Client-Side C++ Mapping

Summary

- Client-Side C++ Mapping:
  - Object references
  - Invoking operations on objects
  - Handling exceptions

Mapping for Interfaces

- The proxy class - location-independent object interface
- IDL:

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

- Generated C++ proxy class:

```
class MyObject : public virtual
CORBA::Object{
  public:
   virtual CORBA::Long get_value() = 0;
};
```

You Must Not

- Declare a pointer to the proxy class
- Declare a reference to the proxy class

```
MyObject myobj;
MyObject * mop;
void f(MyObject &);
```

Object Reference Types

- The IDL compiler generates two types of object references:
  - **`InterfaceName_ptr`**
  - **`InterfaceName_var`**
- Using object references:

```
MyObject_ptr mop = …;// get _ptr reference
CORBA::Long v1 = mop->get_value(); //get value from object.

MyObject_var mov = …; // get another reference
CORBA::Long v2 = mov->get_value(); //get value from object
```

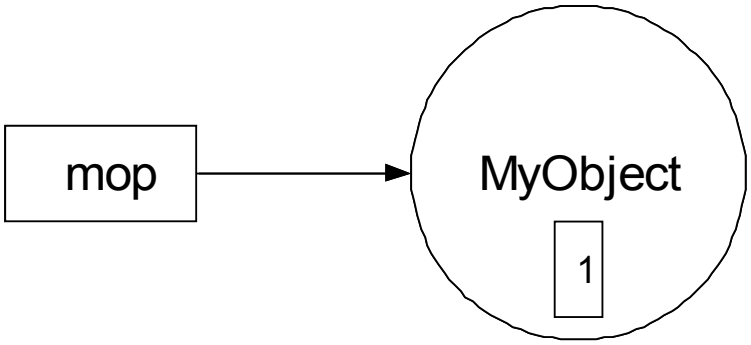
Object References and Clients

- The client can:
  - Obtain `_ptr` reference from ORB
  - Remove reference
  - Copy reference
  - Create an empty reference (nil reference)

Creating Object References

- The proxy class object behave as if they had a reference counter

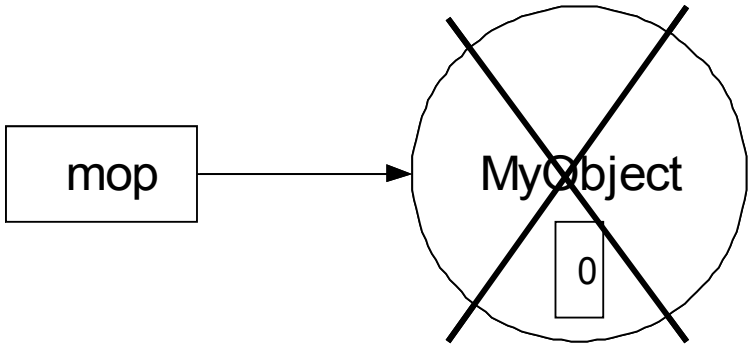
```
MyObject_ptr mop = …; // Get reference from somewhere
```



Removing References

- The proxy class uses the client resources
- The client code informs the proxy, that it no longer needs it

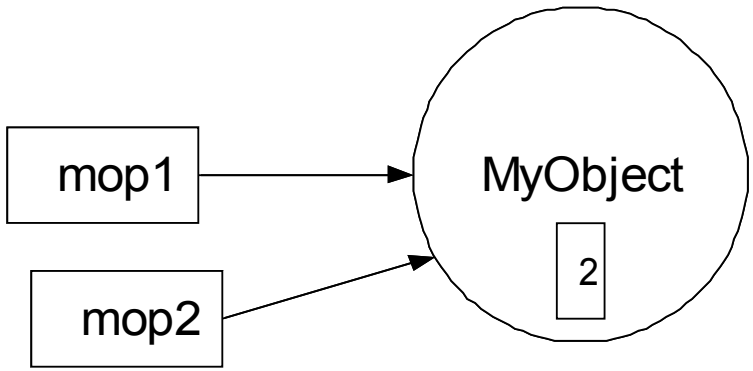
```
CORBA::release(mop);  // Done with this object
```



Copying References

- The IDL compiler generates a static member **`_duplicate`** in every proxy class:

```
class MyObject : public virtual CORBA::Object{
    public:
      virtual CORBA::Long get_value() = 0;
      static MyObject_ptr _duplicate(MyObject_ptr p);
};
MyObject_ptr mop1 =…;
MyObject_ptr mop2 = MyObject::_duplicate(mop1); // Make copy
```



Releasing References

- Using references after they are released is forbidden:

```
MyObject_ptr mop1 =…;
MyObject_ptr mop2 = MyObject::_duplicate(mop1); // Make copy
CORBA::release(mop2); // Release mop2
CORBA::Long v1 = mop2->get_value(); // Illegal, released already
CORBA::Long v1 = mop1->get_value(); // OK
CORBA::release(mop1); // Release mop1
```

- Freeing the same reference twice is not allowed
- Releasing references does not influence any objects on server, it only influences references at the client

Empty References

- The IDL compiler generates the static member **`_nil`** in every proxy class:

```
class MyObject : public virtual CORBA::Object {
 public:
    virtual CORBA::Long get_value() = 0;
    static MyObject_ptr _duplicate(MyObject_ptr p);
    static MyObject_ptr _nil();
    // ...
};
```

Empty References (contd.)

- Empty references can be released and copied as normal references

```
MyObject_ptr p1 = MyObject::_nil();
MyObject_ptr p2 = MyObject::_duplicate(p1);
// ...
// Release both references
CORBA::release(p2); // optional
CORBA::release(p1); // optional
```

- The result of operation invocation via empty references is undefined

```
MyObject_ptr p1 = MyObject::_nil();
CORBA::Long l = p->get_value(); // Crash imminent here!
```

Empty References (contd.)

- Testing for an empty reference is performed using function **`CORBA::is_nil`**:

```
MyObject_ptr p = …;
if(!CORBA::is_nil(p))
   CORBA::Long l = p->get_value(); //Call only if not nil
CORBA::release(p);
```

- The code below is incorrect:

```
MyObject_ptr p = …;
if(p != 0) // illegal
   do_something()
if(p != MyObject::_nil()) // also illegal
   do_something()
```

Why Empty References?

- "Nonexistent" or "optional"

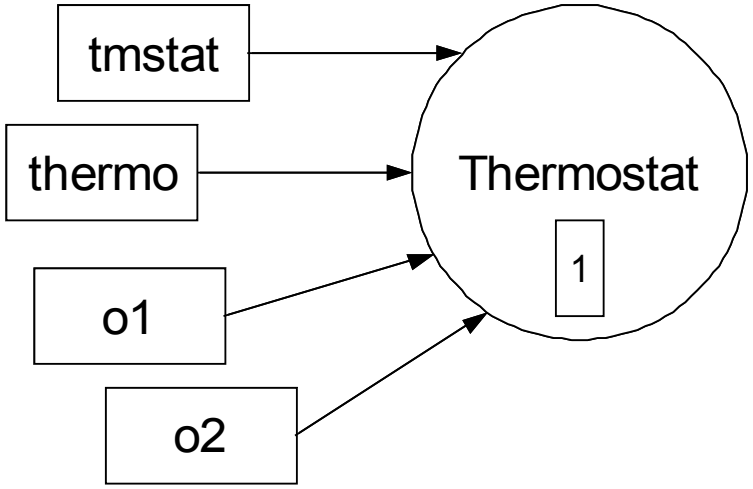
```
interface Callback {
  void disconnect();
};
interface Channel {
  SomeType register_me(in Callback c);
  // ...
};

Channel_ptr ch = ...; // Get a channel reference ...
// Tell the channel we don't want to know about disconnects
Callback_ptr nil_cb = Callback::_nil();
SomeType st = ch->register_me(nil_cb);
//use channel for other things
```

_ptr References

- Implicit widening
  - **derived_ptr** can be assigned to **base_ptr**

```
Module CCS {
   interface Thermostat : Thermometer {
   TempType get_nominal();
       };
};
```



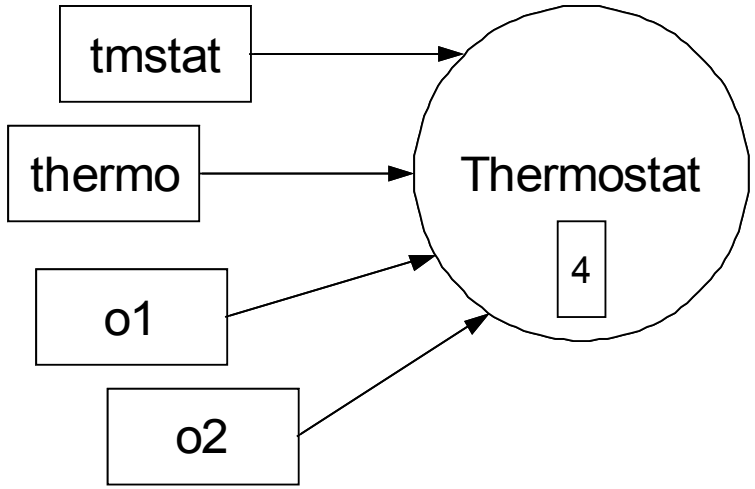

```
CCS::Termostat_ptr tmstat = …;   // Get Thermostat ref…
CCS::Thermometer_ptr thermo = tmstat; // OK, compatible
                                      // assignment
CORBA::Object_ptr o1 = tmstat;
CORBA::Object_ptr o2 = thermo;
```

_ptr References (contd.)

```
CCS::TempType t;
t = tmstat->get_nominal(); //OK, read nominal temperature
t = thermo->get_nominal(); //Compile time error, cannot access
                           //derived part via base reference
t = o1->get_nominal(); //compile time error too
CORBA::release(thermo); //or CORBA::release(tmstat)
                        //or CORBA::release(o1)
                        //or CORBA::release(o2)
//cannot use tmstat, thermo, o1, or o2 from here on ...
```

_ptr References (contd.)

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



_ptr References (contd.)

- Narrowing

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


CCS::Thermostat_ptr tmstat =
                (CCS::Thermostat_ptr)thermo; //Disastrous!!!
```

- Safe narrowing

```
CCS::Thermometer_ptr thermo = ...; //Get Thermometer reference
// Try type-safe down-cast
CCS::Thermostat_ptr tmstat = CCS::Thermostat::_narrow(thermo);
if(CORBA::is_nil(tmstat)) {
  // thermo is not of type Thermostat
} else {
  // thermo *is a* Thermostat, tmstat is now a valid reference
}
CORBA::release(tmstat); // _narrow () calls _duplicate()!
```

Illegal Application of `_ptr` References

- Comparison

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

- Arithmetic operations

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

- Conversion to and from **`void*`**

```
CORBA::Object_ptr o1 = ...;
void *v = (void*) o; // Undefined!
o = (CORBA::Object_ptr)v; // Ditto!
```

Illegal Application of `_ptr` References (contd.)

- Narrowing other than by the means of **`_narrow`**

```
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
```

- Checking for an empty pointer other than by the means of **`CORBA::is_nil`**

```
CCS::Thermostat_ptr tmstat = CCS::Thermostat::_nil();
if(tmstat) ... // Illegal
if(tmstat != 0) ... // Illegal
if(tmstat != CCS::Thermostat::_nil()) ... // Illegal
if(!CORBA::is_nil(tmstat)) ... // OK
```

Pseudointerfaces

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

- Interfaces defined in PIDL
  - Do not inherit from **`Object`**
  - Cannot be passed as parameters to regular interfaces
  - Operations on them cannot be invoked via DII
  - Do not have definitions in Interface Repository
  - May have non-standard language mapping

ORB Initialization

```
module CORBA { //PIDL
   typedef string ORBid;
   typedef sequence<string> arg_list;
//Forward declaration
   interface ORB;
   ORB ORB_init(inout arg_list argv,
                in ORBid orb_identifier);
};
namespace CORBA {
   ORB_ptr ORB_init(
      int &      argc,
      char **    argv,
      const char*  orb_identifier = ""
);
};
```

ORB Initialization (contd.)

```
int main(int argc, char * argv[]){
   CORBA::ORB_ptr orb;
   try {
      orb = CORBA::ORB_init(argc, argv);
    }
   catch(…){ cerr <<"Cannot init ORB" <<endl; }
  // Use ORB …
   CORBA::release(orb);
   return 0;
}
```

Initial References

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

namespace CORBA {
   class ORB {
   public:
  char * object_to_string(Object_ptr p);
  Object_ptr string_to_object(const char *s);
   };
};
```

Initial References - Example

```
CORBA::ORB_ptr orb =
   CORBA::ORB_init(argc, argv);
CORBA::Object_ptr obj;
try{
   obj = orb->
     string_to_object(argv[1]);
catch(…){…}
if(CORBA::is_nil(obj)){
   cerr << "Passed ref nil" <<
                          endl; }
CSS::Controller_ptr ctrl;
try {
   ctrl= CCS::Controller::
_narrow(obj);
} catch(…){…}
```

```
CORBA::release(obj);

if(CORBA::is_nil(ctrl)){
   cerr << "Argument not a
controller ref"<<endl;
}

//
// Use controller reference…
//

//Clean up
CORBA::release(ctrl); //Narrow
calls _duplicate
CORBA::release(orb); //Clean up
```

object_to_string

```
CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);
CCS::Controller_ptr ctrl = ...;// Get reference ...
char * s;
try {
  s = orb->object_to_string(ctrl);
}
catch (...) { ... }
cout << s << endl; // Print reference
CORBA::string_free(s); // Finished with string
CORBA::release(ctrl); // Release controller proxy
CORBA::release(orb); // Shut down runtime
```

The Object Pseudointerface

```
module CORBA { //PIDL
interface Object {
  Object duplicate();
  void release();
  boolean is_nil();
  boolean is_a(in string repository_id)
  boolean non_existent();
  boolean is_equivalent(in Object o_obj);
   unsigned long hash(in unsigned long max);
  };
};
class Object{
public:
  Boolean _is_a(const char * repository_id);
  Boolean _non_existent();
  Boolean _is_equivalent(Object_ptr other_object);
  Ulong _hash(Ulong max);
};
```

The `Object` Pseudointerface (contd.)

- Operations from the previous slide cannod be invoked via empty references

```
CORBA::Object_ptr p = CORBA::Object::_nil() // Make nil ref
if(p->_non_existent()) // Crash imminent!!!
  // ...
```

- If we are not sure, we should check:

```
if(CORBA::is_nil() || p->_non_existent())
  // Objref is nil or dangles
```

Operation _is_a

- Returns true, if the reference has given interface

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

- It can be also of derived type

```
CORBA::Object_ptr obj = ...; // Get actual thermostat
reference
assert(obj->_is_a("IDL:acme.com/CCS/Thermometer:1.0"));
assert(obj->_is_a("IDL:omg.org/CORBA/Object:1.0"));
```

- Used with DII

Operation _non_existent

- Returns true, if the object pointed by reference does not exist

```
CORBA::Object_ptr obj = ...; // Get reference to some object
try {
  if (obj->_non_existent()) {
    // Object is gone forever
  } else {
    // Object definitely exists
  }
}
catch (const CORBA::TRANSIENT & ) {
  // Couldn't decide whether or not object exists
}
catch( ... ) {
  // Something else went wrong
}
```

- Is not a ping-type operation, does not have to contact server
- Can cause start-up of the server

Operation _is_equivalent

- Returns true, if the reference is identical to another one

```
CORBA::Object_ptr o1 = ...; // Get some reference
CORBA::Object_ptr o2 = ...; // Get another reference
if (o1->_is_equivalent(o2)) {
    // o1 and o2 denote the same object
  } else {
    // o1 and o2 may or may not denote the same object
  }
}
catch (const CORBA::TRANSIENT & ) {
```

- If it returns false, the references can point to the same object
- To be sure about it, we have to implement it ourselves

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

The Summary of `Object` Operations

IDL Operation	C++ Function
`Object duplicate();`	`static Interface_ptr Interface::_duplicate(Interface_ptr src);`
`void release();`	`void CORBA::release(Object_ptr p);`
`boolean is_nil();`	`Boolean CORBA::is_nil(Object_ptr p);`
`boolean is_a(in string id);`	`Boolean Object::_is_a(const char* id);`
`boolean is_equivalent (in Object other_obj);`	`Boolean Object::_is_equivalent (Object_ptr other obj);`
`unsigned long hash(in unsigned long max);`	`ULong Object::_hash(ULong max);`
	`static Interface_ptr Interface::_nil();`

Mapping for _var References

```
namespace CCS {
  class Thermometer {/*…*/};
  typedef Thermometer * Thermometer_ptr;

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

Mapping for _var References (contd.)

```
    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 p; // actual reference stored here
   };
}
```

_var and _ptr References

```
// IDL
interface Base {/* … */};
interface Derived : Base {/* … */};

// C++
Base_ptr B_ptr;
Derived_ptr  D_ptr;
Base_var   B_var;
Derived_var D_var;
```

_var and _ptr References (contd.)

```
B_ptr = B_var; // Shallow assignment, B_var retains ownership
B_ptr = D_var; // Shallow assignment, D_var retains ownership
B_var = B_ptr; // Shallow assignment, B_var takes ownership
B_var = D_ptr; // Shallow assignment, B_var takes ownership
B_var = B_var; // Deep assignment
B_ptr = D_ptr; // Shallow assignment

D_ptr = B_var; // Illegal, compile time error
D_var = B_ptr; // Illegal, compile time error

B_var = D_var; // Illegal, compile time-error
// use:
B_var = Derived::_duplicate(D_var);
// or:
B_var = Base::_duplicate(D_var);

D_var = B_var; // Illegal, compile time-error
```

References Nested within User-defined Types

```
// IDL
interface Controller {
    typedef sequence<Thermometer> ThermometerSeq;
    //…
    ThermometerSeq list();
    //…
};

// C++
class ThermometerSeq {
public:
    ThermometerSeq();
    ThermometerSeq(CORBA::ULong max);
    ThermometerSeq(
        CORBA:: ULong max;
        CORBA::ULong len;
        Thermometer_ptr * data;
        CORBA::Boolean release = 0);
    Thermometer_mgr & operator[](CORBA::Ulong idx);
    Thermometer_mgr & operator[](CORBA::Ulong idx) const;
    // etc …
};
```

References Nested within User-defined Types (contd.)

```
CCS::Thermometer_var tv = …;
CCS::Thermometer_ptr tp = …;
{
  CCS::ThermometerSeq seq;
  seq.length(2);
  seq[0] = tv; // Deep assignment
  seq[1] = tp; // seq[1] takes ownership
}
// Sequence releases both seq[0] and seq[1]

CCS::TempType t;
t = tv->temperature(); //OK, tv is still intact
t = tp->temperature(); //Disaster, tp dangles
```

Mapping for Operations

```
// IDL
interface Foo {
   void send(in char c);
   oneway void put(in char c);
   long get_long();
   string id_to_name(in string id);
};
//C++
class Foo {
public:
   //…
   virtual void send(CORBA::Char c) = 0;
   virtual void put(CORBA::Char c) = 0;
   virtual CORBA::Long get_long() = 0;
   virtual char* id_to_name(const char *id) = 0;
   //…
};
```

Parameter Passing - Simple Types

```
// IDL
interface Foo {
   long long_op(in long l_in,
         inout long l_inout,
         out long l_out);
};
// C++
class Foo : public CORBA::Object {
public:
   //…
   virtual CORBA::Long long_op(
      CORBA::Long l_in,
      CORBA::Long & l_inout,
      CORBA::Long_out l_out
      ) = 0;
   //…
};
```

Parameter Passing - Simple Types (contd.)

```
Foo_var fv = ...; //Get reference
CORBA::Long inout_val;
CORBA::Long out_val;
CORBA::Long ret_val;

inout_val = 5;
ret_val = fv->long_op(99, inout_val, out_val);

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

Parameter Passing - Fixed-length Compound Types

```
struct Fls {
    long l_mem;
    double d_mem;
};

interface Foo {
    Fls fls_op(in Fls fls_in,
               inout Fls fls_inout,
               out Fls fls_out);
};
Class Foo : public CORBA::Object{
public:
    //…
    virtual Fls fls_op(const Fls & fls_in,
                  Fls & fls_inout,
                  Fls_out fls_out
                       ) = 0;
    //…
};
```

Parameter Passing - Fixed-length Compound Types (contd.)

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

Fls in_val;
Fls inout_val;
Fls out_val;
Fls ret_val;

in_val.l_mem = 99;
in_val.d_mem = 3.14;

inout_val.l_mem = 5;
inout_val.d_mem = 2.18;

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

Parameter Passing - Arrays of Fixed-Length Elements

```
typedef double Darr[3];
interface Foo{
   Darr darr_op(in Darr darr_in,
         inout Darr darr_inout,
         out Darr darr_out);
};

typedef CORBA::Double Darr[3];
typedef CORBA::Double Darr_slice;
class Foo : public virtual CORBA::Object{
public:
   //…
   virtual Darr_slice * darr_op(
      const Darr darr_in,
      Darr_slice * darr_inout,
      Darr_out darr_out) = 0;
};
void Darr_free(Darr_slice *);
```

Parameter Passing - Arrays of Fixed-Length Elements (contd.)

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

Darr_in_val = {0.0, 0.1, 0.2};
Darr inout_val = {97.0, 98.0, 99,0};
Darr out_val;
Darr_slice * ret_val;

ret_val = fv->darr_op(in_val, inout_val, out_val);
// ...
Darr_free(ret_val); // Must free here!
```

Parameter Passing - Arrays of Fixed-Length Elements (contd.)

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

Darr_in_val = {0.0, 0.1, 0.2};
Darr inout_val = {97.0, 98.0, 99,0};
Darr out_val;
Darr_var ret_val;

ret_val = fv->darr_op(in_val, inout_val, out_val);
// ...
// No need to deallocate anything here
```

Parameter Passing - Arrays of Fixed-Length Elements (contd.)

```
typedef double Darr4[4];
interface Foo {
  Darr4 get_darr4(in Darr4 da4);
};
typedef double Darr3[3];
interface Bar {
  Darr3 get_darr3(in Darr3 da3);
};

Foo_var fv = ...; // Get reference
Darr3 in_val = {1,2,3};
Darr3_var ret_val;

ret_val = fv->get_darr4(in_val); // Double disaster
```

Parameter Passing - Variable-length Types

- Variable-length types returned from called functions are dynamically allocated. The calling function must free memory.

```
// IDL
interface Foo
{
     string string_op( in string s_in,
                       inout string s_inout,
                       out string s_out
                      );
};
// C++
class Foo : public virtual CORBA::Object
{
public:
virtual char * string_op( const char * s_in,
                          char * & s_inout,
                          CORBA::String_out s_out
                         ) = 0;
};
```

Parameter Passing - Variable-length Types (contd.)

```
foo_var fv = …;  // Get reference
// Must use dynamic allocation for inout strings
char * inout_val = CORBA::string_dup("inout string");
// No need to initialize out parm nor return value
char * out_val;
char * ret_val;  //don't need to initialize out parm or return value

ret_val = fv->string_op("Hello", inout_val, out_val);
// inout_val may now point to a different string, possibly
// with a different address
// out_val and ret_val point at a dynamically allocated string,
// filled in by the operation
// use returned values here
// We must deallocate inout_val (we allocated it ourselves)
CORBA::string_free(inout_val);
// We must deallocate out strings and return strings because they
// were allocated by the proxy.
CORBA::string_free(out_val);
CORBA::string_free(ret_val);
```

Parameter Passing - Variable-length Types (contd.)

```
CORBA::String_var in_val;
CORBA::String_var inout_val;
char * out_val;
char * ret_val;

// Looming disaster!!!
ret_val = fv->string_op(in_val, inout_val, out_val);
```

Parameter Passing - Variable-length Types (contd.)

```
// IDL
struct Vls {
  long l_mem;
  string s_mem;
};

interface Foo {
  Vls vls_op(in Vls vls_in,
             inout Vls vls_inout,
             out Vls vls_out);
};
// C++
class Foo : public virtual CORBA::Object {
public:
  virtual Vls * vls_op(const Vls & vls_in,
                       Vls & vls_inout,
                       Vls_out vls_out) = 0;
};
```

Parameter Passing - Variable-length Types (contd.)

```
Foo_var fv = ...; // Get reference
Vls in_val; // Note stack allocation
Vls inout_val; // Note stack allocation
Vls * out_val; // Note pointer
Vls * ret_val; // Note pointer
in_val.l_mem = 99; // Initialize in parm
in_val.s_mem = CORBA::string_dup("Hello");
inout_val.l_mem = 5; // Initialize inout parm
inout_val.s_mem = CORBA::string_dup("World");
ret_val = fv->vls_op(in_val, inout_val, out_val);

delete out_val; // Must deallocate out param
delete ret_val; // Must deallocate return value
```

Parameter Passing - Variable-length Arrays

```
// IDL
struct Vls {
  long number;
  string name;
};
typedef Vls Varr[3];
interface Foo {
  Varr varr_op(in Varr varr_in,
               inout Varr varr_inout,
               out Varr varr_out);
};
// C++
struct Vls {..};
typedef Vls Varr[3];
typedef Vls Varr_slice;
class Foo : public virtual CORBA::Object {
public:
  virtual Varr_slice * varr_op(const Varr varr_in,
                               Varr_slice* varr_inout,
                               Varr_out varr_out) = 0;
};
void Varr_free(Varr_slice *);
```

Parameter Passing - Variable-length Arrays (contd.)

```
Foo_var fv = ...; // Get reference
Varr in_val; // Note stack allocation
in_val[0].numer = 0;
in_val[0].name = CORBA::string_dup("Jocelyn");
in_val[1].numer = 1;
in_val[1].name = CORBA::string_dup("Michi");
in_val[2].numer = 2;
in_val[2].name = CORBA::string_dup("Tyson");
Varr inout_val; // Note stack allocation
inout_val[0].numer = 0;
inout_val[0].name = CORBA::string_dup("Anni");
inout_val[1].numer = 1;
inout_val[1].name = CORBA::string_dup("Harry");
inuot_val[2].numer = 2;
inout_val[2].name = CORBA::string_dup("Michi");
Varr_slice * out_val; // Note no initialization
Varr_slice * ret_val; // Note no initialization

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

Varr_free(out_val); // Must free here
Varr_free(ret_val); // Must free here
```

Parameter Passing - Object References

```
// IDL
interface Foo {
  Foo foo_op(
            in Foo ref_in,
            inout Foo ref_inout,
            out Foo ref_out
            );
};
// C++
class Foo: public virtual CORBA::Object {
public:
  virtual Foo_ptr ref_op(
                         Foo_ptr ref_in,
                         Foo_ptr & ref_inout,
                         Foo_out ref_out
                         ) = 0;
};
```

Parameter Passing - Object References (contd.)

```
Foo_var fv = ...; // Get reference
Foo_ptr in_val = ...; // Initialize in param
Foo_ptr inout_val = ...; // Initialize inout param
Foo_ptr out_val; // No initialization neccesary
Foo_ptr ret_val; // No initialization necessary

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

CORBA::release(in_val); //Need to release all references
CORBA::release(inout_val);
CORBA::release(out_val);
CORBA::release(ret_val);
```

Parameter Passing - Summary

- C++ Language Mapping Specification, p. 104

Parameter Passing - _var Types

```
struct Fls {
  long l_mem;
  double d_mem;
};
struct Vls {
  double d_mem;
  string s_mem;
};
interface Foo {
  string op(out Fls fstruct, out Vls vstruct);
};

Foo_var fv = ...; // Get reference
Fls fstruct; // Note _real_ struct
Vls * vstruct; // Note _pointer_ to struct
char * ret_val;
ret_val = fv->op(fstruct, vstruct);
delete vstruct;
CORBA::string_free(ret_val);
```

Parameter Passing - _var Types (contd.)

```
Foo_var fv = ...; // Get reference
Fls_var fstruct; // Don't care if fixed or variable
Vls_var vstruct; // Ditto
CORBA::String_var ret_val;
ret_val = fv->op(fstruct, vstruct);
// Show some return values
cout << "fstruct.d: " << fstruct->d_mem << endl;
cout << "vstruct.d: " << vstruct->d_mem << endl;
cout << "ret_val: " << ret_val << endl;
// Deallocation (if needed) is taken care of by _var types
```

Parameter Passing - _var Types (contd.)

```
// IDL
interface Foo {
string get();
void modify(inout string s);
void put(in string s);
};
// C++
{
Foo_var fv1 = orb->string_to_object(argv[1]);
Foo_var fv2 = orb->string_to_object(argv[2]);
Foo_var fv3 = orb->string_to_object(argv[3]);
// Test fv1, fv2, and fv3 with CORBA::is_nil() here...
CORBA::String_var s;
s = fv1->get(); // Get string
fv2->modify(s); // Change string
fv3->put(s); // Put string
}
// Everything is deallocated here

s = fv1->get(); // Get string
fv2->modify(s.inout()); // Change string
fv3->put(s.in()); // Put string
```

Parameter Passing - _var Types (contd.)

```
// IDL
struct Vls {
  double d_mem;
  string s_mem;
};
interface Foo {
  void in_op(in Vls s);
};
// C++
{
  Foo_var fv = ...; // Get reference
  Vls_var vv = new Vls; // Need to give memory for the _var
  vv->d_mem = 3.14;
  vv->s_mem = CORBA::string_dup("Hello");
  fv->in_op(vv);
} // fv and vv deallocate here

{
  Foo_var fv = ...; // Get reference
  Vls vv;
  vv.d_mem = 3.14;
  vv.s_mem = CORBA::string_dup("Hello");
  fv->in_op(vv);
} // fv deallocates here
```

Parameter Passing - _out Types

```
// IDL
interface Foo {
  void get_name(out string name);
};
// C++
Foo_var fv = ...; // Get reference
char * name;
fv->get_name(name);
cout << "First name: " << name << endl;
fv->get_name(name);  // Bad news!
cout << "Second name: " << name << endl;
CORBA::string_free(name);

Foo_var fv = ...; // Get reference
char * name;
fv->get_name(name);
cout << "First name: " << name << endl;
CORBA::string_free(name);
fv->get_name(name);
cout << "Second name: " << name << endl;
CORBA::string_free(name);
```

Parameter Passing - _out Types (contd.)

```
Foo_var fv = ...; // Get reference
CORBA::String_var name; // Note _var type
fv->get_name(name);
cout << "First name: " << name << endl;
fv->get_name(name);
cout << "Second name: " << name << endl;
// String_var name deallocates when it is destroyed
```

Parameter Passing - _out Types (contd.)

```
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;
  };
  // ...
  char* & _sref;
};



Foo_var fv = ...;

char * name;
fv->get_name(name);

CORBA::String_var name;
fv->get_name(name);
fv->get_name(name);
```

Parameter Passing Caveats

- Passing null pointers
- Passing uninitializes **`in`** and **`inout`** parameters
  - strings
  - unions
- Ignoring variable-length return values

```
interface Foo { string get(); }
fv->get();
```

- Not freeing variable-length **`out`** parameters

Mapping for Exceptions

```
namespace CORBA {
  // ...
  class Exception { // Abstract
  public:
    // ...
  };

  class UserException : public Exception { // Abstract
    // ...
  };

 class SystemException : public Exception { // Abstract
    // ...
  };

  // Concrete system exception classes
  class UNKNOWN : public SystemException { /* ... */ };
  class OBJECT_NOT_EXIST : public SystemException { /* ... */ };
  class NO_MEMORY : public SystemException { /* ... */ };
  class COMM_FAILURE : public SystemException { /* ... */ };
  class TRANSIENT : public SystemException { /* ... */ };
  // etc.
};
```

Mapping for Exceptions (contd.)

```
// In namespace CORBA
class Exception
{
public:
  Exception(const Exception &);
  virtual ~Exception();
  Exception & operator=(const Exception &);
  virtual void _raise() const = 0;
protected:
  Exception();
};

enum CompletionStatus { COMPLETED_YES, COMPLETED_NO, COMPLETED_MAYBE };
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);
  virtual void _raise() const = 0;
  CompletionStatus completed() const;
  void completed(CompletionStatus);
};
```

Mapping for Exceptions (contd.)

```
CCS::Thermostat_var ts = ...;
CCS::TempType new_temp = ...;

try {
  ts->set_nominal(new_temp);
} catch (const CCS::BadTemp &) {
    // New temp out of range
} catch (const CORBA::UserException &) {
    // Some other user exception
} catch(const CORBA::OBJECT_NOT_EXIST &) {
    // Thermostat has been destroyed
} catch (const CORBA::SystemException &) {
    // Some other system exception
} catch (...) {
    // Non-CORBA exception - should never happen
};
```

Mapping for Exceptions (contd.)

```
CCS::Thermostat_var ts = ...;
CCS::TempType new_temp = ...;

try {
  ts->set_nominal(new_temp);
} catch (const CCS::BadTemp &) {
    // New temp out of range
} catch (const CORBA::UserException &) {
    // Some other user exception
} catch(const CORBA::OBJECT_NOT_EXIST &) {
    // Thermostat has been destroyed
} catch (...) {
    // Other system exceptions or non-CORBA exceptions
    // are an SEP (somebody else's problem);
    throw;
};
```

Mapping for User-defined Exceptions

```
exception DidntWork {
  long requested;
  long min_supported;
  long max_supported;
  string error_msg;
};

class DidntWork : public CORBA::UserException {
public:
  CORBA::Long requested;
  CORBA::Long min_supported;
  CORBA::Long max_supported;
  CORBA::String_mgr error_msg;
  DidntWork();
  DidntWork(CORBA::Long requested, CORBA::Long min_supported,
CORBA::Long max_supported, CORBA::String_mgr error_msg;);
  DidntWork(const DidntWork &);
  DidntWork & operator=(const DidntWork &);
};
```

Exceptions and `out` Parameters

- After an exception during the operation we cannot use the return value nor **`out`** parameters

```
CORBA::String_var name =
CORBA::string_dup("Hello");
// ...
try {
  vf->get_name(name);
} catch (const CORBA::SystemException &) {
cout << name << endl; // Disaster!!!
};
```