Using C++ with Caché

Version 2018.1
2020-11-13

InterSystems Corporation 1 Memorial Drive Cambridge MA 02142 www.intersystems.com

Using C++ with Caché
Caché Version 2018.1 2020-11-13

For Support questions about any InterSystems products, contact:

InterSystems Worldwide Response Center (WRC)
Tel: +1-617-621-0700
Tel: +44 (0) 844 854 2917
Email: support@InterSystems.com

Table of Contents

List of Figures

List of Tables

About This Book

This book is a guide to the Caché C++ Language Binding.

This book contains the following sections:

There is also a detailed Table of Contents.

For general information, see *Using InterSystems Documentation*.

1

The Caché C++ Binding

The Caché C++ binding provides a simple, direct way to use Caché objects within a C++ application. You can create C++ applications that work with the Caché database in the following ways:

- *The Caché C++ binding*

  The Caché C++ binding lets C++ applications work with objects on a Caché server. The Caché Class Generator can create a C++ proxy class for any Caché class. Proxy classes contain standard C++ code that can be compiled and used within your C++ application, providing access to the properties and methods of the corresponding Caché class.

  The C++ binding offers complete support for object database persistence, including concurrency and transaction control. In addition, there is a sophisticated data caching scheme to minimize network traffic when the Caché server and C++ applications are located on separate machines.

- *Dynamic binding*

  Instead of using compiled C++ proxy classes, you can work with Caché classes dynamically, at runtime. This can be useful for writing applications or tools that deal with classes in general and do not depend on particular Caché classes.

- *Light C++ Binding*

  The Light C++ Binding (LCB) is a limited subset of the Caché C++ library intended primarily for loading simple data at very high speed. It combines your C++ application and the Caché Object Server into a single process, using intraprocess communications rather than TCP/IP to exchange data between them. For basic object manipulation (creating objects, opening objects by Id, updating, and deleting), it is ten to twenty times faster than the standard C++ binding.

- *The Caché ODBC driver*

  Caché includes a standard ODBC driver that offers high-performance relational access to Caché, including the ability to execute SQL queries against the database. The C++ binding provides special classes to encapsulate the complexity of ODBC. For maximum flexibility, applications can use ODBC and the Caché C++ Binding at the same time.

Each of these features is discussed in the following chapters.

This document assumes a prior understanding of C++ and the C++ standard library. Several C++ compilers are supported, but Caché does not include a C++ compiler or development environment.

1.1 C++ Binding Architecture

The Caché C++ Binding gives C++ applications a way to access and manipulate objects contained within a Caché server. These objects can be persistent objects stored within the Caché object database, or they can be transient objects that perform operations within a Caché server.

The Caché C++ Binding consists of the following components:

- *The Caché C++ Generator*

  The C++ Generator is a program that generates C++ proxy classes (source and header files) from classes defined in the Caché Class Dictionary.

- *The Caché C++ library*

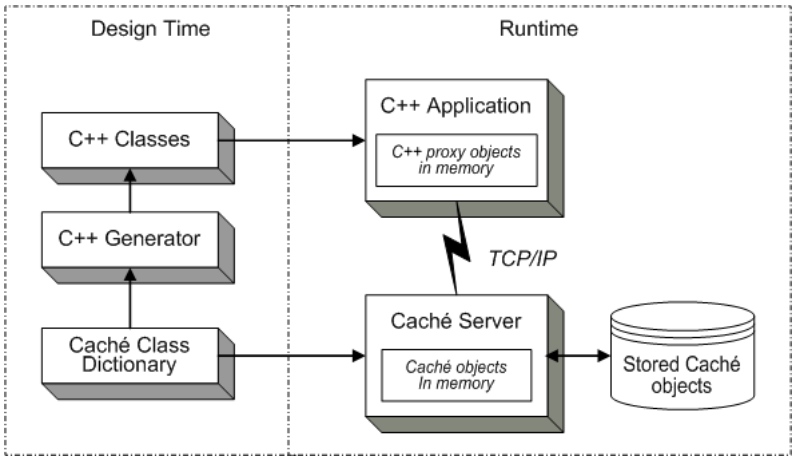
  The C++ library is a set of C++ classes used by the Caché C++ Generator to implement all the functionality of the C++ proxy classes. The library also includes a set of proxy classes for Caché server classes that require specialized adaptations to fit into the framework of the C++ standard library.

- *The Caché Object Server*

  The Caché Object Server is a high performance server process that manages communication between C++ clients and a Caché database server. It communicates using standard networking protocols (TCP/IP), and can run on any platform supported by Caché. The Caché Object Server is used by all Caché language bindings, including C++, Java, JDBC, ODBC, Perl, and Python.

The Caché C++ Generator can create C++ client classes for any classes contained within the Caché Class Dictionary. These generated C++ classes communicate at runtime (using TCP/IP sockets) with their corresponding Caché class on a Caché server. This is illustrated in the following diagram:

Figure 1–1: C++ Client/Server Class Architecture



The basic mechanism works as follows:

- You define one or more classes within Caché. These can be persistent objects stored within the Caché database or transient objects that run within a Caché server.
- The Caché C++ Class Generator creates C++ classes that correspond to your Caché classes. These classes include "stub" methods that invoke the corresponding Caché method on the server as well as accessor (get and set) methods for object properties.
- At runtime, your C++ application connects to a Caché server. It can then create C++ objects that correspond to Caché objects maintained by the Caché Object Server. You can use these objects as you would any other C++ objects.

- The system automatically manages all communications as well as client-side data caching. The actual deployment configuration is up to the application developer. The C++ client application and Caché server may reside on the same physical machine or they may be located on different machines. All communications between the C++ application and the Caché server use the standard TCP/IP protocol.

The runtime architecture consists of the following:

- A Caché database server (or servers). The Caché server is responsible for database operations as well as the execution of Caché object methods.
- A C++ "client" application into which your generated and compiled C++ proxy classes have been linked. (Although C++ is typically used for developing specialized tools and middle-ware, this document refers to such code as a client to differentiate it from the server code).
- A connection between the application and the server provided by the connection classes included with the Caché C++ library.

Note: The architecture of the Light C++ binding is quite different. It trades flexibility for speed by running all client and server operations on the same machine, using intraprocess communications instead of TCP/IP to exchange data between the C++ application and the Caché Object Server.

1.1.1 The Caché C++ Library

The Caché C++ binding's dynamic library of C++ classes implements the basic connection and caching mechanisms required to communicate with a Caché server.

The C++ components required to connect to Caché are contained within the C++ library file, which is located in the <cachesys>\dev\cpp\lib directory (see Default Caché Installation Directory in the *Caché Installation Guide* for the location of <cachesys> on your system). This directory contains different versions of the library that correspond to different build configurations for different platforms. A corresponding set of include files is located in the <cachesys>\dev\cpp\include subdirectory.

This library includes C++ versions of a number of the classes within the Caché class library, including %Persistent, %RegisteredObject, %SerialObject, the various Caché collection classes, and C++ versions of the various data type classes. In addition, the library contains the various classes used within a C++ application to manage communication with the Caché server.

The classes that are available for use in your C++ binding applications are listed and discussed in the following chapters:

- Reference for Simple Datatype Classes — describes literal datatypes containing simple data such as strings or numbers.
- Reference for Object Datatype Classes — describes the predefined proxy classes that correspond to standard Caché object datatype classes such as lists, arrays, and streams.
- Reference for Connectivity and Inherited Proxy Classes — lists the classes that provide functions necessary to generate proxy classes and connect them to the server.
- Reference for Utility Classes — lists some special classes for transactions, batch processing, and SQL queries.

1.2 Installation and Configuration

The Caché C++ binding software is not part of the standard Caché installation, but is offered as a option in the custom installation. For a list of the platforms that support the C++ Binding, see "Supported Client Platforms" in the online *InterSystems Supported Platforms* document for this release.

Caché C++ binding applications require a C++ compiler that supports the C++ standard library. When you compile, your path should include the following directories (see Default Caché Installation Directory in the *Caché Installation Guide* for the location of <cachesys> on your system):

```
<cachesys>\bin
<cachesys>\dev\cpp\lib
```

A compiled C++ binding application will be able to access existing Caché classes with no additional setup, and can run on client machines that do not have Caché installed.

If Caché is installed with level 3 ("locked down") security, `%Service_Bindings` must be enabled in order to run the Caché C++ Generator.

The Light C++ Binding has additional requirements (see Installing the Light C++ Binding).

1.2.1 Building the Caché C++ Binding from Source

In some special situations, it may be useful to build the Caché C++ Binding from source code. The source code is installed if you select the "C++ SDK" option during a custom install of Caché. Windows MSVC projects or UNIX® makefiles can be customized as desired to use different versions of compilers or standard libraries. On UNIX®, gmake is required. As shipped, the sources, projects, and makefiles are identical to those used to build the production version of the C++ binding, but they can be used, without modification, to rebuild with different gcc versions on Linux. This may be necessary, since C++ code built with different gcc versions is often not binary-compatible.

1.2.2 Configuring Microsoft Visual Studio 2008

The following instructions describe the procedure for configuring a Caché C++ binding project in Microsoft Visual Studio 2008 under Windows XP. Some details may be different in other environments.

Setting Windows Environment Variables

Before you configure a project, you must set some Windows environment variables. The procedure is as follows:

- In the Windows **Start** menu, select **Settings > Control Panel > System**. The **System Properties** dialog box opens.
- In the **System Properties** dialog box, select the **Advanced** tab and then click the **Environment Variables** button. The **Environment Variables** dialog box opens.
- In the **System Variables** section of the **Environment Variables** dialog box, add the following variables (see Default Caché Installation Directory in the *Caché Installation Guide* for the location of <cachesys> on your system):

variable name	variable value
CACHEBIN	<cachesys>\bin
CACHECPPLIB	<cachesys>\dev\cpp\lib

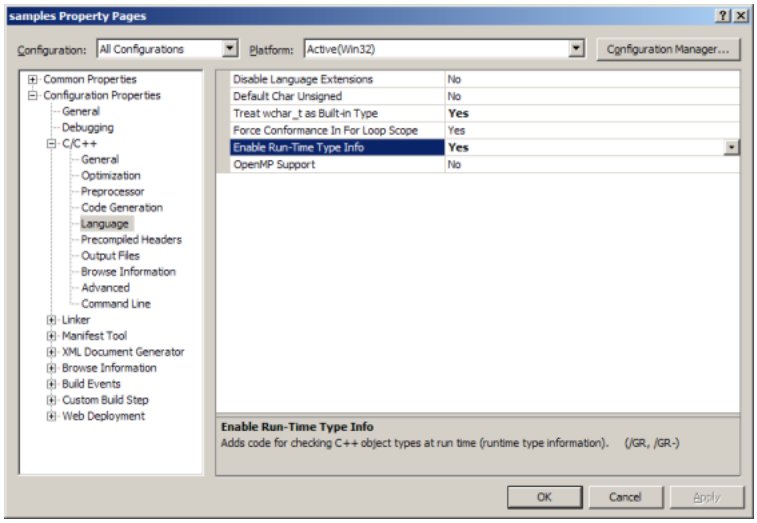
- Append the following to the PATH system variable:

```
;%CACHEBIN%;%CACHECPPLIB%
```

Configuring the Project

Open the Visual Studio 2008 Project References window:

- Open the project to be configured in Visual C++. In the following instructions, it is assumed that you have opened the Samples project located in <cachesys>\Dev\cpp\samples\msvc90\.
- On the main menu, select **Project > samples Properties**. The **samples Property Pages** dialog box opens.



- Click on the topmost item in the tree displayed on the left side of the dialog box. If this is not done, some of the tabs on the right side may be hidden.
- In the **samples Project Pages** dialog box, make the changes described in the following procedures.

Enable wchar_t and Run-Time Type Information (RTTI) support:

- In the **Configuration** drop-down box on the top left, select **All Configurations**.
- In the menu tree on the left, select **Configuration Properties > C/C++ > Language**.
- Make sure that the **Treat wchar_t as Built-in Type** has a value of `Yes`.
- Make sure that the **Enable Run-Time Type Info** has a value of `Yes`.
- Click the **Apply** button.

Specify the location of the C++ Binding header files:

- In the menu tree on the left, select **Configuration Properties > C/C++ > General**.
- In the **Additional Include Directories** text field, add:

  ```
  "<cachesys>\dev\cpp\include\"
  ```

  where `<cachesys>` refers to your Caché installation directory. Use a semicolon to separate it from any previous entries.
- Click the **Apply** button.

Specify the location of the C++ Binding library directory:

- In the **Configuration** drop-down box on the top left, select **All Configurations**.
- In the menu tree on the left, select **Configuration Properties > Linker > General**.
- In the **Additional Library Directories** text field, add:

  ```
  "$(CACHECPPLIB)\"
  ```

  Use a semicolon to separate it from any previous entries.
- Click the **Apply** button.

Specify the location of the C++ Binding release library:

- In the **Configuration** drop-down box on the top left, select **Release**.
- In the menu tree on the left, select **Configuration Properties > Linker > Input**.
- In the **Additional Dependencies** text field, add:

  ```
  cppbind_msvc90.lib
  ```

  Use a space to separate it from any previous entries.
- When using the Light C++ Binding, add the following files:

  ```
  lcbind_msvc90.lib
  lcbclient.lib
  ```

- Click the **Apply** button.

Specify the location of the C++ Binding debug library:

- In the **Configuration** drop-down box on the top left, select **Debug**.
- In the menu tree on the left, select **Configuration Properties > Linker > Input**.
- In the **Additional Dependencies** text field, add:

  ```
  cppbind_msvc90d.lib
  ```

  Use a space to separate it from any previous entries.
- When using the Light C++ Binding, add the following file:

  ```
  lcbind_msvc90d.lib
  ```

- Click the **Apply** button.

Specify the runtime library for Release code generation:

- In the **Configuration** drop-down box on the top left, select **Release**.
- In the menu tree on the left, select **Configuration Properties > C/C++ > Code Generation**.
- From the **Runtime Library** drop-down box, select **Multi-threaded DLL (/MD)**.
- Click the **Apply** button.

Specify the runtime library for Debug code generation:

- In the **Configuration** drop-down box on the top left, select **Debug**.
- In the menu tree on the left, select **Configuration Properties > C/C++ > Code Generation**.
- From the **Runtime Library** drop-down box, select **Multi-threaded Debug DLL (/MDd)**.
- Click the **OK** button to close the **samples Property Pages** dialog.

1.2.3 Using the C++ Binding with ACE Libraries

When using C++ binding (regular or light) with ACE libraries on Windows, Caché header files must appear after ACE header files. This is because ACE headers include Microsoft winsock2.h, and the Caché C++ binding class headers include Microsoft windows.h. When both files are included, winsock2.h must appear before windows.h, or the MSVC compiler will fail due to definition conflicts.

Here is an example that includes ACE headers. Since Sample_Person.h and Sample_Address.h are Caché C++ Binding class headers, they must appear after all of the ACE headers:

```
#include "ace/OS_main.h"
#include "ace/streams.h"
#include "ace/Log_Msg.h"
#include "ace/SOCK_Acceptor.h"
#include "ace/INET_Addr.h"
#include "ace/Service_Config.h"

#include "CPP-acceptor.h"

ACE_RCSID (non_blocking,
    test_sock_acceptor,
    "test_sock_acceptor.cpp,v 4.11 2004/10/27 21:06:58 shuston Exp")

typedef Svc_Handler<ACE_SOCK_STREAM> SVC_HANDLER;
typedef IPC_Server<SVC_HANDLER, ACE_SOCK_ACCEPTOR> IPC_SERVER;

#include "Sample_Person.h"  // C++ binding projected class header
#include "Sample_Address.h"  // C++ binding projected class header
```

1.3 Installing the Light C++ Binding

The Light C++ Binding (LCB) is a special purpose subset of the Caché C++ Binding, and has some extra requirements. For a list of the platforms that support the Light C++ Binding, see "Supported Client Platforms" in the online *InterSystems Supported Platforms* document for this release.

1.3.1 Additional LCB Requirements

The design of LCB imposes the following extra requirements:

- An environment variable named *GLOBALS_HOME* must be set to the full path of your Caché installation's <cache-root> directory (see "Default Caché Installation Directory" for the location of <cache-root> on your system). All connection attempts will fail if this environment variable is not set.

Note: **CACHEMGRDIR Deprecated**

In previous releases, the required environment variable was *CACHEMGRDIR*, set to the <cache-root>/mgr directory rather than <cache-root>. Although this variable can still be used, it is deprecated. *GLOBALS_HOME* will be used instead of *CACHEMGRDIR* if both variables are set.

- Unlike the regular C++ binding, the LCB architecture requires that Caché and the LCB application be installed on the same machine. This is necessary because they must share the same process (see Light C++ Binding Architecture).
- Because LCB depends on the low-level Callin interface, the directory containing any LCB application must have a full path that uses fewer than 232 characters.
- LCB uses a separate set of DLLs or shared libraries. For Windows, the files are: lcbind_msvc90.dll, lcbind_msvc90.lib, lcbclient.dll, lcbclient.lib, lcbind_msvc90d.dll, lcbind_msvc90d.lib

  For UNIX®, they are: liblcbind.so, liblcbclient.so
- LINUX must be #defined when building LCB applications on Linux. The compile flags should include `-DLINUX`. See the Linux LCB sample application makefiles for examples.
- LCB supports both level 1 ("minimal") and level 2 ("normal") security level installations. If Cache is installed with level 2 security, `%Service_Callin` must be enabled to permit LCB to be connected.

1.3.2 Installation on the Windows 64 bit Platform

The Light C++ Binding is available for the Windows 64 bit platform. The Caché installation for 64 bit Windows still installs a 32 bit version of the standard Caché C++ Binding, since this is required for Studio. To build 64 bit versions of the LCB sample applications, use the "win64 Release" or "win64 Debug" configurations in the MSVC project files for these applications.

In order to run debug versions of LCB applications, you will need to download the Microsoft Platform SDK, which contains 64 bit debug versions of the libraries msvcrtd.dll and msvcp60d.dll. On 64 bit systems, installing Microsoft Visual Studio only provides the 32 bit versions of these files. (This is a general issue for C++ development on 64 bit systems, not specific to Caché or to the Light C++ Binding.)

1.3.3 Running Trusted Applications on UNIX®

Light C++ Binding applications on UNIX® must be run either by root, or by a user belonging to the `cacheusr` group, unless the application has been made a "trusted application". The recommended approach for deployed applications is to make them trusted applications. When this is done, the application runs with `cacheusr` as the effective group, but any user who has execute access to the application's executable file can run it (where execute access is controlled in the usual UNIX® manner using `chmod`).

To make a trusted application, do the following:

- In the application makefile, among the flags for linking the application, specify `-rpath <pathname>` for the runtime pathnames of each of the libraries libcachet.so, liblcbind.so, and liblcbclient.so. For g++, use:

  ```
  -Xlinker -rpath -Xlinker <pathname>
  ```

  The runtime pathname is the pathname of the library in the environment in which the application will be run, which may be completely different from its pathname in the environment in which the application is built.

- Alternatively, you could create soft links in `/usr/lib` for each of the libraries:

  ```
  cd /usr/lib
  ln -s <path>/libcachet.so libcachet.so
  ln -s <path>/liblcbind.so liblcbind.so
  ln -s <path>/liblcbclient.so liblcbclient.so
  ```

  If a trusted application uses shared libraries, the runtime locations of those shared libraries must be known at build time, so users can't use `LD_LIBRARY_PATH` to point to an untrusted version of the shared library. If no runtime path was specified for a given shared library at build time, the path /usr/lib/<libraryname> is assumed by default.

- Set the owner, group, and suid bits of the LCB application. For example:

  ```
  chown <whoever> lcbdemo
  chgrp cacheusr lcbdemo
  chmod g+s lcbdemo
  ```

1.4 Sample Programs

The standard Caché installation includes several short sample programs, located in the C++ samples directory, <cachesys>\dev\cpp\samples\ (see Default Caché Installation Directory in the *Caché Installation Guide* for the location of <cachesys> on your system). The following sample programs are available:

- samples.cpp — a simple program to demonstrate the standard Caché C++ binding.
- lcbdemo.cpp — a demonstration of the Light C++ Binding.

- mttest.cpp — a multi-threaded LCB test, to verify thread safety.
- qtest.cpp — a query program using both the regular binding and multithreaded LCB.

MS Visual Studio project files for these programs are available in:..\samples\msvc90

For UNIX® platforms, run_samples.sh can be used in Caché 2008.1 and later.

Required Proxy Classes

The samples.cpp program uses classes from the Sample package in the SAMPLES namespace, which is also part of the standard Caché installation. The following C++ proxy class files must be generated (see Generating Proxy Classes) for Sample.Person and Sample.Address if they are not already present in the main C++ samples directory:

- Sample_Person.h, Sample_Person.cpp
- Sample_Address.h, Sample_Address.cpp

The lcbdemo.cpp program uses classes from the User package in the SAMPLES namespace. The following C++ proxy class files must be generated for User.Person and User.testidkey2 if they are not already present in the main C++ samples directory:

- LC_User_Person.h, LC_User_Person.cpp
- LC_User_testidkey2.h, LC_User_testidkey2.cpp

2

Generating Proxy Classes

Proxy classes are generated by the Caché C++ Class Generator (see Using the C++ Generator), which reads the definition of a Caché class and uses the information to generate a corresponding C++ class. The generated class provides remote access to an instance of a Caché object from within a C++ application.

2.1 Introduction

The C++ Generator produces C++ proxy classes that have the same inheritance hierarchy as the corresponding Caché classes. The type of a class (such as persistent or serial) determines its corresponding C++ superclass. For example, persistent classes have corresponding C++ classes derived from the C++ **Persistent_t** class included in the Caché C++ library. In case of multiple inheritance, a class becomes a subclass of the first superclass in Caché, and all methods and properties from other direct superclasses are generated as members of the proxy class.

The Caché C++ class library includes C++ versions of a number of the classes within the Caché class library, including %Persistent, %RegisteredObject, %SerialObject, the various Caché collection classes, and C++ versions of the various data type classes. In addition, the library contains the various classes used within a C++ application to manage the communication with the Caché server.

Note: The C++ binding doesn't check at runtime to see whether metadata has changed since code was generated. In particular, it doesn't check whether the application is connecting to the same namespace as at code generation time, and doesn't check whether the classes are defined in the runtime namespace. If they aren't, it will go ahead and insert data anyway, but this data won't be accessible via SQL or via the object interface.

2.2 Standard Proxy Class Methods

In addition to any methods defined by a Caché class, all C++ proxy classes inherit a set of methods from the standard Caché C++ library classes **Persistent_t** (for persistent classes) or **Registered_t** (for serial classes).

The C++ Generator also adds a set of static create and open methods to the generated classes. To protect the C++ classes from incorrect usage, the proxy class constructors are made private. The only way to instantiate a generated class `T` is to call one of the static methods **T::create_new()**, **T::open()**, or **T::openid()**, each of which returns a **d_ref<T>** object (see Using Proxy Objects). These methods are generated only if the corresponding Caché methods exist for a given class.

The static methods are defined as follows (where `My_Class` is the name of the proxy class):

- **create_new()** — Creates an object on the server by calling the **%New** method.

```
static d_ref<My_Class> create_new(
   Database* db,
   const_str_t init_val = 0,    // const_str_t is a typedef of const wchar_t*
   Db_err* err = 0)
```

- **open()** — Generated for persistent classes only. Calls **%Open** on the server to open an object using its complete Object ID.

```
static d_ref<My_Class> open(
   Database* db,
   const d_binary& ident,
   int concurrency = -1,
   int timeout = -1,
   Db_err* err = 0)
```

- **openid()** — Calls **%Openid** on the server to open an object using its extent-specific ID value.

```
static d_ref<My_Class> openid(
   Database* db,
   const const_str_t ident,    // const_str_t is a typedef of const wchar_t*
   int concurrency = -1,
   int timeout = -1,
   Db_err* err = 0)
```

2.3 Implementing Proxy Methods

C++ instance methods are generated for Caché instance methods and C++ static methods are generated for Caché class methods. When called on the client, a C++ method invokes the actual method implementation on the Caché server. If a method signature includes arguments with default values, Caché uses the same default values within the generated C++ method. For example, suppose you define a simple Caché class with one method:

```
Class MyApp.Simple Extends %RegisteredObject {
   Method LookupName(id As %String) As %String {
      // lookup a name using embedded SQL
      Set name = ""
      &sql(SELECT Name INTO :name FROM Person WHERE ID = :id)
      Quit name
   }
}
```

The resulting C++ class header would look similar to the following:

```
class MyApp_Simple : public Persistent_t {
   friend d_ref<MyApp_Simple>;
   public:
   // code
   virtual d_string LookupName(d_string id);
}
```

When a method is invoked from C++, the C++ client first synchronizes the server object cache, then invokes the method on the Caché server, and finally, returns the resulting value (if any). If any method arguments are specified as call-by-reference then their value is updated as well.

2.4 Implementing Proxy Properties

Properties in C++ proxy classes are accessed through a pair of accessor methods. Each property has a corresponding get<Property>() method to get its value and a set<Property>() method to set its value.

The values for literal properties (such as strings or integers) are represented using the appropriate C++ data type classes provided with the Caché C++ class library (such as d_string or d_int).

The values for object-valued properties are represented using the d_ref template class (see Using Proxy Objects).

For example, suppose you have defined a persistent class within Caché containing two properties, one literal and the other object-valued:

```
Class MyApp.Student Extends %Persistent {
   // Student's name
   Property Name As %String;
   // Reference to a school object
   Property School As School;
}
```

The C++ representation of MyApp.Student contains get and set accessor methods for both the Name and School properties. In addition, it provides accessors for the object Id for the referenced School object.

For example, in the Caché sample class, Sample.Person, the DOB property is defined as follows:

```
Property DOB As %Date(POPSPEC = "Date()");
```

The POPSPEC content is for populating the class with sample data and would not appear in an actual application. The C++ accessor methods for Sample.Person are:

```
virtual d_date getDOB() const;
virtual void setDOB(const d_date&);
```

When a C++ object is instantiated within C++, it fetches a copy of its property values from the Caché server and copies them into a local client-side cache. Subsequent access to the object's property values are made against this cache, reducing the number of messages sent to and from the server. Caché automatically manages this local cache and ensures that it is synchronized with the corresponding object state on the Caché server.

Property values for which you have defined a Get or Set method within your Caché class definition (to create a property whose value depends on other properties for example) are not stored within the local cache. Instead when you access such properties the corresponding accessor method is invoked on the Caché server. As this can entail higher network traffic, you should exercise care when using such properties within a client/server environment.

2.5 Naming Conventions

A generated C++ identifier, such as the name of a class, method, or variable, is usually the same as that of the corresponding Caché identifier. This section describes the exceptions to that rule.

Note: It is important to remember that, unlike ObjectScript, C++ identifiers must contain only characters A-Z, a-z, 1–9, and "_" (underscore). If a Caché identifier contains characters that are not permitted in C++, those characters will be replaced by underscores. If the Caché identifier consists of high-order Unicode characters, this may result in a C++ identifier that contains nothing but underscores. Alternate class and package names can be defined in Caché, as described below.

- *Class and Package Names*

  Because C++ does not support packages, the Caché package name for a class is added to the start of the C++ class name with the "_" character replacing the "." character. The class name itself is unchanged.

  If your Caché code defines both the package client name and the *ClientName* parameter of a class, the Caché C++ Generator will use these parameters instead of the class and package names. In Studio, you can set the package name by right-clicking on the package name, choosing Package Information in the context menu, and then entering the value of the Client Name field. The *ClientName* parameter of a class can be defined using the Class Inspector.

- *Method Names*

Typically, method names are mapped directly to C++ methods, without changes. Exceptions are:

- If the method name starts with "%", this is replaced by "sys_".
- If the method name is a C++ reserved word, "_" is prepended to the name.

- *Property Names*

  On the server you can refer directly to a Caché object's properties. To encapsulate property behavior for C++, two C++ accessor methods are generated for each Caché property. For a given property Prop, the accessor methods are **getProp()** and **setProp()**. If the property name starts with "%", it is replaced by "sys_". Hence, the accessor methods of a Color property would be **getColor()** and **setColor()**. The accessor methods of a %Concurrency property would be **get_sys_Concurrency()** and **set_sys_Concurrency()**.

- *Formal Variable Names*

  If a variable within a method formal list starts with "%", it is replaced by "_". If the name is a reserved word, "_" is prepended to the name.

For details on Caché Basic and ObjectScript naming conventions, see Variables in *Using Caché ObjectScript*, Naming Conventions in *Using Caché Objects*, Identifiers and Variables in *Using Caché Basic*, and Rules and Guidelines for Identifiers in the *Caché Programming Orientation Guide*.

2.6 Using the C++ Generator

The Caché C++ Generator is a program that generates a C++ class and header file from a Caché class definition. It is available either as a command line program, or as an option in Studio. If Caché is installed with level 3 ("locked down") security, %Service_Bindings must be enabled in order to run the Generator.

To access the Generator from Studio, select Tools > Generate C++ projection from the main menu. This option does not allow you to generate projections for the Light C++ Binding, which must use the command line program with the -lc parameter.

The command line program, cpp_generator.exe, is installed in the <cachesys>\dev\cpp\lib directory, which must be in your Path. (See Default Caché Installation Directory in the *Caché Installation Guide* for the location of <cachesys> on your system).

The syntax for the program is:

```
cpp_generator
-conn <conn>
-user <user>
-pswd <password>
-path <path>
[-class <class>] | [-class-list <filename>]
[-lc]
[-help]
```

For example:

```
cpp_generator
-conn "localhost[1972]:SAMPLES"
-user "MyUserName"
-pswd "MyPassword"
-path "./cppfiles"
-class "Sample.Person"
-lc
```

Table 2–1: C++ Generator Parameters

`-conn`	A connection string with the format `<host>[<port>]:<namespace>`. For example: `-conn "localhost[1972]:SAMPLES"`
`-user`	A string specifying the username.
`-pswd`	A string specifying the password.

`-class`
or
`-class-list`
	Either a classname or the name of a file containing a list of classnames.
	The `-class <class>` option specifies a Caché server classname. For example:
	`-class "Sample.Person"`
	The `-class-list <filename>` option specifies the name of a file containing Caché server class name strings (and nothing else), one per line. For example:
	`-class-list "\Mydir\classlist.txt"`
	where `classlist.txt` contains the following lines:
	`Sample.Person`
	`Sample.Company`

`-path`	A string specifying the directory in which the generated C++ class is to be placed.
`-lc`	Optional. If the `-lc` switch is used, the generator will produce Light C++ Binding classes.
`-help`	Optional. Displays the list of C++ Generator parameters.

The C++ Generator will automatically generate code for any other classes required to implement the specified class. For example, if you specify -class "Sample.Employee", code will also be generated for the Sample.Person and Sample.Address classes, because Sample.Employee is derived from Sample.Person, and Sample.Person has properties of type Sample.Address.

3

Using the C++ Binding

This chapter provides concrete examples of code that uses the Caché C++ binding. The following subjects are discussed:

- C++ Binding Basics — the basics of accessing and manipulating Caché database objects.
- Using Proxy Objects — manipulating objects that are references to a proxy object.
- Using Collections — iterating through Caché lists and arrays.
- Using Relationships — manipulating embedded objects.
- Using Streams — using properties that hold large sequences of character or binary data.
- Using Queries — running Caché queries and dynamic SQL queries.
- Using Transactions — controlling transactions with commit and rollback methods.

Many of the examples presented here are modified versions of the sample programs. The argument processing and error trapping (try/catch) statements have been removed to simplify the code. See Sample Programs for details about loading and running the complete sample programs.

3.1 C++ Binding Basics

A Caché C++ binding application can be quite simple. Here is a complete sample program:

```
#include "Sample_Person.h"
#include "Sample_Address.h"

int main()
{
  //  Connect to the Cache' database
  d_connection conn = tcp_conn::connect("localhost[1972]:Samples",
                                        "_system", "SYS");
  Database db(conn);

  // Create and use a Cache' object
  d_ref<Sample_Person> person = Sample_Person::create_new(&db);
  person->setName("Doe, Joe A");
  person->setSSN("123-45-6789");

  person->save();
  // Print the result
  std::cout << "w p.Name\n"
            << person->getName() << '\n';
  return 0;
}
```

This code imports the Sample header files, and then performs the following actions:

- Connects to the Samples namespace in the Caché database:
  - Defines the information needed to connect to the Caché database.
  - Creates a **d_connection** object (conn).
  - Uses the d_connection object to create a **Database** object (db).
- Creates and uses a Caché object:
  - Uses the Database object to create an instance of the Caché Sample.Person class.
  - Sets the Name property of the Sample.Person object.
  - Gets and prints the Name property.

The following sections discuss these basic actions in more detail.

3.1.1 Connecting to the Caché Database

To establish a physical connection to the Caché database, create an instance of the **d_connection** class. d_connection is a proxy class that acts as a smart pointer to a Conn_t (connection) class instance. It automatically calls **Conn_t::disconnect()** when the last reference of the Conn_t object that it refers to goes out of scope. This means that the user never has to call **Conn_t::disconnect()** directly and that a Database object will always have a valid connection that will not be accidentally disconnected. Conn_t provides a common interface for all these connections.

Before initializing a Database object with a d_connection object, the Conn_t object that the d_connection object refers to has to be connected and not be in use by some other Database instance. In order to test whether a d_connection object satisfies these requirements, you can use the **is_connected()** and **is_busy()** functions. For example, if you want to test a d_connection object called conn:

```
if (!conn->is_connected())
// code that makes conn point to an active connection
```

Because a d_connection object that doesn't point to an active connection is useless, its constructor is made private and the only way to create a d_connection object is to call **Conn_t::connect()** that returns a d_connection object that points to an active connection. If a connection could not be established, the d_connection object refers to an inactive connection.

The TCP/IP connection class is **tcp_conn**. Its static **connect()** method takes the following arguments:

- connection string — in format “host[port]:namespace”
- username
- password
- timeout
- error — optional address of a **Db_err** that will contain the error information if the connect fails.

For example:

```
Db_err conn_err;
d_connection conn = tcp_conn::connect("localhost[1972]:Samples",
      "_SYSTEM", "SYS", 0, &conn_err);
if (conn_err) {
   // error handling
   std::cerr << conn_err << '\n';
   return -1;
}
try {
   // establish the logical connection to Cache'
   Database db(conn);
   // code to use db here
}
catch(const Db_err& err) {
   // handle an error from the C++ binding library
   std::cout << err << std::endl;
}
```

3.1.2 A Sample C++ Binding Application

This section contains a simple C++ application that demonstrates the use of the Caché C++ Binding.

The sample program connects to the Caché SAMPLES database, opens and modifies an instance of a Sample.Person object saved within the database, and saves it back to the database.

```
#include "../Sample_Person.h"
#include "../Sample_Address.h"

typedef d_ref<Sample_Person> d_Sample_Person;

int main()
{
   // establish the physical connection to Cache'
   Db_err conn_err;
   d_connection conn = tcp_conn::connect("localhost[1972]:Samples",
                                         "_SYSTEM", "SYS", 0, &conn_err);
   if (conn_err)
   {
      std::cerr << conn_err << '\n';
      return -1;
   }

   try {
      // establish the logical connection to Cache'
      Database db(conn);

      std::wstring id;
      std::cout << "Enter ID of Person object to be opened:\n";
      std::wcin >> id;

      // open a Sample.Person object using %OpenId
      d_Sample_Person person = Sample_Person::openid(&db, id.c_str());

      // Fetch some properties of this object
      std::cout << "Name " << person->getName() << '\n'
               ~<< "City " << person->getHome()->getCity() << '\n'
               ~<< '\n';

      // Modify some properties
      person->getHome()->setCity("Ulan Bator");

      // Save the object to the database
      person->save();

      // Report the new residence of this person
      std::cout << "New City: " << person->getHome()->getCity() << '\n';

      return 0;
   }
   catch(const Db_err& err) {
      std::cerr << err << '\n';
      return -1;
   }

   // all objects are closed automatically
}
```

3.2 Using Proxy Objects

Object-valued proxy types are represented using the **d_ref< >** template class (or the **lc_d_ref< >** class for Light C++ Binding classes). The template takes the corresponding C++ classname as its parameter. For example, a reference to a Company object would be represented as d_ref<Company>.

An instance of d_ref<T> is a smart pointer to a proxy object of the referenced type T. This means that you can:

- Call methods of the proxy object using the "->" (pointer) operator.
- Copy one d_ref< > to another. The two d_ref< > instances will point to the same proxy object.
- Pass a d_ref< > as an argument to a proxy method that may change the d_ref< > to point to another proxy object.

Note: Two variables that represent the same server object may still point to two different proxy objects.

While the library tries to use only one proxy object for each open server object, it may have to use a different proxy object for the same server object if you open it using a different proxy class. You can use "==" (equality test) and "!=" (the inequality test) to test whether a d_ref<P> and a d_ref<Q> point to the same server object

Even though d_ref<T> acts as a pointer to T, it is not a real pointer, so testing for null or making a d_ref<T> point to null should be done via method calls **is_null()** and **make_null()**. For example,

```
d_ref<Sample_Person> p1 = Sample_Person::openid(&db, L"1");
if (p1.is_null())
   std::cerr << "the object is null";
p1.make_null();
```

These methods are used for data type classes as well.

3.2.1 Casting Proxy Objects

It is possible to cast a d_ref<P> to a d_ref<Q> if P is a subclass of Q and the type checking will work at compile time. For example,

```
d_ref<Sample_Employee> e1 = Sample_Employee::openid(&db, L"1");
d_ref<Sample_Person> p1 = e1; // ok
d_ref<Sample_Employee> e2 = p1; // gives a compile time error
```

It is also possible to cast d_ref<Q> to d_ref<P> if you know that Q is really a subclass of P, but, similarly to the interface related to null, it should be done via a function call **conv_to()**, not **dynamic_cast()**. The reason is that the "isa" relationship is really between P and Q. conv_to takes the d_ref< > that will contain the result as an argument passed by reference and if conversion is impossible sets it to null. For example:

```
d_ref<Sample_Employee> e1 = Sample_Employee::openid(&db, L"1");
d_ref<Sample_Person> p2 = e1;
d_ref<Sample_Employee> e2; p2.conv_to(e2);
```

3.2.2 Resource Management

A d_ref< > automatically takes care of all system resources associated with the proxy object that it points to. For example:

```
d_ref<Sample_Person> p1 = Sample_Person::openid(&db, L"1");
p1->setDOB(1970,2,1);
d_ref<Sample_Person> p2 = p1->getSpouse();
change_to_spouse(p2); // p2 points to the same server object as p1
```

In the first line, **openid()**, a static method of Sample_Person, creates an instance of the d_ref<Sample_Person>. In the second line, the instance is used to modify the date of birth of the Person object. In the third line, p2 is set to point to the person's

spouse, and in the fourth line, p2 is changed to point to the same person as p1. All the resources taken by p1 and p2 are released automatically when p1 and p2 go out of scope.

3.3 Using Collections

The proxies for collections are designed to fit into the framework of the C++ standard library. Proxies for %ListOfObjects and %ListOfDataTypes provide an interface which is almost identical to the interface of std::vector. Similarly, proxies for %ArrayOfObjects and %ArrayOfDataTypes provide an interface which is almost identical to the interface of std::map.

Object Collections

Proxies for collections of objects of type T contain objects of type d_obj_coln_type<T> that can be manipulated as d_ref<T>. They are different from d_ref<T> in that they ensure that all changes with the objects that they point to also take place in collections on the server. In order to change the value of an object for a particular collection and a given key, it is enough to assign the object a different d_ref<T>. These objects also amortize the cost of opening objects in collections by delaying opening of objects on the server until they are needed on the client.

Primitive Data Type Collections

Proxies for collections of data type T contain objects of type d_prim_coln_type<T> that can be manipulated as T. They are similar to the objects contained in collections of objects, but their initialization is not delayed because the overhead is insignificant.

3.3.1 Interface

A %ListOfObjects that holds elements of type T is generated as d_obj_vector<T> that holds elements of type d_obj_coln_type<T> that can be manipulated as d_ref<T>. An %ArrayOfObjects that holds elements of type T is generated as d_obj_map<T> that also holds elements of type d_obj_coln_type<T>.

A %ListOfDataTypes that holds elements of type T is generated as d_prim_vector<T> that holds elements of type d_prim_coln_type<T> that can be manipulated as T. An %ArrayOfDataTypes that holds elements of type T is generated as d_prim_map<T> that also holds elements of type d_prim_coln_type<T>.

Similar to other objects, collections have to be manipulated via d_ref<T>, which means that they can be instantiated by calling the static methods **create_new()**, and **openref()**, the methods used to initialize proxies for serial objects.

d_obj_coln_type<T> and d_prim_coln_type<T> can be constructed from T, which means that any function that takes d_obj_coln_type<T> or d_prim_coln_type<T> can be also called with T if the argument is constant.

3.3.2 Examples

The following are simple examples of collections in use.

Constructors

If a class CPP.Coln has a property, Lst, which is a %ListOfObjects that holds one or more instances of Sample.Person, then that property can be accessed in the C++ binding by

```
d_ref< d_obj_vector<Sample_Person> > list = obj->getLst();
```

where obj is an object of type d_ref<CPP_Coln>.

Element Access

The third Sample.Person in the collection pointed to by list can be accessed by

```
(*list)[2]
```

or

```
*(list->begin()+2)
```

The person's name can be accessed by

```
(*list)[2]->getName()
```

or

```
(*(list->begin()+2))->getName()
```

"->" is used instead of "." because list is a d_ref<T>, so it acts like a pointer to the actual object.

Methods

p(of type d_ref<Sample.Person>) can be inserted into the collection pointed to by list by

```
list->push_back(p);
```

the second Sample.Person in the collection pointed to by list can be erased by

```
list->erase(list->begin()+1);
```

Algorithms

All persons of the collection pointed to by list can be printed by

```
class Print_person : public std::unary_function<d_ref<Sample_Person>, int> {
private:
   std::ostream& out;
public:
   explicit Print_person(std::ostream& o)
      : out(o)
      {};
   result_type operator()(const argument_type& p) const;
      { out << p->getName() << std::endl; return 0; };
};
void print_people(d_ref<CPP_Coln>& obj)
{
   d_ref< d_obj_vector<Sample_Person> > list = obj->getLst();
   std::for_each(list->begin(), list->end(), Print_person(std::cout));
};
```

3.3.3 Using Collection Elements in Methods

Most of the time, it is possible to forget that the actual type of a collection proxy is d_prim_coln_type<T> or d_obj_coln_type<T> (instead of T or d_ref<T>). However, these types (T or d_ref<T>) cannot be used as non-constant arguments to functions, although they can be used as constant arguments. Even if it's possible to get around the compilation error that should result from this incorrect usage, the seemingly changed value will not change in the collection. The proper way to change an element of a collection this way is to use a temporary and then assign the changed value to the element of the proxy object. For example:

```
d_ref<Sample_Person> p = (*list)[2];
change_to_someone_else(p);
(*list)[2] = p;
```

But the following works fine if **calc_some_value()** does not change its argument:

```
int val = calc_some_value((*list)[2]);
```

3.3.4 Data in Collection Proxies

In order to fit into the C++ standard library framework, the proxies for collections have to contain data. This means that two different proxies of the same collection may change the data on the server and their representation of the collection,

but they may not know about each other, so they may lose synchronization. This problem does not exist if a collection is accessed via proxy objects of the same type (which is the intended usage) but if the types are different, loss of synchronization is possible.

3.4 Using Streams

Caché allows you to create properties that hold large sequences of characters, either in character or binary format; these sequences are known as streams. Character streams are long sequences of text, such as the contents of a free-form text field in a data entry screen; binary streams are usually image or sound files, and are akin to BLOBs (binary large objects) in other database systems. When you are writing to or reading from a stream, Caché monitors your position within the stream, so that you can move backward or forward.

Here is a simple program that creates and manipulates a Caché stream object:

```
#include <database.h>
#include <streams.h>

int main(){
// establish the physical connection to Cache'
   Db_err conn_err;
   d_connection conn = tcp_conn::connect("localhost[1972]:Samples",
      "_SYSTEM", "SYS", 0, &conn_err);

// establish the logical connection to Cache'
// database and create a low level stream object.
   db(conn);
   d_ref<d_char_stream> stream = d_char_stream::create_new(&db);

// create an IOStreams extension stream object, put
// "Hello, World!" in the stream, and rewind the stream
   d_iostream io(stream);
   io << "Hello, World!";
   io.rewind();

// read each word and copy it to standard output
   std::string s;
   while (io.good()) {
      io >> s;
      std::cout << s << ' ';
   }
   std::cout << '\n';
   return 0;
}
```

3.5 Using Relationships

As in Caché, relationships are treated as properties. For example, the relationship between Sample.Employee and Sample.Company results in the following generated code:

```
class Sample_Employee : public Sample_Person {
// code
  virtual d_ref<Sample_Company> getCompany() const;
  virtual void setCompany(const d_ref<Sample_Company>&);
// code
};

class Sample_Company : public Persistent_t {
// code
  virtual d_ref< d_relationship<Sample_Employee> > getEmployees() const;
  virtual void setEmployees(const d_ref< d_relationship<Sample_Employee> >&);
// code
};
```

The **d_relationship<T>** class template is a standard container that supports iterators **begin()** and **end()**, and reverse iterators **rbegin()** and **rend()**. Here is a simple program that uses this relationship to access a list of employees:

```
 #include "Sample_Company.h"
#include "Sample_Employee.h"

#include <algorithm>

class Print_person : public std::unary_function<d_ref<Sample_Person>, int> {
  private:
    std::ostream& out;
  public:
    explicit Print_person(std::ostream& o)
      : out(o)
      {}
    result_type operator()(argument_type p) const
      { out << p->getName() << std::endl; return 0; }
};

int main()
{
// establish the connection to Cache'
  Db_err conn_err;
  d_connection conn = tcp_conn::connect(
    "localhost[1972]:Samples", "_SYSTEM", "SYS", 0, &conn_err);
  Database db(conn);

  d_ref<Sample_Company> obj = Sample_Company::openid(&db, L"1");
  d_ref< d_relationship<Sample_Employee> > r = obj->getEmployees();

// print the names of all employees in the order they are
// stored in the relationship
  std::for_each(r->begin(), r->end(), Print_person(std::cout));
  std::cout << std::endl;

// print the names of all employees in the reverse order
  std::for_each(r->rbegin(), r->rend(), Print_person(std::cout));

  return 0;
}
```

3.6 Using Queries

A Caché query is treated as type **d_query**, which is designed to fit into the framework of ODBC but provides a higher level of abstraction by hiding direct ODBC calls behind a simple and complete interface of a dynamic query. It has methods for preparing an SQL statement, binding parameters, executing the query, and traversing the result set.

The basic procedure for using a Caché query is as follows:

- *Prepare*

  The method for preparing a query is:

  ```
  void prepare(const wchar_t* sql_query, int len);
  ```

  where sql_query is the query to execute.

- *Set Parameters*

  Assign values for any parameters.

  ```
  template<typename D_TYPE> void set_par(int index, const D_TYPE& val);
  ```

  This function sets parameter index to val. The function works for any C++ binding data type. This function can be called several times for the same parameter. The previous value for the parameter will be lost. The new value need not be of the same type.

- *Execute*

  This function executes the query. Do not call it until all parameters are bound.

  ```
  void execute();
  ```

- *Fetch*

  Determine if there is data available for retrieval.

  ```
  bool fetch();
  ```

  This function tries to get the next row from the result set. It returns true if it succeeds and false if it fails. This function does not fetch any data. It only checks if there is more data to be fetched.

- *Retrieve Data*

  If the query successfully executes, it returns a result set with one row for each record. The data in each row can be accessed by iterating the row from left to right by calling

  ```
  void get_data(T* val);
  ```

  where T can be any data type of C++ binding. For d_string, you may specify how you want the data to be converted:

  ```
  void get_data(d_string* val, str_conv_t conv = NO_CONV);
  ```

  The default is not to convert the data (the “NO_CONV” value). Using “CONV_TO_MB” converts the data to multibyte; using “CONV_TO_UNI” converts the data to Unicode.

  After each call, the implicit iterator moves to the next column (so you cannot access the data in the same column twice by calling **get_data()** twice). This eliminates the need for the implementation to store all the data on the client. Otherwise, using queries could result in large memory overhead. Applications that need random access to the data should read all the data in a row first.

  You can skip one or several columns by calling:

  ```
  void skip(int num_cols = 1)
  ```

  You can get the index of the column that will be processed next by calling:

  ```
  int get_cur_idx();
  ```

Here is a simple function that queries Sample.Person:

```
void example(Database& db)
{
   d_query query(&db);

   d_string name;
   d_int id;
   d_date dob;

   const wchar_t* sql_query = L"select ID, Name, DOB from Sample.Person
      where ID > ? and FavoriteColors = ? ";
   int size = wcslen(sql_query);

   query.prepare(sql_query, size);

   query.set_par(1, 1);
   query.set_par(2, "Blue", 4);

   query.execute();

   std::wcout << L"results from " << std::wstring(sql_query) << std::endl;
   while (query.fetch())
   {
      query.get_data(&id);
      query.get_data(&name);
      query.get_data(&dob);
      std::cout << std::setw(4) << id
                ~<< std::setw(30) << std::string(name)
                ~<< std::setw(20) << dob << std::endl;
   }
   std::cout << std::endl;
}
```

3.7 Using Transactions

There are two options for performing transactions.

- Database class methods — Perform standard nested transactions.
- Transaction class methods — No nesting, but guarantees an automatic rollback if an exception is encountered.

3.7.1 Using Database Class Methods

Nested transactions can be performed using following methods of the **Database** class (also inherited by the **LC_Database** class):

- **tstart()** — Starts a new level of nested transaction.
- **tcommit()** — Marks the current level of the transaction as committed. Committing the outermost level causes the entire transaction to be committed.
- **trollback()** — Rolls back all levels of the transaction.
- **tlevel()** — Returns the current transaction level.

For example:

```
for (i = 0; i < numPersons; i++) {
   db->tstart()
// perform the transaction
   {...}
   if (goodtransaction)
      db->tcommit();
   else
      db->trollback();
}
```

The **tstart()** and **tcommit()** methods are also called implicitly whenever a proxy object's **save()**, **insert()**, **update()**, or **delete_object()** member functions are called. This ensures a transaction scope for temporary locks, and for rollback in case of error.

3.7.2 Using Transaction Class Methods

The **Transaction** class provides a guaranteed automatic rollback in case of exceptions. When a Transaction object goes out of scope, the transaction is rolled back if neither **commit()** nor **rollback()** has been called. This class does not allow nested transactions.

The Transaction methods are:

- **Transaction()** — The constructor starts the transaction (unlike a Database object, which requires a call to **tstart()**).
- **Transaction::commit()** — Commits the transaction. Calling **commit()** more than once for the same Transaction object does nothing (unlike **Database::tcommit()**, which can be called repeatedly to commit multiple levels of a nested transaction).
- **Transaction::rollback()** — Rolls back the transaction. Called automatically if the Transaction object goes out of scope before the transaction is committed or rolled back.

For example:

```
for (i = 0; i < numPersons; i++) {
   Transaction tran(db);
// perform the transaction
   {...}
// transaction will rolled back if an exception
// occurs before this point
   if (goodtransaction)
      tran->commit();
   else
      tran->rollback();
}
```

As shown above, the Transaction object must be instantiated with:

```
Transaction tran(db);
```

rather than:

```
Transaction tran = new Transaction(db);
```

If the object is allocated from the heap using new, it will not automatically be destroyed when it goes out of scope, and therefore the transaction will not be rolled back.

Using Transactions with the Light C++ Binding

In Light C++ Binding applications, if an exception is thrown within the projection class member functions save(), delete_object(), insert(), or update(), automatic rollback occurs. Exceptions thrown in other contexts do not cause transactions to be automatically rolled back, unless an instance of the Transaction class has been declared as an automatic variable in a scope within which the exception is thrown, and the exception is not caught within that scope.

4

Dynamic Binding

Instead of generating C++ classes, you can use a Dyn_obj, a class that allows you to work with Caché classes dynamically. This can be useful for writing applications or tools that work with classes in general and that do not depend on particular Caché classes. However, this generality comes at the price of the lack of static analysis of the code by a C++ compiler and a slightly slower performance. InterSystems recommends that you work with generated classes if you know what classes you need at the time of writing a program.

4.1 Construction of a Dyn_obj Proxy

As any other proxy, Dyn_obj has the following methods for creating a proxy object:

```
static d_ref<Dyn_obj> openref(Database* db,
   int ref,
   const_name_t type);      // const_name_t is a typedef for const wchar_t*

static d_ref<Dyn_obj> create_new(Database* db,
   const_name_t type,
   const_str_t init_val = 0,   // const_str_t is a typedef of const wchar_t*
   Db_err* err = 0);

static d_ref<Dyn_obj> open(Database* db,
   const d_binary& oid,
   int concurrency = -1,
   int timeout = -1,
   Db_err* err = 0);

static d_ref<Dyn_obj> openid(Database* db,
   const_name_t type,
   const_str_t id,
   int concurrency = -1,
   int timeout = -1,
   Db_err* err = 0);
```

They differ from the same methods for generated classes only in the additional parameter for the name of the class.

In addition, Dyn_obj has a method, init(), that allows you to construct a Dyn_obj instance from the result of get_property(), run_obj_method(), and run_class_method() methods from Dyn_obj:

```
static d_ref<Dyn_obj> init(t_istream& in, Database* db);
```

4.2 Construction of Values from Calling Dyn_obj Methods

All values returned from calling Dyn_obj methods are returned inside t_istream (transport input stream). If the value is an instance of a data type, then it can be constructed from t_istream alone, such as:

```
d_int val(in);
```

where in is of type t_istream.

If the value is an object, then t_istream contains its OREF and class name. This means that you will also need a Database object. If you know the type of the returned object at compile time, then you can use the openref() method of that type. For example, if the returned object is of type Sample.Person, then the object can be constructed as

```
d_ref<Sample_Person> p = Sample_Person::openref(db, d_int(in));
```

d_int(in) creates a temporary d_int value that contains the OREF, and the d_int value is converted to an int. Otherwise, it can be constructed as

```
d_ref<Dyn_obj> p = Dyn_obj::openref(in, db);
```

and the type of the object will be read from the input stream. The signature of the init() method matches constructors from other D_type instances, except that the Database pointer is not 0.

4.3 Properties and Methods

Once you construct a Dyn_obj proxy, you can use it to get or set property values, run queries, and run and methods:

```
t_istream& get_property(const_name_t prop_name);
void set_property(const_name_t prop_name, D_type* val);
void get_query(const_name_t query_name, d_query& query) const;
// const_name_t is a typedef for const wchar_t*
```

In order to run a method, you should pass an array of pointers to arguments (D_type*) and the number of arguments. You can use the buffer allocated by a Database object, such as:

```
D_type* args[2];
args[0] = &arg1;
args[1] = &arg2;
// code
```

it can store max_num_obj_args pointers (the maximum allowed number of arguments). The signature of run_obj_method() is:

```
t_istream& run_obj_method(
   const_name_t mtd_name,   // const_name_t is a typedef for const wchar_t*
   D_type** args,
   int num_args);
```

To run a class method, you can use run_class_method():

```
static t_istream& run_class_method(Database* db,
   const_name_t cl_name,   // const_name_t is a typedef for const wchar_t*
   const_name_t mtd_name,
   D_type** args,
   int num_args);
```

The values for arguments passed by reference will be changed inside these methods. For example,

```
const_name_t cl_name = __NAME_V(Sample.Person);

D_type* args[2];

d_ref<Dyn_obj> p = Dyn_obj::openid(db, cl_name, __STRING_V(1));
// create a proxy

d_string dob(p->get_property(__NAME_V(DOB)));
// get date of birth

d_int arg1(2);
d_int arg2(3);

args[0] = &arg1;
args[1] = &arg2;
d_int res = p->run_obj_method(__NAME_V(Addition), args, 2);

args[0] = &dob;
d_int age =
   Dyn_obj::run_class_method(db, cl_name, __NAME_V(CurrAge), args, 1);

d_query by_name(db);
p->get_query(__NAME_V(ByName), by_name);
```

Note: The pointers in the array of arguments cannot point to temporary variables. Code such as the following may result in a crash:

```
args[ii] = &d_int(1); // DO NOT USE!
```

5

The Light C++ Binding

The Light C++ Binding (LCB) is most useful in applications where high performance is the primary concern, and class design is relatively simple. It is a special purpose, limited subset of the C++ binding, intended primarily for applications that must load data into a persistent database at very high speed. For example, some applications capture raw real-time data at such a high rate that it must typically be stored it in an in-memory database before it can be processed and transferred to persistent storage. LCB can offer a similar level of performance while also offering failover, which is not possible with an in-memory database.

For basic object manipulation (creating objects, opening objects by Id, updating, and deleting), LCB is ten to twenty times faster than the standard C++ binding.

Constraints on LCB Applications

The most significant tradeoff for this added speed is a limitation in the complexity of the objects to be stored. The primary constraints on LCB applications are as follows:

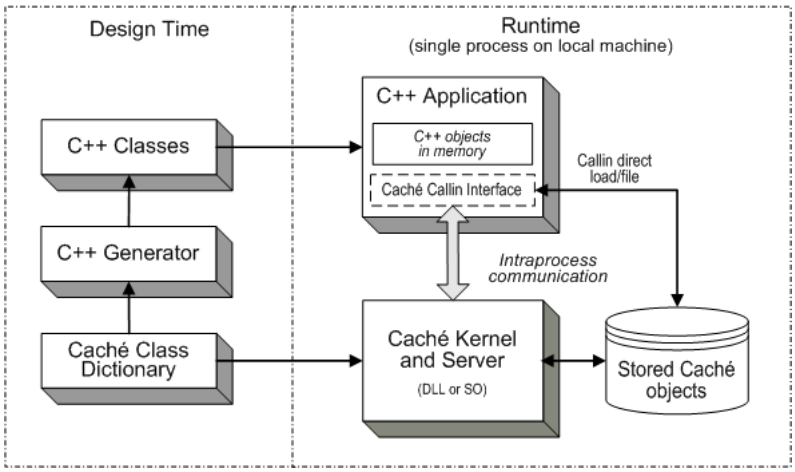
- Caché must be installed with the "minimal" or "normal" security option.
- No dynamic binding (classes must be known at code generation time)
- Since projected LCB objects do not have corresponding Caché objects in memory on the Server, Caché method calls cannot be used.
- The default storage structure must be used.
- No integrity constraints or data validation, except checking for duplicate idkeys during insert (see Error Checking) and checking for duplicates in unique indexes.
- No registered (transient) classes
- No transient or calculated properties
- No collections except lists or arrays of datatype.
- No streams, relationships, or stored OIDs
- Only the following Caché datatype classes are supported for properties and indices: %Integer, %Float, %Decimal, %Double, %String, %Date, %Time, %TimeStamp, and %Currency.
- Idkey properties must be of type %String, %Integer, or %Date
- Regular and bitmap indices only (no bitslice indices).
- No triggers (except through SQL)
- No non-standard LogicalToStorage or StorageToLogical conversions
- The only supported collation types are SQL string, SQL upper, and exact

See Installing the Light C++ Binding for LCB installation requirements.

5.1 Light C++ Binding Architecture

LCB gains much of its improved performance because it provides a much faster way for the C++ application to communicate with the Caché Object Server. LCB does not use the standard binding's client/server architecture, with Caché running as a separate server process. Instead, Caché is loaded as a DLL or shared object, allowing it to execute as part of the application process. The following diagram illustrates this structure:

Figure 5–1: Light C++ Binding Architecture



Both LCB and the standard C++ binding use the Caché C++ Generator and the Caché C++ Binding Library, but there are major differences at runtime:

- Rather than using TCP/IP to communicate with Caché, LCB uses the Callin API to make intraprocess calls to the Caché kernel and Object Server. Although the server and the C++ application must be on the same machine, Caché ECP can still be used by the application to access data on remote machines.
- For simple classes, LCB will use Callin functions to perform object loading and filing directly, entirely bypassing the server routines. The Callin interface provides extremely efficient low-level functions for accessing Caché databases.
- Objects are loaded directly from a Caché database into a corresponding C++ object. The Caché Server does not maintain copies of these objects in memory. The C++ objects are not just proxies for objects on the Server, but contain actual data.
- Since the C++ objects contain the only in-memory copy of the data, the C++ application can continue to work with them even if there is no connection between the Caché Object Server and the persistent database. This is especially important for multithreaded applications that want to share a connection between two or more threads.
- Although most LCB properties behave just as they do in the standard binding, getters for non-numeric types return a reference, as optimization. For example, the getter for a `string` property might have the following signature:

```
virtual const d_string& getname() const;
```

5.2 LCB Classes in the Caché C++ Library

The Caché C++ library implements the Light C++ Binding with a special set of classes, most of which are LCB versions of classes used by the standard binding. The following classes provide the functions that you will need for an LCB application:

- **LC_Persistent_t** — is the base class used to generate LCB persistent projection classes. It is the LCB version of **Persistent_t** (see Generating Proxy Classes).
- **LC_Serial_t** — is the base class used to generate LCB serial projection classes.
- **LC_Database** — is the LCB version of **Database**.
- **lc_conn** — is the LCB version of **tcp_conn**.
- **lc_d_ref<T>** — is the reference class template for LCB objects (see Using Proxy Objects). It is the LCB version of **d_ref<T>**.
- **LC_Batch** — is a special batch insert class that provides an alternate interface for high speed inserts.

5.3 Connections and Multithreading

An LCB connection uses the following classes:

- **d_connection** — is the physical connection handle. This class is used by both LCB and the standard binding.
- **LC_Database** — is a subclass of the standard Database class. It is initialized from an open d_connection, just like Database.
- **lc_conn** — is the LCB connection class, used in place of the standard tcp_conn class. The connect method, **lc_conn::connect()** has the same syntax as **tcp_conn::connect()**.

The following code fragment demonstrates how these classes are used. Compare the calls in this code to the example in Connecting to the Caché Database. Only the class names have changed.

```
#  include "lc_connection.h"
#  include "lc_database.h"
   Db_err conn_err;

   d_connection conn =  lc_conn::connect(
      conn_str, user, pwd, timeout, &conn_err);
   LC_Database db(conn);
```

Important: For all LCB applications, the GLOBALS_HOME environment variable must be set to the root directory of the Caché instance (see "Additional LCB Requirements"). All connection attempts will fail if this environment variable is not set.

5.3.1 Multithreading

LCB is thread-safe, and uses the Callin API to provide parallel multithreading on multiprocessor machines. Performance is similar to that of multithreaded Callin applications. Unlike Callin applications, LCB applications do not require detailed knowledge of class implementations, since the C++ code can be regenerated whenever class definitions change.

A separate database connection (including both **d_connection** and **LC_Database** objects) must be used in each thread. If member functions of LC_Database, or of projection objects connected to an LC_Database instance, are called in a different

thread than the thread in which the LC_Database object was created, the following exception is thrown: `"Database connection may not be shared by multiple threads"`

For examples of multithreaded LCB code, see the mttest.cpp and qtest.cpp sample programs located in <cachesys>\Dev\cpp\samples (see Default Caché Installation Directory in the *Caché Installation Guide* for the location of <cachesys> on your system).

5.3.2 Connections and Multiple Threads

Projection objects can only be connected to one database connection at a time, and can only be used in the thread in which that database connection was created. To use a projection object in more than one thread, use the projection object methods **disconnect()** and **connect()**. The **is_connected()** method can be used to determine the connection state.

- A projection object is connected when it has been returned by **create_new()** or **openid()**.
- A projection object must be detached (see Attaching and Detaching LCB Objects) before being disconnected, and must be disconnected before being (re)connected to a different database connection
- Using **connect()** and **disconnect()** permits one thread to be a factory for projection objects, which are inserted into the database in a different thread.
- Projection object member functions that access the database enforce thread affinity, but **get<name>()** and **set<name>()** functions do not, and are not thread-safe.
- Separate database connections can be used in parallel in different threads.

5.3.3 Attaching and Detaching LCB Objects

Since the C++ objects contain the only in-memory copy of the data, the C++ application can continue to work with them even if there is no connection between the Caché Object Server and the persistent database. This is especially important for multithreaded applications that want to share an object between two or more threads.

An object is *attached* when:

- It has been returned from **openid()**, or it has just been created and **save()** or **insert()** has been called.

An object is *detached* when:

- It has just been returned from **create_new()**
- it has just been deleted by calling **delete_object()**
- **detach()** has been called

5.3.4 Transactions and Multithreading

Each thread / database connection has its own transaction context:

- Threads look just like separate processes to Caché. Two threads may not use the same connection object.
- Locks acquired by one thread block attempt to acquire the same lock in another thread.
- **Database** transaction methods **tstart()**, **tcommit()**, and **trollback()** (see Using Transactions) only affect the calling thread.

5.4 Using Objects in LCB

LCB projected objects are different from standard C++ binding objects in a number of ways. It is important to understand these differences in the following situations:

- Using persistent object references as properties
- Using classes that inherit from other persistent classes
- Using embedded serial object properties
- Using list and array properties

5.4.1 Using Persistent Object References as Properties

When a Caché class uses a persistent class as a property, LCB projects the property as a reference to a persistent object. Each persistent reference property consists of an `lc_d_ref` that points to the referenced object, and a string representing the id of the object.

Property accessors get and set `lc_d_ref<LC_xxx>&` where `LC_xxx` is the client type of the property (for example, `LC_User_Person` is the client type for Caché class `User.Person`). Logically, the accessors work as follows:

- *getters* — If the object pointer is null and the id string is not null, the getter calls `openid()` and sets the id string to the returned id, then returns the object pointer. If the object pointer and the id string are both null, it returns a null object pointer.
- *setters* — The setter sets the object pointer to the referenced object. If the object has an id, the setter puts it in the id string, or otherwise sets the id string to null/empty.

For example, assume a Caché class `Sample.Employee` that contains a property `Office` of persistent class `Sample.Office`. Each class has a user-assigned idkey (`Office` on the office's city, and `Employee` on the employee's name). The following example opens the `Office` object for the New York office, and the `Employee` object for John Kent. If Kent is currently assigned to the Boston office, he is reassigned to the New York office:

```
lc_d_ref<LC_Sample_Office> o = LC_Sample_Office::openid(db, "New York");
lc_d_ref<LC_Sample_Employee> e = LC_Sample_Employee::openid(db, "Kent, John");
if (e->getOffice()->getCity() = "Boston")  {
   e->setOffice(o);
   e->save();
}
```

In generated code for classes that contain persistent reference properties, the `save()` function, and getter and setter functions for persistent reference properties, are generated in the `.cpp` file rather than inline in the `.h` file, as is normally the case. This prevents the `.h` file of a projected class from indirectly including itself due to circular references.

Getter functions for persistent reference properties, unlike other getter functions, are not declared as `const`, since they must modify the referencing object when they cause it to swizzle.

5.4.2 Using Classes that Inherit from Other Persistent Classes

The Light C++ Binding supports retrieval and update of persistent objects whose classes inherit from other persistent classes. Batch insert is not supported for classes that inherit.

The LCB projection class for a Caché class inherits from the projection class of that Caché class's superclass. With multiple inheritance, this applies only to first class in the list of superclasses. Other superclasses are transparent to LCB, which simply sees their properties as belonging directly to the subclass.

Opening a Subclass Object from its Superclass

An object of a class that inherits from another class can be opened by the `openid` method of the inherited superclass, provided that the C++ code for the subclass has been generated and linked into the application.

For example, assume a class `LC_Sample_Employee` that inherits from class `LC_Sample_Person`. If variable `sub_id` contains the id of an `LC_Sample_Employee` object, the following statement can be used to open it:

```
lc_d_ref<LC_Sample_Person> newemployee = LC_Sample_Person::openid(db, sub_id);
```

LCB will detect the actual class of the object to be opened, and will transparently open `newemployee` as an object of subclass `LC_Sample_Employee`.

Since LCB performs its storage operations in client-side code, it needs to be aware of the actual class of the object. If the generated code for a subclass is not linked into the application, LCB cannot open the object as that class. Instead, it will open the object as the nearest ancestor of the subclass for which linked code is available. In that case, an exception is thrown if the application attempts to update or delete the object, because LCB cannot ensure that all index entries will be properly updated or deleted.

Because LCB transparently opens an object as the most specific-possible subclass, if the same object is opened once from the superclass and once from the subclass, both `lc_d_refs` will reference the same subclass object in memory.

Note: Unlike LCB, the standard C++ binding does not have the ability to transparently open an object as a more specific subclass than the class whose openid method was called. However, since the standard binding performs all of the actual storage operations on the Caché Server (which knows the actual class of the object), it is still possible to update or delete objects opened from a superclass that are actually of a subclass.

5.4.3 Using Embedded Serial Object Properties

LCB objects can have embedded serial objects as properties. An LCB serial property has a getter function that returns a pointer to the type of the property. For example, assume that Caché class `Sample.Person` contains a property `Home` of serial class `Sample.Address`. The getter function for the projected serial property would be `LC_Sample_Address *getHome()`. The following example retrieves the old street from a person's home address, and then assigns a new street value:

```
lc_d_ref<LC_Sample_Person> p = LC_Sample_Person::openid(db, 1);
d_string oldStreet;
d_string newStreet("Broadway");
oldStreet = p->getHome()->getStreet();
p->getHome()->setStreet(newStreet);
p->save();
```

LCB serial classes (projected from Caché classes based on %SerialObject) inherit from **LC_Serial_t**. The generated code for an LCB serial class consists only of a header file (unlike persistent LCB classes, which have code in both `.h` and `.cpp` files). For example, a header file named `LC_Sample_Address.h` would contain the only projection code generated for Caché class `Sample.Address`.

5.4.4 Using List and Array Properties

The only collection types that can be used with LCB are lists or arrays of datatype. The following Caché datatype classes are supported: %Integer, %Float, %Decimal, %Double, %String, %Date, %Time, %TimeStamp, and %Currency (see the Reference for Simple Datatype Classes for a discussion of these datatypes).

Long strings must be enabled in order to store lists or arrays larger than 32K. To enable support for long strings system-wide, open the Management Portal and select System Administration > Configuration > System Configuration > Memory and Startup, then select the **Enable Long Strings** check box. The long string setting can also be overridden temporarily using $ZUTIL(69,69).

Lists of Datatypes

LCB projects list properties as `std::vector<d_xxx>`, where `d_xxx` is a valid LCB datatype class. For example, a list of %String would be projected as `std::vector<d_string>`.

Arrays of Datatypes

LCB projects array properties as `std::map<d_string,d_xxx>`, where `d_xxx` is the property's array element datatype. For example, an array of %String would be projected as `std::map<d_string,d_string>`.

Important: When defining an array property in a Caché class, the property parameter `STORAGEDEFAULT` must be set to `list`. For example, a Caché class might define a %String array as follows:

```
Property MyArray As array Of %String(STORAGEDEFAULT = "list");
```

The default parameter value, `array`, is not supported in LCB.

5.5 Standard LCB Projection Class Methods

The C++ Generator provides LCB projection classes with a set of methods similar to those generated for standard proxy classes (see Standard Proxy Class Methods). The methods listed in this section are added to all projection classes.

BuildIndices()

Invokes the Caché **%BuildIndices** class method (see %Library.Persistent) to completely rebuild all indices of the class. It can enhance performance when used after **save()** or **delete_object()** (called with `defer_indices` set to `true`).

```
static InterSystems::d_status BuildIndices(
   InterSystems::LC_Database * db)
```

connect()

Connects (or reconnects) a projection object to a database connection. An exception will be thrown if the object is not disconnected.

```
void connect(
   InterSystems::LC_Database * db)
```

create_new()

Creates a new projection object. The projection object is not saved to the database until **save()** or **insert()** is called. Once **save()** is called, the object is attached. If **save()** is called again, the object is updated. To cause **save()** to create a different new object, you must first call **create_new()** again, or call **detach()**.

```
static lc_d_ref<LCBclass> create_new(
   LC_Database* db,
   const_str_t init_val = 0,    // const_str_t is a typedef of const wchar_t*
   Db_err* err = 0)
```

The **create_new()** method inherited from **LC_Persistent_t** is overridden by a version that returns a reference to the specific class (**lc_d_ref<LCBclass>** where `LCBclass` is the name of the projection class).

delete_object()

Deletes an open object from the database. This method allows you to delete an open object from the database without destroying the projection object.

```
d_status delete_object(
   bool defer_indices = false,
   int timeout = -1,
   Db_err* err = 0)
```

After **delete_object()** is called, the projection object still contains property data values, but is no longer attached. If the object was locked, the lock is released.

Calling **save()** right after **delete_object()** will create a new database object with the same values, except for any properties explicitly set to different values

detach()

Detaches a projection object from an object in database. This is a no-op if the object is not attached). If a retained lock is held on the object, it is released.

```
void detach()
```

Property values are retained in the projection object. This permits reusing the projection object to create multiple new database objects, avoiding the overhead of calling **create_new()** and copying properties that have same values in multiple objects.

direct_save()

A very-high-performance alternate interface for creating new objects while avoiding the overhead of projection object instantiation and data conversions. It can only be used to insert new objects, not to update existing objects. **direct_save()** can be used with a Unicode database, but it only supports ASCII characters in strings.

```
static d_status direct_save(
   LC_Database* db,
   const char *<prop1>,
   ...,
   const char *<propN>)
```

The parameters `<prop1>...<propN>` are properties of the class.

This method does not create index entries (although you can use **BuildIndices()** after saving).

disconnect()

Disconnects a projection object from a database connection. An exception will be thrown if the object is not detached, or if the object's current database connection was created in a different thread.

```
void disconnect()
```

id()

Gets an id from an attached object. The `*buf` parameter can be either a multibyte string or a Unicode string.

```
int id(
   wchar_t *buf,
   size_t bufsiz)

int id(
   char *buf,
   size_t bufsiz)
```

insert()

Inserts a new object into the database.

```
d_status insert(
   bool defer_indices = false,
   int timeout = -1,
   Db_err* err = 0)
```

If called for an attached object, **insert()** detaches the object, and attempts to insert a new object with the same property values. For a user-assigned idkey, this causes a duplicate idkey exception.

is_attached()

Returns `true` if the projection object is attached.

```
bool is_attached()
```

is_connected()

Returns `true` if the projection object currently has a connection to the database.

```
bool is_connected()
```

openid()

Opens an existing object, specified by id. Projection object data members are set to current values from the database object, and the projection object is attached to the database object.

```
static lc_d_ref<LCBclass> openid(
   LC_Database* db,
   const_str_t ident,         // const_str_t is a typedef of const wchar_t*
   int concurrency = -1,
   int timeout = -1,
   Db_err* err = 0)

static lc_d_ref<LCBclass> openid(
   LC_Database* db,
   const char * ident,
   int concurrency = -1,
   int timeout = -1,
   Db_err* err = 0)
```

The `ident` parameter can be either a multibyte string or a Unicode string. The **openid()** method inherited from **LC_Persistent_t** is overridden by a version that returns a reference to the specific class (**lc_d_ref<LCBclass>** where `LCBclass` is the name of the projection class).

release_shared_lock()

Explicitly releases a shared lock when the projection object has been opened with concurrency mode `LC_CONCURRENCY_SHARED_RETAINED` (see LCB and Concurrency).

```
void release_shared_lock()
```

save()

Saves the projection object to the database. It must be explicitly called to save a newly-created object to the database, or to save changes to the database.

```
d_status save(
   bool defer_indices = false,
   int timeout = -1,
   Db_err* err = 0)
```

If the projection object is attached, **save()** updates the object. If the object is detached, **save()** creates a new object. In transactions, **save()** brackets the update with implicit calls to **tstart()** and **tcommit()** (see Using Transactions).

set_from_err_list()

Sets the projection object's property values from the error list entry returned by a batch insert (see Using LCB Batch Insert).

```
void set_from_err_list(
   const std::pair<d_status,d_binary> & list_entry)
```

update()

Updates an existing database object.

```
d_status update(
   bool defer_indices = false,
   int timeout = -1,
   Db_err* err = 0)
```

The object must already be attached. If not, the following exception is thrown: `"Object must be opened or inserted before being updated"`.

5.6 Using Queries in LCB Applications

Queries in LCB applications use the same API calls as the regular binding (see Using Queries), and provide similar performance. To run a query in an LCB application, pass an instance of **LC_Database** as a database argument to the **d_query** constructor. For example:

```
LC_Database *db;
d_query q(db);
```

LCB queries do have some limitations compared to the regular binding:

- Queries cannot reference or return stream-type properties.
- Only ad-hoc queries are currently supported, not pre-defined named queries and stored procedures.
- For authentication and security under UNIX®, users running LCB applications must belong to the cacheusr group, or be running a trusted application (see Running Trusted Applications on UNIX®).

5.7 Using LCB Batch Insert

The **LC_Batch** class provides methods for batch insertion using the Light C++ Binding. The **<< operator** is used to serialize objects and add them to a batch. When a batch is saved with the **flush()** method, there is a separate implicit transaction for each object, and the transaction is rolled back if there is any error. The **get_errors()** method returns a list of failed transactions.

Performance is only slightly better than with **save()**, but a single projection object can be serialized repeatedly with different property values, which may significantly reduce the processing overhead compared to calling **create_new()** for each insert.

- Use the **<< operator** to add objects to a batch.

  Once a projection object has been created with **create_new()** and its properties have been set, it can be serialized into a batch using the **<< operator**. The same projection object can be reused, or different projection objects can be used for each serialization in the batch. In the former case, any properties which are intended to have the same value in every object in the batch only need to be set once.

- Use **clear()** to remove an object from the batch.

  If an application does not wish a batch to be saved, after objects have already been serialized into it, the application should call **clear()**, which resets the batch's number of objects to 0 (but does not reset the error or id lists).

- Use **flush()** to save the batch.

  Once all objects to be inserted have been serialized into the batch, the batch is saved by calling its **flush()** method either directly or indirectly. The **close()** method calls **flush()**, and the LC_Batch destructor calls **close()**.

- After **flush()** is called, the **get_errors()** method can return an error list.
  - Each list entry is pairing of error status and object serialization.
  - If there were no errors, **size()** of the list is 0.
  - The **set_from_err_list()** method can be used to examine properties of the objects that had errors.

Transaction Handling

If the `_do_tx` parameter of the **LC_Batch() constructor** is set to true, there is a separate implicit transaction for each object when a batch is saved. The implicit transaction is committed if the object is saved successfully, or rolled back if there is any error. If the entire batch should be either committed or rolled back, the call to **flush()** should be bracketed with calls to `LC_Database` methods **tstart()** and **tcommit()** (see Using Transactions). This will cause the entire batch insert to be rolled back if an error is encountered.

Optimization

- Compile classes with optimization level `o2`.

  In Visual Studio, go to the `Tools Menu > Options > Compile` tab and check the `Optimize within class and calls to library classes` checkbox. This will improve performance 5 to 10 percent.

- When creating a new **LC_Batch** object, set the `reserve_size` parameter appropriately.

  Performance is enhanced by reserving as many bytes as will actually be needed for the batch's serialization buffer. This will avoid the cost of enlarging the buffer and copying data as objects are added to the batch. Specify `reserve_size` as at least equal to the average object size multiplied by the number of objects to be inserted.

- Set the `do_tx` parameter to `false`.

  Performance is substantially faster if `do_tx` is set to `false` (the default). If it is set to `true`, a **tstart()** and a **tcommit()** or **trollback()** (see Using Transactions) is performed for each insert within the batch.

5.8 LCB and Concurrency

LCB supports the standard Caché concurrency model (see Object Concurrency in *Using Caché Objects*). Use the following constants to specify concurrency level:

- No locking at all:

```
#define LC_CONCURRENCY_NO_LOCKING 0
```

- No lock during create; exclusive lock during update:

```
#define LC_CONCURRENCY_ATOMIC 1
```

- Shared lock during create; exclusive lock during update:

```
#define LC_CONCURRENCY_SHARED 2
```

- Shared lock retained after **openid()**:

```
#define LC_CONCURRENCY_SHARED_RETAINED 3
```

- Exclusive lock retained after **openid()**:

```
#define LC_CONCURRENCY_EXCLUSIVE 4
#define LC_CONCURRENCY_DEFAULT LC_CONCURRENCY_ATOMIC
```

Specify concurrency level when opening an object. For example:

```
d_ref<User_Person> person =
   User_Person::openid(db, id, LC_CONCURRENCY_EXCLUSIVE);
```

Note: You cannot specify a non-default concurrency level when creating a new object (although you can subsequently call **openid()** to set the desired concurrency level). LCB concurrency always defaults to `LC_CONCURRENCY_ATOMIC`.

5.8.1 Update Semantics

Calling **save()** or **update()** sets all properties of a database object from the projection object, whether or not the C++ application has modified them.

An object is protected from modification by other applications if it was opened with concurrency level `SHARED_RETAINED` or `EXCLUSIVE`.

If the object was opened with a lower concurrency level, **save()** may overwrite properties set by someone else's intervening update, or re-create the object after an intervening delete. This will not corrupt the object or indices, because appropriate locks are taken. You can explicitly call **openid()** again with a higher concurrency level, to lock an object that wasn't previously locked. This always reloads the current property values from the database.

When updating an object that was opened with a concurrency level lower than SHARED-RETAINED, if index updating is enabled and an object's indexed properties have been modified, the object is locked and the old property values are reloaded from the database. This is necessary so that the old index entries can be deleted.

5.9 Optimization and Troubleshooting

For best performance:

- Avoid use of wchar_t strings if not needed.
- Avoid unnecessary indices
- For the initial load, save with `defer_indices = true`, then build indices at the end.
- Define properties as %Double rather than %Float when possible.

5.9.1 Workarounds for SUSE 12 Linux Build Problems

Light C++ Binding applications on SUSE 12 Linux systems require one of the following two simple workarounds to build without error, due to a change in ld (the linker) from previous versions of SUSE Linux:

- Option 1: Add `-lpthread` to the library specifications in the ld or g++ command line in the makefile used to build the application. For example, if using a makefile based on one of the Master.mak files installed with Light C++ Binding sample applications, change the line:

  ```
  CACHETLIB = -L$(CACHETPATH) -lcachet
  ```

  to

  ```
  CACHETLIB = -L$(CACHETPATH) -lcachet -lpthread
  ```

- Option 2: Set the environment variable *MULTITHREADED* to `1` in the environment in which make is invoked. In Bourne shell the syntax is:

```
export MULTITHREADED=1
```

Either option resolves the issue, or both options can be used together. The workarounds are made necessary because the linker now requires application makefiles to explicitly specify dependent libraries of other libraries with which they link, even if those other libraries specified the dependent libraries when they were linked. In this case, libcachet.so depends on libpthread.so, so starting with SUSE 12 Linux, both libraries must be explicitly specified.

5.9.2 Detecting “object not found” Errors

If **openid()** is called with the optional `Db_err*` parameter and no object with the specified id exists, the `Db_err` code is set to `-3`, and `msg` is set to `"object not found"`. The d_ref to which the result of **openid()** is assigned is set to `null`, which can be detected by calling its **is_null()** member function.

It is the caller's responsibility to either test whether the `Db_err` code is non-zero, or to test whether the d_ref is null, before dereferencing the d_ref. Dereferencing a null d_ref causes an exception to be thrown, with code `-2` and `msg "Can not dereference a null d_ref<> value"`. For example, assume the following code fragment is executed for id "2", and no object with id value 2 exists:

```
Db_err openerr;
   person = User_Person::openid(db, id, concurrency, timeout, &openerr);
if (openerr)
   std::cerr << openerr << ",\n source = " << openerr.get_src() << '\n';
else
   printf("Object was found\n");
if (person.is_null())
   std::wcout << L"Person with id " << id << L" doesn't exist\n";
// Go ahead and dereference the d_ref whether or not it is null
d_string name = person->get_name();
```

Since the object was not found, the output is:

```
Error: code = -3, msg = object not found,
 source = LC_Database::lc_openid_obj
Person with id 2 doesn't exist
Error: code = -2, msg = Can not dereference a null d_ref<> value,
 source = abs_d_ref::operator->()
```

5.9.3 Calling the LC_Database and d_connection Destructors

It is important to call the `LC_Database` and `d_connection` destructors, in order to cleanly disconnect the application from Caché, causing the license and other resources to be released. The `lcbdemo` sample application shows an example of this.

If `d_connection` or `LC_Database` instances are declared as local variables, their destructors will automatically be called when they go out of scope. But if they are allocated via new and assigned to pointers, they must be explicitly destroyed using C++ "delete".

5.9.4 Using lc_conn::connect

- *lc_conn::connect changes the calling application's working directory*

  `lc_conn::connect` has the side effect (by default) of changing the calling application's current working directory, because it uses ZN to change namespace to the namespace specified in the connect string.

  This behavior can be disabled via a system configuration option in the Management Portal: Configuration->Advanced, ObjectScript: SwitchOSDirectory. Set this to "true" to cause Caché to *not* switch the OS current working directory when changing the namespace (the name “SwitchOSDirectory” is counter-intuitive). This affects any use of ZN, not just via `lc_conn::connect`.

- *Avoid signal handling when using lc_conn::connect*

`lc_conn::connect` uses the Callin API `CacheStart()` function, which sets handlers for various signals. These handlers may conflict with signal handlers set by the calling application.

- *lc_conn::connect() does not set a SIGINT handler*

  No handler is set for `SIGINT` when `lc_conn::connect()` invokes `CacheStart()`. This permits a user application to set its own handler. However, the user's handler should not terminate execution unless it can ensure that all threads which have active LCB connections have terminated them (by explicitly or implicitly destroying the `d_connection` object for the connection, or by directly calling `CacheEnd()`).

6

Reference for Simple Datatype Classes

Caché uses a set of special classes for literal datatypes (containing simple data such as strings or numbers). See Data Types in *Using Caché Objects* for information about how datatype classes differ from standard object classes.

Every Caché data type is mapped to an appropriate C++ object, such as d_int or d_string. If a literal type instance is not null, it is possible to convert it to a standard C++ type: d_int can be converted to int, d_string to std::string or std::wstring, d_time, d_date, and d_timestamp to tm. The C++ object that represents a Caché datatype is determined via the CLIENT-DATATYPE keyword value of the datatype class.

All simple types have:

- Conversion operators that makes it possible to use them as C++ types. For example, d_int can be converted to int and d_double to double.
- A **value()** method (for use in templates).
- **make_null()** and **is_null()** methods.
- An overloaded "<<" operator for output streams.
- An overloaded "=" operator.

The following datatypes are supported:

- Numeric — d_bool, d_int, d_double, d_numeric, d_decimal, and d_currency.
- Binary — d_binary, d_longbinary, d_oid, d_status, d_string, and d_list.
- Wide Strings — d_wstring, d_id, d_longwstring, and d_longstring.
- Date and Time — d_date, d_timestamp, and d_timestamp.

6.1 Numeric Classes

These are simple numbers.

- d_bool — %Library.Boolean corresponds to CLIENTDATATYPE keyword INTEGER.
- d_int — %Library.Integer corresponds to keyword INT or LONG.
- d_double — %Library.Double corresponds to keyword DOUBLE.
- d_numeric — %Library.Numeric corresponds to keyword NUMERIC. d_numeric is a typedef of d_double.

- d_decimal — %Library.Decimal corresponds to keyword DECIMAL.
- d_currency — %Library.Currency corresponds to keyword CURRENCY.

6.1.1 Class InterSystems::d_int

A d_int can be converted to int and be assigned an int. It doesn't have other overloaded operators. The intended usage is to get the int value and assign a changed value back to the object if the object should be changed. For example,

```
d_int t = 2;
d_int q = int(t) + 2;
```

in many cases like this one the conversion is implicit, so the second line can be just

```
d_int q = t + 2;
```

but there are cases where it is necessary.

6.2 Binary Classes

These are classes containing variable-length binary data.

- d_binary — %Library.Binary corresponds to CLIENTDATATYPE keyword BINARY.
- d_longbinary —
- d_oid — A complete Object ID. corresponds to keyword OID. d_oid is a typedef of d_binary.
- d_status — %Library.Status corresponds to keyword STATUS.
- d_string — %Library.String corresponds to keyword VARCHAR or LONG VARCHAR.
- d_list — %Library.List corresponds to Caché `$list` structure.

6.2.1 Class InterSystems::d_binary

A d_binary holds binary data. d_oid is a typedef of d_binary that represents a complete Object ID.

Member list

- **d_binary constructors**
  - No parameters.

    ```
    d_binary();
    ```

  - From null terminated string

    ```
    d_binary(const char* cstr);
    ```

  - From std::string

    ```
    d_binary(const std::string& s);
    ```

  - From string of size sz, starting at cstr

    ```
    d_binary(const char* cstr, int sz);
    ```

- **std::string() operator** — Return the data as std::string

```
operator std::string() const;
```

- **Comparison operators** — Compare to another d_binary

```
bool operator==(const d_binary& t);
bool operator!=(const d_binary& t);
```

- **append_bin()** — Append binary data

```
void append_bin(const char* buf, byte_size_t size);
```

- **assign()** — Assign binary data

```
void assign(const char* buf, byte_size_t size);
```

- **get_buf()** — Get the address of the binary buffer

```
const char* get_buf() const;
```

- **get_size()** — Get the size of the binary buffer

```
long get_size() const;
```

6.2.2 Class InterSystems::d_status

A d_status encapsulates %Library.Status. It should be used only for interpreting a status from the server.

Member list

- **operator int()** — Convert to int with the value of the error code

```
operator int() const;
```

- **get_code()** — Get the error code (returns 0 if no error)

```
int get_code() const;
```

- **get_msg()** — Get the error message

```
const d_string& get_msg() const;
```

- **get_from_srv()** — Analyze the status on the server with potential translation of the message to language lang (if it's a system error)

```
void get_from_srv(Database* db, const char* lang = "", Db_err* err = 0);
```

- **throw_err()** — Throw a Db_err with the code and the message of the error

```
void throw_err() const;
```

6.2.3 Class InterSystems::d_string

A d_string holds string data. It differs from d_binary in that it automatically converts data when necessary and also provides conversion methods.

Member list

- **d_string constructors**

  - No parameters.

```
d_string();
```

  - From null terminated string or wide null terminated string

```
d_string(const char* cstr);
d_string(const wchar_t* cstr);
```

  - From std::string or std::wstring

```
d_string(const std::string& s);
d_string(const std::wstring& s);
```

  - From string or wide string of size sz, starting at cstr

```
d_string(const char* cstr, int sz);
d_string(const wchar_t* cstr, int sz);
```

- **is_unicode()** — Test whether the string is in unicode format

```
bool is_unicode() const;
```

- **to_mb()** — Convert to multibyte.
  - in buffer buf of capacity cap, return the number of bytes put in buf.

```
byte_size_t to_mb(char* buf, char_size_t cap) const;
```

  - Convert to multibyte in place

```
void to_mb();
```

- **to_uni()** — Convert to unicode.
  - Store the result in buffer buf of capacity cap, return the number of characters put in buf

```
char_size_t to_uni(wchar_t* buf, char_size_t cap) const;
```

  - Convert to unicode in place

```
void to_uni();
```

- **std::string() operator** — Convert to std::string or std::wstring.

```
operator std::string() const;
operator std::wstring() const;
```

- **Comparison operators** — Compare to another d_string

```
bool operator==(const d_string& val) const;
bool operator!=(const d_string& val) const;
bool operator<(const d_string& val) const;
```

- **assign()**
  - From null terminated string or wide null terminated string.

```
void assign(const char* buf);
void assign(const wchar_t* buf);
```

  - From string or wide string of size sz, starting at cstr

```
void assign(const char* buf, char_size_t size);
void assign(const wchar_t* buf, char_size_t size);
```

6.2.4 Class InterSystems::d_list

A d_list object is a C++ implementation of the `$list` structure in Caché. In addition to its standard methods, the d_list class has a set of static methods that allow you to extract data from a buffer containing a `$list` without copying it into a d_list object.

6.2.4.1 d_list methods

A d_list object is essentially a forward iterator, but it also provides methods for inserting, deleting and replacing an element at the current position, as well as other methods that work with $list as a whole. A d_list position is 0 based. Since $list is stored in contiguous memory, any operation that changes a $list element may cause a dynamic memory reallocation or copying, which may be expensive.

Member list

- **d_list()**

```
d_list(const char* buf, byte_size_t size)
```

- **append_elem()**

```
void append_elem(__int64 val);
void append_elem(double val);
void append_elem(const d_string& val);
void append_elem(const d_binary& val);
void append_elem(const wchar_t* p, char_size_t size);
void append_elem(const char* p, char_size_t size);
```

- **append_elem_null()**

```
void append_elem_null();
```

- **at_end()**

```
bool at_end() const;
```

- **clear()** — Delete all elements

```
void clear();
```

- **count()** — Count the number of elements

```
int count();
```

- **del_elem()** — Delete the current element

```
void del_elem();
```

- **elem_null()**

```
void ins_elem_null();
```

- **get_elem()**

```
void get_elem(__int64* val) const;
void get_elem(double* val) const;
void get_elem(d_string& val) const;
void get_elem(d_binary& val) const;
void get_elem(bool* is_uni, const char** p_buf,
   byte_size_t* p_size) const;
```

- **get_elem_idx()** — Get the index of the current element

```
int get_elem_idx() const;
```

- **get_elem_type()**

```
char get_elem_type() const;
```

- **ins_elem()**

```
void ins_elem(__int64 val);
void ins_elem(double val);
void ins_elem(const d_string& val);
void ins_elem(const d_binary& val);
void ins_elem(const wchar_t* p, char_size_t size);
void ins_elem(const char* p, char_size_t size);
```

- **is_elem_double()**

```
bool is_elem_double() const;
```

- **is_elem_int()**

```
bool is_elem_int() const;
```

- **is_elem_null()**

```
bool is_elem_null() const;
```

- **is_elem_str()**

```
bool is_elem_str() const;
```

- **move_to()** — Change the current position to idx (0 based)

```
void move_to(int idx) const;
```

- **move_to_front()** — Same as move_to(0) but optimized

```
void move_to_front() const;
```

- **next()** — Similar to **move_to()**, but optimized for moving to the next element

```
void next() const;
```

- **reset()** — Reset the buffer

```
void reset(const char* buf, byte_size_t size);
```

- **set_elem()**

```
void set_elem(__int64 val);
void set_elem(double val);
void set_elem(const d_string& val);
void set_elem(const d_binary& val);
void set_elem(const wchar_t* p, char_size_t size);
void set_elem(const char* p, char_size_t size);
```

- **set_elem_null()**

```
void set_elem_null();
```

6.2.4.2 d_list static member functions

The static member functions allow you to extract data from a buffer that is a $list without copying it into a d_list object. The interface deals with the $list element specified by the buffer. The next element starts at buffer + d_list::get_elem_size(buffer).

Member list

- **get_elem()** — Get an element
  - Get an element as _int64, double, d_string, or d_binary.

```
static void get_elem(const char* buf, __int64* val);
static void get_elem(const char* buf, double* val);
static void get_elem(const char* buf, d_string& val);
static void get_elem(const char* buf, d_binary& val);
```

  - Get an element as a pointer to the string, the string size, and find whether it's unicode or narrow

```
static void get_elem(const char* buf, bool* is_uni,
   const char** p_buf, byte_size_t* p_size);
```

- **get_elem_size()** — Get element size

```
static byte_size_t get_elem_size(const char* buf);
```

- **is_elem_double()** — Test whether an element is stored as double

```
static bool is_elem_double(const char* buf);
```

- **is_elem_int()** — Test whether an element is stored as int

```
static bool is_elem_int(const char* buf);
```

- **is_elem_null()** — Test whether an element is null

```
static bool is_elem_null(const char* buf);
```

- **is_elem_str()** — Test whether an element is stored as string

```
static bool is_elem_str(const char* buf);
```

6.3 Time and Date Classes

Objects of these types can be converted to a tm structure object with all irrelevant values set to -1. They can also be assigned a tm object. The irrelevant values from the tm structure will be ignored. Interfaces of these classes differ only in constructors and assignment operators.

- d_date — %Library.Date corresponds to CLIENTDATATYPE keyword DATE.
- d_time — %Library.Time corresponds to keyword TIME.
- d_timestamp — %Library.TimeStamp corresponds to keyword TIMESTAMP.

6.3.1 Class InterSystems::d_time

Member list

- **d_time**
  - From tm

```
d_time(const tm& ts);
```

- From ODBC structure for time

```
d_time(const TIME_STRUCT& t);
```

- From hour, minute, second

```
d_time(int h, int m, int s);
```

- From ODBC structure for time

```
d_time& operator=(const TIME_STRUCT& t);
```

6.3.2 Class InterSystems::d_date

Member list

- **d_date**
  - From tm

```
d_date(const tm& ts);
```

  - From ODBC structure for date

```
d_date(const DATE_STRUCT& d);
```

  - From year, month, day

```
d_date(int y, int m, int d);
```

  - From ODBC structure for date

```
d_date& operator=(const DATE_STRUCT& d);
```

6.3.3 Class InterSystems::d_timestamp

Member list

- **d_timestamp**
  - From tm

```
d_timestamp(const tm& ts);
```

  - From ODBC structure for timestamp

```
d_timestamp(const TIMESTAMP_STRUCT& ts);
```

7

Reference for Object Datatype Classes

This chapter describes a set of predefined proxy classes that correspond to Caché object datatype classes such as lists, arrays, and streams. All of these proxy classes inherit from both Dyn_obj and Obj_t classes. All of them have the standard **open()**, **create_new()**, **openid()**, and **openref()** methods.

Collection classes:

- d_vector<S> — list collections
- d_map<S> — array collections

Stream classes:

- d_char_stream
- d_bin_stream
- d_file_bin_stream
- d_file_char_stream

Relationships:

- d_relationship<T>

7.1 Collection Classes

Caché supports two kinds of collections: lists and arrays. These are two different kinds of groupings of elements of a single type:

- d_vector list collections — correspond to the Caché %ListOfObjects and %ListOfDataTypes classes.
- d_map array collections — correspond to the Caché %ArrayOfObjects and %ArrayOfDataTypes classes.

Operations on a C++ client usually assume the collection's prior existence.

7.1.1 Class Template d_vector<S> (List Collections)

Proxies for %ListOfObjects and %ListOfDataTypes provide an interface which is almost identical to the interface of std::vector.

Because Caché list objects are generated as d_obj_vector<T> and d_prim_vector<T> classes, they provide the same interface that is specified by d_vector.

List Operations, Stack Operations, and Element Access

- **erase()** — Deletes the element at position pos.

```
iterator erase(iterator pos);
```

- **insert()** — Inserts at position pos an element of value val.

```
iterator insert(iterator pos, const value_type& val);
```

- **pop_back()** — Removes the final element of the list, which must be non-empty.

```
void pop_back();
```

- **push_back()** — Inserts an element of the value val at the end of a list.

```
void push_back(const value_type& val);
```

- **[] operator** — Supports unchecked element access by overloading the "[]" operator:

```
reference operator[](size_type index);
const;_reference operator[](size_type index) const;
```

- **at()** — Provides checked element access. Returns a reference to the list element at position index. If index is not a valid position, the method throws an out_of_range error.

```
reference at(size_type index);
const;_reference at(size_type index) const;
```

Size and Capacity

- **capacity()** — Returns the storage currently allocated for the list.

```
size_type capacity() const;
```

- **empty()** — Checks if the list is empty and returns true if it is.

```
bool empty() const;
```

- **max_size()** — Returns the maximum allowable length of the list.

```
size_type max_size() const;
```

- **reserve()** — Allocates space for a total number of n elements. This method only allocates memory for the n elements, but it does not create them.

```
void reserve(size_type n);
```

- **size()** — Returns the length of the list.

```
size_type size();
```

Iterators

- **begin()** — Returns a random-access iterator pointing to the list's first element.

```
iterator begin();
```

- **end()** — Returns a random-access iterator pointing to the one-past-last element of the array.

```
iterator end();
```

- **rbegin()** — Returns a reverse random-access iterator pointing to the beginning of the list's reverse sequence (just beyond the list's last element).

```
reverse_iterator rbegin();
```

- **rend()** — Returns a reverse random-access iterator pointing to the end of the list's reverse sequence (just before the list's first element).

```
reverse_iterator rend();
```

7.1.2 Class Template d_map<S> (Array Collections)

Proxies for %ArrayOfObjects and %ArrayOfDataTypes provide an interface which is almost identical to the interface of std::map.

Because Caché array objects are generated as d_obj_map<T> and d_prim_map<T> classes, they provide the same interface that is specified by d_map.

List Operations and Element Access

- **erase()** — Removes an element
  - Removes the array element specified by pos.

```
iterator erase(iterator pos);
```

  - Removes the element uniquely identified by the key k (if present).

```
size_type erase(const key_type& k);
```

- **insert()** — Inserts an element.
  - Inserts an element of value val, using pos as a hint

```
iterator insert(iterator pos, const value_type& val);
```

  - Inserts an element of value val.

```
std::pair<iterator, bool> insert(const value_type& val);
```

- **[] operator** — tbd

```
mapped_type& operator[](const key_type& key);
const mapped_type& operator[](const key_type& key) const;
```

Size and Capacity

- **capacity()** — Returns the storage currently allocated for the array.

```
size_type capacity() const;
```

- **empty()** — Returns true if the array is empty.

```
bool empty() const;
```

- **max_size()** — Returns the maximum number of elements that the array can contain.

```
size_type max_size() const;
```

- **size()** — Returns the number of elements in the array.

```
size_type size();
```

Iterators

- **begin()** — Returns a bi-directional iterator pointing to the array's first element.

  ```
  iterator begin();
  ```

- **end()** — Returns a bi-directional iterator to the one-past-last element of the array.

  ```
  iterator end();
  ```

- **rbegin()** — Returns a reverse iterator pointing to the beginning of the array's reverse sequence (just beyond the array's last element).

  ```
  reverse_iterator rbegin();
  ```

- **rend()** — Returns a reverse iterator pointing to the end of the array's reverse sequence (just before the array's first element).

  ```
  reverse_iterator rend();
  ```

- **find()** — Returns a bi-directional iterator designating the element in the array whose sort key has the equivalent ordering to key.

  ```
  iterator find(const key_type& key);
  ```

7.2 Streams

Proxies for Caché streams use adapters that fit them into the standard C++ library streams framework and optimize their performance. There are also a set of proxy classes for streams that inherit their common interface from the d_stream class. The adapters are the recommended way of working with streams. The adapters make the streams buffered, so avoid mixing calls to adapters and proxy objects that change the stream read/write position (as a result of reading or writing to a stream or a direct change in position).

Stream Objects

The following table describes the mapping of Caché stream classes:

Caché Class	C++ Class
%Library.GlobalCharacterStream	d_char_stream
%Library.GlobalBinaryStream	d_bin_stream
%Library.FileBinaryStream	d_file_bin_stream
%Library.FileCharacterStream	d_file_char_stream

All stream classes have static **open()** and **create_new()** methods. The d_file_char_stream class has an **is_unicode()** method, which checks if the stream contains Unicode data.

C++ Stream Adapters

The stream adapters can be used exactly as streams from the C++ standard library, and all the unique to Caché methods that are common to all Caché streams can be also accessed from them.

The adapter classes are:

- d_basic_istream with typedefs for d_istream and d_wistream
- d_basic_ostream with typedefs for d_ostream and d_wostream
- d_basic_iostream with typedefs for d_iostream and d_wiostream

All C++ adapter objects can be constructed from a d_ref to a stream object proxy. For example:

```
// create a low level stream object
d_ref<d_char_stream> stream = d_char_stream::create_new(&db);
// create an IOStreams extension stream object
d_iostream io(stream);
```

All adapters have helper methods that allow you to work with a stream only via its adapter. All C++ adapter objects can be constructed from a d_ref to a stream object proxy. For example:

```
// create a low level stream object
d_ref<d_char_stream> stream = d_char_stream::create_new(&db);
// create an IOStreams extension stream object
d_iostream io(stream);
```

7.2.1 Stream Adapter Classes

d_basic_ostream

There are typedefs d_ostream and d_wostream. In addition to std::basic_ostream interface, the class provides the following methods:

```
d_binary oid();
long size();
d_status erase();
d_status save();
```

d_basic_istream

There are typedefs d_istream and d_wistream. In addition to std::basic_istream interface, the class provides the following methods:

```
d_binary oid();
long size();
d_status rewind();
```

d_basic_iostream

There are typedefs d_iostream and d_wiostream. In addition to std::basic_iostream interface, the class provides the following methods:

```
d_binary oid();
long size();
d_status rewind();
d_status move_to_end();
d_status erase();
d_status save();
```

7.2.2 Class d_stream

The d_stream class provides the common interface for all streams. The d_file_stream class adds to it the common interface for all file streams.

d_stream Methods

The d_stream methods in common between character and binary streams are:

```
d_binary oid();
d_status save();
d_status clear();
d_status rewind();
d_status move_to_end();
long size();
d_stream& copy(const abs_d_ref& stream);
```

Methods specific to character streams are:

```
void read(d_int& len, d_string& res);
void readline(d_int& len, d_string& res);
void write(const d_string& data);
```

Methods specific to binary streams are:

```
void read(d_int& len, d_binary& res);
void write(const d_binary& data);
```

d_file_stream Methods

The additionally available methods from the d_file_stream class are:

```
d_string get_filename();
void set_filename(const d_string& fname);
d_timestamp last_modified();
d_status link_to_file(const_name_t fname);
// const_name_t is a typedef for const wchar_t*
```

7.3 Class Template d_relationship<S>

As in Caché, relationships are treated as properties. If there is a relationship between classes P and Q where P is the single-valued side and Q is the multi-valued side, then the single-valued side is generated as a property of type P (d_ref<P>), and the multi-valued side is generated as a property of type d_relationship<Q> (d_ref<d_relationship<Q>>). As with other properties, when P or Q can be determined only at runtime, P or Q (or both) become Dyn_obj (a dynamic object).

d_relationship Methods

The d_relationship<P> class is a standard container that supports the following methods:

- **begin()** — Returns a bi-directional iterator.

  ```
  iterator begin();
  ```

- **end()** — Returns a bi-directional iterator.

  ```
  iterator end();
  ```

- **rbegin()** — Returns a reverse iterator.

  ```
  reverse_iterator rbegin();
  ```

- **rend()** — Returns a reverse iterator.

  ```
  reverse_iterator rend();
  ```

8

Reference for Connectivity and Inherited Proxy Classes

This chapter describes the classes that are most important for an understanding of how proxy classes interact with the Caché database.

- Proxy base classes — **Persistent_t**, **Registered_t**, **LC_Persistent_t**, and **LC_Serial_t** are base classes for generated proxy classes.
- Database classes — **Database** and **LC_Database**
- Connection classes — **Conn_t** (**d_connection**), **tcp_conn**, and **lc_conn**
- Object reference class templates — **d_ref<T>** and **lc_d_ref<T>**

8.1 Proxy Base Classes

The following classes are available:

- **Persistent_t** — base class used to generate most persistent proxy classes.
- **Registered_t** — base class used to generate all serial proxy classes.
- **LC_Persistent_t** — base class used to generate persistent projection classes for the Light C++ Binding.
- **LC_Serial_t** — base class used to generate serial projection classes for the Light C++ Binding.

8.1.1 Class InterSystems::Persistent_t

Base class used to generate persistent proxy classes. Inherits from **Registered_t**.

8.1.1.1 Constructor

Persistent_t()

Constructor is a PROTECTED member function (see Standard Proxy Class Methods).

```
InterSystems::Persistent_t::Persistent_t
( ) [inline, protected]
```

8.1.1.2 Member list

_delete()

```
d_status Persistent_t::_delete
(  Database * db,
   const d_binary & oid,
   int conc = -1
) [static]
```

_is_null()

```
bool InterSystems::Obj_t::_is_null
(  ) const [inline, inherited]
```

create_new()

```
static d_ref<Registered_t> InterSystems::Registered_t::create_new
(  Database * db,
   const_str_t init_val = 0,   // const_str_t is a typedef of const wchar_t*
   Db_err * err = 0
) [inline, static, inherited]
```

delete_id()

```
d_status Persistent_t::delete_id
(  Database * db,
   const_name_t cl_name,    // const_name_t is a typedef for const wchar_t*
   const_str_t id,          // const_str_t is a typedef of const wchar_t*
   int conc = -1
) [static]
```

downgrade_concurrency()

```
d_status Persistent_t::downgrade_concurrency
(  int conc )
```

exists_id()

```
bool Persistent_t::exists_id
(  Database * db,
   const_name_t cl_name,   // const_name_t is a typedef for const wchar_t*
   const_str_t id          // const_str_t is a typedef of const wchar_t*
) [static]
```

get_cl_name()

```
const wchar_t* InterSystems::Obj_t::get_cl_name
(  ) const [inline, inherited]
```

get_db()

```
Database* InterSystems::Obj_t::get_db
(  ) const [inline, inherited]
```

get_id()

```
d_string Persistent_t::get_id
(  ) const
```

get_ref()

```
int InterSystems::Obj_t::get_ref
(  ) const [inline, inherited]
```

get_val()

```
const Oref_t& InterSystems::Obj_t::get_val
(  ) const [inline, inherited]
```

id()

```
d_string Persistent_t::id
(  ) const
```

oid()

```
d_oid InterSystems::Persistent_t::oid
(  ) const [inline]
```

openref()

```
static d_ref<Registered_t> InterSystems::Registered_t::openref
(  Database * db,
   int oref,
   const_name_t cl_name    // const_name_t is a typedef for const wchar_t*
) [inline, static, inherited]

static d_ref<Registered_t> InterSystems::Registered_t::openref
(  t_istream & in,
   Database * db
) [inline, static, inherited]
```

reload()

```
d_status Persistent_t::reload
(  )
```

save()

```
d_status Persistent_t::save
(  int related = 1 ) const
```

to_xml()

```
void InterSystems::Obj_t::to_xml
(  xml_writer & out ) [inline, inherited]
```

upgrade_concurrency()

```
d_status Persistent_t::upgrade_concurrency
(  int conc )
```

8.1.2 Class InterSystems::Registered_t

Base class used to generate all serial proxy classes.

8.1.2.1 Constructor

Registered_t()

Constructor is a PROTECTED member function (see Standard Proxy Class Methods).

```
InterSystems::Registered_t::Registered_t
( ) [inline, protected]
```

8.1.2.2 Member list

_is_null()

```
bool InterSystems::Obj_t::_is_null
(  ) const [inline, inherited]
```

get_cl_name()

```
const wchar_t* InterSystems::Obj_t::get_cl_name
(  ) const [inline, inherited]
```

create_new()

```
static d_ref&lt;Registered_t> InterSystems::Registered_t::create_new
(  Database * db,
   const_str_t init_val = 0,    // const_str_t is a typedef of const wchar_t*
   Db_err * err = 0
) [inline, static]
```

get_db()

```
Database* InterSystems::Obj_t::get_db
(  ) const [inline, inherited]
```

get_ref()

```
int InterSystems::Obj_t::get_ref
(  ) const [inline, inherited]
```

get_val()

```
const Oref_t& InterSystems::Obj_t::get_val
(  ) const [inline, inherited]
```

openref()

```
static d_ref&lt;Registered_t> InterSystems::Registered_t::openref
(  Database * db,
   int oref,
   const_name_t cl_name    // const_name_t is a typedef for const wchar_t*
) [inline, static]

static d_ref&lt;Registered_t> InterSystems::Registered_t::openref
(  t_istream & in,
   Database * db
) [inline, static]
```

to_xml()

```
void InterSystems::Obj_t::to_xml
(  xml_writer & out ) [inline, inherited]
```

8.1.3 Class InterSystems::LC_Persistent_t

This is the base class used to generate persistent classes for the Light C++ Binding. This class can only be used in Light C++ Binding applications.

8.1.3.1 Constructor

LC_Persistent_t

Both constructors are PROTECTED member functions (see Standard Proxy Class Methods).

```
LC_Persistent_t::LC_Persistent_t
( ) [inline, protected]

LC_Persistent_t::LC_Persistent_t
(  Database * db,
   int  oref,
   const wchar_t * cl_name
) [inline, protected]
```

8.1.3.2 Member list

_is_null()

Do not use (reserved for InterSystems internal use).

connect()

```
void LC_Persistent_t::connect
(  LC_Database * db )
```

detach()

```
void LC_Persistent_t::detach ( )
```

disconnect()

```
void LC_Persistent_t::disconnect ( )
```

get_cl_name()

Do not use (reserved for InterSystems internal use).

get_classname()

```
virtual const unsigned char* LC_Persistent_t::get_classname
(  ) const [pure virtual]
```

get_classname_length()

```
virtual int LC_Persistent_t::get_classname_length
(  ) const [pure virtual]
```

get_db()

```
Database* InterSystems::Obj_t::get_db
(  ) const [inline, inherited]
```

get_ref()

Do not use (reserved for InterSystems internal use).

get_val()

Do not use (reserved for InterSystems internal use).

has_idkey()

```
virtual DLL_DECL bool LC_Persistent_t::has_idkey
(  ) [inline, virtual]
```

id()

```
const wchar_t * LC_Persistent_t::id
(  ) const [inline]

int LC_Persistent_t::id
(  char * buf,
   size_t bufsiz
) [inline]

int LC_Persistent_t::id
(  wchar_t * buf,
   size_t bufsiz
) [inline]
```

id_is_uni()

```
bool LC_Persistent_t::id_is_uni
(  ) const [inline]
```

insert()

```
d_status LC_Persistent_t::insert
(  bool defer_indices = false,
   int timeout = -1,
   Db_err * err = 0 )
```

is_attached()

```
bool LC_Persistent_t::is_attached
(  ) [inline]
```

is_connected()

```
bool LC_Persistent_t::is_connected
(  ) [inline]
```

save()

```
virtual DLL_DECL d_status LC_Persistent_t::save
(  bool defer_indices = false,
   int timeout = -1,
   Db_err * err = 0
) [pure virtual]
```

serialize()

```
virtual DLL_DECL void LC_Persistent_t::serialize
(  lc_dlist_out * ,
   LC_Database *
) [inline, virtual]
```

serialize_idkey()

```
virtual DLL_DECL void LC_Persistent_t::serialize_idkey
(  lc_dlist_out * ,
   LC_Database *
) [inline, virtual]
```

set_id_from_properties()

```
virtual DLL_DECL void LC_Persistent_t::set_id_from_properties
(  ) [inline, virtual]
```

to_xml()

Do not use (reserved for InterSystems internal use).

unlock()

```
virtual void LC_Persistent_t::unlock
(  ) [pure virtual]
```

update()

```
d_status LC_Persistent_t::update
(  bool defer_indices = false,
   int timeout = -1,
   Db_err * err = 0 )
```

8.1.4 Class InterSystems::LC_Serial_t

This is the base class used to generate serial classes for the Light C++ Binding. This class can only be used in Light C++ Binding applications.

8.1.4.1 Constructor

LC_Serial_t()

Constructor is a PROTECTED member function (see Standard Proxy Class Methods).

```
InterSystems::LC_Serial_t::LC_Serial_t
( ) [inline, protected]
```

8.1.4.2 Member list

dirty()

```
bool LC_Serial_t::dirty
(  ) const [inline]
```

getProperties()

```
virtual void  LC_Serial_t::getProperties
(  lc_dlist_in &in,
   bool unicode_srv
)=0 [pure virtual]
```

serialize()

```
virtual void LC_Serial_t::serialize
(  lc_dlist_out *outlist,
   LC_Database *db
)=0 [pure virtual]
```

set_clean()

```
void LC_Serial_t::set_clean
(  ) [inline]
```

set_dirty()

```
void LC_Serial_t::set_dirty
(  ) [inline]
```

8.2 Database Classes

The following database classes are available:

- **Database** — used by the standard Caché C++ binding.
- **LC_Database** — used only in Light C++ Binding applications.

8.2.1 InterSystems::Database Class

This is the database class used by the standard Caché C++ binding.

8.2.1.1 Constructor

Database()

```
Database::Database
(  const d_connection &  conn,
   bool use_cache = true,
   bool is_lc_db = false )
```

8.2.1.2 Member list

del_obj()

```
void InterSystems::Database::del_obj
(  const d_binary & oid,
   int concurrency
) [inline]
```

get_class_global_info()

```
t_istream & Database::get_class_global_info
(  const_name_t cl_name,   // const_name_t is a typedef for const wchar_t*
   cl_meta_info_kind info_kind )
```

get_class_info()

```
t_istream & Database::get_class_info
(  const_name_t cl_name,   // const_name_t is a typedef for const wchar_t*
   cl_meta_info_kind info_kind )
```

get_classes_info()

```
t_istream & Database::get_classes_info
(  const char * msg,
   const std::vector&lt; std::wstring > & cl_names )
```

get_coln_property()

```
void Database::get_coln_property
(  int oref,
   const_name_t prop_name,   // const_name_t is a typedef for const wchar_t*
   int ii,
   int idx,
   d_double & res )

void Database::get_coln_property
(  int oref,
   const_name_t prop_name,
   int ii,
   int idx,
   d_int & res )

void Database::get_coln_property
(  int oref,
   const_name_t prop_name,
   int ii,
   int idx,
   d_string & val )
```

get_conn()

```
d_connection InterSystems::Database::get_conn
( ) [inline]
```

get_hdbc()

```
HDBC InterSystems::Database::get_hdbc
( ) [inline]
```

get_indexes_info()

```
t_istream & Database::get_indexes_info
(  const_name_t cl_name,   // const_name_t is a typedef for const wchar_t*
   cl_meta_info_kind info_kind )
```

get_job_id()

```
int InterSystems::Database::get_job_id
( ) [inline]
```

get_lc_class_info()

```
t_istream & Database::get_lc_class_info
(  const_name_t cl_name,   // const_name_t is a typedef for const wchar_t*
   cl_meta_info_kind info_kind )
```

get_methods()

```
t_istream & Database::get_methods
(  const_name_t class_name )   // const_name_t is a typedef for const wchar_t*
```

get_nsp()

```
const d_string& InterSystems::Database::get_nsp
( ) const [inline]
```

get_oid()

```
d_oid Database::get_oid
(  int oref )
```

get_properties()

```
t_istream & Database::get_properties
(  const_name_t class_name )   // const_name_t is a typedef for const wchar_t*
```

get_property()

```
t_istream & Database::get_property
(  int oref,
   int ii,
   int jj,
   d_type_id type_id,
   const_name_t name )   // const_name_t is a typedef for const wchar_t*

void Database::get_property
(  int oref,
   const_name_t prop_name,
   int ii,
   int jj,
   Args_mgr & args_mgr )
```

get_proxies_info()

```
Proxies_info* InterSystems::Database::get_proxies_info
( ) [inline]
```

get_queries()

```
t_istream & Database::get_queries
(  const_name_t class_name )   // const_name_t is a typedef for const wchar_t*
```

get_query_info()

```
t_istream & Database::get_query_info
(  const wchar_t * class_name,
   const wchar_t * query_name )
```

get_run_mtd_level()

```
int InterSystems::Database::get_run_mtd_level
( ) const [inline]
```

get_serialization_info()

```
t_istream & Database::get_serialization_info
(  const_name_t cl_name,   // const_name_t is a typedef for const wchar_t*
   cl_meta_info_kind info_kind )
```

get_srv_ver()

```
double InterSystems::Database::get_srv_ver
( ) const [inline]
```

get_status_info()

```
void Database::get_status_info
(  const d_status & status,
   d_int & code,
   d_string & msg,
   const char * lang = "",
   Db_err * err = 0 )
```

get_term_input_callback()

```
db_input_callback* InterSystems::Database::get_term_input_callback
( ) [inline]
```

get_term_output()

```
db_output& InterSystems::Database::get_term_output
( ) [inline]
```

init_coln()

```
template<typename C>
void InterSystems::Database::init_coln
(  int oref,
   C & coln
) [inline]
```

make_obj()

```
t_istream & Database::make_obj
(  const_name_t type_name,      // const_name_t is a typedef for const wchar_t*
   const_str_t init_val = 0,    // const_str_t is a typedef of const wchar_t*
   Db_err * err = 0 )
```

open_cl_def()

```
const Class_def* InterSystems::Database::open_cl_def
(  const_name_t class_name,   // const_name_t is a typedef for const wchar_t*
   bool check_exists = false
) [inline]
```

openid_obj()

```
t_istream & Database::open_obj
(  const d_binary & oid,
   int concurrency = -1,
   int timeout = -1,
   Db_err * err = 0 )

t_istream & Database::openid_obj
(  const_name_t name,      // const_name_t is a typedef for const wchar_t*
   const_str_t id,         // const_str_t is a typedef of const wchar_t*
   int concurrency = -1,
   int timeout = -1,
   Db_err * err = 0 )
```

reset_term_input_callback()

```
void InterSystems::Database::reset_term_input_callback
( ) [inline]
```

reset_term_output_callback()

```
void InterSystems::Database::reset_term_output_callback
( ) [inline]
```

run_method()

```
t_istream & Database::run_method
(  int obj_ref,
   const_name_t cl_name,   // const_name_t is a typedef for const wchar_t*
   const_name_t mtd_name,
   D_type ** args,
   int num_args,
   const int * refs,
   int num_refs,
   d_type_id ret_t )

void Database::run_method
(  int oref,
   const_name_t cl_name,
   const_name_t mtd_name,
   Args_mgr & args_mgr )
```

set_property()

```
void Database::set_property
(  int oref,
   const_name_t prop_name,   // const_name_t is a typedef for const wchar_t*
   int ii,
   int jj,
   int mod_flag,
   Args_mgr & args_mgr )

void Database::set_property
(  int oref,
   int ii,
   int jj,
   int mod_flag,
   const_name_t name,   // const_name_t is a typedef for const wchar_t*
   D_type * val )
```

set_term_input_callback()

```
void InterSystems::Database::set_term_input_callback
(  db_input_callback * f ) [inline]
```

set_term_output_callback()

```
void InterSystems::Database::set_term_output_callback
(  db_output_callback * f ) [inline]
```

sync()

```
void Database::sync ( )
```

tcommit()

See Using Transactions for an example.

```
void InterSystems::Database::tcommit
(  Db_err * err = 0 ) [inline]
```

tlevel()

See Using Transactions for an example.

```
int InterSystems::Database::tlevel
(  Db_err * err = 0 ) [inline]
```

trollback()

See Using Transactions for an example.

```
void InterSystems::Database::trollback
(  Db_err * err = 0 ) [inline]
```

tstart()

See Using Transactions for an example.

```
void InterSystems::Database::tstart
(  Db_err * err = 0 ) [inline]
```

unicode_srv()

```
bool InterSystems::Database::unicode_srv
( ) const [inline]
```

8.2.2 InterSystems::LC_Database Class

This is the database class used by the Light C++ Binding. This class can only be used in Light C++ Binding applications.

8.2.2.1 Constructor

LC_Database()

```
InterSystems::LC_Database::LC_Database
(  const d_connection & conn,
   bool use_cache = true
) [inline]
```

8.2.2.2 Member list

add_key_prop()

Do not use (reserved for InterSystems internal use).

build_indexes()

```
void LC_Database::build_indexes
(  const unsigned char * classname,
   int classname_length )
```

check_thread()

Do not use (reserved for InterSystems internal use).

create_index_entry()

Do not use (reserved for InterSystems internal use).

cvtForCollation()

Do not use (reserved for InterSystems internal use).

del_obj()

Do not use (reserved for InterSystems internal use).

delete_direct()

Do not use (reserved for InterSystems internal use).

delete_index_entry()

Do not use (reserved for InterSystems internal use).

delete_object()

Do not use (reserved for InterSystems internal use).

get_coln_property()

Do not use (reserved for InterSystems internal use).

get_conn()

```
d_connection InterSystems::Database::get_conn
( ) [inline, inherited]
```

get_default_concurrency_level()

```
int InterSystems::LC_Database::get_default_concurrency_level
( ) [inline]
```

get_default_max_locks()

```
int InterSystems::LC_Database::get_default_max_locks
( ) [inline]
```

get_default_timeout()

```
int InterSystems::LC_Database::get_default_timeout
( ) [inline]
```

get_hdbc()

Do not use (reserved for InterSystems internal use).

get_idkey_out_list()

Do not use (reserved for InterSystems internal use).

get_in_list()

Do not use (reserved for InterSystems internal use).

get_indexes_info()

Do not use (reserved for InterSystems internal use).

get_job_id()

Do not use (reserved for InterSystems internal use).

get_lc_class_info()

Do not use (reserved for InterSystems internal use).

get_methods()

Do not use (reserved for InterSystems internal use).

get_nsp()

Do not use (reserved for InterSystems internal use).

get_oid()

Do not use (reserved for InterSystems internal use).

get_out_list()

Do not use (reserved for InterSystems internal use).

get_properties()

Do not use (reserved for InterSystems internal use).

get_property()

Do not use (reserved for InterSystems internal use).

get_proxies_info()

Do not use (reserved for InterSystems internal use).

get_queries()

Do not use (reserved for InterSystems internal use).

get_query_info()

Do not use (reserved for InterSystems internal use).

get_run_mtd_level()

Do not use (reserved for InterSystems internal use).

get_serialization_info()

Do not use (reserved for InterSystems internal use).

get_srv_info()

Do not use (reserved for InterSystems internal use).

get_srv_ver()

Do not use (reserved for InterSystems internal use).

get_status_info()

Do not use (reserved for InterSystems internal use).

get_term_input_callback()

Do not use (reserved for InterSystems internal use).

get_term_output()

Do not use (reserved for InterSystems internal use).

init_coln()

Do not use (reserved for InterSystems internal use).

lc_batch_save()

```
void LC_Database::lc_batch_save
(  int num_objs,
   lc_nested_list_iterator & buf,
   std::vector&lt; std::pair&lt; d_status, d_binary > > & errors,
   std::vector&lt; d_string > & ids,
   const unsigned char * classname,
   int classname_length,
   int concurrency = -1,
   bool use_idkeys = false,
   bool return_ids = false,
   byte_size_t cap = 0,
   bool do_tx = true )
```

lc_openid_obj()

Do not use (reserved for InterSystems internal use).

make_obj()

Do not use (reserved for InterSystems internal use).

open_cl_def()

Do not use (reserved for InterSystems internal use).

open_obj()

Do not use (reserved for InterSystems internal use).

openid_obj()

Do not use (reserved for InterSystems internal use).

reset()

Do not use (reserved for InterSystems internal use).

reset_idkey_props()

Do not use (reserved for InterSystems internal use).

reset_term_input_callback()

Do not use (reserved for InterSystems internal use).

reset_term_output_callback()

Do not use (reserved for InterSystems internal use).

run_method()

Do not use (reserved for InterSystems internal use).

save()

Do not use (reserved for InterSystems internal use).

save_direct()

Do not confuse this with the **direct_save()** method used by Light C++ Binding projection classes (see Standard LCB Object Methods).

Do not use (reserved for InterSystems internal use).

set_key_props_id()

Do not use (reserved for InterSystems internal use).

set_key_props_index()

Do not use (reserved for InterSystems internal use).

set_lcb_option ()

Do not use (reserved for InterSystems internal use).

set_property()

Do not use (reserved for InterSystems internal use).

set_term_input_callback()

Do not use (reserved for InterSystems internal use).

set_term_output_callback()

Do not use (reserved for InterSystems internal use).

sync()

Do not use (reserved for InterSystems internal use).

tcommit()

See Using Transactions for an example.

```
void InterSystems::Database::tcommit
(  Db_err * err = 0) [inline, inherited]
```

time_to_string()

```
void LC_Database::time_to_string
(  d_time & in,
   d_string & out )
```

timestamp_to_string()

```
void LC_Database::timestamp_to_string
(  d_timestamp & in,
   d_string & out )
```

tlevel()

See Using Transactions for an example.

```
int InterSystems::Database::tlevel
(  Db_err * err = 0) [inline, inherited]
```

transaction()

See Using Transactions for an example.

```
void LC_Database::transaction
(  trans_flag_t flag,
   Db_err * err,
   int * level = 0
) [virtual]
```

trollback()

See Using Transactions for an example.

```
void InterSystems::Database::trollback
(  Db_err * err = 0) [inline, inherited]
```

tstart()

See Using Transactions for an example.

```
void InterSystems::Database::tstart
(  Db_err * err = 0) [inline, inherited]
```

unicode_srv()

```
bool InterSystems::Database::unicode_srv
( ) const [inline, inherited]
```

unlock()

Do not use (reserved for InterSystems internal use).

unlock_after_delete()

Do not use (reserved for InterSystems internal use).

8.3 Connection Classes

The following connection classes are available:

- **d_connection** — acts as a smart pointer to a Conn_t class instance.
- **Conn_t** — the base connection class.
- **tcp_conn** — connection class for the standard binding.
- **lc_conn** — connection class used only in Light C++ Binding applications.

8.3.1 Class d_connection

d_connection is a proxy class that acts as a smart pointer to a **Conn_t** class instance. See Connecting to the Caché Database for more information.

8.3.2 Class InterSystems::Conn_t

This is the base connection class. Always use **d_connection** rather than accessing this class directly.

8.3.2.1 Constructor

Conn_t()

```
InterSystems::Conn_t::Conn_t ( ) [inline]
```

8.3.2.2 Member list

alloc_messenger()

```
virtual int InterSystems::Conn_t::alloc_messenger
(  void ** ppm) [inline, virtual]
```

free_messenger()

```
virtual int InterSystems::Conn_t::free_messenger
(  void * pm) [inline, virtual]
```

get_thread_check()

```
virtual LC_Thread_Check* InterSystems::Conn_t::get_thread_check
( ) const [inline, virtual]
```

is_busy()

```
bool InterSystems::Conn_t::is_busy
( ) const [inline]
```

is_connected()

```
bool InterSystems::Conn_t::is_connected
( ) const [inline]
```

is_uni_srv()

```
bool InterSystems::is_uni_srv_info::is_uni_srv
( ) const [inline, inherited]
```

lock()

```
void InterSystems::Conn_t::lock
( ) [inline]
```

release_to_pool()

```
virtual void InterSystems::Conn_t::release_to_pool
( ) [inline, virtual]
```

set_uni_srv()

```
void InterSystems::is_uni_srv_info::set_uni_srv
(  bool val) [inline, inherited]
```

unlock()

```
void InterSystems::Conn_t::unlock
( ) [inline]
```

8.3.3 Class InterSystems::tcp_conn

This is the connection class for the standard C++ binding. It inherits from **Conn_t** and uses TCP/IP to implement the connection.

8.3.3.1 Constructor

tcp_conn()

```
InterSystems::tcp_conn::tcp_conn ( ) [inline]
```

8.3.3.2 Member list

alloc_messenger()

```
int tcp_conn::alloc_messenger
(  void ** ppm) [virtual]
```

connect()

```
d_connection tcp_conn::connect
(  const d_string & conn_str,
   const d_string & srv_principal_name,
   int security_level,
   int timeout = 0,
   Db_err * err = 0
) [static]

d_connection tcp_conn::connect
(  const d_string & conn_str,
   const d_string & user,
   const d_string & pwd,
   int timeout = 0,
   Db_err * err = 0
) [static]
```

free_messenger()

```
int tcp_conn::free_messenger
(  void * pm) [virtual]
```

get_connection()

```
d_connection tcp_conn::get_connection
(  const d_string & conn_str,
   const d_string & user,
   const d_string & pwd,
   int timeout = 0,
   Db_err * err = 0
) [static]
```

get_namespaces()

```
void tcp_conn::get_namespaces
(  const wchar_t * host,
   const wchar_t * port,
   const wchar_t * srv_principal_name,
   int security_level,
   int timeout,
   std::list&lt; std::wstring > & res,
   Conn_err * err = 0
) [static]

void tcp_conn::get_namespaces
(  const wchar_t * host,
   const wchar_t * port,
   const wchar_t * user,
   const wchar_t * pwd,
   int timeout,
   std::list&lt; std::wstring > & res,
   Conn_err * err = 0
) [static]
```

get_thread_check()

```
LC_Thread_Check* InterSystems::tcp_conn::get_thread_check
( ) const [inline, virtual]
```

is_busy()

```
bool InterSystems::Conn_t::is_busy
( ) const [inline, inherited]
```

is_connected()

```
bool InterSystems::Conn_t::is_connected
( ) const [inline, inherited]
```

is_uni_srv()

```
bool InterSystems::is_uni_srv_info::is_uni_srv
( ) const [inline, inherited]
```

lock()

```
void InterSystems::Conn_t::lock
( ) [inline, inherited]
```

parse_conn_str()

```
void Conn_t::parse_conn_str
(  const std::wstring & conn_str,
   std::wstring * host,
   std::wstring * port,
   std::wstring * nsp
) [static, protected, inherited]
```

set_uni_srv()

```
void InterSystems::is_uni_srv_info::set_uni_srv
(  bool val) [inline, inherited]
```

unlock()

```
void InterSystems::Conn_t::unlock
( ) [inline, inherited]
```

8.3.4 Class InterSystems::lc_conn

This is the connection class for the Light C++ Binding. It inherits from **Conn_t** and uses intraprocess communications to implement the connection. This class can only be used in Light C++ Binding applications.

8.3.4.1 Constructor

lc_conn()

```
lc_conn::lc_conn ( ) [inline]
```

8.3.4.2 Member list

alloc_messenger()

Do not use (reserved for InterSystems internal use).

connect()

```
static d_connection lc_conn::connect
(  const d_string & conn_str,
   const d_string & user,
   const d_string & pwd,
   int timeout = 0,
   Db_err * err = 0
) [inline, static]
```

disconnect()

```
void lc_conn::disconnect
( ) [virtual]
```

free_messenger()

Do not use (reserved for InterSystems internal use).

get_thread_check()

Do not use (reserved for InterSystems internal use).

is_busy()

Do not use (reserved for InterSystems internal use).

is_connected()

Do not use (reserved for InterSystems internal use).

is_uni_srv()

```
bool InterSystems::is_uni_srv_info::is_uni_srv
( ) const [inline, inherited]
```

lock()

Do not use (reserved for InterSystems internal use).

release_to_pool()

Do not use (reserved for InterSystems internal use).

set_uni_srv()

Do not use (reserved for InterSystems internal use).

unlock()

Do not use (reserved for InterSystems internal use).

8.4 Object Reference Classes

The following classes are available:

- **d_ref<T>** — reference class template used by the standard binding.
- **lc_d_ref<T>** — reference class template used only in Light C++ Binding applications.

8.4.1 Class Template InterSystems::d_ref<T>

See Generating Proxy Classes and Using Proxy Objects for more information about this class.

8.4.1.1 Constructor

d_ref()

```
template<class T>
InterSystems::d_ref< T >::d_ref
( ) [inline]
```

```
template<class T>
InterSystems::d_ref< T >::d_ref
(  T * p) [inline]

template<class T>
InterSystems::d_ref< T >::d_ref
(  bool dummy,
   T * p
) [inline]

template<class T>
InterSystems::d_ref< T >::d_ref
(  T * p,
   int * ref_cnt
) [inline]

template<class T>
InterSystems::d_ref< T >::d_ref
(  bool dummy,
   T * p,
   int * ref_cnt
) [inline]

template<class T>
template<typename P>
InterSystems::d_ref< T >::d_ref
(  const d_ref< P > & p) [inline]

template<class T>
InterSystems::d_ref< T >::d_ref
(  const d_ref< T > & r) [inline]
```

8.4.1.2 Member list

conv_to()

```
template<class T>
template<typename P>
void InterSystems::d_ref< T >::conv_to
(  d_ref< P > & res) [inline]
```

get()

```
template<typename T>
void InterSystems::d_ref< T >::get
(  t_istream & in, Database * db) [virtual]

void D_type::get
(  char * buf,
   byte_size_t size,
   Database * db = 0
) [inherited]
```

get_conv_ptr()

```
template<class T>
const T* InterSystems::d_ref< T >::get_conv_ptr
( ) const [inline]
```

get_cpp_type()

```
static SQLSMALLINT InterSystems::abs_d_ref::get_cpp_type
( ) [inline, static, inherited]
```

get_data()

```
void D_type::get_data
(  d_seq_query & query) [virtual, inherited]
```

get_ignore_null()

```
static bool InterSystems::D_type::get_ignore_null
( ) [inline, static, inherited]
```

get_is_lc_dref()

```
bool InterSystems::abs_d_ref::get_is_lc_dref
( ) const [inline, inherited]
```

get_oref_n_name()

```
void abs_d_ref::get_oref_n_name
(  t_istream & in,
   int * oref,
   cl_name_t name
) [static, inherited]
```

get_type_id()

```
d_type_id InterSystems::abs_d_ref::get_type_id
( ) const [inline, virtual, inherited]
```

is_null()

```
bool InterSystems::D_type::is_null
( ) const [inline, inherited]
```

is_obj()

```
bool InterSystems::D_type::is_obj
( ) const [inline, inherited]
```

make_not_null()

```
void InterSystems::D_type::make_not_null
( ) [inline, inherited]
```

make_null()

```
void InterSystems::abs_d_ref::make_null
( ) [inline, virtual, inherited]
```

make_undef()

```
void InterSystems::D_type::make_undef
( ) [inline, inherited]
```

operator !=

```
bool InterSystems::abs_d_ref::operator!=
(  const abs_d_ref & r) const [inline, inherited]
```

operator ->

```
template<class T>
T* InterSystems::d_ref< T >::operator->
( ) const [inline]
```

operator =

```
template<class T>
template<typename P>
d_ref& InterSystems::d_ref< T >::operator=
(  const d_ref< P > & p) [inline]
```

operator ==

```
bool InterSystems::abs_d_ref::operator==
(  const abs_d_ref & r) const [inline, inherited]
```

operator *

```
template<class T>
T& InterSystems::d_ref< T >::operator *
( ) const [inline]
```

operator <<

```
std::ostream& operator <<
(  std::ostream & out,
   const abs_d_ref & r
) [friend, inherited]
```

put()

```
byte_size_t D_type::put
( char * buf,
byte_size_t cap,
bool uni_srv,
Database * db = 0
) [inherited]

void InterSystems::abs_d_ref::put
(  t_ostream & out,
   Database * db = 0
) const [inline, virtual, inherited]
```

put_empty()

```
void InterSystems::D_type::put_empty
(  t_ostream & out) const [inline, inherited]
```

put_null()

```
void InterSystems::D_type::put_null
(  t_ostream & out) const [inline, inherited]
```

set_ignore_null()

```
static void InterSystems::D_type::set_ignore_null
(  bool val) [inline, static, inherited]
```

set_par()

```
void D_type::set_par
(  abs_d_query & query,
   int idx
) const [virtual, inherited]
```

to_xml()

```
void abs_d_ref::to_xml
(  xml_writer & out) const [virtual, inherited]
```

to_xml_null_value()

```
void InterSystems::D_type::to_xml_null_value
(  xml_writer & out) const [inline, inherited]
```

8.4.2 Class Template InterSystems::lc_d_ref<T>

This is the reference class template used by the Light C++ Binding. The information on d_ref in Generating Proxy Classes and Using Proxy Objects also applies to this class. This class can only be used in Light C++ Binding applications.

8.4.2.1 Constructor

lc_d_ref()

```
template<class T>
lc_d_ref< T >::lc_d_ref
( ) [inline]

template<class T>
lc_d_ref< T >::lc_d_ref
( T * p ) [inline]

template<class T>
lc_d_ref< T >::lc_d_ref
(  T *  p,
   int *  ref_cnt
) [inline]
```

8.4.2.2 Member list

(This class has no public member functions).

9

Reference for Utility Classes

This chapter describes some useful classes that do not correspond to Caché datatypes and are not automatically inherited by proxies.

- Data Processing Classes — transaction control, batch inserts with the Light C++ Binding, and standard queries.
- Error Classes — error reporting.

9.1 Data Processing Classes

- **Transaction** — provides automatic rollback if the program encounters an exception.
- **LC_Batch** — batch insert class for the Light C++ Binding.
- **d_query** — provides methods for preparing an SQL query, binding parameters, executing the query, and traversing the result set.

9.1.1 Class InterSystems::Transaction

This class provides a guaranteed automatic rollback in case of exceptions. When a Transaction object goes out of scope, the transaction is rolled back if neither **commit()** nor **rollback()** has been called. Unlike the **Database** transaction methods, this class does not allow nested transactions. For more information about both types of transaction, see Using Transactions.

9.1.1.1 Constructor

Transaction()

Class constructor starts the transaction (unlike a Database object, which requires a call to **Database::tstart()**).

```
InterSystems::Transaction::Transaction
(  Database * _db ) [inline]
```

9.1.1.2 Member list

commit()

commits the transaction.

```
void InterSystems::Transaction::commit
( ) [inline]
```

Calling **commit()** more than once for the same Transaction object does nothing (unlike **Database::tcommit()**, which can be called repeatedly to roll back multiple levels of a nested transaction).

rollback()

rolls back the current transaction.

```
void InterSystems::Transaction::rollback
( ) [inline]
```

Called automatically if the Transaction object goes out of scope before the transaction is committed or rolled back.

9.1.2 Class InterSystems::LC_Batch

This class provides methods for batch insertion using the Light C++ Binding. For more information, see Batch Insert.

9.1.2.1 Constructor

LC_Batch()

```
InterSystems::LC_Batch::LC_Batch
(  LC_Database * _db,
   int _concurrency = 1,
   bool _return_ids = false,
   bool _throw_errs = true,
   size_t reserve_size = 32768,
   bool _do_tx = false
) [inline]
```

9.1.2.2 Member list

clear()

To avoid saving objects already added to batch, call clear()

```
void InterSystems::LC_Batch::clear
( ) [inline]
```

clear_errors()

```
void InterSystems::LC_Batch::clear_errors
( ) [inline]
```

clear_ids()

```
void InterSystems::LC_Batch::clear_ids
( ) [inline]
```

close()

```
void LC_Batch::close
( )
```

flush()

To save objects to database, call flush(), close(), or destroy the batch object

```
void LC_Batch::flush
( )
```

get_errors()

Return a list of errors.

```
const std::vector< std::pair<d_status, d_binary> >&
   InterSystems::LC_Batch::get_errors
( ) const [inline]
```

After flush(), get_errors() returns list:

- If no errors, size() of list is 0
- If errors, each list entry is pairing of error status and object serialization
- Projection object has set_from_err_list() member function, which can be used to examine properties of objects which had errors

get_ids()

```
const std::vector<d_string>& InterSystems::LC_Batch::get_ids
( ) const [inline]
```

operator <<

```
template<typename T>
LC_Batch& InterSystems::LC_Batch::operator<<
(  const d_ref< T > & obj) [inline]

template<typename T>
LC_Batch& InterSystems::LC_Batch::operator<<
(  const lc_d_ref< T > & obj) [inline]

template<typename T>
LC_Batch& InterSystems::LC_Batch::operator<<
(  T * obj) [inline]
```

9.1.3 Class InterSystems::d_query

Provides methods for preparing an SQL query, binding parameters, executing the query, and traversing the result set. For more information on this class, see Using Queries.

9.1.3.1 Constructor

d_query()

```
InterSystems::d_query::d_query
( ) [inline]

InterSystems::d_query::d_query
(  Database * db) [inline]
```

9.1.3.2 Member list

close()

```
bool abs_d_query::close
( ) [inherited]
```

execute()

```
void abs_d_query::execute
( ) [inherited]

void InterSystems::abs_d_query::execute
(  const wchar_t * sql_query) [inline, inherited]
```

fetch()

```
bool InterSystems::d_seq_query::fetch
( ) [inline, inherited]
```

get_col_name()

```
const SQLWCHAR* InterSystems::abs_d_query::get_col_name
(  int idx) const [inline, inherited]
```

get_col_name_len()

```
SQLSMALLINT InterSystems::abs_d_query::get_col_name_len
(  int idx) const [inline, inherited]
```

get_col_sql_type()

```
SQLSMALLINT InterSystems::abs_d_query::get_col_sql_type
(  int idx) const [inline, inherited]
```

get_cur_idx()

```
int InterSystems::d_seq_query::get_cur_idx
( ) const [inline, inherited]
```

get_data()

```
void d_seq_query::get_data
(  char * buf,
   int * size,
   int cap,
   bool * is_null = 0
) [inherited]

void d_seq_query::get_data
(  d_binary * val) [inherited]

void d_seq_query::get_data
(  d_bool * val) [inherited]

void d_seq_query::get_data
(  d_currency * val) [inherited]

void d_seq_query::get_data
(  d_date * val) [inherited]

void d_seq_query::get_data
(  d_double * val) [inherited]

void d_seq_query::get_data
(  d_int * val) [inherited]

void d_seq_query::get_data
(  d_string * val,
   str_conv_t conv = NO_CONV
) [inherited]

void d_seq_query::get_data
(  d_time * val) [inherited]

void d_seq_query::get_data
(  d_timestamp * val) [inherited]

void d_seq_query::get_data
(  D_type * val) [inherited]

void d_seq_query::get_data
(  d_wstring * val) [inherited]
```

```
void d_seq_query::get_data
(  DATE_STRUCT * val,
   bool * is_null = 0
) [inherited]

void d_seq_query::get_data
(  double * val,
   bool * is_null = 0
) [inherited]

void d_seq_query::get_data
(  long * val,
   bool * is_null = 0
) [inherited]

void d_seq_query::get_data
(  std::string * val,
   bool * is_null = 0
) [inherited]

void d_seq_query::get_data
(  std::wstring * val,
   bool * is_null = 0
) [inherited]

void d_seq_query::get_data
(  TIME_STRUCT * val,
   bool * is_null = 0
) [inherited]

void d_seq_query::get_data
(  TIMESTAMP_STRUCT * val,
   bool * is_nuol = 0
) [inherited]

void d_seq_query::get_data
(  void * buf,
   int * size,
   int cap,
   bool * is_null = 0
) [inherited]

void d_seq_query::get_data
(  wchar_t * buf,
   int * size,
   int cap,
   bool * is_null = 0
) [inherited]

void InterSystems::d_seq_query::get_data
(  bool * val,
   bool * is_null = 0
) [inline, inherited]
```

get_job_id()

```
int InterSystems::abs_d_query::get_job_id
( ) [inline, inherited]
```

get_num_cols()

```
int InterSystems::abs_d_query::get_num_cols
( ) const [inline, inherited]
```

get_num_pars()

```
int InterSystems::abs_d_query::get_num_pars
( ) const [inline, inherited]
```

get_par_col_size()

```
SQLUINTEGER InterSystems::abs_d_query::get_par_col_size
(  int idx) const [inline, inherited]
```

get_par_num_dec_digits()

```
SQLSMALLINT InterSystems::abs_d_query::get_par_num_dec_digits
(  int idx) const [inline, inherited]
```

get_par_sql_type()

```
SQLSMALLINT InterSystems::abs_d_query::get_par_sql_type
(  int idx) const [inline, inherited]
```

is_par_nullable()

```
SQLSMALLINT InterSystems::abs_d_query::is_par_nullable
(  int idx) const [inline, inherited]
```

is_par_unbound()

```
bool InterSystems::abs_d_query::is_par_unbound
(  int idx) const [inline, inherited]
```

prepare()

```
void abs_d_query::prepare
(  const char * cl_name,
   const char * proc_name
) [inherited]

void abs_d_query::prepare
(  const char * sql_query,
   int len
) [inherited]

void abs_d_query::prepare
(  const wchar_t * cl_name,
   const wchar_t * proc_name
) [inherited]

void abs_d_query::prepare
(  const wchar_t * sql_query,
   int len
) [inherited]

void abs_d_query::prepare
(  d_string & sql_name,
   int num_pars
) [inherited]

void InterSystems::abs_d_query::prepare
(  const char * sql_query) [inline, inherited]

void InterSystems::abs_d_query::prepare
(  const wchar_t * sql_query) [inline, inherited]
```

set_cur_idx()

```
void InterSystems::d_query::set_cur_idx
(  int idx) [inline]
```

set_par()

```
void abs_d_query::set_par
(  int idx) [inherited]

void abs_d_query::set_par
(  int idx,
   const char * buf,
   char_size_t size
) [inherited]

void abs_d_query::set_par
(  int idx,
   const d_binary & val
) [inherited]
```

```
void abs_d_query::set_par
(  int idx,
   const d_bool & val
) [inherited]

void abs_d_query::set_par
(  int idx,
   const d_currency & val
) [inherited]

void abs_d_query::set_par
(  int idx,
   const d_date & val
) [inherited]

void abs_d_query::set_par
(  int idx,
   const d_double & val
) [inherited]

void abs_d_query::set_par
(  int idx,
   const d_int & val
) [inherited]

void abs_d_query::set_par
(  int idx,
   const d_string & val
) [inherited]

void abs_d_query::set_par
(  int idx,
   const d_time & val
) [inherited]

void abs_d_query::set_par
(  int idx,
   const d_timestamp & val
) [inherited]

void abs_d_query::set_par
(  int idx,
   const D_type & val
) [inherited]

void abs_d_query::set_par
(  int idx,
   const d_wstring & val
) [inherited]

void abs_d_query::set_par
(  int idx,
   const void * buf,
   byte_size_t size
) [inherited]

void abs_d_query::set_par
(  int idx,
   const wchar_t * buf,
   char_size_t size
) [inherited]

void abs_d_query::set_par
(  int idx,
   double val
) [inherited]

void abs_d_query::set_par
(  int idx,
   int val
) [inherited]

void InterSystems::abs_d_query::set_par
(  int idx,
   const char * val
) [inline, inherited]
```

```
void InterSystems::abs_d_query::set_par
(  int idx,
   const std::string & val
) [inline, inherited]

void InterSystems::abs_d_query::set_par
(  int idx,
   const std::wstring & val
) [inline, inherited]

void InterSystems::abs_d_query::set_par
(  int idx,
   const wchar_t * val
) [inline, inherited]
```

set_par_default()

```
void abs_d_query::set_par_default
(  int idx) [inherited]
```

set_stored_proc()

```
void InterSystems::abs_d_query::set_stored_proc
(  bool is_stored_proc) [inline, inherited]
```

skip()

```
void InterSystems::d_seq_query::skip
(  unsigned int num_cols = 1) [inline, inherited]
```

throw_err()

```
void abs_d_query::throw_err
(  SQLSMALLINT err_src,
   SQLHANDLE handle
) [static, inherited]
```

unbind_pars()

```
void abs_d_query::unbind_pars
( ) [inherited]
```

9.2 Error Classes

Provides error reporting.

9.2.1 Class InterSystems::Db_err

See A Sample C++ Binding Application for an example that uses this class.

9.2.1.1 Constructor

Db_err()

```
InterSystems::Db_err::Db_err
( ) [inline]

InterSystems::Db_err::Db_err
(  int c) [inline]
```

```
InterSystems::Db_err::Db_err
(  int c,
   const char * m,
   int l,
   const char * s
) [inline]

InterSystems::Db_err::Db_err
(  int c,
   const std::string & m
) [inline]

InterSystems::Db_err::Db_err
(  int c,
   const std::string & m,
   const char * s
) [inline]
```

9.2.1.2 Member list

clear()

```
void InterSystems::Db_err::clear
( ) [inline]
```

get()

```
void Db_err::get
(  t_istream & in)
```

get_code()

```
int InterSystems::Db_err::get_code
( ) const [inline]
```

get_msg()

```
const std::string& InterSystems::Db_err::get_msg
( ) const [inline]
```

get_src()

```
const std::string& InterSystems::Db_err::get_src
( ) const [inline]
```

log()

```
void Db_err::log
( ) const
```

make_err_msg()

```
std::string Db_err::make_err_msg
(  const char * msg,
   const char * arg1 = 0,
   const char * arg2 = 0,
   const char * arg3 = 0,
   const char * arg4 = 0,
   const char * arg5 = 0
) [static]

std::string Db_err::make_err_msg
(  const char * msg,
   const wchar_t * arg1 = 0,
   const wchar_t * arg2 = 0,
   const wchar_t * arg3 = 0,
   const wchar_t * arg4 = 0,
   const wchar_t * arg5 = 0
) [static]
```

operator bool()

```
InterSystems::Db_err::operator-bool
( ) const [inline]
```

reset()

```
void InterSystems::Db_err::reset
(  int c,
   const char * m
) [inline]
```

set_code()

```
void InterSystems::Db_err::set_code
(  int code) [inline]
```

set_msg()

```
void InterSystems::Db_err::set_msg
(  const char * m) [inline]

void InterSystems::Db_err::set_msg
(  const char * m,
   int l
) [inline]
```

set_src()

```
void InterSystems::Db_err::set_src
(  const char * s) [inline]
```

to_xml()

```
void Db_err::to_xml
(  xml_writer & out) const
```