Automatic Database Migration

On this page:

The default transformations performed by `transformdb` preserve as much information as possible. However, there are practical limits to the tool's capabilities, since the only information it has is obtained by performing a comparison of the Slice definitions.

For example, suppose our old definition for a structure is the following:

Slice

```
struct AStruct {
    int i;
};
```

We want to migrate instances of this struct to the following revised definition:

Slice

```
struct AStruct {
    int j;
};
```

As the developers, we know that the `int` member has been renamed from `i` to `j`, but to `transformdb` it appears that member `i` was removed and member `j` was added. The default transformation results in exactly that behavior: the value of `i` is lost, and `j` is initialized to a default value. If we need to preserve the value of `i` and transfer it to `j`, then we need to use custom migration.

The changes that occur as a type system evolves can be grouped into three categories:

- Data members
  The data members of class and structure types are added, removed, or renamed. As discussed above, the default transformations initialize new and renamed data members to default values.

- Type names
  Types are added, removed, or renamed. New types do not pose a problem for database migration when used to define a new data member; the member is initialized with default values as usual. On the other hand, if the new type replaces the type of an existing data member, then type compatibility becomes a factor (see the following item).

  Removed types generally do not cause problems either, because any uses of that type must have been removed from the new Slice definitions (e.g., by removing data members of that type). There is one case, however, where removed types become an issue, and that is for polymorphic classes.

  Renamed types are a concern, just like renamed data members, because of the potential for losing information during migration. This is another situation for which custom migration is recommended.

- Type content
  Examples of changes of type content include the key type of a dictionary, the element type of a sequence, or the type of a data member. If the old and new types are not compatible, then the default transformation emits a warning, discards the current value, and reinitializes it with a default value.

Type Compatibility Rules for Automatic Migration

Changes in the type of a value are restricted to certain sets of compatible changes. This section describes the type changes supported by the default transformations. All incompatible type changes result in a warning indicating that the current value is being discarded and a default value for the new type assigned in its place. Additional flexibility is provided by custom migration.

Boolean

A value of type `bool` can be transformed to and from `string`. The legal string values for a `bool` value are `"true"` and `"false"`.

Integer

The integer types `byte`, `short`, `int`, and `long` can be transformed into each other, but only if the current value is within range of the new type. These integer types can also be transformed into `string`.

Floating Point

The floating-point types `float` and `double` can be transformed into each other, as well as to `string`. No attempt is made to detect a loss of precision during transformation.

String

A `string` value can be transformed into any of the primitive types, as well as into enumeration and proxy types, but only if the value is a legal string representation of the new type. For example, the string value `"Pear"` can be transformed into the enumeration `Fruit`, but only if `Pear` is an enumerator of `Fruit`.

Enum

An enumeration can be transformed into an enumeration with the same type ID, or into a string. Transformation between enumerations is performed symbolically. For example, consider our old type below:

Slice

```
enum Fruit { Apple, Orange, Pear };
```

Suppose the enumerator `Pear` is being transformed into the following new type:

Slice

```
enum Fruit { Apple, Pear };
```

The transformed value in the new enumeration is also `Pear`, despite the fact that `Pear` has changed positions in the new type. However, if the old value had been `Orange`, then the default transformation emits a warning because that enumerator no longer exists, and initializes the new value to `Apple` (the default value).

If an enumerator has been renamed, then custom migration is required to convert enumerators from the old name to the new one.

Sequence

A sequence can be transformed into another sequence type, even if the new sequence type does not have the same type id as the old type, but only if the element types are compatible. For example, `sequence<short>` can be transformed into `sequence<int>`, regardless of the names given to the sequence types.

Dictionary

A dictionary can be transformed into another dictionary type, even if the new dictionary type does not have the same type ID as the old type, but only if the key and value types are compatible. For example, `dictionary<int, string>` can be transformed into `dictionary<long, string>`, regardless of the names given to the dictionary types.

Caution is required when changing the key type of a dictionary, because the default transformation of keys could result in duplication. For example, if the key type changes from `int` to `short`, any `int` value outside the range of `short` results in the key being initialized to a default value (namely zero). If zero is already used as a key in the dictionary, or another out-of-range key is encountered, then a duplication occurs. The transformation handles key duplication by removing the duplicate element from the transformed dictionary. (Custom migration can be useful in these situations if the default behavior is not acceptable.)

Structure

A `struct` type can only be transformed into another `struct` type with the same type ID. Data members are transformed as appropriate for their types.

Proxy

A proxy value can be transformed into another proxy type, or into `string`. Transformation into another proxy type is done with the same semantics as in a language mapping: if the new type does not match the old type, then the new type must be a base type of the old type (that is, the proxy is widened).

Class

A `class` type can only be transformed into another `class` type with the same type ID. A data member of a `class` type is allowed to be widened to a base type. Data members are transformed as appropriate for their types. See Transforming Objects for more information on transforming classes.

Default Values for Automatic Migration

Data types are initialized with default values, as shown.

Type	Default Value
Boolean	`false`
Numeric	Zero (`0`)
String	Empty string
Enumeration	The first enumerator
Sequence	Empty sequence
Dictionary	Empty dictionary
Struct	Data members are initialized recursively
Proxy	Nil
Class	Nil

Running an Automatic Migration

In order to use automatic transformation, we need to supply the following information to `transformdb`:

- The old and new Slice definitions
- The old and new types for the database key and value
- The database environment directory, the database file name, and the name of a new database environment directory to hold the transformed database

Here is an example of a `transformdb` command:

```
$ transformdb --old old/MyApp.ice --new new/MyApp.ice \
--key int,string --value ::Employee db emp.db newdb
```

Briefly, the `--old` and `--new` options specify the old and new Slice definitions, respectively. These options can be specified as many times as necessary in order to load all of the relevant definitions. The `--key` option indicates that the database key is evolving from `int` to `string`. The `--value` option specifies that `::Employee` is used as the database value type in both old and new type definitions, and therefore only needs to be specified once. Finally, we provide the pathname of the database environment directory (`db`), the file name of the database (`emp.db`), and the pathname of the database environment directory for the transformed database (`newdb`).

See Also

- Custom Database Migration
- Type IDs
- Using transformdb