

Objectivity/C++ Standard Template Library

Release 5.2

Objectivity/C++ Standard Template Library

Part Number: 52-STL-0

Release 5.2, September 20, 1999

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 1999 by 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/DB Hot Failover, Objectivity/DB Open File System, Objectivity/OFS, Objectivity/DB Secure Framework, Objectivity/Secure, Objectivity/C++, Objectivity/C++ Data Definition Language, Objectivity/DDL, Objectivity/C++ Active Schema, Objectivity/C++ Standard Template Library, Objectivity/C++ STL, Objectivity/C++ Spatial Index Framework, Objectivity/Spatial, Objectivity for Java, Objectivity/Smalltalk, Objectivity/SQL++, Objectivity/SQL++ ODBC Driver, Objectivity/ODBC, and Objectivity Event Notification Services are trademarks of Objectivity, Inc. Standards<ToolKit> is a trademark of ObjectSpace, Inc. Other trademarks and products are the property of their respective owners.

ODMG information in this document is based in whole or in part on material from *The Object Database Standard: ODMG 2.0*, edited by R.G.G. Cattell, and is reprinted with permission of Morgan Kaufmann Publishers. Copyright 1997 by Morgan Kaufmann Publishers.

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	7	
Audience	7	
Organization	7	
Conventions and Abbreviations	8	
Getting Help	9	
Part 1	GUIDE	
<hr/>		
Chapter 1	Understanding Objectivity/C++ STL	13
	STL Components	13
	Objectivity/C++ STL	14
	Persistence-Capability	15
	‘Container’ Term	16
	Implementation Classes	16
	Node-Based Container Classes	16
	VArray-Based Container Classes	17
	Iterators	18
	STL Iterators	18
	Objectivity/C++ STL Iterators	18
	Allocating Storage	20
	Explicitly Requesting the Default Allocator	21
	Implicitly Requesting the Default Allocator	21
	Specifying Your Own Allocator Class	21

Chapter 2	Using Objectivity/C++ STL	23
	General Guidelines	23
	Instantiating Persistence-Capable Template Classes	23
	Including Header Files	24
	Objectivity/C++ Capabilities and Objectivity/C++ STL	25
	Checking for Error Conditions	26
	Using Transient and Persistent STL Containers Together	26
	Converting to Objectivity/C++ STL	28
Chapter 3	Objectivity/C++ STL Examples	33
	Sequential Containers	34
	Splicing Lists	34
	Appending Values to a List	36
	Erasing Vector Elements	37
	Associative Containers	39
	Sorting Characters Using a Set Container	39
	Manipulating the Elements of a Map	41
	Adaptive Containers	43
	Stack Example 1	44
	Stack Example 2	45
	Queue Example	46
	Priority Queue Example	48
Part 2	REFERENCE	
	Objectivity/C++ STL Class Overview	53
	d_basic_string Class	55
	d_list Class	59
	d_map Class	69
	d_multimap Class	77
	d_multiset Class	87
	d_pc_list Class	95
	d_pc_map Class	97
	d_pc_multimap Class	101
	d_pc_multiset Class	105

d_pc_priority_queue Class	109
d_pc_queue Class	111
d_pc_set Class	113
d_pc_stack Class	117
d_pc_vector Class	119
d_set Class	121
d_vector Class	129
priority_queue Class	135
queue Class	137
stack Class	139
Index	141

About This Book

This book, *Objectivity/C++ Standard Template Library*, describes how to use Objectivity/C++ Standard Template Library (Objectivity/C++ STL), an extension of the ObjectSpace Standards<Toolkit> implementation of the ANSI/ISO C++ Standard Template Library and templated string. Objectivity/C++ STL extends ObjectSpace Standards<Toolkit> classes to allow for persistence so your application can store STL containers and strings in an Objectivity/DB database.

For each class, this book describes that part of the Objectivity/C++ STL interface that differs from the corresponding ObjectSpace Standards<Toolkit> interface, as well as constructors, destructors, and nonmember functions. Use this book in conjunction with the ObjectSpace Standards<Toolkit> documentation delivered with Objectivity/C++ STL.

Audience

This book is intended for all users of Objectivity/C++ STL. It assumes that you have experience using Objectivity/DB and some familiarity with the C++ Standard Template Library.

Organization

This book is organized in two parts:

- Part I includes guide chapters that describe the Objectivity/C++ STL classes and tell you how to use them. Objectivity/C++ STL examples are included in this part.
- Part II includes class reference chapters, organized alphabetically, that document sequential container classes, sorted associative container classes, and adaptive container classes.

Conventions and Abbreviations

Navigation

Table of contents entries, index entries, cross-references, and underlined text are hypertext links.

Typographical Conventions

<code>oobackup</code>	Command, literal parameter, code sample, filename, pathname, output on your screen, or Objectivity-defined identifier
<code>installDir</code>	Variable element (such as a filename or a parameter) for which you must substitute a value
Browse FD	Graphical user-interface label for a menu item or button
<code>lock server</code>	New term, book title, or emphasized word

Abbreviations

<i>(administration)</i>	Feature intended for database administration tasks
<i>(FTO)</i>	Objectivity/DB Fault Tolerant Option feature
<i>(DRO)</i>	Objectivity/DB Data Replication Option feature
<i>(ODMG)</i>	Feature conforming to the Object Database Management Group interface

Command Syntax Symbols

<code>[...]</code>	Optional item. You may either enter or omit the enclosed item.
<code>{...}</code>	Item that can be repeated.
<code>... ...</code>	Alternative items. You should enter only one of the items separated by this symbol.
<code>(...)</code>	Logical group of parameters. The parentheses themselves are not part of the command syntax; do not type them.

Command and Code Conventions

In code examples or commands, the continuation of a long line is indented. Omitted code is indicated with the ellipsis (...) symbol. “Enter” refers to the standard key (labelled either Enter or Return) for terminating a line of input.

Getting Help

We have done our best to make sure all the information you need to install and operate Objectivity products is provided in the product documentation. However, we also realize problems requiring special attention sometimes occur.

Technical Support Web Site

You can find answers to frequently asked questions, supported platforms, known bugs, and bug fixes on the Objectivity Technical Support web site. Call Objectivity Customer Support to get access to the site.

How to Reach Objectivity Customer Support

You can contact Objectivity Customer Support by:

- **Telephone:** Call 1.650.254.7100 *or* 1.800.SOS.OBJY (1.800.767.6259) Monday through Friday between 6 AM and 6 PM Pacific Time, and ask for Customer Support.
The toll-free 800 number can be dialed *only* within the 48 contiguous states of the United States and Canada.
- **Fax:** Send a fax to Objectivity at 1.650.254.7171.
- **Electronic Mail:** Send electronic mail to *help@objectivity.com*.

Before You Call

If you need help from Customer Support, please have the following information ready before you contact Objectivity:

- Your name, company name, address and telephone number, fax number, and email address
- Description of your workstation environment, including the type of workstation, its operating system version, compiler or interpreter, and windowing environment
- Information about the Objectivity product you are using, including the version of the Objectivity/DB libraries
- Detailed description of the problem you have encountered

Part 1 GUIDE

Understanding Objectivity/C++ STL

This chapter introduces Objectivity/C++ Standard Template Library (Objectivity/C++ STL) and explains the basic concepts that pertain to Objectivity/C++ STL and its integration with Objectivity/DB.

For comprehensive information about STL, refer to the ANSI/ISO Standard Template Library specification. If you are new to STL, the *STL Tutorial and Reference Guide* by David R. Musser and Atul Saini, published by Addison-Wesley, explains STL and its underlying concepts, describes how to use the STL components and explains the interactions among them, and provides a complete STL reference.

This chapter describes:

- STL components, giving a brief introduction to them
- Objectivity/C++ STL and its relationship to ObjectSpace Standards<Toolkit>
- Implementation classes used by Objectivity/C++ STL container classes for storing elements
- Iterators and how Objectivity/C++ STL handles objects modified by iterator operations
- Storage allocation for Objectivity/C++ STL containers and their contained objects

STL Components

The ANSI/ISO Standard Template Library (STL)—which is part of the C++ Standard Library—consists of various components that work together seamlessly to allow for greater ease of program development and programming problem decomposition.

Following are the main STL components:

- Containers

Containers are generalized classes that hold a collection of data. You can think of them as objects that store other objects of a certain type. There are three categories of containers:

- Sequential containers

Sequential containers organize a collection of objects of the same type into a strictly linear arrangement. A list is an example of a sequential container.

- Associative containers

Associative containers sort objects and retrieve them based on keys. A map is an example of an associative container.

- Adaptive containers (adaptors)

Adaptors modify the interfaces of other containers. They store data using the organization provided by the Objectivity/C++ STL container that is used as the internal implementation, while presenting the adaptor's interface to your application. A stack is an example of an adaptive container that changes the interface of a list; your application can interact with the stack's interface while storing data internally in a list.

- Generic algorithms

Generic algorithms operate on a variety of containers, interacting with the iterators for those containers.

- Iterators

Iterators are abstract data-accessing methods that interact with containers and generic algorithms. Objectivity/C++ STL container classes provide iterators that can be plugged into generic algorithms.

Objectivity/C++ STL

Objectivity/C++ STL is an implementation of STL that allows for persistent storage of STL containers in an Objectivity/DB database. Objectivity/C++ STL is based on the ObjectSpace Standards<Toolkit>, which provides transient STL classes. Objectivity/C++ STL supports the complete set of functions supported by the ObjectSpace Standards<Toolkit>. You can use Objectivity/C++ STL in your Objectivity/C++ applications to create, store, and manipulate instances of ObjectSpace Standards<Toolkit> containers in an Objectivity/DB database.

Objectivity/C++ STL differs from the ObjectSpace Standards<Toolkit> in the following ways:

- In some cases, the Objectivity/C++ STL class interface is not the same as its ObjectSpace Standards<Toolkit> counterpart. The interface for some

functions has been modified to take function parameter names that reflect the type of the class for which the function is used. Part II, which provides the Objectivity/C++ STL class reference chapters, defines and describes these differences for each class.

- ObjectSpace Standards<Toolkit> supports a single container class of a certain type. Objectivity/C++ STL supports two classes of the same container type, a non-persistence-capable container class and a persistence-capable container class. The next section defines these terms and explains the use of these types of classes.

Persistence-Capability

For each container type, Objectivity/C++ STL supports two classes:

- A persistence-capable container class
Persistence-capable classes are classes whose instances can be stored directly in a federated database. Persistence-capable classes obtain their persistence capability by inheriting from a persistence-capable Objectivity/C++ class such as `ooObj`. An Objectivity/C++ STL persistence-capable container class is derived from its STL counterpart and `ooObj`.
Objectivity/C++ STL persistence-capable container classes are named by prefixing the class name with “`d_pc_`”. For example, the Objectivity/C++ STL `d_pc_list` class is the persistence-capable implementation of the ObjectSpace Standards<Toolkit> `list` container class.
- A non-persistence-capable container class
Non-persistence-capable classes are classes whose instances cannot be stored directly in a federated database. However, instances of these classes are stored in a database when they are used as data members of, or base classes for, persistence-capable classes.
Objectivity/C++ STL non-persistence-capable container classes are named by prefixing the class name with “`d_`”. For example, the Objectivity/C++ STL class `d_list`, is the non-persistence capable implementation of the ObjectSpace Standards<Toolkit> `list` container class.

You should read the Objectivity/C++ Data Definition Language book for general information about using persistence-capable and non-persistence capable classes. For example, you can embed a non-persistence-capable container class (such as `d_list`) in a persistence-capable class. However, you *cannot* embed a persistence-capable container class (such as `d_pc_list`) in another persistence-capable class (you should embed the container’s object-reference class—for example, `ooRef(d_pc_list)`—instead).

‘Container’ Term

Industry standard documentation describing STL uses the terms container and collection interchangeably to refer to the STL component that holds data of the same type. To maintain consistency with the ObjectSpace Standards<Toolkit> documentation, which this book supplements, this book uses the terms Objectivity/C++ STL container or Objectivity/C++ STL container class.

Because they share the same term, it is important to remember that an Objectivity/C++ STL container and an Objectivity/DB container are distinct entities.

Objectivity/C++ STL container: As defined previously, an Objectivity/C++ STL container is a generalized class that holds a collection of data of a certain type; the non-persistence-capable class is a base class, the persistence-capable class is derived from its base class counterpart and the Objectivity/C++ class `ooObj`.

Objectivity/DB container: Objectivity/DB supports the clustering of objects within containment objects called containers. Objects placed in the same container are physically clustered together in memory pages and on disk, so access to all objects in a single container is very efficient. An Objectivity/DB container is a persistent object that is an instance of the Objectivity/C++ class `ooContObj` or a subclass of `ooContObj`.

Implementation Classes

Objectivity/C++ STL container classes store their elements internally using various implementation classes. Because of Objectivity/C++ requirements, a container’s implementation class can affect how you use the container in your application. The following subsections describe how implementation classes affect containers; see also the “Implementation” section of each class reference chapter in Part II.

Node-Based Container Classes

Most containers are implemented with one of two Objectivity/C++ STL node classes (`d_list_node` or `d_value_node`) whose instances hold the container’s elements:

- Each sequential container class for a list is implemented as a doubly-linked list of nodes, where each node is an instance of `d_list_node`. These containers include `d_list` and `d_pc_list` (which inherits from `d_list`).

- Each associative container for a map or set is implemented as a red-black tree of nodes, where each node is an instance of `d_value_node`. These containers include `d_map`, `d_multimap`, `d_set`, `d_multiset`, and their derived classes `d_pc_map`, `d_pc_multimap`, `d_pc_set`, and `d_pc_multiset`.
- The ObjectSpace Standards<Toolkit> adaptive containers `queue`, `priority_queue`, and `stack` can optionally be implemented using a `d_list` (and its `d_list_node`) or a `d_vector` (and its `d_value_node`).
The Objectivity/C++ STL adaptive containers `d_pc_queue`, `d_pc_priority_queue`, and `d_pc_stack` are always implemented with either a `d_list` or a `d_vector`.

The `d_list_node` and `d_value_node` classes are persistence-capable. This means that the elements stored in a node-based container can be of any type that is a valid data-member type in a persistence-capable class. The valid types are described in detail in the Objectivity/C++ Data Definition Language book. In brief, a persistence-capable class can contain the following data-member types:

- Primitive types. These include numeric C++ types such as `int`, `char`, and `float`, as well as the portable Objectivity/C++ primitive types.
- Embedded-class types. These include non-persistence-capable classes with valid data members and base classes.
- Object-reference types. These are the object-reference classes generated by the DDL processor for every persistence-capable class—for example, `ooRef(myItem)`, where `myItem` is persistence-capable. **Note:** You cannot embed a persistence-capable class (such as `myItem` itself) directly as an element type.

VArray-Based Container Classes

The sequential container classes for vectors are implemented using Objectivity/C++ variable-size arrays (VArrays). VArrays are similar to C++ arrays, except that they can change in size at runtime. Containers implemented with VArrays include `d_vector` and `d_pc_vector` (which inherits from `d_vector`).

The elements stored in a VArray-based container can be of any type that is a valid element type in a VArray. The valid types are described in detail in the Objectivity/C++ Data Definition Language book. In brief:

- A VArray can contain elements that are of primitive types, embedded-class types, or object-reference types.
- A VArray may not contain another VArray, either directly or indirectly. Consequently, you may not use a vector container as an element of another vector container or any other VArray.

Iterators

Iterators are objects used to traverse and refer to the objects stored in containers. Iterators intercede between containers and generic algorithms, making plug-and-play possible between them.

STL Iterators

STL iterators are grouped hierarchically into five categories: *input*, *output*, *forward*, *bidirectional*, and *random access*.

Each iterator category incorporates and supersedes the capabilities of the categories beneath it in the hierarchy. The iterator category name indicates the additional capability iterators of that type possess. Input iterators are the least powerful; random access iterators are the most powerful. Iterators are often used in pairs to mark a range of objects within a container. The ObjectSpace Standards<Toolkit> documentation provides details on the capabilities of each iterator category.

Iterators of the forward, bidirectional, or random access category can be constant or mutable. A constant iterator prohibits changing the value to which the iterator refers; a mutable iterator allows the value to which it refers to be modified.

Objectivity/C++ STL Iterators

The non-persistence-capable Objectivity/C++ STL string and sequential and associative container classes provide iterators. The persistence-capable Objectivity/C++ STL classes inherit iterators from their non-persistence-capable counterparts. Adaptive containers do not provide iterators because they do not support iteration.

NOTE You cannot embed Objectivity/C++ STL iterators in persistence-capable classes.

Objectivity/C++ provides iterators of class `ooIter`. These iterators should not be confused with Objectivity/C++ STL iterators; they are distinct entities.

Using operator*

Your application can access an object by calling `operator*` on an Objectivity/C++ STL iterator. When this occurs, Objectivity/C++ STL pins the object to which the iterator refers. The object is unpinned when the iterator traverses the container to another object. If the object is stored in a vector, the entire variable-size array that implements the vector is pinned. The variable-size array is unpinned when the object that encloses the array is closed.

When your application calls `operator*` on a constant iterator, a reference to a constant is returned to your application. When it calls `operator*` on a mutable iterator, a reference is returned to your application.

Your application can modify a value of an object belonging to an Objectivity/C++ STL container by calling `operator*` on a mutable iterator that refers to the object. However, you cannot directly modify the contents of those associative containers that contain only keys as their elements. To do so would destroy the ordering of keys and lead to database corruption. To change a key, you must delete it and insert a new one. For this reason, the associative containers that hold only keys and not key-value pairs, `d_multiset`, `d_pc_multiset`, `d_set`, and `d_pc_set`, provide only constant iterators. For completeness, each of these container classes has both `::iterator` and `::const_iterator` types; the mutable iterators behave exactly the same as the constant iterators.

Iterator Summary

Table 1-1 summarizes information about the iterators provided by the Objectivity/C++ STL classes. It identifies the class, its category, and the iterator types the class provides. Persistence-capable container classes are shown along with the non-persistence-capable class from which they are derived and inherit iterators.

Table 1-1: Classes and Iterators They Provide

Class	Category	Iterators
<code>d_basic_string</code>	String	Random access: 1 constant, 1 mutable
<code>d_list</code> , <code>d_pc_list</code>	Sequential	Bidirectional: 1 constant, 1 mutable
<code>d_vector</code> , <code>d_pc_vector</code>	Sequential	Random access: 1 constant, 1 mutable
<code>d_map</code> , <code>d_pc_map</code>	Associative	Bidirectional: 1 constant, 1 mutable
<code>d_multimap</code> , <code>d_pc_multimap</code>	Associative	Bidirectional: 1 constant, 1 mutable
<code>d_multiset</code> , <code>d_pc_multiset</code>	Associative	Bidirectional: 2 constant
<code>d_set</code> , <code>d_pc_set</code>	Associative	Bidirectional: 2 constant

Choosing Between Constant and Mutable Iterators

Any container that your application does not declare as constant is considered mutable. Your application can modify the contents of a mutable container, whereas it cannot modify the contents of a constant container.

For mutable containers, you can use constant or mutable iterators. If you call `operator*` on a mutable iterator, you can modify the value of an object referred to by the iterator. For example, your application could use `*i=` for an assignment operation on a reference returned from a mutable iterator.

When your application accesses an object by calling `operator*` on a mutable iterator, Objectivity/C++ STL marks the object for update. If the mutable iterator refers to an object in a `d_vector` or `d_pc_vector` container, the entire variable-size array that implements the vector is marked for update.

In some cases, you may only perform operations on a mutable container that do not modify its contents, such as those that traverse its objects and those that use nonmutating sequence algorithms. If you were to use a mutable iterator, Objectivity/DB would automatically mark the affected object or variable-size array for update without your explicitly directing it to do so. To prevent unnecessary updates, you should use a constant iterator of type `::const_iterator` for operations that do not modify the contents of the container even when the container is mutable.

Allocating Storage

Every Objectivity/C++ STL container class takes a template parameter that specifies the allocator class to be used to manage storage for that container and its contained objects. The following sections discuss the three ways in which you can specify the allocator class to be used. You can:

- Explicitly specify the default allocator.
If your compiler does not support default template parameters, you must give the allocator class as the value of the `Allocator` parameter.
- Allow the allocator parameter to assume the default.
If your compiler supports default template parameters, you can allow the `Allocator` parameter to assume the default.
- Specify your own allocator class.
If you do not want to use the default allocator class, you can specify your own allocator for the `Allocator` parameter.

Explicitly Requesting the Default Allocator

For each container class, Objectivity/C++ STL provides a default allocator; to use the default allocator, you specify `d_allocator` for the `Allocator` parameter. When the default allocator is used, the Objectivity/C++ STL container and all of the objects that belong to the STL container are allocated within the same Objectivity/DB container.

For non-persistence-capable Objectivity/C++ STL container classes, the enclosing persistent object that embeds the container serves as the clustering directive for allocating the objects. For persistence-capable Objectivity/C++ STL container classes, the persistent container itself is used as the clustering directive.

Implicitly Requesting the Default Allocator

The ANSI/ISO C++ Standard Template Library allows for support of default template parameters. However, not many compilers currently support this feature. If your compiler supports default template parameters, you can allow the `Allocator` parameter to default to the class `d_allocator` instead of explicitly specifying `d_allocator`.

Specifying Your Own Allocator Class

In some cases, you may want to allocate and store objects belonging to a single Objectivity/C++ STL container in different Objectivity/DB containers or even in different databases. For this purpose, you can provide your own allocator class and specify that it be used instead of the default. You can provide a custom allocator class for any of the associative and sequential containers except `d_vector` and `d_pc_vector`, which are implemented as VArrays. The `d_vector` and `d_pc_vector` containers always use the default allocator.

For example, suppose you want the tree node objects that comprise the elements for a map container to be distributed among multiple Objectivity/DB databases. Using your own allocator for the map container, you could provide a clustering directive that creates the tree node objects in the desired databases.

Using Objectivity/C++ STL

You can use the Objectivity/C++ Standard Template Library to develop C++ applications that efficiently store and manipulate objects in Objectivity/DB. This chapter explores using Objectivity/C++ STL with Objectivity/DB. It describes:

- General guidelines for using Objectivity/C++ STL
- Objectivity/C++ capabilities not available in Objectivity/C++ STL
- How to check for error conditions in Objectivity/C++ STL applications
- Transient and persistent Objectivity/C++ STL container interoperability
- Hints on converting an Objectivity/C++ application to Objectivity/C++ STL

In this chapter, code output is shown at the end of each example. This approach conforms with the ObjectSpace Standards<Toolkit>convention.

General Guidelines

When using Objectivity/C++ STL container classes in your application, be sure to perform these tasks as appropriate to the class:

- Instantiate each persistence-capable template class in the DDL file.
- Include the required header files in the DDL file and the C++ source file.

Instantiating Persistence-Capable Template Classes

The Objectivity/C++ STL container classes are created from templates with parameters that specify the type of element to be stored, the storage allocator class, and any required comparators or implementation classes. Furthermore, the node classes used to implement many container classes are themselves template classes with parameters that specify the element type.

For the most part, you use Objectivity/C++ STL template classes as you would any C++ template class; the C++ compiler automatically instantiates (generates a definition for) the template class when it encounters a use of the template name with specified parameters.

The DDL processor requires an extra step when you use template classes created from persistence-capable templates—for every persistence-capable template class to be used in an application, you must provide an explicit instantiation directive in a DDL file. This enables the DDL processor to generate the usual parameterized object-reference and handle classes for the persistence-capable template class. (For more information, see the Objectivity/C++ Data Definition Language book.)

In an Objectivity/C++ STL application, you must put an explicit instantiation directive in a DDL file for every persistence-capable container class that is used by the application to store a particular type of element. Furthermore, if the application uses a container class that is implemented with a persistence-capable node class (either `d_list_node` or `d_value_node`), the DDL file must contain an explicit instantiation directive for the node class as well. The “Application Notes” section of each class reference chapter in Part II gives the template instantiation specifics for that class.

EXAMPLE This sample application uses a persistent list of integers. Consequently, the DDL file `myAppend.ddl` explicitly instantiates the persistence-capable `d_pc_list` class with elements of type `int`. Furthermore, because `d_pc_list` is implemented with the persistence-capable node class `d_list_node`, the DDL file also instantiates `d_list_node` with elements of type `int`.

```
// DDL file myAppend.ddl
#include <d_list.h>           // Obtains d_list class definition

template class d_pc_list< int, d_allocator<int> >;
template class d_list_node<int>;
```

Including Header Files

In order to use STL components in your program, you must use an appropriate `#include` preprocessing directive to include one or more header files. In general, header files are named according to their classes. Some header files contain more than one class; for example, both `d_set` and `d_multiset` are in the `d_set.h` header file.

Objectivity/C++ Capabilities and Objectivity/C++ STL

Objectivity/C++ allows you to create systems of Objectivity/DB objects that are linked through associations, object references, or maps containing object references. Associations and maps provide capabilities that improve the robustness of these systems:

- Propagation of delete and lock operations across associations. When delete propagation is enabled and an object containing associations with other objects is deleted, all the associated objects will also be deleted.
- Referential integrity for bidirectional associations and maps. Referential integrity ensures that if an object is deleted, a reference *to* that object is also deleted.

Figure 2-1 illustrates an Objectivity/DB container *C1* that maintains a bidirectional association with a persistent object *O1*. A bidirectional association means that *C1* has a reference *R1* to *O1* and *O1* has a reference *~R1* to *C1*. If delete propagation has been enabled for this bidirectional association, then deleting *C1* causes *O1*, *R1*, and *~R1* to be deleted automatically. Similarly, because of referential integrity, deleting *O1* causes *R1* to be deleted automatically.

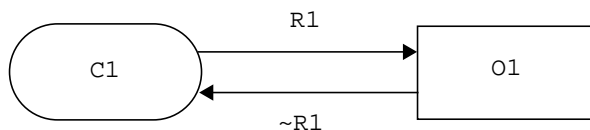


Figure 2-1 Objectivity/DB Container and Contained Object

In contrast, delete and lock operations are not propagated from an Objectivity/C++ STL container to an object referenced by that container, nor is referential integrity enforced for Objectivity/C++ STL containers.

Figure 2-2 illustrates an Objectivity/C++ STL container *C2* that contains an object reference *R2* to the persistent object *O2*. If your application deletes the container *C2*, it is your responsibility to delete *O2* as well. In deleting this container, you remove its object reference *R2* to *O2*, but not *O2* itself. If your application deletes *O2*, it is your responsibility to delete *R2* so that *C2* will not contain a reference to a nonexistent object.

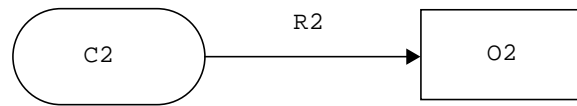


Figure 2-2 Objectivity/C++ STL Container and Contained Object

Similarly, in the case of a container `C2` that has a reference to `O2`, locking `C2` does not lock `O2` if `O2` is in a different Objectivity/DB container or database.

Checking for Error Conditions

To maintain consistency with the ObjectSpace Standards<Toolkit>, Objectivity/C++ STL does not return a status code. Instead, Objectivity/C++ STL uses the error handling facility provided by Objectivity/C++ to handle and report errors encountered during various operations. When an error occurs, Objectivity/C++ STL calls the `ooSignal` function to raise the error signal. This function sets system-defined global variables used as error flags. Your application is responsible for checking these error flags as part of its error handling routine. Objectivity/C++ provides a default error handler; it also allows you to register your own error handler if you do not want to use the default. For complete information on using the Objectivity/C++ error handling facility and writing an error handler, see the chapter that discusses error handling in the Objectivity/C++ book.

Using Transient and Persistent STL Containers Together

You cannot store instances of ObjectSpace Standards<Toolkit>transient containers in an Objectivity/DB database. You can, however, use them along with Objectivity/C++ STL containers. They work together seamlessly through use of generic algorithms.

There are many cases for which you might use transient and persistent STL containers in the same application. For instance, you could use a transient container as an intermediary container in an application process that uses source and destination Objectivity/C++ STL containers. You could move elements to a transient container and perform operations on those elements before moving them to a destination container to be stored in an Objectivity database. You might insert elements into the transient container and sort, merge, or splice them, then move them to the destination container. The following example illustrates the

interoperability between ObjectSpace Standards<Toolkit> transient containers and Objectivity/C++ STL containers.

EXAMPLE This application uses an Objectivity/C++ STL list of integers, so the DDL file `myMerge.ddl` explicitly instantiates the persistence-capable `d_pc_list` and `d_list_node` classes with elements of type `int`.

```
// DDL file myMerge.ddl
#include <d_list.h>
template class d_pc_list< int, d_allocator<int> >;
template class d_list_node<int>;
```

The C++ source file merges the integer elements of the Objectivity/C++ STL `d_pc_list` with those of a ObjectSpace Standards<Toolkit> `deque`, placing the results in a transient `vector`.

```
// C++ source file
#include "myMerge.h" // Generated from myMerge.ddl
#include <deque>
#include <vector>
#include <algorithm>
#include <iostream.h>
{
    typedef d_pc_list< int, d_allocator<int> > pc_list;
    ooHandle(ooContObj) contH;
    ooHandle(pc_list) list_h;

    ... // Open the database and create a new container.
        // contH is the container handle.

    list_h = new (contH) pc_list();

    ... // Check to ensure the object is successfully created.

    deque< int, allocator<int> > t_deque;

    list_h->push_back(1);
    list_h->push_back(2);
    list_h->push_back(3);

    t_deque.push_back(4);
    t_deque.push_back(5);
```

```

// Construct a transient vector of integers with enough
// storage to hold the elements of the list and deque.
vector< int, allocator<int> >
    result(list_h->size() + t_deque.size());

// Merge the list and deque into the vector.
merge(list_h->begin(), list_h->end(), t_deque.begin(),
      t_deque.end(), result.begin());

// Display the contents of the transient vector to
// standard output.
ostream_iterator< int > iter1( cout, " ");
copy( result.begin(), result.end(), iter1 );
cout << endl;
}

1 2 3 4 5

```

Converting to Objectivity/C++ STL

This section describes one approach to converting an application that uses an Objectivity/C++ map into one that uses an Objectivity/C++ STL map.

As its principal task, your conversion application will need to copy the data elements in your Objectivity/C++ map into a functionally equivalent Objectivity/C++ STL map. Objectivity/C++ provides iterators for traversing the data structures of indexes, map classes, and associations. It also provides indexing for traversing variable-size arrays (VArrays). You can use these mechanisms in your copy processes.

The following example gives the DDL (.ddl) file contents and the C++ source (.cxx) file contents for converting an Objectivity/C++ map to an Objectivity/C++ STL map.

EXAMPLE The DDL file `myConvert.ddl` defines a persistence-capable class `Person`. The DDL file also explicitly instantiates the persistence-capable Objectivity/C++ STL class `d_pc_map` with elements that are (key, value) pairs, where each pair maps a string and an object reference to a `Person`. Because `d_pc_map` is implemented with the persistence-capable node class `d_value_node`, the DDL file also explicitly instantiates `d_value_node` with the (key, value) pairs.

```

// DDL file myConvert.ddl
#include <d_map.h>

class Person : public ooObj {
public:
    Person() {}
    Person(char *_address, int _id):
        address(_address), id(_id) {}
    ooVString address;
    int id;
};

template class d_value_node
    < OS_PAIR( ooVString, ooRef(Person) ) >;
template class d_pc_map< ooVString, ooRef(Person),
    less<ooVString>, d_allocator< OS_PAIR
    ( ooVString, ooRef(Person) ) > >;

```

The C++ source file creates an Objectivity/C++ map of class ooMap and assigns a handle to it. Then the code constructs two Person objects and inserts them into the Objectivity/C++ map. Next, the code prints each Objectivity/C++ map element and copies it into the Objectivity/C++ STL map.

After the Objectivity/C++ map elements are copied to the Objectivity/C++ STL map, the code deletes the Objectivity/C++ map and then prints each element of the Objectivity/C++ STL map.

```

// C++ source file
#include "myConvert.h"          // Generated from myConvert.ddl
#include <ooMap.h>
#include <iostream.h>

{
    typedef d_pc_map< ooVString, ooRef(Person), less<ooVString>,
        d_allocator< OS_PAIR
        ( ooVString, ooRef(Person) ) > > oid_map;

    ooHandle(ooContObj) contH;
    ooHandle(Person) personH;
    ooHandle(ooMap) mapH;
    ooHandle(oid_map) stlMapH;

    ... // Open the database and create a new container.
        // contH is the handle to the container.

```

```

mapH = new (contH) ooMap();
... // Check to ensure the object is successfully created.

// Create a Person object with address and ID.
personH = new (contH) Person("1190 Park Ave.,
                             Palo Alto, CA 95444",890801456);
... // Check to ensure the object is successfully created.

mapH->add("John Smith", personH);

// Create another Person object with address and ID.
personH = new (contH) Person("124 Olive St.,
                             Menlo Park, CA 94025", 781547091);
... // Check to ensure the object is successfully created.

mapH->add("Mary Jo", personH);

stlMapH = new (contH) oid_map()
... // Check to ensure the object is successfully created.

ooMapItr mapI = mapH;

cout << "ooMap:" << endl;

ooRef(Person) p;
// Insert (name, oid) pairs into STL map.
while ( mapI.next() ) {
    p = (ooRef(Person)& ) (mapI->oid());
    cout << mapI->name() << ", " << p->address
        << ", " << p->id << endl;
    stlMapH->insert( OS_PAIR( ooVString, ooRef(Person) )
                   (mapI->name(), p) );
}

cout << endl;

// Delete ooMap object

ooDelete(mapH);

cout << "STL map:" << endl;

```

```
oid_map::const_iterator i;
for ( i = stlMapH->begin(); i != stlMapH->end(); ++i ) {
    p = (*i).second;
    cout << (*i).first << ", " << p->address << ", "
        << p->id<<endl;
}
}
```

ooMap:

Mary Jo, 124 Olive St., Menlo Park, CA 94025, 781547091
John Smith, 1190 Park Ave., Palo Alto, CA 95444, 890801456

STL map:

John Smith, 1190 Park Ave., Palo Alto, CA 95444, 890801456
Mary Jo, 124 Olive St., Menlo Park, CA 94025, 781547091

Objectivity/C++ STL Examples

This chapter explores some of the ways in which you can use Objectivity/C++ STL to develop your applications. Objectivity/C++ STL offers you a predefined, standardized group of container classes for commonly used collections such as lists, sets, and maps. Prior to Objectivity/C++ STL, you might have had to define these collection classes yourself or use a proprietary, commercial version of them.

This chapter presents examples that show the DDL file containing the requisite statements for using a particular container. Following each DDL file is the C++ source file. The C++ source file includes the code segment illustrating use of the container, it does not illustrate the standard procedures your Objectivity/C++ application would also contain to handle a database transaction.

Though these examples do not address all non-persistence-capable and persistence-capable Objectivity/C++ STL container classes, each category—sequential, associative, and adaptive containers—is represented by one or more examples.

This chapter includes code segment examples that use Objectivity/C++ STL containers to:

- Splice two lists and add elements to a list
- Progressively erase elements of a vector
- Sort characters using a set container
- Modify the value of an existing map (key, value) pair and insert a new one
- Push values onto, and pop them off, a stack and a priority queue

In this chapter, code output is shown at the end of each example. This approach conforms with the ObjectSpace Standards<Toolkit> convention.

Sequential Containers

Sequential containers organize elements of the same type in a linear arrangement. Lists and vectors, two types of sequential containers having different capabilities, both offer a decided advantage over arrays because they can grow and shrink dynamically as you add and delete elements. One of the major differences between lists and vectors is that list element insertions never invalidate any iterators, and list element deletions invalidate only those iterators that refer to the deleted elements. However, when you erase a vector element, all iterators and references to elements after the point of erasure are invalidated.

The examples in this subsection are as follows:

- “Splicing Lists” on page 34, shows how to splice integers stored in one list into another.
- “Appending Values to a List” on page 36, shows how to insert floating point values at the end of a list.
- “Erasing Vector Elements” on page 37, shows how to use the vector erase member function to remove vector elements until none remain.

Splicing Lists

Lists are linked structures that can be rearranged internally through relinking. Because of how they are implemented and the group of splice member functions they provide, lists offer an efficient means of rearranging elements. This example shows how to splice integers stored in one list into another.

EXAMPLE The DDL file `mySplice.ddl` defines a persistence-capable class `SpliceList`, which embeds the non-persistence-capable class `d_list` with elements of type `int`. Because `d_list` is implemented with the persistence-capable node class `d_list_node`, the DDL file explicitly instantiates `d_list_node` with elements of type `int`.

```
// DDL file mySplice.ddl
#include <d_list.h>           // Obtains d_list class definition

template class d_list_node<int>;

class SpliceList : public ooObj {
public:
    // Construct _list to contain all the elements
    // in the range ['first', 'last').
    SpliceList(const int* first, const int* last) :
        _list(first, last) {}
}
```

```

    // Data member
    d_list < int, d_allocator<int> > _list;
    ...
};

```

The C++ source file creates two objects of class `SpliceList` (called `sourceList` and `targetList`) and splices elements from one into the other.

The code constructs `sourceList`, initializing it with the two integers stored in `myArray2`. Then it constructs `targetList`, initializing it with two integers stored in `myArray1`. Next, the code tests to ensure that the objects were successfully created. If so, the code calls the `splice` member function on `d_list`, inserting the specified range of elements belonging to `sourceList` at the beginning of `targetList`. Finally, the code iterates through `targetList`, printing its contents to standard output.

```

// C++ source file
#include <iostream.h>
#include "mySplice.h"          // Generated from mySplice.ddl

{
    typedef d_list< int, d_allocator<int> > int_list;

    const int myArray1[] = { 2, 8 };
    const int myArray2[] = { 4, 9 };

    ooHandle(ooContObj) contH;
    ooHandle(SpliceList) sourceList, targetList;

    ... // Open the database and create a new container.
        // contH is the container handle.

    targetList = new ( contH ) SpliceList(myArray1, myArray1+2);
    sourceList = new ( contH ) SpliceList(myArray2, myArray2+2);
    ... // Check to ensure the objects are successfully created.

    // Remove the elements from sourceList in the range
    // [sourceList->_list.begin(), sourceList->_list.end())
    // and insert them at position targetList->_list.begin().
    targetList->_list.splice(targetList->_list.begin(),
        sourceList->_list,
        sourceList->_list.begin(),
        sourceList->_list.end() );

    // Iterate through targetList, printing out its
    // contents to show the results of splicing.

```

```

        int_list::const_iterator i;

        for ( i = targetList->_list.begin();
              i != targetList->_list.end();
              ++i ) {
            cout << *i << " ";
        }
        cout << endl;
    }

4 9 2 8

```

Appending Values to a List

This example shows how to insert floating point values at the end of a list.

EXAMPLE The DDL file `myAppend.ddl` explicitly instantiates the persistence-capable class `d_pc_list` with elements of type `float`. Because the class `d_pc_list` is implemented with the persistence-capable node class `d_list_node`, the DDL file also explicitly instantiates `d_list_node` with elements of type `float`.

```

// DDL file myAppend.ddl
#include <d_list.h>           // Obtains d_list class definition

template class d_list_node<float>;
template class d_pc_list< float, d_allocator<float> >;

```

The C++ source file creates a persistent object of class `d_pc_list`, and assigns it to the handle `myList_h`. The code inserts floating point values at the end of the list using the `push_back` member function inherited from `d_list`. The `float_list` type definition is added for program convenience.

```

// C++ source file
#include <iostream.h>
#include "myAppend.h"           // Generated from myAppend.ddl

{
    typedef d_pc_list< float, d_allocator<float> > float_list;

    ooHandle(ooContObj) contH;
    ooHandle(float_list) myList_h;
    ... // Open the database and create a new container.
        // contH is the container handle.
    myList_h = new ( contH ) float_list();
    ... // Check to ensure the object is successfully created.
}

```

```

// Insert floats at the end of the list.
myList_h->push_back(1.2);
myList_h->push_back(2.2);
myList_h->push_back(3.3);
myList_h->push_back(4.1);

// Print the elements of the list from beginning to end.
float_list :: const_iterator i;

for ( i = myList_h->begin(); i != myList_h->end(); ++i ) {
    cout << *i << " ";
}
cout << endl;
}

1.2 2.2 3.3 4.1

```

Erasing Vector Elements

This example shows how to erase elements from a vector.

EXAMPLE The DDL file `myVector.ddl` defines a class `V` that contains an embedded `d_vector` with integer elements.

```

// DDL file myVector.ddl
#include <d_vector.h> // Obtains d_vector class definition

class V : public ooObj {
public:
    V (const int* first, const int* last) :
        _integers(first, last) { }
    // Data member
    d_vector<int, d_allocator<int> > _integers;
    ...
};

```

The C++ source file constructs the embedded `d_vector` to contain six integer elements. Using the `erase` member function, the code erases the first element, prints the contents, erases the last element, prints the contents, and so forth until the vector is empty.

```

// C++ source file
#include <iostream.h>
#include "myVector.h" // Generated from myVector.ddl

```

```
{
    // Assign six integers to array_A.
    const int array_A [] = { 1, 4, 9, 16, 25, 36 };

    ooHandle(ooContObj) contH;
    ooHandle(V) VO_h;

    ... // Open the database and create a new container.
        // contH is the handle to the container.

    VO_h = new (contH) V (array_A, array_A+6);
    ... // Check to ensure the object is successfully created.

    // Print array contents: six elements.
    int i;
    for ( i = 0; i < VO_h->_integers.size(); i++) {
        cout << VO_h->_integers[i] << " ";
    }
    cout << endl;

    // Erase first array element.
    VO_h->_integers.erase ( VO_h->_integers.begin() ) ;

    // Print array contents: five elements.
    for ( i = 0; i < VO_h->_integers.size(); i++) {
        cout << VO_h->_integers[i] << " ";
    }
    cout << endl;

    // Erase last array element.
    VO_h->_integers.erase ( VO_h->_integers.end() - 1);

    // Print array contents: four elements.
    for ( i = 0; i < VO_h->_integers.size(); i++) {
        cout << VO_h->_integers[i] << " ";
    }
    cout << endl;

    // Erase all but first and last elements.
    VO_h->_integers.erase ( VO_h->_integers.begin() + 1,
                          VO_h->_integers.end() - 1);

    // Print array contents: two elements.
    for ( i = 0; i < VO_h->_integers.size(); i++ ) {
        cout << VO_h->_integers[i] << " ";
    }
}
```

```

        cout << endl;

        // Erase all elements.
        VO_h->_integers.erase();
    }

1 4 9 16 25 36
4 9 16 25 36
4 9 16 25
4 25

```

Associative Containers

Sets and maps are two types of associative containers; sets store only keys and maps store (key, value) pairs. Multisets and multimaps, the two other types of associative containers not shown in these examples, allow duplicate keys to be stored, while their set and map counterparts do not.

The examples in this section are as follows:

- “Sorting Characters Using a Set Container” on page 39 shows how to sort items from a list into a persistent set.
- “Manipulating the Elements of a Map” on page 41 shows how to modify the value portion of a (key, value) pair and how to insert a pair.

Sorting Characters Using a Set Container

You can use a set or multiset to sort a collection of items instead of using a list and its `sort` member function. Items you put in a set or multiset are sorted automatically. This example sorts characters from a list into a set.

EXAMPLE The DDL file `mySort.ddl` explicitly instantiates the persistence-capable classes `d_pc_list` and `d_pc_set` with elements of type `char`. Because these classes are implemented with the persistence-capable node classes `d_list_node` and `d_value_node`, respectively, the DDL file also explicitly instantiates the node classes with elements of type `char`.

```

// DDL file mySort.ddl
#include <d_list.h>           // Obtains list class definitions
#include <d_set.h>           // Obtains set class definitions

template class d_list_node<char>;
template class d_pc_list< char, d_allocator<char> >;

```

```
template class d_value_node<char>;
template class d_pc_set< char, less<char>, d_allocator<char> >;
_____
```

The C++ source file creates a handle called `myList` to a persistent object of class `d_pc_list`. Then the code stores the characters that comprise the string `myStr` into the list. To do this, the code uses the list's `push_back` member function. The characters are stored in this order—`b,d,a,h,e,g,f,h,c`—the order in which they occur in the string.

Next, the code puts the characters stored in `myList` into the set of class `d_pc_set` called `mySet`, which automatically sorts them. To accomplish this, the code uses the list's constant iterator to traverse the list and the set's `insert` member function to insert the characters.

Finally, the code prints the characters to standard output to illustrate the sorted order of the characters in the set.

```
// C++ source file
#include "mySort.h"           // Generated from mySort.ddl
#include <iostream.h>

{
    typedef d_pc_list<char, d_allocator<char> > char_list;
    typedef d_pc_set<char, less<char>, d_allocator<char> >
                                   char_set;
    const string myStr = "bdahegfhc";

    ooHandle(ooContObj) contH;
    ooHandle(char_list) myList;
    ooHandle(char_set) mySet;
    ... // Open the database and create a new container.
        // contH is the handle to the container.

    myList = new (contH) char_list();
    ... // Check to ensure the object is successfully created.

    // Insert the contents of the string into the list.
    string::const_iterator j;
    for ( j = myStr.begin(); j != myStr.end(); ++j ) {
        myList->push_back(*j);
    }

    // Put the characters from the list into the set.
    char_list::const_iterator i;
    for (i = myList->begin(); i != myList->end(); ++i) {
        mySet.insert(*i);
    }
}
```

```

// Display the sorted content of mySet.
char_set::const_iterator h;
for ( h = myList->begin(); h != mySet.end(); ++h ) {
    cout << *h << " ";
}

cout << endl;
ooDelete(myList);
}

a b c d e f g h

```

Manipulating the Elements of a Map

Similar to sets, maps provide for fast retrieval based on keys. Maps (and multimaps) differ from sets (and multisets) in that a map element consists of not only a key, but data that is associated with the key in a pair. This is referred to as a (key, value) pair. Maps differ from multimaps in that each key stored in a map must be unique; each map contains at most one pair for each key. A multimap allows you to associate more than one value with a single key. The following example illustrates that a map's key must be unique. The code segment it presents attempts to insert a second instance of a (key, value) pair in the map.

EXAMPLE The DDL file `myMap.ddl` defines a persistence-capable class `mapst` that embeds the non-persistence-capable class `d_map` with key elements of type `char` and value elements of type `int`. Because `d_map` is implemented with the persistence-capable node class `d_value_node`, the DDL file explicitly instantiates `d_value_node` with (key, value) pairs of characters and integers.

```

// DDL file myMap.ddl
#include <d_map.h> // Obtains d_map class definition

typedef d_map< char, int, less<char>,
             d_allocator< OS_PAIR (char, int) > > maptype;

class mapst: public ooObj {
public:
    maptype _map;
    ...
};

template class d_value_node< OS_PAIR(char, int) > ;

```

The C++ source file maps values containing Roman numerals to keys that are their decimal equivalents. For the second pair, a mistake is deliberately introduced in assigning the decimal value of 20 to the key of Roman numeral x. The code prints the second element to standard output, then corrects the mistake and prints out the element again.

Next, the code inserts a (key, value) pair. The code tests for a successful insertion, which is the case, and so prints out the inserted pair. The code then tries to insert the same (key, value) pair, but because the key is not unique, the attempted insertion fails. Instead of printing out the successful insertion message, the code exercises the `else` statement and reports on the existing (key, value) pair it had previously inserted.

```
// C++ source file
#include "myMap.h"           // Generated from myMap.ddl
#include <iostream.h>

{
    ooHandle(ooContObj) contH;
    ooHandle(maptst) map1_h;

    ... // Open the database and create a new container.
        // contH is the handle to the container.

    map1_h = new (contH) maptst ( );
    ... // Check to ensure the object is successfully created.

    // Store mappings between Roman numerals and decimals.
    map1_h->_map['l'] = 50;
    // Introduce a deliberate mistake.
    map1_h->_map['x'] = 20;
    map1_h->_map['v'] = 5;
    map1_h->_map['i'] = 1;

    cout << "_map['x'] = " << map1_h->_map ['x'] << endl;

    // Correct the previous mistake.
    map1_h->_map['x'] = 10;

    cout << "_map['x'] = " << map1_h->_map ['x'] << endl;

    pair< maptype::iterator, bool > p;
    p = map1_h->_map.insert (OS_PAIR(char, int) ('c', 100) );
    if (p.second) {
        cout << "First insertion was successful" << endl;
    }
}
```

```
p = map1_h->_map.insert (OS_PAIR(char, int) ('c', 100));
if (p.second) {
    cout << "Second insertion was successful" << endl;
}
else {
    cout << "Existing pair "
    << (*(p.first)).first
    << " -> "
    << (*(p.first)).second << endl;
}

_map['x'] = 2

_map['x'] = 1

First insertion was successful
Existing pair c -> 100
```

Adaptive Containers

An adaptive container changes the interface of another container by presenting the interface of one container, such as a stack, to your application while using as its internal implementation that of another container, such as a vector or a list. Although your application calls the stack interface, the data you pass to it is stored in the container of either a list or a vector, depending on which internal implementation the adaptive container uses.

Adaptive containers are provided for stacks (first-in, last-out data structures) and queues (first-in, first-out data structures). The examples in this section are as follows:

- “Stack Example 1” on page 44 shows a persistence-capable class that embeds a stack. The stack uses a vector implementation.
- “Stack Example 2” on page 45 shows a persistence-capable stack class. The stack uses a vector implementation.
- “Queue Example” on page 46 shows how to use a queue that uses a list implementation.
- “Priority Queue Example” on page 48 shows how to use a priority queue that uses a vector implementation. A priority queue sorts its elements.

Stack Example 1

This example shows a non-persistence-capable stack embedded in a persistence-capable class. The stack uses a vector implementation.

EXAMPLE The DDL file `myStack.ddl` defines a persistence-capable class `MyStack` that embeds a stack with integer elements and the `d_vector` implementation class.

```
// DDL file myStack.ddl
#include <d_vector.h>    // Obtains d_vector class definition
#include <stack>

class MyStack : public ooObj {
public:
    stack<int, d_vector<int, d_allocator<int> > > a;
    ...
};
```

The C++ source file uses the `push` member function of the embedded stack to push three values onto it. Then the code prints the value at the top of the stack—the last one pushed on it—after which the value is popped off. The code does this successively for all three elements.

```
// C++ source file
#include <iostream.h>
#include "myStack.h"    // Generated from myStack.ddl

{
    ooHandle(ooContObj) contH;
    ooHandle(MyStack) stack_h;

    ... // Open the database and create a new container.
        // contH is the handle to the container.

    stack_h = new (contH) MyStack();
    ... // Check to ensure the object is successfully created.

    stack_h->a.push(100);
    stack_h->a.push(200);
    stack_h->a.push(300);
```

```

    // Print the value at the top of the stack, then pop it off.
    while (!stack_h->a.empty()) {
        cout << stack_h->a.top() << " ";
        stack_h->a.pop();
    }
    cout << endl;
}

300 200 100

```

Stack Example 2

This example shows a persistent stack that uses a vector implementation. A persistent stack is stored directly in an Objectivity/DB database.

EXAMPLE The DDL file `myStackPC.ddl` explicitly instantiates the persistence-capable class `d_pc_stack` with integer elements and the `d_vector` implementation class.

```

// DDL file myStack.ddl
#include <d_vector.h> // Obtains d_vector class definition
#include <d_stack.h> // Obtains d_stack class definition

template class d_pc_stack<int, d_vector<int,
                          d_allocator<int> > >;

```

The C++ source file pushes values onto a `d_pc_stack`, printing the value at the top of the stack, then popping it off.

```

// C++ source file
#include <iostream.h>
#include "myStackPC.h" // Generated from myStackPC.ddl

{
    typedef d_pc_stack<int, d_vector<int,
                      d_allocator<int> > > pc_stk;

    ooHandle(ooContObj) contH;
    ooHandle(pc_stk) stack1_h;

    ... // Open the database and create a new container.
        // contH is the handle to the container.

```

```

stack1_h = new (contH) pc_stk();
... // Check to ensure the object is successfully created.

// Push values onto the stack.
stack1_h->push( 1 );
stack1_h->push( 5 );
stack1_h->push( 42 );

// Print the value at the top of the stack,
// then pop it off the stack.
while (!(start1_h->empty())) {
    cout << stack1_h->top() << " ";
    stack1_h->pop();
}
cout << endl;
}

42 5 1

```

Queue Example

This example illustrates a queue that uses a list as its internal implementation.

EXAMPLE The DDL file `myQueue.ddl` defines a persistence-capable class `MyPCQueue` that embeds a queue with a `d_list` implementation class and with integer elements. Because `d_list` is implemented with the persistence-capable node class `d_list_node`, the DDL file explicitly instantiates `d_list_node` with elements of type `int`.

```

// DDL file myQueue.ddl
#include <d_list.h>           // Obtains list class definitions
#include <queue>

class MyPCQueue : public ooObj {
public:
    queue< int, d_list< int, d_allocator<int> > > a;
    ...
};

template class d_list_node<int>;

```

The C++ source file uses the `push` member function of the embedded queue to push three integer values onto the queue. Then, the code prints the value at the front of the queue—the first one pushed on it—after which the value is popped off. This is done successively for all three elements.

```
// C++ source file
#include "myQueue.h"           // Generated from myQueue.ddl
#include <iostream.h>

{
    ooHandle(ooContObj) contH;
    ooHandle(MyPCQueue) queue_h;

    ... // Open the database and create a new container.
        // contH is the handle to the container.

    queue_h = new (contH) MyPCQueue();
    ... // Check to ensure the object is successfully created.

    // Push values onto the queue.
    queue_h->a.push(100);
    queue_h->a.push(200);
    queue_h->a.push(300);

    // Print the value at the front of the
    // queue, then pop it off the queue.
    while (!(queue_h->a.empty())) {
        cout << queue_h->a.front() << " ";
        queue_h->a.pop();
    }

    cout << endl;
}

100 200 300
```

Priority Queue Example

This example illustrates the use of a persistent priority queue that uses a vector implementation. A priority queue is an adaptive container that uses the comparator you specify to sort the elements stored in the container. In this example, integers pushed onto the priority queue are sorted in descending order.

EXAMPLE The DDL file `myPCPQ.ddl` explicitly instantiates the persistence-capable class `d_pc_priority_queue` with the `d_vector` implementation class and with integer elements.

```
// DDL file myPCPQ.ddl
#include <d_vector.h>      // Obtains d_vector class definition
#include <queue>

template class d_pc_priority_queue<int,
    d_vector<int, d_allocator<int> >, less<int> >;
```

The C++ source file pushes integer values onto a priority queue that sorts them in descending order. After pushing the three values, the code pops them and prints them to standard output.

```
// C++ source file
#include "myPCPQ.h"      // Generated from myPCPQ.ddl
#include <iostream.h>

{
    typedef d_pc_priority_queue<int,
        d_vector<int, d_allocator<int> >, less<int> > pc_p_queue;

    ooHandle(ooContObj) contH;
    ooHandle(pc_p_queue) queue_h;

    ... // Open the database and create a new container.
        // contH is the handle to the container.

    queue_h = new (contH) pc_p_queue();
    ... // Check to ensure the object is successfully created.

    // Push values onto the queue.
    queue_h->push(32);
    queue_h->push(211);
    queue_h->push(88);
```

```
// Pop the values off the queue and print them.
while (!(queue_h->empty())) {
    cout << queue_h->top() << " ";
    queue_h->pop();
}
cout << endl;
}
```

```
211 88 32
```

Part 2 REFERENCE

Objectivity/C++ STL Class Overview

This chapter provides an overview of the Objectivity/C++ STL class reference chapters, which comprise the remainder of this book. Organized alphabetically, it documents sequential, sorted associative, and adaptive container classes.

Objectivity/C++ STL supports two versions of each container class:

- A persistence-capable container class, named by prefixing the class name with “d_pc_”.
- A non-persistence-capable container class, named by prefixing the class name with “d_”.

Objectivity/C++ STL extends the ObjectSpace Standards<Toolkit> so that STL containers can be stored in an Objectivity/DB database. To do this, Objectivity/C++ STL modifies the interface of class constructor, destructor, member, and nonmember functions that take function parameters of STL container class types.

Objectivity/C++ STL supports the complete set of public member and nonmember functions and operators supported by the ObjectSpace Standards<Toolkit>. However, in these reference pages, each Objectivity/C++ STL class description includes only those aspects of the interface that have been modified from the ObjectSpace Standards<Toolkit> interface. For information on the complete interface for any class, see the ObjectSpace Standards<Toolkit> documentation.

Objectivity/C++ STL Class Summary

The following table lists the Objectivity/C++ STL classes with their STL and Objectivity categories.

Class	STL Category	Objectivity Category
<code>d_basic_string</code>	String	Non-persistence-capable
<code>d_list</code>	Sequential container	Non-persistence-capable
<code>d_map</code>	Associative container	Non-persistence-capable
<code>d_multimap</code>	Associative container	Non-persistence-capable
<code>d_multiset</code>	Associative container	Non-persistence-capable
<code>d_pc_list</code>	Sequential container	Persistence-capable
<code>d_pc_map</code>	Associative container	Persistence-capable
<code>d_pc_multimap</code>	Associative container	Persistence-capable
<code>d_pc_multiset</code>	Associative container	Persistence-capable
<code>d_pc_priority_queue</code>	Adaptive container	Persistence-capable
<code>d_pc_queue</code>	Adaptive container	Persistence-capable
<code>d_pc_set</code>	Associative container	Persistence-capable
<code>d_pc_stack</code>	Adaptive container	Persistence-capable
<code>d_pc_vector</code>	Sequential container	Persistence-capable
<code>d_set</code>	Associative container	Non-persistence-capable
<code>d_vector</code>	Sequential container	Non-persistence-capable
<code>priority_queue</code>	Adaptive container	(Non-persistence-capable) ^a
<code>queue</code>	Adaptive container	(Non-persistence-capable) ^a
<code>stack</code>	Adaptive container	(Non-persistence-capable) ^a

- a. ObjectSpace Standards<Toolkit> classes that, when implemented with a non-persistence-capable Objectivity/C++ STL class, can be used as you would use a non-persistence-capable Objectivity/C++ STL class.

d_basic_string Class

Inheritance: **d_basic_string**

`d_basic_string` is a template that supports persistent storage of C++ strings. `d_basic_string` implements the standard, templated string class defined by the ANSI/ISO C++ standard committee.

`d_basic_string` defines a `d_string` template class that supports variable-length strings of characters of type `char`.

`d_basic_string` is non-persistence-capable. You can embed a `d_basic_string` template class in persistence-capable classes, and you can use it as a base class from which you can derive persistence-capable subclasses.

`d_basic_string` is the Objectivity/C++ STL implementation of the ObjectSpace Standards<Toolkit> `basic_string`. Objectivity/C++ STL `d_basic_string` has an identical interface to ObjectSpace Standards<Toolkit> `basic_string` with the exception that `d_basic_string` member functions, operators, and nonmember functions that take the class type as a function parameter use `d_basic_string` instead of `basic_string`. For information on these functions, see the online ObjectSpace Standards<Toolkit> documentation.

Class Declaration

class d_basic_string

```
#include <d_string.h>

template < class charT, class Traits=char_traits<charT>,
          class SAllocator=d_allocator<charT> >
class d_basic_string

typedef d_basic_string < char, char_traits_char,
                      d_allocator<char> > d_string;
```

Parameters *charT*

The type of element comprising the string. For the class `d_string`, the element type is `char`.

Traits

The class that defines the string's character traits for elements of type `charT`. The trait class defines characteristics of the class `d_basic_string`, such as how the class `d_basic_string` assigns and compares elements, calculates the length of strings, and defines the end-of-string value.

SAllocator

The allocator to be used by `d_basic_string`.

Implementation

`d_basic_string` is implemented as an Objectivity/DB variable-size array (VArray) of `char` elements. The internal allocator encapsulates the VArray-of-char allocation.

Because VArrays cannot be elements of other VArrays, `d_basic_string` cannot be an element type for another VArray. Consequently, you cannot have a `d_vector` or a `d_pc_vector` with elements of class `d_basic_string`, because Objectivity/C++ STL vectors are also implemented as VArrays.

Application Notes

Objectivity/C++ STL `d_string` and Objectivity/C++ `d_String` (note the capital S in `d_String`) are distinct classes with different capabilities. For example, `d_string` provides for the specification of certain character traits whereas `d_String` does not.

Iterators

`d_basic_string` provides constant and mutable random access iterators.

When your application accesses a character belonging to a string by calling `operator*` on a mutable iterator, Objectivity/C++ STL pins the variable-size array to which the iterator refers and marks the entire array for update. It unpins the array when the object that encloses it is closed.

You cannot embed the `d_basic_string` iterators in persistence-capable classes.

iterator

A mutable random access iterator that traverses the characters of a string.

```
d_basic_string< charT, Traits, SAllocator >::iterator
```

const_iterator

A constant random access iterator that traverses the characters of a string.

```
d_basic_string< charT, Traits, SAllocator >::const_iterator
```


d_list Class

Inheritance: `d_list`

`d_list` is a sequential container that enables fast insertions and deletions at any position in the sequence, but does not allow for index-based random access. List containers offer the best performance of sequential containers in cases for which you frequently insert and delete interior elements if you do not need to move elements randomly from one position in the sequence to another. List sequences can be rearranged by relinking the list node objects, allowing for constant-time insertions and deletions. This is not true of sequential containers, such as vectors, that support constant time operations at one or both ends only.

`d_list` is a non-persistence-capable class. You can embed class `d_list` in persistence-capable classes, and you can use it as a base class from which you can derive persistence-capable subclasses.

For information on `d_pc_list`, the Objectivity/C++ STL persistence-capable list class, see “`d_pc_list` Class” on page 95.

`d_list` is the non-persistence-capable Objectivity/C++ STL implementation of the ObjectSpace Standards<Toolkit> `list`. In addition to the functions described in this chapter, class `d_list` provides functions for inserting, deleting, and sorting list elements, splicing two lists, filtering lists, assigning and comparing list elements, and merging two lists. For information on these and other functions that operate on lists, see the ObjectSpace Standards<Toolkit> documentation.

Class Declaration

class d_list

```
#include <d_list.h>

template < class T, class Allocator=d_allocator< T > >
class d_list
```

Parameters

T

The type of element stored in the list container. List elements can be of any non-persistence-capable class, including Objectivity/C++ STL non-persistence-capable container classes.

Allocator

The allocator to be used by `d_list`. The default allocator encapsulates the storage allocation and deallocation of list nodes with elements of type `T`. This allocator clusters the list nodes of a `d_list` together with the persistent object that embeds the `d_list`.

Implementation

An Objectivity/C++ STL list container is implemented as a doubly-linked list of nodes, where each node is an instance of the persistence-capable `d_list_node` template class. Each node contains object references to the nodes on its left and right, plus a data member for storing a list element.

Because `d_list_node` is persistence-capable, the element type `T` cannot be persistence-capable (persistence-capable classes are not valid embedded-class types). However, the element type can be an object-reference type for a persistence-capable class—for example `ooRef(myItem)`, where `myItem` is persistence-capable.

Application Notes

Because `d_list` is implemented with the persistence-capable template class `d_list_node`, the DDL file must provide an explicit instantiation directive for `d_list_node` with each element type to be stored.

EXAMPLE To use a `d_list` to store integers, you must add the following explicit instantiation directive to the DDL file:

```
// DDL file
template class d_list_node<int>
```

To store instances of a non-persistence-capable class `Item` in a `d_list` container, you add the following explicit instantiation directive to the DDL file:

```
// DDL file
template class d_list_node<Item>
```

Iterators

`d_list` provides constant and mutable bidirectional iterators. An iterator for `d_list` contains a reference of class `ooRef` to the `d_list_node` of type `T` to which the iterator currently refers.

You cannot embed the list iterators in persistence-capable classes.

iterator

A mutable bidirectional iterator that sequentially traverses objects in a list container.

```
d_list< T, Allocator >::iterator
```

const_iterator

A constant bidirectional iterator that sequentially traverses objects in a list container.

```
d_list< T, Allocator >::const_iterator
```

Reference Summary

`d_list` is the non-persistence-capable Objectivity/C++ STL implementation of the ObjectSpace Standards<Toolkit> `list`. Objectivity/C++ STL `d_list` has an identical interface to ObjectSpace Standards<Toolkit> `list` with the exception of the functions shown in the following table.

Constructors and destructor	<code>d_list</code> <code>~d_list</code>
Replacing the list's contents	<code>operator=</code>
Merging elements into a list	<code>merge</code>
Inserting elements into a list	<code>splice</code>
Exchanging the contents of two lists	<code>swap</code>
Comparing the contents of two lists	<code>operator==</code> <code>operator<</code>

Constructors and Destructor

d_list

Constructor for `d_list`. The `d_list` constructor is overloaded to allow you to construct lists initialized in different ways.

1. `d_list();`
2. `explicit d_list (const Allocator& alloc);`
3. `explicit d_list(
 size_type n,
 const T& value=T(),
 const Allocator& alloc=Allocator());`
4. `d_list(
 const T* first,
 const T* last,
 const Allocator& alloc=Allocator());`
5. `d_list(
 const_iterator first,
 const_iterator last,
 const Allocator& alloc=Allocator());`

```

6.  template < class InputIterator >
    d_list (
        InputIterator first,
        InputIterator last,
        const Allocator& alloc=Allocator());
7.  d_list (const d_list< T, Allocator >& other);

```

Parameters

alloc

Constant reference to the allocator to be used to manage storage for the list.

n

Number of elements, or nodes, that the list is constructed to contain initially.

value

Constant reference to the value used to initialize the elements of the list. The list is constructed to contain *n* nodes containing copies of *value*. The parameter *value* is of type *T*, which can be any non-persistence-capable data type, including non-persistence-capable Objectivity/C++ STL container classes.

first

If the compiler does not support template parameters for member functions, *first* must be either a constant pointer or a constant iterator for a list, which refers to the first element in a range of elements to be copied to that list to initialize it. The new list is constructed to contain a copy of the elements comprising the range [*first*, *last*), which determines its size.

If the compiler supports template parameters for member functions, *first* can be an iterator type defined as a template parameter. This iterator can belong to another container.

last

If the compiler does not support template parameters for member functions, *last* must be either a constant pointer or a constant list iterator that refers to the position after the final element comprising the range [*first*, *last*). The range does not include *last*. The newly constructed list contains a copy of the elements comprising the range [*first*, *last*), which determine its size.

If the compiler supports template parameters for member functions, *last* can be an iterator type defined as a template parameter.

other

Constant reference to a list whose elements are copied and used to initialize the constructed list container.

~d_list

Destructor for `d_list`.

```
~d_list;
```

Discussion The list destructor destroys the list and erases all of its nodes, deallocating the storage used for it.

Member Functions

operator=

Replaces the contents of this list with a copy of list *other*.

```
d_list< T, Allocator >& operator=(  
    const d_list < T, Allocator >& other);
```

Parameters *other*

Constant reference to the list whose contents are copied and used to replace the contents of this list.

merge

Integrates the elements specified by list *other* into this list, sorting them using `operator<`.

```
void merge(d_list< T, Allocator >& other);
```

Parameters *other*

Reference to the list whose elements are sorted into the existing list, based on `operator<`.

Discussion This function assumes that both lists whose elements are to be merged are already sorted using `operator<`.

merge

Merges the elements specified by list *other* into this list, sorting the merged list entries using `compare`.

```
void merge(  
    d_list< T, Allocator >& other,  
    bool (*compare) (const T&, const T&));
```

Parameters *other*
 Reference to the list whose elements are sorted into this target list according to the specified comparison.

Discussion This function assumes that both lists have already been sorted using `compare`.

splice

Inserts all elements from list *other* into this list at the position specified by *pos*.

```
void splice(iterator pos, d_list< T, Allocator >& other);
```

Parameters *pos*
 Position where the element is to be inserted.

other
 Reference to the list whose contents are inserted into this list at *pos*.

splice

Moves the element at position *from* in list *other* and inserts it at position *to* of this list.

```
void splice(
    iterator to,
    d_list< T, Allocator >& other,
    iterator from);
```

Parameters *to*
 Position where the element is to be inserted.

other
 Reference to the list containing the element to be moved and inserted at *to*.

from
 Position of the element to be inserted.

splice

Moves the elements from list *other* in the range [*first*, *last*) and inserts them at position *pos* of this list.

```
void splice(
    iterator pos,
    d_list< T, Allocator >& other,
    iterator first,
    iterator last);
```

Parameters *pos*

Position where the range of elements [*first*, *last*) from list *x* are inserted.

other

Reference to the list containing the range of elements to be moved.

first

The first element in the range of elements to be moved to *pos*.

last

Position following the final element to be included in the range of elements to be moved. The range does not include this position.

swap

Exchanges the contents of this list with those of list *other*.

```
void swap(d_list< T, Allocator >& other);
```

Parameters *other*

Reference to the list whose contents are exchanged with this list.

Nonmember Functions

operator==

global function

Returns true if list *x* contains the same elements in the same order as list *y*.

```
bool operator==(
    const d_list< T, Allocator >& x,
    const d_list< T, Allocator >& y);
```

Parameters `x`Constant reference to a list whose elements are compared with those of `y`.`y`Constant reference to a list. The elements of list `x` are compared with list `y`'s elements for equality.**operator<**

global function

Returns `true` if list `x` is lexicographically less than list `y`. Otherwise, returns `false`.

```
bool operator<(
    const d_list< T, Allocator >& x,
    const d_list< T, Allocator >& y);
```

Parameters `x`Constant reference to a list whose elements are compared with those of `y`.`y`Constant reference to a list. The elements of list `x` are compared lexicographically with those of list `y`.**swap**

global function

Exchanges the contents of list `x` with those of list `y`.

```
void swap(
    d_list< T, Allocator >& x,
    d_list< T, Allocator >& y);
```

Parameters `x`Reference to the list whose contents are copied to and replaced by those of list `y`.`y`Reference to the list whose contents are copied to and replaced by those of list `x`.

d_map Class

Inheritance: **d_map**

`d_map` is an associative container that maintains a collection of sorted (key, value) pairs ordered according to key and sorted using the specified comparator function. Map containers provide for fast retrieval of the data associated with the key in a pair. Each key must be unique; you cannot store duplicate keys in a map container.

`d_map` is a non-persistence-capable class. You can embed `d_map` in persistence-capable classes, and you can use it as a base class from which you can derive persistence-capable subclasses.

For information on `d_pc_map`, the Objectivity/C++ STL persistence-capable map class, see “`d_pc_map Class`” on page 97.

In addition to the functions described in this chapter, `d_map` provides functions that allow you to access pairs stored in a map, insert and erase elements by key or position, obtain information about the content of the map and its order, and so forth. For information on these and other functions that operate on maps, see the ObjectSpace Standards<Toolkit> documentation.

Class Declaration

class d_map

```
#include <d_map.h>

template < class Key, class Value, class Compare=less< Key >,
          class Allocator=d_allocator< OS_PAIR( Key, Value ) > >
class d_map
```

Parameters

Key

The type `Key` of the key element to be stored in the map container. This can be of any non-persistence-capable class, including Objectivity/C++ STL non-persistence-capable container classes. Because pairs are ordered internally by key, you cannot directly change the key element of a (key, value) pair in place. To do so would destroy the ordering and lead to database corruption. To change a key, you must delete the pair containing it and insert a new one.

Value

The type of the value to be stored in the map. This can be of any non-persistence-capable class, including Objectivity/C++ STL non-persistence-capable container classes. You can directly modify the value element of a (key, value) pair.

Compare

The comparator to be used in determining the sort order. This can be either system or user defined.

Allocator

The allocator to be used by `d_map`. The default allocator encapsulates the storage allocation and deallocation of map nodes with elements of type (key, value) pair. This allocator clusters the nodes of a `d_map` together with the persistent object that embeds the `d_map`.

Implementation

This Objectivity/C++ STL map container is implemented as a red-black tree of nodes, where each node is an instance of the persistence-capable `d_value_node` template class. Each node contains object references to its parent, left, and right nodes, plus a data member for storing a map element.

Each map element is comprised of a key and its associated data—that is, a (key, value) pair of the type `pair< const Key, Value >`. The ObjectSpace Standards<Toolkit> implementation provides the `OS_PAIR(Key, Value)` macro for use with compilers that do not support constant template parameters. When a compiler does not support constant template parameters, `OS_PAIR` expands to `pair< Key, Value >`.

Because `d_value_node` is persistence-capable, the key and value element types embedded in it cannot be persistence-capable (persistence-capable classes are not valid embedded-class types).

Invoking `operator[]` on a mutable `d_map` container could cause Objectivity/C++ STL to allocate a `d_value_node` of `(Key, Value())` even if `MyMap[key]` appears in an expression such as `MyMap[100].size()`. This is consistent with the behavior of `operator[]` in ObjectSpace Standards<Toolkit>.

Application Notes

Because `d_map` is implemented with the persistence-capable template class `d_value_node`, the DDL file must provide an explicit instantiation directive for `d_value_node` with each element type to be stored.

EXAMPLE To use a `d_map` that stores keys of type `int` and associated data of class `Person`, you must add the following explicit instantiation directive to the DDL file:

```
// DDL file
template class d_value_node < OS_PAIR( int, Person ) >;
```

If your application uses `less< Key >` or `greater< Key>` function objects for the comparator and there are no built-in `less-than (operator<)` and `greater-than (operator>)` operators for the stored key type, you must define your own operators.

Iterators

`d_map` provides constant and mutable bidirectional iterators.

You cannot embed the map iterators in persistence-capable classes.

EXAMPLE This example shows how to update the value of a (key, value) pair using the mutable iterator. The example shows how to change the data value of the first (key, value) pair in the map to 100. Public data member `first` of the class `pair` contains the key, and public data member `second` contains the associated value.

```
// DDL file
#include <d_map.h>

typedef d_map < int, int, less<int>,
              d_allocator < OS_PAIR (int, int) > > int_map;
```

```
class aMap : public ooObj {
    int_map _map;
    ...
};

template class d_value_node < OS_PAIR(int, int) >;

```

```
// C++ source file

ooHandle (aMap) a;
... // Open existing object a
int_map::iterator i;
// Get the first (key, value) pair in the map
i = a->_map.begin();
// Get the value from the pair
(*i).second = 100;
```

iterator

A mutable bidirectional iterator that traverses objects in a map container in the order in which they are sorted.

```
d_map< Key, Value, Compare, Allocator >::iterator
```

const_iterator

A constant bidirectional iterator that traverses objects in a map container in the order in which they are sorted.

```
d_map< Key, Value, Compare, Allocator >::const_iterator
```

Reference Summary

`d_map` is the non-persistence-capable Objectivity/C++ STL implementation of the ObjectSpace Standards<Toolkit> `map`. Objectivity/C++ STL `d_map` has an identical interface to ObjectSpace Standards<Toolkit> `map` with the exception of the functions shown in the following table.

Constructors and destructor	<code>d_map</code> <code>~d_map</code>
Replacing a map's contents	<code>operator=</code>
Exchanging the contents of two maps	<code>swap</code>
Comparing the contents of two maps	<code>operator==</code> <code>operator<</code>

Constructors and Destructor

d_map

Constructor for `d_map`. The `d_map` constructor is overloaded to allow you to construct maps initialized in different ways.

- `d_map();`
- `explicit d_map(
 const Compare& comp,
 const Allocator& alloc=Allocator());`
- `d_map(
 const OS_PAIR(Key, Value)* first,
 const OS_PAIR(Key, Value)* last,
 const Compare& comp=Compare(),
 const Allocator& alloc=Allocator());`
- `d_map(
 const_iterator first,
 const_iterator last,
 const Compare& comp=Compare(),
 const Allocator& alloc=Allocator());`

```

5.  template <class InputIterator>
    d_map (
        InputIterator first,
        InputIterator last,
        const Compare& comp=Compare(),
        const Allocator& alloc=Allocator());
6.  d_map(
    const d_map< Key, Value, Compare,
    locator >& other);

```

Parameters

comp

Constant reference to the comparator object used to order the elements of the map, based on the key item of the pair.

alloc

Constant reference to the allocator to be used to manage the storage for the map.

first

If the compiler does not support template parameters for member functions, *first* can be either a constant pointer or a constant iterator for the map that refers to the first (key, value) pair in a range of pairs to be copied to the map to initialize it. The map is constructed to contain a copy of the pairs comprising the range [*first*, *last*), which determine its size.

If the compiler supports template parameters for member functions, *first* can be an iterator type defined as a template parameter.

last

If the compiler does not support template parameters for member functions, *last* can be either a constant pointer or a constant iterator that refers to the position after the final pair comprising the range [*first*, *last*). The range to be copied does not include *last*. The new map is constructed to contain a copy of the pairs comprising the range [*first*, *last*), which determine its size.

If the compiler supports template parameters for member functions, *last* can be an iterator type defined as a template parameter.

other

Constant reference to a map whose elements are copied and used to initialize the map container.

~d_map

Destructor for d_map.

```
~d_map;
```

Discussion The map destructor destroys the map and removes its pairs, deallocating the storage used for it.

Member Functions

operator=

Replaces the contents of this map with a copy of the contents of the map *other*.

```
d_map< Key, Value, Compare, Allocator >& operator=(  
    const d_map < Key, Value, Compare, Allocator >& other);
```

Parameters *other*

Constant reference to the map whose contents are copied and used to replace those of this map.

swap

Exchanges the contents of this map with the contents of the map *other*.

```
void swap(d_map < Key, Value, Compare, Allocator >& other);
```

Parameters *other*

Reference to the map whose contents are exchanged with this map.

Nonmember Functions

operator==

global function

Returns true if map *x* contains the same elements in the same order as map *y*.

```
bool operator==(  
    const d_map < Key, Value, Compare, Allocator >& x,  
    const d_map < Key, Value, Compare, Allocator >& y);
```

Parameters x
 Constant reference to the map whose elements are compared with those of y .

y
 Constant reference to a map. The elements of map x are compared with map y 's elements for equality.

operator< global function

Returns true if map x is lexicographically less than map y . Otherwise, returns false.

```
bool operator<(
    const d_map < Key, Value, Compare, Allocator >& x,
    const d_map < Key, Value, Compare, Allocator >& y);
```

Parameters x
 Constant reference to a map whose elements are compared with those of y .

y
 Constant reference to a map. The elements of map x are compared lexicographically with map y 's elements.

swap global function

Exchanges the contents of map x with those of map y .

```
void swap(
    d_map < Key, Value, Compare, Allocator >& x,
    d_map < Key, Value, Compare, Allocator >& y);
```

Parameters x
 Reference to the map whose contents are copied to and replaced by those of map y .

y
 Reference to the map whose contents are copied to and replaced by those of map x .

d_multimap Class

Inheritance: `d_multimap`

`d_multimap` is an associative container that maintains a collection of sorted (key, value) pairs ordered according to key, and sorted using the specified comparator function. A multimap container provides for fast retrieval of the data associated with a key in a pair. A multimap container allows for storage of duplicate keys.

`d_multimap` is a non-persistence-capable class. You can embed the `d_multimap` class in persistence-capable classes, and you can use it as a base class from which you can derive persistence-capable subclasses.

For information on `d_pc_multimap`, the Objectivity/C++ STL persistence-capable map class, see “`d_pc_multimap Class`” on page 101.

In addition to the functions described in this chapter, `d_multimap` provides functions that allow you to access pairs stored in a multimap, insert and erase elements by key or position, obtain information about the content of the multimap and its order, and so forth. For information on these and other functions that operate on multimaps, see the ObjectSpace Standards<Toolkit> documentation.

Class Declaration

class `d_multimap`

```
#include <d_map.h>

template < class Key, class Value, class Compare=less< Key >,
          class Allocator=d_allocator< OS_PAIR( Key, Value ) > >
class d_multimap
```

Parameters

Key

The type `Key` of the key element to be stored in the multimap container. This can be of any non-persistence-capable class, including Objectivity/C++ STL non-persistence-capable container classes. Because pairs are ordered internally by key, you cannot directly change the key element of a (key, value) pair in place. To do so would destroy the ordering and lead to database corruption. To change a key, you must delete the pair containing it and insert a new one.

Value

The type of the value to be stored in the multimap. This can be of any non-persistence-capable class, including Objectivity/C++ STL non-persistence-capable container classes. You can directly modify the value element of a (key, value) pair.

Compare

The comparator to be used in determining the sort order. This can be either system or user defined.

Allocator

The allocator to be used by `d_multimap`. The default allocator encapsulates the storage allocation and deallocation of objects of map nodes with elements of type (key, value) pair. This allocator clusters the nodes of a `d_multimap` together with the persistent object that embeds the `d_multimap`.

Implementation

This Objectivity/C++ STL map container is implemented as a red-black tree of nodes, where each node is an instance of the persistence-capable `d_value_node` template class. Each node contains object references to its parent, left, and right nodes, plus a data member for storing a map element.

Each map element is comprised of a key and its associated data—that is, a (key, value) pair of the type `pair< const Key, Value >`. The ObjectSpace Standards<Toolkit> implementation provides the `OS_PAIR(Key, Value)` macro for use with compilers that do not support constant template parameters. When a compiler does not support constant template parameters, `OS_PAIR` expands to `pair< Key, Value >`.

Because `d_value_node` is persistence-capable, the key and value element types embedded in it cannot be persistence-capable (persistence-capable classes are not valid embedded-class types).

Application Notes

Because `d_multimap` is implemented with the persistence-capable template class `d_value_node`, the DDL file must provide an explicit instantiation directive for `d_value_node` with each element type to be stored.

EXAMPLE To use a `d_multimap` that stores a collection of keys of type `int` and associated values of type `Person`, you must add the following statement to the DDL file:

```
// DDL file
template class d_value_node < OS_PAIR(int, Person) >;
```

If your application uses `less< Key >` or `greater< Key>` function objects for the comparator and there are no built-in `less-than` (`operator<`) and `greater-than` (`operator>`) operators for the stored key type, you must define your own operators.

Iterators

`d_multimap` provides constant and mutable iterator types.

You cannot embed the `multimap` iterators in persistence-capable classes.

EXAMPLE This example shows how to update the value of a (key, value) pair using the mutable iterator. The example shows how to change the data value of the first (key, value) pair in the `multimap` to "Henry". Public data member `first` of the class `pair` contains the key, and public data member `second` contains the associated value.

```
// DDL file
#include <d_map.h>

typedef
    d_multimap < int, ooVString, less<int>,
                d_allocator < OS_PAIR (int,ooVString) > > int_multimap;

class aMap : public ooObj {
    int_multimap _multimap;
    ...
}
template class d_value_node < OS_PAIR (int, ooVString) >;
```

```
// C++ source file

ooHandle (aMap) a;
... // Open existing object a
int_multimap::iterator i;
// Get the first (key, value) pair in the map
i = a->_multimap.begin();
// Get the value from the pair
(*i).second = "Henry";
```

iterator

A mutable bidirectional iterator that traverses objects in a multimap container in the order in which they are sorted.

```
d_multimap< Key, Value, Compare, Allocator >::iterator
```

const_iterator

A constant bidirectional iterator that traverses objects in a multimap container in the order in which they are sorted.

```
d_multimap< Key, Value, Compare, Allocator >::const_iterator
```

Reference Summary

`d_multimap` is the non-persistence-capable Objectivity/C++ STL implementation of the ObjectSpace Standards<Toolkit> `multimap`. Objectivity/C++ STL `d_multimap` has an identical interface to ObjectSpace Standards<Toolkit> `multimap` with the exception of the functions shown in the following table.

Constructors and destructor	<code>d_multimap</code> <code>~d_multimap</code>
Replacing a multimap's contents	<code>operator=</code>
Exchanging the contents of two multimaps	<code>swap</code>
Comparing the contents of two multimaps	<code>operator==</code> <code>operator<</code>

Constructors and Destructor

`d_multimap`

Constructor for `d_multimap`. The `d_multimap` constructor is overloaded to allow you to construct multimaps initialized in different ways.

- `d_multimap();`
- `explicit d_multimap(
 const Compare& comp,
 const Allocator& alloc=Allocator());`
- `d_multimap(
 const OS_PAIR(Key, Value)* first,
 const OS_PAIR(Key, Value)* last,
 const Compare& comp=Compare(),
 const Allocator& alloc=Allocator());`
- `d_multimap(
 const_iterator first,
 const_iterator last,
 const Compare& comp=Compare(),
 const Allocator& alloc=Allocator());`

```

5.  template <class InputIterator>
    d_multimap (
        InputIterator first,
        InputIterator last,
        const Compare& comp=Compare(),
        const Allocator& alloc=Allocator());
6.  d_multimap(
    const d_multimap< Key, Value, Compare,
    Allocator >& other );

```

Parameters

comp

Constant reference to the comparator object used to order the elements of the multimap based on the key item of the pair.

alloc

Constant reference to the allocator to be used to manage the storage for the multimap.

first

If the compiler does not support template parameters for member functions, *first* can be either a constant pointer or a constant iterator for a multimap that refers to the first (key, value) pair in a range of pairs to be copied to the multimap to initialize it. The multimap is constructed to contain a copy of the pairs comprising the range [*first*, *last*), which determine its size.

If the compiler supports template parameters for member functions, *first* can be an iterator type defined as a template parameter.

last

If the compiler does not support template parameters for member functions, *last* can be either a constant pointer or a constant iterator for a multimap that refers to the position after the final (key, value) pair comprising the range [*first*, *last*). The range of pairs to be copied does not include *last*. The multimap is constructed to contain a copy of the pairs comprising the range [*first*, *last*), which determine its size.

If the compiler supports template parameters for member functions, *last* can be an iterator type defined as a template parameter.

other

Constant reference to a multimap whose elements are copied and used to initialize the constructed multimap container.

~d_multimap

Destructor for d_multimap.

```
~d_multimap;
```

Discussion The multimap destructor destroys the multimap and removes its contents, deallocating the storage used for it.

Member Functions

operator=

Replaces the contents of this multimap with a copy of the contents of multimap *other*.

```
d_multimap< Key, Value, Compare, Allocator >& operator=(  
    const d_multimap < Key, Value, Compare, Allocator >&  
    other);
```

Parameters *other*

A constant reference to the multimap whose contents are copied and used to replace those of this multimap.

swap

Exchanges the contents of this multimap with the contents of multimap *other*.

```
void swap(  
    d_multimap < Key, Value, Compare, Allocator >& other);
```

Parameters *other*

Reference to the multimap whose contents are exchanged with this multimap.

Nonmember Functions

operator==

global function

Returns `true` if multimap `x` contains the same elements in the same order as multimap `y`.

```
bool operator==(
    const d_multimap < Key, Value, Compare, Allocator >& x,
    const d_multimap < Key, Value, Compare, Allocator >& y);
```

Parameters

`x`

Constant reference to the multimap whose elements are compared with those of `y`.

`y`

Constant reference to a multimap. The elements of multimap `x` are compared with multimap `y`'s elements for equality.

operator<

global function

Returns `true` if multimap `x` is lexicographically less than multimap `y`. Otherwise, returns `false`.

```
bool operator<(
    const d_multimap < Key, Value, Compare, Allocator >& x,
    const d_multimap < Key, Value, Compare, Allocator >& y );
```

Parameters

`x`

Constant reference to a multimap whose elements are compared with those of `y`.

`y`

Constant reference to a multimap. The elements of multimap `x` are compared lexicographically with multimap `y`'s elements.

swap

global function

Exchanges the contents of multimap `x` with those of multimap `y`.

```
void swap(
    d_multimap < Key, Value, Compare, Allocator >& x,
    d_multimap < Key, Value, Compare, Allocator >& y);
```

Parameters x

Reference to the multimap whose contents are copied to and replaced by those of multimap y .

y

Reference to the multimap whose contents are copied to and replaced by those of multimap x .

d_multiset Class

Inheritance: `d_multiset`

`d_multiset` is an associative container that allows for fast lookups of its sorted collection of keys. `d_multiset` supports storage of multiple copies of the same key.

`d_multiset` is a non-persistence-capable class. You can embed the class `d_multiset` in persistence-capable classes, and you can use it as a base class from which you can derive persistence-capable subclasses.

For information on `d_pc_multiset`, the Objectivity/C++ STL persistence-capable multiset class, see “`d_pc_multiset Class`” on page 105.

In addition to the functions described in this chapter, `d_multiset` provides functions that allow you to clear a multiset, obtain information about its contents, insert and delete keys, obtain an iterator positioned at the first or last element of the multiset, and obtain an iterator to an object that matches a key. For information on these and other functions that operate on multisets, see the ObjectSpace Standards<Toolkit> documentation.

Class Declaration

class `d_multiset`

```
#include <d_set.h>

template < class Key, class Compare=less< Key >,
          class Allocator=d_allocator< Key > >
class d_multiset
```

Parameters

Key

The type `Key` of the key element to be stored in the multiset container. This can be of any non-persistence-capable class, including Objectivity/C++ STL non-persistence-capable container classes. You cannot modify a key in place in a multiset container. To do so would destroy the ordering of keys and lead to database corruption. To change a key, you must delete it and insert a new one. A key stored in a multiset container is used as the sort data, but it is also the retrievable element.

Compare

The comparator to be used in determining the sort order. This can be either system or user defined.

Allocator

The allocator to be used by `d_multiset`. The default allocator encapsulates the storage allocation and deallocation of multiset nodes with elements of type `Key`. This allocator clusters the nodes of the `d_multiset` together with the persistent object that embeds the `d_multiset`.

Implementation

This Objectivity/C++ STL set container is implemented as a red-black tree of nodes, where each node is an instance of the persistence-capable `d_value_node` template class. Each node contains object references to its parent, left, and right nodes, plus a data member for storing a key element.

Because `d_value_node` is a persistence-capable class, the element type `Key` cannot be persistence-capable (persistence-capable classes are not valid embedded-class types). However, the element type can be an object-reference type for a persistence-capable class—for example `ooRef(myItem)`, where `myItem` is persistence-capable.

Application Notes

Because `d_multiset` is implemented with the persistence-capable template class `d_value_node`, the DDL file must provide an explicit instantiation directive for `d_value_node` with each element type to be stored.

EXAMPLE To use a `d_multiset` that stores keys of type `int`, you must add the following explicit instantiation directive to the DDL file:

```
// DDL file
template class d_value_node<int>
```

If your application uses `less< Key >` or `greater< Key >` function objects for the comparator and there are no built-in `less-than` (`operator<`) and `greater-than` (`operator>`) operators for the stored key type, you must define your own operator functions.

For example, to store keys of type `pair<double, int>`, you would define an operator such as the following to be used by `less< Key >`:

```
bool operator< ( const pair<double, int>& x,
               const pair<double, int>& y )
{
    return x.first < y.first;
}
```

Iterators

`d_multiset` provides two iterator types. Both iterators are of the constant bidirectional category, because the only allowed method of changing a key is to first delete it and then insert a replacement key.

You cannot embed the multiset iterators in persistence-capable classes.

iterator

A constant bidirectional iterator that traverses the objects in a multiset container in the order in which they are sorted.

```
d_multiset< Key, Compare, Allocator >::iterator
```

const_iterator

A constant bidirectional iterator that traverses the objects in a multiset container in the order in which they are sorted.

```
d_multiset< Key, Compare, Allocator >::const_iterator
```

Reference Summary

`d_multiset` is the non-persistence-capable Objectivity/C++ STL implementation of the ObjectSpace Standards<Toolkit> `multiset`. Objectivity/C++ STL `d_multiset` has an identical interface to ObjectSpace Standards<Toolkit> `multiset` with the exception of the functions shown in the following table.

Constructors and destructor	<code>d_multiset</code> <code>~d_multiset</code>
Replacing a multiset's contents	<code>operator=</code>
Exchanging the contents of two multisets	<code>swap</code>
Comparing the contents of two multisets	<code>operator==</code> <code>operator<</code>

Constructors and Destructor

d_multiset

Constructor for `d_multiset`. The `d_multiset` constructor is overloaded to allow you to construct multisets initialized in different ways.

1. `d_multiset();`
2. `explicit d_multiset(
 const Compare& comp,
 const Allocator& alloc=Allocator());`
3. `d_multiset(
 const Key* first,
 const Key& last,
 Compare& comp=Compare(),
 const Allocator& alloc=Allocator());`
4. `d_multiset(
 const_iterator first,
 const_iterator last,
 const Compare& comp=Compare(),
 const Allocator& alloc=Allocator());`

```

5.  template < class InputIterator >
    d_multiset (
        InputIterator first,
        InputIterator last,
        const Compare& comp=Compare(),
        const Allocator& alloc=Allocator());

6.  d_multiset(
        const d_multiset< Key, Compare,
        Allocator >& other);

```

Parameters

comp

Constant reference to the comparator object used to order the elements of the multiset.

alloc

Constant reference to the allocator to be used to manage the storage for the multiset.

first

If the compiler does not support template parameters for member functions, *first* can be either a constant pointer or a constant bidirectional iterator that refers to the first key in a range of keys to be copied to the multiset to initialize it. The multiset is constructed to contain a copy of the keys comprising the range [*first*, *last*), which determine its size.

If the compiler supports template parameters for member functions, *first* can be an iterator type defined as a template parameter. This iterator can belong to another container and can be used to insert elements copied from that container to this one.

last

If the compiler does not support template parameters for member functions, *last* can be either a constant pointer or a constant bidirectional iterator that refers to the position after the final key comprising the range [*first*, *last*). The range to be copied does not include *last*. The new key is constructed to contain a copy of the keys comprising the range [*first*, *last*), which determine its size. If the compiler supports template parameters for member functions, *last* can be an iterator type defined as a template parameter. This iterator can belong to another container and can be used to insert elements copied from that container to this one.

other

Constant reference to a multiset whose elements are copied and used to initialize the constructed multiset container.

Discussion The default constructor creates a multiset container that orders its elements using `Compare ()`.

~d_multiset

Destructor for `d_multiset`.

```
~d_multiset;
```

Discussion The multiset destructor destroys the multiset and erases all of its elements, deallocating the storage used for it.

Member Functions

operator=

Replaces the contents of this multiset with a copy of the contents of multiset *other*.

```
d_multiset< Key, Compare, Allocator >& operator=(
    const d_multiset < Key, Compare, Allocator >& other);
```

Parameters *other*

Constant reference to the multiset whose contents are copied and used to replace the contents of this multiset.

swap

Exchanges the contents of this multiset with the contents of multiset *other*.

```
void swap(d_multiset < Key, Compare, Allocator >& other);
```

Parameters *other*

Reference to the multiset whose contents are exchanged with this multiset.

Nonmember Functions

operator==

Returns `true` if multiset `x` contains the same elements in the same order as multiset `y`.

```
bool operator==(
    const d_multiset < Key, Compare, Allocator >& x,
    const d_multiset < Key, compare, Allocator >& y);
```

Parameters `x`

Constant reference to a multiset whose elements are compared with those of `y`.

`y`

Constant reference to a multiset. The elements of multiset `x` are compared with those of multiset `y` for equality.

operator< Returns `true` if multiset `x` is lexicographically less than multiset `y`. Otherwise, returns `false`.

```
bool operator<(
    const d_multiset < Key, Compare, Allocator >& x,
    const d_multiset < Key, Compare, Allocator >& y);
```

Parameters `x`

Constant reference to a multiset whose elements are compared with those of `y`.

`y`

Constant reference to a multiset. The elements of multiset `x` are compared lexicographically with those of multiset `y`.

swap

Exchanges the contents of the multiset `x` with those of multiset `y`.

```
void swap(
    d_multiset < Key, Compare, Allocator >& x,
    d_multiset < Key, Compare, Allocator >& y);
```

Parameters x

Reference to the multiset whose contents are copied to and replaced by those of multiset y .

y

Reference to the multiset whose contents are copied to and replaced by those of multiset x .

d_pc_list Class

Inheritance: `d_list`, `ooObj`

`d_pc_list` is a sequential container that enables fast insertions and deletions at any position in the sequence, but does not allow for index-based random access. List containers offer the best performance of sequential containers for cases in which you frequently insert and delete interior elements if you do not need to move elements randomly from one position in the sequence to another. List sequences can be rearranged by relinking the list node objects, allowing for constant-time insertions and deletions. This is not true of sequential containers, such as vectors, that support constant time operations at one or both ends only.

`d_pc_list` is a persistence-capable class. It is subclassed from `d_list` and `ooObj`. You can store an instance of `d_pc_list` directly in a federated database, but you cannot embed `d_pc_list` in other persistence-capable classes.

For information on the iterators, constructors, destructors, member functions, and nonmember functions for this class, see “`d_list` Class” on page 59. For information on functions for inserting, deleting, and sorting list elements, splicing two lists, filtering lists, assigning and comparing list elements, and merging two lists, see the ObjectSpace Standards<Toolkit> documentation.

Class Declaration

class `d_pc_list`

```
#include <d_list.h>

template < class T, class Allocator=d_allocator< T > >
class d_pc_list : public ooObj,
    public d_list< T, Allocator > {...};
```

Parameters *T*

The type of element stored in the list container. List elements can be of any non-persistence-capable class, including Objectivity/C++ STL non-persistence-capable container classes.

Allocator

The allocator to be used by `d_pc_list`. The default allocator encapsulates the storage allocation and deallocation of list nodes with data elements of type `T`. This allocator clusters the list nodes of a `d_pc_list` together with the `d_pc_list` itself.

Implementation

`d_pc_list` inherits the `d_list` implementation characteristics described in “Implementation” on page 60.

Application Notes

Because `d_pc_list` is persistence-capable, you must explicitly instantiate it with its element type in a DDL file. Furthermore, because `d_pc_list` is implemented with the persistence-capable template class `d_list_node`, you must also instantiate `d_list_node` with the element type to be stored.

EXAMPLE To use a `d_pc_list` that stores integers, you must add the following instantiation directives to the DDL file:

```
// DDL file
template class d_list_node<int>;
template class d_pc_list< int, d_allocator<int> >;
```

d_pc_map Class

Inheritance: `d_map`, `ooObj`

`d_pc_map` is an associative container that maintains a collection of sorted (key, value) pairs ordered according to key, and sorted using the specified comparator function. Map containers provide for fast retrieval of the data associated with the key in a pair. Each key in a map must be unique; storage of duplicate keys is not allowed.

`d_pc_map` is a persistence-capable class. It is subclassed from `d_map` and `ooObj`. You can store an instance of `d_pc_map` directly in a federated database, but you cannot embed `d_pc_map` in other persistence-capable classes.

For information on the iterators, constructors, destructors, member functions, and nonmember functions for this class, see “`d_map` Class” on page 69. For information on functions that allow you to access pairs stored in a map, insert and erase elements by key or position, obtain information about the content of the map and its order, and so forth, see the ObjectSpace Standards<Toolkit> documentation.

Class Declaration

class `d_pc_map`

```
#include <d_map.h>

template < class Key, class Value, class Compare=less< Key >,
          class Allocator=d_allocator< OS_PAIR(Key, Value) > >
class d_pc_map : public ooObj,
  public d_map< Key, Value, Compare, Allocator >{...};
```

Parameters

Key

The type `Key` of the key element to be stored in the map container. This can be of any non-persistence-capable class, including Objectivity/C++ STL non-persistence-capable container classes. Because pairs are ordered internally by key, you cannot directly change the key element of a (key, value) pair in place. To do so would destroy the ordering and lead to database corruption. To change a key, you must delete the pair containing it and insert a new one.

Value

The type of the value in a (key, value) pair to be stored in the map. This can be of any non-persistence-capable class, including Objectivity/C++ STL non-persistence-capable container classes. You can directly modify the value element of the pair, but not its key.

Compare

The comparator to be used in determining the sort order. This can be either system or user defined.

Allocator

The allocator to be used by `d_pc_map`. The default allocator encapsulates the storage allocation and deallocation of map nodes with elements of type (key, value) pair. This allocator clusters the nodes of a `d_pc_map` together with the `d_pc_map` itself.

Implementation

`d_pc_map` inherits the `d_map` implementation characteristics described in “Implementation” on page 70.

Application Notes

Because `d_pc_map` is persistence-capable, you must explicitly instantiate it with its element type in a DDL file. Furthermore, because `d_pc_map` is implemented with the persistence-capable template class `d_value_node`, you must also instantiate `d_value_node` with the element type to be stored.

EXAMPLE To use a `d_pc_map` that stores keys of type `int` and associated data of the non-persistence-capable class `Person`, you must add the following instantiation directives to the DDL file:

```
// DDL file
template class d_value_node< OS_PAIR(int, Person) > ;
template class d_pc_map< int, Person, less<int>,
    d_allocator< OS_PAIR(int, Person) > >;
```

If your application uses `less< Key >` or `greater< Key>` function objects for the comparator and there are no built-in `less-than` (`operator<`) and `greater-than` (`operator>`) operators for the stored key type, you must define your own operators.

d_pc_multimap Class

Inheritance: `d_multimap`, `ooObj`

`d_pc_multimap` is an associative container that maintains a collection of sorted (key, value) pairs ordered according to key and sorted using the specified comparator function. A multimap container provides for fast retrieval of the data associated with a key in a pair. `d_pc_multimap` allows for storage of duplicate keys.

`d_pc_multimap` is a persistence-capable class. It is subclassed from `d_multimap` and `ooObj`. You can store an instance of `d_pc_multimap` directly in a federated database, but you cannot embed `d_pc_multimap` in other persistence-capable classes.

For information on the iterators, constructors, destructors, member functions, and nonmember functions for this class, see “`d_multimap Class`” on page 77. For information on functions that allow you to access pairs stored in a multimap, insert and erase elements by key or position, obtain information about the content of the multimap and its order, and so forth, see the ObjectSpace Standards<Toolkit> documentation.

Class Declaration

class `d_pc_multimap`

```
#include <d_map.h>

template < class Key, class Value, class Compare=less< Key >,
          class Allocator=d_allocator< OS_PAIR(Key, Value) > >
class d_pc_multimap : public ooObj,
    public d_multimap< Key, Value, Compare, Allocator > {...};
```

Parameters

Key

The type `Key` of the key element to be stored in the multimap container. This can be of any non-persistence-capable class, including Objectivity/C++ STL non-persistence-capable container classes. Because pairs are ordered internally by key, you cannot directly change the key element of a (key, value) pair in place. To do so would destroy the ordering and lead to database corruption. To change a key, you must delete the pair containing it and insert a new one.

Value

The type of the value to be stored in the multimap. This can be of any non-persistence-capable class, including Objectivity/C++ STL non-persistence-capable container classes. You can directly modify the value element of a (key, value) pair.

Compare

The comparator to be used in determining the sort order. This can be either system or user defined.

Allocator

The allocator class to be used by `d_pc_multimap`. The default allocator encapsulates the storage allocation and deallocation of map nodes with elements of type (key, value) pair. This allocator clusters the nodes of a `d_pc_multimap` together with the `d_pc_multimap` itself.

Implementation

`d_pc_multimap` inherits the `d_multimap` implementation characteristics described in “Implementation” on page 78.

Application Notes

Because `d_pc_multimap` is persistence-capable, you must explicitly instantiate it with its element type in a DDL file. Furthermore, because `d_pc_multimap` is implemented with the persistence-capable template class `d_value_node`, you must also instantiate `d_value_node` with the element type to be stored.

EXAMPLE To use a `d_pc_multimap` that stores keys of class `int` and associated values of class `Person`, you must add the following instantiation directives to the DDL file:

```
template class d_value_node< OS_PAIR(int, Person) >;  
template class d_pc_multimap< int, Person, less<int>,  
    d_allocator< OS_PAIR(int, Person) > >;
```

If your application uses `less< Key >` or `greater< Key >` function objects for the comparator and there are no built-in `less-than` (`operator<`) and `greater-than` (`operator>`) operators for the key type to be stored, you must define your own operators.

d_pc_multiset Class

Inheritance: `d_multiset`, `ooObj`

`d_pc_multiset` is an associative container that allows for fast lookups of its sorted collection of keys. It is subclassed from the `d_multiset` class and the `ooObj` class. A `d_pc_multiset` class can store multiple copies of the same key.

`d_pc_multiset` is a persistence-capable class. It is subclassed from `d_multiset` and `ooObj`. You can store an instance of `d_pc_multiset` directly in a federated database, but you cannot embed `d_pc_multiset` in other persistence-capable classes.

For information on the iterators, constructors, destructors, member functions, and nonmember functions for this class, see “`d_multiset` Class” on page 87. For information on functions that allow you to clear a multiset, obtain information about its contents, insert and delete keys, obtain an iterator positioned at the first or last element of the multiset, obtain an iterator to an object that matches a key, and so forth, see the ObjectSpace Standards<Toolkit> documentation.

Class Declaration

class `d_pc_multiset`

```
#include <d_set.h>

template < class Key, class Compare=less< Key >,
           class Allocator=d_allocator< Key > >
class d_pc_multiset : public ooObj,
    public d_multiset< Key, Compare, Allocator > {...};
```

Parameters

Key

The type `Key` of the key element to be stored in the multiset container. This can be of any non-persistence-capable class, including Objectivity/C++ STL non-persistence-capable container classes. You cannot modify a key in place in a multiset container. To do so would destroy the ordering of keys and lead to database corruption. To change a key, you must delete it and insert a new one. A key stored in a multiset container is used as the sort data, but it is also the retrievable element.

Compare

The comparator to be used in determining the sort order. This can be either system or user defined.

Allocator

The allocator to be used by `d_pc_multiset`. The default allocator encapsulates the storage allocation and deallocation of multiset nodes of elements of type `Key`. This allocator clusters the nodes of a `d_pc_multiset` together with the `d_pc_multiset` itself.

Implementation

`d_pc_multiset` inherits the `d_multiset` implementation characteristics described in “Implementation” on page 88.

Application Notes

Because `d_pc_multiset` is persistence-capable, you must explicitly instantiate it with its element type in a DDL file. Furthermore, because `d_pc_multiset` is implemented with the persistence-capable template class `d_value_node`, you must also instantiate `d_value_node` with the element type to be stored.

EXAMPLE

To use a `d_pc_multiset` that stores keys of type `float`, you must add the following statements to the DDL file:

```
template class d_value_node<float>;
template class d_pc_multiset<float, less<float>,
                        d_allocator<float> >;
```

If your application uses `less< Key >` or `greater< Key >` function objects for the comparator and there are no built-in less-than (`operator<`) and greater-than (`operator>`) operators for the stored key type, you must define your own operators.

For example, to store keys of type `pair<double, int>`, you would define an operator such as the following to be used by `less< Key >`:

```
bool operator< ( const pair<double, int>& x,
                const pair<double, int>& y )
{
    return x.first < y.first;
}
```


d_pc_priority_queue Class

Inheritance: `priority_queue`, `ooObj`

`d_pc_priority_queue` is an adaptive container that uses a comparator you specify to sort the elements stored in the container. You can only use the `d_vector` implementation as the priority queue's internal data structure.

`d_pc_priority_queue` is a persistence-capable class. It is subclassed from `priority_queue` and `ooObj`. You can store an instance of `d_pc_priority_queue` directly in a federated database, but you cannot embed `d_pc_priority_queue` in other persistence-capable classes.

For information on `priority_queue`, see “`priority_queue Class`” on page 135. For information on functions for pushing values onto a priority queue, popping values off a priority queue, testing whether a priority queue is empty, and getting the size of a priority queue, see the ObjectSpace Standards<Toolkit> documentation.

Class Declaration

class `d_pc_priority_queue`

```
#include <d_queue.h>

template < class T, class Container=d_vector< T >,
          class Compare=less<Container::T> >
class d_pc_priority_queue : public ooObj,
    public priority_queue < T, Container, Compare > {...};
```

Parameters *T*

The type of element to be stored in `d_pc_priority_queue`.

Container

The internal data structure used by `d_pc_priority_queue`, which, for Objectivity/C++ STL, is always `d_vector`.

Compare

The comparator used to order the sorted items stored in `d_pc_priority_queue`.

Application Notes

Because `d_pc_priority_queue` uses `d_vector` as its implementation class, you must include the `d_vector.h` header file in your source code file.

Because `d_pc_priority_queue` is persistence-capable, you must explicitly instantiate it with its element type and implementation class in a DDL file.

EXAMPLE The following directive explicitly instantiates `d_pc_priority_queue` with `float` elements and the `d_vector` implementation class:

```
// DDL file
#include <d_queue.h>
#include <d_vector.h>
template class d_pc_priority_queue< float, d_vector<float,
    d_allocator<float>>, less<float> >;
```

`d_pc_priority_queue` has no associated iterators, because adaptive containers do not support iteration.

d_pc_queue Class

Inheritance: `queue`, `ooObj`

`d_pc_queue` is an adaptive container that provides first-in, first-out data access. You can only use the `d_list` implementation as the queue's internal data structure.

`d_pc_queue` is a persistence-capable class. It is subclassed from `queue` and `ooObj`. You can store an instance of `d_pc_queue` directly in a federated database, but you cannot embed `d_pc_queue` in other persistence-capable classes.

For information on `queue`, see “queue Class” on page 137. For information on functions for pushing values onto a queue, popping values off a queue, testing whether a queue is empty, and getting the size of a queue, see the ObjectSpace Standards<Toolkit> documentation.

Class Declaration

class `d_pc_queue`

```
#include <d_queue.h>

template < class T, class Container=d_list< T > >
class d_pc_queue : public ooObj,
    public queue < T, Container > {...};
```

Parameters `T`

The type of element stored in `d_pc_queue`.

Container

The internal data structure used by `d_pc_queue`, which, for Objectivity/C++ STL, is always `d_list`.

Application Notes

Because `d_pc_queue` uses `d_list` as its implementation class, you must include the `d_list.h` header file in your source code file.

Because `d_pc_queue` is persistence-capable, you must explicitly instantiate it with its element type and implementation class in a DDL file. Furthermore, because the implementation class `d_list` is itself implemented with the persistence-capable template class `d_list_node`, you must also instantiate `d_list_node` with the element type to be stored.

EXAMPLE To use a `d_pc_queue` that stores elements of type `int`, you must add the following statements to the DDL file:

```
// DDL file
#include <d_queue.h>
#include <d_list.h>
template class d_list_node<int>;
template class d_pc_queue
    < int, d_list< int, d_allocator<int> > >;
```

`d_pc_queue` has no associated iterators because adaptive containers do not support iteration.

d_pc_set Class

Inheritance: `d_set`, `ooObj`

`d_pc_set` is an associative container that allows for fast lookups of its sorted collection of keys, none of which may be duplicates.

`d_pc_set` is a persistence-capable class. It is subclassed from `d_set` and `ooObj`. You can store an instance of `d_pc_set` directly in a federated database, but you cannot embed `d_pc_set` in other persistence-capable classes.

For information on the iterators, constructors, destructors, member functions, and nonmember functions for this class, see “`d_set Class`” on page 121. For information on functions that allow you to clear a set, obtain information about its contents, obtain an iterator positioned at the first or last element of the set, obtain an iterator to an object that matches a key, and so forth, see the `ObjectSpace Standards<Toolkit>` documentation.

Class Declaration

class `d_pc_set`

```
#include <d_set.h>

template < class Key, class Compare=less< Key >,
          class Allocator=d_allocator< Key > >
class d_pc_set : public ooObj,
  public d_set< Key, Compare, Allocator > {...};
```

Parameters	<p><i>Key</i></p> <p>The type <code>Key</code> of the key element to be stored in the set container. This can be of any non-persistence-capable class, including Objectivity/C++ STL non-persistence-capable container classes. You cannot modify a key in place in a set container. To do so would destroy the ordering of keys and lead to database corruption. To change a key, you must delete it and insert a new one. The key is used as the sort data, but it is also the retrievable element.</p> <p><i>Compare</i></p> <p>The comparator to be used in determining the sort order. This can be either system or user defined.</p> <p><i>Allocator</i></p> <p>The allocator to be used by <code>d_pc_set</code>. The default allocator encapsulates the storage allocation and deallocation of set nodes of elements of type <code>Key</code>. This allocator clusters the nodes of a <code>d_pc_set</code> together with the <code>d_pc_set</code> itself.</p>
------------	---

Implementation

`d_pc_set` inherits the `d_set` implementation characteristics described in “Implementation” on page 122.

Application Notes

Because `d_pc_set` is persistence-capable, you must explicitly instantiate it with its element type in a DDL file. Furthermore, because `d_pc_set` is implemented with the persistence-capable template class `d_value_node`, you must also instantiate `d_value_node` with the element type to be stored.

EXAMPLE To use a `d_pc_set` that stores keys of type `int`, you must add the following instantiation directives to the DDL file:

```
template class d_value_node<int>;
template class d_pc_set< int, less<int>, d_allocator<int> >;
```

If your application uses `less< Key >` or `greater< Key >` function objects for the comparator and there are no built-in less-than (`operator<`) and greater-than (`operator>`) operators for the stored key type, you must define your own operators.

For example, to store keys of type `pair<double, int>`, you would define an operator such as the following to be used by `less< Key >`:

```
bool operator< ( const pair<double, int>& x,
               const pair<double, int>& y )
{
    return x.first < y.first;
}
```


d_pc_stack Class

Inheritance: **stack**, **ooObj**

`d_pc_stack` is an adaptive container that provides first-in, last-out data access. A `stack` can use either the `d_vector` or `d_list` implementation as its internal data structure.

`d_pc_stack` is a persistence-capable class. It is subclassed from `stack` and `ooObj`. You can store an instance of `d_pc_stack` directly in a federated database, but you cannot embed `d_pc_stack` in other persistence-capable classes.

For information on `stack`, see “`stack Class`” on page 139. For information on functions for pushing values onto a stack, popping values off a stack, testing whether a stack is empty, and getting the size of a stack, see the ObjectSpace Standards<Toolkit> documentation.

Class Declaration

class `d_pc_stack`

```
#include <d_stack.h>

template < class T, class Container=deque< T > >
class d_pc_stack : public ooObj,
    public stack < T, Container > {...};
```

Parameters *T*

The type of element to be stored in the `d_vector` or `d_list` container used to implement `d_pc_stack`.

Container

The internal data structure used by `d_pc_stack`, which, for Objectivity/C++ STL, is always either `d_vector` or `d_list`.

Application Notes

Depending on the implementation you choose (vector or list), you must include either the `d_vector.h` header file or the `d_list.h` header file in your source code.

Because `d_pc_stack` is persistence-capable, you must explicitly instantiate it with its element type and implementation class in a DDL file. Furthermore, if you use the `d_list` implementation class, you must also instantiate `d_list_node` with the element type to be stored.

EXAMPLE The following directive explicitly instantiates `d_pc_stack` with `int` elements and the `d_vector` implementation class:

```
// DDL file
#include <d_stack.h>
#include <d_vector.h>

template class d_pc_stack
    < int, d_vector< int, d_allocator< int > > >;
```

The following directives instantiate `d_pc_stack` with `int` elements and the `d_list` implementation class:

```
// DDL file
#include <d_stack.h>
#include <d_list.h>

template class d_list_node<int>;
template class d_pc_stack
    < int, d_list< int, d_allocator< int > > >;
```

`d_pc_stack` has no associated iterators because adaptive containers do not support iteration.

d_pc_vector Class

Inheritance: `d_vector`, `ooObj`

`d_pc_vector` is a sequential container class that provides fast index-based random access to its contents. A vector container, which is the simplest kind of container, can store a collection of objects of the same type whose lengths can vary. (A `d_pc_vector` is similar to a standard C array except that it is extensible.)

`d_pc_vector` is a persistence-capable class. It is subclassed from `d_vector` and `ooObj`. You can store an instance of `d_pc_vector` directly in a federated database, but you cannot embed `d_pc_vector` in other persistence-capable classes.

For information on `d_vector`, see “`d_vector Class`” on page 129. For information on member and nonmember functions that operate on vectors, see the ObjectSpace Standards<Toolkit> documentation.

Class Declaration

class `d_pc_vector`

```
#include <d_vector.h>

template < class T, class Allocator=d_allocator< T > >
class d_pc_vector : public ooObj,
    public d_vector < T, Allocator > {...}
```

Parameters *T*

The type of element stored in the vector container. Vector elements can be of any non-persistence-capable class, including Objectivity/C++ STL non-persistence-capable container classes.

Allocator

The allocator to be used by `d_pc_vector`. To manage storage for a `d_pc_vector` container, Objectivity/C++ STL uses an internal allocator class.

Implementation

`d_pc_vector` is implemented internally as an Objectivity/DB variable-size array (VArray). Consequently, elements of `d_pc_vector` are subject to the same restrictions as those of a VArray (see the Objectivity/C++ Data Definition Language book). For example, a `d_pc_vector` cannot contain objects of a persistence-capable class, although object references are allowed. Furthermore, a `d_pc_vector` cannot directly or indirectly contain VArrays, so a `d_pc_vector` cannot have elements of type `d_vector` or `d_string`.

Application Notes

Because `d_pc_vector` is persistence-capable, you must explicitly instantiate it with its element type in a DDL file.

EXAMPLE The following directive explicitly instantiates `d_pc_vector` with `int` elements:

```
// DDL file
template class d_pc_vector< int, d_allocator<int> >;
```

d_set Class

Inheritance: **d_set**

`d_set` is an associative container that allows for fast lookups of its sorted collection of keys. `d_set` does not support storage of duplicate keys.

`d_set` is a non-persistence-capable class. You can embed class `d_set` in persistence-capable classes, and you can use it as a base class from which you can derive persistence-capable subclasses.

For information on `d_pc_set`, the Objectivity/C++ STL persistence-capable set class, see “`d_pc_set Class`” on page 113.

In addition to the functions described in this chapter, `d_set` provides functions that allow you to clear a set, obtain information about its contents, obtain an iterator positioned at the first or last element of the set, and obtain an iterator to an object that matches a key. For information on these and other functions that operate on sets, see the ObjectSpace Standards<Toolkit> documentation.

Class Declaration

class `d_set`

```
#include <d_set.h>

template < class Key, class Compare=less< Key >,
           class Allocator=d_allocator< Key > >
class d_set
```

Parameters

Key

The type `Key` of the key element to be stored in the set container. This can be of any non-persistence-capable class, including Objectivity/C++ STL non-persistence-capable container classes. You cannot modify a key in place in a set container. To do so would destroy the ordering of keys and lead to database corruption. To change a key, you must delete it and insert a new one. The key is used as the sort data, but it is also the retrievable element.

Compare

The comparator to be used in determining the sort order. This can be either system or user defined.

Allocator

The allocator to be used by `d_set`. The default allocator encapsulates the storage allocation and deallocation of set nodes with elements of type `Key`. This allocator clusters the nodes of the `d_set` together with the persistent object that embeds the `d_set`.

Implementation

This Objectivity/C++ STL set container is implemented as a red-black tree of nodes, where each node is an instance of the persistence-capable `d_value_node` template class. Each node contains object references to its parent, left, and right nodes, plus a data member for storing a key element.

Because `d_value_node` is a persistence-capable class, the element type `Key` cannot be persistence-capable (persistence-capable classes are not valid embedded-class types). However, the element type can be an object-reference type for a persistence-capable class—for example `ooRef(myItem)`, where `myItem` is persistence-capable.

Application Notes

Because `d_set` is implemented with the persistence-capable template class `d_value_node`, the DDL file must provide an explicit instantiation directive for `d_value_node` with each key type to be stored.

EXAMPLE To use a `d_set` that stores keys of type `int`, you must add the following explicit instantiation directive to the DDL file:

```
// DDL file
template class d_value_node<int>
```

If your application uses `less< Key >` or `greater< Key >` function objects for the comparator and there are no built-in `less-than` (`operator<`) and `greater-than` (`operator>`) operators for the stored key type, you must define your own operators.

For example, to store keys of type `pair<double, int>`, you would define an operator such as the following to be used by `less< Key >`:

```
bool operator< ( const pair<double, int>& x,
                const pair<double, int>& y )
{
    return x.first < y.first;
}
```

Iterators

`d_set` provides two iterator types. Both iterators are of the constant bidirectional category because the only allowed method of changing a key is to first delete it and then insert a replacement key. A constant iterator prohibits changing the value to which the iterator refers.

You cannot embed the set iterators in instances of persistence-capable classes.

iterator

A constant bidirectional iterator that traverses objects in a set container in the order in which they are sorted.

```
d_set< Key, Compare, Allocator >::iterator
```

const_iterator

A constant bidirectional iterator that traverses objects in a set container in the order in which they are sorted.

```
d_set< Key, Compare, Allocator >::const_iterator
```

Reference Summary

`d_set` is the non-persistence-capable Objectivity/C++ STL implementation of the ObjectSpace Standards<Toolkit> `set`. Objectivity/C++ STL `d_set` has an identical interface to ObjectSpace Standards<Toolkit> `set` with the exception of the functions shown in the following table.

Constructors and destructor	<code>d_set</code> <code>~d_set</code>
Replacing a set's contents	<code>operator=</code>
Exchanging the contents of two sets	<code>swap</code>
Comparing the contents of two sets	<code>operator==</code> <code>operator<</code>

Constructors and Destructor

`d_set`

Constructor for `d_set`. The `d_set` constructor is overloaded to allow you to construct sets initialized in different ways.

1. `d_set();`
2. `explicit d_set(
 const Compare& comp,
 const Allocator& alloc=Allocator());`
3. `d_set(
 const Key* first,
 const Key& last,
 const Compare& comp=Compare(),
 const Allocator& alloc=Allocator());`
4. `d_set(
 const_iterator first,
 const_iterator last,
 const Compare& comp=Compare(),
 const Allocator& alloc=Allocator());`

```

5.  template < class InputIterator >
    d_set (
        InputIterator first,
        InputIterator last,
        const Compare& comp=Compare(),
        const Allocator& alloc=Allocator());
6.  d_set(
        const d_set< Key, Compare,
        Allocator >& other);

```

Parameters

comp

Constant reference to the comparator object used to order the elements of the set.

alloc

Constant reference to the allocator to be used to manage the storage for the set.

first

If the compiler does not support template parameters for member functions, *first* can be either a constant pointer or a constant bidirectional iterator that refers to the first key in a range of keys to be copied to the set to initialize it. The set is constructed to contain a copy of the keys comprising the range [*first*, *last*), which determine its size.

If the compiler supports template parameters for member functions, *first* can be an iterator type defined as a template parameter. This iterator can belong to another container and can be used to insert elements copied from that container to this one.

last

If the compiler does not support template parameters for member functions, *last* can be either a constant pointer or a constant bidirectional iterator that refers to the position after the final key comprising the range [*first*, *last*). The range to be copied does not include *last*. The new key is constructed to contain a copy of the keys comprising the range [*first*, *last*), which determine its size. If the compiler supports template parameters for member functions, *last* can be an iterator type defined as a template parameter. This iterator can belong to another container and can be used to insert elements copied from that container to this one.

other

Constant reference to a set whose elements are copied and used to initialize the constructed set container.

Discussion The default constructor creates a set container that orders its elements using `Compare()`. Constructing a set from a range eliminates copies of duplicate keys, should any exist.

~d_set

Destructor for `d_set`.

```
~d_set;
```

Discussion The set destructor removes this `d_set` object and its contents, deallocating the storage used for it.

Member Functions

operator=

Replaces the contents of this set with a copy of the contents of set *other*.

```
d_set< Key, Compare, Allocator >& operator=(
    const d_set < Key, Compare, Allocator >& other);
```

Parameters *other*

A constant reference to the set whose contents are copied and used to replace this set's contents.

swap

Exchanges the contents of this set with the contents of set *other*.

```
void swap(d_set < Key, Compare, Allocator > & other);
```

Parameters *other*

Reference to the set whose contents are exchanged with this set.

Nonmember Functions

operator==

Returns `true` if set `x` contains the same elements in the same order as set `y`.

```
bool operator==(
    const d_set < Key, Compare, Allocator >& x,
    const d_set < Key, Compare, Allocator >& y);
```

Parameters `x`

Constant reference to a set whose elements are compared with those of `y`.

`y`

Constant reference to a set. The elements of set `x` are compared with those of set `y` for equality.

operator<

Returns `true` if set `x` is lexicographically less than set `y`. Otherwise, returns `false`.

```
bool operator<(
    const d_set < Key, Compare, Allocator >& x,
    const d_set < Key, Allocator >& y);
```

Parameters `x`

Constant reference to a set whose elements are compared with those of `y`.

`y`

Constant reference to a set. The elements of set `x` are compared lexicographically with those of set `y`.

swap

nonmember function

Exchanges the contents of the set `x` with those of set `y`.

```
void swap(
    d_set < Key, Compare, Allocator >& x,
    d_set < Key, Compare, Allocator >& y);
```

Parameters x

Reference to the set whose contents are copied to and replaced by those of set y .

y

Reference to the set whose contents are copied to and replaced by those of set x .

d_vector Class

Inheritance: `d_vector`

`d_vector` is a sequential container class that provides fast index-based random access to its contents. A vector container, which is the simplest kind of container, can store a collection of objects of the same type whose lengths can vary. A `d_vector` is similar to a standard C array except that it is extensible.

`d_vector` is a non-persistence-capable class. You can embed `d_vector` in persistence-capable classes, and you can use it as a base class from which you can derive persistence-capable subclasses.

For information on `d_pc_vector`, the Objectivity/C++ STL persistence-capable vector class, see “`d_pc_vector Class`” on page 119.

In addition to the member and nonmember functions described in this chapter, `d_vector` provides functions that allow you to access vector elements, expand or contract a vector by inserting or deleting elements, or replace elements. You can insert or delete copies of elements or values at specific positions within the vector. You can also determine the number of entries the vector currently contains and the maximum number of entries it is capable of holding. For information on these and other functions that operate on vectors, see the ObjectSpace Standards<Toolkit> documentation.

Class Declaration

class `d_vector`

```
#include <d_vector.h>

template < class T, class Allocator=d_allocator< T > >
class d_vector
```

Parameters *T*

The type of element stored in the vector container. Vector elements can be of any non-persistence-capable class, including Objectivity/C++ STL non-persistence-capable container classes.

Allocator

The allocator class to be used by `d_vector`. To manage storage for a `d_vector` container, Objectivity/C++ STL uses an internal allocator class.

Implementation

`d_vector` is implemented as an Objectivity/DB variable-size array (VArray). Consequently, elements of `d_vector` are subject to the same restrictions as those of a VArray (see the Objectivity/C++ Data Definition Language book). For example, a `d_vector` cannot contain objects of a persistence-capable class, although object references are allowed. Furthermore, a `d_vector` cannot directly or indirectly contain VArrays, so a `d_vector` cannot have elements of type `d_vector` or `d_string`.

Iterators

`d_vector` provides constant and mutable types for iterators.

When your application accesses a data element belonging to a vector by calling `operator*` on a mutable iterator, Objectivity/C++ STL pins the variable-size array to which the iterator refers and marks the entire array for update. It unpins the array when the object that encloses it is closed.

You cannot embed the vector iterators in persistence-capable classes.

iterator

A mutable random access iterator that traverses the objects in a vector container.

```
d_vector< T, Allocator >::iterator
```

const_iterator

A constant random access iterator that traverses the objects in a vector container.

```
d_vector< T, Allocator >::const_iterator
```

Reference Summary

`d_vector` is the non-persistence-capable Objectivity/C++ STL implementation of the ObjectSpace Standards<Toolkit> vector. Objectivity/C++ STL `d_vector` has an identical interface to ObjectSpace Standards<Toolkit> vector with the exception of the functions shown in the following table.

Constructors and destructor	<code>d_vector</code> <code>~d_vector</code>
Replacing the vector contents	<code>operator=</code>
Exchanging the contents of one vector with another	<code>swap</code>

Constructors and Destructor

`d_vector`

Constructor for `d_vector`. The `d_vector` constructor is overloaded to allow you to construct vectors that are initialized in different ways.

1. `d_vector();`
2. `explicit d_vector(
 const Allocator& alloc);`
3. `explicit d_vector(
 size_type n,
 const T& value=T(),
 const Allocator& alloc=Allocator());`
4. `d_vector(
 const_iterator first,
 const_iterator last,
 const Allocator& alloc=Allocator());`
5. `template < class InputIterator >
d_vector (
 InputIterator first,
 InputIterator last,
 const Allocator& alloc=Allocator());`
6. `d_vector(
 const d_vector< T, Allocator >& other);`

Parameters	<p><i>alloc</i></p> <p>Constant reference to the allocator to be used to manage storage for the vector.</p> <p><i>n</i></p> <p>Number of vector elements containing copies of the value <i>value</i>. The vector is constructed to contain <i>n</i> copies of <i>value</i>.</p> <p><i>value</i></p> <p>Constant reference to the value of type <i>T</i> used to initialize the elements of the vector. The vector is constructed to contain <i>n</i> copies of this value.</p> <p><i>first</i></p> <p>If the compiler does not support template parameters for member functions, <i>first</i> can be either a constant pointer or a constant iterator for a vector that refers to the first element in a range of elements to be copied to that vector to initialize it. The new vector is constructed to contain a copy of the elements comprising the range [<i>first</i>, <i>last</i>], which determines its size.</p> <p>If the compiler supports template parameters for member functions, <i>first</i> can be an iterator type defined as a template parameter. This iterator can belong to another container and be used for the vector insert member function.</p> <p><i>last</i></p> <p>If the compiler does not support template parameters for member functions, <i>last</i> can be either a constant pointer or a constant vector iterator that refers to the position after the last element comprising the range [<i>first</i>, <i>last</i>]. The range does not include <i>last</i>. The new vector is constructed to contain a copy of the elements comprising the range [<i>first</i>, <i>last</i>], which determine its size.</p> <p>If the compiler supports template parameters for member functions, <i>last</i> can be an iterator type defined as a template parameter. This iterator can belong to another container and be used for the vector insert member function.</p> <p><i>other</i></p> <p>Constant reference to a vector whose elements are copied and used as the contents of the newly constructed vector.</p>
------------	---

~d_vector

Destructor for d_vector.

```
~d_vector;
```

Discussion	The vector destructor destroys the vector and erases all of its elements.
------------	---

Member Functions

operator=

Replaces the contents of this vector with a copy of vector *other*.

```
d_vector< T Allocator >& operator=(  
    const d_vector< T, Allocator >& other);
```

Parameters *other*

A constant reference to the vector to be copied to this vector.

Discussion If necessary, the target vector is resized during the assignment operation to accommodate the replacement elements.

swap

Exchanges the contents of this vector with those of vector *other*.

```
void swap(d_vector< T, Allocator >& other);
```

Parameters *other*

The vector to be copied to and replaced by this vector.

Discussion The contents of this vector replace those of vector *other*, and vice versa.

Nonmember Functions

swap

Exchanges the contents of one vector with those of another.

```
void swap(  
    d_vector< T, Allocator >& x,  
    d_vector< T, Allocator >& y);
```


priority_queue Class

Inheritance: **priority_queue**

`priority_queue` is an adaptive container that uses a comparator you specify to sort the elements stored in the container.

`priority_queue` is provided by `ObjectSpace Standards<Toolkit>`. To store a priority queue in an Objectivity/DB database, you use a `d_vector` as the priority queue's internal data structure.

You can embed a `priority_queue` that uses the `d_vector` implementation in a persistence-capable class. You can also use `priority_queue` as a base class for persistence-capable subclasses.

For information on functions for pushing values onto a priority queue, popping values off a priority queue, testing whether a priority queue is empty, and getting the size of a priority queue, see the `ObjectSpace Standards<Toolkit>` documentation.

Class Declaration

class priority_queue

```
#include <queue>

template < class T, class Container=vector< T >,
           class Compare=less< Container::T > >
class priority_queue
```

Parameters *T*

The type of element to be stored in the priority queue.

Container

The internal data structure used by `priority_queue`, which, for Objectivity/C++ STL, is always `d_vector`.

Compare

The comparator used to order the sorted items stored in the priority queue.

Application Notes

To use `priority_queue` with the `d_vector` implementation class, you must include the `d_vector.h` header file in your DDL file.

EXAMPLE This DDL file defines a persistence-capable class that embeds a `priority_queue` with elements of type `int`:

```
// DDL file
#include <queue>
#include <d_vector.h>

class i_adapt: public ooObj {
    priority_queue < int, d_vector< int, d_allocator<int> >,
                    less <int> > MyPQ;
    ...
}
```

A priority queue has no associated iterators because adaptive containers do not support iteration.

queue Class

Inheritance: **queue**

`queue` is an adaptive container that provides first-in, first-out data access.

`queue` is provided by the ObjectSpace Standards<Toolkit>. To store a `queue` in an Objectivity/DB database, you use a `d_list` as the `queue`'s internal data structure.

You can embed a `queue` that uses the `d_list` implementation in a persistence-capable class. You can also use the class `queue` as a base class from which you can derive persistence-capable subclasses.

For information on functions for pushing values onto a `queue`, popping values off a `queue`, testing whether a `queue` is empty, and getting the size of a `queue`, see the ObjectSpace Standards<Toolkit> documentation.

Class Declaration

class queue

```
#include <queue>

template < class T, class Container=deque< T > >
class queue
```

Parameters *T*

The type of element stored in the `queue`.

Container

The internal data structure used by `queue`, which, for Objectivity/C++ STL, is always `d_list`.

Application Notes

To use `queue` with the `d_list` implementation class, you must include the `d_list.h` header file in your DDL file.

Because the implementation class `d_list` is itself implemented with the persistence-capable template class `d_list_node`, the DDL file must explicitly instantiate `d_list_node` with the element type to be stored.

EXAMPLE This DDL file defines a persistence-capable class that embeds a `queue` with elements of type `int`. Because the `queue` uses the `d_list` implementation class, the DDL file explicitly instantiates `d_list_node` with elements of type `int`.

```
// DDL file
#include <queue>
#include <d_list.h>

template class d_list_node<int>;
typedef queue< int, d_list< int, d_allocator<int> > > q_lst_int;

class i_adapt: public ooObj {
    q_lst_int q_l;
    ...
};
```

A `queue` has no associated iterators because adaptive containers do not support iteration.

stack Class

Inheritance: **stack**

`stack` is an adaptive container that provides first-in, last-out data access.

`stack` is provided by the ObjectSpace Standards<Toolkit>. To store a stack in an Objectivity/DB database, you can use `d_vector` or `d_list` as the stack's internal data structure.

You can embed a `stack` that uses the `d_vector` or `d_list` implementation in a persistence-capable class. You can also use `stack` as a base class for persistence-capable subclasses.

For information on functions for pushing values onto a stack, popping values off a stack, testing whether a stack is empty, and getting the size of a stack, see the ObjectSpace Standards<Toolkit> documentation.

Class Declaration

class stack

```
#include <stack>

template < class T, class Container=deque< T > >
class stack
```

Parameters *T*

The type of element to be stored in the stack.

Container

The internal data structure used by `stack`, which, for Objectivity/C++ STL, is always either `d_vector` or `d_list`.

Application Notes

Depending on the implementation you choose (vector or list), you must include either the `d_vector.h` header file or the `d_list.h` header file in your source code.

EXAMPLE This DDL file defines a persistence-capable class that embeds a `stack` with elements of type `int` and the `d_vector` implementation class:

```
// DDL file
#include <stack>
#include <d_vector.h>

typedef stack< int, d_vector< int, d_allocator<int> > >
    stk_vec_int;

class i_adapt: public ooObj {
    stk_vec_int s_v;
    ...
};
```

This DDL file defines a persistence-capable class that embeds a `stack` with elements of type `int`. Because the `stack` uses the `d_list` implementation class, the DDL file explicitly instantiates `d_list_node` with elements of type `int`.

```
// DDL file
#include <stack>
#include <d_list.h>

template class d_list_node<int>;
typedef stack< int, d_list< int, d_allocator<int> > >
    stk_lst_int;

class i_adapt: public ooObj {
    stk_lst_int s_l;
    ...
};
```

A `stack` has no associated iterators because adaptive containers do not support iteration.

Index

A

adaptive containers

- d_pc_priority_queue class 109
- d_pc_queue class 111
- d_pc_stack class 117
- examples 43
- priority_queue class 135
- queue class 137
- stack class 139

allocator

- class 20
- default 21
- parameter 20

assignment operator member functions

- d_list class 64
- d_map class 75
- d_multimap class 83
- d_multiset class 92
- d_set class 126
- d_vector class 133

associative containers

- d_map class 69
- d_multimap class 77
- d_multiset class 87
- d_pc_map class 97
- d_pc_multimap class 101
- d_pc_multiset class 105
- d_pc_set class 113
- d_set class 121
- examples 39

C

C++ string, persistent 55

classes

- d_basic_string 55
- d_list 60
- d_map 69
- d_multimap 77
- d_multiset 87
- d_pc_list 95
- d_pc_map 97
- d_pc_multimap 101
- d_pc_multiset 105
- d_pc_priority_queue 109
- d_pc_queue 111
- d_pc_set 113
- d_pc_stack 117
- d_pc_vector 119
- d_set 121
- d_string class 55
- d_vector 129
- priority_queue 135
- queue 137
- stack 139

clustering 16

clustering directive 21

constructors

- d_list class 62
- d_map class 73
- d_multimap class 81
- d_multiset class 90
- d_set class 124
- d_vector class 131

container 16

Objectivity/C++ STL 16

Objectivity/DB 16

STL 14

(see also STL container)

customer support 9**D****d_basic_string class** 55

implementation 56

iterators 56

constant random access 57

mutable random access 57

template declaration 55

d_list class 60

constructor 62

destructor 64

element data type

instantiating 60

example 34

implementation 34, 60

iterators 61

constant bidirectional iterator 61

mutable bidirectional iterator 61

member functions

assignment operator 64

merge, using compare 64

merge, using less-than operator 64

splice, inserting a range of elements 66

splice, inserting one list into another 65

splice, moving a list's element 65

swap 66

nonmember functions

equivalency operator 66

less-than operator 67

swap 67

template declaration 60

d_list_node class

for d_list implementation 34, 46, 60

for d_pc_list implementation 36

for d_pc_queue implementation 112

for d_pc_stack implementation 118

for queue implementation 138

d_map class 69

constructor 73

using template parameters 74

defining operators for key type 71

destructor 75

element data type

instantiating 71

Key 70, 78, 102

Value 70

example 41

implementation 70

iterators 71

constant bidirectional iterator 72

mutable bidirectional iterator 72

member functions

assignment operator 75

swap 75

nonmember functions

equivalency operator 75

less-than operator 76

swap 76

OS_PAIR(Key, Value) macro 70

template declaration 69

d_multi_map class

defining operators for key type 79

d_multimap class 77

constructor 81

using template parameters 82

destructor 83

element data type

instantiating 79

Key 78

specifying 78

Value 78

implementation 78

iterators 79

constant bidirectional iterator 80

mutable bidirectional iterator 80

member functions

assignment operator 83

swap 83

- nonmember functions
 - equivalency operator 84
 - less-than operator 84
 - swap 84
- template declaration 77
- d_multiset class** 87
 - constructor 90
 - destructor 92
 - element data type
 - instantiating 88
 - Key 88
 - specifying 88
 - implementation 88
 - iterators 89
 - constant bidirectional iterators 89
 - member functions
 - assignment operator 92
 - swap 92
 - nonmember functions
 - equivalency operator 93
 - less-than operator 93
 - swap 93
 - template declaration 87, 88
- d_pc_list class** 95
 - element data type
 - instantiating 96
 - example 36, 40
 - template declaration 95
 - template instantiation 96
- d_pc_map class** 97
 - element data type
 - instantiating 98
 - template declaration 97
 - template instantiation 98
- d_pc_multimap class** 101
 - element data type
 - instantiating 102
 - template declaration 101
 - template instantiation 102
- d_pc_multiset class** 105
 - element data type
 - instantiating 106
 - template declaration 105
 - template instantiation 106
- d_pc_priority_queue class** 109
 - template declaration 109
 - template instantiation 110
- d_pc_queue class** 111
 - element data type
 - instantiating 112
 - template declaration 111
 - template instantiation 112
- d_pc_set class** 113
 - element data type
 - instantiating 114
 - template declaration 113
 - template instantiation 114
- d_pc_stack class** 117
 - element data type
 - instantiating 118
 - example 45
 - template declaration 117
- d_pc_vector class** 119
 - implementation 120
 - template declaration 119
- d_queue class**
 - element data type
 - instantiating 138
- d_set class** 121
 - constructor 124
 - destructor 126
 - element data type
 - instantiating 122
 - specifying 121
 - implementation 122
 - iterators 123
 - constant bidirectional iterators 123
 - member functions
 - assignment operator 126
 - swap 126
 - nonmember functions
 - equivalency operator 127
 - less-than operator 127
 - swap 127
 - template declaration 121
- d_string class** 55

d_value_node class

- for d_map implementation 70
- for d_multimap implementation 78
- for d_multiset implementation 88
- for d_pc_multimap 102
- for d_pc_multiset 106
- for d_pc_set 114
- for d_set implementation 122

d_vector class 129

- constructor 131
- destructor 132
- element data type
 - specifying 130
- implementation 130
- iterators 130
 - constant random access iterator 130
 - mutable random access iterator 130
- member functions
 - assignment operator 133
 - swap 133
- nonmember functions
 - swap 133
- template declaration 129

DDL file

- d_list_node data type instantiation
 - for d_pc_list 96
 - for d_pc_queue 112
 - for d_pc_stack 118
- d_pc_list data type instantiation 96
- d_pc_map data type instantiation 98
- d_pc_multimap data type instantiation 102
- d_pc_multiset data type instantiation 106
- d_pc_set data type instantiation 114
- d_value_node data type instantiation
 - for d_multimap 79
 - for d_pc_multimap 102
 - for d_pc_multiset 106
- instantiation directive 24

destructors

- d_list class 64
- d_map class 75
- d_multimap class 83

d_multiset class 92

d_set class 126

d_vector class 132

E**equivalency operator nonmember functions**

- d_list class 66
- d_map class 75
- d_multimap class 84
- d_multiset class 93
- d_set class 127

error handling 26**error signal 26****G****generic algorithms 26****H****header files 24****I****iterator**

- Objectivity/C++ 18
- Objectivity/C++ STL 18
 - choosing between constant and mutable 20
 - constant
 - effect of operator* 19
 - mutable
 - and associative containers 19
 - effect of operator* 19, 20
 - summary 19
- STL 18
 - bidirectional 18
 - constant 18
 - forward 18
 - input 18
 - mutable 18
 - output 18
 - random access 18

iterators

- d_basic_string class 56
- d_list class 61
- d_map class 71
- d_multimap class 79
- d_multiset class 89
- d_set class 123
- d_vector class 130

L**less-than operator nonmember functions**

- d_list class 67
- d_map class 76
- d_multimap class 84
- d_multiset class 93
- d_set class 127

M**merge member functions**

- using compare
 - d_list class 64
- using less-than operator
 - d_list class 64

N**non-persistence-capable class 15**

- naming convention 15, 53

non-persistence-capable classes

- d_basic_string 55
- d_list 59
- d_map 69
- d_multimap 77
- d_multiset 87
- d_set 121
- d_vector 129

O**Objectivity/C++**

- application
 - converting to Objectivity/C++ STL 28
- iterator 18
- lock and delete propagation 25
- referential integrity 25

Objectivity/C++ STL 14

- class summary 54
- interoperability with ObjectSpace Standards<Toolkit> 27
- iterator 18
- iterator summary 19

Objectivity/C++STL

- iterator
 - mutable
 - using for assignment operation 20

ObjectSpace Standards<ToolKit> 14**ooSignal function 26****P****persistence-capable class 15**

- naming convention 15, 53

persistence-capable classes

- d_pc_list 95
- d_pc_map 97
- d_pc_multimap 101
- d_pc_multiset 105
- d_pc_priority_queue 109, 110
- d_pc_queue 111, 112
- d_pc_set 113
- d_pc_stack 117, 118
- d_pc_vector 119

priority_queue class 135

- data type implementation 136
- template declaration 135

Q

Q

queue class 137

- data type instantiation 46
- example 46
- template declaration 137

S

sequential containers

- d_list class 59
- d_pc_list class 95
- d_pc_vector class 119
- d_vector class 129
- examples 34

splice member functions

- inserting a range of elements
 - d_list class 66
- inserting one list into another
 - d_list class 65
- moving a list's element
 - d_list class 65

stack class 139

- template declaration 139

STL component 14

- adaptor 14
- container 14
- generic algorithms 14
- iterator 14

STL container

- adaptive 14
- associative 14
- sequential 14

storage allocation 20

- assuming the default allocator 21
- specifying a custom allocator 21
- specifying the default allocator 21

swap member functions

- d_list class 66
- d_map class 75
- d_multimap class 83
- d_multiset class 92
- d_set class 126
- d_vector class 133

swap nonmember functions

- d_list class 67
- d_map class 76
- d_multimap class 84
- d_multiset class 93
- d_set class 127
- d_vector class 133

T

template class instantiation 23, 24