Using Objectivity/C++ Data Definition Language

Version 4
Using Objectivity/C++ Data Definition Language

Version 4, Release 4.0, July 14, 1997

The information in this document is subject to change without notice. Objectivity, Inc. assumes no responsibility for any errors that may appear in this document.

Copyright 1996 Objectivity, Inc. All rights reserved. This document may not be copied, photocopied, reproduced, translated or converted to any electronic or machine-readable form in whole or in part without prior written approval of Objectivity, Inc.

Objectivity and Objectivity/DB are registered trademarks of Objectivity, Inc. Objectivity/DB Fault Tolerant Option, Objectivity/FTO, Objectivity/DB Data Replication Option, Objectivity/DRO, Objectivity/C++, Objectivity/C++ Data Definition Language, Objectivity/DDL, Objectivity/C++ Tools.h++, Objectivity/Tools.h++, Objectivity/Smalltalk, Objectivity/SQL++, Objectivity/SQL++ ODBC Driver, and Objectivity/ODBC are trademarks of Objectivity, Inc. Other brands and products are registered trademarks or trademarks of their respective holders.

The software and information contained herein are proprietary to, and comprise valuable trade secrets of, Objectivity, Inc., which intends to preserve as trade secrets such software and information. This software is furnished pursuant to a written license agreement and may be used, copied, transmitted, and stored only in accordance with the terms of such license and with the inclusion of the above copyright notice. This software and information or any other copies thereof may not be provided or otherwise made available to any other person.

U. S. Government Restricted Rights: Use, duplication or disclosure of the software or other information by the U. S. Government or any unit or agency thereof is subject to restrictions as set forth in subparagraph (c) (1) (ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 and the Government is acquiring only restricted rights in the software and limited rights in any technical data provided (as such terms are defined in such clause of the DFARS). If the software or other information is supplied to any unit or agency of the U. S. other than the Department of Defense, the Government’s rights will be as defined in clause 52.227-19(c)(2) of the FAR or, in the case of NASA, in clause 18-52.227-86 (d) of the NASA Supplement to the FAR.
Contents

About This Book ix

Audience ix
Organization ix

Chapter 1 Processing Data Models 1-1

Overview 1-1
Creating a Schema 1-5
Using the DDL Processor 1-6
  Running the DDL Processor 1-6
  Example of Using the C++ DDL Processor 1-6
  Specifying the Boot File Path for VMS 1-8
Using Files Generated by the C++ DDL Processor 1-9
  Overview 1-9
  Including Generated Header Files 1-10
Using Preprocessor Directives 1-16
  Using #include Directives 1-16
  Using Other Directives 1-16
  Automatic Inclusion of oo.h 1-16
Using Multiple Schema Files 1-17
Modifying the Schema 1-19
  Making Additions that Do Not Require Federated Database Changes 1-20

Chapter 2 Data Definition Language 2-1

Using the DDL 2-1
Class Declarations 2-3
  Data Members 2-3
  Member Functions and Functions 2-3
Constructors and Destructors 2-3
Inheritance 2-3
Persistent-Capable Classes 2-4
Defining a Persistent-Capable Class 2-4
System-Defined Persistent-Capable Classes 2-4
Primitive Types 2-8
   Integer 2-9
   Floating Point 2-10
   Character 2-10
   C++ Primitive Types 2-11
Aggregate Data Types 2-12
   Variable-Size Arrays (VArrays) 2-12
Associations 2-16
   Notation for Class Association Links 2-16
   Bidirectional Association Links 2-18
   Unidirectional Association Links 2-20
   N-ary Associations 2-22
   Association Links 2-24
   Composite Objects 2-29
   Association Links and Object Copying 2-31
   Association Links and Versioning 2-37
   Combining Copy, Lock, Delete, and Versioning 2-38
Templates 2-40
Special Cases 2-42
   Unions and Bit Fields 2-42
   Virtual Base Classes 2-45
   Nested Classes 2-45
   Member Pointers 2-45
Constraints 2-46
   Domain Constraints 2-46
   Object-Level Constraints 2-46
DDL-Generated Constructors 2-46
Chapter 3  Multiple Data Models  3-1

- Schema Names  3-1
- Adding Classes to Alternate Schemas  3-2
  - DDL Files Containing References to One Schema  3-2
  - DDL Files Containing References to Multiple Schemas  3-4

Chapter 4  Multiple Inheritance  4-1

- Vehicle Data Model  4-2
- Persistence through Inheritance  4-4
- Data Modeling Using Root Persistence  4-4
- Using Composite Objects Instead of Multiple Inheritance  4-7
- Data Modeling Using Leaf Persistence  4-9
- Enhanced Leaf Persistence  4-12
- Mixing Root and Leaf Persistence  4-14

Chapter 5  Schema Evolution  5-1

- Basics  5-1
  - Deferred Mode  5-2
  - On-Demand Mode  5-3
  - Immediate Mode with an Upgrade Application  5-3
  - Conversion Function  5-3
  - Other User Applications  5-4
- Effects of Schema Evolution  5-4
  - Effects on Existing Persistent Objects  5-4
- Sequential Schema Evolutions  5-5
- Basic and Complex Operations  5-5
- General Schema Evolution Process  5-6
  - Preparation Phase  5-7
  - Schema Evolution Phase  5-9
  - Cleanup Phase  5-11
Basic Class Content Operations 5-11

Effects of a Class Content Operation on Other Classes 5-12

Initial Schema Used for Class Content Examples 5-13

Deleting a Data Member 5-15

Adding a Data Member 5-16

Renaming a Data Member 5-17

Changing a Primitive Data Member Type 5-19

Changing the Order (Position) of a Data Member 5-21

Changing the Access Control of a Data Member 5-22

Adding the First Virtual Member Function 5-23

Removing the Last Virtual Member Function 5-24

Changing the Size of a Fixed-Length Array Data Member 5-24

Adding an Inline Association 5-26

Adding a Non-Inline Association 5-27

Adding an Object Reference 5-29

Deleting an Association 5-30

Deleting an Object Reference 5-31

Changing an Object Reference from Short to Long or Vice Versa 5-32

Changing the Representation (Inline/Non-Inline) of an Association 5-34

Changing an Inline Association from Short to Long or Vice Versa 5-35

Changing Association Behavior Specifiers 5-37

Renaming an Association 5-38

Changing the Access Control of an Association 5-39

Changing an Array Size for an Array of Object References 5-40

Changing the Order (Position) of a Non-Inline Association 5-41

Changing the Order (Position) of an Inline Association or Object Reference 5-42

Complex Class Content Operations 5-43

Adding a Data Member and Copying Its Value 5-43

Replacing a User-Defined Type with Another Related Type 5-43

Changing the Origin Class of a Data Member to a Base Class 5-45

Changing the Origin Class of a Data Member to a Derived Class 5-47

Changing the Number of Dimensions of a Fixed-Size Array Data Member 5-49

Changing a Fixed-Size Array to a Variable-Size Array 5-49
Changing the Type of a Non-Primitive Data Member  5-49
Changing an Object Reference to an Association  5-50
Changing an Association to an Object Reference  5-50
Changing Association Cardinality  5-50
Changing the Class where a Unidirectional Association Originates  5-51
Change the Domain Class of an Association  5-52
Basic Class Operations  5-53
  Initial Schema Used for Basic Class Examples  5-53
  Renaming a Class  5-53
  Deleting a Class  5-55
Complex Class Operations  5-58
  Deleting a Class with Inherited Associations or Object References  5-58
Basic Inheritance Operations  5-60
  Adding a Non-Persistent-Capable Class as a Base Class  5-60
  Removing a Non-Persistent-Capable Class as a Base Class  5-62
  Replacing a Base Class with a Subset of Its Base Classes  5-64
  Replacing a Base Class with One of Its Derived Classes  5-68
  Changing the Order (Position) of a Base Class  5-71
  Changing the Access Control of a Base Class  5-73
Complex Inheritance Operations  5-75
  Adding a Non-Leaf Base Class  5-75
  Deleting a Non-Leaf Base Class  5-77
  Making a Non-Persistent-Capable Class Persistent-Capable  5-78
  Making a Persistent-Capable Class Non-Persistent-Capable  5-79
Multiple Class Operations  5-79
  Splitting a Class into Two Associated Classes  5-79
  Splitting a Class So That One Class is Derived from the Other  5-81
  Merging Two Associated Classes  5-82
  Merging a Class with its Derived Class  5-83
Schema Evolution Example  5-84
Chapter 6  **Class Versioning**  6-1
  Basics  6-1
  Creating and Using New Class Versions  6-3
    Using #pragma ooclassname  6-4
  Class Versioning Hints  6-6
  Class Versioning Example  6-8
    Versioning the Classes  6-9
    Copying Objects  6-13

Chapter 7  **Data Model Tuning**  7-1
  Tuning Concepts  7-1
    Associations  7-1
    Associations and Basic Object Storage  7-2
  Tuning for Federated Database Size  7-3
    Use Inline Associations  7-3
    Store Data Efficiently  7-5
  Tuning for Speed  7-7
    Use Inline Associations  7-7

Appendix A  **Schema Evolution Basic Operation Summary**  A-1

Appendix B  **Tools**  B-1

Index  i-1
About This Book

This document describes the Objectivity/C++ Data Definition Language (DDL), the DDL processor, and principles of data modeling for Objectivity/DB C++ applications.

The tasks and tools used for database development in Windows, VMS, and MacOS environments are nearly identical to their UNIX equivalents. Because of these similarities, this document describes techniques for UNIX environments, and also presents any differences in usage, behavior, and naming that exist between the Windows, UNIX, VMS, and MacOS environments.

Audience

This document assumes that you are familiar with programming in C++.

Organization

◆ Chapters 1 through 4 cover DDL and the tasks involved in developing and processing data models.
◆ Chapter 5 describes schema evolution tasks that change an existing federated database schema.
◆ Chapter 6 discusses class versioning, which allows you to create multiple versions of a class in a schema.
◆ Chapter 7 provides data modeling and design guidelines that enable you to tune a federated database for size and speed.
◆ The appendices summarize schema evolution operations and provide reference descriptions of DDL processor tools.
Processing Data Models

This chapter presents data modeling principles for Objectivity/DB applications, and describes how to use the Data Definition Language (DDL) processor to process models you define.

Overview

Before you can create objects (class instances) in a federated database, you must provide Objectivity/DB with class definitions. Objectivity/DB uses this information, called the schema, to allocate and manage objects used in your database applications. You define Objectivity/DB classes using the DDL. (The details of the DDL are presented in the “Data Definition Language” chapter.)

The scope of an Objectivity/DB schema is its federated database. Usually, you will define a single common schema to be shared by all of the databases in a federated database. However, you can also create multiple schemas in a federated database (for more information, see the “Multiple Data Models” chapter).

Step 1. Preparing Class Declarations

To define and manipulate objects that reside in an Objectivity/DB federated database, you must identify which classes will be persistent, that is, which objects need to continue to exist after the application programs that define or manipulate them complete execution.

Most database applications will have many persistent-capable and non-persistent-capable classes. Persistent-capable classes are any classes derived from a persistent Objectivity/C++ base class (ooObj, ooContObj, and so on). You can create persistent objects directly from these classes.

Non-persistent-capable classes are classes that cannot be used to create persistent objects directly. However, objects of these classes are stored in the federated database when the classes are used as data member types or as base classes for persistent-capable classes.
Transient classes are non-persistent-capable classes used only to create transient objects, which are used only during the life of an application process. Objects created from these classes are not stored in the federated database.

**Specifying Persistent-Capable Classes**
To make a class persistent-capable, you make the class inherit from the Objectivity/DB class `ooObj`. You only need to modify base classes; derived classes that inherit from these base classes will automatically become persistent-capable.

---

**Example**

To make the following classes persistent-capable:

```cpp
class A : ... {
    ...;
}

class B : public A ... {
    ...;
}
```

add inheritance from `ooObj` to class `A`:

```cpp
class A : public ooObj ... {
    // inherits persistence from ooObj
    ...;
}
```

```cpp
class B : public A ... {
    // inherits persistence from A
    ...;
}
```

---
There are some types that cannot be stored within a persistent-capable class. However, it is usually easy to modify the class definition slightly to avoid using the type. See “Special Cases” on page 2-42 for suggestions on other ways to express these types.

The most common situation is an embedded pointer to an array or character string allocated by the application using the C++ `new` and `delete` functions. These fields must be changed when making the class persistent-capable because they point to storage allocated by these functions that is non-persistent-capable and will not be stored with the object in the federated database.

You can either change these pointers to embedded arrays within the object, or use the special persistent array (ooVArray) and string (ooVString) classes provided with Objectivity/DB. See Using Objectivity/C++ for information about how to convert pointer fields using these classes.

---

**Example**

Change:

```c
#include <string.h>
...
char *dueDate;
to:
ooVString dueDate;
```
Placing Class Definitions in Separate Files

Once you have decided which classes need to be persistent-capable, place their definitions in one or more files, along with any other definitions referred to or used within the persistent-capable classes such as:

- non-persistent-capable classes
- typedef statements
- extern declarations
- preprocessor directives

The files can have any base name, but the extension must be .ddl. If all of the classes to be persistent-capable are already in separate header (.h) files, the simplest way to create the .ddl files is to rename the .h files by changing their extensions to .ddl. For example, you would change the declarations for each class of a library information system application from Book.h, Patron.h, Loan.h, and Library.h to Book.ddl, Patron.ddl, Loan.ddl, and Library.ddl when preparing them for use with Objectivity/DB.

Step 2. Including DDL Processor Source Files

The Data Definition Language (DDL) processor creates a database schema from your .ddl files. When you process your .ddl file, the DDL processor generates three source files:

- A header file:
  `yourSchema.h`
  that contains your original class declarations with additional Objectivity/DB member functions for storing, retrieving, and modifying objects.

- A secondary header file:
  `yourSchema_ref.h`
  that contains ooRef and ooHandle declarations for each persistent-capable class in your `yourSchema.ddl` file. The `yourSchema.h` file automatically includes the second header file. You may also need to explicitly include the second header file before any forward declarations for persistent-capable classes in your application.
Creating a Schema

- A C++ implementation file,
  
  `yourSchema_ddl.c` for UNIX  
  `yourSchema_ddl.cpp` for Windows  
  `yourSchema_ddl.cp` for MacOs  
  `yourSchema_DDL.CXX` for VMS
  that implements Objectivity/DB member functions declared in the header file. This file is compiled and linked with your C++ application.

Creating a Schema

To create an Objectivity/DB schema, perform the following steps:

1. Use the `oonewfd` tool (see the “Federated Database Tasks” chapter in Objectivity/DB Administration). This tool also creates the boot file used by Objectivity/DB to locate the federated database file.

2. Using the DDL, define the classes for your data model. You can define your classes in one or more DDL schema files. Give each file a name of the form `classDefFile.ddl`. For more information on creating multiple schema files, see “Using Multiple Schema Files” on page 1-17.

   When designing your DDL schema files and application source code, follow the guidelines in “Using Files Generated by the C++ DDL Processor” on page 1-9 to handle any interdependencies between your DDL schema files.

3. Run the DDL processor once for each DDL schema file. This creates header files and a schema implementation file for each DDL schema file you process. See “Using the DDL Processor” on page 1-6.

4. Build your application by compiling your source code files and the schema implementation files created by the DDL processor, and link your compiled code with the appropriate libraries. For a complete description of the application development process, see the “Introduction to Objectivity/DB Programming” chapter in Using Objectivity/DB C++.

5. After creating a schema, you can change the definitions of any of the classes in your schema or add new class using the guidelines in “Modifying the Schema” on page 1-19 and the “Class Versioning” chapter.
Using the DDL Processor

The DDL processors for C++ application development. The tools are:
- `ooddlx.exe` for Windows
- `ooddlx` for UNIX and MacOS
- `ODB DDL` for VMS

Before running any of these processors, be sure that it is in your search path.

Running the DDL Processor

Invoke the DDL processor using the syntax described in the “Tools” appendix. When you run the C++ DDL processor, it performs the following actions.
1. Opens the federated database file specified by the boot file.
2. Opens the federated database for update.
3. Adds the classes defined in your DDL schema file to the federated database.
4. Closes the federated database.
5. Generates a schema:
   - Header file with a suffix `.h`
   - Reference header file with the suffix `_ref.h`
   - C++ implementation file.
     - For Windows this file is given the suffix `_ddl.cpp`
     - For UNIX, the suffix is `_ddl.C`
     - For MacOS, the suffix is `_ddl.cp`
     - For VMS, the suffix is `_DDL.CXX`

Example of Using the C++ DDL Processor

The following example shows how to run the `oonewfd` tool to create a boot file named `myExample` and a federated database file named `myExample.FDB` on a UNIX platform. It then shows how to run the DDL processor to load the DDL schema file named `mySchema.ddl` into the schema and create the files `mySchema.h`, `mySchema_ref.h`, and `mySchema_ddl.C`. 

1-6 Using Objectivity/C++ Data Definition Language
Example

> ls
mySchema.ddl
> oonewfd   -fdfilepath myExample.FDB \ 
  -lockserverhost machine95 myExample
> ls
myExample mySchema.ddl myExample.FDB
> ooddlx mySchema.ddl myExample
> ls
myExample
myExample.FDB
mySchema.ddl
mySchema.h
mySchema_ddl.C
mySchema_ref.h
Specifying the Boot File Path for VMS

When running the DDL processor in a VMS environment, you can include a full file specification or local file specification for the `bootFilePath` parameter.

The file specification information in the boot file path is used to locate the boot file for the federated database, which allows you to place the boot file in a directory other than the one running your application. For example, the following are valid boot file paths for the federated database `MYFD`:

- Single name, for example, `MYFD`
- Local file specification, for example, `MNT:[JOHN.DESIGN]MYFD`
- Full file specification, for example, `MYNODE::MNT:[JOHN.DESIGN]MYFD`

Assuming the boot file and the federated database file are in the current directory `MNT:[JOHN.DESIGN]`, and the boot file name and the system name of the federated database are both `MYFD`, the following are acceptable ways of running the DDL processor:

```
ODB DDL MYFD.DDL MYFD
ODB DDL MYFD.DDL MYFD;10
ODB DDL MYFD.DDL MYFD.;11
ODB DDL MYFD.DDL MNT:[JOHN.DESIGN]MYFD
```

In all above cases, Objectivity/DB will use `MYFD` as the system name of the federated database.

**Example**

For example, the VMS command below does the following:

1. Creates a federated database whose system name and boot file are `MYFD`.
2. Names two additional include directories.
3. Defines the `DEBUG` and `RELEASE2` preprocessor tokens.
4. Runs the DDL processor on the DDL schema file named `SCHEMA.DDL`. 
Using Files Generated by the C++ DDL Processor

This section describes the files generated by the DDL processor and how to use them in your schema and application source code.

Overview

The C++ DDL processor generates several files when it processes DDL schema (.ddl) files you provide. Specifically, it generates the following files for each DDL schema file, as shown in Figure 1-1:

- Schema header file with the suffix .h
- Schema reference header file (_ref.h)
- Schema C++ implementation file with the suffix _ddl.C. You must compile and link this file with your application code.
Including Generated Header Files

Depending on your application, you may need to include generated header files in either your application source files, your DDL schema files, or both. Refer to the following sections for detailed information about including these files.

C++ Application Files

If your C++ application code requires a persistent-capable class definition, include the schema header (.h) file generated from the DDL schema file that defines the class.
This example shows a `main.C` program that uses the class definition of `A`.

```cpp
// Schema file: a.ddl
class A : public ooObj {
    ...
};

// Application code: main.C
#include a.h    // Definition of A
{
    ...
    ooHandle(A) aH;
    aH = new() A;    // Requires definition of A
    ...
}
```

### DDL Schema Files

To include generated header files in DDL schema files, use the following guidelines:

- If a DDL schema file contains code that requires a persistent-capable class definition, include the schema header (.h) file generated from the DDL schema file that defines the class.
- However, if a DDL schema file has code that references persistent-capable classes defined in another DDL schema file, add the following pragma statement for each referenced class. The pragma allows the DDL processor to accept a class reference even if the file that contains the class definition has not yet been processed.
The syntax for the pragma is:

```cpp
#pragma ooclassref className <classDefFile_ref.h>
```

where

- `className`: Name of the class that the DDL processor should assume will be defined
- `classDefFile_ref.h`: Name of the schema reference header file where the persistent-capable class is defined. This file does not need to exist at the time the DDL processor scans the `#pragma` statement, since this file may be generated later by processing a different DDL schema file.

You must forward declare class `className` prior to the `#pragma` statement so it is recognized as a class name. Do not define the class `className` in the same DDL schema file.

Referencing a persistent-capable class `className` includes using the following classes in your DDL schema files:

- `ooRef(className)`
- `ooShortRef(className)`
- `ooHandle(className)`
- `ooItr(className)`

You must use either angle brackets `<>` or double quotes `""` when specifying the `classDefFile_ref.h` file. In its output, the DDL processor will preserve the delimiters you use.
For example, the following DDL schema files show class B (defined in the DDL schema file b.ddl) that inherits from class A (defined in the file a.ddl). In this case, class B requires the definition of class A, so it needs to include the file a.h.

```
// Schema file: a.ddl
class A : public ooObj {
    ...
};

// Schema file: b.ddl
#include a.h // Definition of A
class B : public A { // Requires definition of A
    ...
};
```

The following code contains class references. Without the necessary pragma statement, this code generates an error when processed by the DDL processor. In this example, A and B are defined in separate DDL schema files, and A has a bidirectional association to class B:

```
// DDL schema file: a.ddl
#include <b.h> // Generates an error
class A {
    ooRef(B) bA[] <-> aA; // One-to-many association
    ...
};

// DDL schema file: b.ddl
#include <a.h> // Generates an error
class B {
    ooRef(A) aA <-> bA[]; // Many-to-one association
    ...
};
```
Modifying the schema files above, the following class definitions correctly include the necessary generated header files. The changed lines of code are shown in bold font:

// DDL schema file: a.ddl
class B; // Declaration of B
#pragma ooClassref B <b_ref.h>
class A {
    ooRef(B) bA[] <-> aA; // One-to-many association
    ...
};

// DDL schema file: b.ddl
class A; // Declaration of A
#pragma ooClassref A <a_ref.h>
class B {
    ooRef(A) aA <-> bA[]; // Many-to-one association
    ...
};
Errors in Header File Inclusion Order

The inclusion order of header files generated by the DDL processor is important. The steps described above ensure that your header files are included in the proper order.

If you do not include these files as noted, the DDL processor or C++ compiler will issue an error message indicating that a persistent-capable class definition is missing. These messages are described below.

DDL Processor Inclusion Order Error Message
An example of the DDL processor error message is:

"missing definition of ooRef(className)"

For Windows Visual C++ environments, an incorrect inclusion order will generate an error message C2011 from Visual C++, indicating a class redefinition error.

To solve the problem, add one of the following statements to the appropriate DDL schema file:

- #include <classDefFile.h>  
  (See “DDL Schema Files” on page 1-11.)
- #pragma oo_classref className <classDefFile_ref.h>  
  (See “DDL Schema Files” on page 1-11.)
- template class className;  
  (See “Templates” on page 2-40.)

C++ Compiler Inclusion Order Error Message
An example of the C++ compiler error message is:

"Ignore_the_compilers_error_message_The_real_error_is_Missing_definition_of_ooRef<className>"

To solve the problem, add the following statement to your C++ application file:

- #include <classDefFile.h>  
  (See “C++ Application Files” on page 1-10.)
Using Preprocessor Directives

You can use preprocessor directives such as `#include` and `#ifdef` in your DDL schema files. However, because these files are processed by a standard C++ preprocessor before they are processed by the DDL processor, you should observe the following guidelines.

**Using `#include` Directives**

If a `#include` directive in a DDL schema file specifies another DDL schema file, no distinction is made between the declarations in the primary file and those in the included file. The header (.h) files and source (.c) file produced by the DDL processor contain the classes declared in the primary file and any included DDL schema files, along with their DDL-derived definitions.

If the file specified by a `#include` directive in a DDL schema file is not another DDL schema file, the header file produced by the DDL processor contains a corresponding `#include` directive so that subsequent compiles using the header file will also include the appropriate file.

**Using Other Directives**

The DDL processor preserves the C++ preprocessor directives `#define`, `#undef`, and `#pragma` (except for pragmas specific to the DDL). Other directives such as `#if`, `#else`, `#endif` are processed and then removed by the preprocessor.

If you wish to have conditional compilation directives appear in the resultant schema header file, place the directives in a header file, and include that header file in your DDL schema file.

**Automatic Inclusion of `oo.h`**

The DDL processor automatically inserts a `#include <oo.h>` at the beginning of each header file it generates. Therefore, you need not explicitly include `oo.h` or `oci.h` in your application source file, as long as you include one or more of the DDL-generated header files.
Using Multiple Schema Files

Objectivity/DB allows you to separate class definitions into one or more DDL schema files.

In traditional programming, you often write your type definitions in more than one header file. For example, you might want to group all types used by one module in a single header file, separate from the type definitions used by other modules in the system. This allows you to develop separate modules more independently of each other. Of course, often these modules are not entirely independent: one module may use types from another module. For example, a doubly-linked list module might inherit from a singly-linked-list module. So the doubly-linked list header file will include the singly-linked list header file.

Figure 1-2 shows an example using multiple DDL schema files. Each class is declared in a separate DDL schema file, and user-defined member functions are specified in separate source files. File a.ddl declares class A, the base class. File b.ddl declares class B, a class derived from class A. File c.ddl declares class C, a class derived from class B.

When processing DDL schema files, you must remember to process a file of a base class before you process the files of its derived classes.
Figure 1-2  Multiple DDL Schema Files with Dependent Classes
Modifying the Schema

In general, you should always try to determine exactly what classes your schema requires and use the DDL processor once for each DDL schema file to initialize the schema for a federated database. However, it is highly likely that you will eventually find it necessary to modify a schema.

When you modify a schema there are two primary goals:

- Changing the schema
- Preserving any data that may have been stored in the federated database using the old schema

If it becomes necessary to change the schema, there are several possible scenarios:

- The changes you wish to make are considered additions to the existing schema and do not require any special processing (see “Making Additions that Do Not Require Federated Database Changes” on page 1-20). If so, you need simply run the DDL processor again on the modified or added DDL schema files.
- You have not yet stored any useful data in the federated database (or you can easily regenerate any data that is stored there). In this case, the easiest and often fastest procedure is to remove the entire federated database and start from scratch.
- You have a large amount of useful data in the federated database. See the “Schema Evolution” chapter. If you want to use multiple versions of a given class, see the “Class Versioning” chapter.

⚠️ Warning

Whenever you modify the schema, you are generating new schema header and code files. You must recompile and link any applications that use these files. If you do not, runtime errors may occur.
Making Additions that Do Not Require Federated Database Changes

There are several changes you can make to your DDL schema files that do not require any changes in existing data (when none of the fundamental type descriptions are changed). When you make one of the changes listed here, you should:

1. Process any modified or added DDL schema file.
2. Incorporate the new header and source files in your application.
3. Rebuild your application.

Adding to a Class or Type

If you add to or alter any non-persistent type declarations (classes, typedef statements, enumerated types, and so on), whether in your DDL schema files or in any included header files, you will not need to change your federated database unless those changes affect the physical representations of your persistent type definitions.

Assume, for example that you add another enumeration value to an enumerated type. Even though that enumerated type is a data member of a persistent type, the change to the persistent type does not require a change in the federated database.

Adding a Class or Type

If you add any number of new type definitions to a schema, either by creating a new DDL schema file or altering an existing one, you will not need to change your federated database (since no instances of the new type or types can yet exist).

Adding or Changing Member Functions

You may add, alter, or delete any member functions (including operators and constructors) that are defined on any existing types (except those automatically generated for association) without requiring any changes in an existing federated database. When adding or modifying constructors, be aware of the constraints in “Constraints” on page 2-46.
Adding Associations

You may add (but not alter or delete) both unidirectional and bidirectional non-inline associations to existing types at any time. When adding bidirectional non-inline associations, be careful to add both ends of the association or it will not be usable. When adding unidirectional non-inline associations, be careful to define the class referred to by the association or it will also not be usable.
Modifying the Schema
Data Definition Language

This chapter explains the syntax for declaring classes when designing a database data model (schema). The Data Definition Language (DDL) defines the syntax you use to declare classes in Objectivity/DB. Except for a few minor syntactic and semantic extensions, the DDL is identical to ANSI C++ Version 3.0 class declaration syntax. The extensions allow you to customize the persistent behavior of a class.

For information on how to instantiate and manipulate persistent objects, refer to Using Objectivity/ C++.

Using the DDL

To add persistent-capable classes to the database schema, you must declare them in text files using the DDL. These files, called DDL schema files, resemble C++ header files but cannot be directly compiled with a C++ compiler. Each DDL schema file name must end with the extension .ddl.

The DDL processor uses the information in your DDL schema files to generate and store class information in the database schema. It also generates C++ schema header files (with the extension .h), which can be compiled by a standard C++ compiler, and which correspond (on a declaration-by-declaration basis) with your DDL schema files.

For C++ applications, the DDL processor also generates C++ schema source files (with the extension _ddl.C). These files contain system-defined functions, and must be compiled and linked with your application program.

Figure 2-1 shows the development flow for a typical C++ application. To simplify this example, only one DDL schema file is shown. However, the use of multiple DDL schema files is supported and is described in “Using Multiple Schema Files” on page 1-17.
Using Objectivity/C++ Data Definition Language

Figure 2-1  Application Development Flow
Class Declarations

DDL supports class hierarchies and declarations as defined by C++ with very few exceptions. User-defined persistent-capable classes inherit their persistent properties from one of several persistent base classes (see “Persistent-Capable Classes” on page 2-4).

Data Members

Each data member can be a primitive type (see “Primitive Types” on page 2-8), a fixed-size array, a structure or class, an aggregate, or a new type defined in terms of these basic types.

You can declare static data members in a persistent-capable class. But the storage for these data members is allocated outside of the object itself, so the value of a static member field is not persistent.

Member Functions and Functions

DDL allows virtually any C++ function declarations and definitions. All function declarators (heads) are checked for syntax, and legal declarators (with accompanying function bodies, if any) are passed unchanged to the output header file. No function information is stored in the schema.

You cannot use DDL-generated member functions or constants of a class or association in a global variable initialization in the DDL file in which they are defined.

Constructors and Destructors

You define constructors and destructors as in C++. “Constraints” on page 2-46 lists some constraints you may need to follow when defining your constructors and destructors.

The constraints described in “Constraints” on page 2-46 apply to base classes since C++ treats base classes as embedded objects. Objectivity/DB issues an error message if these constraints are violated.

Inheritance

The DDL supports single inheritance using the same syntax as C++, and multiple inheritance as described in the “Multiple Inheritance” chapter.
Persistent-Capable Classes

For an object to exist in the federated database, it must be an instance of a persistent-capable class. When a class is persistent-capable, you can allocate an instance of that class in persistent storage or in transient storage.

Defining a Persistent-Capable Class

Simply defining a class using the DDL does not make it a persistent-capable class; you must also specify it as being a derived class of one of several system-defined persistent-capable classes. These system-defined classes specify the various persistent properties a user-defined class may inherit.

System-Defined Persistent-Capable Classes

The system-defined persistent-capable classes form a class hierarchy illustrated in Figure 2-2, with class ooObj at the root. These persistent-capable classes are ooObj, ooContObj, ooDefaultContObj, ooDBObj, ooAOBj, and ooFDObj:

ooObj

The basic object class; each instance is a basic object. Its properties are:

- User-defined subclasses are allowed; instances of such subclasses are thus also persistent basic objects.
- Instances may be persistent.
- Instances may have associations.
- Instances may be versioned.
- Instances may have one or more scope names.

See ooObj (d_Persistent_Object) in the “Base Classes” appendix in Using Objectivity/C++.
ooContObj

The container class; each instance is a container. Its properties include all of the properties of ooObj, and the following additional properties:

- Instances of user-defined subclasses of ooContObj are also containers.
- Instances may contain other objects whose persistent properties are inherited from only ooObj.
- Instances may have system names.
- Instances may not be versioned.
- Instances may have associations.

See ooContObj in the “Base Classes” appendix in Using Objectivity/ C++.

ooDefaultContObj

The default container class; each instance is a default container. Every time a database is created, a default container is also created by Objectivity/DB. Its properties are:

- User-defined subclasses are not allowed.
- Instances may be persistent.
- Instances may not be versioned.
- Instances may contain other objects whose persistent properties are inherited from only ooObj.
- It is always given the system name _ooDefaultContObj.
- It inherits from ooContObj.

See ooDefaultContObj in the “Base Classes” appendix in Using Objectivity/ C++.

ooDBObj

The database class; each instance is a database. Its properties are:

- User-defined subclasses are not allowed.
- Instances may be persistent.
- Instances have system names.
- Instances may not be versioned.
- Instances may not have scope names.
Instances may contain other objects whose persistent properties are inherited from ooContObj.

See ooDBObj in the “Base Classes” appendix in Using Objectivity/ C++.

ooAPObj

The autonomous partition class; each instance is an autonomous partition. Its properties are:
- User-defined subclasses are not allowed.
- Instances may be persistent.
- Instances have system names. The scope of an autonomous partition system name is the same as for database system names. Therefore, database and autonomous partition system names must be unique.
- Instances may not be versioned.
- Instances may not have scope names.
- Instances may control other objects of class ooDBObj or whose persistent properties are inherited from ooContObj.

See ooAPObj in the “C++ Interface” appendix in Using Objectivity/ FTO and Objectivity/ DRO.

ooFDObj

The federated database class; each instance is a federated database. Its properties are:
- User-defined subclasses are not allowed.
- Instances may be persistent.
- Instances have system names.
- Instances may contain other objects of class ooDBObj.
- Instances may not be versioned.
- Instances may not have scope names.

See ooDBObj in the “Base Classes” appendix in Using Objectivity/ C++.
This example declares two persistent-capable classes, B and C. Both classes inherit their persistent properties from ooObj; B does so directly and C does so indirectly through its inheritance from B.

typedef int Coord;
Struct Point {
    Coord x;
    Coord y;
};
Struct Box {
    Point lowerLeft;
    Point upperRight;
};
class B: public ooObj{
    public:
        Box box;
        B();
        B(Box& box);
        virtual void draw();
};

class C: public B{
    public:
        ooVArray(Point) vertex;
        C();
        C(uint32 number, Box& box);
        virtual void draw();
};

Primitive Types

Primitive types are the building blocks from which complicated structures are formed. The DDL defines several primitive types of the following categories:

- Integer
- Floating point
- Character

Each DDL primitive type name specifies the exact size of the type to ensure portability of data among different machine architectures. In addition, DDL includes the standard C++ primitive types, allowing many existing C++ declarations to be incorporated without modification. C++ primitives are logically mapped to equivalent DDL primitives.
Integer

The type names shown in Table 2-1 specify whether the integers are signed or unsigned and are portable across all supported CPU architectures. Note that for reasons of portability, 8-bit signed integers are not supported.

Also, Objectivity/DB cannot support integers larger than 32 bits. Some compilers support larger integer sizes. For example, Objectivity/DB does not support the 64-bit long long integer type provided by the Sun4 and SPARCstation C++ compiler. It also does not support the 64-bit long integer type provided by the DEC Alpha compiler.

**Table 2-1: Type Names for Integers**

<table>
<thead>
<tr>
<th>Type Name</th>
<th>Size (bits)</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>uint8</td>
<td>8</td>
<td>0 to +255</td>
</tr>
<tr>
<td>int16</td>
<td>16</td>
<td>-32,768 to +32,767</td>
</tr>
<tr>
<td>uint16</td>
<td>16</td>
<td>0 to +65,535</td>
</tr>
<tr>
<td>int32</td>
<td>32</td>
<td>-2,147,483,647 to +2,147,483,646</td>
</tr>
<tr>
<td>uint32</td>
<td>32</td>
<td>0 to +4,294,967,295</td>
</tr>
</tbody>
</table>
Floating Point

Objectivity/DB stores floating point numbers as shown in Table 2-2 in the native format of the CPU architecture on which they are instantiated or modified. Thus, the precision and range supported may vary from architecture to architecture. Objectivity/DB automatically converts floating point formats between architectures in heterogeneous environments.

Also, Objectivity/DB cannot support floating point numbers larger than 64 bits. Some compilers support larger floating point sizes. For example, Objectivity/DB does not support the 128-bit long double integer type provided by the Sun4 and SPARCstation C++ compiler, and the HP C++ compiler.

<table>
<thead>
<tr>
<th>Type Name</th>
<th>Size (bits)</th>
<th>Range</th>
<th>Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>float32</td>
<td>32</td>
<td></td>
<td>Architecture dependent</td>
</tr>
<tr>
<td>float64</td>
<td>64</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Character

The character data type char is shown in Table 2-3.

<table>
<thead>
<tr>
<th>Type Name</th>
<th>Size (bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>8</td>
</tr>
</tbody>
</table>
C++ Primitive Types

We strongly recommend that you use DDL primitive types for all user declarations. However, the syntax does permit the use of C++ primitive types for compatibility with existing declarations, as shown in Table 2-4.

⚠️ Warning

C++ primitive types are mapped by the DDL processor to equivalent DDL primitive types as shown in Table 2-4. These mappings are compatible with most C++ compilers on 32-bit CPU architectures. If you wish to use existing C++ primitive types, you should make sure that the primitive-type mappings specified in Table 2-4 allow portability across all of your target computing environments.

<table>
<thead>
<tr>
<th>C++ Type</th>
<th>DDL Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>char</td>
</tr>
<tr>
<td>unsigned char</td>
<td>uint8</td>
</tr>
<tr>
<td>short</td>
<td>int16</td>
</tr>
<tr>
<td>unsigned short</td>
<td>uint16</td>
</tr>
<tr>
<td>int</td>
<td>int32</td>
</tr>
<tr>
<td>unsigned int</td>
<td>uint32</td>
</tr>
<tr>
<td>long</td>
<td>int32 (not available on DEC Alpha)</td>
</tr>
<tr>
<td>unsigned long</td>
<td>uint32 (not available on DEC Alpha)</td>
</tr>
<tr>
<td>float</td>
<td>float32</td>
</tr>
<tr>
<td>double</td>
<td>float64</td>
</tr>
</tbody>
</table>
Aggregate Data Types

The DDL provides support for the following aggregate types:

- Fixed-size arrays (contiguous storage)
- Variable-size arrays (contiguous storage)

Fixed-size arrays are an integral part of the DDL, while variable-size array support is provided in the form of predefined parameterized classes.

Variable-Size Arrays (VArrays)

Your application can use special Objectivity/DB-defined arrays known as variable-size arrays (VArrays). These arrays are very similar to standard C++ arrays, except that they can change in size at runtime. See the “Using Variable-Size Arrays” chapter in Using Objectivity/ C++ for information on how to use VArrays. Objectivity/DB supports VArrays through two parameterized non-persistent-capable classes, ooVArray and ooTVArray. VArrays of class ooVArray can be either persistent or temporary. VArrays of class ooTVArray are always temporary.

To make a persistent VArray of a parameterized class of ooVArray, you must embed it within a persistent object. To do this, you must first declare the VArray within the persistent object's class, and then create the persistent object. To declare the VArray within a persistent-capable class, follow these steps:

1. Use the C++ declare macro (before the persistent-capable class declaration). This macro has the following syntax:
   ```cpp
   declare(ooVArray, varrayClass);
   ```
2. Declare the VArray as a data member using the following syntax:
   ```cpp
   ooVArray(varrayClass) arrayName;
   ```
   where
   ```
   varrayClass: Class of the elements in the VArray
   arrayName: Name of the VArray
   ```

The Objectivity/DB implementation of a VArray of class ooVArray reads the entire VArray into a contiguously allocated block of virtual memory. This allows a VArray to be accessed as a block of raw data. For example, the type ooVArray(uint8) is simply a contiguous array of bytes and could be used to store a digitized image.
Before you can access a VArray of a particular class, you must use the C++ implement macro to generate code needed for the parameterized class in one and only one of your source files. You must use a corresponding implement macro for each VArray declare macro you use. For more information, see the “Using Variable-Size Arrays” chapter of Using Objectivity/ C++.

Guidelines for Using VArrays

You may define VArrays of almost anything, subject to the following guidelines. These guidelines apply to VArrays of struct, class, or typedef types. The DDL processor detects violations of these guidelines and issues appropriate error messages.

◆ ooTVArray VArrays are not persistent, and cannot be data members of persistent objects. For more information about VArrays of this class, see the “Using Variable-Size Arrays” chapter of Using Objectivity/ C++.

◆ For C++ compilers that support templates, ooVArray differentiates its parameters by type rather than by name. Therefore, if x is a type and y is a type defined as x (typedef x y;), it is illegal to use the declare and implement macros on both ooVArray(x) and ooVArray(y). You can still use both ooVArray(x) and ooVArray(y), but they are different names for the same type. For example, Objectivity/DB defines uint8 and ooBoolean to the same type, so a declaration of ooVArray(uint8) conflicts with ooVArray(ooBoolean).

◆ Objectivity/DB declares and implements ooVArray(char) and ooTVArray(char), so you should not declare or implement these types. Objectivity/DB also declares and implements ooVArray(uint32) and ooTVArray(uint32), so you should not declare or implement this type. Alternatively, you can #ifndef protect any primitive type ooVArrays you declare. For example, the following code shows two examples of how to declare a primitive type ooVArray:

```
// declare(ooVArray,uint32). #ifndef protection needed.
#ifndef OO_VARRAY_UINT32
#define OO_VARRAY_UINT32
declare(ooVArray,uint32)
#endif
```
// declare(ooVArray,int16). #ifndef protection may be needed
// in the future.
#ifndef OO_VARRAY_INT16
#define OO_VARRAY_INT16
declare(ooVArray,int16)
#endif

◆ If you are using templates, you should not declare or implement
ooVArray(int8) or ooTVArray(int8) since these types are expanded into
ooVArray(char) or ooTVArray(char) by the DDL processor. You also may
encounter a multiple declaration error if you have type defined a
parameterized type of a VArray. For example, suppose you have the following
typedef:
typedef float64 salaryType;
Without templates, the following declarations are legal since name mangling
can distinguish the types float64 and salaryType.
#ifndef OO_VARRAY_FLOAT_64
#define OO_VARRAY_FLOAT_64
declare(ooVArray,float64)
#endif
declare(ooVArray,salaryType)
With templates, declare(ooVArray,float64) and
declare(ooVArray,salaryType) are equivalent, making the second
declaration redundant and therefore illegal.
◆ If you use a VArray as a member of a persistent-capable class, you cannot
directly or indirectly embed unions, bit fields, VArrays, persistent objects, or
member pointers as elements of the VArray.
◆ Element types must have a default constructor (a constructor that can take no
arguments). This is similar to the constraint of C++ that you can only declare
regular arrays of objects that have default constructors.
The following code declares two VArrays. `intVarray` is a VArray of integers and `pointVarray` is a VArray of type `Point`.

```c
struct Point{
    int32 xCoord;
    int32 yCoord;
};

declare(ooVArray,uint16);
declare(ooVArray,Point);

class myClass : public ooObj {
    ...
    ooVArray(Point) pointVarray;
    ooVArray(uint16) intVarray;
    ...
};
```

---
Associations

All classes that inherit persistent properties from ooObj or ooContObj may declare association links. You can use associations to model relationships between objects. Association links are inherited like any other class attribute.

Refer to “Associations” on page 1-17 in Using Objectivity/ C++ for an introductory explanation of association concepts. See “Association Links” on page 2-24 for an explanation of how associations are stored.

Refer to “Composite Objects” on page 2-29, “Association Links and Object Copying” on page 2-31, and “Association Links and Versioning” on page 2-37 for information on association properties. Also see “Association Links” on page 2-24 for information on another type of association link that may be more efficient for your application.

Declaring association links makes it possible to associate objects. For every association link declared for a class, the DDL processor generates several member functions for the class. You use these member functions to actually access links and create associations between objects. See the “Using Associations” chapter of Using Objectivity/ C++ for detailed information about these methods and using associations.

Notation for Class Association Links

Figure 2-3 shows Objectivity/DB notation for association links between classes. In this notation, boxes represent classes. Straight lines between classes represent association links. Association link names are placed above or below the association link next to the class in which they are defined. This notation represents cardinality in two ways by:

- Placing a 1, n, m, or a fixed number beside each class
- Adding closed brackets, [ ], at the end of an association link name representing a to-many association. If the link is declared as a to-one association, the closed brackets are not used. You will use the same conventions when creating a database schema with DDL.
Figure 2-3  Class Association Links
**Bidirectional Association Links**

You can establish a bidirectional association between two objects if each object's class contains an association link that references the other class. Bidirectional associations allow you to visit either of the two objects from the other, and also provide Objectivity/DB with enough information to maintain referential integrity.

We recommend that you use bidirectional rather than unidirectional association links. See “Unidirectional Association Links” on page 2-20 for a discussion of unidirectional association links.

Bidirectional association links may be defined anywhere within a class declaration using the following syntax. For details about inline association, see “Association Links” on page 2-24:

- one-to-one
  ```
  [inline] ooRef(class) linkName <-> inverseLinkName
  [: bSpec {, bSpec}];
  ```
- one-to-many
  ```
  [inline] ooRef(class) linkName[] <-> inverseLinkName
  [: bSpec {, bSpec}];
  ```
- many-to-one
  ```
  [inline] ooRef(class) linkName <-> inverseLinkName[]
  [: bSpec {, bSpec}];
  ```
- many-to-many
  ```
  [inline] ooRef(class) linkName[] <-> inverseLinkName[]
  [: bSpec {, bSpec}];
  ```

where

- **class** Name of the target class
- **linkName** Local name of the association link
- **inverseLinkName** Name of the corresponding association link defined in the target class
- **bSpec** Optional behavior specifier
Example

The classes Cont1, A, and B exist in a hypothetical federated database. This example models:

- Many instances of class B associated with a single instance of class A
- Many instances of class A associated with a single instance of class Cont1

The schema diagram for these relationships is shown in Figure 2-4.

```
Figure 2-4    Bidirectional Many-to-One Association Links

The following code declares the bidirectional association link declarations for these classes that support the required relationships. Notice how the appropriate association link names must cross-reference each other. The toB association link in class A is one-to-many, while the corresponding association link, toA in class B, is many-to-one.

class Cont1: public ooContObj {
  public:
    ooRef(A) toA[] <-> toCont1;
};

class A : public ooObj {
  public:
    ooRef(Cont1) toCont1 <-> toA[];
    ooRef(B) toB[] <-> toA;
};
```
class B : public ooObj {
public:
    ooRef(A) toA <-> toB[];
};

Unidirectional Association Links

Objectivity/DB also supports unidirectional associations. In a unidirectional
association only one object’s class declares an association link. A unidirectional
association limits the visibility of classes to one another. Both classes in a
bidirectional association are aware of one another since a bidirectional association
link is declared for both classes. With a unidirectional association, only one class is
aware of the other. A bidirectional association occupies twice as much storage as a
unidirectional association, so unidirectional associations provide better usage of
storage space than bidirectional associations.

The syntax for declaring a one-to-one unidirectional association is as follows. For
details about inline association, see “Association Links” on page 2-24:

❑ [inline] ooRef(class) linkName : bSpec {, bSpec};

The syntax for declaring a one-to-many unidirectional association is as follows:

❑ [inline] ooRef(class) linkName[] : bSpec {, bSpec};

where

class Name of the target class
linkName Name of the association link
bSpec Optional behavior specifier. A behavioral specifier is not required for
inline to-one associations.

Unidirectional associations require at least one explicit behavioral specifier to
distinguish the following two ambiguous syntax forms:

ooRef(class) linkName;        // Ambiguous – Either a data member
                             // or a unidirectional association.
                             // Considered as a data member.

ooRef(class) linkName[];      // Considered an error.
To clarify these syntax forms, add a default behavior specifier, such as `copy(delete)`, as follows:

```plaintext
ooRef(class) linkName : copy(delete);
ooRef(class) linkName[] : copy(delete);
```

---

**Warning**

Unidirectional associations provide no referential integrity checking. If used improperly, unidirectional associations can lead to dangling references that may leave your federated database in an inconsistent state. Use unidirectional links only if you are certain that you need them, and that you can independently maintain 100% referential integrity, even if a program error occurs.

---

**Example**

Using the example used in “Bidirectional Association Links” on page 2-18, assume the following additional relationships:

- Instances of B do not need a direct link back to the instances of A (if any) to which they are associated.
- Instances of A do not need a direct link back to the instances of Cont1 (if any) to which they are associated.

Because of the one-way nature of these associations, the schema can now be drawn as shown in Figure 2-5.

![Figure 2-5](image-url)

**Figure 2-5** Unidirectional Many-to-One Association Links
Using unidirectional associations, the required declarations are now as follows. Class B no longer requires an association link declaration, and A and Cont1 each require only a single association link declaration. Although `copy(delete)` is the default behavior, `copy(delete)` is used explicitly in this example to distinguish the association from a normal data member.

```cpp
class Cont1: public ooContObj {
    public:
        ...
        ooRef(A) toA[] : copy(delete);
};

class A : public ooObj {
    public:
        ...
        ooRef(B) toB[] : copy(delete);
};

class B : public ooObj {
    public:
        ...
};
```

**N-ary Associations**

Although n-ary associations (associations between three or more objects) are not directly supported in the DDL, it is possible to achieve approximately the same effect by using an intermediate object with multiple binary association link declarations.
Example

The classes **Lens**, **Reflector**, and **Bulb** represent various components of the headlight assembly used within the schema of an MCAD federated database. Objectivity/DB does not directly support a ternary association, but you can indicate that one instance of each class may be associated together using the following declarations:

```cpp
class Reflector : public ooObj{
public:
    ...
    ooRef(Headlight) headlight <-> reflector;
};

class Lens : public ooObj{
public:
    ...
    ooRef(Headlight) headlight <-> lens;
};

class Bulb : public ooObj{
public:
    ...
    ooRef(Headlight) headlight <-> bulb;
};

class Headlight : public ooObj{
public:
    ooRef(Reflector) reflector <-> headlight;
    ooRef(Lens) lens <-> headlight;
    ooRef(Bulb) bulb <-> headlight;
};
```
Association Links

There are two kinds of association links: non-inline and inline.

Non-Inline Association Links

Associations set on association links declared as described in “Bidirectional Association Links” on page 2-18, “Unidirectional Association Links” on page 2-20, and “N-ary Associations” on page 2-22 are stored in system default association arrays. Each object with associations has a system default association array in which all of these non-inline associations are stored. For example, in Figure 2-6, the unidirectional associations of object A1 with objects C1, C2, and B are stored in the system default association array for object A1.

Each association is identified by the association link name (an identifier, not a string) and the object identifier (OID) of the object being associated. Twelve bytes are needed to store each association, four bytes for the name of the link, and eight bytes for the OID. To trace a particular association on object A1, an application must traverse the associations in the system default association array until it locates the desired association.

Long and Short Inline Association Links

You can also define inline associations. These associations are similar to non-inline associations except that they are stored differently by Objectivity/DB. To-one inline associations are embedded as fields of an object, while to-many inline associations are placed in their own array instead of the default association array. Inline associations provide faster access to the OIDs of associated objects than non-inline associations, but there are tradeoffs to consider. See the “Data Model Tuning” chapter for information on when inline associations are likely to improve your application’s runtime performance.

There are two different types of inline association links. A long inline association link uses a long OID to refer to the associated object; a short inline association link uses a short OID to refer to the associated object. See “Using the ooRef and ooShortRef Classes” on page 4-3 in Using Objectivity/C++ for information on long (ooRef) and short (ooShortRef) object references. A short inline association link is used in the same way as a long inline association link, but uses less storage space (only four bytes, rather than eight) to maintain the association, resulting in better runtime performance. However, you may only set associations to or between objects in the same container on a short inline association link.
For example, see the associations on the link toG in Figure 2-6. Short association links must be inline and require the inline keyword. Non-inline short association links are not supported.

```c
class A : public ooObj {
    public:
        ooRef(B) toB : copy(delete);
        ooRef(C) toC [] : copy(delete);
        inline ooRef(D) toD;
        inline ooRef(E) toE [];
        inline ooShortRef(F) toF;
        inline ooShortRef(G) toG [];
    ...
};
```

Figure 2-6 Storage of Associations on Object A1 of Class A
Accessing Inline Association Links

The member functions for accessing inline association links are generated in a C++ header file when you run the DDL processor on a DDL schema file. These member functions are the same as those for accessing non-inline association links, as described in the “Using Associations” chapter in Using Objectivity/ C++.

Defining an Inline Association Link

When you define an association link on a composite object, you can specify that you want the association to be inline by using the inline keyword.

You can also specify ooShortRef instead of ooRef. See “Using the ooRef and ooShortRef Classes” on page 4-3 in Using Objectivity/ C++ for information on choosing between long and short object references.

For a bidirectional inline association, you must declare both association links as inline. That is, if one end is an inline association link, the other end must also be an inline association link. And, if one end is a short inline association link, the other end must also be a short inline association link.

Examples

The following DDL schema declares classes A and B, which have bidirectional short inline association links between them.

class A : public ooObj {
public:
  ...
  inline ooShortRef(B) bs[] <-> a; //one-to-many
};

class B : public ooObj {
public:
  ...
  inline ooShortRef(A) a <-> bs[]; //many-to-one
};
The DDL schema in the following example declares the following association links:

- **One-to-one bidirectional inline association links** between class \( A \) and class \( B \), which are called \( \text{toB} \) at the \( A \) end and \( \text{toA} \) at the \( B \) end
- **Bidirectional short inline association links** between class \( A \) and class \( C \), which are called \( \text{toCs} \) at the \( A \) end and \( \text{toA} \) at the \( C \) end
- **Bidirectional short inline association links** between class \( A \) and class \( D \), which are called \( \text{toDs} \) at the \( A \) end and \( \text{toAs} \) at the \( D \) end
- **A unidirectional short inline association link** between class \( A \) and class \( E \), which is called \( \text{toEs} \)
- **Bidirectional non-inline association links** between class \( A \) and class \( F \), which are called \( \text{toFs} \) at the \( A \) end and \( \text{toA} \) at the \( F \) end
- **A unidirectional non-inline association link** between class \( A \) and class \( G \), which is called \( \text{toG} \)

```cpp
class A : public ooObj {
public:
    ...
    // inline association links
    inline ooRef(B) toB <-> toA;
    inline ooShortRef(C) toCs[] <-> toA;
    inline ooShortRef(D) toDs[] <-> toAs[];
    inline ooShortRef(E) toEs[];

    // non-inline association links
    ooRef(F) toFs[] <-> toA;
    ooRef(G) toG : copy(delete);
};

class B : public ooObj {
public:
    ...
    inline ooRef(A) toA <-> toB;
};
```
class C : public ooObj {
public:
    ...  
    inline ooShortRef(A) toA <-> toCs[];
};

class D : public ooObj {
public:
    ...  
    inline ooShortRef(A) toAs[] <-> toDs[];
};

class E : public ooObj {
public:
    ...  
    inline ooShortRef(B) toB;
};

class F : public ooObj {
public:
    ...  
    ooRef(A) toA <-> toFs[];
};

class G : public ooObj {
public:
    ...  
    ooRef(C) toCs[] : copy(delete);
};
Composite Objects

You can configure association links to allow certain database operations to propagate to other objects along the association link. A collective group of objects linked together by associations is known as a composite object. For an introductory discussion of composite objects and propagation, see the "Introduction to Objectivity/DB Programming" chapter in Using Objectivity/ C++.

Behavior Specifiers

When you declare an association link for a composite object, you can use behavior specifiers to specify how you want the association to the object handled during delete, lock, copy, and versioning operations.

When a propagating operation is applied to an object through the appropriate function or member function, Objectivity/DB first identifies all objects that are affected (by identifying associations based on links that are declared to have propagation). It then applies the operation to all affected objects in a single atomic operation. This guarantees that a propagating operation will eventually terminate, even though the propagation graph may contain cycles. The propagation only occurs when you invoke the corresponding member function for that operation on an object in the composite object. For example, to propagate a lock operation, you need to call the C++ lock member function on the object.

Delete Propagation

You specify delete propagation behavior for an association link using the following behavior specifier syntax:

- assocLink : delete(propagate);

where

assocLink Name of an association link

The default propagation behavior is to not propagate delete operations.
Lock Propagation

You specify lock propagation behavior for an association link using the following behavior specifier syntax:

assocLink : lock(propagate);

where

assocLink Name of an association link

The default propagation behavior is to not propagate lock operations.

Example

This example shows two association links that reference each other. Assuming that an association was established between an object of class Cont1 and class A, the following are true:

◆ Locking the Cont1 object also locks the A object.
◆ Deleting the Cont1 object also deletes the A object.
◆ Locking or deleting the A object has no effect on the Cont1 object since no propagation behavior was specified on the association link to Cont1 in class A.

class Cont1 : public ooContObj {
public:
    ...
    ooRef(A) toA[] <-> toCont1 : lock(propagate),
    delete(propagate);
}

class A : public ooObj {
public:
    ...
    ooRef(Cont1) toCont1 <-> toA[];
};
Association Links and Object Copying

An application can copy a basic object using the `copy` member function, as described in the “Copying and Moving Basic Objects” chapter of Using Objectivity/C++. When you declare an association link for a composite object, you can specify what to do with the association to the old object handled during an object copy by using the `copy` keyword.

Figure 2-7 shows the effect each of these keywords has on an association to the old object. The copy behavior specifier causes shallow copy only; that is, it does not cause propagation of the copy along the association links.

**Figure 2-7  Copying Behavior of Associations**

- **Copy**: Copy the association when a new object is created
- **Move**: Move the association when a new object is created
- **Delete**: Delete the association when a new object is created

Key to Symbols

- = Object to be copied
- = Associated object A
- = Association
The syntax for declaring an association link with a copy behavior specifier is as follows:

- `assocLink : copy(copyOp);`

where

- `assocLink`: Name of an association link
- `copyOp`: One of the following keywords:
  - `copy`: Copy the association from the old object to the new object. It can be specified only on the to-one end of a many-to-one association link or on the to-many end of a many-to-many association link.
  - `move`: Move the association from the old object to the new object.
  - `delete`: Delete the association in the new object. The association remains in the old object. This is the default.

Specifying a copy behavior for one association link of a bidirectional association affects the other link of that bidirectional association. However, you can, for example, specify `delete` for one of the association links and `move` for the other. When an object of one of the linked classes is copied, the association link from that class to the other class is handled as requested by the copy behavior specifier for the class.

The examples that follow all refer to the bidirectional association links shown in Figure 2-8.

![Figure 2-8 Bidirectional Association Links between Class A and Class B](image)
The DDL schema file in this example declares the association links shown in Figure 2-8:

- A one-to-many bidirectional association link between class A and class B that is to be moved to the new object when an object of class A is copied
- A many-to-one bidirectional association link between class B and class A that is to be copied to the new object when an object of class B is copied

```c++
class A : public ooObj {
    ...  
    ooRef(B) toB[] <-> toA : copy(move);
};

class B : public ooObj {
    ...
    ooRef(A) toA <-> toB[] : copy(copy);
};
```

The following code sets associations on both links resulting in the bidirectional association shown in Figure 2-9. Then, it first copies an object of class A and later copies an object of class B.

![Bidirectional Associations between Objects A1 and B1](image)

Figure 2-9  Bidirectional Associations between Objects A₁ and B₁
ooHandle(A) aH, aaH;
ooHandle(B) bH, bbH;
...
//aH and bH are valid handles
...
aH-> add_toB(bH);
...
aH.copy(aH, aaH);
...
bH.copy(bH, bbH);

The resulting associations are shown in Figure 2-10. When object A1 is copied, the association link from class A to class B is moved, associating object A2 with object B1. When object B1 is copied, the association link from class B to class A is copied, associating object B2 with object A2.

Figure 2-10  Associations after Copying A1 with move and B1 with copy
The following DDL schema file declares the following association links, which are shown in Figure 2-8:

- A one-to-many bidirectional association link between class A and class B that is to be deleted when an object of class A is copied
- A many-to-one bidirectional association link between class B and class A that is to be moved to the new object when an object of class B is copied

```cpp
class A : public ooObj {
    ...
    ooRef(B) toB[] <-> toA: copy(delete);
};

class B : public ooObj {
    ...
    ooRef(A) toA <-> toB[]: copy(move);
};
```

The following code sets associations on both association links resulting in the bidirectional association shown in Figure 2-9. Then, it first copies an object of class A and later copies an object of class B.

The resulting association is shown in Figure 2-11. When object A1 is copied, the association link from class A to class B is deleted, giving object A2 no associations. When object B1 is copied, the association link from class B to class A is moved, associating object B2 with object A1.
ooHandle(A) aH, aaH;
ooHandle(B) bH, bbH;
...
//aH and bH are valid handles
...
bH->set_toA(ah);
...
aH.copy(aH, aaH);
...
bH.copy(bH, bbH);

Figure 2-11  Associations after Copying A1 with drop and B1 with move
**Association Links and Versioning**

When an object is versioned (that is, when a new version of the object is created), you can specify how you want the association to the old version of the object handled. Refer to Using Objectivity/ C++ for a more detailed discussion of versioning.

You specify the versioning behavior for a link with the following behavior specifier:

- assocLink : version(versOp);

where

- assocLink: Name of an association link
- versOp: One of the following keywords:
  - copy: Copy the association from the old version to the new version. It can be specified only on the to-one end of a many-to-one association link or on the to-many end of a many-to-many association link.
  - move: Move the association from the old version to the new version.
  - delete: Delete the association in the new version. The association remains on the old object. This is the default.

See “Association Links and Object Copying” on page 2-31 for more information on copy, move, and delete.

---

**Example**

This example shows the association link declarations to support a one-to-many association between class Cont1 and class A, and a one-to-one association between class Cont1 and class B. In this example, we assume that an association is established between:

- An object of class Cont1 and class A
- The same object of class Cont1 and an object of class B

The example code has the following effects on these associations:

- Versioning the Cont1 object associates the new version to the B object, and removes the association from the old version of the Cont1 object to the B object. The new version of the Cont1 object would not be associated to the A object.
Versioning the \texttt{A} object associates the new version to the \texttt{Cont1} object and maintains association from the old version to the \texttt{Cont1} object. Versioning the \texttt{B} object will not associate the new version to the \texttt{Cont1} object.

class \texttt{Cont1}: public \texttt{ooContObj} {
public:
    ...
    ooRef(A) toA[] <-> toCont1 : version(delete);
    ooRef(B) toB <-> toCont1 : version(move);
};

class \texttt{A}: public \texttt{ooObj} {
public:
    ...
    ooRef(Cont1) toCont1 <-> toA[] : version(copy)
};

class \texttt{B}: public \texttt{ooObj} {
public:
    ...
    ooRef(Cont1) toCont1 <-> toB : version(delete);
};

\section*{Combining Copy, Lock, Delete, and Versioning}

You may combine the copy, lock, delete, and versioning behavior specifiers for an association link by separating them with a comma.

\begin{itemize}
\item assocLink : bSpec, bSpec;
\end{itemize}

where

- \texttt{assocLink} Name of an association link
- \texttt{bSpec} A particular behavior specifier
Example

The following declaration of association link toA[] specifies the following:

- Both the lock and the delete operations propagate to associated objects.
- If the object is versioned, any association on the link is dropped.
- If the object is copied, any association on the link is moved to the new object.

class Cont1 : public ooContObj {
public:
    ...
    ooRef(A) toA[] <-> toCont1 : lock(propagate),
                   delete(propagate),
                   version(delete),
                   copy(move);
};

class A : public ooObj {
public:
    ...
    ooRef(Cont1) toCont1 <-> toA[];
};
Templates

You can use class templates in DDL schema files. The DDL processor allows you to define persistent-capable class templates as a part of your database schema.

To create a persistent-capable class template, define the template in a DDL schema file using the normal C++ template syntax. Just like a normal DDL class declaration, the template will get its persistence by having oooObj or oooContObj as its left-most ancestor class.

You must include all the oodd1-generated _ref.h files that contain oor e classes for user-defined instantiation classes before including the oodd1-generated .h files that define the template for the user-defined classes.

You cannot use commas to separate template parameters for persistent-capable class templates with two or more template parameters. To workaround this problem, either typedef the template or #define the comma separator.

Instead of:

rorr(x, y));

Use:

typedef foo<x, y> foo_xy;
orRef(foo_xy);

Or, use:

#define COMMA
rorr(x COMMA y));
Consider the following template class:

template<class T>
class myClassTemplate : public ooObj {
  public:
    T field;
};

You must explicitly instantiate all classes that are to be persistent inside a DDL schema file. For the above example, the syntax to do this is as follows:

template class myClassTemplate<float64>;

where `template` and `class` are required C++ keywords. You need to do this once for every class. The template must be visible during the above instantiation.

Consider a template called a Segment. Every segment will have an association to two point objects. The points are can be described in either cartesian or polar coordinates.

// Declare the template inside the DDL schema file
template <class T>
class Segment : public ooObj {
  public:
    ooRef(T) toPointA : copy(delete);
    ooRef(T) toPointB : copy(delete);
};

// Instantiate the segment template for both the Polar and Cartesian classes inside the DDL schema file.
template class Segment<CartesianPoint>;
template class Segment<PolarPoint>;
Special Cases

Certain C++ constructs cannot be processed by the Objectivity/DB DDL processor when used as part of a persistent-capable class or class template definition. These special cases include:

- Unions
- Bit fields
- Classes that inherit from a virtual base class
- Member pointers
- Persistent-capable classes nested within any class

Using any of these constructs in the definition of a persistent-capable class or class template will cause the DDL processor to generate an error and not update the schema. However, there are workarounds for most of these constructs as described in the following sections.

Unions and Bit Fields

Unions and bit fields are not supported in Objectivity/DB because they are not portable between heterogeneous architectures. A persistent-capable class may not incorporate unions or bit fields through inheritance, embedding, or VArrays.

Using Unions

Unions do not contain information specifying what branch of the union has been used. Without this information, it is not possible to determine what conversion operation should be performed when converting the object that contains the union from one architecture to another. The union branch types could have very different conversion operations. For example, floating point numbers are not converted the same way as integers.

If you want to use the same syntax for accessing fields, and storage is not an issue, you can substitute struct for union in your expression. If storage is an issue, and you want an expression that is more closely related to your logical conceptualizing, you can substitute subclasses for unions.
This example shows a C++ union expression and an equivalent expression using subclasses.

class property : public ooObj {
public:
    char name[32]
    int8 propType;
    union {
        int16 integer; // Not supported
        float32 real;
    } value;
};

// Supported alternative to unions
class propertyInteger : property {
public:
    int16 integer;
};

class propertyReal : property {
public:
    float32 real;
};
Using Bit Fields

C++ language definitions do not guarantee a particular order of bit fields within an integer. Thus, you cannot use bit map fields because they would be non-portable between heterogeneous architectures.

You can achieve the same functionality of bit fields by using one of two techniques. You can either separate the bit field into integer components or use `#define` statements to specify the packing of a common integer variable. Both of these techniques are shown in the following example.

---

Examples

This example shows two techniques to achieve the functionality of bit fields. The first code fragment illustrates a typical use of bit fields. The second fragment shows how to achieve the same results by separating the bit field into integer components. The third fragment uses `#define` statements to specify the packing of a common integer variable.

class picture : public ooObj {
public:
    int32 image1: 4; // Not supported
    int32 image2: 28; // Not supported
};

// Supported alternative to bit fields
class picture : public ooObj {
public:
    int32 image1;
    int32 image2;
};

// Supported alternative to bit fields
#define image1 (composite&&0XF0000000) >> 28;
#define image2 (composite&&0X0FFFFFFF);

2-44 Using Objectivity/C++ Data Definition Language
class picture public ooObj { public: int32 composite; }; 

Virtual Base Classes

You cannot use classes that have virtual base classes in any persistent way. The DDL processor generates an error if a persistent-capable class does any of the following:

◆ Inherits from a class that has a virtual base class
◆ Uses a class that has a virtual base class as an embedded member
◆ Declares a VArray whose base class has a virtual base class

Nested Classes

You cannot nest a persistent-capable class within any class. However, you can nest a non-persistent-capable class in any persistent-capable class.

Member Pointers

The DDL does not currently support member pointers, which are offsets into classes and structs that are represented differently by different compilers. If you must use the semantics provided by member pointers, you can either use int variables as field offsets (see “ooGetMemberOffset” on page B-7 in Using Objectivity/ C++) or define a portable class wrapper for member pointers.
Constraints

Constraints on DDL apply at several levels, including domain and object-level.

Domain Constraints

Domain constraints specify the data types of values that data members and association links may accept for a particular class. Using the built-in type checking of C++, domain constraints can be enforced at compile time through strong typing or at runtime by using explicit type checking. The DDL provides for and allows the same domain constraints that C++ does.

Object-Level Constraints

Object-level constraints apply to entire objects (as compared to domain and range constraints, which apply only to individual classes). By convention, every persistent-capable class inherits a virtual function called \texttt{ooValidate} from class \texttt{ooObj} that contains checks for all the object-level constraints that apply to objects of that class. You must invoke \texttt{ooValidate} at appropriate points in your application to ensure that the constraints are being followed. The default \texttt{ooValidate} function is empty, so you must customize it before using it.

DDL-Generated Constructors

Objectivity/DB creates a temporary instance of each persistent-capable class as part of its initialization. To facilitate this process, the DDL processor adds a special \texttt{ooInternalObj} constructor to each persistent-capable class that does not already have one. The Objectivity/DB runtime initialization code then uses this special constructor to create the temporary instance for each class.

The DDL-generated \texttt{ooInternalObj} constructor for a persistent-capable class invokes a subconstructor for each of its embedded data objects (base classes or non-static member variables). For a given type of data object, this subconstructor is either the type's \texttt{ooInternalObj} constructor, if it has one, or the type's default constructor, if it has one (defined explicitly or implicitly). If neither constructor exists, the DDL processor reports an error. Even if no error is reported, the automatically-generated \texttt{ooInternalObj} constructor may invoke a default subconstructor with side effects that are undesirable when they occur during Objectivity/DB runtime initialization.
If an ooInternalObj constructor cannot be automatically generated, or if an automatically generated ooInternalObj constructor would result in undesirable side effects, you should do one or more of the following:

- Define an explicit ooInternalObj constructor for the persistent-capable class that explicitly or implicitly invokes a suitable constructor for each of its embedded data objects.
- Define an ooInternalObj constructor for any of the persistent-capable class’ embedded data object’s classes that are causing problems.
- Define a default constructor for any of the persistent-capable class’ embedded data object’s classes that are causing problems.

**Examples**

In the following example, the DDL processor is unable to automatically generate an ooInternalObj constructor for class PersCap because class Embedded has neither a default nor an ooInternalObj constructor.

```cpp
class Embedded {
public:
    Embedded(int);
};

class PersCap : public ooObj {
public:
    Embedded a;
};
```

One way to solve this problem is to explicitly define an ooInternalObj constructor for class PersCap that calls the class Embedded constructor so that the DDL processor does not need to generate an ooInternalObj constructor.

```cpp
class Embedded {
public:
    Embedded(int);
};
```
class PersCap : public ooObj {
public:
    Embedded a;
    PersCap(ooInternalObj) : a(0) {}
};

However, suppose that the constructor for class Embedded performed an undesirable side effect. In this case, the ooInternalObj constructor for class PersCap would also contain an undesirable side effect. To solve the initial problem while avoiding the new problem, you can define an explicit ooInternalObj constructor for class Embedded, allowing the DDL processor to automatically generate an ooInternalObj constructor for class PersCap.

class Embedded {
public:
    Embedded(int);
    Embedded(ooInternalObj) {}
};

class PersCap : public ooObj {
public:
    Embedded a;
};

As an alternative, you can explicitly define a default constructor for class Embedded, again allowing the DDL processor to automatically generate an ooInternalObj constructor for class PersCap.

class Embedded {
public:
    Embedded(int);
    Embedded() {}
};
class PersCap : public ooObj {
public:
    Embedded a;
};
Multiple Data Models

A default data model (schema) is provided by Objectivity/DB for every federated database when it is created. This initial default schema contains the information necessary to support the Objectivity/DB system-defined classes and primitive types, and is shared by all of the class definitions in a federated database. However, you can use multiple named schemas within a single federated database. For example, a company might have one schema for the engineering data in its federated database, another schema for the accounting data, and a third schema for the human resources data. A federated database with such schemas is illustrated in Figure 3-1.

Each schema describes a different subset of the federated database. Each class can be defined only once and only in one schema (except as described in the “Class Versioning” chapter). A database may contain an object of any class defined in any of the schemas of its containing federated database.

Schema Names

You must name all schemas uniquely, since all schemas are considered equal. That is, there is no hierarchy of schemas, implicit or explicit. You can name a schema using the –schema flag with the DDL processor.
Adding Classes to Alternate Schemas

If you use DDL schema files as defined in the "Data Definition Language" chapter and invoke the DDL processor as described in "Running the DDL Processor" on page 1-6, then all classes defined in the DDL schema file are added to the default schema. However, you can add classes to alternate named schemas using the -schema flag.

**DDL Files Containing References to One Schema**

If all the classes defined or referenced in a given DDL schema file are in a single schema other than the default schema, then invoke the DDL processor with the -schema flag.
The following command asks the DDL processor to add the classes defined in `eng.ddl` to the schema named `engSchema` in the federated database specified in the boot file `DBInc.boot` as shown in Figure 3-2:

```
oodd1x -schema engSchema eng.ddl DBInc.boot
```
DDL Files Containing References to Multiple Schemas

A current schema is the schema you specify when invoking the DDL processor. It is either the schema named by the -schema flag or the default schema, if you do not use the -schema flag.

Each class is assumed to belong to the schema that is current when the class is first referenced in any way.

You can specify the current schema using #pragma ooschema as follows:

- #pragma ooschema schemaName

where

schemaName Current schema. schemaName can be *, indicating the default schema, or can be omitted, indicating the schema specified when invoking the DDL processor.

To reference a class defined in a schema other than the one specified by invoking the DDL processor, do the following to your DDL schema file:

1. Change the current schema with #pragma ooschema.
2. Reference the class.
3. Change the current schema back to the one specified when you invoke the DDL processor.

Example

The DDL schema file eng.ddl references two classes — Rod and Gear. Class Rod is defined to be in the schema specified by the oodlx command line. Class Gear is referenced as being in schema mfgSchema, but is not defined. Some other DDL file must define class Gear to be in schema mfgSchema.

#pragma ooschema mfgSchema
class Gear;
#pragma ooschema
class Rod : public ooObj {
public:
    float diameter;
    float length;
    ooRef(Gear) toGear <-> toRod;
};

The following command asks the DDL processor to add the classes defined in eng.ddl to the schema named engSchema in the federated database specified in the boot file DBInc.boot as shown in Figure 3-3. Class Gear is assumed to be defined in the DDL schema file mfg.ddl, which is separately processed by DDL.

ooddlx -schema engSchema eng.ddl DBInc.boot

________
Figure 3-3 Adding Class Definitions to Two Alternate Schemas

3-6 Using Objectivity/C++ Data Definition Language
Multiple Inheritance

You can define classes that inherit attributes from more than one class by using multiple inheritance. This chapter gives a detailed description of how to use Objectivity/DB multiple inheritance.

The following notational conventions are used in the figures in this chapter:

- Non-persistent-capable class
- Non-persistent-capable class whose objects will not be created (an "abstract" non-persistent-capable class)
- Objectivity/DB persistent-capable class `ooobj` whose objects will not be stored
- Persistent-capable class (made persistent by either direct or indirect inheritance from `ooobj`) whose objects will be stored
- Persistent-capable class (made persistent by either direct or indirect inheritance from `ooobj`) whose objects will not be stored
Vehicle Data Model

In this chapter we introduce and expand upon a vehicle data model that assumes you want to define classes of vehicles that have either two wheels or four wheels, and may also have a top speed or a cargo capacity.

Notice that instances of class Vehicle are not created in this data model; only particular kinds of vehicles such as vans, motorcycles, trailers, and cargo vans are useful. Also notice that we never create objects of class topSpeed, cargoCapacity, v4wheel, or v2wheel. These are part attributes that help define particular kinds of vehicles.

Figure 4-1 and the following example show our basic vehicle data model.

![Vehicle Data Model Diagram]

Figure 4-1  Multiple Inheritance
Example

The classes in the model are defined as follows:

- **Van** inherits from class `topSpeed` and class `v4wheel`.
- **Motorcycle** inherits from class `topSpeed` and class `v2wheel`.
- **Trailer** inherits from class `v2wheel` and class `cargoCapacity`.
- **CargoVan** inherits from class `Van` and class `cargoCapacity`.

```cpp
class Van : public topSpeed, public v4wheel {
...
};
class Motorcycle : public topSpeed, public v2wheel {
...
};
class Trailer : public v2wheel, public cargoCapacity {
...
};
class CargoVan : public Van, public cargoCapacity {
...
};
```
Persistence through Inheritance

To exist in a federated database, an object must be an instance of a persistent-capable class. Objectivity/DB defines object persistence through inheritance. This approach follows the Object Database Management Group (ODMG) standard.

Objectivity/DB allows you to make a class persistent only when necessary for your application by inheriting its persistence from one of the Objectivity/DB persistent-capable classes, `ooObj` and `ooContObj`, in either of two ways: indirectly (root persistence) or directly (leaf persistence). Assuming that we want to be able to store objects of our derived vehicle classes (Van, Motorcycle, Trailer, and CargoVan) in a federated database, we could make these classes persistent in either of these ways.

Each Objectivity/DB derived class can inherit from only one persistent base class and that class must be specified first in the list of base classes that are part of the derived class definition.

Persistent-capable classes in Objectivity/DB can inherit from multiple non-persistent base classes of any kind except virtual base classes.

Data Modeling Using Root Persistence

We could make classes Van, Motorcycle, Trailer, and CargoVan persistent by making their common base class Vehicle persistent (root persistence). When a base (root) class is persistent, all of its derived classes inherit persistence.

The following example and Figure 4-2 show how we can make classes Van, Motorcycle, Trailer, and CargoVan persistent using root persistence:

---

Example

- Vehicle inherits persistence from class `ooObj`.
- v4wheel and v2wheel inherit persistence from class Vehicle.
- Van inherits persistence from class v4wheel.
- Motorcycle and Trailer inherit persistence from class v2wheel.
- CargoVan inherits persistence from class Van.

```class Vehicle : public ooObj {
```
class v4wheel : public Vehicle {
...
};

class v2wheel : public Vehicle {
...
};

class Van : public topSpeed, public v4wheel {
...
};

class Motorcycle : public topSpeed, public v2wheel {
...
};

class Trailer : public v2wheel, public cargoCapacity {
...
};

class CargoVan : public Van, public cargoCapacity {
...
};
Adding persistence at the root level of a class hierarchy is the simplest way of making several derived classes persistent-capable. You can cast and convert among classes in the hierarchy just as you could before persistence was added. However, root persistence does not provide as much flexibility for data modeling as either composite objects (see “Using Composite Objects Instead of Multiple Inheritance” on page 4-7) or leaf persistence (see “Data Modeling Using Leaf Persistence” on page 4-9).
Using Composite Objects Instead of Multiple Inheritance

Suppose that we now want to have a persistent-capable class `Motor` so that we can store `Motor` objects in our federated database, and that we want an object of any of our derived vehicle classes (`Van`, `Motorcycle`, `Trailer`, and `CargoVan`) to have a motor. We could accomplish this with multiple inheritance, but it would be better done using composite objects.

A composite object is a collective group of objects linked together by associations.

To illustrate this we will use a part of the class hierarchy from the previous example. The following example and Figure 4-3 show how we can define our persistent `Motor` class and use it as stated above.

---

**Example**

- Vehicle inherits persistence from class `ooObj`.
- Vehicle has a bidirectional association link to class `Motor`.
- Motor inherits persistence from class `ooObj`.
- Motor has a bidirectional association link to class `Vehicle`.
- We can store objects of class `Motor`.
- We can set an association between an object of class `Motor` and an object of class `Van` or class `Motorcycle`.

```cpp
class Vehicle : public ooObj {
public:
    ooRef(Motor) toMotor <-> toVehicle;
};
```

...
class Motor : public ooObj {
    public:
        ooRef(Vehicle) toVehicle <-> toMotor;
};

Figure 4-3 Using Composite Objects
Data Modeling Using Leaf Persistence

Since we do not want to store objects of classes `Vehicle`, `v4wheel`, and `v2wheel`, these classes do not really need to be persistent. In Figure 4-2, they inherit persistence capability from `ooObj` so that objects of classes `Van`, `Motorcycle`, `Trailer`, and `CargoVan` can be stored. But these latter classes can inherit their persistence capability directly from `ooObj` (leaf persistence). Leaf persistence is a more flexible way to model your data than root persistence.

The following example and Figure 4-4 show how we can make classes `Van`, `Motorcycle`, `Trailer`, and `CargoVan` in Figure 4-1 persistent-capable using leaf persistence:

Example

- Van inherits from class `topSpeed` and class `v4wheel`.
- `pVan` inherits persistent-capability from class `ooObj`, and top speed and four wheels from class `Van`.
- Motorcycle inherits from class `topSpeed` and class `v2wheel`.
- `pMotorcycle` inherits persistent-capability from class `ooObj`, and top speed and two wheels from class `Motorcycle`.
- Trailer inherits from class `v2wheel` and class `cargoCapacity`.
- `pTrailer` inherits persistent-capability from class `ooObj`, and two wheels and cargo capacity from class `Trailer`.
- CargoVan inherits from class `Van` and class `cargoCapacity`.
- `pCargoVan` inherits persistent-capability from class `ooObj`, and top speed and four wheels from class `CargoVan`.
- Classes `Vehicle`, `v4wheel`, `v2wheel`, `Van`, `Motorcycle`, `Trailer`, and `CargoVan` remain non-persistent-capable classes.
class Vehicle {
    ...
};

class pVan: public ooObj, public Van {
    ...
};
class pMotorcycle: public ooObj, public Motorcycle {
    ...
};
class pTrailer: public ooObj, public Trailer {
    ...
};
class pCargoVan: public ooObj, public CargoVan {
    ...
};
Although the data model shown in Figure 4-4 looks more complicated than the root persistence version in Figure 4-2, leaf persistence is the simplest way to make selected classes persistent in an existing class hierarchy. The original non-persistent-capable classes are preserved and can be used with existing class libraries. Using leaf persistence, it is easy to make other classes persistent as you extend the hierarchy.
Enhanced Leaf Persistence

Assume that we want to share a common set of associations between objects of several persistent-capable classes. We can do this by creating a persistent-capable base class for a related group of leaf classes and defining the association links common to the group in the base class. The persistent-capable leaf classes can then inherit both their persistence and their association links from that class.

The following example and Figure 4-5 show how we can let classes pVan and pCargoVan share a set of associations:

Example

- pVan inherits persistent-capability from class ooObj and defines association links to classes Motor and airbag.
- pVan inherits persistent-capability and the association links from class pVan, and topSpeed and v4wheel from class Van.
- pCargoVan inherits persistent-capability and the association links from class pVan, and topSpeed and v4wheel from class CargoVan.

```cpp
class paVan : public ooObj {
public:
    ooRef(Motor) paVanToMotor <-> motorToPaVan[];
    ooRef(airbag) paVanToAirbag[] <-> airbagToPaVan[];
    ...
};
...

class pVan : public paVan, public Van {
    ...
};
...

class pCargoVan : public paVan, public CargoVan {
    ...
};
```

4-12 Using Objectivity/C++ Data Definition Language
Figure 4-5 Enhanced Leaf Persistence
Mixing Root and Leaf Persistence

It is often advantageous to make some classes in a hierarchy persistent-capable through root inheritance and other classes persistent-capable through leaf inheritance. When most of the classes used by your applications are in one branch of the hierarchy, you can use root persistence for that branch and use leaf persistence for selected classes in other branches of the hierarchy.

Assume again, as we did in “Using Composite Objects Instead of Multiple Inheritance” on page 4-7, that we want to have a persistent-capable class Motor so that we can store Motor objects in our federated database, and that we want an object of any of our derived vehicle classes (Van, Motorcycle, Trailer, and CargoVan) to be able to have a motor. We accomplished this in Figure 4-3 by using root persistence in two separate class hierarchies to:

◆ Create the persistent-capable Motor class in one class hierarchy
◆ Pass persistent-capability from the Vehicle class to the derived vehicle classes in another class hierarchy

Then we used a bidirectional association link between classes Motor and Vehicle to enable objects of the vehicle classes to have motors.

We could accomplish the same goals in a single class hierarchy by mixing root and leaf persistence to create the following:

◆ Persistent-capable classes Van, Motorcycle, Trailer, and CargoVan that inherit from the Vehicle class
◆ Non-persistent motor class (Motor) so that objects of these persistent-capable classes can have motors
◆ Persistent-capable motor class (pMotor) so that motor objects can be stored

The following example and Figure 4-6 show how we can mix root and leaf persistence in the same class hierarchy:
Example

- **pMotor** inherits persistent-capability from class `ooObj` and motor attributes from class `Motor` (leaf persistence).
- **Vehicle** inherits persistent-capability from class `ooObj` (root persistence).
- **Van** inherits persistent-capability from class `v4wheel` (root persistence), and a motor from class `Motor`.
- Objects of class `pMotor` can be stored.

```cpp
class Motor {
    ...
};
class pMotor : public ooObj, public Motor {
    ...
};

class Vehicle : public ooObj {
    ...
};
...;
class Van : public v4wheel, public topSpeed, public Motor {
    ...
};
```
By using root persistence to create our persistent-capable derived Vehicle classes such as Van, we have used the easiest way of making all of these derived classes persistent-capable. By using leaf persistence to create class pMotor, we have left open other possibilities for the Motor branch of the class hierarchy. For example, we could now create other non-persistent-capable motor classes for particular kinds of motors (four cylinder, six cylinder, turbo, and so on) through inheritance from class Motor, and use leaf persistence if we wanted to make any of these motor classes persistent-capable.
Schema Evolution

This chapter presents schema evolution tasks you can perform to change an existing federated database schema and the effects of these changes on existing applications. You should use this chapter in conjunction with the “Object Conversion” chapter in Using Objectivity/ C++.

Basics

Over time, you may need to change your data model to meet new or changed requirements. The process of changing your data model is known as schema evolution. Objectivity/C++ schema evolution allows you to:

- Delete, add, or change class contents, including data members, associations, and object references
- Delete and rename classes
- Change inheritance relationships between classes

Schema evolution is required if you make changes to user-defined persistent-capable and non-persistent-capable classes used as data member types or as base classes for persistent-capable classes. Persistent-capable classes are any classes derived from a persistent Objectivity/C++ base class (ooObj, ooContObj, and so on). Non-persistent-capable classes used only for transient objects (that is, transient classes) are not stored in the schema, and therefore do not require schema evolution.

Through schema evolution, you can change any data member of a class. However, certain changes, such as changing the type of a non-primitive data member, may require multiple schema evolution passes.

When you change a schema, existing objects of the changed classes, called affected objects, may need to be converted to reflect the changes. The process of changing existing objects to reflect schema changes is called object conversion. Objectivity/C++ automatically performs object conversion for all schema evolution operations. However, certain schema changes may require you to augment the
automatic object conversion through user-defined applications or conversion functions.

The following is a brief description of object conversion. For detailed information, see the “Object Conversion” chapter in Using Objectivity/ C++.

Objectivity/C++ supports three object conversion modes—deferred, on-demand, and immediate. Which mode you choose to use depends on the nature of your schema changes and your application requirements. The effects and requirements of these modes are summarized in Table 5-1.

### Table 5-1: Object Conversion Modes

<table>
<thead>
<tr>
<th>Mode</th>
<th>Granularity</th>
<th>DDL Processing</th>
<th>Automatic Conversion</th>
<th>Conversion Function</th>
<th>Conversion Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deferred</td>
<td>Object</td>
<td>ooddix -evolve</td>
<td>Yes</td>
<td>Optional</td>
<td>Any user application</td>
</tr>
<tr>
<td>On-Demand</td>
<td>Container</td>
<td>ooddix -evolve</td>
<td>Yes</td>
<td>Optional</td>
<td>Any user application or a separate conversion application</td>
</tr>
<tr>
<td></td>
<td>Database</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Federated Database</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Immediate</td>
<td>Federated Database</td>
<td>ooddix -evolve</td>
<td>Yes</td>
<td>Optional</td>
<td>Upgrade application that calls ooTrans::upgrade() and ooRefHandle(ooFDObj)::upgradeObjects()</td>
</tr>
</tbody>
</table>

### Deferred Mode

With the deferred mode, Objectivity/C++ automatically converts an affected object to its most recent class representation when any user application accesses the object for the first time following the change to its class. This mode is useful for systems that must remain available at all times. You can also use the deferred mode to convert objects as they are accessed until it is convenient to finish converting all objects within a container, database, or federated database using the on-demand mode.
On-Demand Mode

With the on-demand mode, you trigger automatic object conversion from within any user application at the granularity of a container, database, or the entire federated database. You can use this mode to selectively convert objects based on their containment. Performing on-demand conversion on a federated database also purges schema change information from the schema. You might want to use a combination of the deferred mode and on-demand mode at a container or database level until it is convenient to finish converting all objects using on-demand mode on the entire federated database.

Immediate Mode with an Upgrade Application

With the immediate mode, Objectivity/C++ automatically converts all affected objects in a federated database within a single transaction with the help of a single-use upgrade application. Performing immediate conversion also purges changed schema information from the schema.

Immediate mode and an upgrade application are only required for schema operations that include replacing a base class or deleting a class that has inherited associations or object references. For these changes, the DDL processor marks the classes affected by the change as protected until the upgrade application is run. If a non-upgrade application tries to access an object of one of these protected classes before the upgrade application is run, Objectivity/C++ issues an error message.

When processing the schema for these changes, you must run the DDL processor using the -evolve and -upgrade flags (see the “Tools” appendix). When creating the upgrade application, you must use upgrade interface member functions ooTrans::upgrade and ooRefHandle(ooFDObj)::upgradeObjects. For information on how to create an upgrade application, see “Creating an Upgrade Application” on page 22-4 in Using Objectivity/C++.

Conversion Function

For any object conversion mode, you can register a conversion function that will be called after the automatic conversion performed by Objectivity/C++. You can use conversion functions to compute new data member values based on existing values. You can register this function for any user-defined application, including an upgrade application.
Other User Applications

Besides the upgrade application described above, you may need to create and run other applications to convert certain objects before running your normal applications. For example, you may want to create separate applications for on-demand object conversion.

For some schema evolution operations, you may need to create an application to set the values of data members for affected objects. For example, this may be needed for complex schema evolution operations that require more than one run of the DDL processor.

Effects of Schema Evolution

It is important to understand the effects your schema changes will have. This section summarizes these effects. To determine which effects an operation will have, refer to the task description in this chapter for the specific operation. This information is also summarized in tables in the “Schema Evolution Basic Operation Summary” appendix.

Effects on Existing Persistent Objects

Schema evolution operations are categorized by their effects on existing persistent objects. An operation is either a conversion or non-conversion operation. These are defined as follows:

- **Conversion operation**: Changes existing persistent objects. For example, adding a data member is a conversion operation.
- **Non-conversion operation**: Does not change existing persistent objects. For example, changing a data member’s name or its access control are both non-conversion operations.

Both conversion and non-conversion operations require you to rebuild existing applications. For both of these operations, you may also need to modify your application to reflect the changes you made to the schema.

With conversion operations, you can provide a user-defined conversion function to augment the automatic conversion performed by Objectivity/C++. You cannot provide a conversion function for non-conversion operations, since no object conversions are performed. For information on how to write conversion functions, see the “Object Conversion” chapter in Using Objectivity/C++.
Sequential Schema Evolutions

Objectivity/C++ schema evolution retains a history of schema changes to convert affected objects, even if there have been multiple schema evolutions since the affected objects were updated. This history remains in the schema until you perform an on-demand conversion of the entire federated database, or use the immediate mode to convert objects with the help of an upgrade application.

Basic and Complex Operations

Objectivity/C++ schema evolution directly supports many basic schema evolution operations. These basic operations require only one pass through the schema evolution process. Basic operations are described in detail in later sections of this chapter.

You can combine basic operations to perform more complex changes as needed. Complex changes of this kind require two or more passes through the schema evolution process. For complex changes, you should be certain that all affected objects are converted to each new schema before proceeding to the next schema evolution. Later sections of this chapter describe complex operations in detail.
General Schema Evolution Process

The schema evolution process is shown in Figure 5-1. This process creates:

- Modified DDL output files (header and C++ source files), which you need to use to rebuild existing applications
- Evolved schema information stored in the federated database

The schema evolution process consists of three phases—preparation, schema evolution, and cleanup. The process you will follow depends on the changes you need to make to your schema.

![Figure 5-1 Schema Evolution](image)

5-6 Using Objectivity/C++ Data Definition Language
Preparation Phase

Step 1: Plan your schema changes.

Before performing schema evolution, you should plan the changes you need to make to your schema, and determine how these changes will affect your existing applications and deployed federated databases. To help with this analysis, refer to the following sections based on the kind of changes you need to make:

- Class content operations
  - “Basic Class Content Operations” on page 5-11
  - “Complex Class Content Operations” on page 5-43
- Class operations
  - “Basic Class Operations” on page 5-53
  - “Complex Class Operations” on page 5-58
  - “Multiple Class Operations” on page 5-79
- Inheritance operations
  - “Basic Inheritance Operations” on page 5-60
  - “Complex Inheritance Operations” on page 5-75

When making schema changes, you should first test them on a copy of your federated database. This way, you can ensure that your changes are what you intend before you commit them to your actual federated database. You can create a copy of your federated database by making a backup, or by using the oocopyfd tool (see the “Federated Database Tasks” chapter in Objectivity/DB Administration).

You should also make backup copies of any DDL schema files and application source code files before modifying them.
Step 2: Modify your DDL schema files.

Modify your DDL schema files according to the schema changes you planned in step 1. You should include files for all classes affected by the changes in the same run of the DDL processor. For some changes, you may need to provide additional information in the form of DDL pragma statement, as described below.

Specifying Changes Through DDL Pragmas

When evolving a schema, the DDL processor compares the DDL schema files you modified with the existing schema stored in the federated database. Using this technique, the DDL processor is able to deduce many of the changes made to class definitions. However, some schema evolution operations, such as renaming a class or a data member, cannot be deduced in this way. Because of this, you must define these operations using special pragmas. You can also specify object conversion information through pragmas. For example, the default value of a new data member is specified through a pragma.

Table 5-1 lists the pragmas you must use for certain schema operations. These pragmas are described in the tasks in which they are used.

Table 5-1: Schema Evolution Pragmas

<table>
<thead>
<tr>
<th>Pragma</th>
<th>Operation</th>
<th>See page...</th>
</tr>
</thead>
<tbody>
<tr>
<td>#pragma oorename existingName</td>
<td>Rename a data member or class</td>
<td>5-17 5-53</td>
</tr>
<tr>
<td>#pragma oodelete className</td>
<td>Delete a class</td>
<td>5-55</td>
</tr>
<tr>
<td>#pragma oodefault value</td>
<td>Set a default value of a new data member</td>
<td>5-16</td>
</tr>
<tr>
<td>#pragma oochangebase existingBaseClass -&gt; newBaseClass [,moreNewBaseClasses]</td>
<td>Replace a base class with a subset of its base classes</td>
<td>5-64</td>
</tr>
<tr>
<td>#pragma oochangebase existingBaseClass -&gt; newBaseClass</td>
<td>Replace a base class with one of its derived classes</td>
<td>5-68</td>
</tr>
</tbody>
</table>
General Schema Evolution Process

Schema Evolution Phase

**Step 1: Process your DDL schema files.**

Process your DDL schema files using the DDL processor with the `-evolve` flag. This updates the federated database schema and generates a new set of DDL output files.

If you are replacing a base class or deleting a class with inherited associations or object references, you must specify both the `-evolve` flag and the `-upgrade` flag when running the DDL processor. For information about DDL processor flags, see the “Tools” appendix.

If the DDL processor encounters an error, it will issue a warning or error message. There are two different message types that the DDL processor may generate while performing schema evolution. These message types are as follows:

- **Warning**: Acknowledges an explicitly-defined pragma operation. For example, renaming and deleting a class generates warning messages. Warnings also indicate an implicitly-defined operation that is deduced by comparing the modified input DDL schema file and the stored federated database schema. For example, adding and deleting a data member generates warning messages.

- **Error**: Indicates an invalid operation was attempted. The cause of the error is indicated in the message contents.

**Step 2: Build applications.**

After processing your DDL schema files, you must rebuild all existing applications using the newly-generated header files and C++ source files. The schema changes you make may also require you to modify your applications before rebuilding them. For certain operations, you may also need to create a special object conversion application to convert all affected objects before running other applications. See “Other User Applications” on page 5-4.
The following guidelines should help you determine what you need to do to your applications when rebuilding them:

You do not need to modify your existing applications if:

- You are using the deferred conversion mode and do not require a conversion function.

You must modify existing applications if:

- Your object conversion requires a conversion function and you want to call this function from within an existing application using the deferred conversion mode.
- You made a conversion operation and the application accesses affected objects, such as the need to access an added data member.
- You made a non-conversion operation that changes the name of a class or data member, and the application accesses the changed class or data member by name. For example, you must modify and rebuild the application if the application uses queries that reference a renamed class or data member.

You must create a special conversion application if:

- You are using the on-demand conversion mode and want a separate application that can be called to convert all objects within a specified container, database, or the entire federated database.
- You change a base class or delete a class that contains inherited references. In these cases, you need to create and run a single-use, upgrade application that calls upgrade interface member functions. You must run this upgrade application in the immediate conversion mode. It converts all affected objects that require conversion by an upgrade application. These schema changes also require you to run the DDL processor using the `--evolve` and `--upgrade` flags.
- You need a separate application to set the values of data members for all affected objects for complex operations that require more than one run of the DDL processor.
Cleanup Phase

**Step 1: Remove Schema Evolution Pragma Statements.**

To reuse the DDL schema files in subsequent runs of the DDL processor, manually remove any pragma statements from your DDL schema files you added for the schema evolution process.

**Basic Class Content Operations**

This section describes operations you can perform to change class contents in your schema, including data members, virtual member functions, associations, object references, and VAarrays of object references. These operations include:

- Deleting a data member
- Adding a data member with system default values
- Adding a data member with a user-defined default value
- Renaming a data member
- Changing the type of a primitive data member
- Changing the order (position) of data members
- Changing access control of a data member (public, private, protected)
- Adding the first virtual member function
- Removing the last virtual member function
- Changing the size of an array data element
- Adding an inline association
- Adding a non-inline association
- Adding an object reference
- Deleting an association
- Deleting an object reference
- Changing an object reference from short to long and vice versa
- Changing the representation (inline/non-inline) of an association
- Changing an inline association from short to long and vice versa
- Changing the behavior specifiers of an association (lock and delete propagation, versioning and copy properties)
- Renaming an association
- Changing the access control of an association
Basic Class Content Operations

- Changing an array size for an array of object references
- Changing the order (position) of a non-inline association
- Changing the order (position) of an inline association or object reference

Effects of a Class Content Operation on Other Classes

When changing the contents of a persistent-capable class, you may also need to change its related classes, including:

- Derived classes of the changed class, since all conversion operations on class contents are propagated to the derived classes
- All classes that have a data member of a modified non-persistent-capable class
- Classes that have associations or object references to the changed class
- All instantiations (and their specializations) of a modified template class

When changing the contents of a non-persistent-capable class, you may also need to change its related classes, including:

- Derived classes
- Classes that use the changed class as a type for an embedded data member, either locally-defined or inherited

When performing most of these class content operations, you must process the above classes along with the changed class when you run the DDL processor to evolve the schema. You do not need to include these other classes for renaming a data member or changing its access control.
Initial Schema Used for Class Content Examples

The examples presented in this section are based on the following schema. Each example shows the original and modified class definitions for the documented operation.

```
// Original schema used for examples of class content changes

// Persistent-capable class A
class A : public ooObj {
public:
   uint8 aUint8;
   float64 aFloat64;
   char aChar;
};

// Persistent-capable class B
class B : public A {
public:
   uint16 aUint16;
   float32 aFloat32;
};

class D;

// Persistent-capable class C
class C : public ooObj {
public:
   float32 aFloat32;
   int32 anArrayOfInts[20];
   ooRef(D) anArrayOfRefs[20];
   ooRef(B) refToB; // object reference to B
};
```
// Non-persistent-capable class X
class X {
public:
    uint8 aUint8;
    char *aString;
};

// Non-persistent-capable class Y
class Y : X {
public:
    float64 aFloat64;
};

// Persistent-capable class D
class D : public ooObj {
public:
    char *aName;
    float32 anArrayOfFloats[20];
    ooRef(A) assocToA : copy(delete); // unidirectional
    inline ooShortRef(C) assocToC : copy(delete);
    Y embeddedMember;
};
Deleting a Data Member

Deleting a data member is a conversion operation and it creates a new representation of the changed class.

1. Remove the data member from the target class definition.
2. If the changed class is a persistent-capable class, run the DDL processor on the affected class definition and its derived classes. If the changed class is a non-persistent-capable class, run the DDL processor on the affected class, its derived classes, and the classes that embed it.
3. Rebuild your existing applications using the newly-generated header and C++ source files.

---

Example

```cpp
// Original class definition
// Persistent-capable class A
class A : public ooObj {
public:
    uint8 aUint8;
    float64 aFloat64;
    char aChar;
};
...

// Modified class definition
// Delete the float64 data member from class A
class A : public ooObj {
public:
    uint8 aUint8;
   // float64 aFloat64;
    char aChar;
};
...
```

```
Adding a Data Member

Adding a data member is a conversion operation and it creates a new representation of the changed class.

1. Add the data member to the target class definition.
2. If the data member is a primitive member, and you want to specify a default value, use the `oodefault` pragma. You must define this pragma immediately before the added data member. You can specify default values only for newly added data members. The type and range of the default value must match those of the data member type. If you do not specify a default value, the member is set to zero.
   
   ```
   #pragma oodefault value
   ```
   
   where
   ```
   value
   ```
   
   One of the following types: `uint8` (or `unsigned char`), `int8` (or `signed char`), `uint16`, `int16`, `uint32`, `int32`, `float32`, and `float64`

3. Process your DDL schema files by running the DDL processor with the `-evolve` flag.
4. Modify your applications to use the added data member and rebuild them using the newly-generated header and C++ source files.
5. To reuse your DDL schema files, remove the `oodefault` pragma statements.

---

**Example**

```c++
// Original class definition
// Persistent-capable class A
class A : public ooObj {
public:
    uint8 aUint8;
    float64 aFloat64;
    char aChar;
};
```
Basic Class Content Operations

// Modified class definition  
// Add an int16 data member to class A 
class A : public ooObj {
public:
    uint8 aUint8;
    float64 aFloat64;
    char aChar;
    +    #pragma oodefault 100
    +    int16 anInt16; // new data member set to a value of 100
};

Renaming a Data Member

Renaming a data member is a non-conversion operation and it does not create a new representation of the changed class.

1. Change the data member name in the target class definition.
2. Specify the following pragma to indicate the existing data member name. The new name must be unique to the changed class. You must define this pragma immediately before the renamed data member.
   
   #pragma oorename existingName 
   
   where 
   
   existingName                Existing data member name

3. Process your DDL schema files by running the DDL processor with the -evolve flag.
4. Modify your applications to use the renamed data member. For example, if your applications reference the modified data member by name through indexing or query operations, change the applications to use the new name. Rebuild your applications using the newly-generated header and C++ source files.
5. To reuse your DDL schema files, remove the oorename pragma statements.
Example

// Original class definition
// Persistent-capable class A
class A : public ooObj {
public:
    uint8 aUint8;
    float64 aFloat64;
    char aChar;
};

// Modified class definition
// Rename data member aFloat64 to be myFloat64
class A : public ooObj {
public:
    uint8 aUint8;
+    #pragma oorenname aFloat64
Δ    float64 myFloat64; // renamed data member
    char aChar;
};

5-18 Using Objectivity/C++ Data Definition Language
Changing a Primitive Data Member Type

Changing a primitive data member type is a conversion operation and it creates a
new representation of the changed class. When changing the primitive type of a
data member, you should consider whether or not the conversion will preserve the
value of the data member. Use the following guidelines to determine the effect of
the conversion you are performing. These conversions follow standard C++
semantics for type conversions. Converting from a float type to an integer type
truncates the value. Converting an integer to a signed type may be architecture-
dependent if the value cannot be represented in the new type.

Data member values may not be preserved for:
- Integer-to-integer conversions or float-to-float conversions where bits are lost
  from truncation
- Integer-to-integer conversions where signedness is changed
- Float-to-integer where float is a non-integral value
- Integer-to-float where the integer value cannot be exactly represented in the
  float

⚠️

Warning

Conversion semantics for primitive data members types may differ between
compilers. Objectivity/C++ does not resolve these differences across compilers and
platforms.

1. Change the data member type in the target class definition in the DDL schema
   file.
2. Process your DDL schema files by running the DDL processor with the
   -evolve flag.
3. If necessary, modify your application to use the changed data member type.
4. Rebuild your existing applications using the newly-generated header and C++
   source files.
Example

// Original class definition
// Persistent-capable class A
class A : public ooObj {
public:
    uint8 aUint8;
    float64 aFloat64;
    char aChar;
};

// Modified class definition
// Changing data member aUint8’s type from uint8 to int16
class A : public ooObj {
public:
    int16 aUint8; // change type to int16
    float64 aFloat64;
    char aChar;
};
Changing the Order (Position) of a Data Member

Changing the order (position) of a data member is a conversion operation. The DDL processor will issue a warning message if this is the only conversion change on the class.

1. Reorder the data member in the target class definition.
2. Process your DDL schema files by running the DDL processor with the -evolve flag.
3. Rebuild your existing applications using the newly-generated header and C++ source files.

Example

```cpp
// Original class definition
// Persistent-capable class A
class A : public ooObj {
public:
    uint8 aUint8;
    float64 aFloat64;
    char aChar;
};
```
// Modified class definition
// Changing the order of data member aChar
class A : public ooObj {
public:
    char aChar; // changed position
    uint8 aUint8;
    float64 aFloat64;
};

### Changing the Access Control of a Data Member

Changing the access control of a data member is a non-conversion operation. You can change the access control (public, private, protected) of data members, associations, object references, and base classes.

1. Change the access control of the target data member.
2. Process your DDL schema files by running the DDL processor with the `--evolve` flag.
3. Rebuild your existing applications using the newly-generated header and C++ source files.

---

**Example**

// Original class definition
// Persistent-capable class A
class A : public ooObj {
public:
    uint8 aUint8;
    float64 aFloat64;
    char aChar;
};
// Modified class definition
// Changing the access control of data member aChar
class A : public ooObj {
public:
    uint8 aUint8;
    float64 aFloat64;
+     private:
        char aChar; // changed access control
};

Adding the First Virtual Member Function

Adding the first virtual member function to a non-persistent-capable class used by a persistent-capable class is a conversion operation. If you introduce the first virtual member function in a non-persistent-capable class, the affected objects will require additional space to be allocated for their virtual table (vtable) pointers. This increases the size of the affected persistent objects.

Introducing the first virtual member function can occur when you change a non-virtual member function to be virtual in a class that has no virtual member functions, or when you add a new virtual member function to the class.

Note that persistent-capable classes always have a vtable pointer, regardless of their virtual member function usage.

Example

// Original class definition
// Non-persistent-capable class X
class X {
public:
    uint8 aUint8;
    char *aString;
};
// Modified class definition
// Add first virtual member function to
// non-persistent-capable class X
class X {
public:
    uint8 aUint8;
    char *aString;
    +    virtual void aVirtual(); // first virtual member function
};

Removing the Last Virtual Member Function

Removing the last virtual member function from a non-persistent-capable class used by a persistent-capable class is a conversion operation. If you remove the last virtual member function of a non-persistent-capable class, the affected objects will lose the space allocated for their virtual table (vtable) pointers. This decreases the size of the affected objects.

Removing the last virtual member function can occur when you change the last virtual member function to be non-virtual, or when you delete the last virtual member function from the class.

Note that persistent-capable classes always have a vtable pointer, regardless of their virtual member function usage.

Changing the Size of a Fixed-Length Array Data Member

Changing the size of a fixed-length array data member is a conversion operation. Changing the array size (number of elements) of a data member changes the size of the affected objects. Objectivity/DB initializes the new array by making a left-justified copy of the original array. If the array size is increased, additional elements in the new array are set to zero. If the data member is a multidimensional array, the copy of the elements is treated as a single array.

You can also perform related operations on fixed-length arrays, including changing:

◆ A single data member to be an array of the same type or compatible type
◆ An array data member to be a single data member

5-24    Using Objectivity/C++ Data Definition Language
1. Change the array size as desired in the target class definition.
2. Process your DDL schema files by running the DDL processor with the
   -evolve flag.
3. If necessary, modify your application to use the array data member. Rebuild
   your existing applications using the newly-generated header and C++ source
   files.

Example

```cpp
// Original class definition
class D;
// Persistent-capable class C
class C : public ooObj {
public:
    float32 aFloat32;
    int32 anArrayOfInts[20];
    ooRef(D) anArrayOfRefs[20];
    ooRef(B) refToB; // object reference to B
};

// Modified class definition
// Changing the size of a fixed array anArray
class D;
class C : public ooObj {
public:
    float32 aFloat32;
    Δ int32 anArrayOfInts[100]; // changed from 20 to 100
    ooRef(D) anArrayOfRefs[20];
    ooRef(B) refToB; // object reference to B
};
```
Adding an Inline Association

Adding an inline association is a conversion operation and is similar to adding a data member. You must include all classes affected by adding a bidirectional association in the same run of the DDL processor. For example, adding a bidirectional association between two classes A and B requires you to include the DDL class definitions of A and B in the same run of the DDL processor.

1. Add the inline association to the target class definition.
2. Process your DDL schema files by running the DDL processor with the -evolve flag.
3. Modify and rebuild your existing applications using the newly-generated header and C++ source files.

Example

```c++
// Original class definitions
// Persistent-capable class A
class A : public ooObj {
public:
    uint8 aUint8;
    float64 aFloat64;
    char aChar;
};
...

class C;
// Persistent-capable class C
class C : public ooObj {
public:
    float32 aFloat32;
    int32 anArrayOfInts[20];
    ooRef(D) anArrayOfRefs[20];
    ooRef(B) refToB; // object reference to B
};
...
```
// Modified class definitions
// Adding an inline bidirectional association from A to C
class C;
class A : public ooObj {
public:
  uint8 aUint8;
  float64 aFloat64;
  char aChar;
  +  inline ooRef(C) assocToC <-> assocToA; // bidirectional
};
...
// Adding an inline bidirectional association from C to A
class D;
class C : public ooObj {
public:
  float32 aFloat32;
  int32 anArrayOfInts[20];
  ooRef(D) anArrayOfRefs[20];
  ooRef(B) refToB; // object reference to B
  +  inline ooRef(A) assocToA <-> assocToC; // bidirectional
};
...

**Adding a Non-Inline Association**

Adding a non-inline association is a non-conversion operation. You must include all classes affected by adding a bidirectional association in the same run of the DDL processor. For example, adding a bidirectional association between two classes A and B requires you to include the DDL class definitions of A and B in the same run of the DDL processor.
1. Add the non-inline association to the target class definition.
2. Process your DDL schema files by running the DDL processor with the
   `--evolve` flag.
3. Modify and rebuild your existing applications using the newly-generated
   header and C++ source files.

Example

```c++
// Original class definition
// Persistent-capable class A
class A : public ooObj {
public:
    uint8 aUint8;
    float64 aFloat64;
    char aChar;
};

// Modified class definition
// Adding a non-inline unidirectional association from A to C
class C;
class A : public ooObj {
public:
    uint8 aUint8;
    float64 aFloat64;
    char aChar;
+    ooRef(C) assocToC : copy(delete);
};
```
Adding an Object Reference

Adding an object reference is a conversion operation and is similar to adding a data member.

1. Add the object reference to the target class definition.
2. Process your DDL schema files by running the DDL processor with the -evolve flag.
3. Modify and rebuild your existing applications using the newly-generated header and C++ source files.

Example

```c++
// Original class definition
// Persistent-capable class B
class B : public A {
public:
    uint16 aUint16;
    float32 aFloat32;
};

// Modified class definition
// Adding an object reference from B to C
class C;
class B : public A {
public:
    uint16 aUint16;
    float32 aFloat32;
+    ooRef<C> refToC; // object reference to C
};
```
Deleting an Association

Deleting an association is a conversion operation and is similar to deleting a data member. You must include all classes affected by deleting a bidirectional association in the same run of the DDL processor. For example, deleting a bidirectional association between two classes A and B requires you to include the DDL class definitions of A and B in the same run of the DDL processor.

1. Delete the association from the target class definition.
   For bidirectional associations, you must also delete the inverse association in the related class.
2. Process your DDL schema files by running the DDL processor with the -evolve flag.
3. Modify and rebuild your existing applications using the newly-generated header and C++ source files.

Example

```cpp
// Original class definition
// Persistent-capable class D
class D : public ooObj {
    public:
        char *aName;
        float32 anArrayOfFloats[20];
        ooRef(A) assocToA : copy(delete); // unidirectional
        inline ooShortRef(C) assocToC : copy(delete);
};
```
Deleting an Object Reference

Deleting an object reference is a conversion operation and is similar to deleting a data member.

1. Remove the object reference from the target class definition.
2. Process your DDL schema files by running the DDL processor with the `--evolve` flag.
3. Modify and rebuild your existing applications using the newly-generated header and C++ source files.

Example

```cpp
// Original class definition
class D;
// Persistent-capable class C
class C : public ooObj {
    public:
        float32 aFloat32;
        int32 anArrayOfInts[20];
        ooRef(D) anArrayOfRefs[20];
        ooRef(B) refToB; // object reference to B
};
```
// Modified class definition
class D;
// Deleting an object reference from C to B
class C : public ooObj {
public:
    float32 aFloat32;
    int32 anArrayOfInts[20];
    ooRef(D) anArrayOfRefs[20];
    - ooRef(B) refToB; // object reference to B
};

**Changing an Object Reference from Short to Long or Vice Versa**

Changing an object reference from short to long or vice versa is a conversion operation. Changing an object reference from a short reference to a long reference retains its value. Changing from a long reference to a short reference deletes the database and container information in the reference. In this case, object references of the affected objects are nullified to avoid inconsistent references between objects in different containers.

1. Change the object reference type in the target class definition.
2. Process your DDL schema files by running the DDL processor with the -evolve flag.
3. Modify and rebuild your existing applications using the newly-generated header and C++ source files.
Example

// Original class definition
class D;
// Persistent-capable class C
class C : public ooObj {
public:
    float32 aFloat32;
    int32 anArrayOfInts[20];
    ooRef(D) anArrayOfRefs[20];
    ooRef(B) refToB;  // object reference to B
};

// Modified class definition
class D;
// Changing an object reference from long to short
class C : public ooObj {
public:
    float32 aFloat32;
    int32 anArrayOfInts[20];
    ooRef(D) anArrayOfRefs[20];
    inline ooShortRef(B) refToB;  // short object reference to B
};
Changing the Representation (Inline/Non-Inline) of an Association

Changing the representation of an association from inline to non-inline (or vice versa) is a conversion operation.

Changing from a non-inline association (which uses long OIDs) to an inline that uses short OIDs deletes the database and container information in the association. In this case, associations of the affected objects are nullified to avoid inconsistent references between objects in different containers.

For bidirectional associations, the inverse class must be included in the same run of the DDL processor.

1. Change the association representation in the target class definition.
2. Process your DDL schema files by running the DDL processor with the -evolve flag.
3. Modify and rebuild your existing applications using the newly-generated header and C++ source files.

Example

```cpp
// Original class definition
// Persistent-capable class D
class D : public ooObj {
public:
    char *aName;
    float32 anArrayOfFloats[20];
    ooRef(A) assocToA : copy(delete);
    inline ooShortRef(C) assocToC : copy(delete);
};
```
// Modified class definition
// Changing inline associations to non-inline and vice versa
class D : public ooObj {
public:
    char *aName;
    float32 anArrayOfFloats[20];
inline ooRef(A) assocToA : copy(delete);
inline ooShortRef(C) assocToC : copy(delete);
};

### Changing an Inline Association from Short to Long or Vice Versa

Changing an inline association from short to long or vice versa is a conversion operation similar to changing an object reference from short to long and vice versa.

Changing an inline association from a short association to a long retains the value of the association.

Changing an inline association from a long association to a short deletes the database and container information in the association. In this case, associations of the affected objects are nullified to avoid inconsistent references between objects in different containers.

For bidirectional associations, the inverse class must be included in the same run of the DDL processor.

1. Change the association type in the target class definition.
2. Process your DDL schema files by running the DDL processor with the `-evolve` flag.
3. Modify and rebuild your existing applications using the newly-generated header and C++ source files.
Example

// Original class definition
// Persistent-capable class D
class D : public ooObj {
public:
  char *aName;
  float32 anArrayOfFloats[20];
  ooRef(A) assocToA : copy(delete); // unidirectional
    inline ooShortRef(C) assocToC : copy(delete);
};

// Modified class definition
// Changing a short inline association to long
class D : public ooObj {
public:
  char *aName;
  float32 anArrayOfFloats[20];
  ooRef(A) assocToA : copy(delete);
  inline ooRef(C) assocToC : copy(delete);
};
Changing Association Behavior Specifiers

Changing association behavior specifiers is a non-conversion operation. You can change lock and delete propagation, versioning, and copy properties of an association.

1. Change the behavior specifiers of the association in the target class definition.
2. Process your DDL schema files by running the DDL processor with the -evolve flag.
3. Modify and rebuild your existing applications using the newly-generated header and C++ source files.

Example

```c++
// Original class definition
// Persistent-capable class D
class D : public ooObj {
public:
    char *aName;
    float32 anArrayOfFloats[20];
    ooRef(A) assocToA : copy(delete); // unidirectional
    inline ooShortRef(C) assocToC : copy(delete);
};

// Modified class definition
// Changing the behavior specifier of an association
class D : public ooObj {
public:
    char *aName;
    float32 anArrayOfFloats[20];
    ooRef(A) assocToA : delete(propagate); // change specifier
    inline ooShortRef(C) assocToC : copy(delete);
};
```
Renaming an Association

Renaming an association is a non-conversion operation. See “Renaming a Data Member” on page 5-17.

Example

// Original class definition
// Persistent-capable class D
class D : public ooObj {
public:
    char *aName;
    float32 anArrayOfFloats[20];
ooRef(A) assocToA : copy(delete); // unidirectional
    inline ooShortRef(C) assocToC : copy(delete);
};

// Modified class definition
// Renaming an association
class D : public ooObj {
public:
    char *aName;
    float32 anArrayOfFloats[20];
    #pragma oorename assocToA
Delta ooRef(A) assocToClassA : copy(delete); // renamed
    inline ooShortRef(C) assocToC : copy(delete);
};
Changing the Access Control of an Association

Changing the access control of an association is a non-conversion operation. See “Changing the Access Control of a Data Member” on page 5-22.

---

Example

```cpp
// Original class definition
// Persistent-capable class D
class D : public ooObj {
public:
    char *aName;
    float32 anArrayOfFloats[20];
    ooRef(A) assocToA : copy(delete); // unidirectional
    inline ooShortRef(C) assocToC : copy(delete);
};

// Modified class definition
// Changing the access control of an association
class D : public ooObj {
public:
    char *aName;
    float32 anArrayOfFloats[20];
    ooRef(A) assocToA : copy(delete);

    + private:
        inline ooShortRef(C) assocToC : copy(delete);
};
```
Changing an Array Size for an Array of Object References

Changing an array size for an array of object references is a conversion operation. See “Changing the Size of a Fixed-Length Array Data Member” on page 5-24.

Example

```c
// Original class definition
class D;
// Persistent-capable class C
class C : public ooObj {
  public:
    float32 aFloat32;
    int32 anArrayOfInts[20];
    ooRef(D) anArrayOfRefs[20];
    ooRef(B) refToB; // object reference to B
};

// Modified class definition
class D;
// Change size of array of object references
class C : public ooObj {
  public:
    float32 aFloat32;
    int32 anArrayOfInts[20];
    Δ ooRef(D) anArrayOfRefs[500]; // changed from 20 to 500
    ooRef(B) refToB; // object reference to B
};
```
Changing the Order (Position) of a Non-Inline Association

Changing the order (position) of a non-inline association is a non-conversion operation. See “Changing the Order (Position) of a Data Member” on page 5-21.

Example

```c++
// Original class definition
// Persistent-capable class D
class D : public ooObj {
public:
    char *aName;
    float32 anArrayOfFloats[20];
    ooRef(A) assocToA : copy(delete); // unidirectional
    inline ooShortRef(C) assocToC : copy(delete);
};

// Modified class definition
// Change order of a non-inline association
class D : public ooObj {
public:
    ooRef(A) assocToA : copy(delete);
    char *aName;
    float32 anArrayOfFloats[20];
    inline ooShortRef(C) assocToC : copy(delete);
};
```
Changing the Order (Position) of an Inline Association or Object Reference

Changing the order (position) of an inline association or object reference is a conversion operation. See “Changing the Order (Position) of a Data Member” on page 5-21.

Example

```c++
// Original class definition
// Persistent-capable class D
class D : public ooObj {
public:
    char *aName;
    float32 anArrayOfFloats[20];
    ooRef(A) assocToA : copy(delete); // unidirectional
    inline ooShortRef(C) assocToC : copy(delete);
};

// Modified class definition
// Change order of an inline association
class D : public ooObj {
public:
    inline ooShortRef(C) assocToC : copy(delete);
    char *aName;
    float32 anArrayOfFloats[20];
    ooRef(A) assocToA : copy(delete);
};
```
Complex Class Content Operations

You can make more complex class content changes by combining multiple basic operations. These complex operations include:

- Adding a data member and copy its value from another member
- Changing the type of a data member to either a base class or derived class
- Changing the class of origin of a data member
- Changing the dimensions of an array data member
- Changing a fixed-size array to a variable-size one
- Changing the type of a non-primitive data member
- Changing an object reference to an association
- Changing the cardinality of an association
- Changing the domain class of an association

Adding a Data Member and Copying Its Value

To adding a data member and copy its value from another member, create a special application to set the new data member based on an existing value. Note that if the added data member is in the same class as the data member from which values will be copied, you can instead create a user-defined conversion function to copy the values.

1. Add the new data member in the class definition in the DDL schema file. See “Adding a Data Member” on page 5-16.
2. Create and run an application that iterates over all objects of the changed class and its derived classes. Each object returned by the iterator should get the computed value for the added data member.

Replacing a User-Defined Type with Another Related Type

To replace a user-defined type with another related type, create a special application and perform schema evolution twice. The new type can be either derived from the old type or a base class of it.

1. Add a data member with the new type to the class definition in the DDL schema file. See “Adding a Data Member” on page 5-16.
2. Create and run an application to initialize the values of the new data members based on the values of the old data members.
3. Delete the old data member from the class definition in the DDL schema file. See “Deleting a Data Member” on page 5-15.

**Example**

In this example, we want to replace data member `B::x` in class `A` with a related type data member `C::y`. To do this, we:

1. Add a data member with type `C::y` to the class `A` definition.
2. Create and run an application to set the values of the `C::y` data members based on the values of the existing `B::x` data members.
3. Delete the `B::x` data member from the class `A` definition.

![Diagram](image.png)
Changing the Origin Class of a Data Member to a Base Class

To change the origin class of a data member to a base class, perform the following steps.

1. Add a data member to the target class definition and initialize it using the `ookefault` pragma. See “Adding a Data Member” on page 5-16.
2. Create and run an application that iterates over every object of the target class, and initializes the values of new data members based on the values of the old data members.
3. Delete the old data member from the class definition in the DDL schema file. See “Deleting a Data Member” on page 5-15.

Example

In this example, we want to change the origin of data member `x` from class `B` to class `A`. To do this, we:

1. Add a data member `x` to class `A`.
2. Run an application to set the values of `A::x` from the values of the `B::x`.
3. Delete the `B::x` data member.
Figure 5-3 Changing the Origin Class of a Data Member to a Base Class
Changing the Origin Class of a Data Member to a Derived Class

To change the origin class of a data member to a derived class, perform the following steps.

1. Add a data member to the target class definition and initialize it using the oodefault pragma. See “Adding a Data Member” on page 5-16.
2. Create and run an application that iterates over every object of the target class, and initializes the values of new data members based on the values of the old data members.
3. Delete the old data member from the class definition in the DDL schema file. See “Deleting a Data Member” on page 5-15.

Example

In this example, we want to change the origin of data member \( x \) from class \( A \) to class \( B \). To do this, we:

1. Add a data member \( x \) to class \( B \).
2. Run an application to set the values of \( B:x \) to the values of \( A:x \).
3. Delete the \( A:x \) data member.
Figure 5-4  Changing the Origin Class of a Data Member to a Derived Class

Application sets the value of $B::x$ based on $A::x$. The application sets the value of $B::x$ based on $A::x$. After the change, $A::x$ is deleted.
Changing the Number of Dimensions of a Fixed-Size Array Data Member

To change the number of dimensions of a fixed-size array, perform the following steps:

1. Add a new fixed array data member with new dimensions. See “Adding a Data Member” on page 5-16.
2. Create and run an application that iterates over every object of the modified class, setting the value of the added data member from the existing one.
3. Delete the old fixed array data member from the class definition in DDL schema file. See “Deleting a Data Member” on page 5-15.

Changing a Fixed-Size Array to a Variable-Size Array

To change a fixed-size array to a variable-size array, perform the following steps:

1. Add an ooVArray data member. See “Adding a Data Member” on page 5-16.
2. Create and run an application that iterates over every object of the modified class (and its derived classes), setting the value of the added data member from the existing one.
3. Delete the old fixed array data member from the class definition in the DDL schema file. See “Deleting a Data Member” on page 5-15.

Changing the Type of a Non-Primitive Data Member

To change the type of a non-primitive data member, perform the following steps:

1. Add a data member of the new type. See “Adding a Data Member” on page 5-16.
2. Create and run an application that iterates over every object of the modified class (and its derived classes), setting the value of the added data member from the existing one.
3. Delete the old data member from the class definition in the DDL schema file. See “Deleting a Data Member” on page 5-15.
Changing an Object Reference to an Association

To change an object reference to an association, perform the following steps.

1. Add an association to the class definition. See “Adding an Inline Association” on page 5-26 or “Adding a Non-Inline Association” on page 5-27.
2. Create and run an application that iterates over every object of the modified class, setting the values of the new associations based on the existing object references.
3. Delete the old object reference from the class definition in the DDL schema file. See “Deleting a Data Member” on page 5-15.

Changing an Association to an Object Reference

To change an association to an object reference, perform the following steps.

1. Add an object reference to the class definition. See “Adding an Inline Association” on page 5-26 or “Adding a Non-Inline Association” on page 5-27.
2. Create and run an application that iterates over every object of the modified class initializing the values of the new object references based on the existing associations.
3. Delete the old association from the class definition in the DDL schema file. See “Deleting a Data Member” on page 5-15.

Changing Association Cardinality

To changing the cardinality of an association, perform the following steps.

1. Add an association with new cardinality to the class definition. See “Adding an Inline Association” on page 5-26 or “Adding a Non-Inline Association” on page 5-27.
2. Create and run an application that iterates over every object of the modified class initializing the values of the new associations based on the existing associations.
3. Delete the old association from the class definition in the DDL schema file. See “Deleting a Data Member” on page 5-15.
Changing the Class where a Unidirectional Association Originates

As shown in Figure 5-5, you can change the class where a unidirectional association originates. Note that changing the domain of a bidirectional association also changes its origin (see “Change the Domain Class of an Association” on page 5-52).

![Diagram of changing a unidirectional association's origin](image)

**Figure 5-5 Changing a Unidirectional Association's Origin**

1. Add an association to the class definition of the class that is to be the new origin for the association. See “Adding an Inline Association” on page 5-26 or “Adding a Non-Inline Association” on page 5-27.
2. Create and run an application that iterates over every object of the modified class setting the values of the new associations based on the existing associations.
3. Delete the old association from the class definition in the DDL schema file. See “Deleting a Data Member” on page 5-15.
Change the Domain Class of an Association

To change the class an association points to (its domain) perform the following steps. Note that changing the domain of a bidirectional association also changes its origin. Conversely, changing the origin of a bidirectional association also changes its domain.

As shown in Figure 5-6, you can change the class where a unidirectional association originates.

1. Add an association to the class definition. See “Adding an Inline Association” on page 5-26 or “Adding a Non-Inline Association” on page 5-27.
2. Create and run an application to set the values of the new associations based on the existing associations.
3. Delete the old association from the class definition in the DDL schema file. See “Deleting a Data Member” on page 5-15.
Basic Class Operations

This section describes operations you can perform to change classes in your schema. These operations include:

◆ Renaming a class
◆ Deleting a class

Initial Schema Used for Basic Class Examples

The examples presented in this section are based on the schema shown in “Initial Schema Used for Class Content Examples” on page 5-13. Each example shows the original and modified class definitions for the documented operation.

Renaming a Class

Renaming a class is a non-conversion operation. This operation changes the class definition in the federated database schema without creating a new representation of the class. The new class name must be unique. That is, it should not be the name of an existing class in the federated database schema before the schema evolution.

1. Change the class name in the target class definition.
2. Specify the following pragma to indicate the existing class name. You must define this pragma immediately before the class declaration.

```c
#pragma oorename existingName
```

where

existingName Existing class name

3. If the changed class is a persistent-capable class, run the DDL processor on the affected class definition.
4. Modify your any existing applications that reference the renamed class by its old name. Using the newly-generated DDL header and C++ source files, rebuild your applications that reference the changed class by name.
5. To reuse your DDL schema files, remove the oorename pragma statements.
// Original class definition
// Persistent-capable class A
class A : public ooObj {
public:
  uint8 aUint8;
  float64 aFloat64;
  char aChar;
};

// Modified class definition
// Renaming class A to newClassA
+ #pragma oorename A
Δ class newClassA : public ooObj {
public:
  uint8 aUint8;
  float64 aFloat64;
  char aChar;
};
Deleting a Class

Deleting a class (see Figure 5-7) is a conversion operation if other classes have object references or associations to objects of the deleted class. Otherwise, it is a non-conversion operation. This operation marks the existing class representation as a deleted class. You can reuse the name of the deleted class in a subsequent run of the DDL processor.

This operation requires you to modify and rebuild existing applications that use the deleted class.

This operation requires an upgrade application if the deleted class is a derived class from a class B, where B is indirectly referenced through an association or object reference. See “Deleting a Class with Inherited Associations or Object References” on page 5-58.

![Figure 5-7 Deleting a Class](image-url)
1. Remove the definition of the class to be deleted from the DDL schema file. Be sure there are no object references or associations to the deleted class, and that the class is not used as the type of a data member.

2. Add the following class declaration and pragma to the DDL schema file to specify the class to be deleted. You can locate this pragma anywhere in the DDL schema file. You must forward declare the class before the pragma.

```c
class className;
#pragma oodelete className
```

where

```c
className                Name of class to be deleted
```

3. Before running the DDL processor on the new schema, modify your application files to:
   - Unname any object of the deleted class. See “Removing an Object’s Scope Name” on page 8-7 in Using Objectivity/C++.
   - Remove any usage of an object of the deleted class as a named scope.

   This is a special step when deleting a class. It prevents your application from using one of the deleted objects as a name scope before the object is accessed and deleted by Objectivity/C++.

4. If the deleted class is a persistent-capable class, run the DDL processor using the `-evolve` flag on the affected class definition and its derived classes.
   If the deleted class contains inherited associations or object references, run the DDL processor using the `-evolve` and `-upgrade` flags. (See “Deleting a Class with Inherited Associations or Object References” on page 5-58).

   If the deleted class is a non-persistent-capable class, run the DDL processor on the affected class, its derived classes, and the classes that embed it.

5. Rebuild your existing applications using the newly-generated header and C++ source files.

6. To reuse your DDL schema files, remove the `oodelete` pragma statements.
Example

// Original class definition
// Persistent-capable class A
class A : public ooObj {
public:
    uint8 aUint8;
    float64 aFloat64;
    char aChar;
};
...

// Persistent-capable class D
class D : public ooObj {
public:
    char *aName;
    float32 anArrayOfFloats[20];
    ooRef(A) assocToA : copy(delete); // unidirectional
    inline ooShortRef(C) assocToC : copy(delete);
};

// Modified class definition
// Deleting persistent-capable class A and references to it
class A;

#pragma oodelete A

// Persistent-capable class D
class D : public ooObj {
public:
    char *aName;
    float32 anArrayOfFloats[20];
    ooRef(A) assocToA : copy(delete); // unidirectional
    inline ooShortRef(C) assocToC : copy(delete);
};
Complex Class Operations

This section describes complex operations you can perform to change your schema. Most of these operations require you to perform schema evolution and object conversion more than once. These operations include:

- Deleting a class with inherited associations or object references

Deleting a Class with Inherited Associations or Object References

If you delete a class that has inherited associations or object references from another class (as shown in Figure 5-8), you must create and run a single-use upgrade application to convert existing objects. In the example in Figure 5-8, accessing an object of class X or any of its derived classes before running the upgrade application results in an error.

![Figure 5-8 Inherited Non-Inline Association](image)

Figure 5-8  Inherited Non-Inline Association

Before

After

Simple upgrade application deletes the association automatically
1. Delete the class as described in “Deleting a Class” on page 5-55. When running the DDL processor, you must call the -evolve and -upgrade flags.

2. Create and run a simple, single-use upgrade application to automatically delete inherited object references and associations. This upgrade application does not require any special user code to perform the upgrade. All that is required is to call the upgrade interface member functions ooTrans::upgrade and ooRefHandle<ooFDObj>::upgradeObjects. For more information, see the “Object Conversion” chapter in Using Objectivity/C++.

3. Rebuild your existing applications using the newly-generated header and C++ source files.
Basic Inheritance Operations

This section describes operations you can perform to change inheritance relationships between a class and its base classes in your schema. These operations include:

- Adding a non-persistent-capable class as a base class
- Removing a non-persistent-capable class as a base class
- Replacing a base class a subset of its base classes
- Replacing a base class with one of its derived classes
- Changing the order (position) of a base class
- Changing the access control of a base class

Adding a Non-Persistent-Capable Class as a Base Class

Adding a non-persistent-capable class as a base class is a conversion operation. With one schema evolution iteration, you can add a non-persistent-capable class to the base class list of another class (see Figure 5-9).

![Diagram showing before and after adding a non-persistent-capable class as a base class](image)

Figure 5-9 Adding a Non-Persistent-Capable Class as a Base Class

5-60  Using Objectivity/C++ Data Definition Language
1. Add the new base class name to the base class list of the target class.
2. Process your DDL schema files by running the DDL processor with the 
   --evolve flag.
3. Modify and rebuild your existing applications using the newly-generated 
   header and C++ source files.

___

**Example**

// Original class definition
// Persistent-capable class B
class B : public A {
  public:
    uint16 aUint16;
    float32 aFloat32;
};

// Modified class definition
// Adding non-persistent-capable class X as a base class
// to persistent-capable class B
\[class B : public A, X \{ \]
  public:
    uint16 aUint16;
    float32 aFloat32;
};

___
Removing a Non-Persistent-Capable Class as a Base Class

Removing a non-persistent-capable class as a base class is a conversion operation. You can remove a non-persistent-capable class from the base class list of a class (see Figure 5-10).

Figure 5-10 Removing a Class from a Base Class List

1. Remove the target leaf base class name from the base class list of the derived class.
2. Process your DDL schema files by running the DDL processor with the -evolve flag.
3. Modify and rebuild your existing applications using the newly-generated header and C++ source files.
Example

// Original class definition
// Persistent-capable class A
class A : public ooObj {
public:
    uint8 aUint8;
    float64 aFloat64;
    char aChar;
};

// Persistent-capable class B
class B : public A, X {
public:
    uint16 aUint16;
    float32 aFloat32;
};

// Non-persistent-capable class X
class X {
public:
    uint8 aUint8;
    char *aString;
};

// Non-persistent-capable class Y
class Y : X {
public:
    float64 aFloat64;
};
// Modified class definition
// Removing a non-persistent-capable class X as a base class
// to persistent-capable class B
class B : public A {
public:
    uint16 aUint16;
    float32 aFloat32;
};

Replacing a Base Class with a Subset of Its Base Classes

Replacing a base class with a subset of its base classes is a conversion operation that replaces a base class with a subset of its base classes. For example, in Figure 5-11, this operation replaces class Y, which is the base class of class B, with class Y's own base class, class X. You can use this operation to remove a class from the middle of an inheritance hierarchy. You must use an upgrade application to convert objects after this operation.

Figure 5-11 Changing Class B by Replacing Its Base Class Y with a Subset of Base Class Y's Base Classes
1. Add the following pragma to the DDL schema file. You must locate this pragma immediately before the declarations of the base classes.

```plaintext
#pragma oochangebase existingBaseClass -> newBaseClass [,moreNewBaseClasses]*
```

where

- `existingBaseClass`: Existing base class name
- `newBaseClass`: New base class name. This must be a base class of `existingBaseClass`
- `moreNewBaseClasses`: Additional new base class names. These must be base classes of `existingBaseClass`.

2. Process your DDL schema files by running the DDL processor using the `-evolve` and `-upgrade` flags.

3. Create and run an upgrade application to convert existing objects. This application must call the upgrade interface member functions `ooTrans::upgrade` and `ooRefHandle(ooFDObj)::upgradeObjects`. See the “Object Conversion” chapter in Using Objectivity/ C++.

4. Modify and rebuild your existing applications using the newly-generated header and C++ source files.

5. To reuse your DDL schema files, remove the `oochangebase` pragma statements.

---

Example

```plaintext
// Original class definitions
// Persistent-capable class A
class A : public ooObj {
public:
  uint8 aUint8;
  float64 aFloat64;
  char aChar;
};
```

---

Schema Evolution 5-65
// Persistent-capable class B
class B : public A, Y {
public:
    uint16 aUint16;
    float32 aFloat32;
};

// Non-persistent-capable class X
class X {
public:
    uint8 aUint8;
    char *aString;
};

// Non-persistent-capable class Y
class Y : X {
public:
    float64 aFloat64;
};

// Modified class definitions
// Persistent-capable class A
class A : public ooObj {
public:
    uint8 aUint8;
    float64 aFloat64;
    char aChar;
};

// Persistent-capable class B
+ #pragma oochangebase Y -> X
Δ class B : public A, X {
public:
    uint16 aUint16;
    float32 aFloat32;
};
// Non-persistent-capable class X
class X {
public:
    uint8 aUint8;
    char *aString;
};

// Non-persistent-capable class Y
class Y : X {
public:
    float64 aFloat64;
};
Replacing a Base Class with One of Its Derived Classes

Replacing a base class with one of its derived classes is a conversion operation that replaces a base class with one of the derived classes of the base class. For example, in Figure 5-12, Class X, the base class of class B, is replaced with class X’s derived class, class Y. You can use this operation to add a class in the middle of an inheritance hierarchy, or to change a leaf class to be a non-leaf class. You must use an upgrade application to convert objects after this operation.

1. Add the following pragma to the DDL schema file. You must locate this pragma immediately before the declarations of the base classes.

```
#pragma oochangebase existingBaseClass  -> newBaseClass
```

where

- `existingBaseClass` Existing base class name
- `newBaseClass` New base class name. This must be a derived class of `existingBaseClass`.

2. Process your DDL schema files by running the DDL processor with the `-evolve` and `-upgrade` flags.
3. Create and run an upgrade application to convert existing objects. This application must call the upgrade interface member functions `ooTrans::upgrade` and `ooRefHandle(ooFDObj)::upgradeObjects`. See the "Object Conversion" chapter in Using Objectivity/C++.

4. Modify and rebuild your existing applications using the newly-generated header and C++ source files.

5. To reuse your DDL schema files, remove the `oochangebase` pragma statements.

---

**Example**

```c++
// Original class definitions
// Persistent-capable class A
class A : public ooObj {
public:
  uint8 aUint8;
  float64 aFloat64;
  char aChar;
};

// Persistent-capable class B
class B : public A, X {
public:
  uint16 aUint16;
  float32 aFloat32;
};

// Persistent-capable class C
class C : public ooObj, X, Z{
public:
  float64 aFloat64;
  ooRef(B) refToB; // object reference to B
};
```
// Non-persistent-capable class X
class X {
public:
    uint8 aUint8;
    char *aString;
};

// Non-persistent-capable class Y
class Y : public X {
public:
    float64 aFloat64;
};

// Non-persistent-capable class Z
class Z : public X, Y {
public:
    char *aName;
};

// Modified class definitions
// Persistent-capable class A
class A : public ooObj {
public:
    uint8 aUint8;
    float64 aFloat64;
    char aChar;
};

// Persistent-capable class B
#pragma oochangebase X -> Y
class B : public A, Y {
public:
    uint16 aUint16;
    float32 aFloat32;
};
// Non-persistent-capable class X
class X {
    public:
        uint8 aUint8;
        char *aString;
};

// Non-persistent-capable class Y
class Y : X {
    public:
        float64 aFloat64;
};

### Changing the Order (Position) of a Base Class

Changing the order (position) of a base class is a conversion operation. See “Changing the Order (Position) of a Data Member” on page 5-21

---

#### Example

// Original class definition
// Persistent-capable class A
class A : public ooObj {
    public:
        uint8 aUint8;
        float64 aFloat64;
        char aChar;
};
// Persistent-capable class B
class B : public A, X {
    public:
        uint16 aUint16;
        float32 aFloat32;
};

// Persistent-capable class C
class C : public ooObj, X, Z{
    public:
        float64 aFloat64;
        ooRef(B) refToB; // object reference to B
};

// Non-persistent-capable class X
class X {
    public:
        uint8 aUint8;
        char *aString;
};

// Non-persistent-capable class Y
class Y : public X {
    public:
        float64 aFloat64;
};

// Non-persistent-capable class Z
class Z : public X, Y {
    public:
        char *aName;
};
// Modified class definition
// Changing the order of base class Z in class C
class C : public ooObj, Z, X{
    public:
        float64 aFloat64;
        ooRef(B) refToB; // object reference to B
};

---

**Changing the Access Control of a Base Class**

Changing the access control of a base class is a non-conversion operation. See “Changing the Access Control of a Data Member” on page 5-22

---

**Example**

// Original class definition
// Persistent-capable class A
class A : public ooObj {
    public:
        uint8 aUint8;
        float64 aFloat64;
        char aChar;
};

// Persistent-capable class B
class B : public A, X {
    public:
        uint16 aUint16;
        float32 aFloat32;
};
// Persistent-capable class C
class C : public ooObj, X, Z{
  public:
    float64 aFloat64;
    ooRef(B) refToB; // object reference to B
};

// Non-persistent-capable class X
class X {
  public:
    uint8 aUint8;
    char *aString;
};

// Non-persistent-capable class Y
class Y : public X {
  public:
    float64 aFloat64;
};

// Non-persistent-capable class Z
class Z : public X, Y {
  public:
    char *aName;
};

// Modified class definition
// Changing the access control of base class Z in class C
class C : public ooObj, X, private Z {
  public:
    float64 aFloat64;
    ooRef(B) refToB; // object reference to B
};
Complex Inheritance Operations

This section describes complex operations you can perform to change your schema. Most of these operations require you to perform schema evolution and object conversion more than once. These operations include:

- Adding a non-leaf base class
- Deleting a non-leaf base class
- Changing a non-persistent capable class to become a persistent capable class
- Changing a persistent capable class to become a non-persistent capable class

Adding a Non-Leaf Base Class

Perform the following steps to add a non-leaf base class to an existing class hierarchy (see Figure 5-13). This procedure requires that you perform schema evolution twice.

1. Add the new class definition and make it inherit from the target base class.
2. Process your DDL schema files by running the DDL processor with the -evolve flag.
3. Modify and rebuild your existing applications using the newly-generated header and C++ source files.
4. Create and run an upgrade application to convert existing objects. This application must call the upgrade interface member functions ooTrans::upgrade and ooRefHandle(ooFDObj)::upgradeObjects. See the “Object Conversion” chapter in Using Objectivity/ C++.
5. Replace the target base class with one of its derived classes. See “Replacing a Base Class with One of Its Derived Classes” on page 5-68. When running the DDL processor, use the -evolve and -upgrade flags.
Example

In the following example, class B is derived from class A. We want to add class X as a non-leaf base class between class A and B. To do this we:

1. Add class X and make it inherit from class A.
2. Replace the base class A in class B with the derived class X of class A.

Figure 5-13 Adding a Non-Leaf Base Class X to Class B
Deleting a Non-Leaf Base Class

To delete a non-leaf base class from an existing class hierarchy, you must first change it to be a leaf base class (see Figure 5-14 and Figure 5-14). This process requires you perform schema evolution and object conversion twice.

1. Make the target non-leaf base class a leaf base class by replacing the target base class with its derived classes. See “Replacing a Base Class with One of Its Derived Classes” on page 5-68.
2. Delete the target leaf base class from the DDL schema file. See “Deleting a Class” on page 5-55.

Example

In the following example, class B is derived from class A. We want to delete class B. To do this we:

1. Replace base class B in classes C and D by class A. This makes class B a leaf class.
2. Delete the target leaf base class B from the DDL schema file.

Figure 5-14 Deleting a Non-Leaf Class B (First Schema Evolution)
Making a Non-Persistent-Capable Class Persistent-Capable

To make a non-persistent-capable class persistent-capable, perform the following steps. The target non-persistent-capable class must not be the type of any embedded data member. That is, the class and its derived classes must not affect any persistent object in the database. This procedure requires you to perform schema evolution twice.

1. Delete the target non-persistent class definition from the DDL schema file. See “Deleting a Class” on page 5-55.
2. Process your DDL schema files by running the DDL processor with the -evolve flag.
3. Modify and rebuild your existing applications using the newly-generated header and C++ source files.
4. Add the target class definition to the DDL schema file with ooObj as its base class.
5. Process your DDL schema files by running the DDL processor with the -evolve flag.
6. Modify and rebuild your existing applications using the newly-generated header and C++ source files.
Making a Persistent-Capable Class Non-Persistent-Capable

To make a persistent-capable class non-persistent-capable, perform the following steps. This procedure requires you to perform schema evolution twice.

1. Delete the target persistent class definition from the DDL schema file. See “Deleting a Class” on page 5-55.
2. Add the target class definition to the DDL schema file without ooObj as its base class.
3. Process your DDL schema files by running the DDL processor with the -evolve flag.
4. Modify and rebuild your existing applications using the newly-generated header and C++ source files.

Multiple Class Operations

This section describes how to perform multiple class operations. Most of these operations require you to perform schema evolution and object conversion more than once. These operations include:

- Splitting a class into two associated classes
- Splitting a class into two classes where one class is derived from the other
- Merging two associated classes
- Merging two classes where one class is derived from the other

Splitting a Class into Two Associated Classes

You can perform multiple class operations to split a class into two classes with associations between them. For example, to split a class A into two associated classes A' and B, perform the following steps. This procedure refers to the class names and relationships shown in Figure 5-16:

1. Add a new class. In this example, class B.
2. Add an association from the existing class to the new class. In this example, add the association to class B from class A'.
3. Write an application that:
   a. Creates a new object of class B for each object of class A’ and sets the values of the new objects from those of A’.
   b. Set associations between objects of these classes.

4. Change class A’ to A'' by deleting the data members that were moved to class B. See “Deleting a Data Member” on page 5-15.

![Figure 5-16 Splitting a Class into Two Associated Classes](image)

Figure 5-16 Splitting a Class into Two Associated Classes
Splitting a Class So That One Class is Derived from the Other

You can perform multiple class operations to split a class into two classes such that one class is derived from the other. To split a class A into two classes B and A such that class B is a base class of A, perform the following steps. This procedure refers to the class names and relationships shown in Figure 5-17:

1. Add a new class. In this example, class B.
2. Change class A by replacing its base class ooObj by class B.
3. Write an application that iterates over objects of class A', and sets the values of the inherited data members from class B.
4. Change class A' by deleting the data members that were moved to class B. See “Deleting a Data Member” on page 5-15.

Figure 5-17 Splitting a Class into Two Classes Such That One Class is Derived from the Other
Merging Two Associated Classes

You can perform multiple class operations to merge two associated classes. To merge two associated classes A and B, perform the following steps. This procedure refers to the class names and relationships shown in Figure 5-18:

1. Change class A by adding the data members of class B. In this example, we change class A by adding data members of class B.
2. Write an application that iterates over objects of class A to initialize the data members added in step 1.
3. Change the schema by deleting class B and its association to class A. See “Deleting a Class” on page 5-55.

![Figure 5-18 Merging Two Associated Classes](image-url)
Merging a Class with its Derived Class

You can perform multiple class operations to merge two classes where one class is derived from the other. To merge two classes \( A \) and \( B \) where class \( B \) is a base class of class \( A \), perform the following steps. This procedure refers to the class names and relationships shown in Figure 5-19:

1. Change one class by adding the data members of the other. In this example, we change class \( A \) by adding data members of class \( B \).
2. Write an application that:
   a. Iterates over objects of class \( A \) to set the data members added in step 1.
   b. Iterates over objects of class \( B \) and creates equivalent objects of class \( A \).
   c. Deletes objects of class \( B \).
3. Change the base class of class \( A \) such that class \( B \) becomes a leaf class.
4. Delete class \( B \). See “Deleting a Class” on page 5-55.

![Figure 5-19 Merging Two classes Where One Class is Derived from the Other](image-url)
Schema Evolution Example

This section includes examples of schema evolution operations.

Example

This example is based on changing a course enrollment schema shown in Figure 5-20. This schema contains the following user-defined classes—Project, Student, Course, and GradStudent.
class Project: public ooObj {
  ooVString _projectName;
  public:
    ooRef(Student) assigned [] <-> responsibleStudents[];
    Project (const char * projectName);
};

class Course: public ooObj {
  ooVString _courseTitle;
  public:
    Course (const char *title);
};

class Student: public ooObj {
  ooVString name;
  ooVString _major;
  float _GPA;
  uint8_ssn [11];
  uint16 _yearAdmitted;
  public:
    ooRef(Project) responsibleStudents [] <-> assigned[];
    ooRef(Student) friends [] <-> friends []
    Student ();
    Student (const char *name);
    char *getAll ();
};

class gradStudent: public Student {
  float _undergradGPA;
  ooVString _undergraduateMajor;
};
Example

The following example illustrates simple operations on data member and associations.

The following is the DDL representation for a set of schema evolution operations.

```c++
class Project: public ooObj {
    ooVString _projectName;
    public:
        ooRef(Student) assigned [] <-> responsibleStudents[];
        // add a new unidirectional inline association.
        inline ooShortRef(Course) assignedFor;
    Project (const char *projectName);
};

class Course: public ooObj {
    ooVString _courseTitle;
    // Add a data member with user defined values
    #pragma oodefault 3
    uint16 _numberCredits;
    public:
        Course (const char *title);
};
```
class Student: public ooObj {
    ooVString name;
    ooVString _major;
    float32 _GPA;
    + // Change the size of an array
    △ uint8 _ssn [9];
    uint16 _yearAdmitted;
    public:
        ooRef(Project) responsibleStudents [] <-> assigned[];
        + // delete a bidirectional association friends[]<->
        + // friends[];
        - ooRef(Student) friends [] <-> friends []
        Student ();
        Student (const char *name);
        char *getAll ();
};

#pragma oorename gradStudent
class GradStudent: public Student {
    float32 _undergradGPA;
    + // renaming a data member
    + #pragma oorename _undergradMajor
    ooVString _undergraduateMajor;
};
The following example creates a non-persistent-capable class (Address) that is used as a data member in the class Student. In this example, only a subset of the classes are processed by the DDL processor.

```
+ // new class
+ class Address {
+     ooVString _streetName;
+     ooVString _city;
+     char state [2];
+     uint8 zipCode [5];
+ };
```

**Figure 5-21**  Course Enrollment Example Schema with Address Class

---

**Example**

---

5-88  Using Objectivity/C++ Data Definition Language
class Course : public ooObj {
    ooVString _courseTitle;
    uint16 _numberCredits;
    public:
        ooRef(TeachingAssist) assistants [] <-> assistIn[];
        Course (const char *title);
};

+ // another new class
+ class Person {
+    ooVString _name;
+    uint8 _ssn [9];
+};
+ // Adding a base class Person
Δ class Student: public ooObj, public Person {
+    // delete the _name and _ssn data members.
-    ooVString name;
-    ooVString _major;
-    float32 _GPA;
-    uint8_ssn [9];
-    uint16 _yearAdmitted;

Address _homeAddress;
public:
    ooRef(Project) responsibleStudents [] <-> assigned[];
    Student ()
    Student (const char *name);
    char *getAll ()
};

+ // GradStudent has to be included as a side effect of
+ // changing its base class.
+ class TeachingAssist: public Student {
+    ooRef(Course) assistIn [] <-> assistants [];
+};
class GradStudent: public Student {
    float32 _undergradGPA;
    ooVString _undergraduateMajor;
};
Class Versioning

This chapter describes the class versioning feature, which allows you to create multiple versions of a class in a schema.

Basics

Class versioning is different from schema evolution described in the “Schema Evolution” chapter. Using schema evolution, you can evolve existing class definitions. It does not use multiple class versions to store schema changes. In contrast, class versioning does create and maintain separate class versions. Class versioning allows you to:

- Maintain objects of different class versions simultaneously in a federated database
- Access objects of different class versions from the same application, using the member functions defined for each class version

Since class versioning creates separate versions of classes, you can use schema evolution on a versioned class, just like any other class.

When you create a class for the first time, the DDL processor assigns a version identifier of 1 to the class definition. When you create a new version of the class the version identifier is incremented. That is, the version identifier of the second definition is 2, the version identifier of the third definition is 3, and so on.

You can create a new version of an existing class simply by changing the class definition in a DDL schema file and running the DDL processor using the --version flag, which automatically produces changes in the schema.
Example

The following DDL schema file (eng.ddl) creates a new version of the Widget class, adding the partLength attribute:

class Widget : public ooObj {
public:
    float partDiameter;
    float threadSpacing;
    void partCount();
    float partLength; // assume this is added member
};

The following ooddlx command updates to the default schema in the federated database specified in the boot file ABCinc.boot. The DDL processor creates a new version of the Widget class in the default schema as shown in Figure 6-1:

ooddlx -version eng.ddl ABCinc.boot

Figure 6-1 Creating Class WidgetA as a Version of Class Widget
Creating and Using New Class Versions

Perform the following steps to create and use new class versions. In this process, you:

- Produce header files for both the old and new versions of the schema
- Write a program that can use old and new class versions to:
  - Create new objects based on the new class version
  - Copy object information from objects based on the old class version to objects based on the new class version

1. Make a backup copy and two working copies of your original DDL schema file. You will use one working copy for old class versions, and the other working copy to define new versions.

2. Modify the old class version DDL schema file using the ooclassname pragma to change the names of all classes that will also appear in the new DDL schema file. You will probably want to give this file a name that indicates it is a prior version. For example, if the old class name is Widget, you may want to rename it to OldWidget.

3. Modify the new class version DDL schema file by changing the class definitions for the classes you want to version.

4. Use the DDL processor to obtain header (.h) and source (_ddl .c) files for the old class versions.

5. Run the DDL processor with the -nochange flag to test your new class version DDL schema file. It should report comparison errors on all the classes you have changed. (Because of the way it tries to synchronize new versus existing class descriptions, it may not report all differences. If it reports any change to a class, the next step will produce a new version of the class.)

6. Run the DDL processor with the -version flag to create new versions of the appropriate classes as well as new header and source files.

7. Write a program to scan your federated database for all instances of the old class versions and references to the old class versions. For each instance:
   a. Create a corresponding object of the new version.
   b. Make the new version equivalent to the old by assigning initial values, copying associations, and so on, as needed. For each association or object reference, modify it to point to the new object.
   c. If desired, have the application delete the old objects.
   d. For hints on how to copy old object information to new objects, see “Class Versioning Hints” on page 6-6.
8. Build the program. You will have to compile and link both old and new versions of the generated source files. (Compiling both and then link them into a single executable without duplicate class definitions is indication that the class versioning process is working properly.)

9. Execute the program. If possible, run it on a small sample federated database before running it on your master system. Be sure to verify the results.

10. Give the new versions of the header and source files to other application developers and have them rebuild their applications.

Using #pragma ooclassname

In C++, each class (including each union and each struct) has global scope. That is, a given program module can contain only one class of a given name. Without this restriction, static class members would be impossible under most current compilation schemes. This restriction is a handicap when working with class versioning. C++ must allow us access to two versions of the same class when both versions have the same class name. To do so, we rename all old-version classes using a special DDL pragma:

- #pragma ooclassname oldClassName newClassName [public]

When you use #pragma ooclassname in a DDL schema file, you make no changes to your original (persistent) class definitions. Instead, you add lines to the original DDL schema file that specify what classes to rename and how. You then send the file through the DDL processor again. The resulting header (.h) and source (_ddl.C) files reflect the changed names, but your schema remains unchanged.

You must use #pragma ooclassname on all classes defined in a given DDL schema file, even if one or more classes in that file are not being changed. This is because the DDL processor produces initialization code in the source (.C) file that has global scope. Unless there are interdependencies between the versioned and non-versioned classes, you can separate them to cut down on the number of renamed classes.

The optional public keyword on the #pragma ooclassname pragma is provided to make a class definition completely public. When you use it, all the members of the renamed class are forced to public access in the generated header file, which may simplify your conversion program.
Why Use #pragma ooclassname?

You can rename a class by modifying the generated header and source files. However, the process is complicated and error-prone, especially since you must also rename all the Objectivity/DB-defined classes that are automatically derived for each persistent-capable class (for example, ooHandle(widget)).

Where to Place #pragma ooclassname

You should place each #pragma ooclassname pragma after its class' definition.

Using #pragma ooclassname with New Versions of a Class

You can use the #pragma ooclassname pragma to change the application program's view of the older class versions. You can also use it to modify the names of the newer versions. In general, this is the simpler and more consistent choice. The new header and source files use the actual class names and also reflect the latest versions, so they can also be used with general application programs.

However, if you have a body of code that you want to use in your conversion program that could more easily work with original older versions and renamed newer ones, you can use the #pragma ooclassname pragma in the newer file, instead. In fact, you can rename both the older and the newer class versions. If you do rename newer versions, you will most likely want to remove the #pragma ooclassname pragmas from the DDL schema file and produce header and source files before using these generated files in an application program.

When to Use the public Option of #pragma ooclassname

If your classes are heavily encapsulated, there may be no easy way to access the objects in the various persistent objects. You can edit the generated header (.h) files to make the new class versions friends of the old versions (or vice versa), or you could add sufficient access methods to allow the needed conversions. But for a one-shot program, either of those choices can be tedious and unnecessary.

The public option of the #pragma ooclassname pragma is useful when generating header files for your older class versions. (For example, you will probably have constructors that can add the needed objects in the newer versions.) However, if you do use this option to access your newer class versions, do not forget to remove it and rebuild the header file after you have completed the object copy process or your application programs will be able to break encapsulation.
Class Versioning Hints

The following hints may help you to use class versioning efficiently.

Copying Unreferenced Objects to New Class Version Objects

If you have old object class versions that are not referenced in a standard way, you may need to scan for them separately. For example, refer to the example program on page 6-13. If only some two objects were referred to from class three objects or if only some pairs of class one and class two objects were linked, then we would have to also scan (via an iterator similar to oldThreeItr) for the independent objects. We would do so after the scan for class three objects and after the referenced class one and class two objects had been updated. The additional iterators would only find the isolated objects (the old versions we had changed would have already been deleted).

Copying Bidirectional Associations to New Class Version Objects

The process of copying a bidirectional association from a versioned class is fairly easy. Assuming that you have an instance of both a new and an old class version (for example, you have a handle to a pair of corresponding objects), what you do depends on the type of the association.

◆ For to-many associations:
   1. Create an iterator for the class at the other end of the association.
   2. Initialize the iterator via a to-many association defined for the current class.
   3. Subtract each object found via the association iterator from the current association and add it to the corresponding association for the corresponding new object.

◆ For to-one associations:
   1. Try to obtain a handle to the object at the other end of the association. As far as Objectivity/DB is concerned, it is not an error if no such object exists. The definition of an association only provides for the possibility of such a relationship. Your application may, however, require a given association link. In such a case, your program could check for that link.
   2. Assuming such an object exists, delete the current association and set an association from the new object to the other end.

While the above procedures will work in virtually all circumstances, you may be doing extra work in some circumstances if you follow them literally. Your conversion program will probably be most efficient if you copy all objects in a group of related objects or a composite object, at the same time.
Copying One-Way References to Versioned Classes

Object references and unidirectional associations can all be used as one-way references to a versioned class. They all share the same problems.

If a one-way reference to a versioned class exists in a class that is itself a versioned class, the problems are similar to those of bidirectional associations. Working with groups of interrelated objects is most efficient. If at all possible, work with each object only once.

If the class that owns the one-way reference to a versioned class is not itself scheduled to be versioned, then you must scan for all possible instances of these one-way references. You will be changing the contents of non-versioned objects (that is, you will be changing what they reference) even if you are not changing their version. Again, if you can work with groups of interrelated objects your program will be more efficient.

Organizing Your DDL Schema Files

If your original DDL schema file was a single file, you can continue to use it as is. However, if possible, you will probably find it easier to work with one of more smaller DDL schema files (that have, if necessary, been taken from the original larger file). Try to keep the files that will be changed to minimal size, with definitions of only those classes that are to be changed along with any classes on which the versioned classes are dependent. Of course, any classes dependent on a versioned class (for example, classes derived from a versioned class) are themselves also being modified. These classes must also be included in the DDL schema file that will be modified. Even unchanged classes may have to go through the versioning process if they have indirect dependencies to classes that are versioned (for example, associations or object references).

If you have non-persistent-capable classes defined in your DDL schema file, and if those classes have not changed, we recommend that you move them to standard header (.h) files. You can then include these (and other unchanged) header files in both the old and new DDL schema files without modifying or renaming any of the classes they define.

Deploying a Federated Database

Before deploying your product, we strongly recommend that you create a final version of the federated database without versioned classes.
Class Versioning Example

This section presents a very simple example of how to use class versioning. The schema used here has only three user defined persistent-capable classes (one, two, and three). Two of the three classes will be versioned and all instances of those classes in the federated database will be copied to the new classes.

The general process for this example is as follows:

1. We begin by initializing Objectivity/DB, starting a transaction, and then opening the federated database, our test database, and the default container in that database.

   In our test programs, we choose to put all the objects that would need to be copied into this single container, but any database scope would work.

2. We set up the master scan loop so it is not based on a search for all objects of the classes being copied. We are using our knowledge of the way this federated database was built to make the versioning as simple as possible. For example, we know that:
   - All old two objects have a corresponding three object that refers to them (via the toTwo association).
   - All old one objects are linked to a corresponding old two object.

   Therefore, scanning for three objects and chasing the pointers works quite well.

3. We use our knowledge of the structure of the various classes. Specifically, we create new one and two objects and fill in their object fields.

4. We link the new objects together (add_toTwo).

5. We then delete the associations from our three object to the old one and two objects. (We must do this because these associations are defined as to-one, so we delete the old associations before setting the new ones in place.)

6. The program listing indicates where we must trick C++ with our knowledge that three and old_three objects are identical. Specifically, we use a handle to a three to set the new associations in place.

7. Finally, we can delete the old one and two objects.
In this example, we use the altered class names (as generated via the `#pragma ooclassname` pragma) to refer effectively to the old versions of the classes. We need an `old_three` class because if we had only one definition of `three` (that is, the one in `newer.h`), then we would have only one definition of the `toTwo` association and its member functions. That single `toTwo` member function would expect an `ooHandle(two)` as its parameter. But then how could we use an `ooHandle(two)` to refer to an `old_two` object?

In this simple program, we could probably solve this last problem a different way by doing a forced cast of an `ooHandle(two)` to an `ooHandle(old_two)`. We do almost the equivalent when we indirectly cast the `ooItr(old_three)` to an `ooHandle(three)`. In more general situations, though, the technique used here is probably preferable, since the only cast we did was between references to truly identical objects.

**Versioning the Classes**

The following is the original DDL schema file:

```cpp
//
// file original.ddl
//
// this file defines version 1 of the various classes
// that will be updated in newer.ddl
//
class one : public ooObj {
    public: // all object fields are public, to make test
        // programs easy to build
        int i1;
        ooRef(two) toTwo <-> toOne;
};
```

---

Class Versioning 6-9
We create a schema using this file and load several objects of each class into an appropriate database within the federated database. We choose to make new versions of classes one and two. We add an object member to one, change the class of the object member in two, and modify the association between these two classes. The proposed new DDL schema file looks like the following:

```c++
//
// file newer.ddl
//
// this file defines version 2 of the various classes
// it is an update of the classes defined in original.ddl
//
class one : public ooObj { // changed
    public: // all object fields are public,
        // to make test programs easy to build
        int i1;
        int j1;
        ooRef(two) toTwo[] <-> toOne : delete(propagate);
};
```

```
class two : public ooObj {
    public:
        int i2;
        ooRef(one) toOne <-> toTwo;
};

class three : public ooObj {
    public:
        int i3;
        ooRef(two) toTwo : copy(delete); // unidirectional
        ooRef(one) toOne : copy(delete); // unidirectional
};
```
class two : public ooObj { // changed
    public:
        double d2;
        ooRef(one) toOne <-> toTwo[] : delete(propagate);
};

class three : public ooObj { // this class is unchanged!
    public:
        int i3;
        ooRef(two) toTwo : copy(delete);
        ooRef(one) toOne : copy(delete);
};

We keep three (the unchanged class) in the same DDL schema file as the modified classes. Note that three has associations to the classes being changed. We would also have to keep three in this file if it used object references to those versioned classes (ooRef(one), ooShortRef(two), and so on).

We use the DDL processor to validate newer.ddl. The DDL processor (using the -nochange and -nooutput flags) confirms that classes one and two had been modified. It does not modify the schema nor does it generate header or source files.

We now copy original.ddl to a new file, older.ddl (to make sure we did not disturb any of the original generated files). We add three #pragma ooclassname pragmas (one becomes old_one, and so on) to that file to produce the following listing:

```
// file older.ddl
// this is file original.ddl with three #pragma ooclassname // usages added //

class one : public ooObj {
    public: // all object fields are public, to make test // programs easy to build
        int i1;
        ooRef(two) toTwo <-> toOne;
};
```
#pragma ooclassname one old_one

class two : public ooObj {
    public:
    int i2;
    ooRef(one) toOne <-> toTwo;
};

#pragma ooclassname two old_two public

class three : public ooObj {
    public:
    int i3;
    ooRef(two) toTwo : copy(delete); // unidirectional
    ooRef(one) toOne : copy(delete); // unidirectional
};

#pragma ooclassname three old_three

We must rename class three even though it will remain unchanged. Also, we use
the public option on the #pragma ooclassname pragma of class two even
though its object fields were already public. If you actually reconstruct this
example, take time to view the header file generated here (older.h) to see the
effect of the public option.

We pass older.ddl through the DDL processor and produce older.h and
older.C files.

We use two different file names with the same class and versions. Further, we
process the second file against an existing schema. This is all legal DDL usage.
Even if there are multiple versions of one or more classes in the schema, as long as
one version matched each class definition in the DDL schema file, the header and
source files are generated to correspond with the input file.

We pass newer.ddl through the DDL processor again, this time with the -
version flag. We can do so because with the previous pass we found that only
classes one and two had changed, as we had expected. This time, the header and
source files (newer.h and newer.C) are generated as we expect.
Copying Objects

The following program copies old class version objects to new class version objects:

```c
// demo program to copy from version 1 to version 2 of various
// classes

// we get the original header file, but with all its classes
// renamed
#include "older.h"

// then the version 2 header file with the versioned class
// definitions
#include "newer.h"

main() {
    ooTrans trans;

    // generic handles
    ooHandle(ooFDObj) fdH;
    ooHandle(ooDBObj) dbH;
    ooHandle(ooContObj) mcH;

    // handles to instances of the newer versions
    ooHandle(one) oneH;
    ooHandle(two) twoH;
    ooHandle(three) threeH;
}
```
// handles and iterator to instances of older versions
ooHandle(old_one)oldOneH;
ooHandle(old_two)oldTwoH;
ooItr(old_three)oldThreeItr;

// begin executable code
//
// Objectivity/DB startup:
//
(void) ooInit();
trans.start();
fdH.open("up.boot",oocUpdate);       // up.boot is our
dbH.open(fdH, "test", oocUpdate);  // boot file. test is
the
   dbH.getDefaultContObj(mcH);       // only database in our
                                    // example. We put all
                                    // our objects in the
                                    // default container of
                                    // the test database.

//
// find all class three objects, since all other objects
// are tied to them
//
// we have to work with the old version of class three in
// order to get the association names and return values to
// work. There really is no old or new version
// of class three, it lets us have associations, and so on,
// from class three to both versions of other classes.
//
oldThreeItr.scan(mcH, oocUpdate);

while (oldThreeItr.next()) {    // loop through all
    // class three objects
// now try to follow association to the version 1 two object
oldThreeItr->toTwo(oldTwoH); // This line is the one that
// forced us to
// create a dummy version of
// three

// and then from version 1 two to version 1 one
oldTwoH->toOne(oldOneH);

// we do the fix as a unit...
// first, get some new (version 2) objects
//
oneH = new(mcH) one;
twoH = new(mcH) two;

// Do the object copy:
// class one: copy one object field, give default value to
// another
//
oneH->i1 = oldOneH->i1;
oneH->j1 = oldOneH->i1 + 111;

// class two: just change the class of the object field
twoH->d2 = oldTwoH->i2;

    // establish the bidirectional association
oneH->add_toTwo(twoH); // this is potentially a to many
    // association, so the method is
    // add_toTwo instead of the
    // set_toTwo used originally

// now version class three: this class was not modified,
// but its unidirectional associations must be changed
// to refer to the new objects
oldThreeItr->del_toOne(); // just delete the old links...
oldThreeItr->del_toTwo(); // ...to both old objects
// then set the new associations in place...
// Here is where we get C++ to let us use the same
// pseudo-pointer (ooItr/ooHandle) to the two versions...
//
// one way to force them
// identical: ensure both are
// using same OID
(ooHandle(ooObj) &amp;) threeH =
    (ooHandle(ooObj) &amp;) oldThreeItr.oid();
threeH-&gt;set_toOne(oneH);     // same as when we created the
threeH-&gt;set_toTwo(twoH);     // associations originally

// We are done with the old objects and can delete them
ooDelete(oldOneH);
ooDelete(oldTwoH);

} // end of the loop that finds all three objects

trans.commit(); // Save these changes

________
Data Model Tuning

This chapter presents data modeling and design suggestions to help tune the performance of Objectivity/DB federated databases. Following these guidelines, you may be able to improve your federated database in terms of:

◆ Federated database size
◆ Speed of database applications

You should also refer to the “Monitoring and Tuning Performance” chapter in Using Objectivity/ C++ for additional application tuning suggestions.

Tuning Concepts

You should be familiar with the following features and concepts when tuning an Objectivity/DB federated database.

Associations

An association is a typed relationship between two objects. To allow an association between objects, you must define an association link in each object’s class definition. An association between objects is stored in the system default association VArray for each object of a class that declared the link (that is, in the association VArray for one object for a unidirectional association, or in the association VArrays for both objects for a bidirectional association) unless you declared the association link(s) as inline. For example, in Figure 7-1, the unidirectional associations of object A1 with objects C1, C2, and B are stored in the system default association VArray for object A1. Each association is identified by the name of the association link on which it is set and the object identifier (OID) of the object being associated. Twelve bytes are needed to store each association, four bytes for the name of the link and eight bytes for the OID. To trace a particular association on object A1, an application must traverse the associations in the system default association VArray until it locates the desired association.
By declaring an association link as inline, you can reduce both the storage required for associations on that link and the time it may take to trace an association on that link (see “Use Inline Associations” on page 7-3).

```cpp
class A : public ooObj {
    public:
        ooRef(B) toB : copy(delete);
        ooRef(C) toC [] : copy(delete);
};
```

**Figure 7-1  Storage of Associations on Object A1 of Class A**

**Associations and Basic Object Storage**

The standard storage overhead for a basic object is 14 bytes. This overhead is constant and is independent of an application’s use of associations. The following storage requirements are for unidirectional associations. Each bidirectional association link requires storage equivalent to two unidirectional links.

If you use non-inline associations then the additional space required per basic object is as follows:

- 14 bytes if any association links exist
- 12 bytes per unidirectional association link

If you use inline to-one associations, then the additional space required per basic object is as follows:

- 8 bytes per unidirectional to-one association (whether a link exists or not)
- 4 bytes per short unidirectional to-one association link (whether a link exists or not)

**7-2  Using Objectivity/C++ Data Definition Language**
If you use inline to-many associations, then the additional space required per basic object is as follows:

- 4 bytes per unidirectional to-many association link (whether a link exists or not)
- 14 bytes per unidirectional to-many association link (if any association link exists)
- 8 bytes per unidirectional to-many association link
- 4 bytes per short unidirectional to-many association link

Since objects are stored on eight-byte boundaries, you should round up your size calculations to the nearest eight bytes.

### Tuning for Federated Database Size

The following actions may help reduce the size of an Objectivity/DB federated database.

#### Use Inline Associations

By using inline association links instead of non-inline links, you can reduce the storage required for associations on that link. You can further reduce federated database size by using short inline association links instead of long inline association links. Long inline association links use OIDs to refer to associated objects. Short inline association links use short OIDs to refer to associated objects and can be used when the database and container of the associated objects are known. Since OID information about the database and container are not required for basic objects within the same container, you can save space by setting associations between these objects using short inline association links. See "Association Links" on page 2-24.

For example, the OID for a one-to-one association on an inline association link is embedded in the object itself, as shown in Figure 7-2 on page 7-4, where the association toD is embedded in object A1. The OIDs for a one-to-many association on an inline association link are stored in a VArray for that link, such as the VArray for the inline association link toE in Figure 7-2 on page 7-4. In either case, only eight bytes are needed to store each association. Objects of classes F and G reside in the same container as objects of class A, so the association links toF and toG require only four bytes to store each association, since the associated object can be identified by its short OID.
Since inline associations are embedded within objects, they take up space even when they are not used. This may negatively impact federated database size. If you use associations infrequently, you may want to use non-inline associations instead.

```cpp
class A : public ooObj {
    public:
        ooRef(B) toB : copy(delete);
        ooRef(C) toC [] : copy(delete);
        inline ooRef(D) toD;
        inline ooRef(E) toE [];
        inline ooShortRef(F) toF;
        inline ooShortRef(G) toG [];
};
```

Figure 7-2 Storage of Associations on Object A1 of Class A
Store Data Efficiently

Objectivity/DB provides primitive data types as described in “Primitive Types” on page 2-8. Be sure to use the data type that requires the least amount of storage necessary for each element of your data. For example, use type `uint8` instead of type `uint16` for a small integer value. See “C++ Primitive Types” on page 2-11 for the DDL processor’s mapping of C++ primitive types to DDL primitive types.

Also consider the order in which you declare your data types to optimize the use of storage. By declaring larger data types first, you can optimize the packing of your data and use less space, which can make a significant difference when you have many objects.

Example

For example, because most compilers require data of type `float64` to start on an 8-byte boundary, and data of type `uint16` to start on a 2-byte boundary, the following declaration will produce the packing shown in Figure 7-3, occupying 32 bytes and wasting 12 bytes of storage in the shaded areas:

```cpp
class myclass {
    uint16 a;
    float64 b;
    uint16 c;
    float64 d;
};
```

And the following declaration will produce the packing shown in Figure 7-4, occupying 24 bytes and wasting only the 4 bytes of storage shown in the shaded area:

```cpp
class myclass {
    float64 b;
    float64 d;
    uint16 a;
    uint16 c;
};
```
Figure 7-3  Data Packing When Alternating Long and Short Types

Figure 7-4  Data Packing When Declaring Long Types First

---

7-6  Using Objectivity/C++ Data Definition Language
Tuning for Speed

To achieve optimal speed for an application, you must minimize I/O, networking overhead, and CPU time. The following actions may significantly improve the speed of your application by increasing the efficiency with which it can access and manipulate objects in your federated database.

Use Inline Associations

Declaring association links as inline makes the association links more efficient since traversing or updating an association on them is much faster in most cases. For example, a one-to-one inline association can be traced very quickly because it is embedded in the object. And an association on a one-to-many inline association link can also be traced quickly because the application needs to traverse only associations on that link instead of all of the associations on the object.

However, inline associations are less efficient when you need to add more association types because they require you to use class versioning to create a new version of the class.
## Schema Evolution Basic Operation Summary

This appendix summarizes requirements for basic schema evolution operations. Use this information in conjunction with the procedures described in the “Schema Evolution” chapter.

### Table A-1: Basic Class Content Operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Conversion Operation?</th>
<th>DDL Flags</th>
<th>DDLPragma Needed?</th>
<th>Upgrade Application Needed?</th>
<th>See page...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deleting a data member</td>
<td>Yes</td>
<td>-evolve</td>
<td>No</td>
<td>No</td>
<td>5-15</td>
</tr>
<tr>
<td>Adding a data member with system default values</td>
<td>Yes</td>
<td>-evolve</td>
<td>No</td>
<td>No</td>
<td>5-16</td>
</tr>
<tr>
<td>Adding a data member with a user-defined default value</td>
<td>Yes</td>
<td>-evolve</td>
<td>Yes</td>
<td>No</td>
<td>5-16</td>
</tr>
<tr>
<td>Renaming a data member</td>
<td>No</td>
<td>-evolve</td>
<td>Yes</td>
<td>No</td>
<td>5-17</td>
</tr>
<tr>
<td>Changing the type of a primitive data member</td>
<td>Yes</td>
<td>-evolve</td>
<td>Yes</td>
<td>No</td>
<td>5-19</td>
</tr>
</tbody>
</table>
### Table A-1: Basic Class Content Operations (Continued)

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Changing the order (position) of data members</td>
<td>Yes</td>
<td>-evolve</td>
<td>No</td>
<td>No</td>
<td>5-21</td>
</tr>
<tr>
<td>Changing access control of a data member (public, private, protected)</td>
<td>No</td>
<td>-evolve</td>
<td>No</td>
<td>No</td>
<td>5-22</td>
</tr>
<tr>
<td>Adding the first virtual member function</td>
<td>Yes</td>
<td>-evolve</td>
<td>No</td>
<td>No</td>
<td>5-23</td>
</tr>
<tr>
<td>Removing the last virtual member function</td>
<td>Yes</td>
<td>-evolve</td>
<td>No</td>
<td>No</td>
<td>5-24</td>
</tr>
<tr>
<td>Changing the size of an array data element</td>
<td>Yes</td>
<td>-evolve</td>
<td>No</td>
<td>No</td>
<td>5-24</td>
</tr>
<tr>
<td>Adding an inline association</td>
<td>Yes</td>
<td>-evolve</td>
<td>No</td>
<td>No</td>
<td>5-26</td>
</tr>
<tr>
<td>Adding a non-inline association</td>
<td>No</td>
<td>-evolve</td>
<td>No</td>
<td>No</td>
<td>5-27</td>
</tr>
<tr>
<td>Adding an object reference</td>
<td>Yes</td>
<td>-evolve</td>
<td>No</td>
<td>No</td>
<td>5-29</td>
</tr>
<tr>
<td>Deleting an association</td>
<td>Yes</td>
<td>-evolve</td>
<td>No</td>
<td>No</td>
<td>5-30</td>
</tr>
<tr>
<td>Deleting an object reference</td>
<td>Yes</td>
<td>-evolve</td>
<td>No</td>
<td>No</td>
<td>5-31</td>
</tr>
<tr>
<td>Changing an object reference from short to long and vice versa</td>
<td>Yes</td>
<td>-evolve</td>
<td>No</td>
<td>No</td>
<td>5-32</td>
</tr>
</tbody>
</table>
### Table A-1: Basic Class Content Operations (Continued)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Conversion Operation?</th>
<th>DDL Flags</th>
<th>DDLPragma Needed?</th>
<th>Upgrade Application Needed?</th>
<th>See page...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Changing the representation (inline/non-inline) of an association</td>
<td>Yes</td>
<td>−evolve</td>
<td>No</td>
<td>No</td>
<td>5-34</td>
</tr>
<tr>
<td>Changing an inline association from short to long and vice versa</td>
<td>Yes</td>
<td>−evolve</td>
<td>No</td>
<td>No</td>
<td>5-35</td>
</tr>
<tr>
<td>Changing the behavior specifiers of an association (lock and delete propagation, versioning and copy properties)</td>
<td>No</td>
<td>−evolve</td>
<td>No</td>
<td>No</td>
<td>5-37</td>
</tr>
<tr>
<td>Renaming an association</td>
<td>No</td>
<td>−evolve</td>
<td>Yes</td>
<td>No</td>
<td>5-38</td>
</tr>
<tr>
<td>Changing the access control of an association</td>
<td>No</td>
<td>−evolve</td>
<td>No</td>
<td>No</td>
<td>5-39</td>
</tr>
<tr>
<td>Changing an array size for an array of object references</td>
<td>Yes</td>
<td>−evolve</td>
<td>No</td>
<td>No</td>
<td>5-40</td>
</tr>
<tr>
<td>Changing the order (position) of a non-inline association</td>
<td>No</td>
<td>−evolve</td>
<td>No</td>
<td>No</td>
<td>5-41</td>
</tr>
<tr>
<td>Changing the order (position) of an inline association or object reference</td>
<td>Yes</td>
<td>−evolve</td>
<td>No</td>
<td>No</td>
<td>5-42</td>
</tr>
</tbody>
</table>
### Table A-2: Basic Class Operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Conversion Operation?</th>
<th>DDL Flags</th>
<th>DDLPragma Needed?</th>
<th>Upgrade Application Needed?</th>
<th>See page ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Renaming a class</td>
<td>No</td>
<td>-evolve</td>
<td>Yes</td>
<td>No</td>
<td>5-53</td>
</tr>
<tr>
<td>Deleting a class</td>
<td>Maybe^a</td>
<td>-evolve</td>
<td>Yes</td>
<td>Maybe^b</td>
<td>5-55</td>
</tr>
</tbody>
</table>

^a Yes for other objects that reference objects of the deleted class

^b Yes, if an existing class that has either an object reference or association to deleted class or any of its base classes.
Table A-3: Basic Inheritance Operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Conversion Operation?</th>
<th>DDL Flags</th>
<th>DDL Pragma Needed?</th>
<th>Upgrade Application Needed?</th>
<th>See page ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adding a non-persistent-capable class as a base class</td>
<td>Yes</td>
<td>-evolve</td>
<td>No</td>
<td>No</td>
<td>5-60</td>
</tr>
<tr>
<td>Removing a non-persistent-capable class as a base class</td>
<td>Yes</td>
<td>-evolve</td>
<td>No</td>
<td>No</td>
<td>5-62</td>
</tr>
<tr>
<td>Replacing a base class with a subset of its base classes</td>
<td>Yes</td>
<td>-evolve</td>
<td>Yes</td>
<td>Yes</td>
<td>5-64</td>
</tr>
<tr>
<td>Replacing a base class with one of its derived classes</td>
<td>Yes</td>
<td>-evolve</td>
<td>Yes</td>
<td>Yes</td>
<td>5-68</td>
</tr>
<tr>
<td>Changing the order (position) of a base class</td>
<td>Yes</td>
<td>-evolve</td>
<td>No</td>
<td>No</td>
<td>5-71</td>
</tr>
<tr>
<td>Changing the access control of a base class</td>
<td>No</td>
<td>-evolve</td>
<td>No</td>
<td>No</td>
<td>5-73</td>
</tr>
</tbody>
</table>
A-6 Using Objectivity/C++ Data Definition Language
This appendix describes the Objectivity/C++ DDL processor tool for C++ application development. Table B-1 lists the tool’s names for Windows, UNIX, VMS, and MacOS environments. The syntax description begins on page B-2.

**Table B-1: Database Development Tools**

<table>
<thead>
<tr>
<th>Windows</th>
<th>UNIX and MacOS</th>
<th>VMS</th>
</tr>
</thead>
<tbody>
<tr>
<td>ooddlx.exe</td>
<td>ooddlx</td>
<td>ODB DDL</td>
</tr>
</tbody>
</table>

Most VMS qualifiers are identical to their corresponding UNIX flags. Table B-2 lists those that are different.

**Table B-2: UNIX Flag and VMS Qualifier Differences**

<table>
<thead>
<tr>
<th>UNIX Flag</th>
<th>Equivalent VMS Qualifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>-I</td>
<td>/INCLUDE</td>
</tr>
<tr>
<td>-include_header</td>
<td>/INC_HEADER</td>
</tr>
<tr>
<td>-include_ref</td>
<td>/INC_REF</td>
</tr>
<tr>
<td>-D</td>
<td>/DEFINE</td>
</tr>
<tr>
<td>-U</td>
<td>/UNDEFINE</td>
</tr>
<tr>
<td>-E</td>
<td>/PREPROCESS_ONLY</td>
</tr>
</tbody>
</table>
**ooddlx**

**DESCRIPTION**  Process schema definitions and add them to the federated database specified by `bootFilePath`.

- You can abbreviate flag names using enough characters to uniquely identify the flag.
- You can use any number of `-D`, `-I`, and `-U` flags. The `ooddlx` command issues a warning for each repetition of any other flag, except for a flag repeated with different arguments, for which it issues an error.
- The `ooddlx` command issues an error when two `-D` flags set the same preprocessor variable to different values. It issues a warning whenever two `-D`, two `-I`, or two `-U` flag arguments are the same.
- In VMS, although you may have more than one qualifier value for each of the qualifiers `INCLUDE`, `DEFINE`, and `UNDEFINE`, you may not use these qualifiers more than once in a command. For example, in UNIX you would use multiple options and option parameters as follows:
  ```
  -Idir1 -Idir2 -Idir3
  ```
  In VMS, however, you group all three qualifier values after a single qualifier:
  ```
  /INCLUDE_=(dir1,dir2,dir3)
  ```

**SYNTAX**
```
ooddlx | oocddl
  [-D...][-E][-I...][-U...]
  [-c [-noc++][-cheader_suffix suffix]]
  [-ifdef_cheader variable]
  [-c++_suffix suffix]
  [-evolve]
  [-header_suffix suffix]
  [-ifdef_header variable]
  [-ifdef_ref variable]
  [-include_header pathname]
  [-include_ref pathname]
  [-noanachronism]
  [-noincludeoo]
  [-noline]
  [-nooutput]
```
[-noref]
[-notitle]
[-notouch]
[-nowarn]
[-ref_suffix suffix]
[-schema name]
[-standalone]
[-storage_specifier specifier]
[-upgrade]
[-version | -nochange]
[-help]
classDefFile.ddl
[bootFilePath]

where

classDefFile.ddl      Name of the DDL schema file that contains the class definitions

bootFilePath          Optional parameter for the path to the boot file of the federated database whose schema will receive the class definitions.

If you do not include this parameter, the DDL processor uses the OO_FD_BOOT environment variable to locate the boot file. If you do not include this parameter and have not set OO_FD_BOOT, you receive an error message. For information on environment variables, see “Distributed Objectivity/DB Systems” on page 1-7 of Objectivity/DB Administration.

For VMS, the bootFilePath command qualifier is optional only if you have defined an ODB$FD_BOOT logical name.

-c                     Produce C as well as C++ output files.

-chader_suffix suffix  Use suffix as the suffix of the C output header files instead of the default _c.h. This flag requires the -c flag.
-c++_suffix suffix

Use suffix as the suffix of the C++ output implementation files instead of the default _ddl.C.

-evolve

Enable schema evolution. You cannot combine this flag with the -version or -nochange flags. To reuse DDL schema files after evolving your schema, you must manually remove from these files any pragma statements used for the evolution process.

-header_suffix suffix

Use suffix as the suffix of the C++ output header files instead of the default .h.

-help

Describe the ooddlx command and its command line syntax. Ignore all other command line arguments, and do none of the normal ooddlx command processing.

-ifdef_cheader variable

Use variable as the preprocessor variable used by the #ifndef wrapped around the content of the C output header files, instead of the default module_C_H. This flag requires the -c flag.

-ifdef_header variable

Use variable as the preprocessor variable used by the #ifndef wrapped around the content of the C++ output header files, instead of the default module_H.

-ifdef_ref variable

Use variable as the preprocessor variable used by the #ifndef wrapped around the content of the ooRef and related definitions C++ output header file, instead of the default module_REF_H.
-include_header pathName
Use pathName as the name of the file to include as the C++ output header file in the C++ output implementation file, instead of the default, which is the name of the actual C++ output header file.

-include_ref pathName
Use pathName as the name of the file to include as the C++ output ooRef and related definitions header file in the C++ output header file, instead of the default, which is the name of the actual C++ output ooRef and related definitions header file.

-noanachronism
For each use of a feature provided only for backward compatibility, do not issue a warning or silently permit the usage, but instead issue an error.

-noc++
Do not produce C++ output. This flag requires the -c flag. This flag cannot be combined with the -c++_suffix, -header_suffix, -ifdef_header, -ifdef_ref, -include_header, -include_ref, or -ref_suffix flags.

-nochange
Issue an error instead of changing the schema representation when a class is added or changed. You cannot combine this flag with the -version or -evolve flags.

-noincludeoo
Do not implicitly include file oo.h. This flag is useful to avoid multiple inclusion of file oo.h when using the output produced by the ooddlix command with the -E flag as input to another invocation of ooddlix.

-noline
Do not include #line directives in the C++ or C output.
-nooutput  Do not output C++ or C files. You cannot combine this flag with the following flags: 
- c, -cheader_suffix, 
- c++, -header_suffix, 
- ifdef_cheader, -ifdef_header, 
- ifdef_ref, -include_header, 
- include_ref, -noline, -notouch, 
or -ref_suffix.

-noref  Do not output the C++ ooRef and related definition header files. You cannot combine this flag with the following flags: 
- ifdef_ref, 
- include_ref, -noc++, -nooutput, 
or -ref_suffix. This flag is for backward compatibility only.

-notitle  Do not print the Objectivity/DB copyright notice.

-notouch  Do not actually touch any output file that already contains what would otherwise be placed in it.

-nowarn  Issue errors but not warnings. This flag does not apply if the -evolve flag is used.

-ref_suffix suffix  Use suffix as the suffix of the C++ output ooRef and related definitions header files instead of the default _ref.h.

-schema name  Access federated database persistent schema representation name.

-standalone  Do not use a lock server for federated database access.

-storage_specifier specifier  Interpret specifier in the input as __declspec(ddlexport) and preserve it in the output unchanged. This flag is incompatible with the -nooutput and -noc++ flags.
-upgrade

Indicates that certain object conversion operations will be performed by an upgrade application. This flag requires the -evolve flag.

-version

Version the schema representation for each class that has changed instead of issuing an error. You cannot combine this flag with the -nochange flag.

-D varNameAndValue

Same as the UNIX C compiler -D flag. There must not be any whitespace between the -D and the varNameAndValue.

-E

Same as the UNIX C compiler -E flag. You cannot combine this flag with the -nofd and -nooutput flags nor with any flag that is incompatible with -nofd and -nooutput.

-I pathName

Same as the UNIX C compiler -I flag. There must not be any whitespace between the -I and pathName. Included files are searched for in the -I flags' directory arguments in the order in which the flags appeared on the command line.

-U varName

Same as the UNIX C compiler -U flag. There must not be any whitespace between the -U and the varName. All -D flags are processed before all -U flags.
Using Objectivity/C++ Data Definition Language
Index

A

adding
  data member 5-16
    with copied value 5-43
  first virtual member function 5-23
  inline association 5-26
  non-inline association 5-27
  non-leaf base class 5-75
  non-persistent-capable class as base class 5-60
  object reference 5-29

affected objects 5-1
aggregate data types 2-12
application
  object conversion 5-9
  speed 7-7
array
  fixed
    changing number of dimensions 5-49
    changing to variable-size 5-49
    fixed-size 2-12
  of object references
    changing array size 5-40
    system default association 2-24
  variable-size (see VArray)
association 7-1
  adding
    inline 5-26
    non-inline 5-27
  basic object 2-4
  between three or more objects 2-22
  bidirectional 2-18
    converting 6-6
    many-to-many 2-18
    many-to-one 2-18
    one-to-many 2-18
    one-to-one 2-18
  changing
    access control 5-39
    behavior specifiers 5-37
    cardinality 5-50
    domain 5-52
    object reference 5-50
  container 2-5
  deleting 5-30
  inline 7-1
    changing
      order 5-42
      one-to-many 7-7
      one-to-one 7-7
  n-ary 2-22
  non-inline
    changing order 5-41
    renaming 5-38
  storage of 2-24
    inline 7-2, 7-3
    inline (figure) 7-4
    non-inline 7-1, 7-2
    non-inline (figure) 7-2
    system default association array 2-24
  unidirectional 2-20
    changing originating class 5-51
    declaring 2-20
    one-to-many 2-20
    using as one-way reference to versioned class 6-7
association link 2-16, 2-20, 2-29, 2-30, 2-32
  and object copying 2-31
  and propagation 2-29
  and versioning 2-37
  bidirectional 2-32
  combining behavior specifiers 2-38
  inline 2-24, 7-2, 7-3, 7-7
    accessing 2-26
    defining 2-26
    long 2-24, 7-3
    short 2-24, 7-3
    speed of traversing 7-7

affected objects 5-1
aggregate data types 2-12
application
  object conversion 5-9
  speed 7-7
array
  fixed
    changing number of dimensions 5-49
    changing to variable-size 5-49
    fixed-size 2-12
  of object references
    changing array size 5-40
    system default association 2-24
  variable-size (see VArray)
association 7-1
  adding
    inline 5-26
    non-inline 5-27
  basic object 2-4
  between three or more objects 2-22
  bidirectional 2-18
    converting 6-6
    many-to-many 2-18
    many-to-one 2-18
    one-to-many 2-18
    one-to-one 2-18
  changing
    access control 5-39
    behavior specifiers 5-37
    cardinality 5-50
    domain 5-52
    object reference 5-50
  container 2-5
  deleting 5-30
  inline 7-1
    changing
      order 5-42
      one-to-many 7-7
      one-to-one 7-7
  n-ary 2-22
  non-inline
    changing order 5-41
    renaming 5-38
  storage of 2-24
    inline 7-2, 7-3
    inline (figure) 7-4
    non-inline 7-1, 7-2
    non-inline (figure) 7-2
    system default association array 2-24
  unidirectional 2-20
    changing originating class 5-51
    declaring 2-20
    one-to-many 2-20
    using as one-way reference to versioned class 6-7
association link 2-16, 2-20, 2-29, 2-30, 2-32
  and object copying 2-31
  and propagation 2-29
  and versioning 2-37
  bidirectional 2-32
  combining behavior specifiers 2-38
  inline 2-24, 7-2, 7-3, 7-7
    accessing 2-26
    defining 2-26
    long 2-24, 7-3
    short 2-24, 7-3
    speed of traversing 7-7
non-inline 2-24
notation for class association links 2-16
assocLink behavior specifier 2-29, 2-30, 2-32, 2-37
autonomous partition
class 2-6
system name 2-6

B

base class
changing access control 5-73
replacing
with derived class 5-68
with subset of its base classes 5-64
basic object
association 2-4
class 2-4
copying
  copy 2-31
  scope name 2-4
standard storage overhead 7-2
storage for associations 7-2
versioning 2-4
behavior specifier 2-29
assocLink 2-29, 2-30, 2-32, 2-37
combining 2-38
copy 2-32
bit fields 2-42
boot file B-3
VMS 1-8

C

changing
access control
association 5-39
of a data member 5-22
of base class 5-73
array size for array of object references 5-40
association
behavior specifiers 5-37
cardinality 5-50
domain 5-52
to object reference 5-50
basic inheritance relationships 5-60
fixed-size array to variable-size array 5-49
inline association
from long to short 5-35
from short to long 5-35
number of dimensions of fixed-size array 5-49
object reference
from long to short 5-32
from short to long 5-32
object reference to association 5-50
order
of base class 5-71
of data member 5-21
of inline association 5-42
of non-inline association 5-41
of object reference 5-42
origin class
of data member to base class 5-45
of data member to derived class 5-47
originating class of unidirectional association 5-51
primitive data member type 5-19
representation of an association 5-34
size of a fixed-length array of data member 5-24
type of non-primitive data member 5-49
class
autonomous partition 2-6
basic object 2-4
constructor 2-3, 2-14
container 2-5
database 2-5
declarations 2-3
deducing in database schema 2-1
default container 2-5
definitions
  separating 1-17
definitions file 1-4
deleting 5-55
destructor 2-3
federated database 2-6
hierarchy 2-3
inheritance
  multiple 2-3, 4-1
  single 2-3
making
  non-persistent capable class
  persistent-capable 5-78
  persistent-capable class
    non-persistent-capable 5-79
member field 2-3
member function 2-16
merging two associated classes 5-82
nested 2-45
non-persistent-capable 1-1, 5-1
 ooAPObj 2-6
 ooContObj 2-5
 ooDBObj 2-5
 ooDefaultContObj 2-5
 ooFDObj 2-6
 ooObj 2-4
 ooVArray 2-12
persistent-capable 1-1, 4-4, 5-1
  system-defined 2-4
persistent-capable base 2-3
persistent-capable class
  defining 2-4
  restrictions 1-3
  user-defined 2-3
splitting into two classes 5-79, 5-81
templates 2-40
transient 1-2, 5-1
upgrading 6-9
VArray 2-12

version identifier 6-1
versioning 6-1, 7-7
with inherited associations or object
  references
deleting 5-58

class versioning 7-7
  hints 6-6

composite object 2-29, 4-7

constraints
  DDL 2-46
    constructor
      for inheritance 2-3
      domain 2-46
      object-level 2-46
    VArrays 2-13

constructor
  class 2-3, 2-14
  constraints 2-3

container
  association 2-5
  class 2-5
  default class 2-5
  system name 2-5

conversion function 5-3, 5-4

conversion operation 5-4

copy
  association links when copying 2-32
  association links when versioning 2-37

data
  modeling 1-1
  preserving 1-19
  type
    order of declaration 7-5
data member
adding 5-16
  with copied value 5-43
changing
  access control 5-22
  order 5-21
  origin class 5-45
  origin class to derived class 5-47
deleting 5-15
fixed-length array
  changing size 5-24
non-primitive
  changing type 5-49
primitive
  changing type 5-19
renaming 5-17
data modeling 2-1
  default data model 3-1
tuning 7-1
database
class 2-5
  system name 2-5
DDL 1-1
  language description 2-1
  processor 1-4, 2-1
  actions of 1-6
    generated schema files 1-6, 1-9
    running 1-6
  syntax B-1
  VMS qualifiers B-1
schema file 2-1
  multiple 2-1
  organizing 6-7
  using 2-1
declare 2-12
default container
class (ooDefaultContObj) 2-5
deferred object conversion 5-2
delete association links
  when copying 2-32
  when versioning 2-37
delete propagation 2-29
deleting
  association 5-30
  class 5-55
    with inherited associations or object references 5-58
  data member 5-15
  last virtual member function 5-24
  non-leaf base class 5-77
  non-persistent-capable class as a base class 5-62
  object reference 5-31
destructor, class 2-3
e
environment variable
  OO_FD_BOOT B-3
f
federated database 1-1
  class 2-6
  size 7-3
  system name 2-6
files
  DDL schema
    multiple 1-17
    included in schema header file 1-9
  multiple DDL schema 2-1
  schema source code 2-1
h
header file
  automatic inclusion of 1-16
immediate object conversion 5-3, 5-5
implement macro 2-13
including generated header files 1-10
inclusion order errors 1-15
inheritance
  changing basic 5-60
  multiple 2-3, 4-1
  single 2-3
inherited persistence 4-4

leaf persistence 4-9, 4-12
locking
  propagating locks 2-30
logical name
  ODB$FD_BOOT B-3

macros
  implement 2-13
making
  persistent-capable class
  non-persistent-capable 5-79
member function
  adding first virtual 5-23
  deleting last virtual 5-24
member pointers 2-45
merging
  two associated classes 5-82, 5-83
move
  association links when copying 2-32
  association links when versioning 2-37
multiple inheritance (see inheritance, multiple)
nested classes 2-45
non-conversion operation 5-4
non-persistent-capable class 1-1, 5-1
notation
  for class link associations 2-16
object
  constraints 2-46
  copying
    and association link 2-31
    association copy behavior specifier 2-31
    basic object 2-31
    copy 2-31
  model (see data modeling)
  persistent (see persistent object)
  system default association array 2-24
object conversion 5-1
  conversion function 5-3
  deferred 5-2
  immediate 5-3, 5-5
  on-demand 5-3, 5-5
object identifier (OID) 7-3
  short 7-3
object model (see data modeling)
object reference
  adding 5-29
  changing
    association 5-50
    from long to short 5-32
    from short to long 5-32
    order 5-42
  deleting 5-31
ODB DDL (see DDL processor)
ODB$FD_BOOT logical name B-3
on-demand object conversion 5-3, 5-5

oo.h
  automatic inclusion of 1-16

OO_FD_BOOT
  environment variable B-3

ooAPObj 2-6
oochangebase pragma 5-8, 5-65, 5-68
ooclassname 6-3
oocontref 1-11
ooContObj 2-5
ooCopyfd 5-7
ooDBObj 2-5
ooddlx (see DDL processor)
ooddlx.exe (see DDL processor)
oodefault pragma 5-8, 5-16, 5-45, 5-47
ooDefaultContObj 2-5
ooDeleteclass pragma 5-8, 5-56
ooFDObj 2-6
ooObj 2-4
ooRef 2-24, 2-26
oorename 6-4
oorename pragma 5-8, 5-17, 5-53
ooShortRef 2-24, 2-26
ooTrans 5-3, 5-59, 5-65, 5-69, 5-75
ooTVAArray 2-12
ooValidate 2-46
ooVArray 2-12
  constraints 2-13

P

persistence
  leaf 4-4
  mixing root and leaf 4-14
  root 4-4
  through inheritance 4-4
persistent-capable class 1-1, 4-4, 5-1
  base 2-3
  defining 2-4

restrictions 1-3
system-defined 2-4
user-defined 2-3

pointer
  embedded 1-3

pragma 1-16
  non-DDL 1-16
  oochangebase 5-8, 5-65, 5-68
  ooclassname 6-3
  ooclassref 1-11
  ooDeleteclass 5-8, 5-56
  oorename 5-8, 5-17, 5-53, 6-4
  ooschema 3-4
  schema evolution 5-8, 5-11

preprocessor directives
  #include directive 1-16
  using 1-16

primitive types 7-5
  C++ 2-11
  compatibility
    C++ and DDL 2-11
  DDL 2-8
    character 2-8, 2-10
    floating point 2-8, 2-10
    integer 2-8, 2-9

propagation
  and association links 2-29
  and composite objects 2-29
  combining behavior specifiers 2-38
  deleting 2-29
  locking 2-30
**R**

removing (see deleting)

renaming
  association 5-38
  class 5-53
  data member 5-17

replacing
  base class with derived class 5-68
  base class with subset of its base classes 5-64
  user-defined type with related type 5-43

**S**

schema 1-1
  adding class 3-2
  adding class to 1-5
  adding classes to alternate 3-2
  C++ implementation file 1-9
  changing 1-19
  creating 1-1, 1-5
  default 3-1
  defined 1-1
  files 2-1
  multiple 2-1
  generated files 1-6, 1-9
  header file 1-9
  modifying 1-19
  multiple 1-1, 1-17, 3-1
  naming 3-1
  reference header file 1-9
  scope 1-1
  source code file 2-1
  specifying current 3-4
  syntax for declaring classes 2-1

**schema evolution 5-1**

adding
  data member with copied value 5-43
  first virtual member function 5-23
  inline association 5-26
  non-inline association 5-27
  non-leaf base class 5-75
  non-persistent-capable class as base class 5-60
  object reference 5-29
  adding a data member 5-16
  basic class content operations 5-11
  basic class operations 5-53
  basic inheritance operations 5-60
  basic operations 5-5
  changing
    access control of association 5-39
    access control of base class 5-73
    access control of data member 5-73
    array size for array of object references 5-40
    association behavior specifiers 5-37
    association cardinality 5-50
    association domain 5-52
    association to object reference 5-50
    fixed-size array
      to variable-size array 5-49
    inline association from long to short 5-35
    inline association from short to long 5-35
    number of dimensions of fixed-size array 5-49
    object reference
      to association 5-50
      object reference from short to long 5-32
order of data member 5-21
order of a non-inline association 5-41
order of base class 5-71
order of inline association 5-42
order of object reference 5-42
origin class of a data member to derived class 5-47
origin class of data member to base class 5-45
originating class of unidirectional association 5-51
primitive data member type 5-19
representation of an association 5-34
size of a fixed-length array of data member 5-24
type of a non-primitive data member 5-49
changing object reference from long to short 5-32
cleanup phase 5-11
complex class operations 5-58
complex inheritance operations 5-75
complex operations 5-5
deleting association 5-30
class 5-55
class with inherited associations or object references 5-58
data member 5-15
non-leaf base class 5-77
non-persistent-capable class as a base class 5-62
object reference 5-31
deleting last virtual member function 5-24
error message types 5-9
example 5-84
making non-persistent-capable class persistent-capable 5-78
persistent-capable class non-persistent-capable 5-79
merging two associated classes 5-82, 5-83
multiple class operations 5-79
pragmas 5-8
preparation phase 5-7
rebuilding applications 5-9
renaming association 5-38
class 5-53
data member 5-17
replacing base class with derived class 5-68
base class with subset of its base classes 5-64
user-defined type with related type 5-43
retained history 5-5
schema evolution phase 5-9
splitting a class into two classes 5-79, 5-81
upgrade application 5-55, 5-58, 5-64, 5-68
scope name basic object 2-4
system name autonomous partition 2-6
container 2-5
database 2-5
federated database 2-6
T

templates 2-40
transient class 1-2, 5-1
tuning
  concepts 7-1
  data model 7-1
  federated database size 7-3
  performance 7-1
  speed 7-7
type
  aggregate data 2-12
    fixed-size arrays 2-12
    VArrays 2-12
  primitive 7-5
    C++ 2-11
    compatibility
      C++ and DDL 2-11
  DDL 2-8
    character 2-8, 2-10
    floating point 2-8, 2-10
    integer 2-8, 2-9
type versioning (see class versioning)

U

unions 2-42
upgrade
  ooTrans 5-3, 5-59, 5-65, 5-69, 5-75
upgrade application 5-3, 5-10, 5-55, 5-58, 5-64, 5-68, 5-75
upgrade interface 5-3, 5-59, 5-65, 5-69, 5-75
upgradeObjects
  ooFDObj 5-3, 5-65, 5-69, 5-75

V

variable-size array (see VArray)
VArray
  constraints 2-13
  declaring 2-12
versioning
  and association link 2-37
  basic object 2-4
  classes 6-1, 7-7
virtual base classes 2-45