy** abstract base interface:

```
module CORBA {
    typedef unsigned long PolicyType;

    interface Policy {
        readonly attribute PolicyType policy_type;
        Policy  copy();
        void    destroy();
    };
    typedef sequence<Policy> PolicyList;
    // ...
};
```

POA Policies

The **PortableServer** module contains seven interfaces that are derived from the **CORBA::Policy** interface:

- **LifespanPolicy**
- **IdAssignmentPolicy**
- **IdUniquenessPolicy**
- **ImplicitActivationPolicy**
- **RequestProcessingPolicy**
- **ServantRetentionPolicy**
- **ThreadPolicy**

POA Creation

The POA interface allows you to create other POAs:

```
module PortableServer {
    interface POAManager;

    exception AdapterAlreadyExists {};
    exception InvalidPolicy { unsigned short index; };
    interface POA {
        POA create_POA(
                in string               adapter_name,
                in POAManager           manager,
                in CORBA::PolicyList    policies;
            ) raises(AdapterAlreadyExists, InvalidPolicy);
        readonly attribute POAManager   the_POAManager;
        // ...
    };
    // ...
};
```

```
// Initialize ORB and get Root POA
CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
CORBA::Object_var obj =
    orb->resolve_initial_references("RootPOA");
PortableServer::POA_var root_poa =
    PortableServer::POA::_narrow(obj);
assert(!CORBA::is_nil(root_poa));

// Create empty policy list
CORBA::PolicyList policy_list;

// Create Controller POA
PortableServer::POA_var ctrl_poa = root_poa->create_POA(
                    "Controller",
                    PortableServer::POAManager::_nil(),
                    policy_list);

// Create Thermometer POA as a child of the Controller POA
PortableServer::POA_var thermometer_poa = ctrl_poa->create_POA(
                    "Thermometers",
                    PortableServer::POAManager::_nil(),
                    policy_list);

// Create Thermostat POA as a child of the Controller POA
```

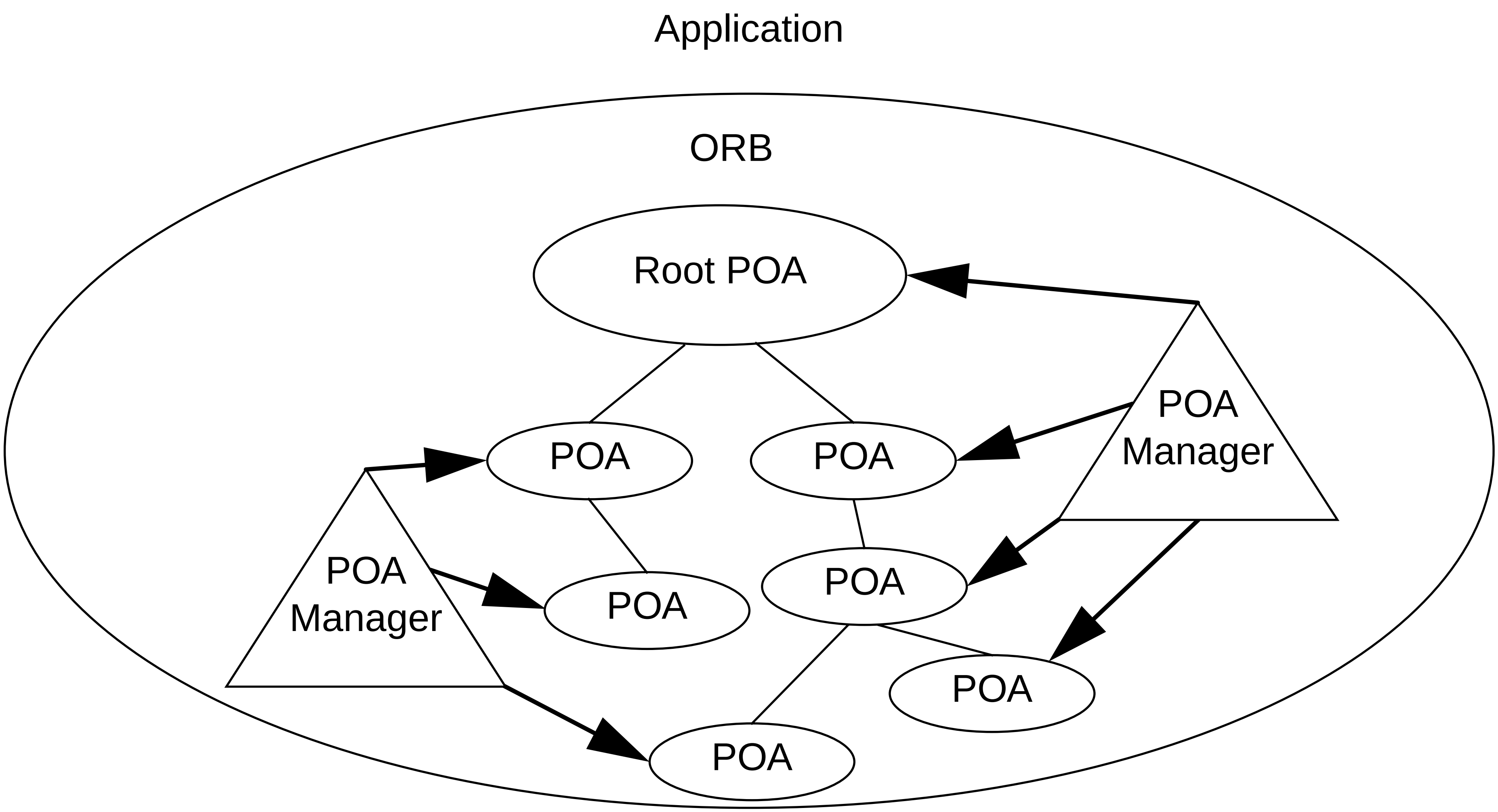
```
PortableServer::POA_var thermostat_poa = ctrl_poa->create_POA(
                    "Thermostats",
                    PortableServer::POAManager::_nil(),
                    policy_list);
```

```
// Initialize ORB and get Root POA
PortableServer::POA_var root_poa = ...;

// Create empty policy list
CORBA::PolicyList policy_list;

// Get the Root POA manager
PortableServer::POAManager_var mgr = root_poa->the_POAManager();

// Create Controller POA, using the Root POA's manager
PortableServer::POA_var ctrl_poa = root_poa->create_POA(
                    "Controller",
                    mgr,
                    policy_list);

// Create Thermometer POA as a child of the Controller POA,
// using the Root POA's manager
PortableServer::POA_var thermometer_poa = ctrl_poa->create_POA(
                    "Thermometers",
                    mgr,
                    policy_list);

// Create Thermostat POA as a child of the Controller POA,
// using the Root POA's manager
```

```
PortableServer::POA_var thermostat_poa = ctrl_poa->create_POA(
                    "Thermostats",
                    mgr,
                    policy_list);
```

POA-to-POA Manager Relationship

With **`create_POA`**, you can create arbitrary POA-to-POA manager relationships:



The Life Span Policy

The life span policy controls whether references are transient or persistent. The default is **TRANSIENT**.

```
enum LifespanPolicyValue { TRANSIENT, PERSISTENT };

interface LifespanPolicy : CORBA::Policy {
    readonly attribute LifespanPolicyValue value;
};
```

You should combine **PERSISTENT** with:

- **ImplicitActivationPolicy**: **NO_IMPLICIT_ACTIVATION**
- **IdAssignmentPolicy**: **USER_ID**

You should combine **TRANSIENT** with:

- **ImplicitActivationPolicy**: **IMPLICIT_ACTIVATION**
- **IDAssignmentPolicy**: **SYSTEM_ID**

The ID Assignment Policy

The ID assignment policy controls whether object IDs are created by the ORB or by the application. The default is **SYSTEM_ID**.

```
enum IdAssignmentPolicyValue { USER_ID, SYSTEM_ID };

interface IdAssignmentPolicy : CORBA::Policy {
    readonly attribute IdAssignmentPolicyValue value;
};
```

You should combine **USER_ID** with:

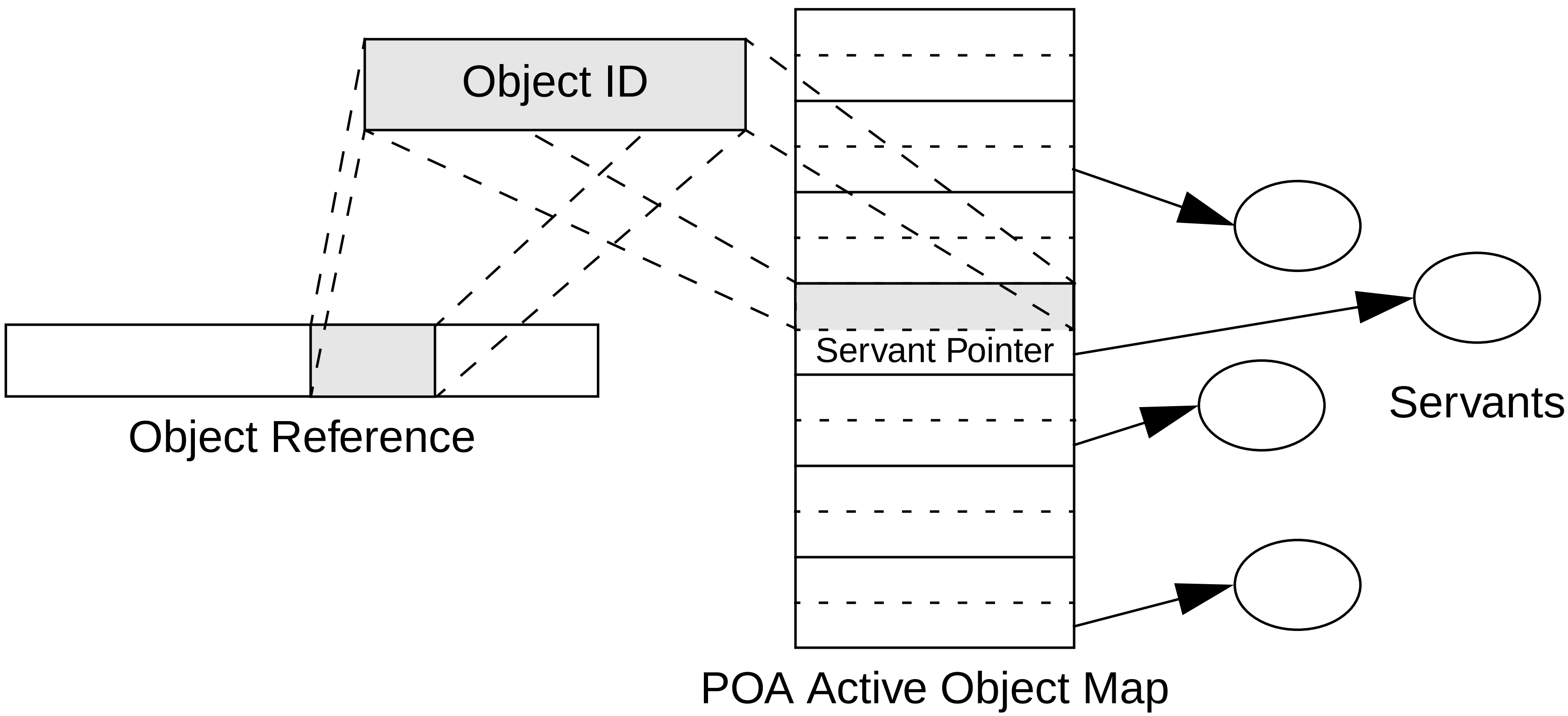
- **ImplicitActivationPolicy**: **NO_IMPLICIT_ACTIVATION**
- **LifespanPolicy**: **PERSISTENT**

You should combine **SYSTEM_ID** with:

- **ImplicitActivationPolicy**: **IMPLICIT_ACTIVATION**
- **LifespanPolicy**: **TRANSIENT**

The Active Object Map (AOM)

Each POA with a servant retention policy of **`RETAIN`** has an AOM. The AOM provides a mapping from object IDs to servant addresses:



The object ID is the index into the map and sent by clients with each incoming request as part of the object key.

The ID Uniqueness Policy

The ID uniqueness policy controls whether a single servant can represent more than one CORBA object. The default is **UNIQUE_ID**.):

```
enum IdUniquenessPolicyValue { UNIQUE_ID, MULTIPLE_ID };

interface IdUniquenessPolicy : CORBA::Policy {
    readonly attribute IdUniquenessPolicyValue value;
};
```

- **UNIQUE_ID** enforces that no servant can appear in the AOM more than once.
- **MULTIPLE_ID** permits the same servant to be pointed at by more than one entry in the AOM.

For **MULTIPLE_ID**, an operation implementation can ask its POA for the object ID for the current invocation.

The Servant Retention Policy

The servant retention policy controls whether a POA has an AOM. (The default is **RETAIN**).

```
enum ServantRetentionPolicyValue { RETAIN, NON_RETAIN };

interface ServantRetentionPolicy : CORBA::Policy {
    readonly attribute ServantRetentionPolicyValue value;
};
```

NON_RETAIN requires a request processing policy of **USE_DEFAULT_SERVANT** or **USE_SERVANT_MANAGER**.

With **NON_RETAIN** and **USE_DEFAULT_SERVANT**, the POA maps incoming requests to a nominated default servant.

With **NON_RETAIN** and **USE_SERVANT_MANAGER**, the POA calls back into the server application code for each incoming request to map the request to a particular servant.

The Request Processing Policy

The request processing policy controls whether a POA uses static AOM, a default servant, or instantiates servants on demand. (The default is **USE_ACTIVE_OBJECT_MAP_ONLY**.)

```
enum RequestProcessingPolicyValue {
            USE_ACTIVE_OBJECT_MAP_ONLY,
            USE_DEFAULT_SERVANT,
            USE_SERVANT_MANAGER
     };

interface RequestProcessingPolicy : CORBA::Policy {
    readonly attribute RequestProcessingPolicyValue value;
};
```

USE_DEFAULT_SERVANT requires **MULTIPLE_ID**.

USE_ACTIVE_OBJECT_MAP_ONLY requires **RETAIN**.

The Implicit Activation Policy

The implicit activation policy controls whether a servant can be activated implicitly or must be activated explicitly. (The default is **NO_IMPLICIT_ACTIVATION**).

```
enum ImplicitActivationPolicyValue {
        IMPLICIT_ACTIVATION, NO_IMPLICIT_ACTIVATION
};

interface ImplicitActivationPolicy : CORBA::Policy {
    readonly attribute ImplicitActivationPolicyValue value;
};
```

- For **IMPLICIT_ACTIVATION** (which requires **RETAIN** and **SYSTEM_ID**), servants are added to AOM by calling **_this**.
- For **NO_IMPLICIT_ACTIVATION**, servants must be activated with a separate API call before you can obtain their object reference.

The Thread Policy

The thread policy controls whether requests are dispatched single-threaded (are serialized) or whether the ORB chooses a threading model for request dispatch. The default is **ORB_CTRL_MODEL**.

```
enum ThreadPolicyValue {
        ORB_CTRL_MODEL, SINGLE_THREAD_MODEL
};

interface ThreadPolicy : CORBA::Policy {
    readonly attribute ThreadPolicyValue value;
};
```

- **ORB_CTRL_MODEL** allows the ORB to chose a threading model. (Different ORBs will exhibit different behavior.)
- **SINGLE_THREAD_MODEL** serializes all requests on a per-POA basis.

The Root POA Policies

The Root POA has a fixed set of policies:

Life Span Policy	`TRANSIENT`
ID Assignment Policy	`SYSTEM_ID`
ID Uniqueness Policy	`UNIQUE_ID`
Servant Retention Policy	`RETAIN`
Request Processing Policy	`USE_ACTIVE_OBJECT_MAP_ONLY`
Implicit Activation Policy	`IMPLICIT_ACTIVATION`
Thread Policy	`ORB_CTRL_MODEL`

Note that the implicit activation policy does *not* have the default value.

The Root POA is useful for transient objects only.

If you want to create persistent objects or use more sophisticated implementation techniques, you must create your own POAs.

Policy Creation

The POA interface provides a factory operation for each policy.

Each factory operation returns a policy with the requested value, for example:

```
module PortableServer {
    // ...
    interface POA {
        // ...
        LifespanPolicy create_lifespan_policy(
                                    in LifespanPolicyValue value
                               );
        // ...
};
```

You must call **destroy** on the returned object reference before you release it.

```
PortableServer::POA_var root_poa = ...;     // Get Root POA

CORBA::PolicyList pl;
pl.length(3);

pl[0] = root_poa->create_lifespan_policy(
            PortableServer::PERSISTENT
        );
pl[1] = root_poa->create_id_assignment_policy(
            PortableServer::USER_ID
        );
pl[2] = root_poa->create_thread_policy(
            PortableServer::SINGLE_THREAD_MODEL
        );

PortableServer::POA_var CCS_poa =
    root_poa->create_POA("CCS", nil_mgr, pl);

pl[0]->destroy();
pl[1]->destroy();
pl[2]->destroy();
```

Creating Persistent Objects

Persistent objects have references that continue to work across server shut-down and re-start.

To create persistent references, you must:

- use **`PERSISTENT`**, **`USER_ID`**, and **`NO_IMPLICIT_ACTIVATION`**
- use the same POA name and object ID for each persistent object
- link the object IDs to the objects’ identity and persistent state
- explicitly activate each servant
- allow the server to be found by clients by
  - either specifying a port number (direct binding)
  - or using the implementation repository (IMR)

It sounds complicated, but it is easy!

Creating a Simple Persistent Server

- Use a separate POA for each interface.
- Create your POAs with the **`PERSISTENT`**, **`USER_ID`**, and **`NO_IMPLICIT_ACTIVATION`** policies.
- Override the **`_default_POA`** method on your servant class. (*Always* do this for all POAs other than the Root POA. If you have multiple ORBs, do this even for the Root POA on non-default ORBs.)
- Explicitly activate each servant with **`activate_object_with_id`**.
- Ensure that servants have unique IDs per POA. Use some part of each servant's state as the unique ID (the identity).
- Use the identity of each servant as its database key.

```
PortableServer::POA_ptr
create_persistent_POA(
    const char *            name,
    PortableServer::POA_ptr parent)
{
    // Create policy list for simple persistence
    CORBA::PolicyList pl;
    pl.length(3);
    pl[0] = parent->create_lifespan_policy(
                PortableServer::PERSISTENT
            );
    pl[1] = parent->create_id_assignment_policy(
                PortableServer::USER_ID
            );
    pl[2] = parent->create_thread_policy(
                PortableServer::SINGLE_THREAD_MODEL
            );

    // Get parent POA's POA manager
    PortableServer::POAManager_var pmanager
        = parent->the_POAManager();

    // Create new POA
    PortableServer::POA_var poa =
        parent->create_POA(name, pmanager, pl);
```

```
    // Clean up
    for (CORBA::ULong i = 0; i < pl.length(); ++i)
        pl[i]->destroy();

    return poa._retn();
}

int
main(int argc, char * argv[])
{
    // ...

    PortableServer POA_var root_poa = ...;  // Get Root POA

    // Create POAs for controller, thermometers, and thermostats.
    // The controller POA becomes the parent of the thermometer
    // and thermostat POAs.
    PortableServer::POA_var ctrl_poa
        = create_persistent_POA("Controller", root_poa);
    PortableServer::POA_var thermo_poa
        = create_persistent_POA("Thermometers", ctrl_poa);
    PortableServer::POA_var tstat_poa
        = create_persistent_POA("Thermostats", ctrl_poa);
```

```
    // Create servants...

    // Activate POA manager
    PortableServer::POAManager_var mgr
        = root_poa->the_POAManager();
    mgr->activate();

    // ...
}
```

Creating a Simple Persistent Server (cont.)

`PortableServer::ServantBase` (which is the ancestor of all servants) provides a default implementation of the **`_default_POA`** function.

The default implementation of **`_default_POA`** always returns the Root POA.

If you use POAs other than the Root POA, you *must* override **`_default_POA`** in the servant class to return the correct POA for the servant.

If you forget to override **`_default_POA`**, calls to **`_this`** and several other functions will return incorrect object references and implicitly register the servant with the Root POA as a transient object.

Always override **`_default_POA`** for servants that do not use the Root POA! If you use multiple ORBs, override it for *all* servants!

```
class Thermometer_impl : public virtual POA_CCS::Thermometer {
public:
    Thermometer_impl(/* ... */)
    {
        if (CORBA::is_nil(m_poa))
            throw "Thermometer_impl::m_poa not set!"
        // ...
    }

    static void
    poa(PortableServer::POA_ptr poa)
    {
        if (!CORBA::is_nil(m_poa))
            throw "Thermometer_impl::m_poa set more than once!"
        m_poa = PortableServer::POA::_duplicate(poa);
    }

    static PortableServer::POA_ptr
    poa()
    {
        if (CORBA::is_nil(m_poa))
            throw "Thermometer_impl::m_poa not set!"
        return m_poa;   // Note: no call to _duplicate() here!
    }
```

```
    virtual PortableServer::POA_ptr
    _default_POA()
    {
        return PortableServer::POA::_duplicate(m_poa);
    }
private:
    static PortableServer::POA_var m_poa;
    // ...
};

int
main(int argc, char * argv[])
{
    // ...
    PortableServer::POA_var thermo_poa
        = create_persistent_POA("Thermometers", ctrl_poa);
    Thermometer_impl::poa(thermo_poa);
    // ...
    PortableServer::POAManager_var mgr
        = root_poa->the_POAManager();
    mgr->activate();
    // ...
}
```

```
Thermometer_impl::poa()->create_reference_with_id(...);
```

Creating a Simple Persistent Server (cont.)

To explicitly activate a servant and create an entry for the servant in the AOM, call **`activate_object_with_id`**:

```
typedef sequence<octet> ObjectId;
// ...
interface POA {
    exception ObjectAlreadyActive {};
    exception ServantAlreadyActive {};
    exception WrongPolicy {};
    void activate_object_with_id(
            in ObjectId id, in Servant p_servant
        ) raises(
             ObjectAlreadyActive,
             ServantAlreadyActive,
             WrongPolicy
        );
    // ...
};
```

```
namespace PortableServer {
    // ...
    char *           ObjectId_to_string(const ObjectId &);
    CORBA::WChar *   ObjectId_to_wstring(const Object Id &);

    ObjectId *       string_to_ObjectId(const char *);
    ObjectId *       wstring_to_ObjectId(const CORBA::WChar *);
}
```

```
class Thermometer_impl : public virtual POA_CCS::Thermometer {
public:
    Thermometer_impl(CCS::AssetType anum /* , ... */);
    // ...
};

Thermometer_impl::
Thermometer_impl(CCS::AssetType anum /* , ... */)
{
    // ...
    ostrstream tmp;
    tmp << anum << ends;
    PortableServer::ObjectId_var oid
        = PortableServer::string_to_ObjectId(tmp.str());
    tmp.rdbuf()->freeze(0);
    m_poa->activate_object_with_id(oid, this);
}
```

```
Thermometer_impl * t1 = new Thermometer_impl(22);
```

Creating a Simple Persistent Server (cont.)

A servant's object ID acts as a key that links an object reference to the persistent state for the object.

- For read access to the object, you can retrieve the state of the servant in the constructor, or use lazy retrieval (to spread out initialization cost).
- For write access, you can write the updated state back immediately, or when the servant is destroyed, or when the server shuts down.

When to retrieve and update persistent state is up to your application.

The persistence mechanism can be anything you like, such as a database, text file, mapped memory, etc.

Creating a Simple Persistent Server (cont.)

POAs using the **PERSISTENT** policy write addressing information into each object reference.

You must ensure that the same server keeps using the same address and port; otherwise, references created by a previous server instance dangle:

- The host name written into the each IOR is obtained automatically.
- You can assign a specific port number using the **`-OAport`** option.

If you do not assign a port number, the server determines a port number dynamically (which is likely to change every time the server starts up).

If you do not have a working DNS, use **`-OAnumeric`**. This forces dotted-decimal addresses to be written into IORs.

Object Creation

Clients create new objects by invoking operations on a factory. The factory operation returns the reference to the new object. For example:

```
exception DuplicateAsset {};

interface ThermometerFactory {
    Thermometer create(in AssetType n) raises(DuplicateAsset);
};

interface ThermostatFactory {
    Thermostat create(in AssetType n, in TempType t)
        raises(DuplicateAsset, Thermostat::BadTemp);
};
```

As far as the ORB is concerned, object creation is just another operation invocation.

```
CCS::Thermometer_ptr
ThermometerFactory_impl::
create(CCS::AssetType n)
throw(CORBA::SystemException, CCS::DuplicateAsset)
{
    // Create database record for the new servant (if needed)
    // ...

    // Instantiate a new servant on the heap
    Thermometer_impl * thermo_p = new Thermometer_impl(n);

    // Activate the servant if it is persistent (and
    // activation is not done by the constructor)
    // ...

    return thermo_p->_this();
}
```

Destroying CORBA Objects

To permit clients to destroy a CORBA object, add a **`destroy`** operation to the interface:

```
interface Thermometer {
    // ...
    void destroy();
};
```

The implementation of destroy deactivates the servant and permanently removes its persistent state (if any).

Further invocations on the destroyed object raise **`OBJECT_NOT_EXIST`**.

As far as the ORB is concerned, **`destroy`** is an ordinary operation with no special significance.

```
interface ThermometerFactory {
    Thermometer create(in AssetType n);
    void        destroy(in AssetType n);    // Not recommended!
};
```

Destroying CORBA Objects (cont.)

The POA holds a pointer to each servant in the AOM. You remove the AOM entry for a servant by calling **deactivate_object**:

```
interface POA {
    // ...
    void deactivate_object(in ObjectId oid)
            raises(ObjectNotActive, WrongPolicy);
};
```

Once deactivated, further requests for the same object raise **OBJECT_NOT_EXIST** (because no entry can be found in the AOM).

Once the association between the reference and the servant is removed, you can delete the servant.

deactivate_object does not remove the AOM entry immediately, but waits until all operations on the servant have completed.

Never call **delete this;** from inside **destroy**!

```
void
Thermometer_impl::
destroy() throw(CORBA::SystemException)
{
    ostrstream tmp;
    tmp << m_anum << ends;
    PortableServer::ObjectId_var oid
        = PortableServer::string_to_ObjectId(tmp.str()));
    tmp.rdbuf()->freeze(0);
    m_poa->deactivate_object(oid);              // Fine
    delete this;                                // Disaster!!!
}
```

Destroying CORBA Objects (cont.)

For multi-threaded servers, you must wait for all invocations to complete before you can physically destroy the servant.

To make this easier, the ORB provides a reference-counting mix-in class for servants:

```
class RefCountServantBase : public virtual ServantBase {
public:
                    ~RefCountServantBase();
    virtual void    _add_ref();
    virtual void    _remove_ref();
protected:
                    RefCountServantBase();
};
```

The ORB keeps a reference count for servants and calls **`delete`** on the servant once the reference count drops to zero.

```
class Thermometer_impl :
        public virtual POA_CCS::Thermometer,
        public virtual PortableServer::RefCountServantBase {
// ...
};
```

```
CCS::Thermometer_ptr
ThermometerFactory_impl::
create(AssetType n) throw(CORBA::SystemException)
{
    CCS::Thermometer_impl * thermo_p = new Thermometer_impl(...);
    // ...

    m_poa->activate_object_with_id(oid, thermo_p);
    thermo_p->_remove_ref();    // Drop ref count
    return thermo_p->_this();
}

void
Thermometer_impl::
destroy() throw(CORBA::SystemException)
{
    ostrstream tmp;
    tmp << m_anum << ends;
    PortableServer::ObjectId_var oid
        = PortableServer::string_to_ObjectId(tmp.str());
    tmp.rdbuf()->freeze(0);
    m_poa->deactivate_object(oid);              // Fine
}
```

Destroying CORBA Objects (cont.)

Destroying a persistent object implies destroying its persistent state.

Generally, you cannot remove persistent state as part of **`destroy`** (because other operations executing in parallel may still need it):

- It is best to destroy the persistent state from the servant's destructor.
- The servant destructor also runs when the server is shut down, so take care to destroy the persistent state only after a previous call to **`destroy`**.
- Use a boolean member variable to remember a previous call to **`destroy`**.

```
class Thermometer_impl :
        public virtual POA_CCS::Thermometer,
        public virtual PortableServer::RefCountServantBase {
public:
    Thermometer_impl(AssetType anum) :
        m_anum(anum), m_removed(false) { /* ... */ }
    ~Thermometer_impl();
    // ...
protected:
    AssetType m_anum;
    bool      m_removed;
};
```

```
Thermometer_impl::
~Thermometer_impl()
{
    if (m_removed) {
        // Destroy persistent state for this object...
    }
    // Release whatever other resources (not related to
    // persistent state) were used...
}
```

Deactivation and Servant Destruction

The POA offers a **destroy** operation:

```
interface POA {
    // ...
    void destroy(
                in boolean etherealize_objects,
                in boolean wait_for_completion
         );
};
```

Destroying a POA recursively destroys its descendant POAs.

If **wait_for_completion** is true, the call returns after all current requests have completed and all POAs are destroyed. Otherwise, the POA is destroyed immediately.

Calling **ORB::shutdown** or **ORB::destroy** implicitly calls **POA::destroy** on the Root POA.

```
// Create a new POA named 'name' and with 'parent' as its
// ancestor. The new POA shares its POA manager with
// its parent.

static PortableServer::POA_ptr
create_persistent_POA(
    const char *            name,
    PortableServer::POA_ptr parent)
{
    // Create policy list for simple persistence
    CORBA::PolicyList pl;
    CORBA::ULong len = pl.length();
    pl.length(len + 1);
    pl[len++] = parent->create_lifespan_policy(
                        PortableServer::PERSISTENT
                    );
    pl.length(len + 1);
    pl[len++] = parent->create_id_assignment_policy(
                        PortableServer::USER_ID
                    );
    pl.length(len + 1);
    pl[len++] = parent->create_thread_policy(
                        PortableServer::SINGLE_THREAD_MODEL
                    );
    pl.length(len + 1);
```

```
    pl[len++] = parent->create_implicit_activation_policy(
                                   PortableServer::NO_IMPLICIT_ACTIVATION
                                );

    // Get parent POA's POA manager
    PortableServer::POAManager_var pmanager
    = parent->the_POAManager();

    // Create new POA
    PortableServer::POA_var poa =
        parent->create_POA(name, pmanager, pl);

    // Clean up
    for (CORBA::ULong i = 0; i < len; ++i)
        pl[i]->destroy();

    return poa._retn();
}
```

```
CCS::Thermometer_ptr
Controller_impl::
create_thermometer(CCS::AssetType anum, const char * loc)
throw(CORBA::SystemException, CCS::Controller::DuplicateAsset)
{
    if (exists(anum))
        throw CCS::Controller::DuplicateAsset();
    if (anum % 2 == 0)
        throw CORBA::BAD_PARAM();   // ICS limitation
    if (ICP_online(anum) != 0)
        abort();
    if (ICP_set(anum, "location", loc) != 0)
        abort();
    Thermometer_impl * t = new Thermometer_impl(anum);
    PortableServer::ObjectId_var oid = make_oid(anum);
    Thermometer_impl::poa()->activate_object_with_id(oid, t);
    t->_remove_ref();

    return t->_this();
}
```

```
// IDL destroy operation.

void
Thermometer_impl::
destroy() throw(CORBA::SystemException)
{
    m_ctrl->remove_impl(m_anum);
    if (ICP_offline(m_anum) != 0)
        abort();
    PortableServer::ObjectId_var oid = make_oid(m_anum);
    PortableServer::POA_var poa = _default_POA();
    poa->deactivate_object(oid);
}
```

```
CCS::Thermostat_ptr
Controller_impl::
create_thermostat(
    CCS::AssetType  anum,
    const char*     loc,
    CCS::TempType   temp)
throw(
    CORBA::SystemException,
    CCS::Controller::DuplicateAsset,
    CCS::Thermostat::BadTemp)
{
    if (exists(anum))
        throw CCS::Controller::DuplicateAsset();
    if (anum % 2)
        throw CORBA::BAD_PARAM();   // ICS limitation
    if (ICP_online(anum) != 0)
        abort();
    if (ICP_set(anum, "location", loc) != 0)
        abort();
    // Set the nominal temperature.
    if (ICP_set(anum, "nominal_temp", &temp) != 0) {

        // If ICP_set() failed, read this thermostat's
        // minimum and maximum so we can initialize the
        // BadTemp exception.
```

```
        CCS::Thermostat::BtData btd;
        ICP_get(
            anum, "MIN_TEMP",
            &btd.min_permitted, sizeof(btd.min_permitted)
        );
        ICP_get(
            anum, "MAX_TEMP",
            &btd.max_permitted, sizeof(btd.max_permitted)
        );
        btd.requested = temp;
        btd.error_msg = CORBA::string_dup(
            temp > btd.max_permitted ? "Too hot" : "Too cold"
        );
        ICP_offline(anum);
        throw CCS::Thermostat::BadTemp(btd);
    }

    Thermostat_impl * t = new Thermostat_impl(anum);
    PortableServer::ObjectId_var oid = make_oid(anum);
    Thermostat_impl::poa()->activate_object_with_id(oid, t);
    t->_remove_ref();

    return t->_this();
}
```

Pre-Loading of Objects

The **USE_ACTIVE_OBJECT_MAP_ONLY** requires one servant per CORBA object, and requires all servants to be in memory at all times.

This forces the server to pre-instantiate all servants prior to entering its dispatch loop:

```
int main(int argc, char * argv[])
{
    // Initialize ORB, create POAs, etc...
    // Instantiate one servant per CORBA object:
    while (database_records_remain) {
        // Fetch record for a servant...
        // Instantiate and activate servant...
    }
    // ...
    orb->run();     // Start dispatch loop
    // ...
}
```

Servant Managers

Servant managers permit you to load servants into memory on demand, when they are needed.

Servant managers come in two flavors:

- **`ServantActivator`** (requires the **`RETAIN`** policy)

  The ORB makes a callback the first time a requests arrives for an object that is not in the AOM. The callback returns the servant to the ORB, and the ORB adds it to the AOM.

- **`ServantLocator`** (requires the **`NON_RETAIN`** policy)

  The ORB makes a callback every time a request arrives. The callback returns a servant for the request. Another callback is made once the request is complete. The association between request and servant is in effect only for the duration of single request.

```
module PortableServer {
    // ...

    interface ServantManager {};

    interface ServantActivator : ServantManager {
        // ...
    };

    interface ServantLocator : ServantManager {
        // ...
    };
};
```

Servant Activators

```
exception ForwardRequest {
    Object forward_reference;
};
interface ServantManager {};
interface ServantActivator : ServantManager {
    Servant incarnate(
                in ObjectId oid,
                in POA      adapter
            ) raises(ForwardRequest);

    void    etherealize(
                in ObjectId oid,
                in POA      adapter,
                in Servant  serv,
                in boolean  cleanup_in_progress,
                in boolean  remaining_activations
            );
};
```

Implementing a Servant Activator

The implementation of **`incarnate`** is usually very similar to a factory operation:

1. Use the object ID to locate the persistent state for the servant.
2. If the object ID does not exist, throw **`OBJECT_NOT_EXIST`**.
3. Instantiate a servant using the retrieved persistent state.
4. Return a pointer to the servant.

The implementation of **`etherealize`** gets rid of the servant:

1. Write the persistent state of the servant to the DB (unless you are using write-through).
2. If **`remaining_activations`** is false, call **`_remove_ref`** (or call **`delete`**, if the servant is not reference-counted).

```
class Activator_impl :
    public virtual POA_PortableServer::ServantActivator {
public:
    virtual PortableServer::Servant
    incarnate(
        const PortableServer::ObjectId & oid,
        PortableServer::POA_ptr          poa
    ) throw(CORBA::SystemException,
            PortableServer::ForwardRequest);

    virtual void
    etherealize(
        const PortableServer::ObjectId & oid,
        PortableServer::POA_ptr          poa,
        PortableServer::Servant          serv,
        CORBA::Boolean                   cleanup_in_progress,
        CORBA::Boolean                   remaining_activations
    ) throw(CORBA::SystemException);
};
```

```
PortableServer::Servant
Activator_impl::
incarnate(
    const PortableServer::ObjectId & oid,
    PortableServer::POA_ptr          poa
) throw(CORBA::SystemException, PortableServer::ForwardRequest)
{
    // Turn the OID into a string
    CORBA::String_var oid_str;
    try {
        oid_string = PortableServer::ObjectId_to_string(oid);
    } catch (const CORBA::BAD_PARAM &) {
        throw CORBA::OBJECT_NOT_EXIST(); // Malformed OID
    }

    // Use OID to look in the DB for the persistent state...
    if (object_not_found)
        throw CORBA::OBJECT_NOT_EXIST();

    // Use the state retrieved from the database to
    // instantiate a servant. The type of the servant may be
    // implicit in the POA, the object ID, or the database state.
    AssetType anum = ...;
    return new Thermometer_impl(anum, /* ... */);
}
```

```
void
Activator_impl::
etherealize(
    const PortableServer::ObjectId & oid,
    PortableServer::POA_ptr          poa,
    PortableServer::Servant          serv,
    CORBA::Boolean                   cleanup_in_progress,
    CORBA::Boolean                   remaining_activations
) throw(CORBA::SystemException)
{
    // Write updates (if any) for this object to database and
    // clean up any resources that may still be used by the
    // servant (or do this from the servant destructor)...
    if (!remaining_activations)
       serv->_remove_ref(); // Or delete serv, if not ref-counted
}
```

Use Cases for Servant Activators

Use servant activators if:

- you cannot afford to instantiate all servants up-front because it takes too long

  A servant activator distributes the cost of initialization over many calls, so the server can start up quicker.

- clients tend to be interested in only a small number of servants over the period the server is up

  If all objects provided by the server are eventually touched by clients, all servants end up in memory, so there is no saving in that case.

Servant activators are of interest mainly for servers that are started on demand.

Servant Manager Registration

You must register a servant manager with the POA before activating the POA's POA manager:

```
interface POA {
    // ...
    void               set_servant_manager(in ServantManager mgr)
                           raises(WrongPolicy);
    ServantManager     get_servant_manager() raises(WrongPolicy);
};
```

If you pass a servant activator, to **set_servant_manager**, the POA must use **USE_SERVANT_MANAGER** and **RETAIN**.

You can register the same servant manager with more than one POA.

You can set the servant manager only once; it remains attached to the POA until the POA is destroyed.

get_servant_manager returns the servant manager for a POA.

```
Activator_impl * ap = new Activator_impl;
PortableServer::ServantManager_var mgr_ref = ap->_this();
some_poa->set_servant_manager(mgr_ref);
```

Type Issues with Servant Managers

How does a servant manager know which type of interface is needed? Some options:

- Use a separate POA and separate servant manager class for each interface. The interface is implicit in the servant manager that is called.
- Use a separate POA for each interface but share the servant manager. Infer the interface from the POA name by reading the **`the_name`** attribute on the POA.
- Use a single POA and servant manager and add a type marker to the object ID. Use that type marker to infer which interface is needed.
- Store a type marker in the database with the persistent state.

The second option is usually easiest.

Servant Locators

```
native Cookie;
interface ServantLocator : ServantManager {
    Servant preinvoke(
                in ObjectId             oid,
                in POA                  adapter,
                in CORBA::Identifier    operation,
                out Cookie              the_cookie
            ) raises(ForwardRequest);

    void    postinvoke(
                in ObjectId             oid,
                in POA                  adapter,
                in CORBA::Identifier    operation,
                in Cookie               the_cookie,
                in Servant              serv
            );
};
```

Implementing Servant Locators

The implementation of **`preinvoke`** is usually very similar to a factory operation (or **`incarnate`**):

1. Use the POA, object ID, and operation name to locate the persistent state for the servant.
2. If the object does not exist, throw **`OBJECT_NOT_EXIST`**.
3. Instantiate a servant using the retrieved persistent state.
4. Return a pointer to the servant.

The implementation of **`postinvoke`** gets rid of the servant:

1. Write the persistent state of the servant to the DB (unless you are using write-through).
2. Call **`_remove_ref`** (or call **`delete`**, if the servant is not reference-counted).

```
// In the PortableServer namespace:
// typedef void * Cookie;

class Locator_impl :
    public virtual POA_PortableServer::ServantLocator,
    public virtual PortableServer::RefCountServantBase {
public:
    virtual PortableServer::Servant
    preinvoke(
        const PortableServer::ObjectId &         oid,
        PortableServer::POA_ptr                  adapter,
        const char *                             operation,
        PortableServer::ServantLocator::Cookie & the_cookie
    ) throw(CORBA::SystemException,
            PortableServer::ForwardRequest);

    virtual void
    postinvoke(
        const PortableServer::ObjectId &        oid,
        PortableServer::POA_ptr                 adapter,
        const char *                            operation,
        PortableServer::ServantLocator::Cookie  the_cookie,
        PortableServer::Servant                 the_servant
    ) throw(CORBA::SystemException);
};
```

```
PortableServer::Servant
Locator_impl::
preinvoke(
    const PortableServer::ObjectId &         oid,
    PortableServer::POA_ptr                  adapter,
    const char *                             operation,
    PortableServer::ServantLocator::Cookie & the_cookie
) throw(CORBA::SystemException, PortableServer::ForwardRequest)
{
    // Turn the OID into a string
    CORBA::String_var oid_str;
    try {
        oid_string = PortableServer::ObjectId_to_string(oid);
    } catch (const CORBA::BAD_PARAM &) {
        throw CORBA::OBJECT_NOT_EXIST(); // Malformed OID
    }

    // Use OID to look in the DB for the persistent state...
    if (object_not_found)
        throw CORBA::OBJECT_NOT_EXIST();

    // Use the state retrieved from the database to
    // instantiate a servant. The type of the servant may be
```

```
    // implicit in the POA, the object ID, or the database state.
    AssetType anum = ...;
    return new Thermometer_impl(anum, /* ... */);
}
```

Use Cases for Servant Locators

Advantages of servant locators:

- They provide precise control over the memory use of the server, regardless of the number of objects supported.
- **`preinvoke`** and **`postinvoke`** bracket every operation call, so you can do work in these operations that must be performed for every operation, for example:
    - initialization and cleanup
    - creation and destruction of network connections or similar
    - acquisition and release of mutexes
- You can implement servant caches that bound the number of servants in memory to the *n* most recently used ones (Evictor Pattern.)

Servant Managers and Collections

For operations such as **`Controller::list`**, you cannot iterate over a list of servants to return references to all objects because not all servants may be in memory.

Instead of instantiating a servant for each object just so you can call **`_this`** to create a reference, you can create a reference without instantiating a servant:

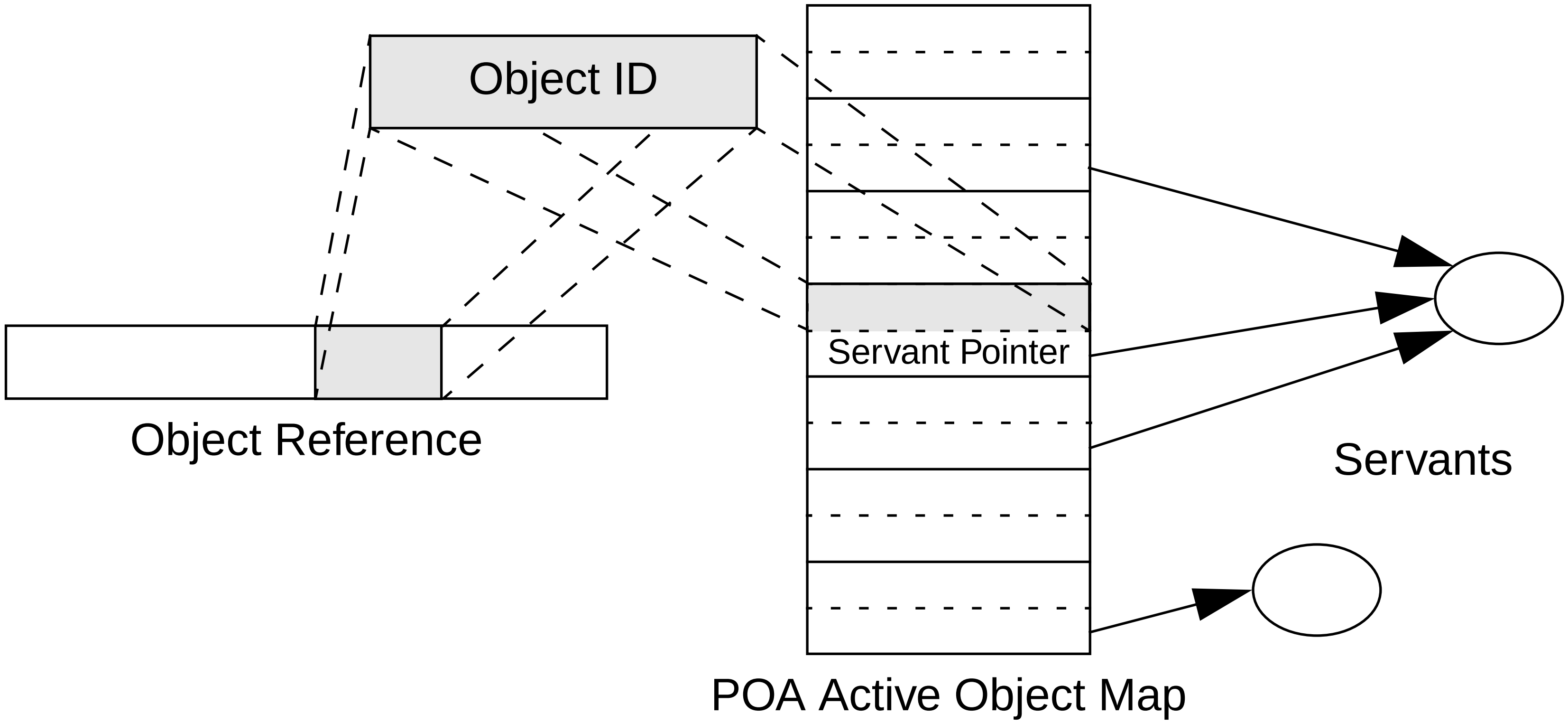
```
interface POA {
    // ...
    Object create_reference(in CORBA::RepositoryId intf)
            raises(WrongPolicy);

    Object create_reference_with_id(
                in ObjectId             oid,
                in CORBA::RepositoryId  intf
           ) raises(WrongPolicy);
};
```

```
CCS::Controller::ThermometerSeq *
Controller_impl::
list() throw(CORBA::SystemException)
{
    CCS::Controller::ThermometerSeq_var return_seq
        = new CCS::Controller::ThermometerSeq;
    CORBA::ULong index = 0;

    // Iterate over the database contents (or other list
    // of existing objects) and create reference for each.
    while (more objects remain) {
        // Get asset number from database and convert to OID.
        CCS::AssetType anum = ...;
        ostrstream ostr;
        ostr << anum << ends;
        PortableServer::ObjectId_var oid =
            PortableServer::string_to_ObjectId(ostr.str());
        ostr.rdbuf()->freeze(0);

        // Use object state to work out which type of device
        // we are dealing with and which POA to use.
        const char * rep_id;
        PortableServer::POA_var poa;
        if (device is a thermometer) {
            rep_id = "IDL:acme.com/CCS/Thermometer:1.0";
```

```
            poa = ...;  // Thermometer POA
        } else {
            rep_id = "IDL:acme.com/CCS/Thermostat:1.0";
            poa = ...;  // Thermostat POA
        }

        // Create reference
        CORBA::Object_var obj =
            poa->create_reference_with_id(oid, rep_id);
        // Narrow and store in our return sequence.
        return_seq->length(index + 1);
        if (device is a thermometer)
            return_seq[index++] = CCS::Thermometer::_narrow(obj);
        else
            return_seq[index++] = CCS::Thermostat::_narrow(obj);
    }
    return return_seq._retn();
}
```

One Servant for Many Objects

If you use the **`MULTIPLE_ID`** policy with **`RETAIN`**, a single servant can incarnate more than one object:



All CORBA objects that are incarnated by the same servant must have the same IDL interface.

The Current Object

The **`Current`** object provides information about the request context to an operation implementation:

```
module PortableServer {
    // ...
    exception NoContext {};

    interface Current : CORBA::Current {
            POA         get_POA() raises(NoContext);
            ObjectId    get_object_id() raises(NoContext);
    };
};
```

The **`get_POA`** and **`get_object_id`** operations must be called from within an executing operation (or attribute access) in a servant.

Note: You must resolve the Root POA before resolving **`POACurrent`**.

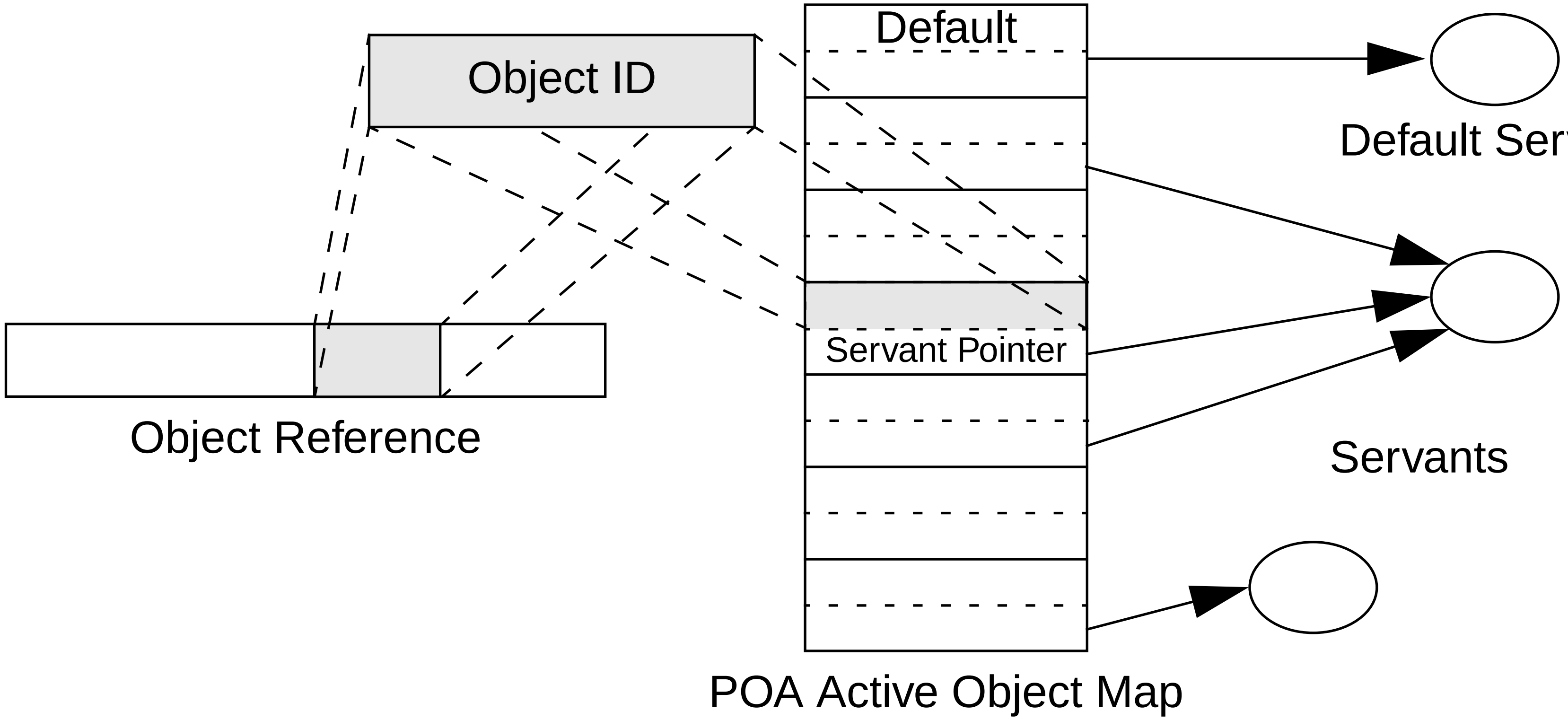
```
CORBA::Object_var obj =
    orb->resolve_initial_references("POACurrent");

PortableServer::Current_var cur =
    PortableServer::Current::_narrow(obj);
```

```
CCS::AssetType
Thermometer_impl::
get_anum()
throw(CORBA::SystemException)
{
    // Get object ID from Current object
    PortableServer::ObjectId_var oid =
        poa_current->get_object_id();

    // Check that ID is valid
    CORBA::String_var tmp;
    try {
        tmp = PortableServer::ObjectId_to_string(oid);
    } catch (const CORBA::BAD_PARAM &) {
        throw CORBA::OBJECT_NOT_EXIST();
    }

    // Turn string into asset number
    istrstream istr(tmp.in());
    CCS::AssetType anum;
    istr >> anum;
    if (str.fail())
        throw CORBA::OBJECT_NOT_EXIST();
    return anum;
}
```

```
CCS::LocType
Thermometer_impl::
location() throw(CORBA::SystemException)
{
    CCS::AssetType anum = get_anum();   // Who are we?

    // Get location string from the database
    CORBA::String_var loc = db_get_field(anum, "LocationField");
    return loc._retn();
}

void
Thermometer_impl::
location(const char * loc) throw(CORBA::SystemException)
{
    CCS::AssetType anum = get_anum();   // Who are we?

    // Set location string in the database
    db_set_field(anum, "LocationField", loc);
}
```

Default Servants

Default servants require **`MULTIPLE_ID`** and **`USE_DEFAULT_SERVANT`**:



Any request for which no explicit entry exists in the AOM is given to the default servant.

Use either **`RETAIN`** or **`NON_RETAIN`** with **`USE_DEFAULT_SERVANT`**.

```
interface POA {
    // ...
    Servant get_servant() raises(NoServant, WrongPolicy);
    void    set_servant(in Servant s) raises(WrongPolicy);
};
```

```
PortableServer::ServantBase_var servant
    = some_poa->get_servant();

// ServantBase_var destructor calls _remove_ref() eventually...

// If we want the actual type of the servant again, we must
// use a down-cast:
Thermometer_impl * dflt_serv =
    dynamic_cast<Thermometer_impl *>(servant);
```

Trade-Offs for Default Servants

Default servants offer a number of advantages:

- simple implementation
- POA and object ID can be obtained from **`Current`**
- ideal as a front end to a back-end store
- servant is completely stateless
- infinite scalability!

The downside:

- possibly slow access to servant state

POA Activators

You can create POAs on demand, similar to activating servants on demand:

```
module PortableServer {
  // ...
  interface AdapterActivator {
    boolean unknown_adapter(in POA parent, in string name);
  };
};
```

This is a callback interface you provide to the ORB.

If a request for an unknown POA arrives, the ORB invokes the **unknown_adapter** operation to allow you to create the POA.

```
class POA_Activator_impl :
    public virtual POA_PortableServer::AdapterActivator {
public:
                POA_Activator_impl() {}
    virtual     ~POA_Activator_impl() {}
    virtual CORBA::Boolean
                unknown_adapter(
                    PortableServer::POA_ptr parent,
                    const char *            name
                ) throw(CORBA::SystemException);
};
```

Implementing POA Activators

The **parent** parameter allows you to get details of the parent POA (particularly, the name of the parent POA).

The **name** parameter provides the name for the new POA.

While **unknown_adapter** is running, requests for the new adapter are held pending until the activator returns.

The implementation of the activator must decide on a set of policies for the new POA and instantiate it.

If optimistic caching is used, the activator must instantiate the servants for the POA. (If combined with **USE_SERVANT_MANAGER**, a subset of the servants can be instantiated instead.)

On success, the activator must return true to the ORB (which dispatches the request as usual.) A false return value raises **OBJECT_NOT_EXIST** in the client.

```
CORBA::Boolean
POA_Activator_impl::
unknown_adapter(
    PortableServer::POA_ptr parent,
    const char *            name
) throw(CORBA::SystemException)
{
    // Check which adapter is being activated and
    // create appropriate policies. (Might use pre-built
    // policy list here...)
    CORBA::PolicyList policies;
    if (strcmp(name, "Some_adapter") == 0) {
        // Create policies for "Some_adapter"...
    } else if (strcmp(name, "Some_other_adapter") == 0) {
        // Create policies for "Some_other_adapter"...
    } else {
        // Unknown POA name
        return false;
    }

    // Select POA manager for new adapter (parent POA
    // manager in this example).
    PortableSerer::POAManager_var mgr = parent->the_POAManager();

    // Create new POA.
```

```
    try {
        PortableServer::POA_var child =
            parent->create_POA(name, mgr, policies);
    } catch (const PortableServer::POA:AdapterAlreadyExists &) {
        return false;
    } catch (...) {
        return false;
    }

    // For optimistic caching, activate servants here...

    return true;
}
```

Registering POA Activators

An adapter activator must be registered by setting the POA's **`the_activator`** attribute:

```
interface POA {
    // ...
    attribute AdapterActivator the_activator;
};
```

You can change the adapter activator of an existing POA, including the Root POA.

By attaching an activator to all POAs, a request for a POA that is low in the POA hierarchy will automatically activate all parent POAs that are needed.

```
CORBA::Boolean
POA_Activator_impl::
unknown_adapter(
    PortableServer::POA_ptr parent,
    const char *            name
) throw(CORBA::SystemException)
{
    // ...

    // Create new POA.
    try {
        PortableServer::POA_var child =
            parent->create_POA(name, mgr, policies);
        PortableServer::AdapterActivator_var act = _this();
        child->the_activator(act);
    } catch (const PortableServer::POA:AdapterAlreadyExists &) {
        return false;
    } catch (...) {
        return false;
    }

    // ...
}
```

```
// ...

PortableServer::POA_var root_poa = ...;

// Create activator servant.
POA_Activator_impl act_servant;

// Register activator with Root POA.
PortableServer::AdapterActivator_var act = act_servant._this();
root_poa->the_activator(act);

// ...
```

Finding POAs

The **find_POA** operation locates a POA:

```
// In module PortableServer: typedef sequence<POA> POAList;
interface POA {
    // ...
    POA find_POA(in string name, in boolean activate_it)
            raises(AdapterNonExistent);
    readonly attribute POAList the_children;
    readonly attribute POA     the_parent;
};
```

You must invoke **find_POA** on the correct parent (because POA names are unique only within their parent POA).

If **activate_it** is true and the parent has an adapter activator, **unknown_adapter** will be called to create the child POA.

You can use this to instantiate all your POAs by simply calling **find_POA**.

```
// ...

PortableServer::POA_var root_poa = ...;

// Create activator servant.
POA_Activator_impl act_servant;

// Register activator with Root POA.
PortableServer::AdapterActivator_var act = act_servant._this();
root_poa->the_activator(act);

// Use find_POA to create a POA hierarchy. The POAs will be
// created by the adapter activator.
PortableServer::POA_var ctrl_poa
    = root_poa->find_POA("Controller", true);
PortableServer::POA_var thermometer_poa
    = ctrl_poa->find_POA("Thermometers", true);
PortableServer::POA_var thermostat_poa
    = ctrl_poa->find_POA("Thermostats", true);

// Activate POAs...
```

Identity Mapping Operations

The POA offers a number of operations to map among object references, object IDs, and servants:

```
interface POA {
    // ...
    ObjectId servant_to_id(in Servant s)
                raises(ServantNotActive, WrongPolicy);
    Object   servant_to_reference(in Servant s)
                raises(ServantNotActive, WrongPolicy);
    Servant  reference_to_servant(in Object o)
                raises(ObjectNotActive, WrongAdapter, WrongPolicy);
    ObjectId reference_to_id(in Object reference)
                raises(WrongAdapter, WrongPolicy);
    Servant  id_to_servant(in ObjectId oid)
                raises(ObjectNotActive, WrongPolicy);
    Object   id_to_reference(in ObjectId oid)
                raises(ObjectNotActive, WrongPolicy);
};
```

```
static PortableServer::POA_ptr
create_persistent_POA(
    const char *            name,
    PortableServer::POA_ptr parent,
    PortableServer::ServantManager_ptr locator
        = PortableServer::ServantLocator::_nil())
{
    // Create policy list for simple persistence
    CORBA::PolicyList pl;
    CORBA::ULong len = pl.length();
    pl.length(len + 1);
    pl[len++] = parent->create_lifespan_policy(
                        PortableServer::PERSISTENT
                    );
    pl.length(len + 1);
    pl[len++] = parent->create_id_assignment_policy(
                        PortableServer::USER_ID
                    );
    pl.length(len + 1);
    pl[len++] = parent->create_thread_policy(
                        PortableServer::SINGLE_THREAD_MODEL
                    );
    pl.length(len + 1);
    pl[len++] = parent->create_implicit_activation_policy(
                       PortableServer::NO_IMPLICIT_ACTIVATION
```

```
                        );

    // Check if we need to register a servant locator
    if (!CORBA::is_nil(locator)) {
        pl.length(len + 1);
        pl[len++] = parent->create_servant_retention_policy(
                                PortableServer::NON_RETAIN
                            );
        pl.length(len + 1);
        pl[len++] = parent->create_request_processing_policy(
                              PortableServer::USE_SERVANT_MANAGER
                            );
    }

    // Get parent POA's POA manager
    PortableServer::POAManager_var pmanager
        = parent->the_POAManager();

    // Create new POA
    PortableServer::POA_var poa =
        parent->create_POA(name, pmanager, pl);

    // Register servant locator, if required
    if (!CORBA::is_nil(locator))
        poa->set_servant_manager(locator);
```

```
    // Clean up
    for (CORBA::ULong i = 0; i < len; ++i)
        pl[i]->destroy();

    return poa._retn();
}
```

```
// Helper function to create object references.

CCS::Thermometer_ptr
make_dref(CCS::AssetType anum)
{
    // Convert asset number to OID
    PortableServer::ObjectId_var oid = make_oid(anum);

    // Look at the model via the network to determine
    // the repository ID and the POA.
    char buf[32];
    if (ICP_get(anum, "model", buf, sizeof(buf)) != 0)
        abort();
    const char * rep_id;
    PortableServer::POA_ptr poa;
    if (strcmp(buf, "Sens-A-Temp") == 0) {
        rep_id = "IDL:acme.com/CCS/Thermometer:1.0";
        poa = Thermometer_impl::poa();
        CORBA::Object_var obj
            = poa->create_reference_with_id(oid, rep_id);
        return CCS::Thermometer::_narrow(obj);
    } else {
        rep_id = "IDL:acme.com/CCS/Thermostat:1.0";
        poa = Thermostat_impl::poa();
        CORBA::Object_var obj
```

```
            = poa->create_reference_with_id(oid, rep_id);
        return CCS::Thermostat::_narrow(obj);
    }
}
```

```
PortableServer::Servant
DeviceLocator::
preinvoke(
    const PortableServer::ObjectId &         oid,
    PortableServer::POA_ptr                  adapter,
    const char *                             operation,
    PortableServer::ServantLocator::Cookie & the_cookie
) throw(CORBA::SystemException, PortableServer::ForwardRequest)
{
    CCS::AssetType anum = make_anum(oid);
    if (!Thermometer_impl::m_ctrl->exists(anum))
        throw CORBA::OBJECT_NOT_EXIST();
    CORBA::String_var poa_name = adapter->the_name();
    if (strcmp(poa_name, "Thermometers") == 0)
        return new Thermometer_impl(anum);
    else
        return new Thermostat_impl(anum);
}
```

```
void
DeviceLocator::
postinvoke(
    const PortableServer::ObjectId &        oid,
    PortableServer::POA_ptr                 adapter,
    const char *                            operation,
    PortableServer::ServantLocator::Cookie  the_cookie,
    PortableServer::Servant                 the_servant
) throw(CORBA::SystemException)
{
    the_servant->_remove_ref();
}
```

Introduction

Some aspects of behavior for ORBacus are controlled by properties.

Properties are scoped name–value pairs. The name is a variable such as **`ooc.orb.client_timeout`**. The value of a property is a string.

ORBacus uses properties to change its behavior in some way.

There are properties to control threading models, to control the return value from **`resolve_initial_references`** for different tokens, to change connection management strategies, etc.

The property configuration mechanism is not standardized and therefore specific to ORBacus.

Property values are read once only, on process start-up. Changing the value of a property has no effect on running processes!

Defining Properties

Properties can be defined in a number of places:

1. in a Windows registry key under
   `HKEY_LOCAL_MACHINE\Software\OOC\Properties\`*`<name>`*
2. in a Windows registry key under
   `HKEY_CURRENT_USER\Software\OOC\Properties\`*`<name>`*
3. in a configuration file specified by the **`ORBACUS_CONFIG`** environment variable
4. by setting a property from within the program
5. in a configuration file specified on the command line
6. by using a command-line option

Property values are retrieved using all these means (if applicable).

Higher numbers have higher precedence.

Setting Properties in the Registry

To set a property in the Windows registry, use the property name, replacing the "**`.`**" by "**`\`**". For example, the property **`ooc.orb.id`** can be set by setting the value of:

```
HKEY_LOCAL_MACHINE\Software\OOC\Properties\ooc\orb\id
```

Defaults for properties that affect all processes on the system can be set under **`HKEY_LOCAL_MACHINE`**.

Defaults for properties that affect only the current user can be set under **`HKEY_CURRENT_USER`**.

Setting Properties in a Configuration File

You can create a configuration file containing property values. For example:

```
# Default client concurrency model is threaded
ooc.orb.conc_model=threaded

# Default server concurrency model is a pool of 20 threads
ooc.orb.oa.conc_model=thread_pool
ooc.orb.oa.thread_pool=20

# Default naming service is on HostA, port 5000
ooc.orb.service.NameService=corbaloc::HostA:5000/NameService
```

Trailing white space is ignored and is not part of the property.

```
ORBACUS_CONFIG=/home/michi/.ob.config
export ORBACUS_CONFIG
```

Setting Properties Programmatically

You can set properties from within your program using the **`OB::Properties`** class and **`OBCORBA::ORB_init`**.

```
// Get default properties (established by config file)
OB::Properties_var dflt
    = OB::Properties::getDefaultProperties();

// Initialize a property set with the defaults
OB::Properties_var props = new OB::Properties(dflt);

// Set the properties we want
props->setProperty("ooc.orb.conc_model", "threaded");
props->setProperty("ooc.orb.oa.conc_model", "thread_pool");
props->setProperty("ooc.orb.oa.thread_pool", "20");

// Initialize the ORB with the given properties
CORBA::ORB_var orb = OBCORBA::ORB_init(argc, argv, props);
```

Setting Properties from the Command Line

You can pass the **`-ORBconfig <pathname>`** option to a process.

The specified file overrides the defaults that are taken from registry keys or the **`ORBACUS_CONFIG`** environment variable, and overrides values that are set with **`OBCORBA::ORB_init`**.

You can also set most properties from the command line directly. For example:

```
./a.out -ORBconfig $HOME/ob.config \
-ORBthreaded -OAreactive
```

Explicit property values override the defaults in a configuration file, so this process will use **`ooc.orb.conc_model=threaded`** and **`ooc.orb.oa.conc_model=reactive`**.

See the manual for a complete list of options.

Commonly Used Properties

- **ooc.orb.service.*<name>*=*<IOR>***

  Specify the IOR to returned by **resolve_initial_references**.

- **ooc.orb.trace.connections=*<level>***.

  Trace connection establishment and closure at ***<level>*** (0–2).

- **ooc.orb.trace.retry=*<level>***.

  Trace connection reestablishment attempts at ***<level>*** (0–2).

- **ooc.orb.oa.numeric=*{*true*,* false*}***

  Use a dotted-decimal IP address in IORs instead of a name.

- **ooc.orb.oa.port=*<port>***

  Set the port number to be embedded in IORs.

Introduction

Copying stringified references from a server to all its clients is clumsy and does not scale.

The Naming Service provides a way for servers to advertise references under a name, and for clients to retrieve them. The advantages are:

- Clients and servers can use meaningful names instead of having to deal with stringified references.
- By changing a reference in the service without changing its name, you can transparently direct clients to a different object.
- The Naming Service solves the bootstrapping problem because it provides a fixed point for clients and servers to rendezvous.

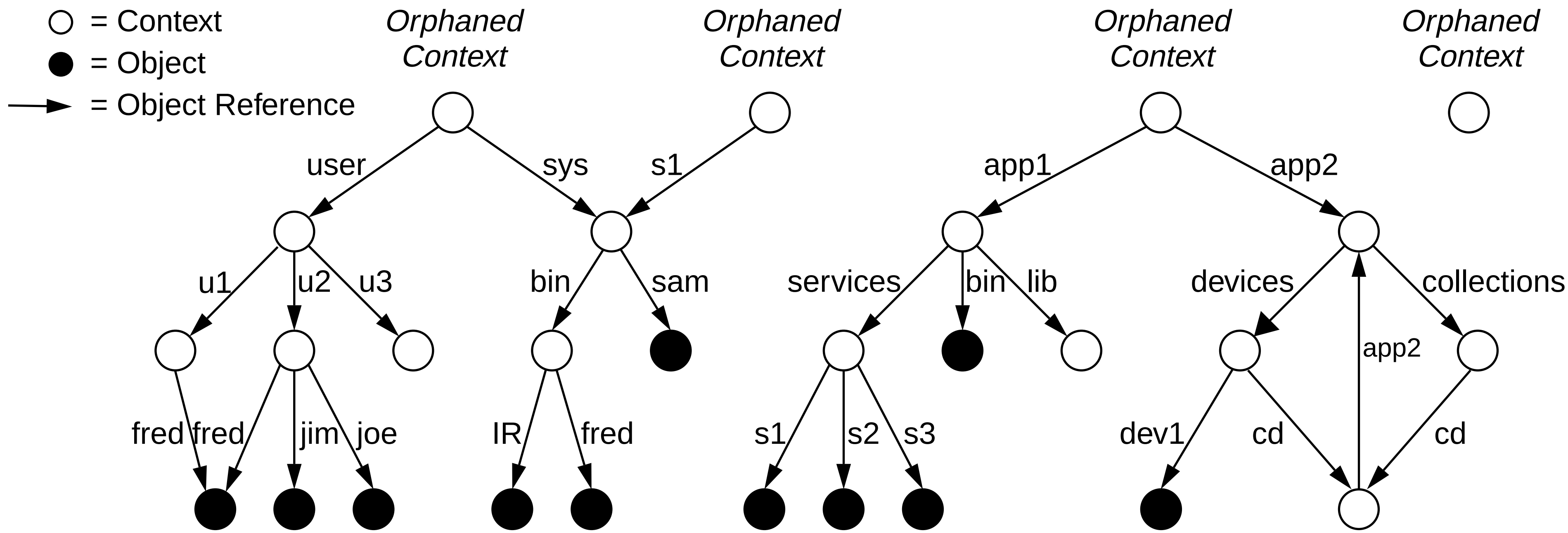
The Naming Service is much like a white pages phone book. Given a name, it returns an object reference.

Terminology

- A name-to-IOR association is called a *name binding*.
- Each binding identifies exactly one object reference, but an object reference may be bound more than once (have more than one name).
- A *naming context* is an object that contains name bindings. The names within a context must be unique.
- Naming contexts can contain bindings to other naming contexts, so naming contexts can form graphs.
- *Binding* a name to a context means to add a name–IOR pair to a context.
- *Resolving* a name means to look for a name in a context and to obtain the IOR bound under that name.

Example Naming Graph

A naming service provides a graph of contexts that contain bindings to other contexts or objects.



The graph is similar to (but not the same as) a file system hierarchy with directories and files.

Naming IDL Structure

The IDL for the Naming Service has the following overall structure:

```
//File: CosNaming.idl
#pragma prefix "omg.org"
module CosNaming {
    // Type definitions here...
    interface NamingContext {
        // ...
    };
    interface NamingContextExt : NamingContext {
        // ...
    };
    interface BindingIterator {
        // ...
    };
};
```

Note that all OMG-defined IDL uses the prefix **omg.org**.

Name Representation

A name component is a *pair* of strings. A sequence of name components forms a pathname through a naming graph:

```
module CosNaming {
    typedef string Istring; // Historical hangover
    struct NameComponent {
        Istring id;
        Istring kind;
    };
    typedef sequence<NameComponent> Name;
    // ...
};
```

The **`kind`** field is meant to be used similarly to a file name extension (such as “filename.cc”).

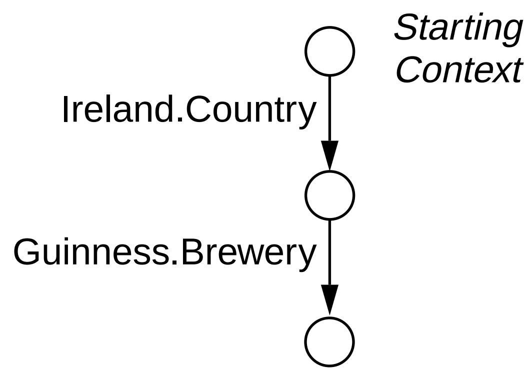
For two name components to be considered equal, both **`id`** and **`kind`** must be equal.

Stringified Names

Names are sequences of string pairs. We can show a name as a table:

Index	id	kind
0	`Ireland`	`Country`
1	`Guinness`	`Brewery`

This is a two-component name corresponding to the following graph:



The same name can be written as a string as:

```
Ireland.Country/Guinness.Brewery
```

```
Guinness.Beer
Budweiser.Beer
Chair.Person
Chair.Furniture
```

Pathnames and Name Resolution

There is no such thing as an absolute pathname in a Naming Service.

All names must be interpreted relative to a starting context (because a Naming Service does not have a distinguished root context).

Name resolution works by successively resolving each name component, beginning with a starting context.

A name with components $\mathbf{c_1,\ c_2,\ \ldots,\ c_n}$: is resolved as:

$$\mathbf{cxt} \rightarrow \mathbf{op}([\mathbf{c_1}, \mathbf{c_2}, \ldots, \mathbf{c_n}]) \equiv \mathbf{cxt} \rightarrow \mathbf{resolve}([\mathbf{c_1}]) \rightarrow \mathbf{op}([\mathbf{c_2}, \ldots, \mathbf{c_n}])$$

This looks complex, but simply means that operation **op** is applied to the final component of a name after all the preceding components have been used to locate the final component.

Obtaining an Initial Naming Context

You must obtain an initial naming context before you can do anything with the service.

The configured initial naming context is returned by

```
resolve_initial_references("NameService")
```

This returns an object reference to either a **NamingContext** or a **NamingContextExt** object. (For ORBacus, you always get a **NamingContextExt** interface.)

Exactly which context is returned depends on the ORB configuration.

You can override the default with the **-ORBInitRef** option:

```
./a.out -ORBInitRef NameService=<ior>
```

```
// Initialize the ORB.
CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

// Get initial naming context.
CORBA::Object_var obj
    = orb->resolve_initial_references("NameService");

// Narrow to NamingContext
CosNaming::NamingContext_var inc;   // Initial naming context
inc = CosNaming::NamingContext::_narrow(obj);

// ...
```

```
// Initialize the ORB.
CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

// Get initial naming context.
CORBA::Object_var obj
    = orb->resolve_initial_references("NameService");

// Narrow to NamingContextExt
CosNaming::NamingContextExt_var inc;    // Initial naming context
inc = CosNaming::NamingContextExt::_narrow(obj);

if (!CORBA::is_nil(inc)) {
    // It's an Interoperable Naming Service...
} else {
    // Doesn't support INS, must be the old service then...
}

// ...
```

```
$ ./myclient -ORBInitRef NameService=IOR:013a0d0...
```

```
$ ./myclient -ORBInitRef MyFavouriteService=IOR:013a0d0...
```

Naming Service Exceptions

The **NamingContext** interface defines a number of exceptions:

```
interface NamingContext {
    enum    NotFoundReason { missing_node, not_context, not_object };
    exception NotFound {
        NotFoundReason  why;
        Name            rest_of_name;
    };
    exception CannotProceed {
        NamingContext   cxt;
        Name            rest_of_name;
    };
    exception InvalidName {};
    exception AlreadyBound {};
    exception NotEmpty {};
    // ...
};
```

Creating and Destroying Contexts

`NamingContext` contains three operations to control the life cycle of contexts:

```
interface NamingContext {
    // ...
    NamingContext   new_context();
    NamingContext   bind_new_context(in Name n) raises(
                        NotFound, CannotProceed,
                        InvalidName, AlreadyBound
                    );
    void            destroy() raises(NotEmpty);
    // ...
};
```

Creating Bindings

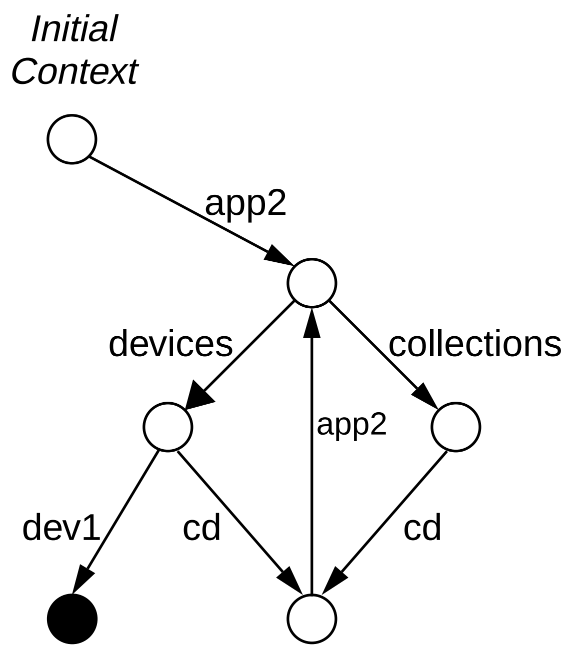
Two operations create bindings to application objects and to contexts:

```
interface NamingContext {
    // ...
    void    bind(in Name n, in Object obj)
                raises(
                    NotFound, CannotProceed,
                    InvalidName, AlreadyBound
                );
    void    bind_context(in Name n, in NamingContext nc)
                raises(
                    NotFound, CannotProceed,
                    InvalidName, AlreadyBound
                );
    // ...
};
```

Context Creation Example

To create a naming graph, you can use names that are all relative to the initial context or you can use names that are relative to each newly-created context.

The code examples that follow create the following graph:



```
CosNaming::NamingContext_var inc = ...;    // Get initial context

CosNaming::Name name;
name.length(1);
name[0].id = CORBA::string_dup("app2");    // kind is empty

CosNaming::NamingContext_var app2;
app2 = inc->bind_new_context(name);        // inc -> app2

name.length(2);
name[1].id = CORBA::string_dup("collections");
CosNaming::NamingContext_var collections;
collections = inc->bind_new_context(name); // app2 -> collections

name[1].id = CORBA::string_dup("devices");
CosNaming::NamingContext_var devices;
devices = inc->bind_new_context(name);     // app2 -> devices

name.length(3);
name[2].id = CORBA::string_dup("cd");
CosNaming::NamingContext_var cd;
cd = inc->bind_new_context(name);          // devices -> cd

name.length(4);
name[3].id = CORBA::string_dup("app2");
```

```
inc->bind_context(name, app2);                  // cd -> app2

CCS::Controller_var ctrl = ...;
name.length(3);
name[2].id = CORBA::string_dup("dev1");
inc->bind(name, ctrl);                          // devices -> dev1

name[1].id = CORBA::string_dup("collections");
name[2].id = CORBA::string_dup("cd");
inc->bind_context(name, cd);                    // collections -> cd
```

```
CosNaming::NamingContext_var inc = ...; // Get initial context

CosNaming::Name name;                   // Initialize name
name.length(1);
name[0].id = CORBA::string_dup("app2"); // kind is empty string

CosNaming::NamingContext_var app2;
app2 = inc->bind_new_context(name);     // Create and bind

name[0].id = CORBA::string_dup("devices");
CosNaming::NamingContext_var devices;
devices = app2->bind_new_context(name); // Create and bind

name[0].id = CORBA::string_dup("collections");
CosNaming::NamingContext_var collections;
collections = app2->bind_new_context(name); // Create and bind

name[0].id = CORBA::string_dup("cd");   // Make cd context
CosNaming::NamingContext_var cd;
cd = devices->bind_new_context(name);   // devices -> cd

collections->bind_context(name, cd);    // collections -> cd

name[0].id = CORBA::string_dup("app2");
cd->bind_context(name, app2);           // cd -> app2
```

```
CCS::Controller_var ctrl = ...;          // Get controller ref

name[0].id = CORBA::string_dup("dev1");
devices->bind(name, ctrl);               // Add controller
```

Rebinding

The **rebind** and **rebind_context** operations replace an existing binding:

```
interface NamingContext {
    // ...
    void    rebind(in Name n, in Object obj)
                raises(
                    NotFound, CannotProceed, InvalidName
                );
    void    rebind_context(in Name n, in NamingContext nc)
                raises(
                    NotFound, CannotProceed, InvalidName
                );
    // ...
};
```

Use **rebind_context** with caution because it may orphan contexts!

```
CORBA::Object_var obj = ...;              // Get an object
CosNaming::NamingContext_var cxt = ...; // Get a context...

CosNaming::Name name;
name.length(1);
name[0].id = CORBA::string_dup("Some name");

cxt->rebind(name, obj);                   // Fine
cxt->rebind(name, obj);                   // Fine
```

Resolving Bindings

The **`resolve`** operation returns the reference stored in a binding:

```
interface NamingContext {
    // ...
    Object  resolve(in Name n) raises(
                NotFound, CannotProceed, InvalidName
            );
    // ...
};
```

The returned reference is (necessarily) of type **`Object`**, so you must narrow it to the correct type before you can invoke operations on the reference.

```
CosNaming::NamingContext inc = ...; // Get initial context...

CosNaming::Name name;
name.length(3);
name[0].id = CORBA::string_dup("app2");
name[1].id = CORBA::string_dup("devices");
name[2].id = CORBA::string_dup("dev1");

CORBA::Object_var obj;
try {
    obj = inc->resolve(name);
} catch (const CosNaming::NamingContext::NotFound &) {
    // No such name, handle error...
    abort();
} catch (const CORBA::Exception & e) {
    // Something else went wrong...
    cerr << e << endl;
    abort();
}

if (CORBA::is_nil(obj)) {
    // Polite applications don't advertise nil references!
    cerr << "Nil reference for controller! << endl;
    abort();
}
```

```
CCS::Controller_var ctrl;
try {
    ctrl = CCS::Controller::_narrow(obj);
} catch (CORBA::SystemException & e) {
    // Can't figure it out right now...
    cerr << "Can't narrow reference: " << e << endl;
    abort();
}

if (CORBA::is_nil(ctrl)) {
    // Oops!
    cerr << "Someone advertised wrong type of object!" << endl;
    abort();
}

// Use ctrl reference...
```

Removing Bindings

You can remove a binding by calling **unbind**:

```
interface NamingContext {
    // ...
    void    unbind(in Name n) raises(
                NotFound, CannotProceed, InvalidName
            );
    // ...
};
```

unbind removes a binding whether it denotes a context or an application object.

Calling **unbind** on a context *will* create an orphaned context. To get rid of a context, you must both **destroy** *and* **unbind** it!

```
CosNaming::NamingContext_var inc = ...; // Get initial context

CosNaming::Name name;
name.length(3);
name[0].id = CORBA::string_dup("app2");
name[1].id = CORBA::string_dup("devices");
name[2].id = CORBA::string_dup("dev1");

inc->unbind(name);
```

```
// ...
// Name is currently initialized "app2/devices/dev1".
// Change name to "app2/devices/cd/app2".
name.length(4);
name[2].id = CORBA::string_dup("cd");
name[3].id = CORBA::string_dup("app2");
inc->unbind(name);  // Get rid of app2 link

name.length(3);
CosNaming::NamingContext_var tmp = inc->resolve(name);
tmp->destroy();     // Destroy cd context
name.length(2);
inc->unbind(name);  // Remove binding in parent context
```

Listing Name Bindings

```
// In module CosNaming:
enum BindingType { nobject, ncontext };

struct Binding {
    Name        binding_name;
    BindingType binding_type;
};
typedef sequence<Binding> BindingList;

interface BindingIterator;  // Forward declaration
interface NamingContext {
    // ...
    void    list(
        in unsigned long    how_many,
        out BindingList     bl,
        out BindingIterator it
    );
};
```

```
interface BindingIterator {
    boolean next_n(
                in unsigned long    how_many,
                out BindingList     bl
    );
    boolean next_one(out Binding b);
    void    destroy();
};
```

```
void
show_chunk(const CosNaming::BindingList & bl) // Helper function
{
    for (CORBA::ULong i = 0; i < bl.length(); ++i) {
        cout << bl[i].binding_name[0].id;
        if (   bl[i].binding_name[0].id[0] == '\0'
            || bl[i].binding_name[0].kind[0] != '\0') {
            cout << "." << bl[i].binding_name[0].kind;
        }
        if (bl[i].binding_type == CosNaming::ncontext)
            cout << ": context" << endl;
        else
            cout << ": reference" << endl;
    }
}

void
list_context(CosNaming::NamingContext_ptr nc)
{
    CosNaming::BindingIterator_var it;      // Iterator reference
    CosNaming::BindingList_var bl;          // Binding list
    const CORBA::ULong CHUNK = 100;         // Chunk size

    nc->list(CHUNK, bl, it);                // Get first chunk
    show_chunk(bl);                         // Print first chunk
```

```
    if (!CORBA::is_nil(it)) {                       // More bindings?
        while (it->next_n(CHUNK, bl))               // Get next chunk
            show_chunk(bl);                         // Print chunk
        it->destroy();                              // Clean up
    }
}
```

Pitfalls in the Naming Service

Here are a handful of rules you should adhere to when using the Naming Service:

- Do not advertise nil references.
- Do not advertise transient references.
- Stay clear of unusual characters for names, such as “.”, “/”, “*”, etc.
- Take care to destroy contexts correctly.
- Call destroy on iterators.
- Make the graph a single-rooted tree.

Stringified Name Syntax

- A stringified name uses "**/**" and "**.**" to separate name components and **id** and **kind** fields:

  a.b/c.d (id[0] = "**a**", kind[0] = "**b**", id[1] = "**c**", kind[1] = "**d**")

- A backslash ("****") escapes the meaning of these characters:

  a\.b\/c\\d.e (id = "**a.b/c\d**", kind = "**e**"

- A name without a trailing "**.**" denotes an empty **kind** field:

  hello (id = "**hello**", kind = "")

- A name with a leading "**.**" indicates an empty **id** field:

  .world (id = "", kind = "**world**")

- A single "**.**" denotes a name with empty id and kind fields:

  . (id = "", kind = "")

Using Stringified Names

The **NamingContextExt** interface provides convenience operations for using stringified names:

```
interface NamingContextExt : NamingContext {
    typedef string StringName;

    StringName  to_string(in Name n) raises(InvalidName);
    Name        to_name(in StringName sn) raises(InvalidName);

    Object      resolve_str(in StringName sn) raises(
                    NotFound, CannotProceed, InvalidName
                );
    // ...
};
```

to_string and **to_name** are like C++ static helper functions. They do the same thing no matter on what context you invoke them.

URL-Style IORs

The specification defines two alternative styles of object references:

- **`corbaloc`**

  An IOR that denotes an object at a specific location with a specific object key, for example:

  ```
  corbaloc::bobo.acme.com/obj17359
  ```

- **`corbaname`**

  An IOR that denotes a reference that is advertised in a naming service, for example:

  ```
  corbaname::bobo.acme.com/NameService#CCS/controller
  ```

URL-style IORs are useful for bootstrapping and configuration.

Do *not* use them as a general replacement for normal IORs!

URL-Style IORs (cont.)

A **`corbaloc`** IOR encodes a host, a port, and an object key:

```
corbaloc::myhost.myorg.com:3728/some_object_key
```

The port number is optional and defaults to 2809:

```
corbaloc::myhost.myorg.com/some_object_key
```

Dotted-decimal addresses are legal:

```
corbaloc::123.123.123.123/some_object_key
```

You can specify a protocol and version. (The default is **`iiop`** and **`1.0`**):

```
corbaloc:iiop:1.1@myhost.myorg.com:3728/some_object_key
```

Multiple addresses are legal:

```
corbaloc::hostA:372,:hostB,:hostC:3728/some_object_key
```

URL-Style IORs (cont.)

A **`corbaname`** IOR is like a **`corbaloc`** IOR with an appended stringified name:

```
corbaname::myhost:5000/NameService#controller
```

This URL denotes a naming context on **`myhost`** at port 5000 with object key **`NameService`**. That context, under the stringified name **`controller`**, contains the object denoted by the URL.

Complex names are possible:

```
corbaname::myhost:5000/ns#Ireland.Country/Guinness.Brewery
```

In this example, the naming context with key **`ns`** must contain a context named **`Ireland.Country`** containing a binding **`Guinness.Brewery`** that denotes the target object.

URL Escape Sequences

ASCII alphabetic and numeric characters can appear in URL-style IOR without escaping them. The following characters can also appear without escapes:

“;”, “/”, “:”, “?”, “@”, “&”, “=”, “+”, “$”,
“,”, “-”, “_”, “.”, “!”, “~”, “*”, “‘”, “(“, “)”

All other characters must be represented in escaped form. For example:

Stringified Name	Escaped Form
<a>.b/c.d	%3ca%3e.b/c.d
a.b/ c.d	a.b/%20%20c.d
a%b/c%d	a%25b/c%25d
a\\b/c.d	a%5c%5cb/c.d

A **%** is always followed by two hex digits that encode the byte value (in ISO Latin-1) of the corresponding character.

Resolving URL-Style IORs

You can pass a URL-style IOR directly to **`string_to_object`**:

```
CORBA::Object obj
    = orb->string_to_object("corbaname::localhost/nc#myname");
```

The ORB resolves the reference like any other stringified IOR, including the required **`resolve`** invocation on the target naming context.

This is useful particularly for configuration:

```
./myclient -ORBInitRef \
NameService=corbaloc::localhost/NameService
```

Creating URL-Style IORs

Apart from simply writing them down, you can create a URL-style IOR using a **NamingContextExt** object:

```
interface NamingContextExt : NamingContext {
    // ...
    typedef string StringName;
    typedef string Address;
    typedef string URLString;

    exception InvalidAddress;

    URLString to_url(in Address addrkey, in StringName sn)
                raises(InvalidAddress, InvalidName);
};
```

addrkey must be an address, optional port, and object key in **corbaloc** syntax.

```
CosNaming::NamingContextExt_var nc = ...;
CORBA::String_var url =
    nc->to_url(":localhost:5789/abc", "CCS/controller");
cout << url << endl;
```

```
corbaname::localhost:5789/abc#CCS/controller
```

```
url = nc->to_url(":bobo.ooc.com.au/nc", "a\\b%/c.d");
```

```
corbaname::bobo.ooc.com.au/nc#a%5c%5cb%25/c.d
```

What and When to Advertise

You should advertise key objects in the Naming Service, such as the CCS controller. (Such objects are public integration points.)

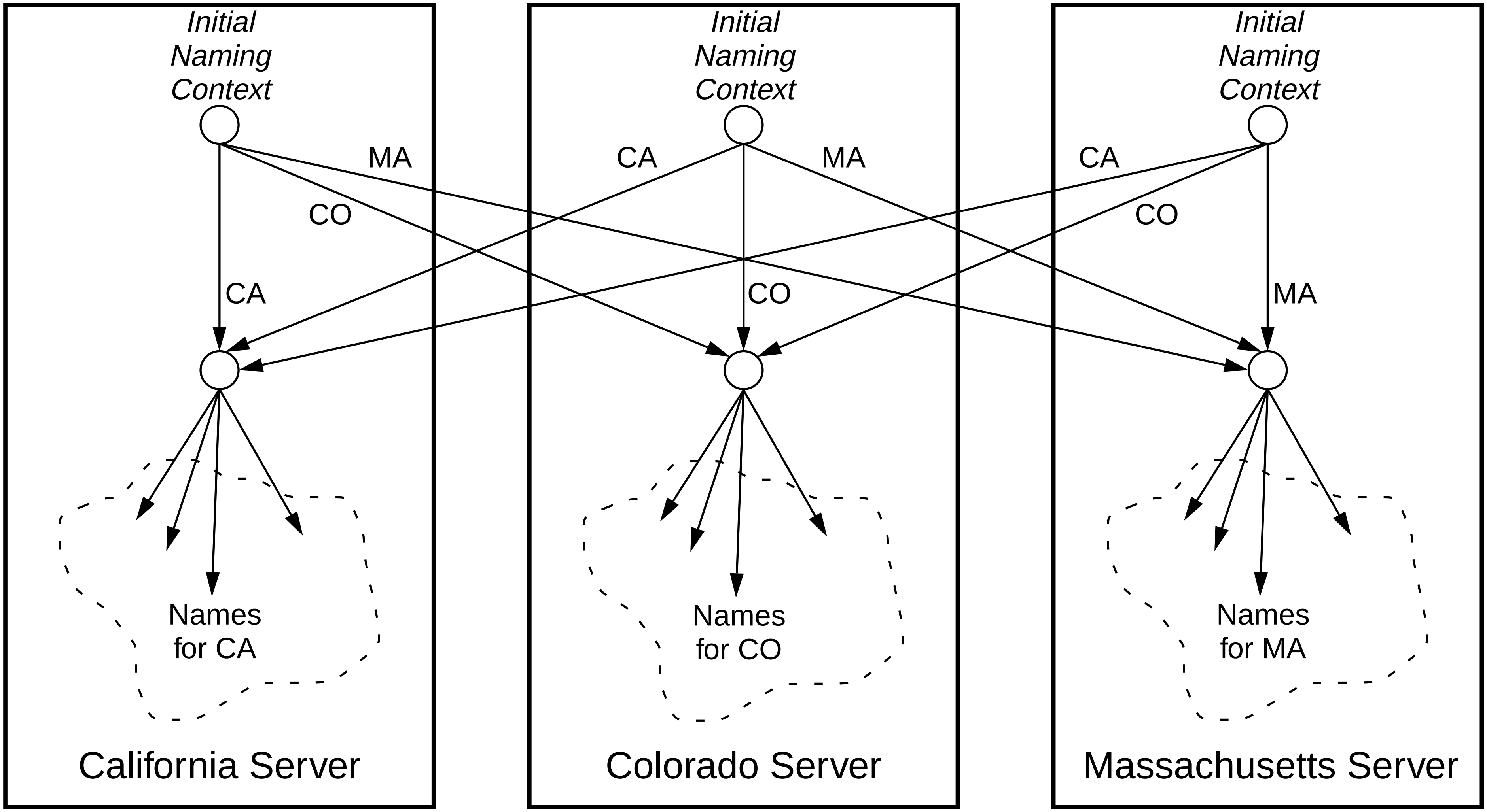
Bootstrap objects are normally added to the service at installation time.

Provide a way to recreate key bindings with a tool.

You can advertise all persistent objects you create. If you do, tie the updates to the Naming Service to the life cycle operations for your objects.

Decide on a strategy of what to do when the Naming Service is unavailable (deny service or live with the inconsistency).

Federated Naming



Running the Naming Service

ORBacus Names is provided as the **nameserv** executable. Common options (use **nameserv -h** for a list):

- **-i**

  Print initial naming context IOR on **stdout**

- **-d database_file**

  Specifies database file for the service. (Without **-d**, the service is not persistent and uses an in-memory database.)

- **-s**

  (Re)initializes the database. Must be used with **-d** option.

The object key of the initial naming context is **NameService**.

The nsadmin Tool

The **nsadmin** tool provides a way to manipulate the Naming Service from the command line. Common options (use **nsadmin -h** for a list):

- **-b name IOR**

  Bind **IOR** under the name **name**.

- **-c name**

  Create and bind a new context under the name **name**.

- **-l [name]**

  List the contents of the context **name**. (Initial context, by default.)

- **-r name**

  Print the IOR for the binding identified by **name**.

IORs can be in normal or URL-style syntax.

```
nsadmin -c CCS
nsadmin -b CCS/controller `cat ctrl.ref`
```

```
nsadmin -r CCS/controller
```

Compiling and Linking

The IDL for ORBacus Names is installed in the ORBacus directory as `idl/OB/CosNaming.idl`.

The header files for the service are in `include/OB/CosNaming.h` and `include/OB/CosNaming_skel.h`.

The stubs and skeletons for ORBacus Names are pre-compiled and installed in `lib/libCosNaming.sl`.

To compile a client or server that uses ORBacus Names, compile with **`-I /opt/OB4/include`** and link with **`-L /opt/OB4/lib -lCosNaming`**.

```
$ cat ob.config
ooc.orb.service.NameService=corbaloc::janus.ooc.com.au:5000/NameS
ervice
$ ORBACUS_CONFIG=`pwd`/ob.config; export ORBACUS_CONFIG
```

```
$ /opt/OB4/bin/nameserv -OAport 5000
```

```
$ nsadmin -c student_1
$ nsadmin -l
Found 1 binding:
student_1 [context]
```

```
// Get Naming Service reference
obj = orb->resolve_initial_references("NameService");
CosNaming::NamingContext_var inc
    = CosNaming::NamingContext::_narrow(obj);
if (CORBA::is_nil(inc))
    throw "Cannot find initial naming context!";

// Advertise the controller reference in the naming service.
CosNaming::Name name;
name.length(2);
name[0].id = CORBA::string_dup("student_1");
name[1].id = CORBA::string_dup("controller");
obj = ctrl_servant->_this();
inc->rebind(name, obj);
```

```
// Get controller reference from Naming Service
CORBA::Object_var obj
    = orb->resolve_initial_references("NameService");
CosNaming::NamingContext_var inc
    = CosNaming::NamingContext::_narrow(obj);
CosNaming::Name name;
name.length(2);
name[0].id = CORBA::string_dup("student_1");
name[1].id = CORBA::string_dup("controller");
obj = inc->resolve(name);

// Try to narrow to CCS::Controller.
CCS::Controller_var ctrl;
try {
    ctrl = CCS::Controller::_narrow(obj);
} catch (const CORBA::SystemException &se) {
    cerr << "Cannot narrow controller reference: " << se << endl;
    throw 0;
}
if (CORBA::is_nil(ctrl)) {
    cerr << "Wrong type for controller ref." << endl;
    throw 0;
}
```

Purpose of an Implementation Repository

An implementation repository (IMR) has three functions:

- It maintains a registry of known servers.
- It records which server is currently running on what machine, together with the port numbers it uses for each POA.
- It starts servers on demand if they are registered for automatic activation.

The main advantage of an IMR is that servers that create persistent references

- need not run on a fixed machine and a fixed port number
- need not be running permanently

Binding

There are two methods of binding object references:

- Direct Binding (for persistent and transient references)

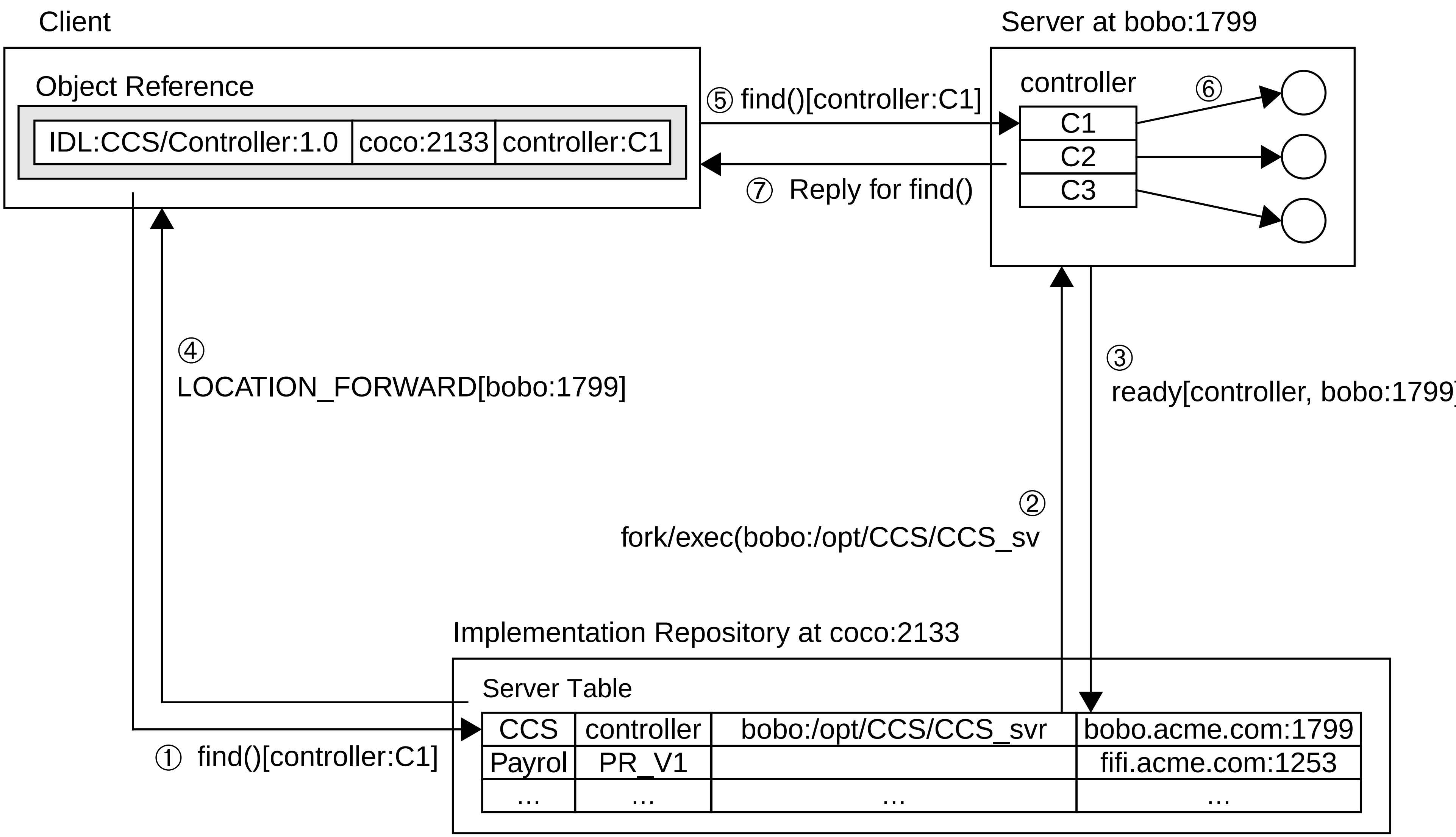
  References carry the host name and port number of the server. This works, but you cannot move the server around without breaking existing references.

- Indirect Binding (for persistent references)

  References carry the host name and port number of an Implementation Repository (IMR). Clients connect to the IMR first and then get a reference to the actual object in the server. This allows servers to move around without breaking existing references.

IMRs are proprietary for servers (but interoperate with all clients).

Indirect Binding



Automatic Server Start-Up

The IMR can optionally start server processes.

Two modes of operation are supported by the IMR:

- shared

  All requests from all clients are directed to the same server. The server is started on demand.

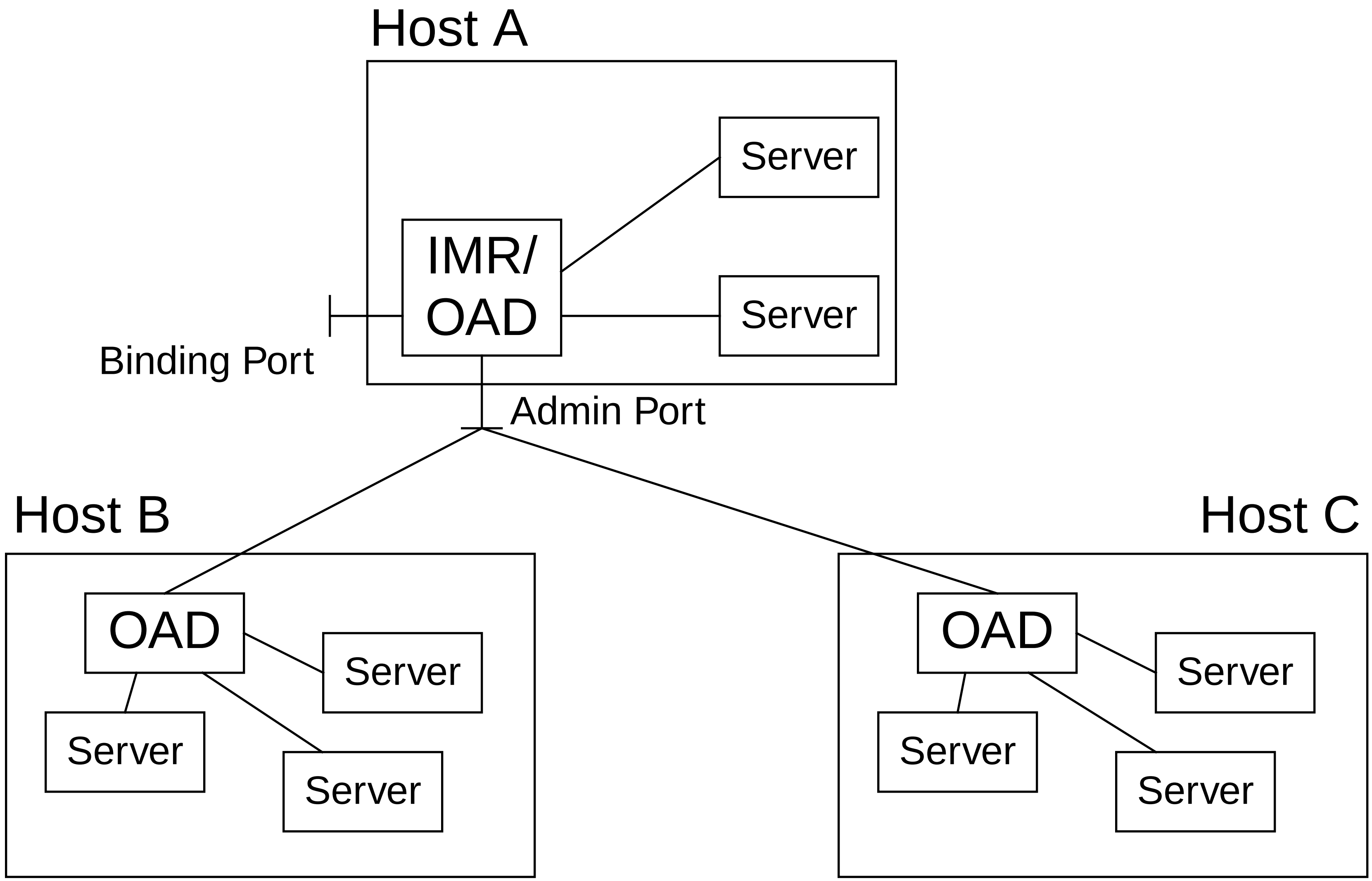
- persistent

  Same as the shared mode, but the server is started whenever the IMR starts and kept running permanently.

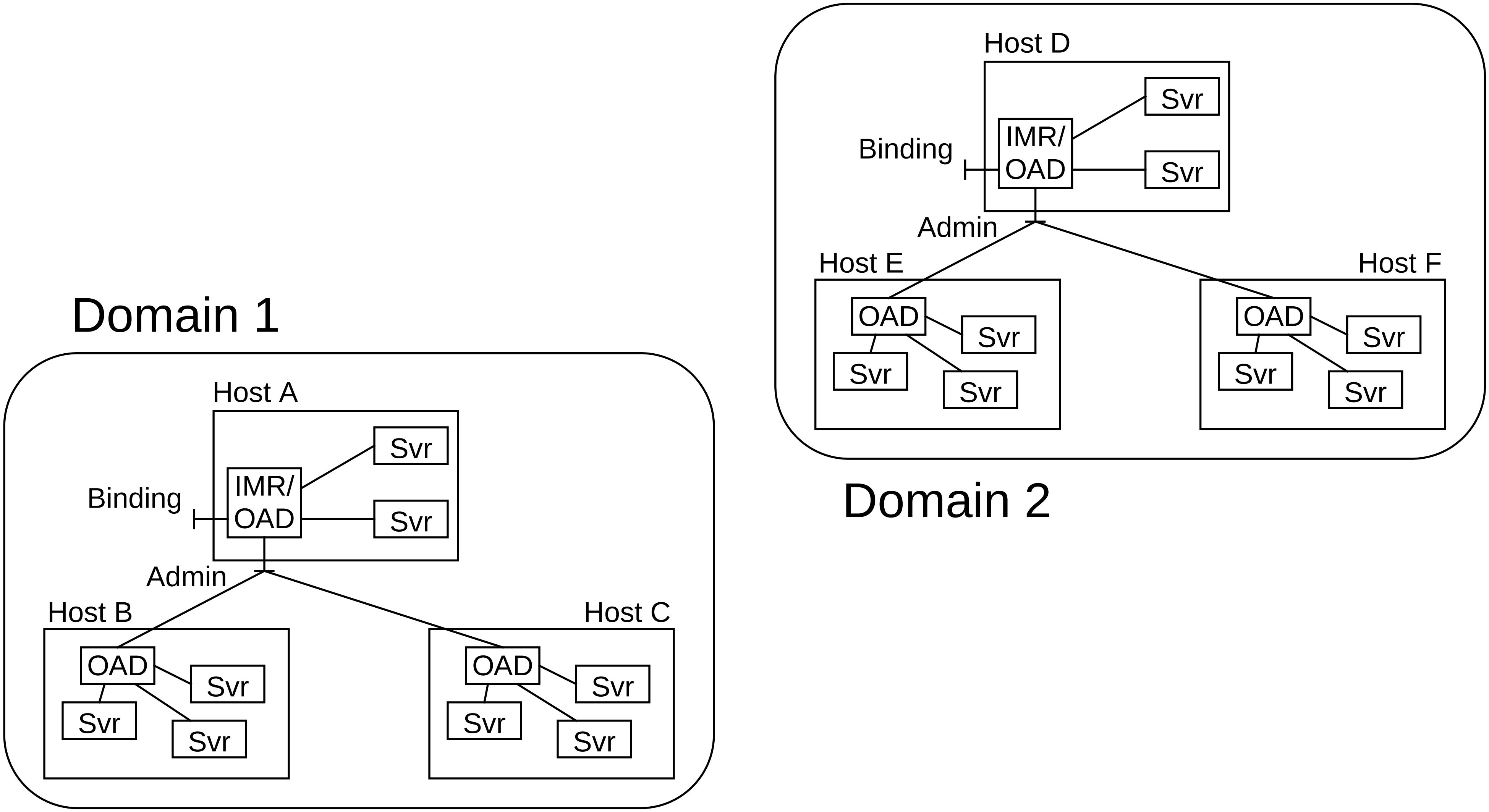
Servers are started by an Object Activation Daemon (OAD).

A single repository can have multiple OADs. An OAD must be running on each machine on which you want to start servers.

IMR Process Structure



Location Domains



The `imradmin` Tool

`imradmin` allows you to register servers with the IMR. General syntax:

```
imradmin <command> [<arg>...]
```

You must register each server under a server name with the IMR. The server name must be unique for that IMR:

```
imradmin --add-server CCS_server /bin/CCSserver
```

When the IMR starts the server, it automatically passes the server name in the **`-ORBserver_name`** option. (So the server knows that it should contact the IMR.)

If you want to manually start a server that is registered with the IMR, you must add the **`-ORBserver_name`** option when you start the server:

```
/bin/CCSserver -ORBserver_name CCS_server
```

Servers automatically register their persistent POAs with the IMR.

Server Execution Environment

An NT server started by the IMR becomes a detached process.

A UNIX server started by the IMR has the execution environment of a daemon:

- File descriptors **`0`**, **`1`**, and **`2`** are connected to **`/dev/null`**
- One additional file descriptor is open to the OAD.
- The **`umask`** is set to **`027`**.
- The working directory is `/`.
- The server has no control terminal.
- The server is a session and group leader.
- The user and group ID are those of the OAD.
- Signals have the default behavior.

```
#!/bin/sh
exec "$@"
```

```
/usr/local/bin/launch /bin/CCSserver
```

```
#!/bin/sh
umask 077
PATH=/bin:/usr/bin:/usr/local/bin; export PATH
HOME=/tmp; export HOME
cd $HOME
exec 1>>/$HOME/CCSserver.stdout
exec 2>>/$HOME/CCSserver.stderr
exec "$@"
```

Server Attributes

`imradmin` permits you to set attributes for a registered server with the **`--set-server`** command:

`imradmin --set-server` ***`<server-name> <mode>`***

Valid modes are:

- **`exec`**

  Changes the executable path for the server.

- **`args`**

  Changes the arguments passed to the server.

- **`mode`**

  Changes the mode. The mode must be **`shared`** or **`persistent`**.

```
imradmin --set-server CCS_server exec /usr/local/bin/CCSserver
```

```
imradmin --set-server CCS_server args -dbfile /tmp/CCS_DB
```

```
/usr/local/bin/CCSserver -dbfile /tmp/CCS_DB
```

```
imradmin --set-server CCS_server mode persistent
```

Server Attributes (cont.)

- **`activate_poas`**

  If **`true`**, persistent POAs are registered automatically. If **`false`**, each persistent POA must be registered explicitly.

- **`update_timeout`** (msec)

  The amount of time the IMR waits for updates to propagate.

- **`failure_timeout`** (sec)

  How long to wait for the server to start up and report as ready.

- **`max_spawns`**

  The number of times to try and restart the server before giving up.
  `imradmin --reset-server` ***`<server-name>`***
  resets the failure count.

```
imradmin --set-server CCS_server activate_poas false
imradmin --add-poa CCS_server Controller
imradmin --add-poa CCS_server Controller/Thermometers
```

```
imradmin --reset-server CCS_server
```

Getting IMR Status

A number of **`imradmin`** commands show you the status of the IMR and its OADs:

- **`imradmin --get-server-info`** ***<server-name>***
- **`imradmin --get-oad-status [`*<host>*`]`**
- **`imradmin --get-poa-status`** ***<server-name> <poa-name>***
- **`imradmin --list-oads`**
- **`imradmin --list-servers`**
- **`imradmin --list-poas`** ***<server-name>***
- **`imradmin --tree`**
- **`imradmin --tree-oad [`*<host>*`]`**
- **`imradmin --tree-server`** ***<server-name>***

IMR Configuration

1. Set up a configuration file for each host in the location domain.
2. Run an IMR in master or dual mode on exactly one machine in your location domain.
3. Run an OAD on each of the other hosts in the location domain by running the IMR in slave mode.

Once you have configured the IMR, run the **`imr`** commands from a start-up script in **`/etc/rc`**.

You can explicitly add an OAD (instead of having OADs add themselves implicitly) with:

```
imradmin --add-oad [<host>]
```

To remove an OAD from the configuration:

```
imradmin --remove-oad [<host>]
```

IMR Properties

The IMR and OAD use configuration properties:

- `ooc.imr.dbdir=<dir>`
- `ooc.imr.forward_port=<port>`
- `ooc.imr.admin_port=<port>`
- `ooc.imr.administrative=<true/false>`
- `ooc.imr.slave_port=<port>`
- `ooc.imr.mode=<dual/master/slave>`
- `ooc.orb.service.IMR=<corbaloc URL>`
- `ooc.imr.trace.peer_status=<level>`
- `ooc.imr.trace.process_control=<level>`
- `ooc.imr.trace.server_status=<level>`

The Boot Manager

The IMR can act as a boot manager for **corbaloc** references.

A URL of the form

corbaloc::*<IMR-host>/<token>*

returns the object reference for the service identified by ***<token>*** as configured for the IMR.

For example:

corbaloc::janus.iona.com/NameService

denotes the naming service as returned by **resolve_initial_references** when called by the IMR.

-ORBDefaultInitRef corbaloc::janus.iona.com

configures the initial reference environment of a client as for the IMR.

```
./client -ORBDefaultInitRef corbaloc::janus.iona.com
```

The `mkref` Tool

You can create an object reference on the command line:

`mkref <server-name> <object-ID> <poa-name>`

For example:

`mkref CCS_server the_controller Controller`

This writes a **`corbaloc`** reference to standard output that you can use to configure clients:

```
corbaloc::janus.ooc.com.au:9998/%AB%AC%AB0_RootPOA%00forward%00
%00%AB%AC%AB0CCS_server%00Controller%00%00the_controller
```

`mkref` is useful during installation, for example, if you want to produce an IOR for bootstrapping.

```
$ cat ob.config
ooc.orb.service.NameService=corbaloc::janus.ooc.com.au:5000/NameS
ervice
ooc.orb.service.IMR=corbaloc::janus.ooc.com.au:9999/IMR
ooc.imr.mode=dual
ooc.imr.administrative=true
ooc.imr.dbdir=/home/michi/imr
ooc.imr.admin_port=9999
ooc.imr.forward_port=9998
ooc.imr.slave_port=9997
ooc.imr.trace.peer_status=2
ooc.imr.trace.process_control=2
ooc.imr.trace.server_status=2
$ ORBACUS_CONFIG=`pwd`/ob.config
$ export ORBACUS_CONFIG
```

```
$ /opt/OB4/bin/nameserv -OAport 5000 &
[1] 7292
$ imr &
[2] 7295
[ IMR: register_oad: janus ]
[ IMR: OAD for janus processes: EMPTY ]
[ OAD: ready for janus ]
$ imradmin --tree
domain
`-- janus  (up)
```

```
$ imradmin --add-server CCS_server `pwd`/launch
$ imradmin --tree
domain
`-- janus  (up)
    `-- CCS_server (stopped)
$ imradmin --set-server CCS_server args `pwd`/server
$ imradmin --get-server-info CCS_server
Server CCS_server:
    ID:                          1
    Status:                      stopped
    Name:                        CCS_server
    Host:                        janus
    Path:                        /home/michi/labs/imr/launch
    RunDir:
    Arguments:                   /home/michi/labs/imr/server
    Activation Mode:             shared
    POA Activation:              true
    Update timeout (ms):         20000
    Failure timeout (secs):      60
    Maximum spawn count:         2
    Started manually:            no
    Number of times spawned:     0
```

```
$ mkref CCS_server the_controller Controller >ctrl.ref
$ cat ctrl.ref
corbaloc::janus:9998/%AB%AC%AB0_RootPOA%00forward%00%00%AB%AC%AB0
CCS_server%00Controller%00%00the_controller
$ nsadmin -c michi
$ nsadmin --bind michi/controller `cat ctrl.ref`
$ nsadmin -r michi/controller
IOR:01f289400100000000000000010000000000000005a0000000101005011000
0006a616e75732e6f6f632e636f6d2e617500000e273a000000abacab305f526f
6f74504f4100666f72776172640000abacab306d6963686900436f6e74726f6c6
c65720000007468655f636f6e74726f6c6c6572
```

Overview

ORBacus supports a number of concurrency models for clients and servers. These models control how requests are mapped onto threads:

- Blocking (default for clients, applies only to clients)
- Reactive (default for servers)
- Threaded
- Thread-per-Client
- Thread per request
- Thread pool

You can select a concurrency model by setting a property, by passing an option on the command line, or programmatically.

The Blocking Concurrency Model

The blocking concurrency model applies only to clients and is the default model.

- After sending a request, the client-side run time enters a blocking read to wait for the reply from the server.
- For **oneway** requests, the ORB avoids blocking the client by holding the request in a buffer if it would block the client. Buffered requests are sent during the next request that goes to the same server.

No other activity can take place in the client while the client waits for a request to complete.

Each call is synchronous for the application code and the ORB.

The blocking model is simple and fast.

The Reactive Concurrency Model

The reactive concurrency model applies to clients and servers and is the default for servers.

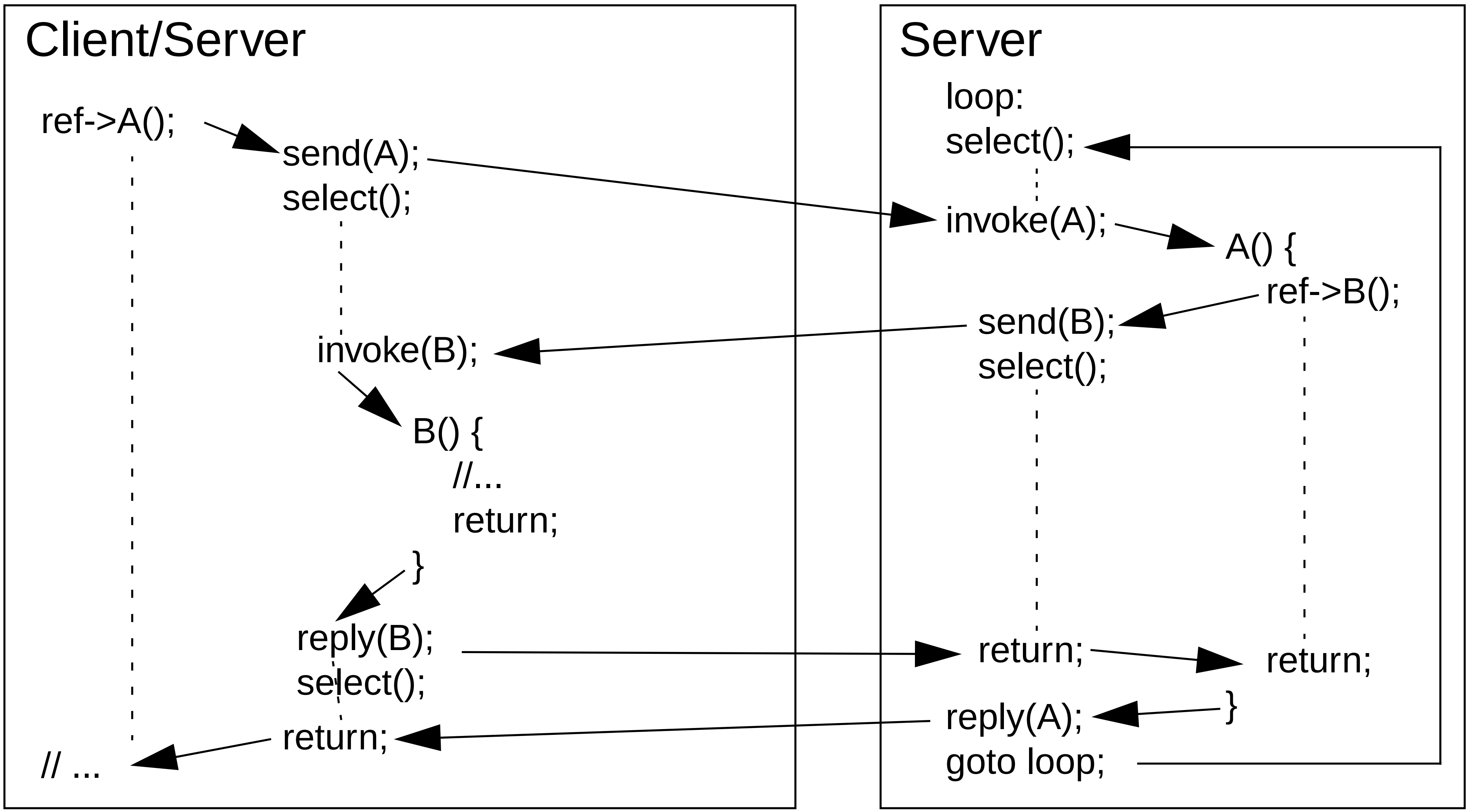
The reactive model is single-threaded.

- For servers, a **`select`** loop is used to monitor existing connections. This permits the server to accept requests from several clients.
- For clients, after sending a request, the client-side run time calls **`select`** instead of using a blocking read.

  If the client is also a server, it can accept incoming calls to its objects while it is acting as the client for a synchronous call to some other server.

The reactive model permits nested callbacks for single-threaded servers. (Many ORBs cannot support this without multiple threads.)

The Reactive Concurrency Model (cont.)



The Reactive Concurrency Model (cont.)

Advantages of the reactive concurrency model:

- permits creation of single-threaded processes that are both client and server
- avoids deadlock if callbacks are nested
- asynchronous dispatch of multiple buffered oneway requests to different servers
- transparent to the application code (but beware that operations may be called reentrantly)
- permits integration of foreign event loops

The Threaded Concurrency Model

The threaded concurrency model applies to clients and servers.

The ORB run time runs with threads, so sending and receiving of network packets can proceed in parallel for many requests.

- For clients, multiple deferred requests sent with the DII are truly dispatched in parallel, and **oneway** invocations do not block.
- For servers, the threaded model demultiplexes requests and unmarshals in parallel.

To the application code, the threaded model appears single-threaded.

Operation bodies in the server are strictly serialized!

This model is useful on multi-processor machines for servers under high load.

The Thread-per-Client Concurrency Model

The thread-per-client concurrency model applies to the server side.

The ORB creates one thread for each incoming connection.

- Requests coming in on different connections are dispatched in parallel.
- Requests coming in on the same connection are serialized.
- Requests on POAs with **`SINGLE_THREAD_MODEL`** are serialized.
- Requests on POAs with **`ORB_CTRL_MODEL`** are dispatched in parallel if those requests are dispatched by different POA managers or are coming from different clients.

The model would better be named “thread-per-connection” because a single server can use multiple POA managers.

You must take care of critical regions in your application with this model!

Thread-per-Request Concurrency Model

The thread-per-request concurrency model only applies to servers.

- Each incoming request creates a new thread and is dispatched in that thread.
- No request for a POA with **`ORB_CTRL_MODEL`** is ever blocked from dispatch.
- On return from a request, its thread is destroyed.

The thread-per-request model supports nested callbacks with unlimited nesting depth (subject to memory constraints and limits on the maximum number of threads).

The model is inefficient for small operations (thread creation and destruction overhead dominates throughput).

Use this model only for long-running operations that do a substantial amount of work and can proceed in parallel.

The Thread Pool Concurrency Model

The thread pool concurrency model dispatches requests onto a fixed-size pool of threads.

- If a thread is idle in the pool, each incoming request is dispatched in a thread taken from the pool.
- The number of concurrent operations in the server is limited by the number of threads in the pool. (The run time uses two additional threads for each connection).
- Requests that arrive while all threads are busy are transparently delayed until a thread becomes idle.

This model is efficient because threads are not continuously created and destroyed and provides a high degree of parallelism.

For general-purpose threaded servers, it is the best model to use.

Selecting a Concurrency Model

Concurrency models are selected by:

- setting a property in a configuration file
- passing a command-line option
- setting a property programmatically

Properties that apply to concurrency models:

- **`ooc.orb.conc_model`** (client side)

  `blocking` (default), **`reactive`**, **`threaded`**

- **`ooc.orb.oa.conc_model`** (server side)

  `reactive` (default), **`threaded`**, **`thread_per_client`**, **`thread_per_request`**, **`thread_pool`**

- **`ooc.orb.oa.thread_pool=`*<n>***

```
# Select threaded for the client role
ooc.orb.conc_model=threaded

# Select thread pool with 20 threads for the server role
ooc.orb.oa.conc_model=thread_pool
ooc.orb.oa.thread_pool=20
```

```
./a.out -ORBthreaded -OAthread_pool 20
```

```
// Get default properties (established by config file)
OB::Properties_var dflt = OB::Properties::getDefaultProperties();

// Initialize a property set with the defaults
OB::Properties_var props = new OB::Properties(dflt);

// Set the properties we want
props->setProperty("ooc.orb.conc_model", "threaded");
props->setProperty("ooc.orb.oa.conc_model", "thread_pool");
props->setProperty("ooc.orb.oa.thread_pool", "20");

// Initialize the ORB with the given properties
CORBA::ORB_var orb = OBCORBA::ORB_init(argc, argv, props);
```

Overview of JThreads/C++

- JThreads/C++ (JTC) is required for ORBacus to support threaded models.
- JTC is a threads abstraction library.
- JTC is implemented as a thin layer on top of the underlying native threads package.
- JTC adds virtually no overhead.
- JTC provides a Java-like thread model (simpler than POSIX threads).
- JTC shields you from idiosyncrasies of the underlying native threads package.
- JTC provides common synchronization, mutual exclusion, and thread control primitives.

JTC Initialization

Your code must contain a **`#include <JTC/JTC.h>`** directive.

You must initialize JTC before making any other JTC-related calls by constructing a **`JTCInitialize`** instance:

```
void JTCInitialize();
void JTCInitialize(int & argc, char * * argv);
```

The second constructor works like **`ORB_init`** in that it looks for JTC-specific command line options and strips these options from **`argv`**.

Valid options are:

- **`-JTCversion`**
- **`-JTCss`** ***`<stack_size>`*** (in kB)

If you call **`ORB_init`**, you need not use **`JTCInitialize`**.

```
#include <JTC/JTC.h>

// ...

int
main(int argc, char * argv[])
{
    // Initialize JTC
    JTCInitialize jtc(argc, argv);

    // ...
}
```

Simple Mutexes

The **`JTCMutex`** class provides a simple non-recursive mutex:

```
class JTCMutex {
public:
    void lock();
    void unlock();
};
```

You must:

- call **`unlock`** only on a locked mutex
- call **`unlock`** only from the thread that called **`lock`**

Calling **`lock`** on a mutex that the calling thread has already locked causes deadlock.

Never destroy a mutex that is locked!

```
class MyClass {
public:
    void do_something() {
        // ...

        // Start critical region
        m_mutex.lock();

        // Update shared data structure here...

        // End
        m_mutex.unlock()

        // ...
    }
private:
    JTCMutex m_mutex;
};
```

Recursive Mutexes

`JTCRecursiveMutex` provides a mutex that can be locked multiple times by its owner:

```
class JTCRecursiveMutex {
public:
    void lock();
    void unlock();
};
```

- The first thread to call **`lock`** locks the mutex and the calling thread becomes its owner.
- Multiple calls to **`lock`** increment a lock count.
- The owner must call **`unlock`** as many times as **`lock`** to unlock the mutex.

Otherwise, the same restrictions apply as for non-recursive mutexes.

Automatic Unlocking

You must ensure that a mutex is unlocked before it is destroyed. **`JTCSynchronized`** makes this easy:

```
JTCSynchronized {
    JTCSynchronized(JTCMutex & m);          // Lock mutex
    JTCSynchronized(JTCRecursiveMutex & m); // Lock rec. mutex
    JTCSynchronized(JTCMonitor & m);        // Lock monitor
    ~JTCSynchronized();                     // Unlock
};
```

The constructor calls **`lock`** and the destructor calls **`unlock`**. This makes it impossible to leave a block containing a **`JTCSynchronized`** object without calling **`unlock`**.

`JTCSynchronized` makes errors much less likely, especially if you have multiple return paths or call something that may throw an exception. The class works for mutexes, recursive mutexes, and monitors.

```
class MyClass {
public:
    void do_something() {
        // ...

        // Start critical region
        JTCSynchronized lock(m_mutex);

        // Update shared data structure here...
        // ...

    } // Critical region ends here
private:
    JTCMutex m_mutex;
};
```

Monitors

The **`JTCMonitor`** class implements a Java-like monitor:

```
class JTCMonitor {
public:
            JTCMonitor();
    virtual ~JTCMonitor();
    void    wait();        // Wait for condition
    void    wait(long n); // Wait at most n msec for condition
    void    notify();      // Wake up one thread
    void    notifyAll();  // Wake up all threads
};
```

Only one thread can enter the critical region protected by a monitor.

A thread inside the region can call **`wait`** to suspend itself and give access to another thread.

When a thread changes the condition, it calls notify to wake up a thread that was waiting for the condition to change.

Simple Producer/Consumer Example

Assume we have a simple queue class:

```
template<class T> class Queue {
public:
    void enqueue(const T & item);
    T    dequeue();
};
```

- Producer threads read items from somewhere and place them on the queue by calling **enqueue**.
- Consumer threads fetch items from the queue by calling **dequeue**.
- The queue is a critical region and the consumer threads must be suspended when the queue is empty.

```
#include <list>

template<class T> class Queue {
public:
    void enqueue(const T & item) {
            m_q.push_back(item);
        }
    T    dequeue() {
            T item = m_q.front();
            m_q.pop_front();
            return item;
        }
private:
    list<T> m_q;
};
```

```
#include <list>
#include <JTC/JTC.h>

template<class T> class Queue : JTCMonitor {
public:
    void    enqueue(const T & item) {
                JTCSynchronized lock(*this);
                m_q.push_back(item);
                notify();
            }
    T       dequeue() {
                JTCSynchronized lock(*this);
                while (m_q.size() == 0) {
                    try {
                        wait();
                    } catch (const JTCInterruptedException &) {
                    }
                }
                T item = m_q.front();
                m_q.pop_front();
                return item;
            }
private:
    list<T> m_q;
};
```

Rules for Using Monitors

You must always catch and ignore a **`JTCInterrupted`** exception around a **`wait`**:

```
T dequeue() {
    JTCSynchronized lock(*this);
    while (m_q.size() == 0)
        wait();                        // WRONG!!!
    T item = m_q.front();
    m_q.pop_front();
    return item;
}
```

Failure to catch and ignore the exception may result in undefined behavior.

In addition, you must call wait and notify with the mutex locked; otherwise, you get a **`JTCIllegalMonitorState`** exception.

```
T dequeue() {
    JTCSynchronized lock(*this);
    while (m_q.size() == 0) {
        try {
            wait();
        } catch (const JTCInterruptedException &) { // Correct
        }
    }
    T item = m_q.front();
    m_q.pop_front();
    return item;
}
```

Rules for Using Monitors (cont.)

Always test the condition under protection of the monitor:

```
T dequeue() {
    while (m_q.size() == 0) {            // WRONG!!!
        JTCSynchronized lock(*this);
        try {
            wait();
        } catch (const JTCInterruptedException &) {
        }
        T item = m_q.front();
        m_q.pop_front();
        return item;
    }
}
```

If you do not acquire access to the critical region first, the condition may be changed by another thread in between the test and the update.

```
T dequeue() {
    JTCSynchronized lock(*this); // Correct
    while (m_q.size() == 0) {
        try {
            wait();
        } catch (const JTCInterruptedException &) {
        }
    }
    T item = m_q.front();
    m_q.pop_front();
    return item;
}
```

Rules for Using Monitors (cont.)

Always retest the condition when coming out of a **`wait`**:

```
T dequeue() {
    JTCSynchronized lock(*this);
    if (m_q.size() == 0) {          // WRONG!!!
        try {
            wait();
        } catch (const JTCInterruptedException &) {
        }
    }
    T item = m_q.front();
    m_q.pop_front();
    return item;
}
```

If you do not retest the condition, it may not be what you expect!

```
T dequeue() {
    JTCSynchronized lock(*this);
    while (m_q.size() == 0) {           // Correct
        try {
            wait();
        } catch (const JTCInterruptedException &) {
        }
    }
    T item = m_q.front();
    m_q.pop_front();
    return item;
}
```

Static Monitors

Occasionally, you need to protect static data from concurrent access. You can safely use a **`JTCMonitor`** to do this:

```
class StaticCounter {
public:
    static void             inc() { JTCSynchronized lock(m_m);
                                    ++m_counter;             }
    static void             dec() { JTCSynchronized lock(m_m);
                                    --m_counter;             }
    static unsigned long    val() { JTCSynchronized lock(m_m);
                                    return m_counter;        }
private:
    static unsigned long    m_counter;
    static JTCMonitor       m_m;
};

unsigned long   StaticCounter::m_counter = 0;
JTCMonitor      StaticCounter::m_m;
```

The JTCThread Class

To create a new thread, you instantiate a class instance that is derived from **JTCThread**:

```
class JTCThread : public virtual JTCRefCount {
public:
                    JTCThread();
                    ~JTCThread();
    virtual void    run();
    void            start();
    // ...
};
```

Override the **run** method to provide a starting stack frame for the thread.

Call **start** to set the thread running in its starting stack frame.

```
class ProducerThread : public virtual JTCThread {
public:
                    ProducerThread(Queue<int> & q, int c) :
                                            m_q(q), m_c(c) {}
    virtual void    run() {
                        for (int i = 0; i < m_c; ++i)
                            m_q.enqueue(i);
                    }
private:
    Queue<int> &    m_q;    // Thread-safe queue
    int             m_c;
};
```

```
class ConsumerThread : public virtual JTCThread {
public:
                    ConsumerThread(Queue<int> & q, int c) :
                                            m_q(q), m_c(c) {}
    virtual void    run() {
                        for (int i = 0; i < m_c; ++i)
                            m_q.dequeue(i);
                    }
private:
    Queue<int> &    m_q;    // Thread-safe queue
    int             m_c;
};
```

```
// ...

JTCInitialize jtcinit;      // Important!!!

Queue<int> the_queue;       // Queue to use

JTCThreadHandle consumer;   // Note: JTCThreadHandle
JTCThreadHandle producer;   // Note: JTCThreadHandle

// Start both threads
consumer = new ConsumerThread(the_queue, 10000);
producer = new ProducerThread(the_queue, 10000);
consumer->start();
producer->start();
```

Joining with Threads

Given a thread, any other thread can join with it:

```
class JTCThread : public virtual JTCRefCount {
public:
    // ...
    void            join();
    void            join(long msec);
    void            join(long msec, int nsec);
    bool            isAlive() const;
    //...
};
```

The purpose of **`join`** is to suspend the caller until the thread being joined with terminates.

Always join with threads in a loop, catching **`JTCInterrupted`** and reentering **`join`** if that exception was thrown!

```
// ...

// Wait for consumer thread to finish
do {
    try {
        consumer->join();
    } catch (const JTCInterruptedException &) {
    }
} while (consumer->isAlive());
```

Other JThreads/C++ Functionality

JThreads/C++ offers many more features:

- Named threads
- thread groups
- thread priorities
- **`sleep`** and **`yield`**
- thread-specific storage

Please consult the manual for details.

Synchronization Strategies for Servers

You can use several strategies for synchronization:

- Permit only one concurrent request per servant.

  This approach is very easy to implement with a monitor.

- Allows multiple concurrent read operations but require exclusive access to the entire object for a write operation.

  This approach provides more parallelism at the cost of greater implementation complexity. (You need to create reader/writer locks and synchronize explicitly by calling **`wait`** and **`notify`**.)

  Use this approach only if you have high contention on a servant, for example, with default servants.

For both approaches, take care of interactions among life cycle and collection manager operations!

Basic Per-Servant Synchronization

For basic per-servant synchronization, use inheritance from **`JTCMonitor`** for the servant:

```
class Thermometer_impl :
    public virtual POA_CCS::Thermometer,
    public virtual PortableServer::RefCountServantBase,
    public virtual JTCMonitor {
// ...
};
```

In each operation body, instantiate a **`JTCSynchronized`** object on entry to the operation.

With almost zero effort, all operations on the servant are serialized.

`JTCMonitor` uses recursive mutexes, so an operation implementation can invoke operations on its own servant without deadlock.

```
// IDL model attribute.
CCS::ModelType
Thermometer_impl::
model() throw(CORBA::SystemException)
{
    JTCSynchronized lock(*this);
    // ...
}

// IDL asset_num attribute.
CCS::AssetType
Thermometer_impl::
asset_num() throw(CORBA::SystemException)
{
    JTCSynchronized lock(*this);
    // ...
}

// etc...
```

Life Cycle Considerations

You must pay attention to potential race conditions for life cycle operations and collection manager operations:

- Factory operations, such as **`create_thermometer`** and **`create_thermostat`**, must interlock with themselves and with **`destroy`**.
- **`destroy`** must interlock with itself and with the factory operations.
- Collection manager operations, such as **`list`** and **`find`**, must interlock among each other and with the life cycle operations.

The easiest solution is to have a global life cycle lock.

This serializes all life cycle and collection manager operations, but permits other operations to proceed in parallel (if they are for different target objects).

```
class Controller_impl :
    public virtual POA_CCS::Controller,
    public virtual PortableServer::RefCountServantBase,
    public virtual JTCMonitor {
private:
    bool m_lifecycle_ok;    // True if OK to do a life cycle op

public:
    // ...
};
```

```
class Controller_impl :
    public virtual POA_CCS::Controller,
    public virtual PortableServer::RefCountServantBase,
    public virtual JTCMonitor {
private:
    bool m_lifecycle_ok;    // True if OK to do a life cycle op
public:
    // Life cycle guard methods
    void lifecycle_lock() {
        JTCSynchronized lock(*this);
        while (!m_lifecycle_ok) {
            try {
                wait();
            } catch (const JTCInterruptedException &) {
            }
        }
        m_lifecycle_ok = false;
    }
    void lifecycle_unlock() {
        JTCSynchronized lock(*this);
        m_lifecycle_ok = true;
        notify();
    }
    // ...
};
```

```
Controller_impl::
Controller_impl(const char * asset_file) throw(int)
    : m_asset_file(asset_file), m_lifecycle_ok(true)
{
    // ...
}
```

```
CCS::Thermometer_ptr
Controller_impl::
create_thermometer(CCS::AssetType anum, const char * loc)
throw(CORBA::SystemException, CCS::Controller::DuplicateAsset)
{
    m_ctrl->lifecycle_lock();

    if (exists(anum))
        throw CCS::Controller::DuplicateAsset();    // OOPS!!!
    if (ICP_online(anum) != 0)
        abort();
    if (ICP_set(anum, "location", loc) != 0)
        abort();
    Thermometer_impl * t = new Thermometer_impl(anum);
    PortableServer::ObjectId_var oid = make_oid(anum);
    Thermometer_impl::poa()->activate_object_with_id(oid, t);
    t->_remove_ref();

    m_ctrl->lifecycle_unlock();

    return t->_this();
}
```

```
class Controller_impl :
    public virtual POA_CCS::Controller,
    public virtual PortableServer::RefCountServantBase,
    public virtual JTCMonitor {
private:
    bool m_lifecycle_ok;    // True if OK to do a life cycle op
public:
    // Life cycle methods
    void lifecycle_lock() { /* ... */ };
    void lifecycle_unlock() { /* ... */ };
    class LifeCycleSynchronized {
    public:
        static Controller_impl * m_ctrl;
        LifeCycleSynchronized() { m_ctrl->lifecycle_lock(); }
        ~LifeCycleSynchronized() { m_ctrl->lifecycle_unlock(); }
    };
    // ...
};
```

```
CCS::Thermometer_ptr
Controller_impl::
create_thermometer(CCS::AssetType anum, const char * loc)
throw(CORBA::SystemException, CCS::Controller::DuplicateAsset)
{
    LifeCycleSynchronized lock;
    // ...
}

CCS::Thermostat_ptr
Controller_impl::
create_thermostat(
    CCS::AssetType  anum,
    const char*     loc,
    CCS::TempType   temp)
throw(
    CORBA::SystemException,
    CCS::Controller::DuplicateAsset,
    CCS::Thermostat::BadTemp)
{
    LifeCycleSynchronized lock;
    // ...
}

CCS::Controller::ThermometerSeq *
```

```
Controller_impl::
list() throw(CORBA::SystemException)
{
    LifeCycleSynchronized lock;
    // ...
}

void
Controller_impl::
find(CCS::Controller::SearchSeq & slist)
throw(CORBA::SystemException)
{
    LifeCycleSynchronized lock;
    // ...
}
```

```
void
Thermometer_impl::
destroy() throw(CORBA::SystemException)
{
    m_ctrl->lifecycle_lock();

    // Remove entry in the AOM for the servant.
    // Controller map and persistent state are cleaned up in
    // the servant destructor.
    PortableServer::ObjectId_var oid = make_oid(m_anum);
    PortableServer::POA_var poa = _default_POA();
    poa->deactivate_object(oid);

    m_removed = true;   // Mark device as destroyed

    // Note: lifecycle lock is still held.
}
```

```
Thermometer_impl::
~Thermometer_impl()
{
    // Remove device from map and take it off-line
    // if it was destroyed.
    if (m_removed) {
        m_ctrl->remove_impl(m_anum);
        if (ICP_offline(m_anum) != 0)
            abort();
    }

    // Permit life cycle operations again.
    m_ctrl->lifecycle_unlock();
}
```

Threading Guarantees for the POA

- **`_add_ref`** and **`_remove_ref`** are thread safe.
- For requests arriving on the same POA
  - calls to **`incarnate`** and **`etherealize`** on a servant activator are serialized,
  - calls to **`incarnate`** and **`etherealize`** are mutually exclusive,
  - **`incarnate`** is never called for a specific object ID while that object ID is in the AOM.
- For requests arriving on different POAs with the same servant activator, no serialization guarantees are provided.
- **`preinvoke`** and **`postinvoke`** are not interlocked. **`preinvoke`** may be called concurrently for the same object ID.
- **`preinvoke`**, the operation, and **`postinvoke`** run in one thread.