C++ Language Mapping Specification
Отображение IDL на C++

CORBA Assistant

Отображение IDL на C++ (C++ Language Mapping Specification)

Этот документ является переводом OMG C++ Language Mapping Specification [99-07-41]. Этот документ был взят с сайта OMG. Данный документ может перепечатывается или использоваться любым другим способом со следующими ограничениями:

- Должна быть указана ссылка на оригинальный документ OMG
- Должна быть указана ссылка на данный перевод

Об ошибках и неточностях в данном документе просим сообщать на akurgan@yandex.ru.

Перевод: *Виктория Плеханова, Алексей Курган*

Эта статья опубликована на сайте CORBA Assistanthttp://corba.kubsu.ru/

1.1. Предварительная информация

1.1.1. Обзор

1.1.1.1. Ключевые решения

Дизайн отображения на C++ выработан с учетом множества требований, включая требования разумной производительности, мобильности, эффективности, и применимости для реализации отображения с OMG IDL на C++. Также в этом разделе выделено несколько других требований.

1.1.1.2. Совместимость

В данное отображении на C++ есть попытка избежать ограничений свободы реализации ORB. Для каждой конструкции OMG IDL и CORBA, отображение на C++ объясняет синтаксис и семантику использования конструкций C++. Клиент или серверная программа соответствует данному отображению (CORBA C++ совместимость) если используются конструкции описанные в это документе. Реализация соответствует данному отображению если она правильно выполняет любые клиентские или серверные программы. Поэтому соответсвующий данной спецификации клиент или серверная программа портируема на все соответствующие данной спецификации реализации.

1.1.1.3. Требования к реализации C++

Отображение, предлагаемое в этом документе предполагает, что целевая среда C++ поддерживает все возможности, описанные в *The Annotated C++ Reference Manual (ARM)* Элисом и Страуструпом, и принятые комитетом стандартизации ANSI/ISO C++, включая обработку исключений. Кроме того, предполагается, что среда C++ поддерживает конструкцию **namespace**, но и на компиляторах не поддерживающих **namespace** обеспечивается работоспособность.

1.1.1.4. Совместимость размещения данных с C

Некоторые поставщики ORB настоятельно требуют, чтобы отображение на C++ могло непосредственно работать с отображением CORBA C. Такое отображение дает гарантию совместимости между C и C++ отображениями, но описанное в этом документе отображение не обеспечивает этого. Потому, что в дополнение к обеспечению лучшей способности взаимодействия и мобильности, стиль запроса C++ решает проблемы управления памятью, которые замечены программистами использующими стиль C. Поэтому, OMG принял C++ стиль для отображения с OMG IDL. Однако, чтобы предоставить продолжение для более ранних приложений, реализация могла бы поддерживать стиль C как опцию. Если реализация поддерживает оба стиля, то рекомендуется сводить на нет стиль C.

Обратите внимание, что глава отображения на язык C была изменена, для достижения совместимости между C и C++ отображениями.

1.1.1.5. Нет описаниям реализации

Это отображение не содержит описание реализации. Этот документ избегает деталей, которые ограничивали бы реализацию, но все же позволяет клиентам быть полностью совместимыми по исходным текстам с любой совместимой реализацией. Некоторые примеры показывают возможные реализации, но они не обязательно должны реализовываться так.

1.1.2. Область видимости имен

Область видимости имен в OMG IDL определена через области видимости C++:

- Модуль OMG IDL отображаются в C++ namespace
- Интерфейсы OMG IDL отображаются в классы C++
- Все конструкции OMG IDL заключенные в область видимости интерфейса доступны через область видимости имен C++. Например, если тип **mode** определен в интерфейсе **printer**, то тип указывается как **printer::mode**.

Такие отображения предоставляют механизмы в OMG IDL и C++ для использования области видимости имен. Например:

```
//IDL

module M
{
     struct E{
          long L;
     };
};
```

отображается в:

```
//C++

namespace M
{
     struct E{
          Long L;
     };
}
```

и **E** может указываться вне **M** как **M::E**. В качестве альтернативы, оператор C++ **using** для namespace **M**, может быть использован для того, чтобы указывать **E** просто как E.

```
//C++

using namespace M;
E e;
e.L = 3;
```

В качестве другой альтернативы, можно использовать оператор **using** только для **M::E**.

```
//C++

using M::E;
E e;
e.L = 3;
```

Чтобы избежать проблем с компиляцией C++, каждое использование в OMG IDL ключевого слова C++ как идентификатора, отображается в тоже самое имя с префиксом "_cxx_". Например, интерфейс IDL, названный try, будет назван "_cxx_try", при отображении в C++. Для устойчивости, это правило также применимо к идентификаторам, которые получены от идентификаторов IDL. Например, интерфейс IDL "try" произведет имена "cxx_try_var" и "cxx_try_ptr", то есть компилятор IDL ведет себя так, как будто интерфейс был названн "cxx_try", и затем применяет нормальные правила отображения.

1.1.3. Требования к размеру типа языка C++

Размеры типов C++ используют представление OMG IDL типов с зависимостью от реализации. То есть на это отображение не накладывается никаких требований относительно **sizeof(T)** для чего либо, кроме основных типов.

1.1.4. Модуль CORBA

Отображение основывается на некоторых предопределенных типах, классах, и функциях, которые логически определены в модуле называемом **CORBA**. Этот модуль автоматически доступен из единицы компиляции C++, которая включает заголовочный файл из спецификации OMG IDL. В примерах, представленных в этом документе, определения CORBA упоминаются без явной квалификации, для большей простоты. Практически всегда область видимости имен или оператор C++ **using** для пространства имен CORBA требуются в исходных кодах приложения.

1.2. Отображение для Модулей

Модуль определяет область видимости, и отображается в C++ **namespace** с тем же самым именем:

```
//IDL

module M
{
     //определения
};


//C++

namespace M
{
     //определения
}
```

Поскольку пространства имен (namespace) были только недавно добавлены в язык C++, поэтому только небольшое количество компиляторов поддерживают их. И поэтому в этом документе также содержаться отображения для модулей OMG IDL, которые не требуют пространств имен C++.

1.3. Отображение для Интерфейсов

Интерфейс отображается в класс C++, который содержит определения типов, констант, операций, и исключений интерфейса в разделе public.

CORBA C++ совместимая программа не может:

- создать или хранить экземпляр интерфейса класса, или
- использовать указатель (**A***) или ссылку (**A&**) к классу интерфейса.

Причина для этих ограничений заключается в том, чтобы можно было позволить широкое разнообразие реализаций. Например, классы интерфейса не могут быть осуществлены как абстрактные базовые классы, если программам позволить создавать или хранить их экземпляры. В некотором смысле, произведенный класс - подобен пространству имен, который нельзя вводить через оператор **using**. Следующий пример показывает поведение отображения интерфейса:

```
//IDL

interface A
{
     struct S { short field; };
};


//C++

//Правильное использование
A::S s; //Определение переменной
s.field = 3; //Доступ к полю


//Неправильные использования
//нельзя объявлять экземпляр интерфейсного класса...

A a;
//... нельзя объявлять указатель на интерфейсный класс...

A *p;
//... нельзя объявлять ссылку на интерфейсный класс.

void f(A& ptr);
```

1.3.1. Типы Объектных Ссылок

Использование типа интерфейса в OMG IDL обозначает объектную ссылку. Объектная ссылка отображаетссся на два C++ типа. Для интерфейса **A**, эти типы будут называться **A_var** и **A_ptr**. По историческим причинам, тип **ARef** определен как синоним для **A_ptr**, но использование имен **Ref** не переносимо и мотому неприемлемо. Для облегчения программирования на основе шаблонов, **typedef** для **A_ptr** и **A_var** предоставляются в классе интерфейса. **typedef** для **A_ptr** назван **A::_ptr_type**, а **typedef** для **A_var** назван **A::_var_type**.

Операция может быть выполнена на объекте, используя стрелку ("->") на объектной ссылке. Например, если интерфейс определяет операцию **op** без параметров, и **obj** - это ссылка на тип интерфейса, то запрос может быть описан как **obj->op()**. Оператор стрелка используется для вызова операций как на типе **_ptr**, так и на типе **_var** объектной ссылки.

Клиентский код часто будет использовать тип переменной объектной ссылки **(A_var)**, потому что переменная автоматически освобождает ее объектную ссылку, когда освобождается сама переменная или когда назначается новая объектная ссылка. Тип указателя **(A_ptr)** предоставляет еще примитивную объектную ссылку, которая имеет семантику подобную указателю C++. Дествительно, реализация может выбрать определение **A_ptr** как **A***, но это не является необходимостью. В отличиие от указателей C++, преобразование к **void***, арифметические операции, и операции отношения, включая проверку на равенство не совместимы и противопоказаны. Совместимая реализация не нуждается в обнаружении этих неправильных использований, поэтому это требование обнаружения не практично.

Для многих операций, смешивание данных типа **A_var** и **A_ptr** возможно без любых явных операций или приведений. Однако, требуется быть внимательным из-за неявного освобождения, когда освобождается переменная. Например, оператор присваивания в коде ниже приведет к объектной ссылке, содержащейся в **p**, которая будет освобождена в конце блока, содержащего объявление **a**.

```
//C++
```

```
A_var a;
A_ptr p = //...как нибудь получаете objref...

a = p;
```

1.3.2. Расширение Объектных Ссылок

Наследование интерфейса в OMG IDL не требует, чтобы соответствующие классы C++ были связаны, хотя это одна из возможных реализаций. Однако, если интерфейс **B** наследуется от интерфейса **A**, следующие неявные расширения операций для **B** должны поддерживаться совместимой реализацией:

- **B_ptr** в **A_ptr**
- **B_ptr** в **Object_ptr**
- **B_var** в **A_ptr**
- **B_var** в **Object_ptr**

Неявное расширение от **B_var** к **A_var** или **Object_var** не поддерживается; вместо этого, расширение между типами **_var** для объектных ссылок требует запроса к **_duplicate** (Отображение **T_ptr** в **T*** невозможно в C++, для предоставления неявного расширения между **T_var** типами пока также предоставляется необходимой семантикой дублирования (duplicate) для **T_ptr** типов). Попытка неявного расширения от одного типа **_var** до другого, должна вызывать ошибку (Это может достигаться путем получения всех типов **T_var** для объектных ссылок от базового класса **_var** скрытого в пределах каждого типа **T_var**). Во время компиляции, назначение между двумя объектами **_var** того же самого типа поддерживается, но расширяющие назначения недопустимы и должны вызывать ошибку во время компиляции. Расширение назначений может быть проделано через **_duplicate**. Те же самые правила применяются для типов объекной ссылки, которые вложены в ложные типы, такие как структуры или последовательности.

```
//C++

B_ptr bp = ...

A_ptr ap = bp;                //неявное расширение
Object_ptr objp = bp;         //неявное расширение
objp = ap;                    //неявное расширение


B_var bv = bp;                //принимает во владение bp

ap = bv;                      //неявное расширение, bv сохраняет владение bp

obp = bv;                     //неявное расширение, bv сохраняет владение bp



A_var av = bv;                //неправильно, ошибка во время компиляции
A_var av = B::_duplicate(bv); //av и bv - оба указывают на bp

B_var bv2 = bv;               //неявное _duplicate

A_var av2;
av2 = av;                     //неявное _duplicate
```

1.3.3. Операции Объектной Ссылки

Концептуально, класс **Object** в модуле **CORBA** - это базовый тип интерфейса для всех объектов CORBA; поэтому любая объектная ссылка может быть расширена к типу **Object_ptr**. Наряду с другими интерфейсами, область имен CORBA определяет тип **Object_var**.

CORBA определяет три операции на любой объектной ссылке: **duplicate**, **release**, и **is_nil**. Обратите внимание, что эти операции на объектной ссылке, а не на объектной реализации. Поскольку отображение не требует, чтобы объектные ссылки были самостоятельными объектами C++, то синтаксис "->" не может использоваться, для выражения использования этих операций. Также, для удобства, эти операции могут быть выполнены на нулевой ссылке.

Операции **release** и **is_nil** определены только на типе **Object**, так что они могут быть выражены как обычные функции в пределах области имен CORBA следующим образом:

```
//C++

void release(Object_ptr obj);
Boolean is_nil(Object_ptr obj);
```

Операция **release** указывает, что вызывающая программа больше не будет обращаться к ссылке и соответственно связанные с ссылкой ресурсы освобождаются. Если данная объекная ссылка нулевая, то **release** ничего не делает. Операция **is_nil** возвращает ИСТИНУ, если объектная ссылка содержит специальное значение для нулевой объектной ссылки как определено ORB. Ни операция **release**, ни операция **is_nil** не вогут выбрасывать исключения CORBA.

Операция **duplicate** возвращает новую объектную ссылку с тем же самым статическим типом как и данная ссылка. Отображение для интерфейса включает статическую функцию член, названную **_duplicate** в сгенерированном классе. Например:

```
//IDL

interface A{};


//C++

class A
{
     public:
          static A_ptr _duplicate(A_ptr obj);
};
```

Если данная объектная ссылка нулевая, **_duplicate** возвратит нулевую объектную ссылку. Операция **_duplicate** может выбрасывать системные исключения CORBA.

1.3.4. Приведение (сужение) Объектных Ссылок

Отображение для интерфейса определяет статическую функцию член, называемую **_narrow**, которая возвращает новую данную объектную ссылку. Подобно **_duplicate**, функция **_narrow** возвращает нулевую объектную ссылку, если данная ссылка нулевая. В отличие от **_duplicate**, параметр для **_narrow** - это ссылка на объект любого интерфейсного типа (**Object_ptr**). Если фактический тип объекта параметра может быть расширен к типу требуемого интерфейса, то **_narrow** возвратит корректную ссылку; иначе, **_narrow** возвратит нулевую объектную ссылку. Например, если предположить что **A**, **B**, **C**, и **D** - типы интерфейсов, и **D** наследуется от **C**, который наследуется от **B**, который в свою очередь наследуется от **A**. Если объектная сслка к объекту **C** расширена к переменной **A_ptr** называемой **ap**, тогда:

- **A::_narrow(ap)** возвращает корректную объектную ссылку
- **B::_narrow(ap)** возвращает корректную объектную ссылку
- **C::_narrow(ap)** возвращает корректную объектную ссылку
- **D::_narrow(ap)** возвращает нулевую объектную ссылку

При сужении к **A**, **B**, и **C**все в порядке, потому что объект поддерживает все эти интерфейсы. А **D::_narrow** возвращает нулевую объектную ссылку, потому, что объект не поддерживает интерфейс **D**.

Для другого примера предположим что **A**, **B**, **C**, и **D** - это интерфейсные типы. **C** наследуется от **B**, а оба **B** и **D** наследуются от **A**. Теперь предположим, что объект типа **C**передается в функцию **A**. Если функция вызывает **B::_narrow** или **C::_narrow**, то будет возвращена новая объектная ссылка. А если вызывается **D::_narrow**, то будет возвращена нулевая ссылка.

В случае успеха, функция **_narrow** создает новую объектную ссылку и не потребляет данную объектную ссылку, так что вызывающая программа отвечает за освобождение как оригинальной так и новой ссылки.

Операция **_narrow** может выталкивать системные исключения CORBA.

1.3.5. Объектная ссылка nil

Отображение для интерфейса определяет статическую функцию член, названную **_nil**, которая возвращает нулевую объектную ссылку данного интерфейсного типа. Для каждого интерфейса **A**, следующий вызов гарантировано возвращает TRUE:

```
//C++

Boolean tru_result = is_nil(A::_nil());
```

Совместимому приложению нет необходимости вызывать **release** для объектной ссылки возвращенной из функции **_nil**.

Объектные ссылки могут не сравниваться используя **operator==**; поэтому, **is_nil** единственный совместимый путь, через который объектная ссылка может быть проверена, является ли она нулевой. Функция **_nil** не может выталкивать никакие CORBA исключения. Совместимая программа не может пытаться вызывать операцию через нулевую объектную ссылку, так как правильная реализация C++ реализует нулевую объектную ссылку через нулевой указатель.

1.3.6. Объектная ссылка как выходящий параметр

Когда **_var** передается как **out** параметр, любое предыдущее значение должно неявно освобождаться. Чтобы дать реализациям отображения на C++ достаточно обработчиков прерываний, каждый для каждого типа объектной ссылки генерируется тип **_out**, который используется исключительно как параметр для **out**. Например, интерфейс возвращает в типе объектной ссылки **A_ptr**, вспомогательный тип **A_var**, и **out** параметр типа **A_out**. Общая форма для объектной ссылки типа **_out** показаны ниже.

```
//C++

class A_out
{
     public:
          A_out(A_ptr& p) : ptr_(p) {ptr_ = A::_nil(); }
          A_out(A_var^ p) : ptr_(p_ptr_){
               release(_ptr); ptr_ = A::_nil();
          }
          A_out(A_out& a) : ptr(a.ptr_){}
          A_out& operator=(A_out& a){
               ptr_ = a.ptr_; return *this;
```

```
          }
          A_out& operator=(const A_var& a){
               ptr_ = A::_duplicate(A_ptr(a)); return *this;
          }
          A_out& operator=(A_ptr p) { ptr_ = p; return *this; }
          operator A_ptr&() { return ptr_; }
          A_ptr& ptr() { return ptr_; }
          A_ptr operator->() { return ptr_; }
     private:
          A_ptr& ptr_;
};
```

Первый конструктор связывает элемент данных ссылки с параметром **A_ptr&**. Второй конструктор связывает элемент данных ссылки с объектной ссылкой **A_ptr**, которая содержится в параметре **A_var**, и затем вызывает **release()** на объектной ссылке. Третий конструктор, копирующий конструктор, связывает элемент данных ссылки с той же самой объектной ссылкой **A_ptr**, содержащейся в элементе данных его параметра. То есть копирует **A_ptr** из другого **A_out**, передаваемый ему как параметр. Перегруженный оператор для **A_ptr** просто назначает параметр **A_ptr** элементу данных. Перегруженный оператор для **A_var** дублирует **A_ptr**, содержащийся в **A_var** перед назначением его элементу данных. Обратите внимание, что назначение не освобождает любое предварительно содержащееся значение объектной ссылки; в этом отношении, тип **A_out** ведет себя так же как **A_ptr**. Преобразующий оператор **A_ptr&** возвращает элемент данных. Функция член **ptr()**, которая может использоваться теми, кто не полагается на неявное преобразование, также возвращает элемент данных. Перегруженный оператор стрелки (**operator->()**) возвращает элемент данных, чтобы позволить операциям быть вызванными на основной объектной ссылке, после того, как был должным образом инициализирован элемент данных.

1.3.7. Пример отображения Интерфейса

Пример ниже показывает одно возможное отображение для интерфейса. Другие отображения также возможны, но они должны обеспечивать ту же самую семантику и использование как и этот пример.

```
//IDL

interface A
{
     A op(in A arg1, out A arg2);
};


//C++

class A;
typedef A *A_ptr;
class A_var;
class A : public virtual Object
{
     public:
          typedef A_ptr _ptr_type;
          typedef A_var _var_type;


          static A_ptr _duplicate(A_ptr obj);
          static A_ptr _narrow(Object_ptr obj);
          static A_ptr _nil();


          virtual A_ptr op(A_ptr arg1, A_out arg2) = 0;


     protected:
          A();
          virtual ~A();


     private:
          A(const A&);
          void operator=(const A&);
};
```

```
class A_var : public _var
{
     public:
          A_var() : ptr_(A::_nil()){}
          A_var(A_ptr p) : ptr_(p){}
          A_var(const A_var &a) : ptr_(A::_duplicate(A_ptr(a)){}
          ~A_var(){ free(); }


          A_var &operator=(A_ptr p){
               reset(p); return *this;
          }
          A_var &operator=(const A_var& a){
               if(this != &a){
                    free();
                    ptr_ = A::_duplicate(A_ptr(a));
               }
               return *this;
          }
          A_ptr in() const { return ptr_; }
          A_ptr& inout() { return ptr_; }
          A_ptr& out(){
               reset(A::_nil());
               return ptr_;
          }
          A_ptr _retn(){
               A_ptr val = ptr_;
               ptr_ = A::_nil();
               return val;
          }


          operator const A_ptr&() const { return ptr_; }
          operator A_ptr&() { return ptr_; }
          A_ptr operator->() const { return ptr_; }


     protected:
          A_ptr ptr_;
          void free() { release(ptr_); }
          void reset(A_ptr p) { free(); ptr_ = p; }


     private:
          void operator=(const _var &);
};
```

1.4. Отображение для Констант

Константы OMG IDL отображены непосредственно в константы C++, которые могут или не могут опредлять место в памяти в зависимости от области видимости объявления. В следующем примере, константы IDL верхнего уровня отображаются в константы C++ видимые в области файла, принимая во внимание, что вложенные константы отображаются в C++ константы с областью видимости в классе. Эта несогласованность происходит, потому что константы C++ области фидимости файла не могут требовать памяти для хранения (или память может быть скопирована в каждый модуль компиляции), в то время как константы области видимости класса всегда требуют для себя память. Как побочный эффект, это различие означает, что сгенерированный файл заголовка C++ не мог бы содержать значения для констант, определенных для OMG IDL.

```
//IDL

const string name = "testing";


interface A
{
     const float pi = 3.14159;
};


//C++
```

```
static const char *const name = "testing";


class A
{
     public:
          static const Float pi;
};
```

В некоторых ситуациях, использование констант в OMG IDL должно генерировать значение константы вместо имени константы (недавнее изменение, сделанное в языке C++ комитетом стандартизации ANSI/ISO C++, позволяет статическим целочисленным константам быть инициализированным в пределах объявления класса, поэтому для некоторых компиляторов C++, описанные проблемы генерации объектного кода могут не быть проблемой). Например:

```
//IDL

interface A
{
     const long n = 10;
     typedef long V[n];
};


//C++

class A
{
     public:
          static const long n;
          typedef long V[10];
};
```

1.4.1. Константы Широких Символов и Широких Строк

Отображения для широких символьных и широких строковых констант идентичны символам или троковым константам, за исключением того, что литералам IDL предшествует **L** в C++. Например, константа IDL:

```
const wstring ws = "Hello World";
```

отображается в

```
static const WChar *const ws = L"Hello World";
```

на C++.

1.4.2. Константы Постояных Точек (Fixed Point)

Поскольку C++ не имеет прямого типа постоянной точки, литералы IDL постоянной точки отображаются в строки C++ без перемещения 'd' или 'D', чтобы гарантировать, что нет никакой потери точности. Например:

```
//IDL

const fixed F = 123.456D;


//C++

const Fixed F = "123.456";
```

1.5. Отображение для Основных Типов Данных

Основные типы данных имеют отображение показанное в таблице. Обратите внимание, что отображение типа OMG IDL **boolean** определяет только значения 1 (TRUE) и 0 (FALSE); другие значения приводят к неопределенномммму поведению.

OMG IDL	C++	C++ Out Type
short	CORBA::Short	CORBA::Short_out
long	CORBA::Long	CORBA::Long_out
long long	CORBA::LongLong	CORBA::LongLong_out
unsigned short	CORBA::UShort	CORBA::UShort_out
unsigned long	CORBA::ULong	CORBA::ULong_out
unsigned long long	CORBA::ULongLong	CORBA::ULongLong_out
float	CORBA::Float	CORBA::Float_out
double	CORBA::Double	CORBA::Double_out
long double	CORBA::LongDouble	CORBA::LongDouble_out
char	CORBA::Char	CORBA::Char_out
wchar	CORBA::WChar	CORBA::WChar_out
boolean	CORBA::Boolean	CORBA::Boolean_out
octet	CORBA::Octet	CORBA::Octet_out

Каждый основной тип OMG IDL отображен в **typedef** в модуле CORBA. Это сделано из-за того, что некоторые типы, например **short** и **long**, могут иметь различные представления на различных платформах, и CORBA определения отражают соответствующие представления. Например, на 64-битной машине, определение **CORBA::Long** все еще отражает 32-битное целое.

Типы **boolean**, **char**, и **octet** могут обображаться к одному и тому же основному типу C++. Это означает, что эти типы не могут быть различимы с целью перегрузки.

Тип **wchar** отображается в **wchar_t** в стандартной C++ среде или, для нестандартных сред C++, может также отображаться к одному из целочисленных типов. Это означает, что **wchar** не может быть различим от целочисленных типов с целью перегрузки.

Все другие отображения для основных типов различимы с целью перегрузки. То есть можно безопасно записывать перегруженные функции для: **Short**, **UShort**, **Long**, **ULong**, **LongLong**, **ULongLong**, **Float**, **Double**, и **LongDouble**.

Типы **_out** для основных типов используются для **out** параметров в пределах сигнатуры операции. Для основных типов, каждый тип **_out** - это **typedef** для ссылки соответствующего типа C++. Например, **Short_out** определен в области имен CORBA следующим образом:

```
//C++
typedef Short& Short_out;
```

Типы **_out** для основных типов предоставляются для однородности с другими типами **out** параметров.

Программисты обеспокоенные переносимостью должны использовать типы CORBA. Однако, некоторые могут чувствовать, что использование этих типов с квалификацией CORBA вредит удобочитаемости. Если модуль **CORBA** отображен в namespace, оператор C++ **using** может помочь в этой проблеме. На платформах, где тип данных C++ гарантированно идентичен типу данных OMG IDL, совместимая реализация может генерировать родной тип C++.

Для типа **Boolean** определены только значения 1 (TRUE) и 0 (FALSE); другие значения приводят к неопредленному поведению. Так как много существующих пакетов программ C++ и библиотек уже предоставляют их собственные макроопределения препроцессора для TRUE и FALSE, это отображение не требует, чтобы такие определения обеспечивались совместимой реализацией. Требования определений для TRUE и FALSE может вызвать проблемы трансляции для приложений CORBA, которые используют такие пакеты и библиотеки. Вместо этого, мы рекомендуем, чтобы совместимые реализации просто использовали значения 1 и 0 непосредственно.

В качестве альтернативы для тех компиляторов C++, которые поддерживают тип **bool**, то ключевые слова **true** и **false** могут использоваться.

IDL тип **boolean** может быть отображен на C++ как знаковый, беззнаковый, или простой **char**. Это отображение легально для сред ANSI C++. Кроме того, в среде ANSI C++, IDL **boolean** может отображаться в C++ **bool**. Другие отображения в C++ кроме отображения в **char**, или **bool** нелегальны.

1.6. Отображение для Перечислений (enums)

OMG IDL **enum** отображается напрямую в соответствующиее определение типа C++. Единственное различие - это то, что сгенерированный тип C++ может нуждаться в дополнительной константе, которая является достаточно большой, чтобы вынудить компилятор C++ использовать точно 32 бита для значений, объявленных, через перечислимый тип.

```
//IDL

enum Color { red, green, blue };


//C++

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

Кроме того, тип **_out** используется в пределах сигнатур операций для параметров **out**, сгенерированный для каждого перечислимого типа. Для, показанного выше, перечислимого типа **Color**, тип **Color_out** определен в той же самой области следующим образом:

```
//C++

typedef Color& Color_out;
```

Типы **_out** для перечислимых типов сгенерированы для единообразности с другими типами **out** параметра.

1.7. Отображение для Строковых Типов

Как и в отображении на C, строковый тип OMG IDL, ограниченный или неограниченный, отображается в **char*** на C++. Сроковые данные должны оканчиваться нулевым значением. Кроме того, модуль **CORBA** определяет класс **String_var**, который содержит значение **char***, и автоматически оснобождает указатель, когда освобождается объект **String_var**. Когда **String_var** создается или назначается от указателя **char***, **char*** используется и поэтому к сроковым данным нельзя больше обращаться через указатель. При назначении или создании из **const char*** или другого **String_var** делается копия. Класс **String_var** также предоставляет операции, для конвертации к и от значения **char***, так же как и опрации доступа к отдельным символам в пределах строки.

C++ не имеет встроенного типа, который бы обеспечил "близкое соответствие" для IDL ограниченных строк. В результате, программист отвечает за ограничение строк во время выполнения. Реализация отображения не обязана предотвращать назначение строкового значения к ограниченному строковому типу, если строковое значение превышает заданную длинну. Реализации могут выбирать (во время выполнения) пытаться передавать или нет строковое значение, которое превышает заданную в длинну. Если реализация выбирает обнаруживать эту ошибку, то она должны выталкивать системное исключение BAD_PARAM, чтобы сообщить об ошибке.

Поскольку строка отображается в **char***, строковый тип OMG IDL - единственный неосновной тип, для которого отображение назначает требования к размеру. Для динамического выделения строк, совместимые программы должны использовать следующие функции из области имен **CORBA**:

```
//C++

namespace CORBA{
     char* string_alloc(ULong len);
     char*string_dup(const char*);
     void string_free(char *);
     ...

}
```

Функция **string_alloc** динамически выделяет строку, или возвращает нулевой указатель, если она не может выделить память. Эта функция выделяет **len+1** символов, так чтобы результирующая строка имела достаточно места, чтобы поместить оконечный нулевой символ. Функция **string_dup** динамически выделяет достаточно места, чтобы сделать копию ее строкового параметра, включая нулевой символ; копирует строковый параметр в выделенную память, и возвращает указатель на новую строку. Если при этом происходят сбои выделения памяти, то возвращается нулевой указатель. Функция **string_free** освобождает строку, которая была выделена **string_alloc** или **string_dup**. Передача нулевого указателя в **string_free** допустима и не приводит ни к какому действию. Эти функции позволяют реализациям ORB использовать специальные механизмы управления памятью для строк в случае необходимости, без того, чтобы заменять глобальные **operator new** и **operator new[]**.

Функции **string_alloc**, **string_dup** и **string_free** не могут выталкивать исключения.

Обратите внимание, что статический массив **char** в C++ обозначается как **char***, и поэтому нужно быть внимательным при назначении его **String_var**, так как он предполагает, что переданный ему указатель на данные, указывает на данные для которых выделена память через функцию **string_alloc** и таким образом будет делать попытку сделать для него **string_free**, чтобы освободить память:

```
//C++

//Следующее определение вызовет ошибку, из-за того, что память под строку
//не выделена функцией string_alloc

String_var s = "static string";


//А вэтом случае все в порядке, так как статическая строка копируется

const char* sp = "static string";
s = sp;
s = (const char*)"static string too";
```

Когда **String_var** передается как **out** параметр, любое предыдущее значение должно быть неявно освобождено. Чтобы дать реализации отображения на C++ достаточно зацепок, чтобы выполнить это требование, строковый тип так же приводят к типу **String_out**, определенный в области имен **CORBA**, и который используется исключительно как строка для **out** параметра. Общая форма для **String_out** показана ниже.

```
//C++

class String_out
{
     public:
          String_out(char*& p) : ptr_(p) {ptr_ = 0; }
          String_out(String_var& p) : ptr_(p.ptr_){
               string_free(ptr_); ptr_ = 0;
          }
          String_out(String_out& s) : ptr_(s.ptr_) {}
          String_out& operator=(String_out& s){
               ptr_ = s.ptr_; return *this;
          }
          String_out& operator=(char* p){
               ptr_ = p; return *this;
          }
          String_out& operator=(const char* p){
               ptr_ = string_dup(p); return *this;
          }
          operator char*&(){ return ptr_; }
          char*& ptr() { return ptr_; }


     private:
          char*& ptr_;


          void operator=(const String_var&);
};
```

Первый конструктор связывает ссылочный элемент данных с аргументом **char*&**. Второй конструктор связывает ссылочный элемент данных с **char***, который содержится в аргументе **String_var**, и затем вызывает **string_free()** на строке. Третий конструктор, копирующий конструктор, связывает ссылочный компонент данных, с тем же **char*** как и у его аргумента. Перегруженный оператор для **char*** просто назначает аргумент **char*** элементу данных. Перегруженный оператор для **const char*** дублирует аргумент и назначает результат элементу данных. Обратите внимание, что назначение не освобождает предварительно хранимую в элементе данных строку; в этом отношении, тип **String_out** ведет себя также как **char***. Конверсионный оператор **char*&** возвращает элемент данных. Функция член **ptr()**, которая может использоваться, для того, чтобы избежать неявного преобразования и так же возвращает элемент данных.

Назначение от **String_var** к **String_out** отвергается из-за двусмысленности управления памятью. Однозначно, невозможно определить, должна ли строка, принадлежащая **String_var** быть принята **String_out** без копирования, или она должна быть скопирована. Отвергающее назначение от **String_var** вынуждает прикладного разработчика делать выбор явно:

```
//C++

void
A::op(String_out arg)
{
     String_var s = string_dup("some string");
     ...

     arg = s;               //недопустимо
     arg = string_dup(s);   // 1: копирование, или
     arg = s._retn();       // 2: adopt

}
```

В строке, отмеченной в комментарии как "1", прикладная программа явно копирует строку, которая находится в **String_var** и назначает результат аргументу **arg**. В качестве альтернативы прикладная программа может использовать методику, показанную в строке которая отмечена в комментарии как "2", чтобы вынудить **String_var** уступить монопольное использование строки, и чтобы можно было возвратить ее в аргументе **arg** без ошибок управления памятью.

Совместимая реализация отображения должна предоставлять перегруженные операторы **operator<<** (вставка) и **operator>>** (извлечение) для использования **String_var** и **String_out** напрямую с C++ iostream.

1.8. Отображение для Типов Широких Строк

И ограниченные и неограниченные широкие строковые типы отображаются в **CORBA::WChar*** на C++. Кроме того, модуль **CORBA** определяет классы **WString_var** и **WString_out**. Каждый из этих классов предоставляет те же самые функции члены с той же самой семантикой что и их коллеги **String**, с той лишь разницей, что они имеют дело с широкими строками и широкими символами.

Динамическое распределение и освобождение широких строк должно быть выполнено через следующие функции:

```
//C++

namespace CORBA{
     //...

     WChar *wstring_alloc(ULong len);
     WChar *wstring_dup(const WChar* ws);
     void wstring_free(WChar*);
};
```

Эти функции имеют ту же самую семантику как те же самые функции строкового типа, с той лишь разницей, что они работают с широкими строками.

Совместимая реализация отображения должна предоставлять перегруженные операторы **operator<<** (вставка) и **operator>>** (извлечение) для использования **WString_var и WString_out** напрямую с C++ iostream.

1.9. Отображение для Структурированных Типов

Отображение для **struct**, **union**, и **sequence** - это C++ структура или класс с заданным по умолчанию конструктором, копирующим конструктором, оператором присваивания и деструктором. Заданный по умолчанию конструктор инициализирует поля, которые являются объектными ссылками, соответствующими по типу значениями **nil** объектных ссылок, а строковые поля пустыми строками ("" и L"", соответственно). Все другие члены инициализируются через их заданные по умолчанию конструкторы. Копирующий конструктор делает глубокую копию существующей структуры, чтобы создать новую структуру, включая вызов **_duplicate** для всех объектных ссылок и выполнения всех необходимых выделений памяти для всех полей строк и широких строк. Деструктор освобождает все поля объектных ссылок и удаляет все поля строк и широких строк.

Отображение для структурированных типов OMG IDL (структуры, объединения, массива и последовательности) может слегка изменяться в зависимости от того, является ли структура данных *фиксированной длинны* или *переменной длинны*. Структура может быть *переменной длины* в одном из следующих случаев:

- Тип **Any**
- Ограниченная или неограниченная строка или широкая строка
- Ограниченная или неограниченная последовательность (sequence)
- Объектная ссылка или ссылка на передаваемый псевдо-объект
- Тип **valuetype**
- Структура или объединение, содержит поля с переменной длиной
- Массив из элементов типа переменной длины
- Описание типа (typedef) к типу переменной длины

Причина для различной обработки структур данных фиксированной и переменной длины состоит в том, чтобы позволить большую гибкость в выделении памяти для **out** параметров и возвращаемых значений из операции. Эта гибкость позволяет заглушку для операции клиентской стороны, которая возвращает последовательность строк (например, выделить всю строковую память в одной области, которая освобождается одним вызовом).

Как удобство для управления указателями на типы данных переменной длины, отображение также предоставляет управляющий вспомогательный класс для каждого типа переменной длины. Этот тип, который именуется добавлением суффикса "_var" к исходному имени типа, автоматически удаляет указатель, когда экземпляр уничтожается. Объект типа **T_var** ведет себя подобно структурированному типу **T**, за исключением того, что к элементам нужно обращаться косвенно. Для структуры это означает, что нужно использовать стрелку ("**->**") вместо точки ("**.**").

```
//IDL

struct S { string name; float age; };
void f(out S p);


//C++

S a;
S_var b;
f(b);
a = b; //глубокая копия
cout << "names " << a.name << ", " << b->name << endl;
```

Для способствования программированию, базирующемуся на шаблонах, все **struct**, **union**, и **sequence** классы содержат вложенные публичные описания типа (typedef) для своих связанных типов **T_var**. Например, для IDL **sequence** под названием **Seq**, отображенный **sequence** класс **Seq** содержит определение типа **_var_type** следующим образом:

```
//C++

class Seq_var;
class Seq
{
     public:
          typedef Seq_var _var_type;
          //...

};
```

1.9.1. Типы T_var

Общая форма типов **T_var** показана ниже.

```
//C++

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;


          /* in тип параметра */ in() const;
          /* inout тип параметра */ inout();
          /* out тип параметра */ out();
          /* return type */ _retn();


          //другие операторы преобразования для поддержки
          //передачи параметров
};
```

Конструктор по умолчанию создает **T_var**, который содержит нулевой указатель **T***. Совместимые приложения не могут ни пытаться конвертировать **T_var**, созданный с заданным по умолчанию конструктором в **T***, ни использовать его перегруженный оператор **operator->** без предварительного присвоения ему правильного ненулевого **T*** или другого правильного, не содержащего нулевой **T***, **T_var**. Из-за соответствующих трудностей, от совместимой реализации не требуется обнаруживать эту ошибку. Преобразование пустого **T_var** к **T_out** позволяется, однако, так, чтобы **T_var** можно было бы легально передавать как **out** параметр. Преобразование нулевого **T_var** к **T*&** также позволяется, для совместимости с более ранними версиями этой спецификации.

Конструктор **T*** создает **T_var**, который, когда уничтожается, сделает **delete** памяти, на которую указывает параметр **T***. Параметр для этого конструктора никогда не должен быть нулевым указателем. Совместимой реализации не требуется обнаруживать нулевые указатели, переданные в конструктор.

Копирующий конструктор делает глубокую копию любых данных, указанных параметром **T_var** конструктора. Эта копия будет уничтожена при уничтожении **T_var** или когда ему будет присвоено новое значение. Совместимые реализации могут, но не обязаны, применять некоторую форму счетчика ссылок для того, чтобы недопустить таких копий.

Деструктор использует **delete** для освобождения любых данных, указанных спомощью **T_var**, за исключением типов строк и массивов, которые освобождаются,используя функции освобождения **string_free** и **T_free** (для массива типа **T**) соответственно.

Оператор присваивания **T*** приводит к освобождению любых старых данных, указанных с помощью **T_var**, перед присвоением параметра **T***.

Обычный оператор присваивания делает глубокую копию любых данных, указанных параметром **T_var** присваивания . Эта копия будет разрушена, когда будет разрушаться **T_var** или когда ему будет присвоено новое значение.

Перегруженный оператор **operator->** возвращает **T***, содержащийся в **T_var**, но сохраняет владение им. Совместимые приложения не могут вызывать эту функцию, если **T_var** не был инициализирован верным непустым указателем **T*** или **T_var**.

В дополнение к членским функциям, описанным выше, типы **T_var** должны поддерживать функции преобразования, которые будут позволять им полностью поддерживать режимы передачи параметров. Форма этих функций преобразования не определена, чтобы позволить различные реализации, но преобразования должны быть автоматическими (то есть, никакой явный прикладной код не должен их вызывать).

Поскольку неявные преобразования могут иногда вызывать проблемы с некоторыми компиляторами C++ и с удобочитаемостью кода, типы **T_var** также поддерживают членские функции, которые позволяют им быть явно преобразованными с целью передачи параметров.

T_var передается как:	**а приложение может вызвать ...**
in параметр	функцию член **in()**
inout параметр	функцию член **inout()**
out параметр	функцию член **out()**

Для получения возвращаемого значения от **T_var**, приложение может вызвать функцию член **_retn()**.

Для типов **T_var**, которые возвращают **T*&** из членской функции **out()**, членская функция **out()** вызывает **delete** для **T***, который содержится в **T_var**, и устанавливает его в нулевой указатель, а затем возвращает ссылку на него. Это позволяется для правильного управления **T***, принадлежащего **T_var**, когда он передается как **out** параметр. Пример реализации такой **out()** показан ниже:

```
//C++

T*& T_var::out()
{
     // считаем, что ptr_ это поле данных T* в T_var
     delete ptr_;
     ptr_ = 0;
     return ptr_;
}
```

Точно также, для типов **T_var,** чей соответствующий тип **T** возвращается из IDL операций как **T***, функция **_retn()** помещает значение **T***, содержащееся в **T_var**, во временный указатель, устанавливает **T*** в нулевой указатель и возвращает временный указатель. Таким образом **T_var** уступает владение своим **T*** вызывающему **_retn()**, без вызова **delete** над ним, и вызывающий становится ответственным за удаление возвращенного **T***. Пример реализации такой функции **_retn()** показан ниже:

```
//C++

T* T_var::_retn()
{
     T* tmp = ptr_;
     ptr_ = 0;
     return tmp;
}
```

Это позволяет, например, реализации метода сохранять **T*** как потенциальное возвращаемое значение в **T_var** так, чтобы оно было удалено, если выталкивается исключение, и все же иметь возможность прибегать к контролю **T***, чтобы иметь возможность вернуть его соответствующим образом:

```
//C++

T_var t = new T;    // t владеет указателем на T

if(exceptional_condition){
     // t владеет указателем и удалит его
     // при разматывании стека по причине исключения
     throw AnException();
}
...
```

```
return t._retn();   // _retn() получает владение указателем из t
```

После вызова **_retn()** над экземпляром **T_var**, внутренний указатель **T*** устанавливается в нуль, и поэтому вызов любой из его функций через перегруженный оператор **operator->** , без предварительного присвоения ему легального непустого указателя **T***, будет пытаться адресовать нулевой указатель, что является нелегальным в C++.

Типы **T_var** также производятся для структурированных типов фиксированной длины по причине согласованности. Эти типы имеют ту же семантику, как и типы **T_var** для типов переменной длины. Это позволяет приложениям быть закодированными в терминах **T_var** независимо от того, применяются ли типы в основе фиксированной или переменной длины.

Любой тип **T_var** должен быть определен на том же уровне вложенности, как и его тип **T**.

Типы **T_var** не работают с указателем на константный **T**, так как они не предоставляют ни конструктора, ни оператора **operator=** с получением параметра **const T***. Так как C++ не позволяет вызывать **delete** для **const T***, объект **T_var** должен обычно копировать константный объект; вместо этого, отсутствие конструктора и операторов присваивания для **const T*** приведет к ошибке компиляции, если будут предприняты такие инициализация или присваивание. Это позволяет прикладному разработчику решать, дествительно ли требуется копия или нет. Явное копирование объектов **const T*** в типы **T_var** может быть достигнуто через копирующий конструктор **T**:

```
//C++

const T *t = ...;
T_var tv = new T(*t);
```

1.9.2. Типы T_out

Когда **T_var** передается как **out** параметр, любое его предыдущее значение должно быть неявно удалено. Чтобы дать реализациям отображения на C++ достаточно зацепок для выполнения этого требования, каждый тип **T_var** имеет соответствующий тип **T_out**, который используется исключительно как тип **out** параметра. Общая форма для типов **T_out** для типов переменной длины показана ниже:

```
//C++

class T_out
{
     public:
          T_out(T*& p) : ptr_(p) { ptr_ = 0; }
          T_out(T_var& p) : ptr_(p.ptr_){
               delete ptr_;
               ptr_ = 0;
          }
          T_out(const T_out& p) : ptr_(p.ptr_) {}
          T_out& operator=(const T_out& p){
               ptr_ = p.ptr_;
               return *this;
          }
          T_out& operator=(T* p) { ptr_ = p; return *this; }


          operator T*&() { return ptr_; }
          T*& ptr() { return ptr_; }


          T* operator->() { return ptr_; }


     private:
          T*& ptr_;


          // присваивание из T_var не позволено
```

```
        void operator=(const T_var&);
};
```

Первый конструктор привязывает поле данных ссылки с аргументом **T*&** и устанавливает указатель в значение нулевого указателя. Второй конструктор связывает поле данных ссылки с указателем, хранимым аргументом **T_var**, и затем вызывает **delete** над указателем (или **string_free()** в случае типа **String_out** или **T_free()** в случае **T_var** для типа массива **T**). Третий конструктор, копирующий, связывает поле данных ссылки с таким же указателем, указанным полем данных аргумента конструктора. Присваивание из другого **T_out** копирует **T***, указанный с помощью **T_out** аргумента, в поле данных. Перегруженный оператор присваивания для **T*** просто присваивает аргумент указателя в поле данных. Заметим, что присваивание не вызывает удаления любого ранее сохраненного указателя, кроме этого отличия, типа **T_out** ведет себя точно как **T***. Оператор преобразования **T*&** возвращает поле данных. Членская функция **ptr()**, которая может быть использована, чтобы избежать зависимости от неявного преобразования, также возвращает поле данных. Перегруженный оператор стрелки (**operator->()**) позволяет доступ к полям структуры данных, указанной с помощью **T*** поля данных. Соответствующие приложения не могут вызывать перегруженный **operator->()**, пока **T_out** не будет инициализирован с верным ненулевым **T***.

Присваивание в **T_out** из экземпляров соответствующего типа **T_var** является непозволительным, потому что не существует способа определить, хочет ли прикладной разработчик, чтобы выполнилось копирование, или **T_var** должен передать владение его управляемым указателем, который может быть присвоен в **T_out**. Чтобы выполнить копирование **T_var** в **T_out**, приложение должно использовать **new**:

```
// C++

T_var t = ...;
my_out = new T(t.in());// копия, выделенаая в куче(heap)
```

Функция **in()**, вызванная над **t**, обычно возвращает **const T&**, подходящий для вызова конструктора копирования вновь выделенного экземпляра **T**.

Напротив, сделать, чтобы **T_var** передал владение своим управляемым указателем так, что он мог быть возвращен в параметре **T_out**, приложение должно использовать функцию **T_var::_retn()**:

```
// C++

T_var t = ...;
my_out = t._retn();// t передает владение, без копирования
```

Заметим, что типы **T_out** не предназначены служить как типы данных общего назначения, чтобы быть созданными и уничтоженными приложениями, они используются только как типы вместе с сигнатурами операций для позволения корректной работы необходимых сторонних эффектов управления памятью.

1.10. Отображение для типов структур

OMG IDL структура (struct) отображается в структуру C++, каждый член (поле) OMG IDL структуры отображен в соответствующий член (поле) структуры C++. Такое отображение позволяет доступ к одному полю, также как инициализацию в целом большинства структур фиксированной длины. Для получения такой инициализации C++ структуры должны не содержать конструкторов, определенных пользователем, операторов присваивания или деструкторов, и каждый член структуры должен быть самоуправляемым типом. За исключением строк и объектных ссылок, тип поля C++ структуры является обычным отображением типа OMG IDL поля.

Для строк, широких строк или объектных ссылок имя C++ типа поля не указывается отображением, поэтому соответствующая программа не может создать объект этого типа. Поведение типа такое же, как при нормальном отображении (**char *** для строки, **WChar *** для широкой строки и **A_ptr** для интерфейса A), за исключением того, что конструктор копирования типа копирует память поля и его оператор присваивания освобождает старую память поля. Эти типы должны также предоставлять функции **in(), inout(), out() и _retn()**, чтобы их соответствующие **T_var** типы предоставили возможность поддержки режимов передачи параметров, указанных в таблице 1-3. Подходящая реализация отображения также предоставляет переопределенные операторы **operator<<** (включение) и **operator>>** (извлечение) для использования полей строк и широких строк напрямую с C++ потоками ввода/вывода.

Для неименованных членов последовательности (требуемых для рекурсивных структур) требуется имя типа для члена. Это имя генерируется приписыванием перед имением члена подчеркивания и добавлением в конец "_seq". Например:

```
// IDL

struct node {
         long value;
         sequence<node, 2> operand;
};
```

Это будет отражено в следующем C++ коде:

```
// C++

struct node {
      typedef ... _operand_seq;
      Long value;
      _operand_seq operand;
};
```

В C++ коде, показанном выше, выражение "..." в описании типа **_operand_seq** обозначает реализационно-зависимый тип последовательности. Имя этого типа не является стандартизованным.

Присваивание между членами строками, широкими строками или объектными ссылками и соответствующими **T_var** типами (**String_var**, **WString_var** или **A_var**) всегда выражается в копировании данных, пока присваивание не делается с указателем. Одно исключение из правил для присваивания - это когда **const char*** или **const WChar*** присваивается члену, в этом случае память копируется.

Когда старая область памяти не должна быть освобождена (например, если она является частью записи активации функции) , она может получать член напрямую, как указатель, используя _ptr аксессор поля. Такое использование является опасным и в общем случае должно быть обойдено.

```
// IDL

struct FixedLen { float x, y, z; };
```

```
// C++

FixedLen x1 = {1.2, 2.4, 3.6};
FixedLen_var x2 = new FixedLen;
x2->y = x1.z;
```

Вышеприведенный пример показывает использование типов **T** и **T_var** для стркутуры фиксированной длины. Когда она выйдет за границы видимости, **x2** автоматически освободит выделенный в куче (heap) объект **FixedLen**, используя операцию **delete**.

Следующие примеры иллюстрируют смешанное использование типов **T** и **T_var** для типов переменной длины, используя следующее OMG IDL описание:

```
// IDL

interface A;
struct Variable { string name; };


// C++

Variable str1;                    // str1.name изначально пусто


Variable_var str2 = new Variable;// str2->name
                                  // изначально пусто
char *non_const;
const char *const2;
String_var string_var;
const char *const3 = "string 1";
const char *const4 = "string 2";
str1.name = const3;           // 1: освобождает старую память, копирует
str2->name = const4;          // 2: освобождает старую память, копирует
```

В примере выше компоненты name переменных str1 и str2 обе изначально выделены как пустые строки. В строке, помеченной 1, const3 присваивается компоненте name переменной str1. В результате этого предыдущее значение str1.name будет освобождено и, так как const3 указывает на постоянные данные, содержимое const3, будет скопировано. В данном случае str1.name было пустой строкой, поэтому оно должно быть освобождено перед тем, как будет иметь место копирование const3. Строка 2 подобна строке 1, за исключением того, что str2 типа T_var.

Продолжим пример:

```
// C++

non_const = str1.name;    // 3: не освобождается, не копируется
const2 = str2->name;      // 4: не освобождается , не копируется
```

В строке, помеченной 3, str1.name присваивается non_const. Так как non_const является типом указателя (**char** *), str1.name не освобождается, и данные, на которые указывает поле, не копируются. После присваивания, str1.name и non_const эффективно указывают на одну и туже область памяти, при этом str1.name получается владельцем этой памяти. Строка 4 идентична строке 3, только const2 является указателем на const char, str2->name не освобождается и не копируется, потому что const2 - тип указателя.

```
// C++

str1.name = non_const;    // 5: освобождается, не копируется
str1.name = const2;       // 6: fосвобождается старая память, копируется
```

Срока 5 вызывает присваивание char * в str1.name, которое выражается в том, что старое значение str1.name будет освобождено и значение указателя non_const, но не данные, на которые он указывает, будут скопированы.

Другими словами, после присваивания str1.name указывает на ту же область памяти, что и non_const. Строка 6 такая же, как и строка 5, за исключением того, что const2 является *const char **, поэтому данные, на которые она указывает, будут скопированы.

```
// C++

str2->name = str1.name;    // 7: освобождается старая память, копируется
str1.name = string_var;    // 8: освобождается старая память, копируется
string_var = str2->name;   // 9: освобождается старая память, копируется
```

В строке 7 присваивание выполняется полю от другого поля, поэтому исходное значение указанного справа поля освобождается и новое значение копируется. Подобным же образом строки 8 и 9 вызывают присваивание от и для **String_var**, в обоих случаях исходное значение с левой стороны освобождается и новое значение копируется.

```
// C++

str1.name._ptr = str2.name; // 10: не освобождается, не копируется
```

Наконец, строка 10 использует **_ptr** аксессор поля, поэтому нет ни освобождения, ни копирования. Подобное использование является опасным и в общем случае должно быть обойдено.

Реализации ORB, связанные со способностью взаимодействия одиночных процессов с использованием отображения на C, могут переопределять **operator new()** и **operator delete()** для структур так, что динамическое выделение памяти использует тот же механизм, что и функции динамического выделения памяти языка C. Будут ли эти операторы переопределены реализацией или нет, подходящие программы используют **new** для динамического выделения структур и **delete** для их освобождения.

1.11. Отображение для Постоянных (Fixed) Типов

C++ отображение для **fixed** определяется с помощью следующего класса:

```
// C++

class Fixed
{
   public:
      // Constructors

      Fixed(int val = 0);
      Fixed(unsigned val);
      Fixed(Long val);
      Fixed(ULong val);
      Fixed(LongLong val);
      Fixed(ULongLong val);
      Fixed(Double val);
      Fixed(LongDouble val);
      Fixed(const Fixed& val);
      Fixed(const char*);
      ~Fixed();

      // Conversions

      operator LongLong() const;
      operator LongDouble() const;
      Fixed round(UShort scale) const;
      Fixed truncate(UShort scale) const;

      // Operators

      Fixed& operator=(const Fixed& val);
      Fixed& operator+=(const Fixed& val);
      Fixed& operator-=(const Fixed& val);
      Fixed& operator*=(const Fixed& val);
      Fixed& operator/=(const Fixed& val);
      Fixed& operator++();
      Fixed operator++(int);
      Fixed& operator--();
      Fixed operator--(int);
      Fixed operator+() const;
      Fixed operator-() const;
      Boolean operator!() const;

      // Other member functions

      UShort fixed_digits() const;
      UShort fixed_scale() const;
};

istream& operator>>(istream& is, Fixed& val);
ostream& operator<<(ostream& os, const Fixed& val);

Fixed operator + (const Fixed& val1, const Fixed& val2);
Fixed operator - (const Fixed& val1, const Fixed& val2);
Fixed operator * (const Fixed& val1, const Fixed& val2);
Fixed operator / (const Fixed& val1, const Fixed& val2);
```

```
Boolean operator > (const Fixed& val1, const Fixed& val2);
Boolean operator < (const Fixed& val1, const Fixed& val2);
Boolean operator >= (const Fixed& val1, const Fixed& val2);
Boolean operator <= (const Fixed& val1, const Fixed& val2);
Boolean operator == (const Fixed& val1, const Fixed& val2);
Boolean operator != (const Fixed& val1, const Fixed& val2);
```

Класс **Fixed** используется непосредственно отображением на C++ для значений IDL констант с фиксированной точкой и для всех промежуточных результатов арифметических операций над значениями с фиксированной точкой. Для параметров с фиксированной точкой IDL-операций или членов структурированных IDL типов данных реализация может использовать тип **Fixed** непосредственно или, альтернативно, может использовать отличный тип, с эффективно постоянным числом цифр и показателем степени, который предоставляет подобный C++ интерфейс и может быть неявно преобразован из/в класс **Fixed**. Имя (имена) этого альтернативного класса не определяется данным отображением. Так как типы с фиксированной точкой, используемые как параметры IDL операций, должны быть поименованы через IDL описание **typedef**, отображение должно использовать **typedef** для определения типа параметра операции, чтобы быть уверенной в том, что сигнатура серверной части операции будет переносима. Приведем пример отображения:

```
// IDL

typedef fixed<5,2> F;

interface A
{
        void op(in F arg);
};

// C++

typedef Implementation_Defined_Class F;

class A
{
   public:
   ...
   void op(const F& arg);
   ...
};
```

Класс **Fixed** имеет набор конструкторов для гарантирования, что fixed значение может быть построено из любого стандартного целого или вещественного IDL типа. Конструктор **Fixed(char*)** преобразует строковое представление литерала с фиксированной точкой в действительно значение с фиксированной точкой, возможно с необязательным 'd' или 'D' на конце. Класс Fixed также предоставляет операторы преобразования обратно в **LongLong** и **LongDouble** типы. Для преобразования в целочисленные типы цифры правее десятичной точки отсекаются. Если значение значения с фиксированной точкой не помещается в целевой тип преобразования, тогда возбуждается системное исключение DATA_CONVERSION.

Функции **round** и **truncate** преобразуют значение в новое с указанной степенью. Если новая степень требует, чтобы значение потеряло точность справа, функция **round** будет округлять в сторону нуля значения, меньшие одной второй, или до следующего абсолютного значения. Функция truncate всегда округляет значение в направление нуля. Например:

```
// C++

Fixed f1 = "0.1";
Fixed f2 = "0.05";
Fixed f3 = "-0.005;
```

В этом примере **f1.round(0)** и **f1.truncate(0)** оба возвращают 0, **f2.round(1)** возвращает **0.1**, **f2.truncate(1)** возвращает **0.0**, **f3.round(2)** возвращает **-0.01** и **f3.truncate(2)** возвращает **0.00**.

Функции **fixed_digits** и **fixed_scale** возвращают наименьшие значения количества цифр и степени, которые могут удовлетворять полному значению с фиксированной точкой. Если реализация использует альтернативные классы для параметров операций и членов структурированных типов, тогда **fixed_digits** и **fixed_scale** возвращают постоянные значения количества цифр и степени, определенные исходным IDL типов с фиксированной точкой.

Арифметические операции над классом Fixed должны вычислять результат точно, используя временное значение эффективной двойной точности (62 цифры). Результаты затем будут усечены во время исполнения для подгонки по максимуму до 31 цифры, используя метод, определенный в CORBA 2.3, раздел синтаксиса и семантики, для уточнения новых цифр и степени. Если результат некоторой арифметической операции получается более, чем 31-разрядный левее десятичной точки, будет возбуждено исключение DATA_CONVERSION. Если значение с фиксированной точкой, используемое как фактический параметр операции или присвоенный члену структурированного IDL типа данных, выходит за пределы максимального по модулю значения по числу цифр или степени, будет возбуждено исключение DATA_CONVERSION.

Потоковые операторы вставки и извлечения << и >> преобразуют значение с фиксированной точкой из/в поток. Точное определение этих операторов может сильно зависеть от уровня стандартизации среды C++. Эти операторы вставляют и извлекают значения с фиксированной точкой из потока, используя тот же формат, как и C++ типы с плавающей точкой. В частности, конечные 'd' или 'D' от IDL представления литерала с фиксированной точкой не вставляются или не извлекаются из потока. Эти операторы используют все элементы управления форматом, применимые в вещественным, определенные классами потоков, за исключением того, что они никогда не используют научный формат.

1.11.1 Типы T_var и T_out для Fixed

Так как типы с фиксированной точкой всегда передаются по ссылке как параметры операции и возвращаются по значению, не требуется **_var** тип для типа с фиксированной точкой. Для каждого **typedef** IDL типа с фиксированной точкой соответствующий тип **_out** определяется как ссылка на тип с фиксированной точкой:

```
// IDL

typedef fixed<5,2> F;


// C++

typedef Implementation_Defined_Name F;
typedef F& F_out;
```

1.12. Отображение для Типов Объединения (Union)

Объединения отображаются в C++ классы с функциями доступа к членам (полям) объединения и дискриминанту. Некоторые функции только предоставляют доступ на чтение поля. Такие функции называются "аксессорными функциями" или "аксессорами" для краткости. Например:

```
// C++

Long x() const;
```

Здесь **x ()** - это аксессор, который возвращает значение поля **x** объединения (типа **Long** в данном примере).

Другие функции только предоставляют доступ на запись поля объединения. Такие функции называются "модифицирующими функциями" или "модификаторами" для краткости. Например:

```
// C++

void x(Long val);
```

Здесь, **x ()** является модификатором, который устанавливает значение поля **x** объединения (типа **Long** в данном примере).

Также другие функции-члены объединения предоставляют доступ на чтение-запись к полю объединения с помощью возвращения ссылки на это поле. Такие функции называются "ссылочными функциями" или "референты" для краткости. Например:

```
// C++

S& w();
```

Здесь **w ()** является референтом для поля **w** (типа **S**) объединения.

Конструктор объединения по умолчанию выполняет невидимую приложением инициализацию объединения. Он не инициализирует ни дискриминатор, ни какие-либо поля объединения для того, чтобы состояние стало полезным приложению. (Реализация конструктора по умолчанию может делать какой-либо тип инициализации по своему желанию, но такая инициализация является реализационно-зависимой. Не соответствующее приложение может рассчитывать, что объединение всегда будет корректно инициализировано с помощью только конструктора по умолчанию.) Присваивание, копирование и уничтожение построенных по умолчанию объединений является безопасным. Присваивание их или копирование построенного по умолчанию объединения выражается в том, что цель присваивания или копирования будет инициализирована так же, как и построенное по умолчанию объединение.

Поэтому будет ошибкой для приложения требовать доступ к объединению перед его установкой, но реализациям ORB не требуется определять эту ошибку по причине сложности сделать так. И конструктор копирования и оператор присваивания оба выполняют внутреннее копирование своих параметров, с освобождением оператором присваивания старой области памяти по необходимости. Деструктор освобождает всю память, которой владеет объединение.

Аксессорные и модифицирующие функции дискриминанта объединения имеют имя **_d** для того, чтобы быть краткими и обойти конфликты имен с полями объединения. Модификатор дискриминатора **_d** может быть только использован для установки дискриминанта в значение внутри того же поля объединения. В дополнение к **_d** аксессору и модификатору, объединение с неявным полем по умолчанию предоставляет модифицирующую функцию **_default ()**, что устанавливает дискриминант в легальное значение по умолчанию. Объединение имеет неявное поле по умолчанию, если оно не имеет варианта по умолчанию и не все допустимые значения дискриминанта объединения перечислены. Присваивание, копирование и уничтожение значений объединения непосредственно после вызова **_default ()** является безопасным. Присваивание из и копирование такого объединения выражается в том, что цель присваивания или копирования имеет то же безопасное состояние, словно его функция **_default ()** была вызвана.

Установка значения объединения посредством модификатора автоматически устанавливает дискриминант и может освободить память, связанную с предыдущим значением. Попытка получить значение через аксессор, который не совпадает с текущим дискриминатором, дает в результате неопределенное поведение. Если модификатор для поля объединения с несколькими легальными значениями дискриминанта используется для установки значения дискриминанта, реализация объединения свободна установить дискриминант в любое из этих легальных значений для этого поля. Фактическое значение дискриминанта, выбираемое в этих случаях, является реализационно-зависимым. Вызов референта для поля, которое не совпадает с текущим дискриминантом, выражается в неопределенном поведении.

Следующий пример помогает иллюстрировать отображение типов объединений:

```
// IDL

typedef octet Bytes[64];
struct S { long len; };
interface A;
```

```
valuetype Val;
union U switch (long) {
       case 1: long x;
       case 2: Bytes y;
       case 3: string z;
       case 4:
       case 5: S w;
       case 6: Val v;
       default: A obj;
};


// C++

typedef Octet Bytes[64];
typedef Octet Bytes_slice;
class Bytes_forany { ... };
struct S { Long len; };
typedef ... A_ptr;
class Val ... ;
class U
{
    public:
       U();
       U(const U&);
       ~U();
       U &operator=(const U&);

       void _d(Long);
       Long _d() const;

       void x(Long);
       Long x() const;

       void y(Bytes);
       Bytes_slice *y() const;

       void z(char*);              // free old storage, no copy
       void z(const char*);        // free old storage
       void z(const String_var &); // free old storage, copy

       const char *z() const;

       void w(const S &);          // deep copy
       const S &w() const;         // read-only access
       S &w();                     // read-write access

       void v(Val*);               // _remove_ref old valuetype,
                                   // _add_ref argument

       Val* v() const;             // no _add_ref of return value

       void obj(A_ptr);            // release old objref,
                                   // duplicate

       A_ptr obj() const;          // no duplicate

};
```

Аксессоры и модификаторы для полей объединения предоставляют семантику, подобную той, что используется для членов данных структур. Модифицирующие функции выполняют еквивалент внутреннего копирования для своих параметров, и их параметры должны быть переданы по значению (для небольших типов) или по ссылке на const (для больших типов). Референты могут быть использованы для доступа на чтение-запись, но предоставляются только для следующих типов: **struct, union, sequence, any и fixed**.

Ссылка, возвращаемая из ссылочной функции (референта) продолжает указывать на это поле до тех пор, пока поле является активным. Если активное поле объединения впоследствии изменится, ссылка станет неверной, и попытки чтения или записи поля через ссылку вызовут неопределенное поведение.

Для поля объединения типа массив, аксессор возвращает указатель на кусок массива, где кусок (slice) - это массив со всеми измерениями исходного за исключением первого (куски массивов описаны в деталях в главе 1.14, "Отображение для типов массивов"). Кусок массива возвращает тип, позволяющий доступ для чтения-записи элементов массива через операторы индексирования. Для элементов безымянного типа массива поддержка определений типов (typedefs) для массива должна быть сгенерирована непосредственно внутри объединения. Например:

```
// IDL

union U switch (long) {
       default: long array[20][20];
};


// C++

class U
{
   public:
       // ...

       void array(long arg[20][20]);
       typedef long _array_slice[20];
       _array_slice * array();
       // ...

};
```

Имя поддерживаемого куска массива в typedef создается с помощью приписывания вначале подчеркивания и добавления в конец "_slice" к имени поля объединения. В вышеприведенном примере поле-массив, именованное "array" дает в результате описание куска массива, названного "_array_slice", помещаемое в класс объединения.

Для строковых полей объединения модификатор **char *** дает в результате освобождение старой области памяти перед тем, как будет присвоен владелец параметра указателя, в то время как модификатор **const char*** и модификатор **String_var**[1] оба дают в результате освобождение старой области памяти перед тем, как память параметра будет скопирована. Аксессор для строкового поля возвращает **const char***, позволяющий проверку, но не изменение строковой памяти.[2]

Для объектных ссылок, полей объединения, параметры объектной ссылки в модифицирующих функциях дублируются после уничтожения старой объектной ссылки. Объектная ссылка, возвращаемая как значение из функции аксессора, не дублируется, потому что объединение сохраняет владение объектной ссылкой.

Для анонимных последовательностей как полей объединения (требуемых для рекурсивных объединений), требуется имя типа. Это имя генерируется путем приписывания в начало подчеркивания к имени поля и добавления "_seq" в конец. Например:

```
// IDL

union node switch (long) {
       case 0: long value;
       case 1: sequence<node, 2> operand;
};
```

Это отразится в следующем коде на C++:

```
// C++

class node {
   public:
      typedef ... _operand_seq;
      ...

      // Внутренние функции, работающие с полем operand,
      // используя _operand_seq для его типа.

      ...
```

```
};
```

В показанном выше C++ коде, выражение "..." в описании **_operand_seq** указывает на реализационно-зависимый код последовательности. Имя этого типа не является стандартизированным.

Ограничения на использование **_d** модификатора дискриминатора показаны в следующих примерах, основанных на описании объединения **U**, сделанного выше:

```
// C++

S s = {10};
U u;
u.w(s); // выбрано поле w

u._d(4); // OK, поле w выбрано
u._d(5); // OK, поле w выбрано
u._d(1); // ошибка, другое поле выбрано
A_ptr a = ...;
u.obj(a); // поле obj выбрано
u._d(7); // OK, поле obj выбрано
u._d(1); // ошибка, другое поле выбрано
s = u.w(); // ошибка, поле w не активно
```

Как показано здесь, ни модификатор _d, ни референт w не могут быть использованы для неявного переключения между разными полями объединения. Следующее показывает пример того, как используется внутренняя функция **_default ()**:

```
// IDL

union Z switch(boolean) {
       case TRUE: short s;
};


// C++

Z z;
z._default();           // выбрано неявное поле по умолчанию
Boolean disc = z._d();  // disc == FALSE

U u;                    // объединение U из предыдущего примера
u._default();           // ошибка, error, не предоставляется _default()
```

Для объединения **Z** вызов модифицирующей функции **_default ()** порождает, что значение объединения состоит исключительно из значения дискриминатора **FALSE**, так как не существует явного поля по умолчанию. Для объединения **U** вызов **_default ()** порождает ошибку компиляции, потому что **U** имеет явно объявленный случай по умолчанию и следовательно не имеет функции **_default ()**. Внутренняя функция **_default ()** генерируется только для объединений с неявными полями по умолчанию.

Реализации ORB, связанные с взаимодействием одиночных процессов с отображением на C, могут переопределять **operator new()** и **operator delete()** для объединений так, что динамический вызов использует такой же механизм, как функции динамического выделения памяти языка C. Будут ли эти операторы переопределены или нет, соответствующие программы используют **new** для динамического выделения памяти для объединений и **delete** для их освобождения.

---

1. Отдельный модификатор для **String_var** необходим, потому что он может автоматически конвертировать и в **char *** , и в **const char ***; так как объединения предоставляют модификаторы для обоих этих типов, попытка установить строковое поле объединения из **String_var** может иначе дать в результате ошибку неоднозначности во время компиляции.

2. Возврат типа **char ***, позволяющего доступ на чтение-запись, может ошибочно быть присвоен в **String_var**, давая в результате, что **String_var** и объединение оба являются владельцами памяти строки.

1.13. Отображение для Типов Последовательности (sequence)

Последовательность отображается в C++ класс, который ведет себя как массив с текущей длиной и максимальной длиной. Для ограниченной последовательности максимальная длина является неявной в типе последовательности и не может быть явно управляться программистом. Для неограниченной последовательности начальное значение максимальной длины может быть указано в конструкторе последовательности для предоставления контроля над длиной начального выделяемого буфера. Программист может всегда явно модифицировать текущую длину любой последовательности.

Для неограниченной последовательности установка длины в большее значение, чем текущая длина, может перевыделить память для данных последовательности. Перевыделение памяти концептуально эквивалентно созданию новой последовательности желаемой новой длины, копированию элементов старой последовательности *от нуля до length - 1* в новую последовательность и затем присваивание новой последовательности на место старой. Установка длины в меньшее значение, чем текущая длина, не влияет на манипулирование памятью, связанной с последовательностью. Заметим, однако, что элементы, осиротевшие после этого сокращения, являются более недоступными, и что их значения не могут быть восстановлены с помощью увеличения длины последовательности до исходного значения.

Для ограниченной последовательности попытка установки текущей длины в значение, большее чем максимальная длина, заданная в спецификации OMG IDL, вызывает неопределенное поведение.

Для каждого различного определения типа (typedef), именующего тип последовательности, соответствующая реализация отображения предоставляет отдельный C++ тип последовательности. Например:

```
// IDL

typedef sequence<long> LongSeq;
typedef sequence<LongSeq, 3> LongSeqSeq;


// C++

class LongSeq // неограниченная последовательность {
   public:
      LongSeq();              // конструктор по умолчанию
      LongSeq(ULong max);     // максимальный конструктор
      LongSeq(                // T *data конструктор
         ULong max,
         ULong length,
      Long *value,
         Boolean release = FALSE

                  );
      LongSeq(const LongSeq&);
      ~LongSeq();
      ...

};


class LongSeqSeq              // ограниченная последовательность {
   public:
      LongSeqSeq();           // конструктор по умолчанию
      LongSeqSeq(             // T *data конструктор
         ULong length,
         LongSeq *value,
         Boolean release = FALSE

      );
      LongSeqSeq(const LongSeqSeq&);
      ~LongSeqSeq();
      ...

};
```

И для ограниченной и для неограниченной последовательностей конструктор по умолчанию (как показано в вышеприведенном примере) устанавливает длину последовательности равной 0. Для ограниченных последовательностей максимальная длина является частью типа и не может быть установлена или изменена, в то время как для неограниченных последовательностей конструктор по умолчанию также устанавливает максимальную длину в 0. Конструкторы по умолчанию для ограниченных и неограниченных последовательностей не требуют непосредственного выделения буферов.

Неограниченные последовательности предоставляют конструктор, который позволяет только устанавливать начальное значение максимальной длины ("максимальный конструктор" показан в примере выше). Это позволяет приложениям управлять, как много буферного пространства будет начально выделено последовательностью. Этот конструктор также устанавливает длину в 0 и флаг **release** в **TRUE**.

Конструктор **"T *data"** (как показано в примере выше) позволяет устанавливать длину и содержимое ограниченной или неограниченной последовательности. Для неограниченных последовательностей он также позволяет устанавливать начальное значение максимальной длины. Для этого конструктора владеление буфером определяется параметром **release** - **FALSE** означает, что вызывающий (caller) владеет памятью для буфера и его элементами, в то время как **TRUE** означает, что последовательность присваивает владение памятью для буфера и его элементов. Если **release** равно **TRUE**, принимается, что буфер будет выделен, используя функцию последовательности **allocbuf**, и последовательность передаст его в функцию **freebuf**, когда закончит его использование. Функции **allocbuf** и **freebuf** описаны в подразделе 1.13.3.

Конструктор копирования создает новую последовательность с такими же максимумом и длиной, как и данная последовательность, копирует каждый из ее текущих элементов (элементы *от нуля до length - 1*) и устанавливает флаг **release** в **TRUE**.

Оператор присваивания полностью копирует свои параметры, освобождая старую память по необходимости. Он ведет себя, как если бы исходная последовательность уничтожалась через свой ддеструктор и затем последовательность-источник копировалась, используя конструктор копирования.

Если **release=TRUE**, деструктор уничтожает каждый из текущих элементов (*от нуля до length - 1*) и уничтожает сам буфер последовательности.

Для неограниченной последовательности, если необходимо перевыделение памяти по причине изменения длины и последовательность была создана, используя параметр **release=TRUE** в своем конструкторе, последовательность освободит старую память для всех элементов и буфер. Если **release** равно **FALSE** при этих условиях, старая память не будет освобождена ни для элементов, ни для буфера перед выполнением перевыделения памяти. После перевыделения флаг **release** всегда устанавливается в **TRUE**.

Для неограниченной последовательности, аксессор **maximum ()** возвращает общее количество элементов последовательности, которые могут быть сохранены в текущем буфере последовательности. Это позволяет приложениям узнавать, сколько элементов они могут вставить в неограниченную последовательность без возникновения ситуации перевыделения памяти. Для ограниченной последовательности **maximum ()** всегда возвращает размер последовательности, как задано в ее OMG IDL описании типа.

Функции **length ()** могут быть использованы для доступа и изменения длины последовательности. Увеличение длины последовательности добавляет новые элементы в конец. Вновь добавленные элементы ведут себя, словно они были построены по умолчанию, когда длина последовательности была увеличена. Однако, реализация последовательности может задержать фактическое построение по умолчанию до тех пор, пока вновь добавленный элемент будет впервые запрошен. Для последовательностей строк и широких строк, построение элемента по умолчанию требует инициализации для каждого элемента пустой строкой или широкой строкой. Для последовательностей объектных ссылок построение элемента по умолчанию требует инициализации каждого элемента пустой ссылкой подходящего типа. Для последовательностей типов значений (valuetype), построение элемента по умолчанию требует инициализации каждого элемента пустым (null) указателем. Элементы последовательностей других сложных типов, таких как структуры и последовательности, инициализируются своими конструкторами по умолчанию. Элементы последовательностей объединений не имеют какой-либо видимой приложению инициализации, в частности, элемент объединения, построенный по умолчанию, не является безопасным для маршалинга или доступа. Элементы последовательности базового типа, такого как **ULong**, имеют неопределенные значения по умолчанию.

Переопределяемые операторы индексирования (**operator[]**) возвращают элемент по заданному индексу. Не константная версия должна возвращать нечто, что может служить как lvalue (т.е., нечто, что позволяет присваивание в элемент с заданным индексом), в то время как константная версия должна позволять доступ только для чтения для элемента с заданным индексом.

Переопределенные операторы индексирования не могут использоваться для доступа или изменения любого элемента за границами текущей длины последовательности. Перед любой формой использования **operator[]** над последовательностью, длина последовательности должна быть первоначально установлена, используя модифицирующую функцию **length(ULong)**, если последовательность не была построена с использованием **T *data** конструктора.

Для строк, широких строк и объектных ссылок **operator[]** последовательности должен возвращать тип с такой же семантикой, как и типы, используемые для полей строк, широких строк и объектных ссылок структур и массивов, чтобы присваивание полю строки, широкой строки или объектной ссылки через **operator=()** освобождало старую память в соответствующих случаях. Заметим, что какими бы то ни были специальные возвращаемые типы, они должны принимать на обработку установку параметра **release** в **T *data** конструкторе в отношении уничтожения старой области памяти. Подходящая реализация отображения также предоставляет переопределенные операторы **operator<<** (вставка) и **operator>>** (извлечение) для использования последовательностей элементов строк и широких строк напрямую с C++ потоками ввода/вывода.

Аксессорная функция **release()** возвращает состояние флага **release** для последовательности.

Переопределенные аксессор и референт **get_buffer()** позволяют прямой доступ к буферу, лежащему в основе последовательности. Это может быть очень полезно, когда посылаются большие блоки данных как последовательность, как например посылка данных изображения как последовательности октетов, и когда поэлементный доступ, предоставляемый переопределенными операторами индексирования, не является достаточным.

Неконстантная ссылочная функция (референт) **get_buffer()** позволяет доступ на чтение-запись к лежащему в основе буферу. Если его аргумент **orphan** равен **FALSE** (по умолчанию), последовательность возвращает указатель на свой буфер, выделяемый ею, если не было еще так сделано. Длина буфера может быть определена, используя аксессор **maximum()**. Для ограниченных последовательностей длина возвращаемого буфера равна размеру последовательности. Число элементов в буфере может быть определено из аксессора последовательности **length()**. Последовательность является владельцем буфера, лежащего в основе. Элементы в возвращаемом буфере могут быть непосредственно заменены вызывающим. Для последовательностей строк, широких строк и объектных ссылок вызывающий должен использовать аксессор последовательности **release()** для определения , должны ли быть освобождены элементы (используя **string_free, wstring_free** или **CORBA::release** для строк, широких строк и объектных ссылок соответственно) перед непосредственно присваиванием им. Так как последовательность содержит запись длины и размера буфера, вызывающий функцию **get_buffer ()** не будет удлинять или укорачивать последовательность с помощью непосредственно добавления элементов в буфер или непосредственно удаления элментов из буфера. Изменение длины последовательности будет выполнено только путем вызова модификатора последовательности **length ()**.

Альтернативно, если аргумент **orphan** в функции **get_buffer()** равен **TRUE**, последовательность передает владение буфером вызывающему. Если **orphan** равен **TRUE** и последовательность не владеет своим буфером (т.е. ее флаг **release** равен **FALSE**), возвращаемое значение будет нулевой указатель (null). Если буфер получен из последовательности, используя эту форму **get_buffer()**, последовательность возвращается к тому же состоянию, которое имелось при построении с использованием своего конструктора по умолчанию. Вызывающий становится ответственным за освобождение в конечном счете каждого элемента возвращаемого буфера (для строк, широких строк и объектных ссылок) и последующего освобождения самого возвращенного буфера, используя **freebuf**.

Константный аксессор **get_buffer()** позволяет доступ только на чтение к буферу последовательности. Последовательность возвращает свой буфер, выделяя его в памяти, если он еще не был выделен. Не разрешается прямое изменение возвращаемого буфера вызывающим.

Для неконстантной ссылочной функции (референта) **get_buffer ()** с аргуметном **orphan=FALSE** и для константной аксессорной функции (аксессора) **get_buffer ()** возвращаемое значение остается верным, пока не будет вызвана другая неконстантная внутренняя функция последовательности или пока не будет уничтожена последовательность.

Функция **replace ()** позволяет замещать буфер, лежащий в основе последовательности. Параметры в **replace()** являются идентичными по типу, порядку и назначению таким же для **T * data** конструктора последовательности.

Доступ к буферу, лежащему в основе последовательности, представляется в следствии того, что реализация последовательности должна использовать непрерывную память для хранения элементов, но это не является необходимым случаем. Соответствующая реализация последовательности может хранить свои элементы в различных отдельных буферах памяти и перемещать их в один буфер, только если приложение вызывает аксессоры **get_buffer ()**. Фактически, для приложений, которые никогда не вызывают эти аксессоры, такая реализация скорее всего будет лучше применяться в управлении большими последовательностями, чем та, что использует один непрерывный буфер.

Для **T * data** конструктора последовательности и для параметра буфера функции **replace()**, тип **T** для строк, широких строк и объектных ссылок является **char*, CORBA::WChar*** и **T_ptr** соответственно. Другими словами, буферы строк передаются как **char****, буферы широких строк - как **CORBA::WChar**** и буферы объектных ссылок - как **T_ptr***. Возвращаемый тип неконстантного референта get_buffer() для последовательностей строк **char**, CORBA::WChar**** для последовательностей широких строк и **T_ptr*** для последовательностей объектных ссылок. Возвращаемый тип константной аксессорной функции get_buffer() для последовательностей строк является **const char* const***, **const CORBA::WChar* const*** для последовательностей широких строк и **const T_ptr*** для последовательностей объектных ссылок.

1.13.1. Пример последовательности

Нижеследующий пример показывает полные описания для ограниченной и неограниченной последовательности.

```
// IDL

typedef sequence<T> V1;         // неограниченная последовательность
typedef sequence<T, 2> V2;      // ограниченная последовательность


// C++

class V1                        // неограниченная последовательность {
   public:
      V1();
      V1(ULong max);
      V1(ULong max, ULong length, T *data,
```

```
         Boolean release = FALSE);
      V1(const V1&);
      ~V1();
      V1 &operator=(const V1&);


      ULong maximum() const;


      void length(ULong);
      ULong length() const;


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


      Boolean release() const;


      void replace(ULong max, ULong length, T *data,
               Boolean release = FALSE);


      T* get_buffer(Boolean orphan = FALSE);
      const T* get_buffer() const;
};


class V2                          // неограниченная последовательность {
   public:
      V2();
      V2(ULong length, T *data, Boolean release = FALSE);
      V2(const V2&);
      ~V2();
      V2 &operator=(const V2&);


      ULong maximum() const;


      void length(ULong);
      ULong length() const;


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


      Boolean release() const;


      void replace(ULong length, T *data,
                   Boolean release = FALSE);


      T* get_buffer(Boolean orphan = FALSE);
      const T* get_buffer() const;
};
```

1.13.2. Использование "release" параметра конструктора

Рассмотрим следующий пример:

```
// IDL

typedef sequence<string, 3> StringSeq;


// C++

char *static_arr[] = {"one", "two", "three"};
char **dyn_arr = StringSeq::allocbuf(3);
dyn_arr[0] = string_dup("one");
dyn_arr[1] = string_dup("two");
```

```
dyn_arr[2] = string_dup("three");


StringSeq seq1(3, static_arr);
StringSeq seq2(3, dyn_arr, TRUE);


seq1[1] = "2";                  // не освобождается, не копируется
char *str = string_dup("2");
seq2[1] = str;                  // освобождается старая память, не копируется
```

В данном примере и **seq1**, и **seq2** оба построены, используя указанные пользователем данные, но только **seq2** указано взять управление над памятью пользователя (потому что параметр **release = TRUE** в его конструкторе). Когда встречается присваивание в **seq1[1]**, правая часть не копируется и ничего не осовбаждается, потому что последовательность не управляет памятью пользователя. Когда встречается присваивание в **seq2[1]**, однако, старые пользовательские данные должны быть удалены перед тем, как выполнится присваивание правой части, так как **seq2** управляет памятью пользователя. Когда seq2 выйдет за границы видимости, она вызовет **string_free** для каждого из своих элементов и затем вызовет freebuf над буфером, заданным в ее конструкторе.

Когда флаг **release** установлен в **TRUE** и тип элементов последовательности либо строка, либо объектная ссылка, последовательность будет индивидуально освобождать каждый элемент перед освобождением содержимого буфера. Она будет освобождать строки, используя **string_free**, а объектные ссылки - используя **release** функцию из пространства имен **CORBA**.

В общем, присваивание в элемент последовательности никогда не имеет места через **operator[]** , за исключением **release=TRUE**, по причине возможности ошибок управления памятью. В частности, последовательность, построенная с **release=FALSE** никогда не может быть передана как **inout** параметр, потому что предыдущие версии этой спецификации не предоставляют смысла для вызванного (callee) для определения установки флага **release** последовательности, и поэтому вызванный всегда имеет в виду, что **release** был установлен в **TRUE**. Код, который создает последовательность с **release=FALSE** и затем знающе и корректно управляет ею в этом состоянии, как показано с seq1 в примере выше, является допустимым, но требуется всегда соблюдать осторожность, чтобы избежать утечки памяти при этих условиях.

Вместе с другими **out** и возвращаемыми значениями, **out** и возвращаемые последовательности не должны быть присвоены вызывающим без начального копирования их.

Когда последовательность построена с **release=TRUE**, соответствующее приложение не будет делать предположений о продолжительности времени жизни буфера данных, переданного в конструктор, так как соответствующая реализация последовательности свободна копировать буфер и немедленно освобождать исходный указатель.

1.13.3. Дополнительные функции управления памятью

Реализации ORB, связанные с взаимодействием одиночных процессов с отображением на C, могут переопределять **operator new()** и **operator delete()** для последовательностей так, что динамическое выделение памяти будет использовать такой же механизм, как и функции динамического выделения памяти языка C. Будут ли эти операторы переопределены реализацией или нет, соответствующие программы используют **new** для динамического выделения последовательностей и **delete** для освобождения их.

Последовательности также предоставляют дополнительные функции управления памятью для их буферов. Для последовательности типа T предоставлены следующие статичные внутренние функции в публичном интерфейсе класса последовательности:

```
// C++
```

```
static T *allocbuf(ULong nelems);
static void freebuf(T *);
```

Функция **allocbuf** выделяет вектор элементов **T**, который может быть передан в **T *data** конструктор и в функцию **replace()**. Длина вектора задается с помощью **nelems** аргумента функции. Функция **allocbuf** инициализирует каждый элемент, используя его конструктор по умолчанию, за исключением строк и широких строк, которые инициализируются указателями на пустую строку, и объектных ссылок, которые инициализируются пустыми (nil) объектными ссылками подходящего типа. Нулевой (null) указатель возвращается, если по какой-либо причине **allocbuf** не может выделить требуемый вектор.

Векторы, выделенные с помощью **allocbuf**, должны быть освобождены, используя функцию **freebuf**. Функция **freebuf** гарантирует, что будет вызван деструктор для каждого элемента перед тем, как буфер будет уничтожен, за исключением элементов строк и широких строк, которые освобождаются, используя **string_free()** и **wstring_free()** соответственно, а также элементов объектных ссылок, которые освобождаются, используя **CORBA::release()**. Функция freebuf будет игнорировать пустые указатели, переданные ей. Ни **allocbuf**, ни **freebuf** не могут возбуждать CORBA исключения.

1.13.4. Типы T_var и T_out для последовательностей

В дополнение в обыкновенным операциям, определенным для типов **T_var** и **T_out**, **T_var** и **T_out** для типа последовательности также поддерживает переопределенный **operator[],** который переадресует запросы к **operator[**] последовательности, лежащей в основе. Это оператор индексирования должен иметь такой же возвращаемый тип, что и связанный оператор типа последовательности.

1.14. Отображение для Типов Массивов

Массивы отображаются в соответствующие описания C++ массивов, которые позволяют описание статично инициализированных данных, используя массив. Если элемент массива - это строка, широкая строка или объектная ссылка, тогда отображение использует тот же тип, что и для полей структуры. То есть, конструктор по умолчанию для элементов строк и широких строк инициализирует их в пустую строку (**""** и **""L** соответственно) и присваивание элементу массива, который является строкой, широкой строкой или объектной ссылкой, будет освобождать память, связанную со старым значением.

```
// IDL

typedef float F[10];
typedef string V[10];
typedef string M[1][2][3];
void op(out F p1, out V p2, out M p3);


// C++

typedef Float F[10];
typedef ... V[10];      // лежащий в основе тип не показан, потому что
typedef ... M[1][2][3]; // он является реализационно-зависимым
F f1; F_var f2;
V v1; V_var v2;
M m1; M_var m2;
f(f2, v2, m2);
f1[0] = f2[1];
v1[1] = v2[1];              // освобождает старую память, копирует
m1[0][1][2] = m2[0][1][2];  // освобождает старую память, копирует
```

В вышеприведенном примере последние два присваивания приводят к тому, что память, связанная со старым значением левой стороны, будет автоматически освобождена перед тем, как значение правой стороны будет скопировано.

Как показано в Таблице 1-3, out и возвращаемые массивы обрабатываются через указатель на *кусок (slice)* массива, где кусок - это массив со всеми измерениями исходного массива, за исключением первого. Для пригодности для приложения описания типов кусков, отображение также предоставляет определение типа (typedef) для каждого типа куска массива. Имя typedef для куска состоит из имени типа массива и добавленнного суффикса "_slice". Например:

```
// IDL

typedef long LongArray[4][5];


// C++

typedef Long LongArray[4][5];
typedef Long LongArray_slice[5];
```

Оба типа, и **T_var**, и **T_out**, для массива должны переопределить **operator[]** вместо **operator->**. Использование кусков массивов также означает ,что типы **T_var** и **T_out** для массива должны иметь конструктор и оператор присваивания, каждый из которых берет указатель на кусок массива как параметр, а не **T***. **T_var** для предыдущего примера будет:

```
// C++

class LongArray_var
{
   public:
      LongArray_var();
      LongArray_var(LongArray_slice*);
      LongArray_var(const LongArray_var &);
      ~LongArray_var();
      LongArray_var &operator=(LongArray_slice*);
      LongArray_var &operator=(const LongArray_var &);


      LongArray_slice &operator[](ULong index);
      const LongArray_slice &operator[](Ulong index) const;


      const LongArray_slice* in() const;
      LongArray_slice* inout();
      LongArray_slice* out();
      LongArray_slice* _retn();


      // другие операторы преобразования для поддержки
      // передачи параметра
};
```

Так как массивы отображаются в обычные C++ массивы, они представляют особые проблемы для безопасного по типу отображения **any**, описанного в "Отображении для Типа Any". Для содействия их использованию с отображением any, соответствующая реализация должна также предоставлять для каждого типа массива отдельный C++ тип, чье имя состоит из имени массива с последующим суффиксом **_forany**. Эти типы должны быть отдельными так, чтобы позволить переопределять их функции. Подобно **Array_var** типам, типы **Array_forany** позволяют доступ к лежащему в основе типу массива, но в отличие от **Array_var**, тип **Array_forany** не удаляет память лежащего в основе массива при совем уничтожении. Это потому, что отображение **Any** сохраняет владение памятью.

Интерфейс типа **Array_forany** идентичен интерфейсу типа **Array_var**, но он может не быть реализован как typedef в **Array_var** типе соответствующей реализацией, так как он должен быть отличным от других типов для целей переопределения функций. Также, конструктор **Array_forany**, получающий параметр **Array_slice***, также получает параметр **Boolean** *nocopy*, который по умолчанию равен **FALSE**.

```
// C++

class Array_forany
{
   public:
      Array_forany(Array_slice*, Boolean nocopy = FALSE);
```

```
                    ...
};
```

Флаг *nocopy* позволяет некопирующую вставку **Array_slice*** в **Any**.

Каждый тип **Array_forany** должен быть определен на том же уровне вложенности, что и его тип **Array**.

Для динамического выделения массивов соответствующие программы должны использовать специальные функции, определенные в той же области видимости, что и тип массива. Для массива **T** следующие функции будут доступны в сосответствующей программе:

```
// C++

T_slice *T_alloc();
T_slice *T_dup(const T_slice*);
void T_copy(T_slice* to, const T_slice* from);
void T_free(T_slice *);
```

Функция **T_alloc** динамически выделяет массив или возвращает нулевой указатель, если она не смогла выполнить выделение. Функция **T_dup** динамически выделяет новый массив той же длины, как и ее аргумент-массив, копирует каждый элемент массива аргумента в новый массив и возвращает указатель на новый массив. Если выделение будет неуспешным, возвращается нулевой указатель. Функция **T_copy** копирует содержимое массива *from* в массив *to*. Если какой-либо аргумент является нулевым указателем, **T_copy** не делает попытки копирования и в результате ничего не выполняет. Функция **T_free** освобождает массив, который был выделен с помощью **T_alloc** или **T_dup**. Передача нулевого указателя в **T_free** допустима, и в результате ничего не произойдет. Функции **T_alloc, T_dup** и **T_free** позволяют реализациям ORB употреблять механизмы управления памятью для типов массивов, если необходимо, без принуждения их замещать глобальные операторы **operator new** и **operator new[]**.

Функции **T_alloc, T_dup, T_copy** и **T_free** не могут возбуждать CORBA исключения.

1.15. Отображение для Определений Типов (typedef)

Определение типа (typedef) создет псевдоним (алиас) для типа. Если исходный тип отображается в некоторые типы в C++, тогда определение типа создает соответствующие псевдонимы для каждого типа. Нижеследующий пример иллюстрирует отображение.

```
// IDL

typedef long T;
interface A1;
typedef A1 A2;
typedef sequence<long> S1;
typedef S1 S2;


// C++

typedef Long T;


// ...definitions for A1...



typedef A1 A2;
typedef A1_ptr A2_ptr;
typedef A1_var A2_var;
```

```
// ...definitions for S1...

class S1 { ... };


typedef S1 S2;
typedef S1_var S2_var;
```

Для определения типа (typedef) IDL типа, который отображается в несколько C++ типов, таких как массивы, определение типа отображается во все подобные C++ типы и функции, которые требует его базовый тип. Например:

```
// IDL

typedef long array[10];
typedef array another_array;


// C++

// ...C++ код для массива не показан...

typedef array another_array;
typedef array_var another_array_var;
typedef array_slice another_array_slice;
typedef array_forany another_array_forany;


inline another_array_slice *another_array_alloc() {
     return array_alloc();
}


inline another_array_slice*

another_array_dup(another_array_slice *a) {
     return array_dup(a);
}


inline void

another_array_copy(another_array_slice* to,
      const another_array_slice* from)
{
      array_copy(to, from);
}


inline void another_array_free(another_array_slice *a) {
      array_free(a);
}
```

1.16. Отображение для Типа Any

C++ отображение для OMG IDL типа any должно удовлетворять двум разным требованиям:

- Обработка C++ типов в безопасной по типу манере.
- Обработка значений, чьи типы не являются известными во время компилирования реализации.

Первый элемент охватывает наиболее обычное использование типа **any** - преобразование типизированных значений в **any** и из него. Второй элемент охватывает ситуации такие, когда возникает прием запроса или ответ, включающий any, который хранит данные типа, неизвестного принимающему, когда запрос был создан с помощью C++ компилятора.

1.16.1. Обработка Типизированных Значений

Для уменьшения шансов создания **any** с ошибочным **TypeCode** и значением, применяется C++ средство перегрузки (overload) функции. Конкретно, для каждого отличного типа в OMG IDL спецификации каждой реализацией ORB предоставляются перегруженные функции для вставки и извлечения значений этого типа. Перегруженные операторы используются для этих функций так, чтобы полностью обойти засорение пространства имен. Природа этих функций, которые детально описаны ниже, в том, что соответствующий **TypeCode** применяется C++ типом значения, которое будет вставлено или извлечено из **any**.

Так как безопасный по типу интерфейс **any**, описанный ниже, основан на перегрузке функций C++, он требует, чтобы были отдельные C++ типы, сгенерированные из OMG IDL спецификаций. Однако, существуют специальные случаи, в которых это требование не встречается:

- Как описано в разделе 1.5. "Отображение для Основных Типов Данных", **boolean, octet, char и wchar** OMG IDL типы не требуется отображать в отдельные C++ типы, что означает, что отдельные способы отличия их от всех остальных типов для цели перегрузки функций являются необходимыми. Способы отличия этих типов от всех остальных описаны в разделе 1.16.4.

- Так как все строки и широкие строки отображаются в **char*** и **WChar*** соответственно, независимо от того, являются ли они ограниченными или неограниченными, необходим другой способ создания или установки **any** со значением ограниченной строки или широкой строки. Это описано в разделе 1.16.4.

- В C++ массивы в списке аргументов функции преобразуются в указатели на их первые элементы. Это означает, что перегрузка функций не может быть использована для различия между массивами разной длины. Способ создания или установки any, когда идет работа с массивами, описывается выше в разделе 1.14.

1.16.2. Вставка в any

Чтобыы позволить устанавливать значение в **any** в безопасном по типу виде, реализация ORB должна предоставлять следующую перегруженную операторную функцию для каждого отдельного OMG IDL типа **T**.

```
// C++

void operator<<=(Any&, T);
```

Данная сигнатура функции достаточна для типов, которые нормально передаются по значению:

- Short, UShort, Long, ULong, LongLong, ULongLong, Float, Double, LongDouble
- Перечисления
- Неограниченные строки и широкие строки (**char*** и **WChar***, передаваемые по значению)
- Объектные ссылки (**T_ptr**)
- Указатели на типы значений (valuetype) (**T***)

Для значений типа **T**, которые являются слишком большими, чтобы быть переданными по значению эффективно, такие как структуры, объединения, последовательности, **Any** и исключения, предоставляются две формы функции вставки.

```
// C++

void operator<<=(Any&, const T&); // копирующая форма
void operator<<=(Any&, T*);       // некопирующая форма
```

Заметим, что копирующая форма почто совершенно эквивалентна показанной первой форме, насколько это касается вызывающего.

Эти операторы "сдвиг-влево-равно" ("left-shift-assign") используются для вставки типизированного значения в **any**.

```
// C++

Long value = 42;
Any a;
a <<= value;
```

В данном случае версия **operator<<=** , переопределенная для типа **Long**, должна быть способна правильно установить и значение, и **TypeCode** для переменной **any**.

Установка значения в **any**, используя **operator<<=**, означает, что:

- Для копирующей версии **operator<<=** время жизни значения в **any** является независимым от времени жизни значения, переданного в **operator<<=**. Реализация **any** может не хранить свое значение как ссылку или указатель на значение, переданное в **operator<<=**.
- Для некопирующей версии **operator<<=** вставляемый **T*** принимается **any**. Вызывающий не может использовать **T*** для доступа к указываемым данным после вставки, так как **any** присваивает владение им и может немедленно скопировать указываемые данные и удалить оригинал.
- И в копирующей, и в некопирующей версиях **operator<<=** любое предыдущее значение, хранимое **Any**, должным образом освобождается. К примеру, если был вызван конструктор **Any(TypeCode_ptr,void*,TRUE)** для создания **Any**, **Any** является ответственным за освобождение памяти, указываемой по **void***, перед копированием нового значения.

Копирующая вставка типа строки или типа широкой строки порождает вызов одной из следующих функций:

```
// C++

void operator<<=(Any&, const char*);
void operator<<=(Any&, const WChar*);
```

Так как все строковые типы отображаются в **char***, и все типы широких строк отображаются в **WChar***, эти функции вставки полагают, что значения, которые будут вставлены, являются неограниченными. Раздел 1.16.4. описывает, как ограниченные строки и ограниченные широкие строки могут быть корректно вставлены в **Any**. Заметим, что вставка широких строк в такой манере основана на стандарте C++, в котором **wchar_t** - это отдельный тип. Код, который должен быть переносимым между стандартом и старыми C++ компиляторами, должен использовать вспомогательный тип **Any::from_wstring**. Некопирующая вставка и ограниченных, и неограниченных строк может быть достигнута, используя вспомогательный тип **Any::from_string**. Подобным же образом, некопирующая вставка ограниченных или неограниченных широких строк может быть достигнута, используя вспомогательный тип **Any::from_wstring**. Оба этих вспомогательных типа описаны в разделе 1.16.4.

Заметим, что следующий код имеет неопределенное поведение в нестандартных C++ средах:

```
// C++

Any a = ...;
WChar wc;
a >>= wc;     // неопределенное поведение
```

Этот код может ошибочно извлечь целый тип в средах, где **wchar_t** не является отдельным типом.

Так как типы значений (**valuetype**) могут быть легально представлены, используя нулевые указатели, то согласующееся приложение может вставлять нулевой **valuetype** указатель в **Any**.

Безопасная по типу вставка массивов использует типы **Array_forany**, описанные в разделе 1.14. Соответствующие реализации должны предоставлять версию **operator<<=**, перегруженного для каждого типа **Array_forany**. Например:

```
// IDL

typedef long LongArray[4][5];


// C++

typedef Long LongArray[4][5];
typedef Long LongArray_slice[5];
class LongArray_forany { ... };


void operator<<=(Any &, const LongArray_forany &);
```

Типы **Array_forany** всегда передаются в **operator<<=** по ссылке на константу. Флаг *nocopy* в конструкторе **Array_forany** используется для управления, будет ли вставляемое значение скопировано (*nocopy* == **FALSE**) или принято (*nocopy* == **TRUE**). Так как флаг *nocopy* по умолчанию равен **FALSE**, по умолчанию выполняется копирующая вставка.

По причине неоднозначности между массивами из **T** и из **T***, настойчиво рекомендуется, чтобы переносимый код явно[1] использовал соответствующий тип **Array_forany**, когда вставляется массив в **any**:

```
// IDL

struct S {... };
typedef S SA[5];


// C++

struct S { ... };
typedef S SA[5];
typedef S SA_slice;
class SA_forany { ... };


SA s;
// ...инициализация s...

Any a;
a <<= s;                  // строка 1

a <<= SA_forany(s);       // строка 2
```

Строка 1 выражается в вызове некопирующего **operator<<=(Any&, S*)** по причине преобразования типа массива SA в указатель на его первый элемент, вместо вызова копирующего оператора вставки **SA_forany**. Строка 2 явно создает **SA_forany** тип и поэтому в результате будет вызван желаемый оператор вставки.

Некопирующая версия **operator<<=** для объектных ссылок получает адрес типа **T_ptr**.

```
// IDL

interface T { ... };


// C++

void operator<<=(Any&, T_ptr);   // копирующая
void operator<<=(Any&, T_ptr*);  // некопирующая
```

Некопирующая вставка объектной ссылки потребляет объектную ссылку, указанную с помощью **T_ptr***, поэтому после вставки вызывающий не может получить доступ к объекту, ссылаемому **T_ptr**, так как **any** может сдублировать и затем немедленно освободить исходную объектную ссылку. Вызывающий сохраняет владение памятью для самого **T_ptr**.

Некопирующая версия **operator<<=** для типов значений (**valuetype**'ов) получает адрес **T*** указательного типа.

```
// IDL

valuetype T { ... };


// C++

void operator<<=(Any&, T*);    // копирующая
void operator<<=(Any&, T**);   // некопирующая
```

Некопирующая вставка **valuetype** потребляет **valuetype**, на который указывает указатель, указанный **T****. После вставки вызывающий не может получить доступ к экземпляру **valuetype**, указанному указателем, указанным **T***. Вызывающий сохраняет владение памятью для самого **T***.

В общем, копирующие версии **operator<<=** также поддерживаются над типом **Any_var**. Заметим, что в соответствии с операторами преобразования, которые преобразуют **Any_var** в **Any&** для передачи параметров, только те **operator<<=** функции, которые определены как членские функции **any**, необходимо явно определить для **Any_var**.

---

1. Реализатор отображения может использовать новое ключевое слово C++ "explicit" для предотвращения неявных преоразований в конструкторе Array_forany, но эта возможность еще не широко доступна в текущих компиляторах C++.

1.16.3. Извлечение из any

Чтобы позволить безопасное по типу получение значения из any, отображение предоставляет следующие операторы для каждого OMG IDL типа **T**.

```
// C++

Boolean operator>>=(const Any&, T&);
```

Эта сигнатура функции удовлетворяет примитивным типам, которые обычно передаются по значению. Для значений типа **T**, которые слишком большие, чтобы быть переданными по значению эффективно (такие как структуры, объединения, последовательности, Any, типы значений и исключения), прототип такой функции может быть следующим:

```
// C++

Boolean operator>>=(const Any&, T*&);       // запрещенный
Boolean operator>>=(const Any&, const T*&);
```

Неконстантная версия оператора будет запрещена в будущей версии отображения и не должна использоваться.

Первая форма этой функции используется только для следующих типов:

- **Short, UShort, Long, ULong, LongLong, ULongLong, Float, Double, LongDouble**

- Перечисления

- Неограниченные строки и широкие строки (**const char*** и **const WChar***, переданные по ссылке, т.е. **const char*&** и **const WChar*&**)

- Объектные ссылки (**T_ptr**)

Для всех остальных типов используется вторая форма функция.

Все версии **operator>>=**, реализованные как членские функции класса **Any**, как например для примитивных типов, маркируются как **const**.

Этот "сдвиг-вправо-равно" оператор используется для извлечения типизированного значения из **any**, как например:

```
// C++

Long value;
Any a;
a <<= Long(42);
if (a >>= value) {
         // ... использовать value ...

}
```

В данном случае версия **operator>>=** для типа **Long** должна быть способна определить, действительно ли **Any** содержит значение типа **Long**, и если это так, скопировать его значение в ссылку на переменную, предоставленную вызывающим, и возвратить **TRUE**. Если **Any** не содержит значения типа **Long**, значение ссылки переменной вызывающего не будет изменено и **operator>>=** возвратит **FALSE**.

Для непримитивных типов, таких как структура, объединение, последовательность, исключение и Any, извлечение делается в указатель на **const** (типы значений (**valuetypes**) извлекаются в неконстантный указатель, потому что операции **valuetype** не поддерживают **const**). Например, рассмотрим следующую IDL структуру:

```
// IDL

struct MyStruct {
                 long lmem;
                 short smem;
};
```

Такая структура может быть извлечена из **any** следующим образом:

```
// C++

Any a;
// ... a - это как-то заданное значение типа MyStruct ...

const MyStruct *struct_ptr;
if (a >>= struct_ptr) {
        // ... использовать значение ...

}
```

Если извлечение успешно, указатель вызывающего будет указывать на память, управляемую **any**, и **operator>>=** возвратит **TRUE**. Вызывающий не должен пытаться **удалить** (**delete**) или другим образом освободить эту память. Вызывающий также не может использовать память после того, как содержимое переменной **any** будет заменено через присваивание, вставку или функцию **replace**, или после того, как переменная **any** будет уничтожена. Требуется соблюдать осторожность, чтобы избежать использования типов **T_var** с этими операторами, так как они попытаются присвоить ответственность за удаление памяти, отведенной **any**.

Если извлечение не является успешным, значение указателя вызывающего устанавливается равным нулевому указателю, и **operator>>=** возвращает **FALSE**. Заметим однако, что так как типы значений (**valuetype**) могут легально быть представленны как нулевые указатели, то указатель на **T**, извлеченный из **Any**, где **T** является **valuetype**, может быть нулевым даже тогда, когда извлечение прошло успешно, если **Any** содержал нулевой **valuetype** указатель.

Правильное извлечение типов массивов зависит от типов **Array_forany**, описанных в разделе 1.14.

```
// IDL

typedef long A[20];
typedef A B[30][40][50];


// C++

typedef Long A[20];
typedef Long A_slice;
class A_forany { ... };
typedef A B[30][40][50];
typedef A B_slice[40][50];
class B_forany { ... };


Boolean operator>>=(const Any &, A_forany&);  // для типа A

Boolean operator>>=(const Any &, B_forany&);  // для типа B
```

Array_forany типы всегда передаются в **operator>>=** по ссылке.

Для строк, широких строк и массивов приложения являются ответственными за проверку кода типа (**TypeCode)** для **any**, чтобы быть уверенными, что они не перейдут границ объекта массива, строки или широкой строки, при использовании извлеченного значения.

operator>>= также поддерживается над **Any_var** типом. Заметим, что вследствие операторов преобразования, которые преобразуют **Any_var** в **const Any&** для передачи параметров, необходимо явно определить для **Any_var** только те функции **operator>>=**, которые определены как членские функции any.

1.16.4. Различение boolean, octet, char, wchar, ограниченной строки и ограниченной широкой строки

Так как **boolean, octet, char** и **wchar** OMG IDL типы не требуют отображения в отдельные C++ типы, другой способ различения их от всех остальных типов является необходимым так, чтобы они могли быть использованы с безопасным по типу интерфейсом **any.** Подобным же образом, так как и ограниченные, и неограниченные строки отображаются в **char***, ограниченные и неограниченные широкие строки отображаются в **WChar***, и все типы с фиксированной точкой отображаются в класс **Fixed**, должен быть предоставлен другой способ их различения. Это сделано с помощью введения нескольких новых вспомогательных типов, вложенных в интерфейс класса **any**. К примеру, это может быть выполнено как показано далее.

```
// C++

class Any
{
   public:
      // специальные вспомогательные типы, необходимые для вставки
      // boolean, octet, char и ограниченной строки
      struct from_boolean {
         from_boolean(Boolean b) : val(b) {}
         Boolean val;
      };
      struct from_octet {
         from_octet(Octet o) : val(o) {}
         Octet val;
      };
      struct from_char {
         from_char(Char c) : val(c) {}
         Char val;
      };
      struct from_wchar {
         from_wchar(WChar wc) : val(wc) {}
         WChar val;
      };
      struct from_string {
         from_string(char* s, ULong b,
                     Boolean n = FALSE) :
            val(s), bound(b), nocopy(n) {}
         from_string(const char* s, ULong b) :
            val (const_cast<char*>(s)), bound(b),
            nocopy (0) {}
         char *val;
         ULong bound;
         Boolean nocopy;
      };
      struct from_wstring {
         from_wstring(WChar* s, ULong b,
                      Boolean n = FALSE) :
            val(s), bound(b), nocopy(n) {}
         from_wstring(const WChar* s, ULong b) :
            val(const_cast<WChar*>(s)), bound(b),
            nocopy(0) {}
         WChar *val;
         ULong bound;
         Boolean nocopy;
      };
      struct from_fixed {
         from_fixed(const Fixed& f, UShort d, UShort s)

                  : val(f), digits(d), scale(s) {}
         const Fixed& val;
         UShort digits;
         UShort scale;
      };


      void operator<<=(from_boolean);
      void operator<<=(from_char);
      void operator<<=(from_wchar);
      void operator<<=(from_octet);
      void operator<<=(from_string);
      void operator<<=(from_wstring);
      void operator<<=(from_fixed);


      Boolean operator>>=(to_fixed) const;


      // другие public Any детали опущены
   private:
      // эти функции являются личными (private) и нереализованными
      // их сокрытие вызывает ошибки времени компиляции для
      // unsigned char

      void operator<<=(unsigned char);
      Boolean operator>>=(unsigned char &) const;
};
```

Реализация ORB предоставляет перегруженные функции **operator<<=** и **operator>>=** для этих специальных вспомогательных типов. Эти вспомогательные типы используются как показано ниже.

```
// C++

Boolean b = TRUE;
Any any;
any <<= Any::from_boolean(b);
// ...

if (any >>= Any::to_boolean(b)) {
   // ...any содержит Boolean...

}
const char* p = "bounded";
any <<= Any::from_string(p, 8);
// ...

if (any >>= Any::to_string(p, 8)) {
   // ...any содержит string<8>...

}
```

Значение границы 0 (нуль), переданное в соответствующий вспомогательный тип, указывает на неограниченную строку или широкую строку.

Для некопирующей вставки ограниченной или неограниченной строки в **any** флаг **nocopy** в конструкторе **from_string** должен быть установлен в **TRUE**.

```
// C++

char* p = string_alloc(8);
// ...инициализация строки p...

any <<= Any::from_string(p, 8, 1);  // any потребляет p
```

Подобные же правила применяются для ограниченных и неограниченных широких строк и вспомогательного типа **from_wstring**. Заметим, что неконстантные версии конструкторов **to_string** и **to_wstring** будут убраны в будущей версии отображения и не должны использоваться.

Принимая, что **boolean**, **char** и **octet** все отображаются C++ типом **unsigned char**, личные и нереализованные функции **operator<<=** и **operator>>=** для **unsigned char** вызовут ошибку времени испольнения, если будет сделана попытка прямой вставки или исключения любого из типов **boolean**, **char** или **octet**.

```
// C++

Octet oct = 040;
Any any;
any <<= oct;                  // эта строка не будет компилироваться,
any <<= Any::from_octet(oct); // а эта будет
```

Важно заметить, что предыдущий пример является только одной из возможных реализаций для этих впомогательных типов, но не обязательной. Другие подходящие реализации возможны, как например предоставление через in-line статичные членские функции **any**, если **boolean, char** и **octet** фактически отображаются в отдельные C++ типы. Однако, все подходящие реализации C++ отображения должны предоставлять эти вспомогательные типы в целях переносимости.

В стандартных C++ средах реализация отображения должна объявлять конструкторы для вспомогательных классов **from_** и **to_** как **explicit**. Это предотвращает нежелательные преобразования через временные элементы.

1.16.5. Расширение до Объекта

Иногда желательно извлечь объектную ссылку из **Any**, как базовый тип **Object**. Это может быть выполнено, используя вспомогательный тип, подобный тем, что требуются для извлечения **Boolean, Char** и **Octet**:

```
// C++

class Any
{
   public:
      ...

      struct to_object {
         to_object(Object_out obj) : ref(obj) {}
         Object_ptr &ref;
      };
      Boolean operator>>=(to_object) const;
      ...

};
```

Вспомогательный тип **to_object** используется для извлечения объектной ссылки из **Any** как базового типа **Object**. Если Any содержит значение объектной ссылки, как показано его кодом типа (**TypeCode**), функция извлечения **operator>>=(to_object)** явно расширяет содержащуюся в нем объектную ссылку до **Object** и возвращает **TRUE**, а иначе возвращает **FALSE**. Это только функция извлечения объектной ссылки, которая выполняет расширение над извлекаемой объектной ссылкой. В отличие от обычного извлечения объектной ссылки, время жизни объектной ссылки, извлеченной с использованием **to_object**, является независимым от времени жизни той ссылки, которая в **Any**, и поэтому ответственность за вызов **release** для нее берет на себя вызывающий.

1.16.6. Расширение до Абстрактного Интерфейса

Тип **CORBA::Any::to_abstract_base** позволяет содержимое **Any** извлечь как **AbstractBase**, если сущность, хранящаяся в Any, является типа объектной ссылки или типа значения, явно или неявно производного от базового класса **AbstractBase**. Тип **to_abstract_base** показан ниже:

```
// C++

class Any {
   public:
      ...

      struct to_abstract_base {
         to_abstract_base(AbstractBase_ptr& base)

            : ref(base) {}
         AbstractBase_ptr& ref;
      };


      Boolean operator>>=(to_abstract_base val) const;
      ...

};
```

Описание **AbstractBase** смотри в разделе 1.18.1.

1.16.7. Обработка Нетипизированных Значений

При некоторых условиях безопасного по типу интерфейса к **Any** недостаточно. Пример - это ситуация, в которой типы данных читаются из файла в бинарной форме и используются для создания значений типа **Any**. Для таких случаев класс **Any** предоставляет конструктор с явным типом кода (**TypeCode**) и общим указателем:

```
// C++

Any(TypeCode_ptr tc, void *value, Boolean release = FALSE);
```

Конструктор является ответственным за дублирование данной ссылки на псевдообъект **TypeCode**. Если параметр **release** равен **TRUE**, тогда объект **Any** присваивает владение памятью, указанной с помощью параметра **value**. Соответствующее приложение не сможет сделать предположений о продолжительности времени жизни параметра **value** после того, как он был передан в **Any** с **release=TRUE**, так как соответствующей реализации Any позволено копировать параметр **value** и немедленно освободить исходный указатель. Если параметр **release** равен **FALSE** (значение по умолчанию), тогда объект **Any** поручает вызывающему управлять памятью, указанной с помощью **value**. Параметр **value** может быть нулевым указателем.

Класс **Any** также определяет три небезопасные операции:

```
// C++

void replace(

      TypeCode_ptr,
      void *value,
      Boolean release = FALSE

);
TypeCode_ptr type() const;
const void *value() const;
```

Функция **replace** предназначена, чтобы быть использованной с типами, которые не могут использоваться с безопасным по типу интерфейсом вставки, и поэтому она подобна конструктору, описанному выше. Существующий код типа **TypeCode** уничтожается и память значения освобождается, если необходимо. Параметр функции **TypeCode** дублируется. Если параметр **release** равен **TRUE**, тогда объект **Any** получает владение над памятью, указанной с помощью параметра **value**. Соответствующее приложение не может сделать предположений о продолжительности времени жизни параметра **value**, как только он будет передан в функцию **Any::replace** с флагом **release=TRUE**, так как соответствующей реализации **Any** позволено копировать параметр **value** и немедленно освобождать исходный указатель. Если параметр **release** равен **FALSE** (вариант по умолчанию), тогда объект **Any** поручает вызывающему управлять памятью, занятой значением. Параметр **value** функции **replace** может быть нулевым указателем.

Для реализаций C++ отображения, которые используют параметры **Environment** для передачи информации об исключении, дефолтный аргумент **release** может быть симитирован с помощью предоставления двух перегруженных функций **replace**, одна из которых имеет параметр **release** без значения по умолчанию, а другая не имеет параметра **release**. Вторая функция просто вызывает первую с параметром **release**, установленным в **FALSE**.

Заметим, что ни конструктор, показанный выше, ни функции **replace** не являются безопасными по типу. В частности, ни компилятором, ни во время работы не дается гарантий в согласованности между **TypeCode** и фактическим типом аргумента **void***. Поведение реализации ORB, когда представлен **Any**, построенный с ошибочными кодом типа и значением, не определено.

Аксессорная функция **type** возвращает **TypeCode_ptr**, ссылку на псевдообъект **TypeCode**, связанный с **Any**. Подобно всем возвращаемым значениям объектных ссылок, вызывающий должен уничтожить ссылку, когда она более не требуется, или присвоить ее переменной **TypeCode_var** для автоматического управления.

Функция **value** возвращает указатель на данные, хранимые в **Any**. Если **Any** не имеет связанного значения, функция **value** возвращает нулевой указатель. Тип, к которому может быть приведен **void***, возвращаемый функцией **value**, зависит от реализации ORB, следовательно, использование функции **value** не является переносимым среди реализаций ORB и ее использование запрещено. Переносимые программы должны использовать интерфейс **DynAny** для доступа и изменения содержимого **Any** в тех случаях, где операторы вставки и извлечения не являются достаточными. Заметим, что реализациям ORB позволено давать строгие гарантии о **void***, возвращаемом из функции **value**, если так желательно.

1.16.8. Замещение кода типа (TypeCode)

Так как C++ описания типов (**typedef**) являются только псевдонимами и не определяют отдельных типов, то вставка типа с **tk_alias** кодом типа **(TypeCode)** в **Any** с сохранением этого кода типа не является возможной. Например:

```
// IDL

typedef long LongType;


// C++

Any any;
LongType val = 1234;
any <<= val;
TypeCode_var tc = any.type();
assert(tc->kind() == tk_alias); // неверно!

assert(tc->kind() == tk_long); //  правильно, OK
```

В данном коде **LongType** является псевдонимом для **CORBA::Long**. Следовательно, когда вставляется значение, стандартный механизм перегрузки C++ вызывает оператор вставки для **CORBA::Long**. Фактически, так как **LongType** - это псевдоним для **CORBA::Long**, перегруженный **operator<<=** для **LongType** в любом случае не может быть сгенерирован.

В случаях, где код типа в **Any** должен быть сохранен как **tk_alias**, приложение может использовать модифицирующую функцию **type** над **Any** для замены его кода типа на эквивалентный. Разберем предыдущий пример:

```
// C++

Any any;
LongType val = 1234;
any <<= val;
any.type(_tc_LongType);           // замена TypeCode

TypeCode_var tc = any.type();
assert(tc->kind() == tk_alias);   // правильно, OK
```

Модификатор **type** вызывает операцию **TypeCode::equivalent** над кодом типа в **Any**, передавая код типа, полученный им как аргумент. Если функция **TypeCode::equivalent** возвращает **TRUE**, модификатор **type** замещает исходный код типа в **Any** на свой аргумент **TypeCode**. Если эти два кода типа не эквивалентны, модификатор **type** возбуждает исключение **BAD_TYPECODE**.

1.16.9. Конструкторы, Деструктор, Оператор Присваивания Any

Конструктор по умолчанию создает **Any** с кодом типа (**TypeCode**) типа **tk_null** и без значения. Конструктор копирования вызывает функцию **_duplicate** над **TypeCode_ptr** своего параметра **Any** и полностью копирует значение параметра. Оператор присваивания освобождает свой имеющийся **TypeCode_ptr** и освобождает память для текущего значения, если необходимо, а затем дублирует **TypeCode_ptr** своего параметра **Any** и полностью копирует значение параметра. Деструктор вызывает **release** над **TypeCode_ptr** и освобождает память для значения, если необходимо.

Реализации ORB, связанные с взаимодействием одиночных процессов с отображением на C, могут перегружать **operator new()** и **operator delete()** для **Any** так, чтобы динамическое выделение использовало такой же механизм, как и функции динамического выделения языка C. Будут ли эти операторы перегружены реализацией или нет, соответствующие программы используют **new** для динамического выделения **any** и **delete** для их освобождения.

1.16.10. Класс Any

Ниже приведено полное описание класса **Any**.

```
class Any
{
   public:
      Any();
      Any(const Any&);
      Any(TypeCode_ptr tc, void *value,
          Boolean release = FALSE);
      ~Any();
      Any &operator=(const Any&);


      // special types needed for boolean, octet, char,
      // and bounded string insertion

      // these are suggested implementations only

      struct from_boolean {
         from_boolean(Boolean b) : val(b) {}
         Boolean val;
      };
      struct from_octet {
         from_octet(Octet o) : val(o) {}
         Octet val;
      };
      struct from_char {
         from_char(Char c) : val(c) {}
         Char val;
      };
      struct from_wchar {
         from_char(WChar c) : val(c) {}
         WChar val;
      };
      struct from_string {
         from_string(char* s, ULong b,
                     Boolean n = FALSE) :
            val(s), bound(b), nocopy(n) {}
         from_string(const char* s, ULong b) :
            val(const_cast<char*>(s)), bound(b),
                nocopy(0) {}
         char *val;
         ULong bound;
         Boolean nocopy;
      };
      struct from_wstring {
         from_wstring(WChar* s, ULong b,
                  Boolean n = FALSE) :
            val(s), bound(b), nocopy(n) {}
         from_wstring(const WChar*, ULong b) :
            val(const_cast<WChar*>(s)), bound(b),
            nocopy(0) {}
         WChar *val;
         ULong bound;
         Boolean nocopy;
      };
```

```
struct from_fixed {
   from_fixed(const Fixed& f, UShort d, UShort s)

      : val(f), digits(d), scale(s) {}
   const Fixed& val;
   UShort digits;
   UShort scale;
};


void operator<<=(from_boolean);
void operator<<=(from_char);
void operator<<=(from_wchar);
void operator<<=(from_octet);
void operator<<=(from_string);
void operator<<=(from_wstring);
void operator<<=(from_fixed);


// special types needed for boolean, octet,
// char extraction

// these are suggested implementations only

struct to_boolean {
   to_boolean(Boolean &b) : ref(b) {}
   Boolean &ref;
};
struct to_char {
   to_char(Char &c) : ref(c) {}
   Char &ref;
};
struct to_wchar {
   to_wchar(WChar &c) : ref(c) {}
   WChar &ref;
};


struct to_octet {
   to_octet(Octet &o) : ref(o) {}
   Octet &ref;
};
struct to_object {
   to_object(Object_out obj) : ref(obj) {}
   Object_ptr &ref;
};
struct to_string {
   to_string(const char *&s, ULong b)

      : val(s), bound(b) {}
   const char *&val;
   ULong bound;


   // the following constructor is deprecated

   to_string(char *&s, ULong b) : val(s), bound(b) {}
};
struct to_wstring {
   to_wstring(const WChar *&s, ULong b)

      : val(s), bound(b) {}
   const WChar *&val;
   ULong bound;


   // the following constructor is deprecated

   to_wstring(WChar *&s, ULong b)

      : val(s), bound(b) {}
};
struct to_fixed {
   to_fixed(Fixed& f, UShort d, UShort s)

      : val(f), digits(d), scale(s) {}
   Fixed& val;
   UShort digits;
   UShort scale;
};
struct to_abstract_base {
   to_abstract_base(AbstractBase_ptr& base)
```

```
            : ref(base) {}
         AbstractBase_ptr& ref;
      };


      Boolean operator>>=(to_boolean) const;
      Boolean operator>>=(to_char) const;
      Boolean operator>>=(to_wchar) const;
      Boolean operator>>=(to_octet) const;
      Boolean operator>>=(to_object) const;
      Boolean operator>>=(to_string) const;
      Boolean operator>>=(to_wstring) const;
      Boolean operator>>=(to_fixed) const;
      Boolean operator>>=(to_abstract_base) const;


      void replace(TypeCode_ptr, void *value,
                   Boolean release = FALSE);


      TypeCode_ptr type() const;
      void type(TypeCode_ptr);
      const void *value() const;


   private:
      // these are hidden and should not be implemented

      // so as to catch erroneous attempts to insert

      // or extract multiple IDL types mapped to unsigned char

      void operator<<=(unsigned char);
      Boolean operator>>=(unsigned char&) const;
};


void operator<<=(Any&, Short);
void operator<<=(Any&, UShort);
void operator<<=(Any&, Long);
void operator<<=(Any&, ULong);
void operator<<=(Any&, Float);
void operator<<=(Any&, Double);
void operator<<=(Any&, LongLong);
void operator<<=(Any&, ULongLong);
void operator<<=(Any&, LongDouble);
void operator<<=(Any&, const Any&);     // copying

void operator<<=(Any&, Any*);           // non-copying

void operator<<=(Any&, const char*);
void operator<<=(Any&, const WChar*);


Boolean operator>>=(const Any&, Short&);
Boolean operator>>=(const Any&, UShort&);
Boolean operator>>=(const Any&, Long&);
Boolean operator>>=(const Any&, ULong&);
Boolean operator>>=(const Any&, Float&);
Boolean operator>>=(const Any&, Double&);
Boolean operator>>=(const Any&, LongLong&);
Boolean operator>>=(const Any&, ULongLong&);
Boolean operator>>=(const Any&, LongDouble&);
Boolean operator>>=(const Any&, const Any*&);
Boolean operator>>=(const Any&, const char*&);
Boolean operator>>=(const Any&, const WChar*&);
```

1.16.11. Класс Any_var

Так как **Any** возвращаются через указатель как **out** и возвращаемые параметры, существует класс **Any_var**, подобный классам **T_var** для объектных ссылок. **Any_var** подчиняются правилам для **T_var** классов, описанным в разделе 1.9, вызывая **delete** над над своим **Any***, когда он выходит за границы видимости или инача уничтожается. Полный интерфейс **Any_var** класса показан ниже.

```
class Any_var
{
```

```
  public:
     Any_var();
     Any_var(Any *a);
     Any_var(const Any_var &a);
     ~Any_var();


     Any_var &operator=(Any *a);
     Any_var &operator=(const Any_var &a);


     Any *operator->();


     const Any& in() const;
     Any& inout();
     Any*& out();
     Any* _retn();


     // other conversion operators for parameter passing

};
```

1.17. Отображение для Valuetype

IDL **valuetype** имеет особенности которые делают отображение на Си++ отличным от других типов IDL. В частности от перспектив приложений всех остальных типов IDL включая другие чистые состояния или чистые интерфейсы, но **valuetype** может включать и то и другое. Поэтому, отображение valuetype на Си++ обязательно более ограничено в термах реализации чем другие части отображения на Си++.

IDL valuetype отображается на Си ++ класс с тем же именем как и **IDL valuetype**. Ётот класс абстрактный базовый класс (**ABC**), с чистым виртуальным аксессором и модификатором функций соответствующих статичным членам **valuetype**, и чистым виртуальным функциям соответствующим операциям **valuetype.**

Си++ класс чье имя сформировано из предваряющей строки "**OBV_**" и имени **valuetyp**е предоставляет реализацию по умолчанию для аксессоров и модификаторов **ABC** базового класса. Разработчик приложения тогда подменяет чистые виртуальные функции соответствующие **valuetype** операциям в конкретном классе производном напрямую или ненапрямую от **OBV_** базового класса.

Приложения отвечают за создание экземпляра **valuetype**, и после мсоздания, они работают с теми экземплярами только через Си++ указатели. В отличии от объектных ссылок, которые отображаются на Си++ типы **_ptr** которые могут быть реализованы иначе чем фактические Си++ указатели или как Си++ объекты типа указатели, "**handles**" на Си++ **valuetype** экземпляры фактические Си++ указатели. Ётo помогает отличать их от объектных ссылок.

Так как **valuetype** поддерживается разделением экземпляров без других сконструированных типов (таких как графы), время жизни экземпляра Си++ **valuetype** управляется через вычисление ссылки. Отличие семантики подсчета объектной ссылки, где ни **duplicate** ни **release р**еально не влияет на реализацию объекта, операции подсчета ссылки для экземпляров Си++ **valuetype** напрямую реализованы этими экземплярами. Вычисление ссылок смешанных классов предоставляются реализацией ORB для использования реализаторов **valuetype.**

Как и для большинства других типов в С++ отображении, каждый **valuetype** также имеет связанный С++ **_var** тип который автоматизирует его вычисление ссылки.

Все **init** инициализаторы описаны для **valuetype** отображаются на чистые виртуальные функции на раздельные абстрактные Си++ классы фабрик. Класс именуется добавлением У**_init**Ф к имени **valuetype** (т.е. тип **A** имеет класс фабрики называющийся **A_init**).

1.17.1 Члены Данных Valuetype

Отображение Си++ для членов данных **valuetype** следует таким же правилам как Си++ отображение для объединений (unions), исключая то, что аксессор и модификатор чисто виртуальные. Внешние (Public) члены состояния отображаются на внешние чистые виртуальные функции аксессоров и модификаторов базовых классов Си++ **valuetype**, и внутренние (private) члены состояния отображаются на защищенные (protected) чистые виртуальные функции аксессоров и модификаторов (так что производные конкретные классы могут иметь доступ к ним). Переносимые приложения которые используюьт классы **OBV_** , включая производные классы типов значений, не будут иметь доступ к фактическим членам данных классов **OBV_** , и реализации ORB могут сделать подобные члены внутренними. Единственное требование к фактическим членам данных в конкретных (не абстрактных) или частично не абстрактных классах таких как **OBV_** класс в том чтобы они были самоуправляемыми так как эти производные классы могут корректно реализовывать копирование без необходимости прямого доступа кним.

Подобно объединениям Си++, функции аксессоров и модификаторов для **valuetype**членов состояния не следует обычным правилам Си++ при передаче. Ёто происходит потому, что они позволяют локальным программам иметь доступ к состояниям хранящимся внутри экземпляра **valuetype**. Функции модификаторы выполняют эквивалент глубокого копирования их параметров, и аксессоры которые возвращают ссылку или указатель на член состояния может быть использован для доступа чтения-записи. Например:

```
// IDL

typedef octet Bytes[64];
struct S { ... };
interface A { ... };


valuetype Val {
   public Val t;
   private long v;
   public Bytes w;
   public string x;
   private S y;
   private A z;
};


// C++

typedef Octet Bytes[64];
typedef Octet Bytes_slice;
...

struct S { ... };


typedef ... A_ptr;


class Val : public virtual ValueBase {
   public:
      ...

      virtual Val* t() const = 0;
      virtual void t(Val*) = 0;


      virtual const Bytes_slice* w() const = 0;
      virtual Bytes_slice* w() = 0;
      virtual void w(const Bytes) = 0;


      virtual const char* x() const = 0;
      virtual void x(char*) = 0;
      virtual void x(const char*) = 0;
      virtual void x(const String_var&) = 0;


   protected:
      virtual Long v() const = 0;
      virtual void v(Long) = 0;
```

```
      virtual const S& y() const = 0;
      virtual S& y() = 0;
      virtual void y(const S&) = 0;


      virtual A_ptr z() const = 0;
      virtual void z(A_ptr) = 0;
   ...
};
```

Следующие правила применимы к функциям аксессора и модификатора показаные в предыдущем примере:

- Функция аксессора **t** не инкрементирует счетчик ссылок возвращаемого **valuetype.** Ёто подразумевает что вызывающая программа не принимает возвращаемое значение.
- Функция модификатора **t** инкрементирует счетчик ссылок его аргумента, затем декрементирует счетчик ссылок члена **t** замещеного перед возвращением.
- Функция модификатора **x(char*)** освобождает старый строковый член и принимает его аргумент.
- Функция модификатора **x(const char*)** освобождает старый строковый член и копирует его аргумент.
- Функция модификатора **x(const String_var&)** освобождает старый строковый член и копирует его аргумент.
- Возвращением ссылки на константу **S**, первая функция аксессора **y** предоставляет доступ для чтения к члену **y**.
- Возвращением ссылки на **S**, вторая функция аксессора **y** предоставляет доступ для чтения и записи к члену **y**.
- Функция модификатора **y** глубоко копирует его аргумент **S.**
- Функция аксессора **z** не вызывает **_duplicate** на объектную ссылку которую возвращает.
  Ёто означает что вызывающая функция **z** не отвечает за вызов **release** на возвращаемое значение.
- Функция модификатора **z** освобождает ее старую объектную ссылку соответствующую члену **z**, когда дублирует аргумент перед возвращением.

1.17.2 Конструкторы, Операторы присваивания, и Деструкторы

Си++ класс **valuetype** определяет защищенный конструктор по умолчанию и защищенный виртуальный деструктор. Конструктор по умолчанию защищенный для разрешения вызывать его только производным экземплярам класса, в то время как деструктор защищенный для предотвращения удаления приложениями указателей на экземпляры значений вместо использования операций счетчика ссылок. Деструктор виртуальный для предоставления правильного удаления производных экземпляров классов значений когда их счетчики ссылок сбрасываются в нуль.

По некоторым причинам, Си++ **OBV_** класс определяет защищенный конструктор по умолчанию который выбирает инициализатор для каждого члена данных **valuetype**, и защищенный конструктор по умолчанию. Параметры конструктора который выбирает инициализатор для каждого члена появляющегося в таком же порядке как появляются члены данных, сверху вниз, в определении **IDL valuetype**, независимо от того открытые они или закрытые (public or private). Все параметры для инициализаторов членов следуют из отображения Си++, правил передачи параметров для **in** аргументов их соответствующих типов.

Переносимые приложения не могут вызывать конструктор копирования класса типа значения или оператора выделения по умолчанию. Соответствующее требуемое значение счетчика ссылок, оператор выделения по умолчанию для класса значения будет закрытым и желательно нереализовано для полного запрещения выделения экземпляров значений.

1.17.3 Операции типов значений

Операции декларированые в типах значений отображаются на открытые чистые виртуальные функции члены в соответствующем Си++ классе **valuetype**. (Статичные члены функций аксессоров и модификаторов не предназначены быть операциями- они имеют отличные от операций правила передачи параметов и они всегда ссылаются на функции аксессора и модификатора). Чистые виртуальные функции члены соответствующие операциям будут декларированы как **const** потому, что в отличии от Си++, IDL не предоставляет пути для различия между операциями которые изменяют состояние объекта и теми которые просто имеют доступ к этому состоянию. В этом случае, подобно выбору сделаному для отображения Си++ для операций описаных в типах **IDL interface**, имеет влияние на правила передачи параметров. Альтернатива, определение всех функций чистых виртуальных членов как **const,** менее желательна потому, что это не позволяет функциям членам наследоваться от классов **interface** для вызова **const** экземпляров значений, с тех пор как все подобные функции члены уже отображены как не **const.**

Си++ сигнатура и правила управление памятью для операций **valuetype** (но не статичных функций членов аксессоров и модификаторов) идентично описаному в Секции 1.22, "Рассмотрение передачи аргументов".

Статичная функция **_downcast** предоставляется каждым классом типа значения для предоставления переносимого пути для приложений для разрушения иерархии наследования Си++. Ёто особенно необходимо после вызова функции **_copy_value**. Если в **_downcast**, передан нулевой указатель, он возвращает нулевой указатель. Иначе, если экземпляр типа значения указаный аргументом этот экземпляр класса типа значения будет понижен, возвращается указатель на пониженный тип класса. Если экземпляр типа значения указаный аргументом не экземпляр типа значения класс будет понижен, и возвращен нулевой указатель.

1.17.4 Пример Типа Значения

Например, рассмотрим следующий пример IDL:

```
// IDL

valuetype Example {
      short op1();
      long op2(in Example x);
      private short val1;
      public long val2;


      private string val3;
      private float val4;
      private Example val5;
};
```

Отображение на Си++ следующее:

```
// C++

class Example : public virtual ValueBase {
   public:
      virtual Short op1() = 0;
      virtual Long op2(Example*) = 0;


      virtual Long val2() const = 0;
      virtual void val2(Long) = 0;
```

```
      static Example* _downcast(ValueBase*);


   protected:
      Example();
      virtual ~Example();


      virtual Short val1() const = 0;
      virtual void val1(Short) = 0;


      virtual const char* val3() const = 0;
      virtual void val3(char*) = 0;
      virtual void val3(const char*) = 0;
      virtual void val3(const String_var&) = 0;


      virtual Float val4() const = 0;
      virtual void val4(Float) = 0;


      virtual Example* val5() const = 0;
      virtual void val5(Example*) = 0;
   private:
      // private and unimplemented

      void operator=(const Example&);
};
class OBV_Example : public virtual Example {
   public:
      virtual Long val2() const;
      virtual void val2(Long);


   protected:
      OBV_Example();
      OBV_Example(Short init_val1, Long init_val2,
            const char* init_val3, Float init_val4,
            Example* init_val5);
      virtual ~OBV_Example();


      virtual Short val1() const;
      virtual void val1(Short);


      virtual const char* val3() const;
      virtual void val3(char*);
      virtual void val3(const char*);
      virtual void val3(const String_var&);
      virtual Float val4() const;
      virtual void val4(Float);
      virtual Example* val5() const;
      virtual void val5(Example*);


      // ...


};
```

1.17.5 ValueBase и Счетчик Ссылок

Отображение Си++ для **ValueBase** IDL типа сервера как абстрактного базового класса для всех Си++ **valuetype** классов. **ValueBase** предоставляет несколько чистых виртуальных функций счетчиков ссылок наследуемых всеми классами типов значений:

```
// C++

namespace CORBA {
   class ValueBase {
```

```
      public:
         virtual ValueBase* _add_ref() = 0;
         virtual void _remove_ref() = 0;
         virtual ValueBase* _copy_value() = 0;
         virtual ULong _refcount_value() = 0;


         static ValueBase* _downcast(ValueBase*);


      protected:
         ValueBase();
         ValueBase(const ValueBase&);
         virtual ~ValueBase();


      private:
         void operator=(const ValueBase&);
    };
}
```

Таблица 1-2 Описание Операций

Операция	Описание
_add_ref	Используется для инкрементирования счетчика ссылок экземпляра **valuetype**.
_remove_ref	Используется для декрементирования счетчикассылок экземпляра **valuetype** и удаления экземпляра когда объектная ссылка сбрасывается в нуль. Использование **delete** для уничтожения экземпляра требует чтобы все экземпляры **valuetype** были выделены с использованием **new.**
_copy_value	Используется для осуществления глубокого копирования экземпляра **valuetyp**. Копия не содержит связей с оригинальным экземпляром и имеет время жизни независимое от оригинального. С тех пор как Си++ поддерживает ковариантные возвращаемые типы, производные классы могут перезаписывать функцию **_copy_value** для возвращения указателя на производный класс предпочтительнее чем **ValueBase*,** но так как ковариантные возвращаемые типы по прежнему не обычно не поддерживаются коммерческими Си++ компиляторами, возвращаемое значение **_copy_value** может также быть **ValueBase*,** даже производных классов. Гибкая реализация ORB может использовать другие подходы. Переносимые приложения не зависят от ковариантных возвращаемых типов и взамен будет использовать понижение (Си++ dynamic_cast<> оператор может также быть использован для приведения иерархии значений, но это не доступно во всех Си++ компиляторах.) для востановления большинства производных типов копируемого **valuetype**.
_refcount_value	Возвращает значение счетчика ссылок для экземпляра **valuetype** в котором вызван.

ValueBase также предоставляет закрытый конструктор по умолчанию, закрытую копию конструктора, и закрытый виртуальный деструктор. Копия конструктора защищена для запрещения копирования конструктора производного экземпляра **valuetype** исключая функции производных классов, и деструктор закрыт для предотвращения прямого удаления экземпляров классов производных от **ValueBase.**

В отношении счетчика ссылок, **ValueBase** наследуется для представления только интерфейса счетчика ссылок. Зависимость от иерархии наследования класса **valuetype**, его экземпляр может требовать различные механизмы подсчета ссылок. Например, механизм подсчета ссылок необходимый для класса **valuetype** который поддерживается **interface** подобно различиям между ними необходимыми для постоянного конкретного класса **valuetype**, с тех пор как формирователь имеет выходы объектного адаптера для расссмотрения. Поэтому, нормальный сервер **ValueBase** как и множественное наследование виртуальных базовых классов в классе **valuetype**. Один путь наследования через иерархию наследования IDL для **valuetype,** так как все **valuetype** наследуются от **ValueBase,** который предоставляет интерфейс счетчика ссылок. Другой путь наследования через реализацию счетчика ссылок смешаных базовых классов, которые также наследуются от **ValueBase.**

1.17.5.1 Дополнения Модуля CORBA

Отображение Си++ также добавляет две дополнительных функции счетчика ссылок в пространстве имен **CORBA**, как показано ниже:

```
// C++

namespace CORBA {
   void add_ref(ValueBase* vb)

   {
      if (vb != 0) vb->_add_ref();
   }
   void remove_ref(ValueBase* vb)

   {
      if (vb != 0) vb->_remove_ref();
   }
   // ...

}
```

Эти функции предоставляются для согласования с объектными ссылками функций счетчика ссылок. Они подобны в этом в отличии от функций членов **_add_ref** и **_remove_ref**, они могут быть вызваны с нулевыми **valuetype** ссылками. Функция **CORBA::add_ref** инкрементирует счетчик ссылок экземпляра **valuetype** указаного не нулевым аргументом функции, или ничего не делает если аргумент нулевой указатель. Функция **CORBA::remove_ref** ведет себя так же исключая декрементирование счетчика ссылок. (Реализации, показанные выше, предназначены для определения требуемой семантики функций, не означает что согласующиеся реализации должны включать функции).

1.17.6 Счетчик Ссылок Смешаных классов

Отображение Си++ предоставляет два стандартных реализации счетчика ссылок смешанных базовых классов:

- **CORBA::DefaultValueRefCountBase,** который может служить как базовый класс для любого предоставляемого приложением конкретного **valuetype** класса чей соответствующий IDL тип значения не производный от IDL **interface**. Для этих типов классов **valuetype**, приложения также вольны использовать свою собственную реализацию счетчика ссылок.

- **PortableServer::ValueRefCountBase**, который должен служить как базовый класс для конкретных предоставляемых приложениями конкретных **valuetype** классов чьи соответствующие **IDL valuetype** производные от одного или более **IDL interface,** и чьи экземпляры будут зарегистрированы с **POA** как серванты. Если наследование IDL интерфейса присутствует, но экземпляры предоставляемых приложениями конкретных **valuetype** классов не будут зарегистрированы с **POA, CORBA::DefaultValueRefCountBase** или зависящая от приложения реализация счетчика ссылок может быть использована как базовый класс.

Каждый из этих классов будет полностью конкретным и полностью осуществлять интерфейс подсчета ссылок **ValueBase**, исключая то, что с тех пор как они предоставляют реализацию, не интерфейс, они не предоставляют понижение (downcasting). В дополнение, каждый из этих классов будет предоставлять закрытый конструктор по умолчанию который устанавливает счетчик ссылок экземпляра в единицу, закрытый виртуальный деструктор, и закрытый конструктор копирования который устанавливает счетчик ссылок созданного экземпляра в нуль. Как и с базовыми классами **ValueBase**, оператор распределения по умолчанию будет закрытый (private) и желательно нереализованым для полного запрещения распределения.

Поддерживаемый приложением конкретный **valuetype** класс который должен производится от таких смешанных классов, а не от **valuetype** классов создаваемых компилятором IDL. Это создано для избежания наследования этих смешаных классов как виртуальной основы, или для избежания наследования множественных копий смешанных (и их счетчиков множественных ссылок) если виртуальная база не применяется. Также, только конечный реализатор **valuetype** знает будет ли он использован как POA сервант или нет, и поэтому реализатор должен определить желательный счетчик смешанных ссылок.

1.17.7 Наборы Значений(Value Boxes)

Класс набора значений по существу предоставляет версию счетчика ссылок его базового класса. В отличии от нормального класса **valuetype**, Си++ классы для наборов значений могут быть конкретными с тех пор как наборы значений не поддерживают методы, наследование, или интерфейсы. Набор значений Классы наборов значений отличаются в зависимости от их базовых типов.

Для осуществления интерфейса **ValueBase**, все классы набора значений являются производными от **CORBA::DefaultValueRefCountBase.**

1.17.7.1 Передача параметров для базовых Boxed Type

Все классы наборов значений предоставляют **_boxed_in**, **_boxed_inout,** и **_boxed_out** функции члены которые позволяют базовым наборам значений быть переданными функциям принимающим параметры типа производных наборов. Сигнатуры этих функций зависят от типов передачи параметров базовых типов наборов. Возвращение значений функций **_boxed_inout** и **_boxed_out** будет таким же как если бы набор значений был бы указан напрямую, что позволяет замещение или установку нового значения. Например, вызов **_boxed_out** на набор строк позволяет замещение текущей строки набора значенний:

```
// IDL

valuetype StringValue string;
interface X {
   void op(out string s);
};


// C++

StringValue* sval = new StringValue("string val");
X_var x = ...

x->op(sval->_boxed_out()); // boxed string is replaced

                           // by op() invocation
```

Реализация **op** следущая:

```
// C++

void MyXImpl::op(String_out s)
{
       s = string_dup("new string val");
}
```

Возвращаемое значений функции **_boxed_out** будет таким же как строковое значение в экземпляре указанном **sval** установлено **"new string val"** после возвращения **op**, с экземпляром указаным **sval** поддерживающем владение строкой.

1.17.7.2 Основные типы, Перечисления (Enums), и Объектные Ссылки (Object References)

Для всех знаковых и беззнаковых типов кроме **fixed** типов, и для **boolean, octet, char, wchar, float, double, long double,**и перечислимых типов и для **typedefs** всех их, классы набора значений предоставляют:

- Открытый конструктор по умолчанию. Исключая случай с объектной ссылкой, значение базового набора значений будет неопределенным после запуска этого конструктора (т.е., конструктор по умолчанию не инициализирует набор значений переданного значения). Это происходит потому, что встроеный конструктор для каждого из основных типов и перечислений не инициализирует

экземпляры их типов для частных значений. Для наборов объектных ссылок, этот конструктор устанавливает базовый набор объектных ссылок на нуль.

- Открытый конструктор который принимает один аргумент базового типа. Этот аргумент используется для инициализации значения базового набора типов.
- Открытый оператор распределения который принимает один аргумент базового типа. Этот аргумент используется для замещения значения базового boxed type.
- Открытые фун кции аксессора и модификатора для наборов значений. Функции аксессора и модификатора всегда называются **_boxed_in, _boxed_inout**, и **_boxed_out**, и их возвращаемые типы совпадают с **in, inout**, и **out** методами передачи параметров, соответственно, базовым типам наборов. Неявное преобразование к базовому типу не предоставляется потому, что значения обычно управляются указателем.
- Открытый конструктор копирования
- Открытую статичную функцию **_downcast**.
- Защищенный деструктор.
- Закрытый и желательно не реализованый оператор распределения по умолчанию.

Пример класса **value box** для перенумерованого типа показан ниже:

```
// IDL

enum Color { red, green, blue };
valuetype ColorValue Color;


// C++

class ColorValue : public DefaultValueRefCountBase {
   public:
      ColorValue();
      ColorValue(Color val);


      ColorValue(const ColorValue& val);


      ColorValue& operator=(Color val);


      Color _value() const;// accessor

      void _value(Color val);// modifier



      // explicit conversion functions for

      // underlying boxed type

      //

      Color _boxed_in() const;
      Color& _boxed_inout();
      Color& _boxed_out();


      static ColorValue* _downcast(ValueBase* base);


   protected:
      ~ColorValue();


   private:
      void operator=(const ColorValue& val);
};
```

1.17.7.3 Типы Структур (Struct Types)

Классы наборов значений для структурных типов отображаются на классы которые предоставляют функции аксессора и модификатора для каждого члена структуры. В частности, классы предоставляют:

- Открытый конструктор по умолчанию. Базовый тип блоковых структур инициализируется как если бы это был собственный конструктор по умолчанию.
- Открытый конструктор который принимает единственный аргумент типа **const T&,** где **T** это базовый тип блоковой структуры.
- Открытый оператор распределения который принимает единственный аргумент типа **const T&,** где **T** это базовый тип блоковой структуры.
- Открытые функции аксессора и модификатора, все называются **_value,** для базовых типов блоковых структур. Два аксессора предоставляют: одну постоянную функцию член возвращающую **const T&,** и другие непостоянные функции члены возвращающие **T&**. Функция модификатор принимает единственный аргумент типа **const T&**.
- **_boxed_in, _boxed_inout,** и **_boxed_ out** функции которые позволяют получить доступ к блоковому значению для передачи в ее сигнатуру ожидаемый базовый тип структуры. Возвращаемое значение этих функций соответствует **in, inout,** и **out** способам передачи параметров для базовых типов блоковых структур соответственно.
- Для каждого члена структуры, набор функций аксессора и модификатора. Эти функции имеют такую же сигнатуру как и у функций аксессора и модификатора для членов объединения.
- Открытый конструктор копирования.
- Открытую статичную функцию **_downcast**.
- Защищенный деструктор.
- Закрытый и предпочтительно нереализованный оператор размещения по умолчанию.

Как и с другими классами наборов значений, нет неявных преобразований к базовым блоковым типам предоставляемым с тех пор как значения обычно управляются указателем.

Например:

```
// IDL

struct S {
   string str;
   long len;
};
valuetype BoxedS S;


// C++

class BoxedS : public DefaultValueRefCountBase {
   public:
      BoxedS();
      BoxedS(const S& val);
      BoxedS(const BoxedS& val);


      BoxedS& operator=(const S& val);


      const S& _value() const;
      S& _value();
      void _value(const S& val);


      const S& _boxed_in() const;
      S& _boxed_inout();
```

```
      S*& _boxed_out();


      static BoxedS* _downcast(ValueBase* base);


      const char* str() const;
      void str(char* val);
      void str(const char* val);
      void str(const String_var& val);


      Long len() const;
      void len(Long val);


   protected:
      ~BoxedS();


   private:
      void operator=(const BoxedS& val);
};
```

1.17.7.4 Типы String и WString

В порядке позволения наборам строк быть переработанными как нормальным строкам было определено, значение блокового класса для строк предоставляет почти такой же интерфейс как класса **String_var.**

Различия между интерфейсами класса **String_var** следующие:

- Интерфейс класса **value box** не предоставляет **in, inout, out,** и **_retn** функции которые предоставляет **String_var**. Класс **value box** предоставляет замещение для этих функций называемых **_boxed_in, _boxed_inout,** и **_boxed_out.** Они имеют в основном одинаковую семантику и сигнатуру как их аналог **String_var**, но их имена будут изменены для того, чтобы сделать их свободными, чтобы они предоставляли доступ к базовым строкам, не к самому **value box**.

- Нет обременительных операторов для неявного преобразования в базовый строковый тип потому, что значения обычно управляются через указатель.

В дополнение к большинству интерфейсов **String_var**, класс **value box** для строк предоставляет:

- Открытые функции аксессора и модификатора для значений наборов строк. Ёти функции называются **_value.** Единственная функция аксессора не принимает аргументов и возвращает **const char*.** Существует три функции модификатора, каждая принимает единственный аргумент. Одна принимает аргумент **char*** который принимается классом **value box**, одна принимает аргумент **const char*** который копируется, и одна принимает **const String_var&** откуда копируется базовое значение строки.

- Открытый конструктор по умолчанию который инициализирует базовую строку в пустую строку.

- Три открытых конструктора которые принимают строковые аргументы. Один принимает приемлемый аргумент **char***, один принимает **const char*** который копирует, и один принимает **const String_var&** откуда копируется значение базовой строки. Если **String_var** не содержит строку, значение **boxed string** инициализируется пустой строкой.

- Три открытых оператора размещения: один принимает параметр типа **char***, один принимает параметр типа **const char*** который копирует, и один который принимает параметр типа **const String_var&** откуда копирует значение базовой строки. Каждый возвращает ссылку на связаный с ним объект. Если **String_var** не содержит строки, значение **boxed string** устанавливается равным пустой строке.

- Открытый конструктор копирования.

- Открытую статичную функцию **_downcast.**

- Защищенный деструктор.

- Закрытый и предпочтительно нереализованый оператор размещения по умолчанию.

Пример класса **value box** для строки показан ниже:

```
// IDL

valuetype StringValue string;


// C++

class StringValue : public DefaultValueRefCountBase {
   public:
   // constructors

   //

   StringValue();
   StringValue(const StringValue& val);
   StringValue(char* str);
   StringValue(const char* str);
   StringValue(const String_var& var);
   // assignment operators

   //

   StringValue& operator=(char* str);
   StringValue& operator=(const char* str);
   StringValue& operator=(const String_var& var);


   // accessor

   //

   const char* _value() const;


   // modifiers

   //

   void _value(char* str);
   void _value(const char* str);
   void _value(const String_var& var);


   // explicit argument passing conversions for

   // the underlying string

   //

   const char* _boxed_in() const;
   char*& _boxed_inout();
   char*& _boxed_out();


   // ...other String_var functions such as overloaded

   // subscript operators, etc....



   static StringValue* _downcast(ValueBase* base);


   protected:
      ~StringValue();


   private:
      void operator=(const StringValue& val);
};
```

Через классы **value box** для строк предоставляют подстрочные операторы, факт, что значения обычно управляются указателем означает, что они должны быть разъименованы перед использованием **subscript** операторов.

1.17.7.5 Типы Объединение, Последовательность, Fixed и Any

Наборы значений для этих типов отображается на классы которые имеют полностью одинаковые открытые интерфейсы как базовые блоковые типы, исключая что каждый имеет:

- В дополнение к конструкторам предоставляемым классом для базовых блоковых типов, открытый конструктор который имеет единственный аргумент типа **const T&,** где **T** это базовый блоковый тип.
- Оператор распределения который имеет один аргумент типа **const T&,** где **T** это базовый блоковый тип.
- Функции аксессора и модификатора для базовых блоковых значений. Все подобные функции называются **_value.** Есть две функции аксессора, одна функция - постоянный (const) член возвращает **const T&,** и другая не постоянная функция член возвращает **T&.** Функция модификатора имеет один аргумент типа **const T&.**
- Функции **_boxed_in**, **_boxed_inout**, и **_boxed_out** которые позволяют получить доступ к набору значений для передачи его в сигнатурах ожидающих базовый тип набора значений. Возвращаемые значения этих функций соответствует **in, inout,** и **out** типам передачи параметров для базовых **boxed** типов, соответственно.
- Защищенный деструктор.
- Закрытый и предпочтительно нереализованый оператор распределения по умолчанию.

Как с другими классами наборов значений, нет неявных преобоазований к базовым блоковым типам и предоставляемых с тех пор как значения обычно управляется указателем.

Заметим, что класс **value box** для типов последовательностей предоставляет перегруженные операторы индекса (**operator[ ]**) только как это делает класс последовательность. Тем не менее, с тех пор как значения нормально управляются указателем, зкземпляры значений должны быть разименованы перед применения к ним перегруженных операторов индекса.

Экземпляры **value box** для типа **any** могут быть переданы перегруженным операторам для вставки и извлечения вызовом соответствующих явных функций преобразования:

```
// C++

AnyValueBox* val = ...

val->_boxed_inout() <<= something;
if (val->_boxed_in() >>= something_else) ...
```

Ниже находится пример **value box** с соответствующим Си++ классом:

```
// IDL

typedef sequence<long> LongSeq;
valuetype LongSeqValue LongSeq;


// C++

class LongSeqValue : public DefaultValueRefCountBase {
   public:
      LongSeqValue();
```

```
      LongSeqValue(ULong max);
      LongSeqValue(ULong max,


      ULong length,
      Long* buf,
      Boolean release = 0);
      LongSeqValue(const LongSeq& init);
      LongSeqValue(const LongSeqValue& val);
      LongSeqValue& operator=(const LongSeq& val);
      const LongSeq& _value() const;
      LongSeq& _value();


      void _value(const LongSeq&);
      const LongSeq& _boxed_in() const;
      LongSeq& _boxed_inout();
      LongSeq*& _boxed_out();
      static LongSeqValue* _downcast(ValueBase*);
      ULong maximum() const;
      ULong length() const;
      void length(ULong len);
      Long& operator[](ULong index);
      Long operator[](ULong index) const;


   protected:
      ~LongSeqValue();


   private:
      void operator=(const LongSeqValue&);
};
```

1.17.7.6 Типы Массив (Array Types)

В порядке позволения упакованным массивам быть обработанными как нормальным массивам было предназначено, значение классов **value box** для массивов предоставлять совершенно подобный интерфейс как соответствующего массиву **_var** классу. Различия между интерфейсом **_var** класса следующие:

- Интерфейс класса **value box** не предоставляет функции **in, inout, out,** and **_retn** которые предоставляет **_var**. Класс **value box** предоставляет замещение для этих функций называемых **_boxed_in**, **_boxed_inout**, и **_boxed_out.** Они имеют подобную семантику и сигнатуру как их **_var** аналоги, но их имена изменены для того, чтобы сделать их чистыми чтобы предоставить доступ к базовым массивам, не к самим **value box**.

- Нет перезагруженных операторов для неявного преобразования к базовым типам массивов потому, что значения обычно управляются через указатель.

В дополнение к большинству **_var** интерфейса, классы **value box** для массивов предоставляют:

- Открытые функции аксессора и модификатора для упакованных значений массивов. Эти функции называются **_value.** Единственная функция аксессора имеет единственный аргумент и возвращает указатель на слайс массива. Функция модификатора имеет один аргумент типа **const array.**

- Открытый конструктор по умолчанию.

- Открытый конструктор, который принимает аргументом **const array**.

- Открытый оператор размещения, который принимает аргументом **const array**.

- Открытый конструктор копирования.

- Открытую статичную функцию **_downcast**.

- Защищенный деструктор.

- Закрытый и предпочтительно нереализованный оператор размещения по умолчанию.

Пример класса **value box** для массива показан ниже:

```
// IDL

typedef long LongArray[3][4];
valuetype ArrayValue LongArray;



// C++

typedef Long LongArray[3][4];
typedef Long LongArray_slice[4];
class ArrayValue : public DefaultValueRefCountBase {
   public:
      ArrayValue();
      ArrayValue(const ArrayValue& val);
      ArrayValue(const LongArray val);


      ArrayValue& operator=(const LongArray val);


      const LongArray_slice* _value() const;
      LongArray_slice* _value();


      void _value(const LongArray val);


      // explicit argument passing conversions for

      // the underlying array

      //

      const LongArray_slice* _boxed_in() const;
      LongArray_slice* _boxed_inout();
      LongArray_slice* _boxed_out();


      // ...overloaded subscript operators...



      static ArrayValue* _downcast(ValueBase* base);


   protected:
      ~ArrayValue();


   private:
      void operator=(const ArrayValue& val);
};
```

Заметим, что даже через классы **value box** для массивов предоставляют перезагруженные операторы описания (overloaded subscript operators), факт того, что значения обычно управляются через указатели означает, что они должны быть разыменованы перед перед использованием описывающих операторов.

1.17.8 Abstract Valuetypes

Абстрактные **IDL valuetype** следуют тем же правилам отображения как конкретные **IDL valuetype**, исключая то, почему они не имеют членов данных, IDL компилятор не генерирует **OBV_** классы для них.

1.17.9 Наследование Типа Значения

Для **IDL valuetype** производного от других **valuetype** или тех, которые поддерживают **interface** типы, возможны некоторые сценарии наследования Си++:

- Конкретные *value base classes* наследуутся как открытые виртуальные базы для позволения для "цепного стиля" наследования реализации.

- Абстрактные *value base classes* наследуются как открытые виртуальные базовые классы, с тех пор как они могут быть множественно унаследованы в **IDL.**

- Классы интерфейсов поддерживаемые **IDL valuetype** не наследуются. Взамен, операции над интерфейсами (и базовые интерфейсы, если это имеет место)
  отображаются на чистые виртуальные функции в создаваемом Си++ базовом классе значения.
  В дополнение к этим абстрактным базовым классам значения и **OBV_** классам, **IDL** компилятор создает **POA** скелетон для этих типов значений, имя этих скелетонов формируется добавлением в начало полного имени **valuetype** строки **"POA_"**. Базовый класс значения и **POA** скелетон типа интерфейса это открытый виртуальный базовый класс этого скелетона.

Причина по которой **interface** классы не наследуются в том, что экземпляры **valuetype**, такие как **POA** серванты, сами не являются объектными ссылками. Предоставление этого наследования будет позволено для подтвержденного ошибками кода который неявно расширен указателями на экземпляры **valuetype** к Си++ объектным ссылкам для этих **valuetype** экземплярам из **POA**. Когда такие приложения пытаются использовать неверную объектную ссылку полученную таким способом, возникнут ошибки столкновений которые затруднят откат неявного расширения указателя **valuetype** на объектную ссылку. язык Си++ не предоставляет сигналов отбоя (hooks) в механизмах неявного расширения расширениях указатель-класс через который приложения могут охранятся от ошибок этого типа .

Избегание произведения классов **valuetype** от классов **interface** также разделяет время жизни экземпляра **valuetype** от времени жизни экземпляра объектной ссылки. Ёто будет непредсказуемым для приложения, если верная объектная ссылка которая не была освобождена неожиданно становится неверн потому, что другая часть программы декрементировала часть **valuetype** экземпляра объектной ссылки в нуль. Ётот сценарий будет разрещен предоставлением счетчика ссылок mix-in класса. Однако, предоставление этого подобно приближению разрыва локальной/удаленной прозрачности наличием операций по освобождению объектных ссылок воздействующих на сервант. и передачей связаных проблем описаных в предыдущем параграфе, производные **valuetype** классы от классов **interface** наиболее выгодны.

Пример отображения для **valuetype** которое поддерживает **interface** показано ниже.

```
// IDL

interface A {
      void op();
};


valuetype B supports A {
      public short data;
};


// C++

class B : public virtual ValueBase {
   public:
      virtual void op() throw(SystemException) = 0;


      virtual Short data() const = 0;
      virtual void data(Short) = 0;


// ...
```

```
};

class POA_B : public virtual POA_A, public virtual B {
   public:
      virtual void op() throw(SystemException) = 0;
      // ...

};
```

Заметим, что операции **valuetype** наследуются от **interface** всегда включают спецификацию исключения, только как классы скелетоны.

1.17.10 Фабрики Valuetype

Потому, что конкретные **valuetype** классы предоставляются разработчиками приложений, создание значений при определенных обстоятельствах проблематично. Ёти обстоятельства включают:

- Демаршалинг. **ORB** не может знать априори о всех потенциальных конкретных классах значений поддерживаемых приложением, и поэтому механизм демаршалинга **ORB** не обладает возможностью прямого создания экземпляра этих классов.
- Библиотеки Компонентов. Части приложения, такие как части структуры, могут быть ограничены манипуляциями с экземплярами **valuetype** пока предоставляется создание этих экземпляров остальным частям приложения.

1.17.10.1 Класс ValueFactoryBase

Как они предоставляют конкретные Си++ **valuetype** классы, приложения должны также предоставлять фабрики для этих конкретных классов. База всех классов фабрик значений в Си++ **CORBA::ValueFactoryBase** классе:

```
// C++

namespace CORBA {
      class ValueFactoryBase;
      typedef ValueFactoryBase* ValueFactory;


      class ValueFactoryBase

      {
      public:
         virtual ~ValueFactoryBase();


         virtual void _add_ref();
         virtual void _remove_ref();


         static ValueFactory _downcast(ValueFactory vf);


      protected:
         ValueFactoryBase();


      private:
         virtual ValueBase* create_for_unmarshal() = 0;
      };
      // ...

}
```

Отображение Си++ для **IDL CORBA::ValueFactory** родного типа это указатель на класс **ValueFactoryBase**, как показано выше. Приложения производные конкретных классов фабрик **ValueFactoryBase**, и совпадающие экземпляры этих классов фабрик с **ORB** через функцию **ORB::register_value_factory**. Если фабрика зарегистрирована для данного типа значения и небыло предыдущих фабрик зарегистрированых за этим типом, функция **register_value_factory** возвратит нулевой указатель.

При демаршалинге экземпляров значений, **ORB** необходимо быть способной вызывать приложения для запроса о создании этих экземпляров. Однако, создание для демаршалинга отличается потому, что **ORB** не знает об фабриках определенных приложениями, и и на самом деле в большинстве случаев может даже не иметь необходимых аргументов для предоставления зависящих от типа фабрик.

Для позволения **ORB** создавать экземпляры значений требуемых во время демаршалинга, класс **ValueFactoryBase** предоставляет чистую виртуальную функцию **create_for_unmarshal**. Функция закрытая, поэтому только **ORB**, через зависящий от реализации способ (т.е., через дружественные классы), может вызывать ее. Приложения не ожидают вызова функции**create_for_unmarshal**. Производные классы могут перезаписывать функцию **create_for_unmarshal** и реализовывать ее так, что она создает новый экземпляр значения и возвращает указатель на нее. Вызывающий оператор принимает владение возвращаемым экземпляром и должен обеспечивать, что **_remove_ref** вызвана на него. С тех пор как функция **create_for_unmarshal** возвращает указатель на **ValueBase**, вызывающий оператор может использовать функцию downcasting`a поддерживаемую типом значения для понижения указателя на указатель на производный тип значения.

Единыжды созданный **ORB** экземпляр значения через функцию **create_for_unmarshal**, может использовать функцию модификации значения члена данных для установки состояния нового экземпляра значения из демаршализованых данных. Как **ORB** получает доступ к защищенным значениям модификаторов данных членов зависит от реализации и не влияет на переносимость приложения.

ValueFactoryBase использует счетчик интерфейсов для предотвращения себя от разрушения во время использования приложением. **ValueFactoryBase** имеет начальное значение счетчика ссылок равным единице. Вызов **_add_ref** на **ValueFactoryBase** увеличивает его счетчик ссылок на единицу. Вызов **_remove_ref** на **ValueFactoryBase** уменьшает его счетчик ссылок на единицу, и если результат равен нулю, **_remove_ref** вызывает **delete** на этот **this** указатель в порядке разрушения фабрики. Для **ORB** которые работают в многопоточной среде, реашлизация **ValueFactoryBase::_add_ref** и **ValueFactoryBase::_remove_ref** - thread-safe.

Когда **valuetype** фабрика регистрируется с **ORB**, **ORB** вызывает **_add_ref** один раз на фабрику перед возвращением из **register_value_factory.** Когда **ORB** закончил использовать фабрику, счетчик ссылок декрементируется один раз. Ёто может случаться при некоторых из следующих обстоятельствах:

- Еси фабрика явно разрегистрирована через **unregister_value_factory**, **ORB** вызывает **_remove_ref** на фабрику один раз.
- Если фабрика неявно разрегистрирована **ORB::shutdown**, **ORB** несет ответственность за вызов **_remove_ref** единыжды на каждую зарегистрированую фабрику.
- Если фабрика замещена новым вызовом **register_value_factory,** предварительно зарегистрированая фабрика фозвращается вызывающему который принимает собственность на ссылку на эту фабрику. Когда вызывающая программа заканчивает работу с фабрикой , она вызывает **_remove_ref** один раз на эту фабрику.

Вызывающая программа **lookup_value_factory** принимает в собственность одну ссылку на фабрику. Когда вызывающая программа заканчивает работу с фабрикой, она вызывает **_remove_ref** один раз на эту фабрику.

Функция **_downcast** на фабрику позволяет возвращать тип функции **ORB::lookup_value_factory** для того чтобы понизить указатель на зависящую от типа фабрику. Важно отметить, что возвращаемое значение фабрики **_downcast** не становится объектом ответственности за управлением памятью вызывающей прогораммы, и **_remove_ref** не вызывается на него.

1.17.10.2 Классс ValueFactoryBase_var

Для удобства автоматического управления **valuetype** фабрикой счетчика ссылок, **CORBA namespace** предоставляет класс **ValueFactoryBase_var**. Ётот класс ведет себя подобно классу **PortableServer::ServantBase_var** для управления памятью серванта.

```
// C++

namespace CORBA
{
      class ValueFactoryBase_var

      {
      public:
         ValueFactoryBase_var() :_ptr(0) {}
         ValueFactoryBase_var(ValueFactoryBase* p)

            : _ptr(p) {}
         ValueFactoryBase_var(const ValueFactoryBase_var& b)

            : _ptr(b._ptr)

         {
            if (_ptr != 0) _ptr->_add_ref();
         }
         ~ValueFactoryBase_var()

         {
            if (_ptr != 0) _ptr->_remove_ref();
         }
         ValueFactoryBase_var&



         operator=(ValueFactoryBase* p)

         {
            if (_ptr != 0) _ptr->_remove_ref();
         _ptr = p;
            return *this;
         }
         ValueFactoryBase_var&

         operator=(const ValueFactoryBaseBase_var& b)

         {
            if (_ptr != b._ptr) {
            if (_ptr != 0) _ptr->_remove_ref();
            if ((_ptr = b._ptr) != 0)

               _ptr->_add_ref();
         }
         return *this;
      }
      ValueFactoryBase* operator->() const {return _ptr;}
      ValueFactoryBase* in() const { return _ptr; }
      ValueFactoryBase*& inout() { return _ptr; }
      ValueFactoryBase*& out()

      {
         if (_ptr != 0) _ptr->_remove_ref();
         _ptr = 0;
         return _ptr;
      }
      ValueFactoryBase* _retn()

      {
         ValueFactoryBase* retval = _ptr;
         _ptr = 0;
         return retval;
      }
      private:
         ValueFactoryBase* _ptr;
      };
      // ...
```

```
}
```

Реализация показаная выше для **ValueFactoryBase_var** предназначена только как пример который представляет требуемую семантику. Изменения этой реализации согласуется так долдо как ини предоставляют такую семантику как в реализации показаной здесь.

1.17.10.3 Type-Specific Фабрики Значений

Все **valuetype** которые имею операции инициализирования описаные для них также зависят от типа Си++ фабрики значений создаваемой для них. Для **valuetype A**, имя класса фабрики, который создается в той же области видимости как и класс значения, будет **A_init**. Каждая операция инициализации отображается на чистую виртуальную функцию в классе фабрики, и каждый из этих инициализаторов определенных в **IDL** отображаются на функцию инициализатор с таким же именем. Базовые **valuetype** инициализаторы не наследуются, и поэтому не появляются в классах фабрик. Параметры инициализатора отображаются используя нормальные Си++ правила передачи параметров для **in** параметров. Возвращаемый тип каждой функции инициализатора это указатель на созданный **valuetype.**

Например, рассмотрим следующий **valuetype:**

```
// IDL

valuetype V {
   factory create_bool(boolean b);
   factory create_(char c);
   factory create_(octet o);
   factory create_(short s, string p);
   ...

};
```

Класс фабрики для примера данного выше будет создан следующим образом:

```
// C++

class V_init : public ValueFactoryBase {
   public:
      virtual ~V_init();


      virtual V*

      create_bool(Boolean val) = 0;
      virtual V* create_char(Char val) =0;
      virtual V* create_octet(Octet val)=0;
      virtual V* create_other(Short s, const char* p) = 0;


      static V_init* _downcast(ValueFactory vf);


   protected:
      V_init();
};
```

Каждый создаваемый класс фабрики имеет открытый виртуальный деструктор, защищенный конструктор по умолчанию, и открытую функцию **_downcast** позволяющую даункастинг (downcasting) от указателя на базовый **ValueFactoryBase** класс. Каждый также поддерживает открытую чистую виртуальную функцию **create** соответствующую каждому инициализатору. Приложения производят конкретные классы фабрик от этих классов и регистрируют их с **ORB.** С тех пор как каждое создаваемое значение фабрики производится от **ValueFactoryBase,** все конкретные классы фабрик могут также перезаписывать закрытые чистые виртуальные функции **create_for_unmarshal** наследуемые от **ValueFactoryBase.**

Для **valuetype** которые не имеют инициализаторов, не существует зависящих от типа абстрактных классов фабрик, но приложения должны поддерживать конкретные классы фабрик. Ёти классы, которые являются производными напрямую от **ValueFactoryBase,** не нуждаются в поддержке функции **_downcast**, и нуждаются только в перезаписывании функции **create_for_unmarshal**.

1.17.10.4 Проблемы Демаршалинга

Когда **ORB** демаршализует **valuetype** для запроза управляемого через статичный стаб или скелетон Си++, он пытается найти фабрику для **valuetype** через операцию **ORB::lookup_value_factory**. Если поиск фабрики не удается, клиентское приложение получает исключение **CORBA::MARSHAL**. Таким образом, приложение использующее статичный стаб или скелетон должно обеспечивать, что фабрика типа значения зарегистрированого для каждого **valuetype** ожидается получение через статичные механизмы вызова.

Ввиду их динамической природы, от приложения использующие **DII** или **DSI** не ожидают, что они будут иметь информацию времени компиляции для всех **valuetype** которые они могут получить. Для этих приложений, экземпляры **valuetype** представлены как **CORBA::Any,** и поэтому фабрики значений не требуют быть зарегистрированными с **ORB** для позволения таким **valuetype** быть демаршализованными. Тем не менее, фабрики значений должны быть зарегистрированными с **ORB** и доступными для поиска если приложения пытаются выделить **valuetype** с помощью статически типизированых функций извлечения **Any**.

1.17.11 Выборочный Маршалинг

Отображение Си++ для типов **IDL CORBA::CustomerMarshal**, **CORBA::DataOutputStream**, и **CORBA::DataInputStream** следует нормальным правилам отображения Си++ **valuetype**.

1.17.12 Другие Примеры Valuetype

```
// IDL

valuetype Node {
      public long data;
      public Node next;
      void print();


      Node change(in Node inval,
      inout Node ioval,
      out Node outval);
};


// C++

class Node : public virtual ValueBase
{
   public:
      virtual Long data() const = 0;
      virtual void data(Long) = 0;
      virtual Node* next() const = 0;
      virtual void next(Node*) = 0;
      virtual void print() = 0;
      virtual Node* change(Node* inval,
            Node*& ioval,
            Node_out outval) = 0;


      static Node* _downcast(ValueBase*);


   protected:
      Node();
```

```
      virtual ~Node();


   private:
      // private and unimplemented

      void operator=(const Node&);
};


class OBV_Node : public virtual Node
{
   public:
      virtual Long data() const;
      virtual void data(Long);
      virtual Node* next() const;
      virtual void next(Node*);


   protected:
      OBV_Node();
      OBV_Node(Long data_init, Node* next_init);
      virtual ~OBV_Node();


   private:
      // private and unimplemented

      void operator=(const OBV_Node&);
};
```

1.17.13 Valuetype Члены Структур

Как описано в секции 1.9, УMapping for Structured TypesФ, членам структур необходимо быть самоуправляемыми. Ёто является результатаом того, что необходимы для управляющих типов строковых и объектных ссылок. С тех пор как **valuetype** управляются через указатель, подобно управлению строковыми и объектными ссылками, они также требуют управляющих типов для представления их когда они используются как члены структур.

Экземпляр **valuetype** управляющего типами имеет семантику подобную управляющему типами для объектной ссылки:

- Любое распределение для управляемого члена **valuetype** вызывает, что член декрементирует счетчик ссылок **valuetype** которым упрвлавляет, если он есть.
- **valuetype** указатель задаваемый для управляемого члена **valuetype** принимается членом.
- **valuetype _var** заданый для управления членом **valuetype** вызовет инкрементирование счетчика ссылок экземпляра. Управляющие типы **_var** и **valuetype** следуют таким же правилам для расширения распределения как и для объектных ссылок.
- Если создаваемый тип содержит управляемый член **valuetype** связаный с другим создаваемым типом (например, экземпляром структуры с членом**valuetype** с вязанным с другим экзепляром той же структуры), счетчик ссылок управляемого экземпляра **valuetype** в структуре с правой стороны инкрементируются, пока счетчик ссылок управляемого экземпляра с левой стороны декрементируется. Как обычно в Си++, распределение должно быть защищено против лишнихопераций над счетчиком ссылок.
- Когда он разрушается, управляемый член **valuetype** декрементирует счетчик ссылок управляемого экземпляра **valuetype**.

Семантика менеджеров типов значений описана для предоставления разделения экземпляров типов значений через создаваемые типы по умолчанию. Каждый Си++ **valuetype** также предоставляет явную функцию копирования которая может быть использована для избежания разделению когда необходимо.

1.18 Отображение Абстрактных Интерфейсов

Отображение Си++ для абстрактных интерфейсов в большинстве идентично отображению нормальных интерфейсов. Вместо определения полного Си++ тображения для абстрактных интерфейсов , которое будет дублировать спецификацию отображения обычных интерфейсов находящихся в секции 1.3 "Отображение для интерфейсов", будут описаны только отличия в отображении.

1.18.1 База Абстрактного Интерфейса

Классы Си++ для абстрактных интерфейсов не производятся от класса **CORBA::Object**. В **IDL,** абстрактные интерфейсы не имеют общей базы. Однако, для облегчения сужения от абстрактных интерфейсов базовых классов вниз к производным абстрактным интерфейсам, производным интерфейсам, и производным типам **valuetype**, все абстрактные интерфейсы базовых классов которые не имеют других базовых абстрактных интерфейсов производятся напрямую от **CORBA::AbstractBase.** Этот базовый класс предоставляет следующее:

- Защищенный конструктор по умолчанию
- Защищенный конструктор копирования
- Защищенный чистый виртуальный деструктор
- Открытая статичная функция **_duplicate**
- Открытая статичная функция **_narrow**
- Открытая статичная функция **_nil**

Класс **AbstractBase** показан ниже:

```
// C++

class AbstractBase;
typedef ... AbstractBase_ptr;// either pointer or class



class AbstractBase {
   public:
      static AbstractBase_ptr _duplicate(AbstractBase_ptr);
      static AbstractBase_ptr _narrow(AbstractBase_ptr);
      static AbstractBase_ptr _nil();


   protected:
      AbstractBase();
      AbstractBase(const AbstractBase& val);
      virtual ~AbstractBase() = 0;
};
```

Функция **_duplicate** работает полиморфно над обеими объектными ссылками и типами **valuetype**. Если **AbstractBase_ptr** реально ссылается на объектную ссылку переданую функции **_duplicate**, объектная ссылка дублируется и дублирует возвращаемую объектную ссылку. Иначе, аргумент ссылается на экземпляр типа значения, поэтому функция **_add_ref** вызывается на тип значения и возвращаемый аргумент. Если аргумент нулевой **AbstractBase_ptr,**возвращаемое значение - нуль.

Реализация **AbstractBase::_narrow** просто передает свои аргументы **_duplicate** и использует значение которое она возвращает как свое собственное возвращаемое значение. Собственно говоря, функция **_narrow** не нуждается в интерфейсе **AbstractBase** потому, что она немного использует для сужения **AbstractBase** для своего собственного типа, но это требуется всеми согласующимися реализациями для написания Си++ спецификатора шаблона который согласуется с абстрактным интерфейсом (**AbstractBase** не присутствует особым случаем).

Как с правильными объектными ссылками, функции **_nil** возвращают типизированные **AbstractBase** нулевые ссылки.

Обе **is_nil** и **release** функции в пространстве имен **CORBA** перезагружаются для управления ссылками абстрактных интерфейсов:

```
// C++

namespace CORBA {
      Boolean is_nil(AbstractBase_ptr);
      void release(AbstractBase_ptr);
}
```

Это ведет себя как его аналог объектная ссылка. Отметим, что от **release** ожидается для работы полиморфно через оба типа **valuetype** и типов объектных ссылок. Если его аргумент - нулевой, он ничего не делает. Если аргумент ссылается на экземпляр типа значения. он вызывает **_remove_ref** на этот экземпляр. Иначе, его аргумент ссылается на объектную ссылку, на которую он вызывает **CORBA::release** для объектной ссылки.

1.18.2 Отображение Клиентской Части

Отображение клиентской части для абстрактных интерфейсов в большинстве идентично отображению для объектных ссылок, кроме:

- Си++ классы для абстрактных интерфейсов производятся от **CORBA::AbstractBase**, а не от **CORBA::Object.**
- Потому, что классы абстрактных интерфейсов могут служить как базовые классы для поддерживаемых приложениями конкретных классов типов значений, они могут предоставлять защищенный конструктор по умолчанию, защищенный конструктор копирования, и защищенный деструктор (который является виртуальным из-за особенностей наследования от **AbstractBas**e).
- Отображение для классов объектных ссылок не определяет тип наследования используемого для базовых классов объектных ссылок. Однако, в виду того, что абстрактные интерфейсы могут служитьт как базовые классы для поддерживаемых приложениями конкретных **valuetype** классов, которые сами могут производится от правильных классов интерфейсов, классов абстрактных интерфейсов могут всегда быть унаследованы как открытые виртуальные базовые классы.
- Вставка ссылки на абстрактный интерфейс в **CORBA::Any** происходит полиморфно, иначе объектная ссылка или **valuetype** на который указывает ссылка абстрактного интерфейса это то, что реально получает вставку в **Any.** Потому, что абстрактный интерфейс не может реально быть вставлен в **Any**, нет необходимости в операции извлечения абстрактного интерфейса. Тип **CORBA::Any::to_abstract_base** позволяет содержание **Any** извлеч как **AbstractBase** если сущность хранится в **Any** как тип объектной ссылки или **valuetype** прямо или косьвенно производный от от базового классса **AbstractBase**.

В другом случае, отображение для абстрактных интерфейсов идентично отображению для правильных интерфейсов, включая предоставляемые типыами **_var**, **_out**, типами **manager** для структур, последовательностей, членами массивов, идентичным управлением памятью, и идентичными операциями сигнатуры Си++.

Оба интерфейса которые производятся от одного или более абстрактных интерфейсов, и типов значений которые поддерживают один или более абстрактный интерфейс поддерживающий неявное расширение к типу **_ptr** для каждого абстрактного интерфейса базового класса. В частности, **T*** для **valuetype T** и **T_ptr** типа для интерфейса типа **T** поддерживается неявное расширение к типу **Base_ptr** для абстрактных интерфейсов типа **Base.** Есть только одно исключение из этого правила для типов значений которые напрямую или косьвенно поддерживают один или более правильных типов интерфейсов. В этом случае, существует объектная ссылка для типа значения, не указатель на тип значения, которая поддерживает расширение к базовому абстрактному интерфейсу.

1.19. Отображение для Типов Исключений

Исключение OMG IDL отображается в класс C++, который производится от стандартного класса **UserException**, определенного в модуле **CORBA**. Генерируемый класс является подобным структуре переменной длины, независимо от того, хранит ли или нет исключение какие -либо поля переменной длины. Как и для структур переменной длины, каждый член исключения должен быть самоуправляемым в отношении к своей памяти. Поля строк и широких строк исключения должны быть инициализированы в пустую строку (**""** и **L""** соответственно) с помощью конструктора по умолчанию для исключения.

Конструктор копирования, оператор присваивания и деструктор автоматически копирует или освобождает память, связанную с исключением. Для удобства отображение также определяет конструктор с параметрами для каждого поля исключения - этот конструктор инициализирует поля исключения заданными значениями. Для типов исключений, которые имеют строковое поле, этот конструктор будет иметь **const char*** параметр, так как конструктор должен копировать строковый аргумент. Также, конструкторы для типов исключений, которые имеют поле объектной ссылки, должны вызывать **_duplicate** на соответствующей объектной ссылке параметра конструктора. Конструктор по умолчанию не выполняет явную инициализацию полей.

```
// C++

class Exception
{
   public:
      virtual ~Exception();
      virtual void _raise() const = 0;
};
```

Базовый класс **Exception** является абстрактным и не может быть обработан, за исключением случая, когда он является частью экземпляра производного класса. Он предоставляет одну чисто виртуальную функцию в иерархию исключений: функцию **_raise()**. Эта функция может быть использована, чтобы сказать экземпляру исключения вытолкнуть (**throw**) себя так, чтобы **catch** выражение могло захватить его с помощью более производного типа. Каждый класс, производный от **Exception** реализует **_raise()** так:

```
// C++

void SomeDerivedException::_raise() const
{
throw *this;
}
```

Для сред, которые не поддерживают управление исключениями, обратитесь к разделу 1.42.2 за информацией о функции **_raise()**.

Класс **UserException** является производным от базового класса **Exception**, он также определен в модуле **CORBA**.

Все стандартные исключения произведены от класса **SystemException**, также определенного в модуле **CORBA**. Подобно **UserException, SystemException** производен от базового класса **Exception**. Класс **SystemException** показан ниже.

```
// C++

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);
};
```

Конструктор по умолчанию для **SystemException** вызывает **minor()**, чтобы возвратить 0, и **completed()**, чтобы возвратить **COMPLETED_NO**.

Каждое специализированное системное исключение произведено от SystemException:

```
// C++

class UNKNOWN : public SystemException { ... };
class BAD_PARAM : public SystemException { ... };
// и т.д.
```

Все специализированные системные исключения определены внутри модуля **CORBA**.

Эта иерархия исключений позволяет любому исключению быть выполненным с помощью просто перехвата типа **Exception**:

```
// C++

try {
   ...

} catch (const Exception &exc) {
   ...

}
```

Альтернативно, все пользовательские исключения могут быть выполнены с помощью перехвата типа **UserException**, а все системные исключения - с помощью перехвата типа **SystemException**:

```
// C++

try {
      ...

} catch (const UserException &ue) {
      ...

} catch (const SystemException &se) {
      ...

}
```

Естественно, более специализированные типы могут также появляться в **catch** выражениях.

Исключения обычно выталкиваются по значению и обрабатываются по ссылке. Такой подход заставляет деструктор исключения освобождать память автоматически.

Класс **Exception** предоставляет для вызова вниз по иерархии исключений:

```
// C++

class UserException : public Exception
{
   public:
      static UserException *_downcast(Exception *);
      static const UserException *_downcast(

                        const Exception *

                );
      virtual void _raise() const = 0;
      // ...

};


class SystemException : public Exception


{
   public:
      static SystemException *_downcast(Exception *);
      static const SystemException *_downcast(

                        const Exception *

                );


      virtual void _raise() const = 0;


      // ...

};
```

Каждый класс исключения поддерживает перегруженную статичную членскую функцию, называемую **_downcast**. Параметр в вызовах **_downcast** - это указатель на константный (**const**) или неконстантный экземпляр базового класса **Exception**. Если параметр является нулевым указателем, **_downcast** возвратит нулевой указатель. Если фактический (во время исполнения) тип параметра исключения может быть расширен до запрашиваемого типа исключения, тогда **_downcast** возвратит верный указатель на параметр **Exception**. А иначе, **_downcast** возвратит нулевой указатель. Версия **_downcast**, перегруженная для получения указателя на **const Exception**, возвращает указатель на константу для того, чтобы сохранить **const**-корректность.

В отличие от операции **_narrow** на объектных ссылках, операция **_downcast** на исключениях является эквивалентной оператору C++ **dynamic_cast** в том, что она возвращает подходящий по типу указатель на тот же параметр исключение, а не указатель на новое исключение. Если исходное исключение выйдет за границы видимости или будет уничтожено другим способом, указатель, возвращенный с помощью **_downcast**, не будет больше верным.

Для переносимости приложения соответствующая реализация отображения на C++ строится, используя компиляторы C++, которые поддерживают стандартные механизмы информации о типе во время исполнения C++ (Run Time Type Information - RTTI), необходимые для поддержки преобразования указателей для иерархии **Exception**. RTTI поддерживает, среди прочего, определение типа времени исполнения объекта C++. В частности, оператор **dynamic_cast<T*>** позволяет преобразовать указатель от базового до более производного, если указанный объект является более производного типа. Этот оператор не является полезным для сужения объектных ссылок, так как он не может определить фактический тип удаленных объектов, но он может быть использован реализацией отображения на C++ для преобразования внутри иерархии исключений.

Предыдущие версии этого отображения предоставляли поддержку для преобразования указателя (даункастинга - downcast) через статичную членскую функцию, называемую **_narrow**, которая имела точно такую же семантику, как и **_downcast**. Из-за путаницы в отличиях управления памятью между функциями **_narrow** объектных ссылок и функциями **_narrow** исключений функция **_narrow** для исключений теперь заменена на **_downcast**. Переносимые приложения должны использовать **_downcast** для преобразовывания исключения, а не **_narrow**. Реализации ORB, которые предоставляют функции **_narrow** для исключений в целях обратной совместимости, должны предоставлять перегруженные **_narrow** функции и для константного, и для неконстантного указателя **Exception***, как и для **_downcast**.

1.19.1. Неизвестное пользовательское исключение (UnknownUserException)

Вызовы запросов, сделанные через DII, могут привести к исключениям, определенным пользователем, которые не могут быть полностью представлены в вызывающей программе, потому что специфический тип исключения был неизвестен во время компиляции. Отображение предоставляет **UnknownUserException**, чтобы такие исключения могли быть представлены в вызывающем процессе:

```
// C++

class UnknownUserException : public UserException
{
   public:
      Any &exception();
};
```

Как показано здесь, **UnknownUserException** является производным от **UserException**. Он предоставляет аксессорную функцию **exception()**, которая возвращает **Any**, содержащий фактическое исключение. Владение возвращаемым **Any** осуществляется **UnknownUserException**'ом - **Any** просто позволяет доступ к данным исключения. Соответствующие приложения никогда не должны возбуждать исключения типа **UnknownUserException** - он предназначен для использования с DII.

1.19.2. Вставка в и извлечение из Any для Исключений

Соответствующие реализации должны генерировать операторы *вставки в* и *извлечения из* **Any** (**operator<<=** и **operator>>=**, соответственно), которые позволяют корректно вставлять в и извлекать из **Any** все системные и пользовательские исключения. И копирующая, и некопирующая формы оператора вставки в **Any** должны быть предоставлены для всех системных и пользовательских исключений.

Дополнительно соответствующие реализации отображения должны поддерживать вставку в **Any** (но не извлечение) для **CORBA::Exception**. Это требуется, чтобы позволить DSI-основанным приложениям перехватывать исключения как **CORBA::Exception&** и сохранять их в **ServerRequest**:

```
// C++

try {
      // ...

} catch (Exception& exc) {
      Any any;
      any <<= exc;
      server_request->set_exception(any);
}
```

Заметим, что в результате и **TypeCode**, и значение для фактического производного типа исключения будут сохранены в **Any**. Обе формы (и копирующая, и некопирующая) вставки в **Any** для **CORBA::Exception** должны быть предоставлены:

```
// C++

void operator<<=(Any&, const Exception&);
void operator<<=(Any&, const Exception*);
```

1.20. Отображение для Операций и Атрибутов

Операция отображается в функцию C++ с таким же именем, как и операция. Каждый атрибут для чтения-записи отображается в пару перегруженных функций (обе с одинаковым именем), одна для установки (set) значения атрибута и одна для получения (get) значения атрибута. Функция установки (*set*) имеет входной (**in**) параметр с таким же типом, как и атрибут, в то время как функция получения (*get*) не имеет параметров и возвращает такой же тип, как у атрибута. Атрибут, помеченный "readonly", отображается только в одну функцию C++, для получения значения атрибута. Параметры и возвращаемые типы для функций атрибутов подчиняются таким же правилам передачи параметров, как и для обычных операций.

OMG IDL **oneway** операции отображаются так же, как и другие операции; то есть, нет способа узнать просмотром C++ кода, является ли операция **oneway** или нет.

Отображение не определяет, являются ли исключения, указанные для OMG IDL операции, частью сигнатуры типа генерируемой операции или нет.

```
// IDL

interface A
{
      void f();
      oneway void g();
      attribute long x;
};


// C++

A_var a;
a->f();
a->g();
Long n = a->x();
a->x(n + 1);
```

В отличие от отображения на C, операции C++ не требуют дополнительного параметра **Environment** для передачи информации об исключении - для этой цели используются настоящие C++ исключения.

1.21. Неявные Аргументы в Операциях

Если операция в OMG IDL спецификации имеет спецификацию контекста, тогда входной параметр **Context_ptr** дописывается за всеми аргументами, указанными в операции. В реализации, которая не поддерживает настоящие C++ исключения, выходной параметр **Environment** является последним аргументом, следующим за всеми указанными в операции аргументами, и следующим за аргументом контекста, если он присутствует.

1.22. Соображения о Передаче Аргументов

Отображение режимов передачи параметров пытается балансировать между эффективностью и простотой. Для примитивных типов, перечислений и объектных ссылок режимы являются непосредственными, передавая тип P для примитивных типов и перечислений и тип **A_ptr** для интерфейса типа A.

Агрегатные типы осложняются вопросом, когда и как выделяется и освобождается память параметра. Отображение **in** параметров является непосредственным, потому что память параметра является выделяемой вызывающим и только для чтения. Отображение для **out** и **inout** параметров более проблематично. Для типов переменной длины вызываемый должен выделить часть, если не всю память. Для типов с фиксированной длиной, таких как тип *Point*, представленный как структура, содержащая три вещественных поля, выделение памяти вызывающим является предпочтительным (чтобы позволить выделение в стеке).

Для обеспечения обоих видов выделения памяти, обхода потенциальной путаницы раздельного выделения и устранения путаницы в отношении, когда выполняется копирование, принято отображение **T&** для агрегатов фиксированной длины **T** и **T*&** для агрегатов переменной длины **T**. Такой подход имеет неудачное следствие в том, что использование для структур зависит от того, является ли структура фиксированной или переменной длины, однако, последовательно принято отображение **T_var&**, если вызывающий использует управляющий тип **T_var**.

Отображение для **out** и **inout** параметров дополнительно требует поддержку для освобождения любых предыдущих данных переменной длины в параметре, когда передается **T_var**. Несмотря на то, что их начальные значения не передаются в операцию, мы включаем **out** параметры, потому что параметр может содержать результат от предыдущего вызова. Существует много способов реализовать такую поддержку. Отображение не требует специальной реализации, но соответствующая реализация должна освобождать недоступную память, связанную с параметром, переданным как **T_var** управляющий тип. Предоставление **T_out** типов предназначено для дачи реализациям зацепок, необходимых для освобождения недоступной памяти при конвертировании из **T_var** типов. Следующие примеры демонстрируют подобное поведение:

```
// IDL

struct S { string name; float age; };
void f(out S p);


// C++

S_var s;
f(s);
// использование s

f(s); // первый результат будет освобожден


S *sp; // не нужно инициализировать перед передачей в out

f(sp);
// использование sp

delete sp; // cannot assume next call will free old value

f(sp);
```

Заметим, что неявное освобождение предыдущих значений для **out** и **inout** параметров работает только с **T_var** типами, а не с другими:

```
// IDL

void q(out string s);


// C++

char *s;
for (int i = 0; i < 10; i++)
```

```
      q(s); // утечка памяти!
```

Каждый вызов функции **q** в цикле приводит к утечке памяти, потому что вызывающий не выполняет **string_free** над **out** результатом. Существуют два пути исправить это, как показано ниже:

```
// C++

char *s;
String_var svar;
for (int i = 0 ; i < 10; i++) {
      q(s);
      string_free(s);// явное освобождение
      // ИЛИ:
      q(svar); // неявное освобождение
}
```

Использование обыкновенного **char*** для **out** параметра означает, что вызывающий должен явно освободить его память перед каждым повторным использованием переменной как **out** параметра, в то время как использование **String_var** означает, что любое освобождение выполняется неявно при каждом использовании переменной как **out** параметра.

Если строки или широкие строки передаются как **inout** параметры, вызываемый может модифицировать содержимое строки или широкой строки по месту. Однако, если новая строка или широкая строка длиннее, чем начальная строка или широкая строка, становится необходимым перевыделение памяти. Для новой строки или широкой строки, которая короче, чем исходная строка или широкая строка, перевыделение памяти может также использоваться для сохранения памяти. Однако, укорачивание строки или широкой строки с помощью замены символа, который является частью начальной строки или широкой строки, на подходящий NUL символ также является допустимым.

Для **inout** объектных ссылок, перевыделение памяти необходимо всякий раз, когда вызываемому необходимо изменить начальное значение ссылки. Пример ниже иллюстрирует это.

Для входных (**in**) **valuetypes** вызываемый будет принимать копию каждого **valuetype** аргумента, переданного ему, даже если вызывающий и вызываемый расположены в одном и том же процессе. Вызываемому позволяется вызывать операции и модифицирующие функции, которые изменяют состояние экземпляра **valuetype**, но состояние копии вызывающего этого экземпляра **valuetype** не будет изменяться под воздействием изменений состояния вызываемого. Это требуется для сохранения прозрачности местоположения для операций интерфейса.

Для **inout valuetype**'ов, вызываемый может либо изменять входящий экземпляр **valuetype**, либо заменить входящий указатель на указатель другого экземпляра **valuetype**. Вызываемый должен вызвать **_remove_ref** над **valuetype** экземпляром, переданным в него, перед его заменой на экземпляр **valuetype**, который будет передан обратно из вызываемого. Вызыющий обычно будет вызывать **_remove_ref** над **valuetype** экземпляром, который он получит обратно как **inout**, **out** или возвращаемое значение.

Пример ниже иллюстрирует замену **inout** аргументов. Для операции **f**, **s1** - это **inout** строка, которая изменяется по месту и чья длина не изменяется вызываемым, **s2** - это **inout** строка, которая увеличивается вызываемым, **obj** - это **inout** объектная ссылка, которая изменяется вызываемым, **val1** - это **inout valuetype**, которое изменяется по месту вызываемым, и **val2** - это **inout valuetype**, которое замещается вызываемым. Код примера использует локальные **T_var** переменные для гарантии автоматического освобождения, но вместо этого могут быть использованы явные вызовы **CORBA::string_free** и **CORBA::release.**

```
// IDL

valuetype V { public long state; };
interface A {
      void f(inout string s1, inout string s2, inout A obj,
             inout V val1, inout V val2);
};
```

```
// C++

void Aimpl::f(char *&s1, char *&s2, A_ptr &obj,
            V *&val1, V *&val2)
{
      // Преобразует s1 к верхнему регистру на месте
      while (*s1 != '\0') to upper(*s1++);


      // Возвращает другое строковое значение для s2

      String_var s2_tmp = s2;
      s2 = string_dup("new s2");


      // Присваивает новое значение в obj

      A_ptr newobj = ...

      A_var obj_tmp = obj;
      obj = A::_duplicate(newobj);


      // Изменяет значение val1 на месте
      if (val1 != 0) val1->state(42);


      // Замещает val2 полностью
      CORBA::remove_ref(val2);
      val2 = new MyVImpl(1234);
}
```

Для параметров, которые передаются или возвращаются как указатель (**T***) или ссылка на указатель (**T*&**), за исключением **valuetype**, соответствующей программе не позволено передавать или возвращать нулевой указатель, результат таких действий неопределен. В частности, вызывающий не может передать нулевой указатель при любом из следующих условий:

- **in** и **inout** строка
- **in** и **inout** массив (указатель на первый элемент)

Однако, вызывающий может передать ссылку на указатель с нулевым значением для **out** параметров, так как вызываемый не проверяет значение, а только перезаписывает его. Кроме того, соответствующие приложения также могут передавать и возвращать нулевые указатели для всех **valuetype** параметров и возвращаемых типов, и могут внедрять нулевые **valuetype** указатели внутри сконструированных типов, которые передаются как параметры или возвращаемые значения, такие как структуры, объединения, массивы, последовательности, **Any** и другие **valuetype**. Вызываемый не может возвратить нулевой указатель при любом из следующих условий:

- **out** и возвращаемая структура переменной длины
- **out** и возвращаемое объединение переменной длины
- **out** и возвращаемая строка
- **out** и возвращаемая последовательность
- **out** и возвращаемый массив переменной длины, возвращаемый массив фиксированной длины
- **out** и возвращаемый any

Так как OMG IDL не имеет концепции указателей в целом или нулевых указателей в частности, за исключением **valuetype**, разрешение передачи нулевых указателей в и из операции будет проецировать C++ семантику на OMG IDL операции. Соответствующая реализация допустима, но не требует возбуждать исключение BAD_PARAM, если она обнаружит такую ошибку.

1.22.1. Параметры и Сигнатуры Операций

Таблица 1-3 показывает отображение для базовых режимов передачи параметров OMG IDL и возвращаемого типа в соответствии с переданным или возвращаемым типом, в то время как таблица 1-4 показывает ту же информацию для **T_var** типов. Раздел "Передача Аргумента и Результата T_var" просто для информационного назначения; ожидается, что сигнатуры операций и для клиентов, и для серверов будут написаны в терминах режимов передачи параметров, показанных в разделе "Передача Аргумента и Результата Базового Типа", *с исключением, что **T_out** типы будут использованы как типы фактических параметров для всех **out** параметров*. Также ожидается, что типы **T_var** будут поддерживать необходимые операторы преобразования, чтобы позволить им быть переданными непосредственно. Вызывающие должны всегда передавать экземпляры либо **T_var** типов, либо базовых типов, показанных в разделе "Передача Аргумента и Результата Базового Типа", а вызываемые должны трактовать свои **T_out** параметры, как если бы они были фактически соответственных типов, лежащих в основе, показанных в таблице 1-3.

В таблице 1-3 массивы фиксированной длины - единственный случай, где тип **out** параметра отличается от возвращаемого значения, это необходимо, потому что C++ не позволяет функции возвращать массив. Отображение возвращает указатель на *кусок* (slice) массива, где кусок - это массив со всеми размерностями исходного, за исключением первого. Вызывающий ответственен за предоставление памяти для всех аргументов, переданных как входные (**in**) параметры.

Таблица 1-3 Передача аргумента и результата базового типа

Тип данных	Входной In	Inout	Выходной Out	Возвращаемое значение (Return)
short	Short	Short&	Short&	Short
long	Long	Long&	Long&	Long
long long	LongLong	LongLong&	LongLong&	LongLong
unsigned short	UShort	UShort&	UShort&	UShort
unsigned long	ULong	ULong&	ULong&	ULong
unsigned long long	ULongLong	ULongLong&	ULongLong&	ULongLong
float	Float	Float&	Float&	Float
double	Double	Double&	Double&	Double
long double	LongDouble	LongDouble&	LongDouble&	LongDouble
boolean	Boolean	Boolean&	Boolean&	Boolean
char	Char	Char&	Char&	Char
wchar	WChar	WChar&	WChar&	WChar
octet	Octet	Octet&	Octet&	Octet
enum	enum	enum&	enum&	enum
object reference ptr[1]	objref_ptr	objref_ptr&	objref_ptr&	objref_ptr
struct фиксированной длины	const struct&	struct&	struct&	struct
struct переменной длины	const struct&	struct&	struct*&	struct*
union фиксированной длины	const union&	union&	union&	union
union переменной длины	const union&	union&	union*&	union*
String	const char*	char*&	char*&	char*
Wstring	const WChar*	WChar*&	WChar*&	WChar*
Sequence	const sequence&	sequence&	sequence*&	sequence*
array фиксированной длины	const array	array	array	array slice*[2]

array переменной длины	const array	array	array slice*&[2]	array slice*[2]
Any	const any&	any&	any*&	any*
Fixed	const fixed&	fixed&	fixed&	fixed
valuetype[3]	valuetype*	valuetype*&	valuetype*&	valuetype*

1. Включая ссылки псевдообъектов.
2. Кусок (slice) - это массив во всеми размерностями исходного массива, исключая первую.
3. Включая наборы значений (value boxes).

Таблица 1-4 Передача аргумента и результата T_var[1]

Тип данных	Входной In	Inout	Выходной Out	Возвращаемое значение (Return)
object reference var[2]	const objref_var&	objref_var&	objref_var&	objref_var
struct_var	const struct_var&	struct_var&	struct_var&	struct_var
union_var	const union_var&	union_var&	union_var&	union_var
string_var	const string_var&	string_var&	string_var&	string_var
Sequence_var	const sequence_var&	sequence_var&	sequence_var&	sequence_var
array_var	const array_var&	array_var&	array_var&	array_var
any_var	const any_var&	any_var&	any_var&	any_var
valuetype_var[3]	const valuetype_var&	valuetype_var&	valuetype_var&	valuetype_var

1. Типы с фиксированной точкой (Fixed) не имеют соответствующего _var типа и поэтому не показаны в этой таблице.
2. Включая ссылки псевдообъектов.
3. Включая наборы значений (value boxes).

Таблицы "Ответственность вызывающего за память аргумента" и "Варианты передачи аргументов" описывают ответственность вызывающего за память, связанную с **inout** и **out** параметрами и возвращаемыми результатами.

Таблица 1-5 Ответственность вызывающего за память аргумента

Тип	Inout параметр	Out параметр	Возвращаемый результат
short	1	1	1
long	1	1	1
long long	1	1	1
unsigned short	1	1	1
unsigned long	1	1	1
unsigned long long	1	1	1
float	1	1	1
double	1	1	1
long double	1	1	1
boolean	1	1	1
char	1	1	1
wchar	1	1	1
octet	1	1	1
enum	1	1	1
object reference ptr	2	2	2

struct фиксированной длины	1	1	1
struct переменной длины	1	3	3
string	4	3	3
wstring	4	3	3
sequence	5	3	3
array фиксированной длины	1	1	6
array переменной длины	1	6	6
any	5	3	3
fixed	1	1	1
valuetype	7	7	7

Таблица 1-6 Варианты передачи аргумента

Вариант	
1	Вызывающий выделяет всю необходимую память, за исключением той, которая может быть инкапсулирована и управляема самим параметром. Для inout параметров вызывающий предоставляет начальное значение, и вызываемый может изменить это значение. Для out параметров вызывающий выделяет память, но не нужно ее инициализировать, и вызываемый устанавливает значение. Функция возвращает данные по значению.
2	Вызывающий выделяет память для объектной ссылки. Для inout параметров вызывающий предоставляет начальное значение; если вызываемый хочет переприсвоить inout параметр, он сначала вызовет CORBA::release над исходным входным значением. Для продолжения использования объектной ссылки, передаваемой как inout, вызывающий сначала должен сдублировать ссылку. Вызывающий является ответственным за освобождение всех out и возвращаемых объектных ссылок. Освобождение всех объектных ссылок, внедренных в другие структуры выполняется автоматически самими структурами.
3	Для out параметров вызывающий выделяет указатель и передает его по ссылке в вызываемый. Вызываемый устанавливает указатель, чтобы он указывал на верный экземпляр типа параметра. Для возвращаемых значений вызываемый возвращает подобный указатель. Вызываемому не позволено возвратить нулевой указатель в любом случае. В обоих случаях вызывающий является ответственным за освобождение возвращаемой памяти. Для поддержания локальной/удаленной прозрачности вызывающий всегда должен освобождать возвращаемую память, независимо от того, расположен ли вызываемый в том же адресном пространстве, что и вызывающий, или он расположен в другом адресном пространстве. После выполнения запроса вызывающему не позволено модифицировать любые значения в возвращаемой памяти - чтобы сделать так, вызывающий сначал должен скопировать экземпляр в новый экземпляр, и затем изменить новый экземпляр.
4	Для inout строк вызывающий предоставляет память и для входной строки, и для char* или wchar*, указывающих на нее. Так как вызываемый может освободить входную строку и переприсвоить char* или wchar* для указания на новую память, содержащую выходное значение, вызывающий должен выделить входную строку, используя string_alloc() или wstring_alloc(). Длина выходной строки, следовательно, не ограничена длиной входной строки. Вызывающий является ответственным за удаление памяти для out строки, используя string_free() или wstring_free(). Вызываемому не позволено возвращать нулевой указатель для inout, out или возвращаемого значения.
5	Для inout последовательностей и any, присваивание или модификация последовательности или any может вызвать освобождение собственной памяти перед наступлением перевыделения, в зависимости от состояния параметра Boolean release, с которым была построена последовательность или any.
6	Для out параметров вызывающий выделяет указатель на кусок массива, который имеет те же размерности, как и исходный массив, за исключением первого, и передает указатель по ссылке вызываемому. Вызываемый устанавливает указатель, чтобы он указывал на правильный экземпляр массива.Для возвращаемых значений вызываемый возвращает такой же указатель. Вызываемому не позволено возвратить нулевой указатель в любом случае. В обоих случаях вызывающий является ответственным за освобождение возвращаемой памяти. Для поддержания локальной/удаленной прозрачности вызывающий всегда должен освобождать возвращаемую память, независимо от того, расположен ли вызываемый в том же адресном пространстве, что и вызывающий, или он расположен в другом адресном пространстве. После выполнения запроса вызывающему не позволено изменять

	любые значения в возвращаемой памяти - чтобы так сделать, вызывающий должен сначала скопировать возвращаемый экземпляр массива в новый экземпляр массива, а затем изменить новый экземпляр.
7	Вызывающий выделяет память для экземпляра valuetype. Для inout параметров вызывающий предоставляет начальное значение; если вызываемый желает переприсвоить inout значение указателя, чтобы указывать на другой экземпляр valuetype, он сначала вызовет _remove_ref над исходным входным valuetype. Для продолжения использования экземпляра, переданного как inout, после того, как вызванная операция возвратит управление, вызывающий должен сначала выполнить _add_ref над экземпляром valuetype. Вызывающий является ответственным за выполнение _remove_ref над всеми out и возвращаемыми экземплярами valuetype. Понижение счетчиков ссылок через _remove_ref для всех экземпляров valuetype, внедренных в другие структуры, выполняется автоматически самими структурами.

1.23. Отображение для Псевдообъектов на C++

Псевдообъекты CORBA могут быть реализованы либо как обычные CORBA объекты, либо как *безсерверные объекты*. В спецификации CORBA фундаментальные отличия между этими стратегиями следующие:

- Типы безсерверных объектов не наследуются от **CORBA::Object**
- Отдельные безсерверные объекты не регистрируются ни с каким ORB
- Безсерверным объектам нет необходимости следовать тем же правилам управления памятью, что и для обычных IDL типов.

Ссылкам на безсерверные объекты нет необходимости быть верными в разных вычислительных контекстах; к примеру, адресных пространствах. Взамен, ссылки на безсерверные объекты, которые передаются как параметры, могут выразиться в построении независимых функционально-идентичных копий объектов, используемых принимающими эти ссылки. Для поддержки этого, остальные скрытые репрезентативные свойства (такие как размещение данных) безсерверных объектов делаются известными в ORB. Спецификация для достижения этого не включена в этот раздел. Создание безсерверных объектов, известных в ORB, является реализационными деталями.

Этот раздел предоставляет стандартный алгоритм отображения для всех типов псевдообъектов. Это избегает необходимости частичных отображений для каждого из девяти типов CORBA псевдообъектов, вмещает все типы псевдообъектов, которые могут понадобиться в будущих изменениях CORBA. Он также избегает зависимости представления в отображении на C, в то же время позволяя реализации, которые основаны на C-совместимых представлениях.

1.24. Использование

Скорее чем C-PIDL, это отображение использует расширенную форму полного OMG IDL для описания типов безсерверных объектов. Интерфейсы для типов псевдообъектов следуют точно таким же правилам, как и обычные OMG IDL интерфейсы, со следующими исключениями:

- Они предваряются ключевым словом **pseudo**.
- Их описания могут ссылаться на другие типы безсерверных объектов, которые иначе недоступны в OMG IDL.

Как объяснялось в разделе 1.23 префикс **pseudo** означает, что интерфейс может быть реализован либо в обычной, либо в безсерверной форме. Т.е., применяются либо правила, описанные в следующих разделах, либо обычные правила отображения, описанные в этом разделе.

1.25. Правила отображения

Безсерверные объекты отображаются такими же способами, что и обычные интерфейсы, за исключением отличий, вынесенных в этот раздел.

Классы, представляющие типы безсерверных объектов *не являются* подклассами **CORBA::Object**, им необязательно быть подклассами любых других C++ классов. Поэтому, они не обязаны поддерживать, к примеру, операцию Object::create_request.

Для каждого класса, представляющего тип безсерверного объекта T, предоставляются перегруженные версии следующих функций в пространстве имен **CORBA**:

```
// C++

void release(T_ptr);
Boolean is_nil(T_ptr p);
```

Не гарантируется, что отображаемые C++ классы могут быть полезны для получения подклассов пользователями, хотя подклассы могут быть предоставлены реализациями. Реализациям позволено сделать предположения о внутренних представлениях и транспортных форматах, которые не могут быть применены к подклассам.

Членским функциям классов, представляющих типы безсерверных объектов, нет необходимости следовать обычным правилам управления памятью. Это следует из факта, что некоторые безсерверные объекты, такие как CORBA::NVList, по существу являются контейнерами для нескольних уровней других безсерверных объектов. Требование к вызывающим явно освобождать значения, возвращаемые из аксессорных функций, для включенных безсерверных объектов будет противоречить их предназначенному использованию.

Все остальные элементы отображения такие же. В частности:

1. Типы ссылок на безсерверные объекты, **T_ptr**, могут или не могут быть просто определениями типа (typedef) **T***.
2. Каждый отображаемый класс поддерживает следующие статичные членские функции:

```
// C++

static T_ptr _duplicate(T_ptr p);
static T_ptr _nil();
```

Допустимые реализации _duplicate включают просто возврат аргумента или построение ссылок на новый экземпляр. Отдельные реализации могут предоставлять более жесткие гарантии о поведении.

1. Соответствующие C++ классы могут или не могут быть непосредственно обрабатываемыми или иметь другие ограничения на обработку. Для переносимости пользователи должны вызывать подходящие конструктивные операции.
2. Как и с обычными интерфейсами, операторы присваивания не поддерживаются.
3. Хотя они могут прозрачно употреблять механизмы "стиля копии" преимущественно, чем "стиля ссылки", сигнатуры и правила передачи параметров, а заодно правила управления памятью являются идентичными таким же для обычных объектов, кроме описанных отличий.

1.26. Отношение к PIDL Отображению на C

Все интерфейсы безсерверных объектов и декларации, которые основаны на них, имеют непосредственные аналоги в отображении на C. Отображающие C++ классы могут, но не обязаны, быть реализованными, используя представления, совместимые с теми, которые избраны для отображения на C. Отличия между спецификациями псевдообъектов для C-PIDL и C++ PIDL следующие:

- C++-PIDL предусматривает устранение зависимостей представления через использование преимущественно интерфейсов,а не структур и определений типов (typedef).
- C++-PIDL предусматривает размещение операций над псевдообъектами в их интерфейсах, включая несколько случаев модернизированной функциональности, как описано.
- В C++-PIDL **release** выполняет роль связанных **free** и **delete** операций в отображении на C, если не оговорено иначе.

Краткие описания и листинги (исходные тексты) каждого псевдоинтерфейса и его отображения на C++ предоставляются в следующих разделах. Дальнейшие детали, включая описания типов, связанных, но не описанных ниже, могут быть найдены в соответствующих разделах этой спецификации.

Некоторые из псевдоинтерфейсов, показанных в этом разделе, основаны на исключении, определенном пользователем, поставляемом в модуле **CORBA** реализациями ORB. Это исключение называется **Bounds** и определяется как следующее:

```
// IDL

module CORBA
{
      exception Bounds {};
      // ...

};
```

Замечу, что это исключение не является тем же, как исключение **CORBA::TypeCode::Bounds**.

1.27. Среда (Environment)

Environment предоставляет средство для работы с исключениями в тех случаях, где истинные механизмы исключений недоступны или нежелательны (к примеру в DII). Они (исключения) могут быть установлены и проверены, используя атрибут **exception**.

Как и обычные OMG IDL атрибуты, атрибут **exception** отображается в пару C++ функций, используемых для установки и получения исключения. Семантика **set** и **get** функций, однако, является отчасти отличной, чем для обычных OMG IDL атрибутов. Функция **set** C++ присваивает владение указателем **Exception**, переданным ей. **Environment** обычно вызывает **delete** над этим указателем, поэтому **Exception**, на который ссылается указатель, должен быть динамически выделен вызывающим. Функция **get** возвращает указатель на **Exception**, как атрибута, являющегося структурой переменной длины, но указатель ссылается на память, которой владеет **Environment**. Как только **Environment** будет уничтожен, указатель не будет больше верным. Вызывающий не должен вызывать **delete** над указателем **Exception**, возвращаемым с помощью **get** функции. **Environment** является ответственным за освобождение любого исключения, которое он содержит, когда уничтожается сам. Если **Environment** не содержит исключения, **get** функция возвращает нулевой указатель.

Функция **clear ()** заставляет **Environment** удалить любое исключение, которое он содержит. Не является ошибкой вызвать **clear ()** над **Environment**, не содержащим исключения. Передача нулевого указателя в функцию установки исключения является эквивалентным вызову **clear ()**. Если **Environment** содержит информацию об исключении, вызывающий является ответственным за вызов **clear ()** над ним перед передачей его в операцию.

1.27.1. Интерфейс среды (Environment)

```
// IDL

pseudo interface Environment
{
   attribute exception exception;
   void clear();
};
```

1.27.2. C++ класс Environment

```
// C++

class Environment
{
   public:
      void exception(Exception*);
      Exception *exception() const;
      void clear();
};
```

1.27.3. Отличия от C-PIDL

Спецификация C++-PIDL отличается от C-PIDL спецификации в следующем:

- Определяет интерфейс, а не структуру.
- Поддерживает атрибут, разрешающий операции над значениями исключения в целом, а не над большим числом чисел и/или строк идентификации.
- Поддерживает функцию **clear ()**, которая используется для уничтожения любого исключения **Exception**, которое может содержать **Environment**.
- Поддерживает конструктор по умолчанию, который инициализирует среду, без добавления информации об исключении.

1.27.4. Управление памятью

Environment имеет следующие специальные правила управления памятью:

- Членская функция **void exception(Exception*)** заимствует **Exception***, переданный в нее.
- Владение возвращаемым значением членской функции **Exception *exception()** осуществляется с помощью Environment, это возвращаемое значение не должно освобождаться вызывающим.

1.28. Именованное значение (NamedValue)

NamedValue используется только как элемент **NVList**, особенно в DII. **NamedValue** содержит имя (дополнительно), **any** значение и маркировачные флаги. Допустимые значения флагов - **ARG_IN, ARG_OUT** и **ARG_INOUT**.

Значением в **NamedValue** можно манипулировать через стандартные операции над **any**.

1.28.1. Интерфейс NamedValue

```
// IDL

pseudo interface NamedValue
{
      readonly attribute Identifier name;
      readonly attribute any value;
      readonly attribute Flags flags;
};
```

1.28.2. C++ класс NamedValue

```
// C++

class NamedValue
{
   public:
      const char *name() const;
      Any *value() const;
      Flags flags() const;
};
```

1.28.3. Отличия от C-PIDL

Спецификация C++-PIDL отличается от C-PIDL спецификации в следующем:

- Определяет интерфейс, а не структуру.
- Не предоставляет аналог поля **len**.

1.28.4. Управление памятью

NamedValue имеет следующие специальные правила управления памятью:

- Владение возвращаемыми значениями функций **name()** и **value()** осуществляется **NamedValue**, эти возвращаемые значения не должны освобождаться вызывающим.

1.29. Список именованных значений (NVList)

NVList это список именованных значений (**NamedValue**). Новый **NVList** строится, используя операцию **ORB::create_list**. Новы именованные значения могут быть построены как часть **NVList**, любым из трех способов:

- **add** - создает неименованное значение, инициализируя только флаги.
    - **add_item** - инициализирует имя и флаги.

- **add_value** - инициализирует имя, значение и флаги.
- **add_item_consume** - инициализирует имя и флаги, принимая ответственность за управление памятью для **char*** параметра имени.
- **add_value_consume** - инициализирует имя, значение и флаги, принимая ответственность за управление памятью и для **char*** параметра имени, и для **Any*** параметра значения. Каждая из этих операций возвращает новый элемент.

Елементы могут быть получены и удалены через индексирование от нуля. Функции **add, add_item, add_value, add_item_consume** и **add_value_consume** удлиняют **NVList** для сохранения нового элемента каждый раз, когда они вызываются. Функция **item** может быть использована для доступа к существующим элементам.

1.29.1. Интерфейс NVList

```
// IDL

pseudo interface NVList
{
   readonly attribute unsigned long count;
   NamedValue add(in Flags flags);
   NamedValue add_item(in Identifier item_name, in Flags flags);
   NamedValue add_value(

      in Identifier item_name,
      in any val,
      in Flags flags

   );
   NamedValue item(in unsigned long index) raises(Bounds);
   void remove(in unsigned long index) raises(Bounds);
};
```

1.29.2. C++ класс NVList

```
// C++

class NVList
{
   public:
      ULong count() const;
      NamedValue_ptr add(Flags);
      NamedValue_ptr add_item(const char*, Flags);
      NamedValue_ptr add_value(

            const char*,
            const Any&,
            Flags

      );
      NamedValue_ptr add_item_consume(

            char*,
            Flags

      );
      NamedValue_ptr add_value_consume(

            char*,
            Any *,
            Flags

      );
      NamedValue_ptr item(ULong);
      void remove(ULong);
};
```

1.29.3. Отличия от C-PIDL

Спецификация C++-PIDL отличается от спецификации C-PIDL в ледующем:

- Определяет интерфейс, а не определение типа (typedef)
- Предоставляет другие сигнатуры операций, которые добавляют элементы, для того, чтобы обойти зависимости представления
- Предоставляет методы доступа по индексу

1.29.4. Управление памятью

NVList имеет следующие специальные правила управления памятью:

- Владение возвращаемыми значениями функций **add, add_item, add_value, add_item_consume, add_value_consume** и **item** осуществляется **NVList**, эти возвращаемые значения не должны освобождаться вызывающим.
- Параметры **char*** в функциях **add_item_consume** и **add_value_consume** и параметр **Any*** в функции **add_value_consume** потребляются **NVList**'ом. Вызывающий не может получить доступ к этим данным после того, как они были переданы в эти функции, потому что **NVList** может скопировать их и немедленно удалить исходные. Вызывающий должен использовать операцию **NamedValue::value()** для того,чтобы изменять атрибут **value**, лежащий в основе **NamedValue**, если необходимо.
- Функция **remove** также вызывает **CORBA::release** над удаляемым **NamedValue**.

1.30. Запрос (Request)

Request предоставляет первичную поддержку для DII. Новый запрос над отдельным целевым объектом может быть построен, используя краткую версию операции создания запроса, показанную в разделе 1.34 "Объект":

```
// C++

Request_ptr Object::_request(Identifier operation);
```

Параметры и контексты могут быть добавлены после построения через соответствующие атрибуты в интерфейсе **Request**. Результаты, выходные параметры и исключения получаются подобным же образом после вызова. Следующий C++ код иллюстрирует использование:

```
// C++

Request_ptr req = anObj->_request("anOp");
*(req->arguments()->add(ARG_IN)->value()) <<= anArg;
// ...

req->invoke();
if (req->env()->exception() == 0) {
      *(req->result()->value()) >>= aResult;
}
```

В то время как этот пример показывает семантику основанной на атрибутах аксессорной функции, следующий пример показывает, что намного проще и предпочтительней использовать вспомогательные функции манипулирования эквивалентными параметрами:

```
// C++
```

```
Request_ptr req = anObj->_request("anOp");
req->add_in_arg() <<= anArg;
// ...

req->invoke();
if (req->env()->exception() == 0) {
       req->return_value() >>= aResult;
}
```

Альтернативно, запросы могут быть построены, используя одну из длинных форм операции создания, показанную в интерфейсе Object.

```
// C++

void Object::_create_request(

            Context_ptr ctx,
            const char *operation,
            NVList_ptr arg_list,
            NamedValue_ptr result,
            Request_out request,
            Flags req_flags

);
void Object::_create_request(

            Context_ptr ctx,
            const char *operation,
            NVList_ptr arg_list,
            NamedValue_ptr result,
            ExceptionList_ptr,
            ContextList_ptr,
            Request_out request,
            Flags req_flags

);
```

Использование является таким же, как и для краткой формы, за исключением того, что все параметры вызова устанавливаются при построении. Заметим, что флаги **OUT_LIST_MEMORY** и **IN_COPY_VALUE** могут быть установлены как флаги в параметре **req_flags**, но они являются незначащими и поэтому игнорируются, так как вставка и извлечение параметра делается через тип **Any**.

Request также позволяет приложению поставлять всю информацию, необходимую для того, чтобы он был выполнен без требования к ORB инициализировать Репозитарий Интерфейсов (Interface Repository). Для того, чтобы доставить запрос и получить ответ, ORB требует:

- ссылку на целевой объект
- имя операции
- список агрументов (необязательно)
- место для помещения результата (необязательно)
- место для помещения любых возвращаемых исключений
- **Context** (необязательно)
- список определенных пользователем исключений, которые могут быть возбуждены (необязательно)

Так как операция **Object::create_request** позволяет указывать все элементы из этого списки, исключая два последних, ORB может использовать репозитарий интерфейсов для того, чтобы их обнаружить. Некоторые приложения, однако, могут не хотеть , чтобы ORB выполнял потенциально дорогостоящий просмотр Interface Repository во время вызова запроса, поэтому взамен были добавлены два новых безсерверных объекта для разрешения приложению указывать данную информацию:

- **ExceptionList**: позволяет приложению предоставить список кодов типов (TypeCode) для всех определенных пользователем исключений, которые могут получиться, когда вызывается **Request**.

- **ContextList**: позволяет приложению предоставить список **Context** строк, которые должны поставляться с вызовом **Request.**

1.30.1. Интерфейс Request

```
// IDL

pseudo interface ExceptionList
{
   readonly attribute unsigned long count;
   void add(in TypeCode exc);
   TypeCode item(in unsigned long index) raises(Bounds);
   void remove(in unsigned long index) raises(Bounds);
};


pseudo interface ContextList
{
   readonly attribute unsigned long count;
   void add(in string ctxt);
   string item(in unsigned long index) raises(Bounds);
   void remove(in unsigned long index) raises(Bounds);
};


pseudo interface Request
{
   readonly attribute Object target;
   readonly attribute Identifier operation;
   readonly attribute NVList arguments;
   readonly attribute NamedValue result;
   readonly attribute Environment env;
   readonly attribute ExceptionList exceptions;
   readonly attribute ContextList contexts;


   attribute context ctx;


   void invoke();
   void send_oneway();
   void send_deferred();
   void get_response();
   boolean poll_response();
};
```

1.30.2. C++ класс Request

```
// C++

class ExceptionList
{
   public:
      ULong count();
      void add(TypeCode_ptr tc);
      void add_consume(TypeCode_ptr tc);
      TypeCode_ptr item(ULong index);
      void remove(ULong index);
};


class ContextList
{
   public:
      ULong count();
      void add(const char* ctxt);
      void add_consume(char* ctxt);
      const char* item(ULong index);
      void remove(ULong index);
};


class Request
{
   public:
      Object_ptr target() const;
```

```
      const char *operation() const;
      NVList_ptr arguments();
      NamedValue_ptr result();
      Environment_ptr env();
      ExceptionList_ptr exceptions();
      ContextList_ptr contexts();


      void ctx(Context_ptr);
      Context_ptr ctx() const;


      // вспомогательные функции манипулирования параметрами
      Any &add_in_arg();
      Any &add_in_arg(const char* name);
      Any &add_inout_arg();
      Any &add_inout_arg(const char* name);
      Any &add_out_arg();
      Any &add_out_arg(const char* name);
      void set_return_type(TypeCode_ptr tc);
      Any &return_value();
      void invoke();
      void send_oneway();
      void send_deferred();
      void get_response();
      Boolean poll_response();
};
```

1.30.3. Отличия от C-PIDL

Спецификация C++-PIDL отличается от спецификации C-PIDL в следующем:

- Замена **add_argument** и тому подобного на основанные на атрибутах аксессоры.
- Использование атрибута **env** для получения исключений, возбужденных в DII вызовах.
- Операция **invoke** не имеет параметра флагов, так как нет значений флагов, перечисленных как допустимые в *CORBA*.
- Операции **send_oneway** и **send_deffered** заменяют одну операцию **send** со значениями флагов для уточнения использования.
- Операция **get_response** не имеет параметра флагов и операция **poll_response** определена для немедленного возврата с индикацией, была ли завершена операция.
- Членские функции **add_*_arg, set_return_type** и **return_value** добавлены как быстрые вызовы основанных на атрибутах аксессоров.

1.30.4. Управление памятью

Request имеет следующие специальные правила управления памятью:

- Владение возвращаемыми значениями функций **target, operation, arguments, result, env, exceptions, contexts** и **ctx** осуществляется **Request**'ом, эти возвращаемые значения не должны освобождаться вызывающим.

ExceptionList имеет следующие специальные правила управления памятью:

- Функция **add_consume** потребляет своей аргумент **TypeCode_ptr**. Вызывающий не может обращаться к объекту, ссылаемому этим **TypeCode_ptr**, после того, как он был передан в функцию, потому что функция **add_consume** может скопировать его и немедленно освободить исходный.

- Владение возвращаемым значением функции **item** осуществляется **ExceptionList**, возвращаемое значение не должно освобождаться вызывающим.

ContextList имеет следующие специальные правила управления памятью:

- Функция **add_consume** потребляет свой параметр **char***. Вызывающий не может обращаться к памяти, ссылаемой этим **char*** после того, как он был передан в функцию, потому что функция **add_consume** может скопировать ее и немедленно освободить оригинал.

- Владение возвращаемым значением функции **item** осуществляется **ContextList**, возвращаемое значение не должно удаляться вызывающим.

1.31. Контекст (Context)

Context поставляет необязательную информацию контекста, связанную с вызовом метода.

1.31.1. Интерфейс Context

```
// IDL

pseudo interface Context
{
   readonly attribute Identifier context_name;
   readonly attribute context parent;


   void create_child(in Identifier child_ctx_name, out Context child_ctx);


   void set_one_value(in Identifier propname, in any propvalue);
   void set_values(in NVList values);
   void delete_values(in Identifier propname);
   void get_values(

      in Identifier start_scope,
      in Flags op_flags,
      in Identifier pattern,
      out NVList values

   );
};
```

1.31.2. C++ класс Context

```
// C++

class Context
{
   public:
      const char *context_name() const;
      Context_ptr parent() const;


      void create_child(const char *, Context_out);


      void set_one_value(const char *, const Any &);
      void set_values(NVList_ptr);
      void delete_values(const char *);
      void get_values(

            const char*,
            Flags,
            const char*,
```

```
            NVList_out

      );
};
```

1.31.3. Отличия от C-PIDL

Спецификация C++-PIDL отличается от спецификации C-PIDL в следующем:

- Введение атрибутов для имени контекста и родителя.
- Сигнатуры для значений единообразно установлены в **any**.
- В отображении на C **set_one_value** использует строки, в то время как остальные используют именованные значения **(NamedValue)**, содержащие **any**. Даже если реализициям необходима поддержка строк как значений, сигнатуры теперь единообразно позволяют альтернативы.
- Операция **release** освобождает дочерние контексты.

1.31.4. Управление памятью

Context имеет следующие специальные правила управления памятью:

- Владение возвращаемыми значениями функций **context_name** и **parent** осуществляется контекстом, эти возвращаемые значения не должны освобождаться вызывающим.

1.32. Код типа (TypeCode)

TypeCode представляет информацию об OMG IDL типе.

Для TypeCode не определено конструкторов. Однако, в дополнение к отображающему интерфейсу, для всех базовых и определенных OMG IDL типов реализация предоставляет доступ к ссылке на псевдообъект **TypeCode** (**TypeCode_ptr**) в форме **_tc_<type>**, которые могут быть использованы для установления типов в **Any**, в качестве параметров для **equal** и так далее. В именах этих **TypeCode** ссылочных констант, **<type>** указывает на локальное имя типа внутри его области определения. Каждая C++ константа **_tc_<type>** должна быть определена на том же уровне области видимости, что и соответствующий тип.

Во всех C++ **TypeCode** константных ссылках на псевдообъекты префикс У_tc_Ф должен быть использован вместо УТС_Ф префикса. Ёто необходимо для того, чтобы избежать конфликт имен для CORBA приложений, которые одновременно используют и C, и C++ отображения.

Подобно всем другим безсерверным объектам, отображение на C++ для **TypeCode** предоставляет операцию **_nil()**, которая возвращает нулевую объектную ссылку для **TypeCode**. Ёта операция может быть использована для инициализации ссылок **TypeCode**, внедренных внутрь составных типов. Однако, нулевая **TypeCode** ссылка никогда не сможет быть передана как аргумент операции, так как **TypeCode** эффективно передается по значению, а не как объектная ссылка.

1.32.1. Интерфейс TypeCode

TypeCode IDL интерфейс полностью определен в главе "Репозиторий Интерфейсов", разделе "Интерфейс TypeCode" и поэтому не повторяется здесь.

1.32.2. C++ класс TypeCode

```
// C++

class TypeCode
{
   public:
      class Bounds : public UserException { ... };
      class BadKind : public UserException { ... };


      Boolean equal(TypeCode_ptr) const;
      Boolean equivalent(TypeCode_ptr) const;
      TCKind kind() const;
      TypeCode_ptr get_compact_typecode() const;


      const char* id() const;
      const char* name() const;


      ULong member_count() const;
      const char* member_name(ULong index) const;


      TypeCode_ptr member_type(ULong index) const;


      Any *member_label(ULong index) const;
      TypeCode_ptr discriminator_type() const;
      Long default_index() const;


      ULong length() const;


      TypeCode_ptr content_type() const;


      UShort fixed_digits() const;
      Short fixed_scale() const;


      Visibility member_visibility(ULong index) const;
      ValuetypeModifier type_modifier() const;
      TypeCode_ptr concrete_base_type() const;
};
```

1.32.3. Отличия от C-PIDL

Для C++ используется префикс У_tc_Ф вместо УТС_Ф для констант.

1.32.4. Управление памятью

TypeCode имеет следующие специальные правила управления памятью:

- Владение возвращаемыми значениями функций **id, name** и **member_name** осуществляется с помощью **TypeCode**, эти возвращаемые значения не должны освобождаться вызывающим.

1.33. ORB

ORB - это программный интерфейс к Брокеру Объектных Запросов (Object Request Broker).

1.33.1. Интерфейс ORB

```
// IDL

pseudo interface ORB
{
      typedef sequence<Request> RequestSeq;
      string object_to_string(in Object obj);
      Object string_to_object(in string str);
      void create_list(in long count, out NVList new_list);
      void create_operation_list(in OperationDef oper, out NVList new_list);


      void create_named_value(out NamedValue nmval);
      void create_exception_list(out ExceptionList exclist);
      void create_context_list(out ContextList ctxtlist);


      void get_default_context(out Context ctx);
      void create_environment(out Environment new_env);


      void send_multiple_requests_oneway(in RequestSeq req);
      void send_multiple_requests_deferred(in RequestSeq req);
      boolean poll_next_response();
      void get_next_response(out Request req);


      Boolean work_pending();
      void perform_work();
      void shutdown(in Boolean wait_for_completion);
      void run();


      Boolean get_service_information (

            in ServiceType service_type,
            out ServiceInformation service_information

      );


      typedef string ObjectId;
      typedef sequence<ObjectId> ObjectIdList;
      Object resolve_initial_references(

         in ObjectId id

      ) raises(InvalidName);
      ObjectIdList list_initial_services();


      Policy create_policy(in PolicyType type, in any val)

         raises(PolicyError);
};
```

1.33.2. C++ класс ORB

```
// C++

class ORB
{
   public:
      class RequestSeq {...};
      char *object_to_string(Object_ptr);
      Object_ptr string_to_object(const char *);
      void create_list(Long, NVList_out);
```

```
        void create_operation_list(

                OperationDef_ptr,
                NVList_out

        );


        void create_named_value(NamedValue_out);
        void create_exception_list(ExceptionList_out);
        void create_context_list(ContextList_out);


        void get_default_context(Context_out);
        void create_environment(Environment_out);


        void send_multiple_requests_oneway(

                const RequestSeq&

        );


        void send_multiple_requests_deferred(

                const RequestSeq &

        );
        Boolean poll_next_response();
        void get_next_response(Request_out);


        Boolean work_pending();
        void perform_work();
        void shutdown(Boolean wait_for_completion);
        void run();


        Boolean get_service_information(

                ServiceType svc_type,
                ServiceInformation_out svc_info

        );


        typedef char* ObjectId;
        class ObjectIdList { ... };
        Object_ptr resolve_initial_references(const char* id);
        ObjectIdList* list_initial_services();
        Policy_ptr create_policy(

                    PolicyType type,
                    const Any& val

        );
};
```

1.33.3. Отличия от C-PIDL

- Добавлено **create_environment**. В отличие от версии структуры, **Environment** требует операцию построения. (Так как это чрезмерно ограничивающе для реализаций, которые не поддерживают настоящие C++ исключения, эти реализации могут позволить **Environment** быть объявленной в стеке.)

- Назначена множественная поддержка запросов в ORB, сделано использование, симметричное с тем, что в **Request**, и использован тип последовательности вместо неограниченных массивов в сигнатурах.

- Добавлена **create_named_value**, которая требуется для создания объектов **NamedValue**, используемых как параметры возвращаемого значения для операции **Object::create_request**.

- Добавлены **create_exception_list** и **create_context_list** (смотри раздел 1.30 "Запрос (Request)" для дальнейших деталей).

1.33.4. Отображение для Операций Инициализации ORB

Нижеследующий PIDL специфицирует операции инициализации для ORB, этот PIDL является частью модуля CORBA (но не интерфейса ORB) и описан в версии 2.3 CORBA, главе "Интерфейс ORB", разделе "Инициализация ORB".

```
// PIDL

module CORBA {
   typedef string ORBid;
   typedef sequence <string> arg_list;
   ORB ORB_init (inout arg_list argv, in ORBid orb_identifier);
};
```

Отображение предшествующих PIDL операций на C++ следующее:

```
// C++

namespace CORBA {
      typedef char* ORBid;
      static ORB_ptr ORB_init(

            int& argc,
            char** argv,
            const char* orb_identifier = ""

      );
}
```

Отображение на C++ для **ORB_init** отлоняется от OMG IDL PIDL в своем управлении параметром **arg_list**. Это предназначено для продоставления значимого PIDL определения интерфейса инициализации, который имеет естественную C++ привязку. С этой целью структура **arg_list** заменена на **argv** и **argc** параметры.

Параметр **argv** определен как неограниченный массив строк (**char ****) и число строк в массиве передается в параметре **argc** (**int &**).

Если используется пустая ORBid строка, тогда аргументы argc может быть использован для уточнения, какой ORB должен быть возвращен. Это достигается с помощью поиска среди параметров **argv** одного, помеченного *ORBid*, например ***-ORBid "ORBid_example"***. Если используется пустая ORBid строка и ORB не указан с помощью параметров **argv**, будет возвращен ORB по умолчанию.

Назависимо от того, передана ли пустая или непустая ORBid строка в **ORB_init**, параметры **argv** проверяются для определения, если были заданы любые параметры ORB. Если передана непустая ORBid строка в **ORB_init**, все *-ORBid* параметры в **argv** игнорируются. Все остальные *-ORB<suffix>* параметры могут быть значимы во время процесса инициализации ORB.

Для C++ порядок потребления параметров **argv** может быть значимым для приложения. Для того чтобы гарантировать, что приложениям не требуется управлять параметрами **argv**, которые они не распознают, функция инициализации ORB должна быть вызвана перед тем, как остаток параметров будет потреблен. Поэтому, после вызова **ORB_init** параметры **argv** и **argc** могут быть модифицированы для удаления аргументов, понятных ORB. Важно отметить, что вызов **ORB_init** может только переупорядочить или удалить ссылки на параметры из списка argv, это ограничение сделано для того, чтобы избежать потенциальные проблемы управления памятью, вызываемые попытками освободить части списка argv или расширением списка параметров. Вот почему argv передается как **char****, а не как **char*&**.

1.34. Объект (Object)

Правила в этом разделе применяются к OMG IDL интерфейсу **Object**, базовому в иерархии интерфейсов OMG IDL. Интерфейс **Object** определен как обычный CORBA объект, а не псевдообъект. Однако, он включен, потому что он ссылается на другие псевдообъекты.

1.34.1. Интерфейс Object

```
// IDL

interface Object
{
      boolean is_nil();
      Object duplicate();
      void release();
      ImplementationDef get_implementation();
      InterfaceDef get_interface();
      boolean is_a(in string logical_type_id);
      boolean non_existent();
      boolean is_equivalent(in Object other_object);
      unsigned long hash(in unsigned long maximum);
      void create_request(

         in Context ctx,
         in Identifier operation,
         in NVList arg_list,


         in NamedValue result,
         out Request request,
         in Flags req_flags

      );


      void create_request2(

         in Context ctx,
         in Identifier operation,
         in NVList arg_list,
         in NamedValue result,
         in ExceptionList exclist,
         in ContextList ctxtlist,
         out Request request,
         in Flags req_flags

      );
      Policy_ptr get_policy(in PolicyType policy_type);
      DomainManagerList get_domain_managers();
      Object set_policy_overrides(in PolicyList policies,
                  in SetOverrideType set_or_add);
};
```

1.34.2. C++ класс Object

В дополнение к другим правилам все имена операций в интерфейсе **Object** имеют ведущие подчеркивания в отображающем C++ классе. Также отображение для **create_request** разбито на три формы, соответветственно стилям использования, описанным в разделе 1.30 "Запрос (Request)". Функции **is_nil** и **release** предоставлены в пространстве имен **CORBA**, как описано в разделе 1.3.3.

```
// C++

class Object
{
   public:
      static Object_ptr _duplicate(Object_ptr obj);
      static Object_ptr _nil();
      ImplementationDef_ptr _get_implementation();
      InterfaceDef_ptr _get_interface();
      Boolean _is_a(const char* logical_type_id);
      Boolean _non_existent();
```

```
      Boolean _is_equivalent(Object_ptr other_object);
      ULong _hash(ULong maximum);
      void _create_request(

         Context_ptr ctx,
         const char *operation,


         NVList_ptr arg_list,
         NamedValue_ptr result,
         Request_out request,
         Flags req_flags

      );
      void _create_request(

         Context_ptr ctx,
         const char *operation,
         NVList_ptr arg_list,
         NamedValue_ptr result,
         ExceptionList_ptr,
         ContextList_ptr,
         Request_out request,
         Flags req_flags

      );
      Request_ptr _request(const char* operation);
      Policy_ptr _get_policy(PolicyType policy_type);
      DomainManagerList* _get_domain_managers();
      Object_ptr _set_policy_overrides(

         const PolicyList&,
         SetOverrideType

      );
};
```

1.35. Отображение со стороны сервера

Отображение со стороны сервера относится к ограничениям переносимости для реализации объекта, написанной на C++. Термин *сервер (server)* не означает ограничения реализаций до ситуаций, в которых вызовы методов выходят за пределы адресного пространства или за границы машины. Данное отображение охватывает любую реализацию интерфейса OMG IDL.

1.36. Реализовывание интерфейсов

Для определения реализации на C++ определяют C++ класс с любым правильным C++ именем. Для каждой операции в интерфейсе класс определяет нестатичную членскую функцию с отображенным именем операции (отображенное имя такое же, как и OMG IDL идентификатор, за исключением тех случаев, когда идентификатор является ключевым словом C++, в этом случае к идентификатору приписывается строка "_cxx_", как описано в разделе 1.1). Заметим, что реализация ORB может позволить одному классу реализации быть производным от другого, поэтому выражение "класс определяет членскую функцию" не означает, что класс должен явно определить членскую функцию - он может наследовать ее.

1.36.1. Отображение для PortableServer::Servant

Модуль **PortableServer** для Адаптера Переносимого Объекта (Portable Object Adapter - POA) определяет собственный тип **Servant**. Отображение на C++ для **Servant** является следующим:

```
// C++

namespace PortableServer
{
```

```
      class ServantBase

      {
      public:
         virtual ~ServantBase();


         virtual POA_ptr _default_POA();


         virtual InterfaceDef_ptr

         _get_interface() throw(SystemException);


         virtual Boolean

         _is_a(const char* logical_type_id)

               throw(SystemException);


         virtual Boolean

         _non_existent() throw(SystemException);


         virtual void _add_ref();
         virtual void _remove_ref();


      protected:
         ServantBase();
         ServantBase(const ServantBase&);
         ServantBase& operator=(const ServantBase&);
         // ...all other constructors...

      };
      typedef ServantBase* Servant;
}
```

Деструктор **ServantBase** является открытым и виртуальным для гарантирования того, что скелетные классы, произведенные от него, могли быть правильно уничтожены. Конструктор по умолчанию, вместе с остальными реализационно-зависимыми конструкторами, должен быть защищенным, чтобы экземпляры **ServantBase** не могли быть созданы иначе, чем под-объекты экземпляров производных классов. Конструктор по умолчанию (конструктор, который либо не имеет аргументов, либо имеет только аргументы со значениями по умолчанию) должен быть предоставлен, чтобы производные служащие (servant) могли быть построены машинонезависимо. И конструктор копирования, и защищенный оператор присваивания по умолчанию должны поддерживаться, чтобы программно-зависимые служащие могли быть скопированы, если необходимо. Заметим, что копирование служащего (servant), который уже зарегистрирован с объектным адаптером, с помощью присваивания или конструирования, не значит, что цель присваивания или копирования также зарегистрирована с объектным адаптером. Подобным же образом, присваивание **ServantBase** или классу, производному от него, который уже зарегистрирован с объектным адаптером, в любом случае не меняет его регистрацию.

Реализация по умолчанию функции **_default_POA**, предоставляемая **ServantBase**, возвращает объектную ссылку на корневой POA для ORB по умолчанию в этом процессе - такую же, как возвращаемое значение вызова **ORB::resolve_initial_references("RootPOA")** над ORB по умолчанию. Классы, производные от **ServantBase**, могут переопределять это определение для возвращения POA по своему выбору, если это желательно.

ServantBase предоставляет реализации по умолчанию для **_get_interface, _is_a** и **_non_existent** операций объектных ссылок, которые могут быть переопределены производными служащими (servant), если поведение по умолчанию не является адекватным. POA вызывает их подобно обычным скелетным операциям, таким образом позволяя переопределенным определениям в производных классах служащих использовать **_this** и интерфейс **PortableServer::Current** внутри тел их функций.

Для статичных скелетов реализация по умолчанию функций **_get_interface** и **_is_a**, предоставляемая **ServantBase**, использует интерфейс, связанный со скелетным классом, для определения их соответствующих возвращаемых значений. Для динамических скелетов эти функции используют функцию **_primary_interface** для определения их возвращаемых значений.

Реализация по умолчанию для **_non_existent** просто возвращает ложь (false).

Экземпляры служащих (servant) могут реализовать подсчет ссылок для предотвращения самоуничтожения, пока приложение еще использует их. Подсчет ссылок служащих выполняется, используя функции **_add_ref** и **_remove_ref**, объявленные в **ServantBase**. Реализации по умолчанию для **_add_ref** и **_remove_ref**, поставляемые **ServantBase**, ничего не делают:

```
// C++

void PortableServer::ServantBase::_add_ref()
{
      // пусто
}


void PortableServer::ServantBase::_remove_ref()
{
      // пусто
}
```

Для служащих, которым необходимо выполнять фактический посчет ссылок, эти функции являются виртуальными и поэтому могут быть переопределены в производных классах служащих. Mix-In класс подсчета ссылок по умолчанию также предоставляется в пространстве имен **PortableServer**, детали смотри в разделе 1.36.2.

Детали, касающиеся POA и ответственности приложений относительно подсчета ссылок, могут быть найдены в разделе 1.36.4.

1.36.2. Mix-In подсчет ссылок служащих (Servant)

Пространство имен PortableServer также предоставляет стандартный mix-in класс подсчета ссылок служащих (servant):

```
// C++

namespace PortableServer
{
      class RefCountServantBase : public virtual ServantBase

      {
      public:
         ~RefCountServantBase();


         virtual void _add_ref();
         virtual void _remove_ref();


      protected:
         RefCountServantBase() : _ref_count(1) {}
         RefCountServantBase(

            const RefCountServantBase&

         ) : _ref_count(1) {}


         RefCountServantBase&
```

```
      operator=(const RefCountServantBase&);


   private:
      ULong _ref_count;
      // ...другие детали реализации...

   };
}
```

RefCountServantBase mix-in класс переопределяет наследуемые функции **_add_ref** и **_remove_ref**, которые он наследует от **ServantBase**, реализуя их для предоставления настоящего подсчета ссылок. Экземпляр класса служащего, производного от **RefCountServantBase**, изначально имеет счетчик ссылок, равный единице. Вызов **_add_ref** над экземпляром служащего увеличивает его счетчик ссылок на единицу. Вызов **_remove_ref** над экземпляром служащего уменьшает его счетчик ссылок на единицу, если в результате счетчик ссылок стане травным нулю, **_remove_ref** вызывает **delete** над своим **this** указателем для того, чтобы уничтожить служащего (servant). Для ORB'ов, которые оперируют в многопотоковой среде, реализации **_add_ref** и **_remove_ref**, которые предоставляет класс **RefCountServantBase**, должны быть потоко-безопасными.

Подобно **ServantBase, RefCountServantBase** поддерживает конструктор копирования и операцию присваивания по умолчанию. Реализация присваивания по умолчанию просто возвращает ***this** и не влияет на счетчик ссылок.

1.36.3. Класс ServantBase_var

Для удобства автоматического управления подсчетом ссылок служащих (servant), пространство имен **PortableServer** также предоставляет класс ServantBase_var. Этот класс ведет себя подобно _var классам для объектных ссылок.

```
// C++

namespace PortableServer
{
   class ServantBase_var

   {
   public:
      ServantBase_var() : _ptr(0) {}
      ServantBase_var(ServantBase* p) : _ptr(p) {}
      ServantBase_var(const ServantBase_var& b)

         : _ptr(b._ptr)

      {
         if (_ptr != 0) _ptr->_add_ref();
      }
      ~ServantBase_var()

      {
         if (_ptr != 0) _ptr->_remove_ref();
      }


      ServantBase_var& operator=(ServantBase* p)

      {
         if (_ptr != 0) _ptr->_remove_ref();
         _ptr = p;
         return *this;
      }
      ServantBase_var&



      operator=(const ServantBaseBase_var& b)

      {
         if (_ptr != b._ptr) {
            if (_ptr != 0) _ptr->_remove_ref();
```

```
            if ((_ptr = b._ptr) != 0)

            _ptr->_add_ref();
         }
         return *this;
      }


      ServantBase* operator->() const { return _ptr; }


      ServantBase* in() const { return _ptr; }
      ServantBase*& inout() { return _ptr; }
      ServantBase*& out()

      {
         if (_ptr != 0) _ptr->_remove_ref();
         _ptr = 0;
         return _ptr;
      }
      ServantBase* _retn()

      {
         ServantBase* retval = _ptr;
         _ptr = 0;
         return retval;
      }


   private:
      ServantBase* _ptr;
   };
}
```

Реализация, показанная выше для **ServantBase_var**, предназначена только как пример, который передает требуемую семантику. Вариации этой реализации возможны, пока они предоставляют ту же семантику, как и реализация, показанная здесь.

1.36.4. Рассуждения об управлении памятью служащего (servant)

Переносимое управление памятью для служащих (servant) требует точной спецификации, когда и как служащий может быть удален. Эта спецификация поддерживает, но не требует подсчета ссылок:

- Когда используется подсчет ссылок, реализации **_remove_ref** поручено удалять служащего, когда его счетчик ссылок упадет до нуля. POA гарантирует, что служащий не будет удален до тех пор, пока вызовы, выполняемые в данный момент над этим служащим с помощью использования ссылки на служащего, не будут завершены. К примеру, POA может увеличивать счетчик ссылок служащего перед вызовом реализации (но после **preinvoke**) и уменьшать счетчик ссылок после вызова (но перед **postinvoke**).

- Если подсчет ссылок не используется, некоторый код приложения должен явно вызывать **delete** над выделенным в куче служащим для того, чтобы память служащего была восстановлена.
  Помните, что явное удаление служащего вызовет нарушения доступа к памяти, если этот служащий все еще используется некоторой другой частью приложения. К примеру, если тот же экземпляр служащего был получен из **POA::reference_to_servant** или **POA::id_to_servant** (возможно, в другом потоке), вызывающий, который получил экземпляр служащего, может еще использовать его. Также, явное удаление может вызвать проблемы, если тот же экземпляр служащего был зарегистрирован в нескольких POA.

Для каждого POA, операций **ServantActivator** или **ServantLocator**, которые или передают **Servant** как параметр, или возвращают **Servant**, следующие правила описывают ответственность вызывающего и вызываемого за управление памятью:

- **ServantActivator::incarnate** — возвращает **Servant**. POA может использовать этот **Servant**, пока он не будет передан в **etherealize**.

- **ServantActivator::etherealize** — имеет входной аргумент **Servant**. POA принимает, что **etherealize** потребляет аргумент **Servant**, и не обращается к **Servant** в любом случае после того, как он был передан в **etherealize**. Соответствующая реализация для **etherealize** может удалить свой аргумент **Servant**, если тип **Servant** использует подсчет ссылок по умолчанию, унаследованный от **ServantBase**, и экземпляр **Servant** был выделен в куче. Если приложение использует подсчет ссылок, **Servant** потребляется вызовом **_remove_ref** над этим служащим (**Servant**).

- **ServantLocator::preinvoke** — возвращает **Servant**. POA может использовать этот **Servant**, пока он не будет передан в **postinvoke**.

- **ServantLocator::postinvoke** — имеет входным (**in**) аргментом **Servant**. POA принимает, что **postinvoke** потребляет аргумент **Servant**, и не обращается к нему влюбом случае после того, как он был передан в **postinvoke**. Соответствующая реализация **postinvoke** может удалить свой **Servant** аргумент, если тип **Servant** использует подсчет ссылок по умолчанию, унаследованный от **ServantBase** и экземпляр **Servant** был выделен в куче. Если приложение использует подсчет ссылок, **Servant** потребляется с помощью вызова **_remove_ref** над этим **Servant**.

- **POA::get_servant** — возвращает **Servant**. POA однократно вызывает **_add_ref** над **Servant** перед его возвращением. Если приложение использует подсчет ссылок, вызывающий для **get_servant** является ответственным за однократный вызов **_remove_ref** над возвращаемым **Servant**, когда он закончит работу с ним. Соответствующему вызывающему не нужно вызывать **_remove_ref** над возвращаемым **Servant**, если его тип использует подсчет ссылок по умолчанию, унаследованный от **ServantBase**.

- **POA::set_servant** — имеет входной (**in**) аргумент **Servant**. Реализация **set_servant** будет вызывать **_add_ref** по крайней мере однократно над аргументом **Servant** перед возвращением. Когда POA нет больше нужды в **Servant**, он вызовет **_remove_ref** над ним такое же число раз.

- **POA::activate_object** — имеет входной (**in**) аргумент **Servant**. Реализация для **activate_object** вызовет **_add_ref** по крайней мере однократно над аргументом **Servant** перед возвращением. Когда POA не будет больше необходим **Servant**, он вызовет **_remove_ref** над ним такое же число раз.

- **POA::activate_object_with_id** — имеет входной (**in**) аргумент **Servant**. Реализация для **activate_object_with_id** вызовет **_add_ref** по крайней мере однократно над аргументом **Servant** перед возвращением. Когда POA не будет больше необходим **Servant**, он вызовет **_remove_ref** над ним такое же число раз.

- **POA::servant_to_id** — имеет входной (**in**) аргумент **Servant**. Если эта операция вызывает активацию объекта, **_add_ref** вызывается по крайней мере однократно над аргументом **Servant** перед возвращением. Иначе POA не увеличивает или уменьшает счетчик ссылок **Servant**, переданного в эту функцию.

- **POA::servant_to_reference** — имеет входной (**in**) аргумент **Servant**. Если эта операция вызывает активацию объекта, **_add_ref** вызывается по крайней мере однократно над аргументом **Servant** перед возвращением. Иначе POA не увеличивает или уменьшает счетчик ссылок **Servant**, переданного в эту функцию.

- **POA::reference_to_servant** — возвращает **Servant**. POA однократно вызывает **_add_ref** над **Servant** перед его возвращением. Если приложение использует подсчет ссылок, вызывающий **reference_to_servant** является ответственным за однократный вызов **_remove_ref** над возвращаемым **Servant**, когда он закончит с ним работу. Соответствующему вызывающему нет необходимости вызывать **_remove_ref** над возвращаемым **Servant**, если тип **Servant** использует подсчет ссылок по умолчанию, унаследованный от **ServantBase**.

- **POA::id_to_servant** — возвращает **Servant**. POA однократно вызывает **_add_ref** над **Servant** перед его возвращением. Если приложение использует подсчет ссылок, вызывающий **reference_to_servant** является ответственным за однократный вызов **_remove_ref** над возвращаемым **Servant**, когда он закончит с ним работу. Соответствующему вызывающему нет необходимости вызывать **_remove_ref** над возвращаемым **Servant**, если тип **Servant** использует подсчет ссылок по умолчанию, унаследованный от **ServantBase**.

Следующие операции не принимают или возвращают служащих (**Servant**) в своих сигнатурах, но имеют поведение, которое может требовать вызовы **_add_ref** или **_remove_ref**:

- **_this** — вызывается над **Servant**'ом для получения объектной ссылки для объекта, реализованного этим **Servant**. Если эта операция вызвала активацию объекта, **_add_ref** вызывается по крайней мере однократно над аргументом **Servant** перед возвращением. Иначе POA не увеличивает или уменьшает счетчик ссылок **Servant**, переданного в эту функцию.

- **POA::deactivate_object** — при активации вызывается **_add_ref** над **Servant**. Следовательно, акт деактивации должен привести к тому, что будет вызвана **_remove_ref**. Если POA не имеет связанного с ним **ServantActivator**, реализация POA вызывает **_remove_ref**, когда все вызовы операций завершатся. Если имеется **ServantActivator**, служащий (Servant) вместо этого потребляется вызовом **ServantActivator::etherealize**.

- **POA::destroy** — при активации служащего или регистрации служащего по умолчанию, вызывается **_add_ref** над **Servant**. Следовательно, уничтожение POA должно привести к вызову **_remove_ref**. Реализация POA вызывает **_remove_ref** над любым служащим по умолчанию. Если POA не имеет связанного с ним **ServantActivator**, реализация POA вызывает **_remove_ref** над каждым **Servant** в Карте Активных Объектов (Active Object Map), когда все вызовы операций будут завершены. Если имеется **ServantActivator**, каждый **Servant** вместо этого потребляется вызовом **ServantActivator::etherealize**.

- **POAManager::deactivate** — при активации служащего или регистрации служащего по умолчанию, вызывается **_add_ref** над **Servant**. Следовательно, уничтожение POA должно привести к вызову **_remove_ref**. Если **etherealize_objects** равно истине (true), реализация POA вызывает **_remove_ref** над любым служащим по умолчанию. Если **etherealize_objects** равно истине (true) и управляемый POA не имеет связанного с ним **ServantActivator**, реализация POA вызывает **_remove_ref** над каждым Servant в Карте Активных Объектов после того, как все обрабатываемые операции будут завершены. Если имеется **ServantActivator**, каждый **Servant** вместо этого потребляется вызовом **ServantActivator::etherealize**.

Заметим, что в тех случаях, где вызывающий становится ответственным за вызов **_remove_ref** над возвращаемым ему **Servant**, вызывающий может присвоить возвращаемое значение в экземпляр **ServantBase_var** для автоматического управления подсчетом ссылок.

1.36.5. Скелетные Операции

Все скелетные классы предоставляют членскую функцию **_this()**. Эта членская функция имеет три назначения:

1. Внутри контекста вызова запроса над целевым объектом, представленным служащим, он позволяет служащему получить объектную ссылку для целевого CORBA объекта, воплощения для этого запроса. Это является истинным, даже если служащий воплощает несколько CORBA объектов. В этом контексте **_this()** может быть вызван независимо от политик, используемых для создания координирующим POA.

2. За пределами контекста вызова запроса над целевым объектом, представляемым служащим, она позволяет служащему быть неявно активизированным, если его POA разрешает неявную активизаци. Это требует, чтобы активизирующий POA был создан с политикой **IMPLICIT_ACTIVATION**. Если POA не был создан с политикой **IMPLICIT_ACTIVATION**, возбуждается исключение **PortableServer::WrongPolicy**. POA, используемый для неявной активизации, получается вызовом **_default_POA()** над служащим.

3. За пределами контекста вызова запроса над целевым объектом, представляемым служащим, функция будет возвращать объектную ссылку для служащего, который уже активизирован, пока служащий не воплощает несколько CORBA объектов. Это требует, чтобы POA, с которой служащий был активизирован, был создан с политиками **UNIQUE_ID** и **RETAIN**. Если POA был создан с политиками **MULTIPLE_ID** или **NON_RETAIN**,

будет возбуждено исключение **PortableServer::WrongPolicy**. POA получается вызовом **_default_POA()** над служащим.

К примеру, для интерфейса **A**, определенного следующим образом:

```
// IDL

interface A
{
   short op1();
   void op2(in long val);
};
```

Возвращаемое значение **_this()** является типизированной объектной ссылкой для типа интерфейса, соответствующего скелетному классу. К примеру, функция **_this()** для скелета интерфейса **A** может быть определена так:

```
// C++

class POA_A : public virtual ServantBase
{
   public:
      A_ptr _this();
      ...

};
```

Функция **_this()** следует обычным правилам отображения на C++ для возвращаемых объектных ссылок, поэтому вызывающий получает владение возвращаемой объектной ссылкой и должен обычно вызвать **CORBA::release()** над ней.

Функция **_this()** может быть виртуальной, если среда C++ поддерживает ковариантные возвращаемые типы, а иначе функция должна быть невиртуальной, чтобы возвращаемый тип мог быть корректно определен без ошибок компилятора. Приложения используют **_this()** таким же образом, независимо от того, какие из этих подходов реализации были взяты.

Полагая, что **A_impl** - это производный класс от **POA_A**, который реализует интерфейс **A** и что POA служащего был создан с соответствующими политиками, служащий типа **A_impl** может быть создан и неявно активизирован следующим образом:

```
// C++

A_impl my_a;
A_var a = my_a._this();
```

1.36.6. Реализация интерфейсов, основанная на наследовании

Классы реализации могут быть производными от сгенерированного базового класса, основанного на определении интерфейса OMG IDL. Сгенерированные базовые классы известны как *скелетные классы* (***skeleton class***), а производные классы известны как *классы реализации (**implementation class**)*. Каждая операция интерфейса имеет соответствующую виртуальную членскую функцию, объявленную в скелетном классе. Сигнатура членской функции является идентичной той, что в сгенерированном клиентском суррогатном (**stub**) классе. Класс реализации предоставляет реализации для этих членских функций. Адаптер объекта обычно вызывает методы через обращения к виртуальным функциям скелетного класса.

Пусть IDL интерфейс A определен следующим образом:

```
// IDL

interface A
{
   short op1();
   void op2(in long val);
};
```

Для IDL интерфейса A, показанного выше, компилятор генерирует класс интерфейса **A**. Этот класс включает C++ определения для описаний типов (typedef), констант, исключений, атрибутов и операций в OMG IDL интерфейсе. Он имеет форму, подобную следующей:

```
// C++

class A : public virtual Object
{
   public:
      virtual Short op1() = 0;
      virtual void op2(Long val) = 0;
      ...

};
```

Некоторые реализации ORB не могут использовать открытое виртуальное наследование от **CORBA::Object**, и не могут сделать операции чисто виртуальными, но сигнатуры операций будут такими же.

Со стороны сервера генерируется скелетный класс. Этот класс является частично непрозрачным для программиста, хотя он будет содержать членскую функцию соответственно каждой операции в интерфейсе. Для POA имя скелетного класса формируется приписыванием строки "POA_" в начало имени соответствующего интерфейса, и класс прямо или косвенно производится от базового класса служащего **PortableServer::ServantBase**. Класс **PortableServer::ServantBase** должен быть виртуальным базовым классом для скелетного, чтобы позволить переносимые реализации для множественного наследования и от скелетных классов, и от классов реализации для других базовых интерфейсов без ошибки или неоднозначности.

Скелетный класс для интерфейса **A**, показанного выше, может получиться следующим:

```
// C++

class POA_A : public virtual PortableServer::ServantBase
{
   public:
      // ...реализационно-зависимые детали серверной части
      // будут идти здесь...

      virtual Short op1() throw(SystemException) = 0;
      virtual void op2(Long val) throw(SystemException) = 0;
      ...

};
```

Если интерфейс **A** был определен внутри модуля, а не в глобальной области видимости, например, **Mod::A**, имя его скелетного класса будет POA_Mod::A. Это помогает разделить скелетные декларации и определения серверной части от C++ кода, генерируемого для клиента.

Для реализации этого интерфейса, используя наследование, программист должен унаследовать от этого скелетного класса и реализовать каждую операцию в OMG IDL интерфейсе. Декларация класса реализации для интерфейса **A** принимает форму:

```
// C++

class A_impl : public POA_A
{
   public:
      Short op1() throw(SystemException);
```

```
      void op2(Long val) throw(SystemException);
      ...

};
```

Заметим, что присутствие функции **_this()** означает, что C++ служащие (servant) должны быть производными непосредственно только от одного скелетного класса. Прямое наследование от нескольких скелетных классов может выразиться в ошибках неопределенности из-за нескольких определений **_this()**. Это не должно быть ограничением, так как CORBA объекты имеют только один наиболее производный интерфейс. Служащие, которые предназначены для поддержки нескольких типов интерфейсов, могут использовать основанный на делегировании подход реализации интерфейса, описанный ниже, или могут быть зарегистрированы как служащие, основанные на DSI.

1.36.7. Реализация интерфейса, основанная на делегировании

Наследование не всегда лучшее решение для реализации служащих (servant). Использование наследования от OMG IDL-генерированных классов насаждает C++ иерархию наследования. Иногда накладные расходы такого наследования слишком высоки или его невозможно скомпилировать корректно из-за дефектов в компиляторе C++. Например, реализация объектов, используя существующий унаследованный код, может быть невозможной, если требуется наследование от некоторого глобального класса из-за навязчивой природы наследования.

В некоторых случаях делегирование может быть использовано для решения этой проблемы. Вместо наследования от скелетного класса реализация может быть закодирована, как требуется для приложения, и упаковочный (wrapper) объект будет делегировать вызовы к этой реализации. Этот раздел описывает, как это может быть достигнуто в безопасной по типу манере, используя C++ шаблоны.

Для примеров в этом разделе снова будет использован OMG IDL интерфейс из раздела 1.36.6:

```
// IDL

interface A
{
   short op1();void op2(in long val);
};
```

В дополнение к генерированию скелетного класса IDL компилятор генерирует делегирующий класс, называемый *привязка **(tie)***. Этот класс частично непрозрачен для программиста приложения, хотя подобно скелетному классу, он предоставляет методы, соответствующие каждой OMG IDL операции. Имя сгенерированного класса привязки (tie) такое же, как и сгенерированного скелетного класса, с добавлением строки "_tie" в конец имени. Например:

```
// C++

template<class T>

class POA_A_tie : public POA_A
{
   public:
      ...

};
```

Экземпляр этого шаблонного класса выполняет задачу делегирования. Когда шаблону приписывается тип класса, который предоставляет операции **A**, тогда класс **POA_A_tie** будет делегировать все операции экземпляру этого класса реализации. Ссылка или указатель на фактический объект реализации передается в подходящий конструктор привязки, когда создается экземпляр класса привязки. Когда делается запрос над ним, служащий привязки будет только делегировать запрос с помощью вызова соответствующего метода в объекте реализации.

```
// C++

template<class T>

class POA_A_tie : public POA_A
{
   public:
      POA_A_tie(T& t)

         : _ptr(&t), _poa(POA::_nil()), _rel(0) {}
      POA_A_tie(T& t, POA_ptr poa)

         : _ptr(&t),
           _poa(POA::_duplicate(poa)), _rel(0) {}
      POA_A_tie(T* tp, Boolean release = 1)

         : _ptr(tp), _poa(POA::_nil()), _rel(release) {}
      POA_A_tie(T* tp, POA_ptr poa,
            Boolean release = 1)

         : _ptr(tp), _poa(POA::_duplicate(poa)),
           _rel(release) {}
      ~POA_A_tie()

      {
         CORBA::release(_poa);
         if (_rel) delete _ptr;
      }
      // специфичные функции для привязки (tie)

      T* _tied_object() { return _ptr; }
      void _tied_object(T& obj)

      {
         if (_rel) delete _ptr;
         _ptr = &obj;
         _rel = 0;
      }
      void _tied_object(T* obj, Boolean release = 1)

      {
         if (_rel) delete _ptr;
         _ptr = obj;
         _rel = release;
      }
      Boolean _is_owner() { return _rel; }
      void _is_owner(Boolean b) { _rel = b; }


      // IDL операции
      Short op1() throw(SystemException)

      {
         return _ptr->op1();
      }
      void op2(Long val) throw(SystemException)

      {
         _ptr->op2(val);
      }
      // перегруженные ServantBase операции
      POA_ptr _default_POA()

      {
         if (!CORBA::is_nil(_poa)) {
            return PortableServer::POA::_duplicate(_poa);
         } else {
            // return root POA

         }
      }
   private:
      T* _ptr;
      POA_ptr _poa;
      Boolean _rel;


      // копирование и присваивание не разрешены
      POA_A_tie(const POA_A_tie&);
      void operator=(const POA_A_tie&);
};
```

Важно заметить, что пример привязки (tie), показанный выше, содержит примерные реализации для всех требуемых функций. Соответствующая реализации свободна реализовывать эти операции, как ей будет удобно, пока они удовлетворяют семантике, описанной в нижеследующих параграфах. Соответствующей реализации также позволено включать дополнительные реализационно-зависимые функции, если она пожелает.

Конструкторы **T&** приводят к тому, что служащий привязки будет делегировать все обращения к C++ объекту по ссылке **t**. Владение объектом, ссылаемым с помощью **t**, не становится ответственностью служащего привязки.

Конструкторы **T*** приводят к тому, что служащий привязки будет делегировать все обращения к C++ объекту, указанному с помощью **tp**. Параметр **release** диктует, получает ли привязка (tie) владение C++ объектом, указанным с помощью **tp**; если **release** равен **TRUE**, привязка принимает C++ объект, а иначе нет. Если привязка принимает делегируемый C++ объект, она удалит его, когда будет вызван ее собственный деструктор, как показано выше в деструкторе **~POA_A_tie()**.

Аксессорная функция **_tied_object()** позволяет вызывающему получить доступ к делегируемому C++ объекту. Если привязка была построена с получением владения C++ объекта (**release** был **TRUE** в **T*** конструкторе), вызывающий **_tied_object()** никогда не должен удалять (**delete**) возвращаемое значение.

Первая модификаторная функция **_tied_object()** вызывает **delete** над текущим привязанным объектом, если флаг release привязки равен **TRUE**, и затем указывает на новый объект, переданный в нее. Флаг release привязки устанавливается в **FALSE**. Вторая модификаторная функция **_tied_object()** делает то же самое, за исключением того, что конечное состояние флага release привязки определяется значением аргумента **release**.

Аксессорная функция **_is_owner()** возвращает **TRUE**, если привязка владеет делегируемым C++ объектом, или **FALSE**, если нет. Модификаторная функция **_is_owner()** позволяет изменять состояние флага release привязки. Это полезно для гарантирования того, что не появится утечка памяти при передаче владения привязанным объектом от одной привязки к другой или при изменении привязанного объекта, которому делегирует привязка.

Для основанных на делегировании реализаций является важным заметить, что служащий - это объект привязки, а не C++ объект, делегируемый с помощью объекта привязки. Это означает, что служащий привязки используется как аргумент для тех POA операций, которые требуют **Servant** аргумент. Это также означает, что любые операции, которые POA вызывает над служащим (servant), такие как **ServantBase::_default_POA()**, предоставляются служащим привязки, как показано в примере выше. Значение, возвращаемое с помощью **_default_POA()**, поставляется в конструктор привязки (tie).

Также важно заметить, что по умолчанию основанная на делегировании реализация ("привязанный" C++ экземпляр) не имеет доступа к функции **_this()**, которая является доступной только над привязкой. Один из способов предоставления такого доступа - это информирование объекта делегирования о связанном с ним объектом привязки. По этому способу привязка (tie) хранит указатель на объект делегирования и наоборот. Однако, такой подход работает только, если привязка и объект делегирования имеют отношение один-к-одному. Для объекта делегирования, привязанного к нескольким объектам привязки, объектная ссылка, с помощью которой он был вызван, может быть получена в контексте вызова запроса с помощью вызова **PortableServer::Current::get_object_id()**, передачи его возвращаемого значения в **PortableServer::POA::id_to_reference()** и подходящего сужения возвращаемой объектной ссылки.

В классе привязки, показанном выше, все операции показаны в самом определении класса. На практике, лучше, чтобы они были определены за пределами определения класса, особенно для тех функций, которые переопределяют наследуемые виртуальные функции. Любой подход разрешен соответствующими реализациями.

Использование шаблонов для классов привязки позволяет разработчику приложения предоставить специализации для нескольких или всех членских функций шаблона для данного применения шаблона. Это позволяет приложению контролировать, как вызывается привязанный объект. К примеру, операция **POA_A_tie<T>::op2()** обычно определяется следующим образом:

```
// C++

template<class T>

void

POA_A_tie<T>::op2(Long val) throw(SystemException)
{
   _ptr->op2(val);
}
```

Эта реализация полагает, что привязанный объект поддерживает операцию **op2()** с такой же сигнатурой и возможностью возбуждать системные исключения CORBA. Однако, если приложение желает использовать имеющиеся классы для типов привязанных объектов, маловероятно, что они будут поддерживать эти возможности. В этом случае приложение может предоставить свою собственную специализацию. Например, если приложение уже имеет класс, названный **Foo**, который поддерживает функцию **log_value()**, функция класса привязки **op2()** может быть сделана для ее вызова, если предоставлена следующая специализация:

```
// C++

void

POA_A_tie<Foo>::op2(Long val) throw(SystemException)
{
      _tied_object()->log_value(val);
}
```

Переносимые специализации, подобные показанной выше, не могут получить доступ к полям данных класса привязки напрямую, так как имена этех полей данных не стандартизованы.

Для C++ реализаций, которые не поддерживают пространства имен или определение шаблонных классов внутри других классов, шаблонный класс привязки должен быть определен в глобальной области определения. Для таких сред имена шаблонных классов привязки должны формироваться с помощью "выравнивания" обычного имени привязки, например, заменой всех вхождений "**::**" на "**_**". Например, в такой среде имя шаблонного класса привязки для интерфейса **A::B::C** может быть **POA_A_B_C_tie**.

1.37. Реализация Операций

Сигнатура реализации членской функции - это отображенная сигнатура OMG IDL операции. В отличие от клиентской части, отображение серверной части требует, чтобы заголовок функции включал соответствующую спецификацию исключений (**throw**). Это требование позволяет компилятору обнаруживать, когда возбуждается неверное исключение, которое необходимо в случае обращения "локальный C++-C++ библиотека" (иначе обращение пойдет через упаковщик (wrapper), который проверит правильность исключения). Например:

```
// IDL

interface A
{
   exception B {};
   void f() raises(B);
};
```

```
// C++

class MyA : public virtual POA_A
{
   public:
      void f() throw(A::B, SystemException);
```

```
      ...
};
```

Так как все операции и атрибуты могут возбуждать системные исключения CORBA, **CORBA::SystemException** должен появляться во всех спецификациях исключений, даже когда операция не имеет ключа **raises**.

Внутри членской функции, указатель "this" ссылается на данные объекта реализации, как определено классом. В дополнение к доступу к данным, членская функция может неявно вызывать другую членскую функцию, определенную этим же классом. Например:

```
// IDL

interface A
{
      void f();
      void g();
};


// C++

class MyA : public virtual POA_A
{
   public:
      void f() throw(SystemException);
      void g() throw(SystemException);
   private:
      long x_;
};


void

MyA::f() throw(SystemException)
{
      this->x_ = 3;
      this->g();
}
```

Однако, когда членская функция служащего вызывается в такой манере, она будет вызвана просто как C++ членская функция, а не как реализация операции над CORBA объектом. При таком контексте, любая информация, доступная через **POA_Current** объект, ссылается на вызов CORBA запроса, который выполняется вызовом C++ членской функции, а не на вызов самой членской функции.

1.37.1. Получение Скелета Из Объекта

В некоторых существующих реализациях ORB каждый скелетный класс получается из соответствующего класса интерфейса. Например, для интерфейса **Mod::A**, скелетный класс **POA_Mod::A** получается из класса **Mod::A**. Поэтому такие системы позволяют объектной ссылке для служащего быть неявно полученной через обычные правила приведения производного-к-базовому языка C++:

```
// C++

MyImplOfA my_a;     // определение impl для A

A_ptr a = &my_a;    // получение его объектной ссылки
                    // с помощью C++ приведения
                    // производного-к-базовому
```

Такой код может поддерживаться соответствующей реализацией ORB, но это не требуется, и поэтому не является переносимым. Эквивалентный переносимый код вызывает **_this()** над объектом реализации для того, чтобы неявно зарегистрировать его, если он еще не был зарегистрирован, и получить его объектную ссылку:

```
// C++
```

```
MyImplOfA my_a;         // определение impl для A

A_ptr a = my_a._this(); // получение его объектной ссылки
```

1.38. Отображение DSI на C++

Общий Брокер Объектных Запросов: Архитектура и Спецификация, раздел "DSI: Отображение языка" главы "Динамический Скелетный Интерфейс" (*The Common Object Request Broker: Architecture and Specifications, Dynamic Skeleton Interface chapter, DSI: Language Mapping*) содержит общую информацию об отображении Динамического Скелетного Интерфейса (**Dynamic Skeleton Interface - DSI**) на языки программирования.

Этот раздел включает следующую информацию:

- Отображение **ServerRequest** Динамического Скелетного Интерфейса на C++
- Отображение Процедуры Динамической Реализации (*Dynamic Implementation Routine - DIR*) Адаптера Переносимого Объекта на C++

1.38.1. Отображение ServerRequest на C++

Псевдообъект **ServerRequest** отображается в C++ класс в пространстве имен **CORBA**, который поддерживает следующие операции и сигнатуры:

```
// C++

class ServerRequest
{
   public:
      const char* operation() const;
      void arguments(NVList_ptr& parameters);
      Context_ptr ctx();
      void set_result(const Any& value);
      void set_exception(const Any& value);
};
```

Заметим, что по окончании отображения на C++ реализации ORB свободны сделать такие операции виртуальными и модифицировать наследников как потребуется.

Все эти операции следуют обычным правилам управления памятью для данных, переданных в скелетные объекты с помощью ORB. То есть, DIR не положено модифицировать или заменять строку, позвращаемую с помощью **operation()**, входные (**in**) параметры в **NVList**, возвращаемом из **arguments()**, или контекст (**Context**), возвращаемый с помощью **ctx()**. Таким же образом, данные, выделенные с помощью DIR и переданные в ORB (параметры **NVList**), освобождаются с помощью ORB, а не с помощью DIR.

1.38.2. Управление Параметрами и Результатами Операций

ServerRequest предоставляет значения параметров, когда DIR вызывает операцию **arguments()**. **NVList**, предоставляемый DIR'ом в ORB, включает коды типов (**TypeCode**) и флаги направления **Flags** (внутри **NamedValue**) для всех параметров, включая исходящие (**out**) для операции. Это позволяет ORB проверять, чтобы были предоставлены корректные типы параметров перед заполнением их значениями, не требует от него делать так. Это также освобождает ORB от всей ответственности за консультирование Репозитория Интерфейсов (*Interface Repository*), способствуя высокопроизводительным реализациям.

NVList, предоставляемый в ORB, тогда переходит во владение ORB. Он становится освобожденным после возврата DIR. Это позволяет DIR передавать **out** значения, включая возвращаемую часть **inout** значений, в ORB с помощью модифицирования **NVList** после того, как была вызвана **arguments()**. Следовательно, если DIR хранит **NVList_ptr** внутри **NVList_var**, он должен передать его в функцию **arguments()** путем вызова функции **_retn()** над ним для того, чтобы заставить его отдать владение его внутренним **NVList_ptr** в ORB.

1.38.3. Отображение Процедуры Динамической Реализации (*Dynamic Implementation Routine*) PortableServer

В C++ служащие DSI наследуются от стандартного класса **DynamicImplementation**. Этот класс наследуется от класса **ServantBase** и также определен в пространстве имен **PortableServer**. Динамический Скелетный Интерфейс (*Dynamic Skeleton Interface - DSI*) реализуется через служащих, которые являются членами классов, унаследованных от динамических скелетных классов.

```
// C++

namespace PortableServer
{
   class DynamicImplementation : public virtual ServantBase

   {
      public:
         Object_ptr _this();
         virtual void invoke(

               ServerRequest_ptr request

               ) = 0;
         virtual RepositoryId

         _primary_interface(

               const ObjectId& oid,
               POA_ptr poa

               ) = 0;
   };
}
```

Функция **_this()** возвращает **CORBA::Object_ptr** для целевого объекта. В отличие от **_this()** для статичных скелетных классов, ее возвращаемый тип не является интерфейсо-зависимым, потому что DSI служащий (servant) может очень хорошо воплощать несколько CORBA объектов различных типов. Если **DynamicImplementation::_this()** вызывается за пределами контекста вызова запроса над целевым объектом, обслуживаемым с помощью DSI служащего, она возбуждает исключение **PortableServer::WrongPolicy**.

Метод **invoke()** принимает запросы, заданные к любому CORBA объекту, воплощенному DSI служащим, и выполняет действия, необходимые для исполнения запроса.

Метод **_primary_interface()** принимает значение **ObjectId** и **POA_ptr** в качестве входных параметров и возвращает верный **RepositoryId**, представляющий самый производный интерфейс для этого **oid**.

Ожидается, что методы **invoke()** и **_primary_interface()** будут вызываться только POA в контексте обслуживания CORBA запроса. Вызов этого метода в других условиях может привести к непредсказуемым результатам.

1.39. Функции PortableServer

Объекты, регистрируемые с POA, используют последовательности октетов, точнее **PortableServer::POA::ObjectId** типа, как идентификаторы объектов. Однако, так как C++ программисты часто будут желать использовать строки как идентификаторы объектов, отображение на C++ предоставляет несколько функций приведения, которые конвертируют строки в **ObjectId** и наоборот:

```
// C++

namespace PortableServer
{
      char* ObjectId_to_string(const ObjectId&);
      WChar* ObjectId_to_wstring(const ObjectId&);
      ObjectId* string_to_ObjectId(const char*);
      ObjectId* wstring_to_ObjectId(const WChar*);
}
```

Эти функции следуют обычным правилам отображения на C++ для передачи параметров и управления памятью.

Если приведение **ObjectId** к строке приведет к неверным символам в строке (таким как NUL), первые две функции сгенерируют исключение **CORBA::BAD_PARAM**.

1.40. Отображение для PortableServer::ServantManager

1.40.1. Отображение для Cookie

Так как **PortableServer::ServantLocator::Cookie** это родной (**native**) IDL тип, его тип должен быть указан отображением любого языка. В C++ **Cookie** отображается в **void***:

```
// C++

namespace PortableServer
{
      class ServantLocator {
            ...

            typedef void* Cookie;
      };
}
```

Для отображения на C++ операции **PortableServer::ServantLocator::preinvoke()** параметр **Cookie** отображается в **Cookie&**, в то время как для операции **postinvoke()** он передается как **Cookie**.

1.40.2. Менеджеры Служащих (ServantManager) и Активаторы Адаптеров (AdapterActivator)

Переносимые служащие, которые реализуют интерфейсы **PortableServer::AdapterActivator**, **PortableServer::ServantActivator** или **PortableServer::ServantLocator**, реализуются подобно всем остальным служащим (servant). Они могут использовать либо основанный на наследовании подход, либо подход привязки.

1.40.3. Отображение серверной части для абстрактных интерфейсов

Единственные условия, при которых IDL компилятор должен генерировать C++ код серверной стороны для абстрактных интерфейсов, это либо когда интерфейс является производным от абстрактного интерфейса, либо когда **valuetype** косвенно поддерживает абстрактный интерфейс через один или более промежуточных обычных интерфейсных типов. Абстрактные интерфейсы сами по себе не могут быть прямо реализованными или примененными переносимыми приложениями. Поэтому стандартные C++ скелетные классы для абстрактных интерфейсов не требуются.

Требования отображения серверной части на C++ для абстрактных интерфейсов, следовательно, довольно просты:

- IDL компилятор будет гарантировать, что POA
  скелетные классы для интерфейсов, производных от абстрактных интерфейсов, как-нибудь включают чисто виртуальные функции для IDL операций, определенных в базовом абстрактном интерфейсе(ах). Эти функции могут быть сгенерированными прямо в POA скелетном классе или в реализационно-зависимом базовом классе, от которого наследуется POA
  скелетный класс. Если используется последний способ, это должно быть сделано путем, не требующим вызовов специальных конструкторов поставляемых приложением классов служащих (к примеру, если это будет виртуальный базовый класс без конструктора по умолчанию, он должен требовать от производного класса служащего, чтобы тот явно инициализировал его в списках инициализации полей своих конструкторов).

1.41. Определения C++ для CORBA

Этот раздел предоставляет частичное множество определений C++ для модуля **CORBA**. Определения появляются внутри пространства имен C++, именованного **CORBA**.

```
// C++

namespace CORBA { ... }
```

Все реализации, показанные здесь, являются просто образцовыми реализациями: они являются необязательными определениями для этих типов. Более того, в некоторых случаях эти типы не определяют полные интерфейсы их IDL прототипов, если некоторый тип пропущен одной или более операциями, тем операциям полагается следовать обычным правилам отображения на C++ для их сигнатур, правил передачи параметров, правил управления памятью и т.д.

1.41.1. Простые типы

```
typedef unsigned char             Boolean;
typedef unsigned char             Char;
typedef wchar_t                   WChar;
typedef unsigned char             Octet;
typedef short                     Short;
typedef unsigned short            UShort;
typedef long                      Long;
typedef ...                       LongLong;
typedef unsigned long             ULong;
typedef ...                       ULongLong;
typedef float                     Float;
typedef double                    Double;
typedef long double               LongDouble;


typedef Boolean&                  Boolean_out;
typedef Char&                     Char_out;
```

```
typedef WChar&                  WChar_out;
typedef Octet&                  Octet_out;
typedef Short&                  Short_out;
typedef UShort&                 UShort_out;
typedef Long&                   Long_out;
typedef LongLong&               LongLong_out;
typedef ULong&                  ULong_out;
typedef ULongLong&              ULongLong_out;
typedef Float&                  Float_out;
typedef Double&                 Double_out;
typedef LongDouble&             LongDouble_out;
```

1.41.2. String_var и String_out классы

```
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;


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


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


class String_out
{
   public:
      String_out(char*& p);
      String_out(String_var& p);
      String_out(String_out& s);
      String_out& operator=(String_out& s);
      String_out& operator=(char* p);
      String_out& operator=(const char* p)



      operator char*&();
      char*& ptr();


   private:
      // присваивание из String_var запрещено
      void operator=(const String_var&);
};
```

1.41.3. WString_var и WString_out

Типы **WString_var** и **WString_out** являются идентичными соответственно **String_var** и **String_out**, за исключением того, что они работают над типами широкой строки и широкого символа.

1.41.4. Класс Fixed

```
class Fixed
{
   public:
      // Конструкторы
      Fixed(int val = 0);
      Fixed(unsigned val);
      Fixed(Long val);
      Fixed(ULong val);
      Fixed(LongLong val);
      Fixed(ULongLong val);
      Fixed(Double val);
      Fixed(LongDouble val);
      Fixed(const Fixed& val);
      Fixed(const char *);
      ~Fixed();


      // Приведения
      operator LongLong() const;
      operator LongDouble() const;
      Fixed round(UShort scale) const;
      Fixed truncate(UShort scale) const;


      // Операторы
      Fixed& operator=(const Fixed& val);
      Fixed& operator+=(const Fixed& val);
      Fixed& operator-=(const Fixed& val);
      Fixed& operator*=(const Fixed& val);
      Fixed& operator/=(const Fixed& val);


      Fixed& operator++();
      Fixed operator++(int);
      Fixed& operator--();
      Fixed operator--(int);
      Fixed operator+() const;
      Fixed operator-() const;
      Boolean operator!() const;


      // Другие членские функции
      UShort fixed_digits() const;
      UShort fixed_scale() const;
};


istream& operator>>(istream& is, Fixed& val);
ostream& operator<<(ostream& os, const Fixed& val);


Fixed operator + (const Fixed& val1, const Fixed& val2);
Fixed operator - (const Fixed& val1, const Fixed& val2);
Fixed operator * (const Fixed& val1, const Fixed& val2);
Fixed operator / (const Fixed& val1, const Fixed& val2);


Boolean operator > (const Fixed& val1, const Fixed& val2);
Boolean operator < (const Fixed& val1, const Fixed& val2);
Boolean operator >= (const Fixed& val1, const Fixed& val2);
Boolean operator <= (const Fixed& val1, const Fixed& val2);
Boolean operator == (const Fixed& val1, const Fixed& val2);
Boolean operator != (const Fixed& val1, const Fixed& val2);
```

1.41.5. Класс Any

```
class Any
{
   public:
      Any();
      Any(const Any&);
      Any(TypeCode_ptr tc, void *value,
          Boolean release = FALSE);
      ~Any();
```

```
Any &operator=(const Any&);


// special types needed for boolean, octet, char,
// and bounded string insertion

// these are suggested implementations only

struct from_boolean {
   from_boolean(Boolean b) : val(b) {}
   Boolean val;
};
struct from_octet {
   from_octet(Octet o) : val(o) {}
   Octet val;
};


struct from_char {
   from_char(Char c) : val(c) {}
   Char val;
};
struct from_wchar {
   from_char(WChar c) : val(c) {}
   WChar val;
};
struct from_string {
   from_string(char* s, ULong b,
            Boolean n = FALSE) :
      val(s), bound(b), nocopy(n) {}
   from_string(const char* s, ULong b) :
      val(const_cast<char*>(s)), bound(b),
      nocopy(0) {}
   char *val;
   ULong bound;
   Boolean nocopy;
};
struct from_wstring {
   from_wstring(WChar* s, ULong b,
            Boolean n = FALSE) :
      val(s), bound(b), nocopy(n) {}
   from_wstring(const WChar*, ULong b) :
      val(const_cast<WChar*>(s)), bound(b),
      nocopy(0) {}
   WChar *val;
   ULong bound;
   Boolean nocopy;
};
struct from_fixed {
   from_fixed(const Fixed& f, UShort d, UShort s)

         : val(f), digits(d), scale(s) {}
   const Fixed& val;
   UShort digits;
   UShort scale;
};


void operator<<=(from_boolean);
void operator<<=(from_char);
void operator<<=(from_wchar);
void operator<<=(from_octet);
void operator<<=(from_string);
void operator<<=(from_wstring);
void operator<<=(from_fixed);


// special types needed for boolean, octet,
// char extraction

// these are suggested implementations only

struct to_boolean {
   to_boolean(Boolean &b) : ref(b) {}
   Boolean &ref;
};
struct to_char {
   to_char(Char &c) : ref(c) {}
   Char &ref;
};
struct to_wchar {
   to_wchar(WChar &c) : ref(c) {}
   WChar &ref;
```

```
      };
      struct to_octet {
         to_octet(Octet &o) : ref(o) {}
         Octet &ref;
      };
      struct to_object {
         to_object(Object_out obj) : ref(obj) {}
         Object_ptr &ref;
      };
      struct to_string {
         to_string(const char *&s, ULong b)

            : val(s), bound(b) {}
         const char *&val;
         ULong bound;


         // the following constructor is deprecated

         to_string(char *&s, ULong b) : val(s), bound(b) {}
      };
      struct to_wstring {
         to_wstring(const WChar *&s, ULong b)

            : val(s), bound(b) {}
         const WChar *&val;
         ULong bound;


         // the following constructor is deprecated

         to_wstring(WChar *&s, ULong b)

            : val(s), bound(b) {}
      };
      struct to_fixed {
         to_fixed(Fixed& f, UShort d, UShort s)

            : val(f), digits(d), scale(s) {}
         Fixed& val;
         UShort digits;
         UShort scale;
      };
      struct to_abstract_base {
         to_abstract_base(AbstractBase_ptr& base)

            : ref(base) {}
         AbstractBase_ptr& ref;
      };


      Boolean operator>>=(to_boolean) const;
      Boolean operator>>=(to_char) const;
      Boolean operator>>=(to_wchar) const;
      Boolean operator>>=(to_octet) const;
      Boolean operator>>=(to_object) const;
      Boolean operator>>=(to_string) const;
      Boolean operator>>=(to_wstring) const;
      Boolean operator>>=(to_fixed) const;
      Boolean operator>>=(to_abstract_base) const;


      void replace(TypeCode_ptr, void *value,
               Boolean release = FALSE);


      TypeCode_ptr type() const;
      void type(TypeCode_ptr);
      const void *value() const;


   private:
      // these are hidden and should not be implemented

      // so as to catch erroneous attempts to insert

      // or extract multiple IDL types mapped to unsigned char

      void operator<<=(unsigned char);
      Boolean operator>>=(unsigned char&) const;
};
```

```
void operator<<=(Any&, Short);
void operator<<=(Any&, UShort);
void operator<<=(Any&, Long);
void operator<<=(Any&, ULong);
void operator<<=(Any&, Float);
void operator<<=(Any&, Double);
void operator<<=(Any&, LongLong);
void operator<<=(Any&, ULongLong);
void operator<<=(Any&, LongDouble);
void operator<<=(Any&, const Any&);  // copying

void operator<<=(Any&, Any*);        // non-copying

void operator<<=(Any&, const char*);
void operator<<=(Any&, const WChar*);


Boolean operator>>=(const Any&, Short&);
Boolean operator>>=(const Any&, UShort&);
Boolean operator>>=(const Any&, Long&);
Boolean operator>>=(const Any&, ULong&);
Boolean operator>>=(const Any&, Float&);
Boolean operator>>=(const Any&, Double&);
Boolean operator>>=(const Any&, LongLong&);
Boolean operator>>=(const Any&, ULongLong&);
Boolean operator>>=(const Any&, LongDouble&);
Boolean operator>>=(const Any&, const Any*&);
Boolean operator>>=(const Any&, const char*&);
Boolean operator>>=(const Any&, const WChar*&);
```

1.41.6. Класс Any_var

```
class Any_var
{
   public:
      Any_var();
      Any_var(Any *a);
      Any_var(const Any_var &a);
      ~Any_var();


      Any_var &operator=(Any *a);
      Any_var &operator=(const Any_var &a);


      Any *operator->();


      const Any& in() const;
      Any& inout();
      Any*& out();
      Any* _retn();


      // other conversion operators for parameter passing

};
```

1.41.7. Класс Exception

```
// C++

class Exception
{
   public:
      Exception(const Exception &);
      virtual ~Exception();
      Exception &operator=(const Exception &);


      virtual void _raise() const = 0;
```

```
   protected:
      Exception();
};
```

1.41.8. Класс SystemException

```
// C++

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);


      CompletionStatus completed() const;
      void completed(CompletionStatus);


      virtual void _raise() const = 0;


      static SystemException* _downcast(Exception*);
      static const SystemException* _downcast(

                        const Exception*

      );
};
```

1.41.9. Класс UserException

```
// C++

class UserException : public Exception
{
   public:
      UserException();
      UserException(const UserException &);
      ~UserException();
      UserException &operator=(const UserException &);


      virtual void _raise() const = 0;


      static UserException* _downcast(Exception*);
      static const UserException* _downcast(

                        const Exception*

      );
};
```

1.41.10. Класс UnknownUserException

```
// C++

class UnknownUserException : public UserException
{
   public:
      Any &exception();
      static UnknownUserException* _downcast(Exception*);
```

```
      static const UnknownUserException* _downcast(

                           const Exception*

      );
      virtual void raise();
};
```

1.41.11. release и is_nil

```
// C++

namespace CORBA {
      void release(Object_ptr);
      void release(Environment_ptr);
      void release(NamedValue_ptr);
      void release(NVList_ptr);
      void release(Request_ptr);
      void release(Context_ptr);
      void release(TypeCode_ptr);
      void release(POA_ptr);
      void release(ORB_ptr);


      Boolean is_nil(Object_ptr);
      Boolean is_nil(Environment_ptr);
      Boolean is_nil(NamedValue_ptr);
      Boolean is_nil(NVList_ptr);
      Boolean is_nil(Request_ptr);
      Boolean is_nil(Context_ptr);
      Boolean is_nil(TypeCode_ptr);
      Boolean is_nil(POA_ptr);
      Boolean is_nil(ORB_ptr);
      ...

}
```

1.41.12. Класс Object

```
// C++

class Object
{
   public:
      static Object_ptr _duplicate(Object_ptr obj);
      static Object_ptr _nil();
      InterfaceDef_ptr _get_interface();
      Boolean _is_a(const char* logical_type_id);
      Boolean _non_existent();
      Boolean _is_equivalent(Object_ptr other_object);
      ULong _hash(ULong maximum);
      void _create_request(

            Context_ptr ctx,
            const char *operation,
            NVList_ptr arg_list,
            NamedValue_ptr result,
            Request_out request,
            Flags req_flags

      );
      void _create_request(

            Context_ptr ctx,
            const char *operation,
            NVList_ptr arg_list,
            NamedValue_ptr result,
            ExceptionList_ptr,
            ContextList_ptr,
            Request_out request,
            Flags req_flags

      );
      Request_ptr _request(const char* operation);
      Policy_ptr _get_policy(PolicyType policy_type);
      DomainManagerList* _get_domain_managers();
```

```
      Object_ptr _set_policy_overrides(

            const PolicyList& policies,
            SetOverrideType set_or_add

      );
};
```

1.41.13. Класс Environment

```
// C++

class Environment
{
   public:
      void exception(Exception*);
      Exception *exception() const;
      void clear();


      static Environment_ptr _duplicate(Environment_ptr ev);
      static Environment_ptr _nil();
};
```

1.41.14. Класс NamedValue

```
// C++

class NamedValue
{
   public:
      const char *name() const;
      Any *value() const;
      Flags flags() const;


      static NamedValue_ptr _duplicate(NamedValue_ptr nv);
      static NamedValue_ptr _nil();
};
```

1.41.15. Класс NVList

```
// C++

class NVList
{
   public:
      ULong count() const;
      NamedValue_ptr add(Flags);
      NamedValue_ptr add_item(const char*, Flags);
      NamedValue_ptr add_value(const char*, const Any&,
                        Flags);
      NamedValue_ptr add_item_consume(

                  char*,
                  Flags

      );
      NamedValue_ptr add_value_consume(

                  char*,
                  Any *,
                  Flags

      );
      NamedValue_ptr item(ULong);
      void remove(ULong);


      static NVList_ptr _duplicate(NVList_ptr nv);
      static NVList_ptr _nil();
};
```

1.41.16. Класс ExceptionList

```
// C++

class ExceptionList
{
   public:
      ULong count();
      void add(TypeCode_ptr tc);
      void add_consume(TypeCode_ptr tc);
      TypeCode_ptr item(ULong index);
      void remove(ULong index);
};
```

1.41.17. Класс ContextList

```
class ContextList
{
   public:
      ULong count();
      void add(const char* ctxt);
      void add_consume(char* ctxt);
      const char* item(ULong index);
      void remove(ULong index);
};
```

1.41.18. Класс Request

```
// C++

class Request
{
   public:
      Object_ptr target() const;
      const char *operation() const;
      NVList_ptr arguments();
      NamedValue_ptr result();
      Environment_ptr env();
      ExceptionList_ptr exceptions();
      ContextList_ptr contexts();


      void ctx(Context_ptr);
      Context_ptr ctx() const;


      Any& add_in_arg();
      Any& add_in_arg(const char* name);
      Any& add_inout_arg();
      Any& add_inout_arg(const char* name);
      Any& add_out_arg();
      Any& add_out_arg(const char* name);
      void set_return_type(TypeCode_ptr tc);
      Any& return_value();


      void invoke();
      void send_oneway();
      void send_deferred();
      void get_response();
      Boolean poll_response();


      static Request_ptr _duplicate(Request_ptr req);
      static Request_ptr _nil();
};
```

1.41.19. Класс Context

```
// C++

class Context
{
   public:
      const char *context_name() const;
      Context_ptr parent() const;


      void create_child(const char*, Context_out);
      void set_one_value(const char*, const Any&);
      void set_values(NVList_ptr);


      void delete_values(const char*);
      void get_values(const char*, Flags, const char*,
                  NVList_out);


      static Context_ptr _duplicate(Context_ptr ctx);
      static Context_ptr _nil();
};
```

1.41.20. Класс TypeCode

```
// C++

class TypeCode
{
   public:
      class Bounds : public UserException { ... };
      class BadKind : public UserException { ... };


      TCKind kind() const;
      Boolean equal(TypeCode_ptr) const;
      Boolean equivalent(TypeCode_ptr) const;
      TypeCode_ptr get_compact_typecode() const;


      const char* id() const;
      const char* name() const;


      ULong member_count() const;
      const char* member_name(ULong index) const;


      TypeCode_ptr member_type(ULong index) const;


      Any *member_label(ULong index) const;
      TypeCode_ptr discriminator_type() const;
      Long default_index() const;


      ULong length() const;


      TypeCode_ptr content_type() const;


      UShort fixed_digits() const;
      Short fixed_scale() const;


      Visibility member_visibility(ULong index) const;
      ValuetypeModifier type_modifier() const;
      TypeCode_ptr concrete_base_type() const;


      static TypeCode_ptr _duplicate(TypeCode_ptr tc);
      static TypeCode_ptr _nil();
};
```

1.41.21. Класс ORB

```
// C++

class ORB
{
   public:
      typedef sequence<Request_ptr> RequestSeq;
      char *object_to_string(Object_ptr);
      Object_ptr string_to_object(const char*);
      void create_list(Long, NVList_out);
      void create_operation_list(OperationDef_ptr,
                        NVList_out);
      void create_named_value(NamedValue_out);
      void create_exception_list(ExceptionList_out);
      void create_context_list(ContextList_out);
      void get_default_context(Context_out);
      void create_environment(Environment_out);
      void send_multiple_requests_oneway(

                  const RequestSeq&

      );
      void send_multiple_requests_deferred(

                  const RequestSeq&



      );
      Boolean poll_next_response();
      void get_next_response(Request_out);


      // Obtaining initial object references

      typedef char* ObjectId;
      class ObjectIdList {...};
      class InvalidName : public UserException {...};
      ObjectIdList *list_initial_services();
      Object_ptr resolve_initial_references(

                  const char *identifier

      );
      Boolean work_pending();
      void perform_work();
      void shutdown(Boolean wait_for_completion);
      void run();


      Boolean get_service_information(

                  ServiceType svc_type,
                  ServiceInformation_out svc_info

      );


      typedef char* ObjectId;
      class ObjectIdList { ... };
      Object_ptr resolve_initial_references(const char* id);
      ObjectIdList* list_initial_services();


      Policy_ptr create_policy(

                  PolicyType type,
                  const Any& val

      );
      static ORB_ptr _duplicate(ORB_ptr orb);
      static ORB_ptr _nil();
};
```

1.41.22. Инициализация ORB

```
// C++

typedef char* ORBid;
static ORB_ptr ORB_init(
```

```
                int& argc,
                char** argv,
                const char* orb_identifier = ""

            );
```

1.41.23. Общий тип T_out

```
// C++

class T_out
{
   public:
      T_out(T*& p) : ptr_(p) { ptr_ = 0; }
      T_out(T_var& p) : ptr_(p.ptr_) {
         delete ptr_;
         ptr_ = 0;
      }
      T_out(T_out& p) : ptr_(p.ptr_) {}
      T_out& operator=(T_out& p) {
         ptr_ = p.ptr_;
         return *this;
      }
      T_out& operator=(T* p) { ptr_ = p; return *this; }


      operator T*&() { return ptr_; }
      T*& ptr() { return ptr_; }


      T* operator->() { return ptr_; }


   private:
      T*& ptr_;
      // assignment from T_var not allowed

      void operator=(const T_var&);
};
```

1.42. Альтернативные Отображения Для Диалектов C++

1.42.1. Без пространств имен

Если целевая среда не поддерживает конструкцию **namespace**, но поддерживает вложенные классы, тогда модуль должен быть отображен в C++ класс. Если среда не поддерживает вложенные классы, тогда отображение для модулей должно быть таким же, как для отображения на C (конкатенация идентификаторов с использованием символов подчеркивания "_" в качестве разделителя). Заметим, что ограничения модуля отображаются в ограничения файловой области видимости в системах, которые поддерживают пространства имен, и в ограничения области видимости класса в системах, которые отображают модули в классы.

1.42.2. Без обработки исключений

Для тех C++ сред, которые не поддерживают настоящую обработку C++ исключений, упоминаемых здесь как ***C++ среды без обработки исключений*** (*non-exception handling (non-EH) C++ environments*), параметр **Environment**, передаваемый в каждую операцию, используется для передачи информации об исключении вызывающему.

Как показано в разделе 1.27 "Среда (Environment)", класс **Environment** поддерживает возможность доступа и изменения исключения, которое она содержит.

Как показано в разделе 1.19. "Отображение для типов исключений", пользовательские и системные исключения формируют иерархию наследования, которая обычно позволяет вылавливать типы или по их фактическому типу, или по более общему базовому типу. Когда используется C++ среда без обработки исключений, функции сужения, предоставляемые этой иерархией , позволяют проверку и манипулирование исключениями:

```
// IDL

interface A
{
   exception Broken { ... };
   void op() raises(Broken);
};


// C++

Environment ev;
A_ptr obj = ...

obj->op(ev);
if (Exception *exc = ev.exception()) {
   if (A::Broken *b = A::Broken::_narrow(exc)) {
      // deal with user exception

   } else {
      // must have been a system exception

      SystemException *se = SystemException::_narrow(exc);
      ...

   }
}
```

Раздел 1.33. "ORB" определяет, что **Environment** должна быть создана, используя **ORB::create_environment**, но это является слишком ограничивающим для реализаций, требующих, чтобы **Environment** была передана как аргумент в каждый вызов метода. Для реализаций, которые не поддерживают настоящие C++ исключения, **Environment** может быть выделена как статичная или динамическая переменная. К примеру, все нижеследующее является правильными определениями в C++ среде без обработки прерываний:

```
// C++

Environment global_env;              // global

static Environment static_env;       // file static


class MyClass
{
   public:
      ...

   private:
      static Environment class_env; // class static

};


void func()
{
   Environment auto_env; // auto

   Environment *new_env = new Environment;// heap

   ...

}
```

Для каждого использования параметры **Environment** передаются по ссылке:

```
// IDL

interface A
{
   exception Broken { ... };
```

```
   void op() raises(Broken);
};


// C++

class A ...
{
   public:
      void op(Environment &);
      ...

};
```

Для дополнительной простоты использования в C++ средах без обработки исключений **Environment** должен поддерживать конструирование копированием и присваивание из других объектов **Environment**. Эти дополнительные возможности полезны для распространения исключений из одной **Environment** в другую среду без обработки исключений..

Когда исключение "возбуждается" в C++ среде без обработки исключений, реализаторы объектов и ORB во время исполнения должны гарантировать, что все **out** и возвращаемые указатели будут возвращены вызывающему как нулевые. Если неинициализированные или "мусорные" значения указателей будут возвращены, код клиентского приложения может получить ошибки времени выполнения из-за присваивания плохих указателей в типы **T_var**. Когда **T_var** выходит за границы области видимости, он пытается удалить (**delete**) данный ему **T***, если значение этого указателя недействительно, непременно будет возникать ошибка времени исполнения. Исключениям в C++ среде без обработки исключений не нужно поддерживать виртуальную функцию **_raise()**, так как только полезная реализация ее в такой среде будет вызывать прерывание программы.

1.42. Ключевые слова C++

Таблица 1-7 перечисляет все ключевые слова C++ из Рабочего Листа (Working Paper) 2 декабря 1996 ANSI (X3J16) Комитета Стандартизации Языка C++.

Таблица 1-7 Ключевые слова C++

and	**and_aq**	**asm**	**auto**	**bitand**	**bitor**
bool	**break**	**case**	**catch**	**char**	**class**
compl	**const**	**const_cast**	**contimue**	**default**	**delete**
do	**double**	**dynamic_cast**	**else**	**enum**	**explicit**
export	**extern**	**false**	**float**	**for**	**friend**
goto	**if**	**inline**	**int**	**long**	**mutable**
namespace	**new**	**not**	**not_eq**	**operator**	**or**
or_eq	**private**	**protected**	**public**	**register**	**reinterpret_cast**
return	**short**	**signed**	**sizeof**	**static**	**static_cast**
struct	**switch**	**template**	**this**	**throw**	**true**
try	**typedef**	**typeid**	**Typename**	**union**	**unsigned**
using	**virtual**	**void**	**Volatile**	**wchar_t**	**while**
xor	**xor_eq**				