# Specification of Addition to CORBA IDL C++ Mapping

**Dyshlevoi K.V., Solovskaya L.B.**

Moscow

1998

# 1.0 Bases

The standard CORBA IDL to C++ mapping has some shortcomings in sense of variable initialization, parameter passing, reliability of memory management.

For example, so-called _var types are provided for all aggregate types. They help to simplify managing pointers to the corresponded types and used for user's convenience. But usage of _var types simultaneously with direct types and possibility of implicit casting between them do not protect user from incorrect assignments. That leads to possibility of incorrect memory management and causes a lot of user's efforts to write correct application.

Also, argument passing rules represent the most difficult and badly understandable part of mapping. They are based on the unnatural distinguishing between fixed-length types and variable-length ones.

Union type is one of the unreliable types in standard mapping. There is no default value of union type, so attempt to get value from the union, which was not explicitly initialized, leads to undefined behavior.

To avoid these shortcomings and many others we propose additional part (superstructure) to the standard CORBA IDL to C++ mapping. It was developed in the style of pure C++ without care of coherence with IDL C mapping, which is a weak point in standard IDL to C++ mapping.

Application using that superstructure can run not only concrete ORB, supporting and optimizing that part of mapping, but any CORBA 2.0 compliant C++ object broker. So, portability of such applications is provided.

# 2.0 Common Use

## 2.1 T_cvr Types

For all complex types (structures, unions, sequences, arrays, strings, object references and Any) special class-mediators, called covers, are defined.

As well as T_var type, cover can be called a "smart" pointer to the associate type T. But unlike T_var type, cover provides more reliable mechanism of memory management on the one hand, and a set of explicit operations for user's optimization on the other.

The main idea of a cover approach is that user manipulates data only by means of such covers. In particular, it allows to simplify and increase reliability of arguments passing and getting results.

There are two mode of covers usage. The first mode is simple and reliable, and another one is effective, but a little bit more complicated. That more flexible mode of covers usage just needs good knowing of C++ memory management. User can choose the scheme of work with data according to needs.

The name of cover class is composed by adding the suffix “_cvr” to the original type’s name. While adding new types into existing scope is impossible in general, all cover types are declared in the file scope and global names are used. So, all components of complex scoped type’s name are separated by “_” (underscore) instead of “::”. In that sense the names mapping is the only thing that resembles IDL C mapping.

The same way as in IDL C mapping, the use of underscores to replace the “::” separators can lead to ambiguity if the OMG IDL specification contains identifiers with underscores in them. Due to such ambiguities, it is advisable to avoid the indiscriminate use of underscores in identifiers.

Cover is a class with a default constructor, a set of copy constructors, assignment operators, and a destructor.

The general form of T_cvr class is as following.

```
// C++
class T_cvr {
public:
T_cvr ();
T_cvr (const T*);
T_cvr (const T_cvr&);

~T_cvr ();

T_cvr &operator= (const T_cvr&);
T_cvr &operator= (const T*);

T* copy () const;

...
};
```

The default constructor creates a T_cvr containing a NULL pointer. But any legal operation on the cover will check that pointer and, if it is still NULL, will create data initialized by default. In the most case, the default constructor of corresponded T type is used. But if CORBA mapping does not provide correct default initialization (as in the case of union type) we propose special mechanism to support such situations (see 2.3).

The T* constructor deep-copies data pointed to by the parameter. NULL pointer as parameter is available. In this case cover will contain NULL pointer too.

The T_cvr& constructor deep-copies data contained in argument.

Destructor releases memory pointed to by the cover.

While the assignment is the most frequently used operation, the normal assignment operator provides reliable scheme of memory management (but not very effective). It deep-copies data from the given cover (using its old storage). In the result both covers will point to different storage with equal content. If parameter cover contains NULL pointer, this assignment operator is equivalent to release() method.

The T* assignment operator results in deep-copying data too.

Method copy() allocates dynamic memory, deep-copies data pointed to by the cover and returns pointer to that memory. The data in the cover is not changed.

It is important to note, that in contrast to T_var type there is no conversion operator from the cover into the T*. It is impossible to define reliable semantics of such conversion (copy or non-copy), since user can suggest different memory management in the different parts of application.

```
// IDL
struct S { long age; string name;};

// C++
S* tp = new S;
S* sp;
S_cvr tc;
tc = tp;  // deep-copy
sp = tc;  // error
```

Cover's definition also has is_nil() method.

```
// C++
class T_cvr {
public:
...
Boolean is_nil ();
};
```

In the examples presented in this paper, all definitions encapsulated in standard CORBA module are referenced without explicit qualification for simplicity. I.e. in the above example we mean CORBA::Boolean type.

This method returns TRUE if corresponded cover contains NULL pointer. Otherwise it returns FALSE.

For cover created by means of default constructor is_nil() always returns TRUE. After release() operation is_nil() also returns TRUE.

To provide user friendly and reliable environment, methods based on the covers (e.g. public methods of object reference, union cover's accessor and modifier functions) always return valid covers. If some incorrect situation happens they return covers containing NULL pointer and raise exception. To check the return cover content method is_nil() is used. It is important, there is no another way to control it, since after attempt to use cover with NULL pointer, it will be initialized by default value.

For concrete IDL complex types appropriate covers overload some additional operators to provided access to the field's data. For struct type operator "->" is overloaded by cover. In turn, for sequence, array and string types cover's operator [] returns appropriate element of array or sequence. Full definitions of cover and it's concretizations are presented in Appendix 1.

To manipulate memory in more effective way and to simulate cover's filling by non-dynamical storage special set of explicit cover's operation can be used (see section 3.1).

## 2.2 T_cvr Type for Object References

Interface type in OMG IDL denotes object reference. According to CORBA mapping an interface named A maps to two C++ types - A_ptr and A_var. Variable of A_var type automatically releases its object reference when it is deallocated or assigned another object reference. A_ptr type provides a more primitive semantics, which is similar to a C++ pointer. It is possible to mix data of types A_var and A_ptr without any explicit operations or casts in the compliant application. That can lead to incorrect memory usage and difficultly detectable errors.

Covers addition to CORBA mapping provides only one type A_cvr for IDL object reference type. User should not use A_ptr or A_var types simultaneously with A_cvr type.

On the one hand cover type for object reference provides reliable memory management, on the other it plays the role of a mediator. In the case of CORBA 2.0 compliant object broker, which does not support covers mapping, that mediator forwards invocation to the real object reference contained in the cover and transforms parameters of complex types (T_cvr <-> T) before and after real method call. Parameters of primitive and enumerated type are passed without changing. These cover's methods are invoked via "." (not "->").

```
// IDL
struct S { long age; string name;};
interface A {
        void op (in long p1, out S p2);
};

// C++
class A_cvr {
public:
A_cvr ();
A_cvr (const A_cvr &another_cvr);
A_cvr (const ..._cvr &another_obj);

~A_cvr ();

A_cvr &operator= (const A_cvr &another_cvr);
A_cvr &operator= (const ..._cvr &another_obj);

A_ptr operator->();
const A_ptr operator->() const;
...
void op (Long p1, S_cvr p2) {
        _ptr->op (p1, *(p2.look()));
}
...
protected:
        A_ptr _ptr;
};
```

Default constructor is resulted in the cover contained NULL object reference.

A_cvr constructor creates the cover with object reference duplicated from the given argument.

..._cvr constructor is intended for initialization by means of any covers contained object references. At first, implicit narrowing operation to check run-time type of argument is performed. If argument can be casted to the needed object reference type of current cover, given object reference is duplicated. Otherwise, the cover is initialized by NULL pointer.

Destructor releases object reference contained in the cover.

Assignment operators have the same semantics as corresponded constructors. So, assignment operation between two covers containing object references always correct and to check the result of such assignment cover's method is_nil() should be used.

For example,

```
// C++
A_cvr ac;
B_cv bc;
...
ac = bc;
if (!(ac.is_nil())) {
... use correct ac in some way
}
```

If content of cover bc could not be narrowed to needed type of cover ac content, cover ac will contain NULL pointer after such assignment.

Operator “->” is overloaded, so real methods of object reference contained in the cover are available too. But usage of this possibility is not recommended because of disturbance of covers approach and unreliability. Operator “->” have to be used only for particular ORB-dependent methods.

Public methods corresponded to the methods of the real object reference performs needed parameter transformation and forwards invocation to the real method. In the previous example, exactly content of struct cover is passed as INOUT argument to the real object reference method. And value will be returned in the memory, managed by the cover. Arguments of primitive types are passed without any transformation.

To get pointer to the current object reference in the way of the cover approach method _this_cvr() is used.

```
// C++
Object_cvr _this_cvr();
```

It returns covers containing pointer to the current object reference. To present all that possibilities.

## 2.3 T_cvr Type for Union

***accessor and modifier functions***

Additionally to common cover’s methods, cover overloads data access methods (“->” for structure, [] - for sequence, array and string). Also union cover provides additional accessor and modifier functions for the union members and discriminator.

Appropriate pair of “set” and “get” methods are based on the covers for complex types and invoked via “.”. Environment parameter is added to provide possibility of passing exception information, if an implementation does not support real C++ exceptions. For primitive and enumerated types these methods have the same signatures as standard ones, but also extended by Environment parameter if needed. In general, these methods just invoke corresponded method of real union and perform parameters transformation (T_cvr <-> T).

Also, cover's "get" method is more reliable than standard one because of discriminator checking. If current discriminator value does not coincide with the needed discriminator value of the field, "get" method of which is invoked, empty cover (with NULL pointer) of appropriate type will be returned and system exception BAD_OPERATION will be raised. If the type of field is primitive or enum, 0 is returned. To control such situation cover method is_nill() should be used to check returned cover. E.g.

```
// IDL
struct S {long l; string str;};
union U switch (octet) {
        case 1: long l;
        default: S s;
};

// C++
class U_cvr {
...
void _d(Octet);
Octet _d ();

void l (Long);
Long l ();

void s (const S_cvr&);
S_res s ();
};

...

U_cvr uc;
S_cvr sc;

...
sc = uc.s();
if (!(sc.is_nill())) {
... use correct sc in some way
}
```

In the common way of the cover approach these "get" and "set" methods perform deep-copying of the union member data in contrast to CORBA mapping. So, one is recommended to save union member data in appropriate cover and then manipulate own copy. If read-write access to the direct union storage is needed, such situation can be simulated by means of union methods provided by CORBA mapping, that are also available via "->". But usage of them should be restricted, since they do not provide reliable memory management.

```
// C++
sc = uc.s();                          // copy
cout << (uc->s()).str << endl;   // read-write access
(uc->s()).l = 1;                     // read-write access
```

***initialization***

According to standard IDL C++ mapping, the default union constructor performs no application-visible initialization of the union. So, it is an error to access the union before setting it.

Union cover provides initialization of containing union. It's first field is initialized by means of default constructor. In the case of primitive and enumerated types value 0 is used. Discriminator is initialized by any value corresponded to the first field.

```
// C++
U_cvr uc;
S_cvr sc;
Long x;

switch (uc._d()) {
        case 1:
            x = uc.l();
            break;
        default:
            sc = uc.s();
            break;
}
```

So, union cover can be used without explicit initialization. It can be extremely convenient, if union is passed as an argument into the method.

```
// IDL
union U switch (long) {
        case 1: octet data;
        default: string str;
};
```

```
interface A {
        void op (inout U param);
};

// C++
A_cvr a = ...;
U_cvr uc;

a.op (uc);
```

## 2.4 Strings

Supporting of bounded strings.

## 2.5 Cover for the Any Type

For IDL Any type appropriate cover Any_cvr is provided and for all IDL types pair of operators for insertion into and extraction from Any_cvr is supplied. “Shift-assign” operators are given for direct primitive types and covers of complex type:

```
// C++
void operator<<= (Any_cvr &target, const T_cvr &source);
Boolean operator>>= (const Any_cvr &source, T_cvr &target);
```

Also, it is possible to work with covers types and Any itself. Appropriate “shift-assign” operators are overloaded for T_cvr types:

```
// C++
void operator<<= (Any &target, const T_cvr &source);
Boolean operator>>= (const Any &source, T_cvr &target);
```

To use Any and Any_cvr operations for data from complex type members, we recommend fill the corresponded cover and call appropriate “shift-assign” operators for covers.

```
// IDL
struct Point {long x; long y; long z;};
struct S {Point f1; boolean f2;};

// C++
Any a;
Any_cvr ac;
S_cvr sc;
Point_cvr pc;
```

```
pc.point (sc->f1);
ac <<= pc;
...
a >>= pc;
```

In this example cover pc points to the first member of the struct contained in the cover sc. So, exactly that data of the struct field will be inserted into the Any cover ac. And after extraction from Any object a into the cover pc, struct in the cover sc will contain that given data.

## 2.6 Argument Passing Considerations

Argument passing considerations is one of CORBA mapping weakness. For primitive types and enumerations the modes are straightforward. But rules for aggregate types are complicated by the question of when and how parameter memory is allocated and deallocated and depends on variable-length or fixed-length type is.

Covers mapping argument passing considerations for primitive and enum types are the same as standard one. But rules for covers of complex types are greatly simplified and do not depend on whether type has fixed or variable size.

**TABLE 1. Argument Passing Considerations**

| **Data type** | **IN** | **INOUT** | **OUT** | **RETURN** |
|---|---|---|---|---|
| primitive type (P) | P | P & | P & | P |
| complex type (T) | const T_cvr & | T_cvr & | T_cvr & | T_res |

Since this mapping deals with only primitive and managed T_cvr types, there is no problem with memory allocation and deallocation. Each cover contains it’s own memory. T_res type is a special managed type which serves for data passing between covers with minimal expenses. It helps to optimize returning of result from the method and is used only in operation signatures.

As well as T_cvr, object of T_res type contains pointer to appropriate type and provides some semantics of managing this pointer. Unlike T_cvr T_res type does not overload operators for data members manipulations (“->”, []). The T_res type is intended to be used only in operation signatures and nowhere else.

```
// IDL
struct Point {long x; long y; long z;};
struct S { Point p; string name;};
interface A {
        S op ();
};
```

```
// C++
class A_cvr {
...
S_res op ();
};

...
S s;
S_cvr sc;
Point_cvr pc;
A_cvr ac = ...

sc = ac.op ();  // O.K.
s = ac.op ();    // error
cout << (ac.op ())->name << endl;  // error
```

On the client side result of such methods have to be assigned into the cover of appropriate type. It is the only possible way to get result from the method, since other variants do not provide needed reliability. In the implementation of method that value can be returned as corresponded cover or direct type.

According to standard CORBA IDL C++ mapping the client could not pass NULL pointer as IN or INOUT parameter and vice versa the caller could not return NULL pointer as OUT parameter or RETURN value. Such parameters always should be initialized. Implicit initialization of covers helps to avoids such inconveniences.

```
// IDL
struct S { long age; string name;};
interface A {
        S op (in S p1, inout S p2,  out S p3);
};

// C++
class A_cvr {
...
S_res op (const S_cvr&, S_cvr&, S_cvr&);
};

// client side
A_cvr ac;
S_cvr s1, s2, s3, res;

ac = ...
res = ac.op (s1, s2, s3);  // legal argument passing
```

To increase effectiveness of argument passing and to simulate passing non-dynamical memory special set of explicit cover's operation can be used (see section 3.1).

# 3.0 Additional Cover's Features

## 3.1 Explicit operations to manage memory inside covers

Cover operations listed in this section provide user possibility to work with memory, contained in the cover, in effective way. They have more flexible semantics than cover's constructors and assignment operators, and so cause good understanding of it. But user whose is familiar to C++ memory management will find these operations useful and quite natural.

```
// C++
void assume (T*);
```

The cover assumes memory given as argument. It is worth to note, that such memory must be dynamic. No memory of others types is available, since the cover will automatically release it.

```
// C++
void point (const T*);
```

This operation of cover just saves pointer to data (of any memory type). The pointer assigned in this way, will be never released by cover and user should take responsibility of it. It is important to note, the data given to the operation can be changed further independently of user. So, one should be careful when passes data allocated in const memory.

```
// C++
T* look();
```

By means of this operation user have opportunity to "look" at the cover's data via own pointer. The cover continues to manage this memory and user should not release it. In the most general case, this operation returns not NULL pointer. If cover is empty it will be automatically initialized and pointer to the created data will be returned. Exception consists in the following case. Method look() of String_cvr and covers of object reference returns NULL if cover is empty, since String and object reference are the pointers themselves.

```
// C++
S* sp;
S_cvr sc;
sp = sc.look();
```

```
...
delete sp; // error
```

In the example, pointer sp points to the data contained in the cover sc. This cover is initialized automatically when method look() is invoked.

In general outline, each cover contains it's own memory and manages it itself. But sometimes it can be useful to have several covers that point to one and the same data. Such situation can be simulated by means of look() and point() cover's operations.

```
// C++
T_cvr ac, bc;
bc.point(ac.look());
```

Now covers ac and bc point to the same data, which was created by means of cover's initializing mechanism. Ambiguities in memory holding do not appear since point() operation do not provide rights of memory management. Common memory will be released only once by cover ac.

```
// C++
T* give_data();
```

Unlike look() operation, this method "gives up" the memory contained in the cover. In the result of this operation the cover becomes empty and not releases given up memory. So, user should manage that memory himself. If the cover is already empty, new memory will be allocated and pointer to it will be returned. NULL pointer will be returned only for strings and object references.

```
// C++
S* sp;
S_cvr sc;
sp = sc.give_data();
...
delete sp; // O.K.
```

```
// C++
void release();
```

In the result of release operation the cover will be in the same state as after creation by means of default constructor. It does not mean removing of the cover itself.

## 3.2 Strings and Object References as fields of complex types

According to standard CORBA mapping, for a string or object reference member of complex types, the name of the C++ member's type is not specified. So, to support work with data contained in the fields of complex types in the way of covers approach special operations are used. It is necessary, since covers mapping suggests usage of appropriate covers instead of direct string and object reference types everywhere in compliant applications.

All the following operations have the same semantics as correspondent cover's methods. They used to manipulate data of struct and union fields, elements of arrays and sequences of string and object reference types.

```
// C++
void _assign (const T &source, T &target);
```

A set of _assign () operations for both strings and object references provides possibility to perform various assignments between members of aggregate type, corresponded covers and direct types (char*, A_ptrs). These operations are intended to be used instead of traditional assignment operator "=" and as well suggest reliable deep-copy semantics.

```
// IDL
struct Coord {long x; long y; long z;};
struct S {Point p; string str;};

// C++
Coord_cvr pc;
S_cvr sc;
String_cvr name;
String_cvr strc = "coordinate";
const char * const_str = "constant_string";
char * str;

sc->p = pc;                 // O.K.
sc->str = strc;             // error
_assign(strc, sc->str);   // O.K.
_assign(const_str, sc->str);  // O.K.
...
_assign (sc->str, name); // O.K.
name = sc->str;           // O.K.
pc = sc->p;                 // copy
pc.point (sc->p);          // non-copy
_assign (sc->str, str);     // O.K.
```

If for struct member's type corresponded cover is provided (but not string and object reference), it is possible to use cover's assignment operator and cover's additional methods to work with data from the aggregate type. In the example cover pc can be assigned by struct's member data in various ways. The simplest mode is to use assignment operator "=", which performs data deep-copy. If you need more flexible memory management, e.g. you want struct cover to be contained of exactly struct's member memory, cover's method point() have to be applied. But it is important to remember, that after such assignment memory in the cover and struct's member will change simultaneously.

The following example demonstrates the work with object references:

```
// IDL
interface A;
struct SA {long number; A obj};

// C++
SA_cvr sac;
A_cvr ac, bc;

sac->number = 1;
sac->obj = ac;             // error
_assign (ac, sac->obj);  // O.K.
...
_assign (sac->obj, bc);  // O.K.
bc = sac->obj;             // error
```

Also, the assignment between aggregate type's members of string or object reference type is performed by means of operation _assign(). Assignment operator "=" is used for primitive and enumerated types as well as for complex types (with exception of strings and object references).

```
// C++
S_cvr sc1, sc2;

...
_assign (sc1->str, sc2->str);   // O.K.
sc2->str = sc1->str;             // error
pc.point(sc2->p);
pc = sc1->p;
sc2->p = sc1->p;
```

For both strings and object references encapsulated inside aggregate type special operator _is_nil () is provided. It checks whether given struct's member of string and object reference type is NULL or not. Operation _release (), which is also available for both strings and object references, releases given memory.

```
// C++
char *_copy (char *source);
void _assume (char *source,  char * target);
```

```
char *_look (char *source);
char *_give_data (char * source);
```

This set of operations provides user possibility to manipulate strings incapsulated in the aggregate types. They have the same semantics as correspondent cover's methods.

```
// C++
_assume (const_str, sc->str);  // copy
str = _look (sc->str);             // no copy
```

## 3.3 Server Side Mapping

Server writer is supplied by special templated class, which serves as mediator between server object based on covers and standard ORB methods. It's primary goal is transformation between parameters received by broker according to standard mapping and corresponded covers used by implementation and vise verse. In general, such mediator plays the role inverse to the cover of object reference on the client side.

```
// IDL
struct S { long age; string name;};
interface A {
        void op (in long p1, out S p2);
};

// C++
class A_real_impl {
public:
void op (Long a1, S_cvr a2);
};

template <class Impl>
class A_cvr_skel {
public:
  Impl *ptr;
  A_cvr_skel () : ptr(NULL) {}
  A_cvr_skel (Impl* _ptr) : ptr(_ptr) {}

  // public methods
  inline void op (Long p1, S*& p2) {
        // parameters transformation before invocation
        S_cvr _param2;
        _param2.point (&p2);

        ptr->op (p1, _param2);
```

```
        // parameters transformation after invocation
        if (_param2.look() != &p2)
            p2 = _param2;
};
```

Argument Impl of this template is intended to be a type of real server implementation written using covers approach (e.g. class A_real_impl). Default constructor initializes data member ptr by NULL. Impl* constructor saves given pointer to server implementation. It is worth to note, data member ptr is a public one. So, user can change this pointer after registration too.

Names of public methods of this class are the same as method names of corresponded implementation class. Signatures of these methods satisfy the requirements of standard mapping. Implementations just transform parameters into covers and forward request to the appropriate real implementation method.

Covers mechanisms, used in these methods, allow correct work with mediator initialized by NULL pointer. Empty covers will be returned as OUT and RETURN parameters and system exception will be raised. In the case of primitive or enumerated types, such parameters will be initialized by 0 value.

It is important to note, that caller can return memory of any type as OUT, INOUT and RETURN parameter. For example, static memory, saved in the cover by means of point() method can be returned. In the case of local invocation client will receive exactly that memory in a very effective way. Of course, server writer have to be aware of this situation.

```
// server side
// static memory example
static S s;

void op (Long in_param, S_cvr out_param) {
...
out_param.point(&s);
};

// dynamic memory example
void op (Long in_param, S_cvr out_param) {
S* s = new S;
...
out_param.point(s);
};
```

If a cover contained dynamic memory is passed back by caller it will be always worked out correctly by covers mechanisms. So, it is most often and reliable way to return data.

According to CORBA mapping ORB implementation provides methods for registration of server objects, based on A_ptr types. To use such standard methods simultaneously with cover approach we recommend server side writer to use cover assume() method.

For example,

```
// C++
...
typedef A_cvr_skel <A_real_impl> A_cvr_impl;
A_cvr_impl * as = new A_cvr_impl(new A_real_impl);
A_cvr ac;
A_ptr ap = registrate_implementation(as);
ac.assume (ap);
```

A_real_impl is a class providing implementation of methods of IDL interface A, based on covers approach. To registrate implementation in the ORB we use class A_cvr_impl, which is a given templated class initialized by A_real_impl type. So, object as is registered in the ORB and plays the role of mediator between standard ORB and implementation using covers. Finally, cover ac contains object reference corresponded given implementation. Note, that in the result of assume() operation cover assumes memory and will delete it when scope is out.

# Appendix 1. C++ Definitions for Addition to CORBA Mapping

### 1. T_cvr Type Definition

```
// C++
class T_cvr {
public:
T_cvr ();
T_cvr (const T*);
T_cvr (const T_cvr&);
T_cvr (const T_res&);

~T_cvr ();

T_cvr &operator= (const T_cvr&);
T_cvr &operator= (const T_res&);
T_cvr &operator= (const T*);

T* copy () const;

// other operations
void assume (T*);
void point (const T*);
```

```
T* look();
T* give_data();

Boolean is_nil ();
...
};
```

## 2. T_res Type Definition

```
// C++
class T_res {
public:
T_res ();
T_res (const T*);
T_res (const T_res&);
T_res (const T_cvr&);

~T_res ();

T_res &operator= (const T_res&);
T_res &operator= (const T_cvr&);
T_res &operator= (const T*);
...
};
```

## 3. T_cvr Concretizations

### 3.1. T_cvr for Structured, Union, Sequence and Any Types

Covers for structured, union, sequence and Any types provides some operations additional to the common set of cover’s operation.

```
// C++
class T_cvr {
...
T_cvr (const T&);
...
T_cvr &operator= (const T&);


operator T () const;

T *operator->();
};
```

The T& constructor and assignment operator with deep-copy semantics are added. Operator “->” is overloaded and used for structure fields access.

Also see additional operations for the T_res of the appropriate T type.

### 3.2. T_cvr for Array Type

For array type additional operation is operator [].

```
// C++
class T_cvr {
...
T_slice &operator[] (unsigned long index);
};
```

T_slice is a slice type for array T, provided by standard IDL to C++ mapping. It is an array with all the dimensions of the original specified except the first one. A T_cvr type for an array overloads operator [], which returns appropriate element of array.

To manipulate array's elements of string and object reference type special set of operations is used (see 3.2).

### 3.3. T_cvr for Sequence Type

Refinements of T_cvr type common definition consist in addition of [] operator.

```
// C++
class T_cvr {
...
T_elem &operator[] (unsigned long index);
};
```

T_elem is a type of sequence's element. Overloaded operator [] forwards request to the operator [] of contained sequence and returns the item of the given index.

Accessor functions length() and maximum() described in the CORBA mapping are available by means of operator "->".

Additional memory management functions allocbuf() and freebuf(), proposed by CORBA mapping for dynamic memory allocation, are not overloaded for covers. Since covers provide equivalent mechanisms themselves. Dynamic memory in empty cover for sequence will be allocated automatically under attempt to access it (for example, in the cover's method look()). In turn, method release() frees dynamically allocated memory.

As well as for array's members work with sequence's elements of string and object reference type should be performed by means of a special set of operations (see 3.3).

### 3.4. T_cvr for String Type

String_cvr class additionally to common cover's operations provides [] method and it's const version:

```
// C++
class String_cvr {
...
Char &operator[] (unsigned long index);
Char operator[] (unsigned long index) const;
};
```

They return string's element corresponding to the given index.

### 3.5. T_cvr for Any Type

```
// C++
class Any_cvr {
public:
// constructors
Any_cvr ();
Any_cvr (const Any *ptr);
Any_cvr (const Any_cvr &another_cvr);
Any_cvr (const Any_res &another_cvr);
Any_cvr (const Any &s);
Any_cvr (const Any_cvr &another_cvr);

// assignment operation
Any_cvr &operator= (const Any_cvr &another_cvr);

// "shift" operators
...
};
```

## 4. T_res Concretizations

For structured, union, sequence and Any types additional T& constructor and assignment operator with deep-copy semantics are provided.

```
// C++
class T_cvr {
...
T_cvr (const T&);
...
T_cvr &operator= (const T&);
};
```

Operator [] of T_res type for array, sequence, string is not overloaded, since semantics of res type does not suppose data manipulation.

## 5. T_cvr Type for Pseudo Objects

To provide work with Pseudo Objects in the style of cover approach the following covers declarations and methods are used.

- ORB

```
// C++
class ORB_cvr {
...
}

ORB_cvr _orb_cvr();
```

The last operation returns cover containing appropriate pointer (ORB_ptr) to the object broker.

- Context

```
// C++
class Context_cvr {
...
}
```

This cover contains pointer to the corresponded Context data and is passed as additional parameter in object reference methods calls.

- Environment

```
// C++
class Environment_cvr {
...
}
```