##### Default literal for an enumeration

Normally the default value for an object of a type is pre-defined based on the generic rules based on the characteristics of the type. For example, for an integer it would be the value zero and for an enumeration it is the constant with the lowest member ID.

This generic rule is not desirable in some situations. The annotation @DefaultMember allows this behavior to be changed.

@annotation DefaultMember {};

The application to enumerated types is illustrated in the example below:

enum MyEnum {

ENUM1,

ENUM2,

@DefaultMember ENUM3,

ENUM4

};

##### TryConstruct Elements and Members

The construction of an object of a collection or aggregate type operates recursively; it requires constructing objects of the nested element/member types. Therefore failure to construct any object of the nested element/member type failure may impact the ability to construct the whole collection/aggregate type:

* In some cases the consequence will be that there is no object of the collection/aggregate type that can be constructed.
* In other cases the failure in the nested element/member will be mitigated and the collection/aggregate object successfully created.

The specific behavior depends on the TryConstruct behavior associated with the element or member of the type being constructed as described in 7.2.2.5.

The TryConstruct annotation is used to explicitly set the TryConstruct behavior of element of a collection type and/or member of an Aggregate type.

The IDL definition of the TryConstruct annotation is:

enum TryConstructFailAction {

DISCARD,

USE\_DEFAULT,

TRIM

};

@annotation TryConstruct {

TryConstructFailAction value default USE\_DEFAULT;

};

As specified in 7.2.2.5 the default behavior is DISCARD. Therefore if the TryConstruct annotation is not specified it is the same as if it had been explicitly set to DISCARD. For example:

struct T1 {

long important\_member;

@TryConstruct(DISCARD) string<4> m1;

};

Is the same as:

struct T1 {

long important\_member;

string<4> m1;

};

If the TryConstruct annotation is specified without a value, or if the value is set to USE\_EFAULT, then the TryConstruct behavior is USE\_EFAULT as specified in 7.2.2.5. This means the element or member will be constructed to have its default value (according to its type as described in Table 9) and does not cause the aggregate container to fail the construction.

As specified in 7.2.2.5, the TryConstruct annotation may be used in structure and union members, the union discriminator, the elements of arrays and sequences, and the key and/or values of map types.

###### TryConstruct Example 1

Assume T1 is defined:

struct T1 {

long a\_long;

@TryConstruct(USE\_DEFAULT) string<5> member;

};

Or alternatively T1 is defined:

struct T1 {

long a\_long;

@TryConstruct string<5> member;

};

Assume further that T2 is defined as:

struct T2 {

long a\_long;

string<32> m2;

};

In this situation if O2 is an object of type T2, and the value of the nested member object O2.member is the string “Hello World!”, then O2.member cannot construct any object of type String4 (string<5>). However since the TryConstruct behavior associated with the Ti member “member” is USE\_DEFAULT, then the failure is mitigated and a O1 object of type T1 can be successfully constructed. The constructed object would have O1.member set to the empty string.

###### TryConstruct Example 2

Assume T1 and T2 are is defined as:

struct T1 {

long a\_long;

@TryConstruct(TRIM) string<5> member;

};

struct T2 {

long a\_long;

string<32> member;

};

In this situation if O2 is an object of type T2, and the value of the nested member object O2.member is the string “Hello World!”, then the object O2.member cannot construct any object of the type of the corresponding member of T1 (string<5>). However, since the TryConstruct behavior associated with the member is TRIM, then the failure is mitigated and an object O1 of type T1 can be successfully constructed. The constructed object would have O1.member contain the characters of O2.member that can fit on its string<5> type, that is, the string “Hello”.

###### TryConstruct Example 3

Assume T1 and T2 are defined as:

struct T1 {

long a\_long;

@TryConstruct(TRIM) sequence<long,4> member;

};

struct T2 {

long a\_long;

sequence<long,32> member;

};

In this situation if O2 is an object of type T2, and the value of the nested member object O2.member is the sequence of longs [1, 2, 3, 4, 5, 6, 7, 8], then the object O2.member cannot construct any object of the type of the corresponding member of T1 (sequence<long,4>). However since the TryConstruct behavior associated with the member is TRIM, then the failure is mitigated and an object O1 of type T1 can be successfully constructed. The constructed object would have O1.member as a sequence of 4 longs containing the first four elements of O2.member.

###### TryConstruct Example 4

Assume T1 and T2 are defined as:

typedef string<5> String5;

struct T1 {

long a\_long;

sequence<@TryConstruct(TRIM) String5, 10> member;

};

typedef string<16> String16;

struct T2 {

long a\_long;

sequence<String16, 10> member;

};

In this situation if O2 is an object of type T2, and the value of the nested member object O2.member is a sequence of String16 where the first element (O2.member[0]) is “Hello World” , then the object O2.member [0] cannot construct any object of the type of the corresponding element of T1 (String5). However since the TryConstruct behavior associated with the element of the sequence is TRIM, then the failure is mitigated and an object O1 of type T1 can be successfully constructed. The constructed object would have O1.member[0] as the string “Hello” (i.e. the result of trimming “Hello World!” to the length that can fit into the String5 element type).

###### TryConstruct Example 5

Assume T1 and T2 are defined as:

enum T1Enum {

ENUM1,

@DefaultValue ENUM2

};

union T1 switch ( T1Enum ) {

case ENUM1:

long e1\_value;

case ENUM2:

long e2\_value;

};

enum T2Enum {

ENUM1,

@DefaultValue ENUM2,

ENUM3

};

union T2 switch ( T2Enum ) {

case ENUM1:

long e1\_value;

case ENUM2:

long e2\_value;

case ENUM3:

long e3\_value;

};

In this situation if O2 is an object of type T2, and the value of the discriminator is ENUM3, then O2.discriminator cannot construct an object of type T1Enum and as a consequence O2 cannot construct any object of type T1.

However if T1 and T2 had been defined to have USE\_DEFAULT TryConstruct behavior for the discriminator as in:

union T1 switch ( @TryConstruct T1Enum ) {

case ENUM1:

long e1\_value;

case ENUM2:

long e2\_value;

};

union T2 switch (T2Enum ) {

case ENUM1:

long e1\_value;

case ENUM2:

long e2\_value;

case ENUM3:

long e3\_value;

};

Then in this situation the failure to construct a T1Enum from O2.discriminator would be mitigated and O1.discriminator would be set to its default value (ENUM2) and O1.e1\_value would be constructed from O2.e3\_value. This would allow the successful construction of a O1 object of type T1.