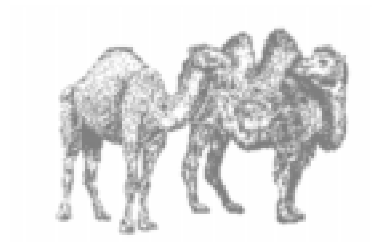
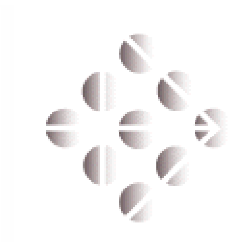
Bill Karwin's

IBPerl User's Guide





IBPerl

http://www.karwin.com

Copyright 2000 Bill Karwin. All rights reserved.

All InterBase products are trademarks or registered trademarks of InterBase Software Corporation. All Borland products are trademarks or registered trademarks of INPRISE Corporation, Borland and Visibroker Products. Other brand and product names are trademarks or registered trademarks of their respective holders.

Table of Contents

Chapter 1 Introduction

IBPerl capabilities . . . 8
System requirements . . . 8
Licensing . . . 8
Installing on Linux or UNIX . . . 9
Installing on Microsoft Windows . . . 10
  Differences from the Linux/UNIX instructions . . . 10
  Using IBPerl with ActiveState Perl . . . 10
Using IBPerl in a Perl script . . . 11

Chapter 2 Connecting to a database

Starting a new connection session . . . 14
Specifying server and database path . . . 14
  Local databases . . . 14
  Remote databases . . . 14
  Protocol parameter . . . 14
Specifying user and password . . . 15
Creating a new database . . . 15
Using database options . . . 16
  SQL role . . . 16
  Cache size . . . 16
  Page size . . . 16
  Character set . . . 16
  SQL dialect of a Connection . . . 17
Ending a connection . . . 18
Using concurrent connections . . . 18
Handling connection errors . . . 18
Connection class reference . . . 19

Chapter 3 Using Transactions

Understanding transactions . . . 22
  Atomicity . . . 22
  Consistency . . . 22
  Isolation . . . 22
  Durability . . . 22
Starting a new transaction . . . 22
Committing a transaction . . . 23
Rolling back a transaction . . . 23
Using concurrent transactions . . . 23
Handling transaction errors . . . 24
Understanding transaction options . . . 24

Transaction class reference . . . 25

Chapter 4 Basic SQL

Preparing a new query . . . 28
Executing the query . . . 28
Fetching results . . . 29
    Fetching into a scalar . . . 29
    Fetching into a list . . . 29
    Fetching into a hash list . . . 29
    Fetching all rows . . . 29
    Fetching Blob values . . . 30
Using query options . . . 30
    Setting the date and time formats . . . 30
    Setting the field Separator . . . 31
    SQL Dialect of a Statement . . . 31
Closing a query . . . 31
Using concurrent statements . . . 32
Executing statements other than SELECT . . . 32
    INSERT, UPDATE, and DELETE . . . 32
    EXECUTE PROCEDURE . . . 33
    Data definition language . . . 33
    Other statements . . . 33
Building SQL statements . . . 34
Handling statement errors . . . 34
Statement class reference . . . 35

Chapter 5 Advanced Queries

Parameterized queries . . . 40
    Specifying parameters . . . 40
    Specifying parameters with NULL state . . . 40
    Invoking a prepared query multiple times . . . 40
Blob parameters . . . 41
Positioned updates & deletes . . . 41
    Using update( ) . . . 41
    Using delete( ) . . . 42
Fetching dates as a list . . . 42
Accessing Statement internals . . . 43
    Statement properties . . . 43
    Dataset arrays . . . 44
    Counting rows affected by an operation . . . 44
    Counting executions and fetches . . . 44
Translating quotes for SQL . . . 45
    How InterBase interprets quotes . . . 45
    How IBPerl translates quotes . . . 45

Caveat: quotes and binary parameters . . . 45

Chapter 6 Examples
Copying data from one database to another . . . 48
Joining queries from two databases . . . 50
Output an InterBase dataset as an HTML table . . . 53

Appendix A License terms
The Artistic License . . . 55
GNU General Public License . . . 56

Appendix B Format Strings
Date/Time format elements . . . 61

CHAPTER 1

Introduction

IBPerl is a module and extension for Perl 5. It implements an interface to the dynamic SQL query facility of InterBase, a relational database management system. IBPerl is open source and freeware.

Topics in this chapter are:

- **IBPerl capabilities**
- **System requirements**
- **Licensing**
- **Installing on Linux or UNIX**
- **Installing on Microsoft Windows**
- **Using IBPerl in a Perl script**

IBPerl capabilities

- IBPerl has the following high-level capabilities:
  - Write Perl 5 scripts to connect to local or remote InterBase server software
  - Execute SQL statements in the context of this InterBase database connection
  - Retrieve results of SQL queries
- Specific features of IBPerl include those in the following list:
  - Connect to an InterBase database on the same host where your IBPerl script runs
  - Connect to an InterBase database on a networked host other that the host where your IBPerl script runs; this host can even run a different operating system
  - Optionally specify connection properties, including cache buffers, international character set, and SQL dialect
  - Connect to two or more databases simultaneously in one script; each database can be on different hosts
  - Execute SELECT queries and retrieve the resulting dataset, row by row
  - Execute other data manipulation statements (INSERT, UPDATE, DELETE)
  - Execute stored procedures
  - Supply input parameters to a prepared query
  - Perform positioned updates and deletes to a live query
  - Execute *data definition statements* (CREATE, ALTER, DROP) to update database metadata, or schema
  - Use transactions to achieve concurrent and consistent database operations
  - Commit or roll back work performed in the context of a transaction

System requirements

You need the InterBase client library (e.g., libgds.so on Linux or gds32.dll on Win32) installed on the host where you intend to compile and use IBPerl. IBPerl needs the InterBase library present to function.

IBPerl supports InterBase V4.0, V4.1, V4.2.1, V5.0, V5.1.1, V5.5, V5.6, and V6.0. Earlier versions of InterBase are not likely to work with IBPerl. Later versions of InterBase are more recent than the date that this document, and no assumption is made for future compatibility.

Of course, you need Perl 5 installed. IBPerl requires some minor features of Perl 5.003, and it is recommended to use stable builds of Perl 5.004 or later, because of their superior reliability. IBPerl makes use of the `ExtUtils::MakeMaker` and `Config` packages, which should be included with the standard Perl distribution.

Licensing

You can use, copy,and distribute IBPerl under terms described in the “Artistic License” which you can find:

- In **Appendix A, License terms on page 55** of this document.
- Included with the IBPerl distribution in Artistic.txt
- Included with the Perl 5 distribution
- At the URL: http://www.perl.com/pub/language/misc/Artistic.html.

You can alternately use, copy, and distribute IBPerl under terms described in the GNU General Public License, which you can find:

- In **Appendix A, License terms on page 55** of this document.
- Included with most GNU and Linux distributions
- At the URL: http://www.gnu.org/copyleft/gpl.html.

Installing on Linux or UNIX

IBPerl is distributed in source code form, and it should be easy to compile it on any platform that has a C or C++ compiler. IBPerl is a Perl extension, partially written in C. It does require you to compile it.

1. Unpack the IBPerl distribution using tar.

   ```
   tar xzvf IBPerl.tar.gz
   ```

2. Change directory into the top directory of the IBPerl distribution.

   ```
   cd IBPerl-08
   ```

3. Build a Makefile according to local configuration.

   ```
   perl Makefile.PL
   ```

   This should report its actions, or give an error describing why it could not finish. For example, if it is successful building a Makefile with InterBase 6.0 on Linux, the output should appear like the following:

   ```
   IBPerl 0.8 configuring for InterBase LI-B6.0.0.553 on the linux platform.
   Writing Makefile for IBPerl
   ```

   Note that you must compile IBPerl using the same compiler that was used to build the perl executable itself. IBPerl builds as a dynamically loadable library and the dynamic linking mechanism is usually compiler-dependent.

4. Compile the Perl extension.

   ```
   make
   ```

   This should take less than one minute unless you have an exceptionally slow or overburdened computer.

5. You can test IBPerl before you install it into system directories on your host. Change directory into the examples subdirectory of the IBPerl distribution.

   ```
   cd examples
   cp /usr/interbase/examples/employee.gdb .
   perl -Mblib select.pl
   ...run other scripts in this directory...
   cd ..
   ```

6. Once you are satisfied that the example scripts run and IBPerl functions correctly, you can install it into the Perl library on your hosts's system directory. There are two ways you can do this:

   - Install using the “install” target in the Makefile.

     ```
     make install
     ```

   - Run the Install script provided in the distribution.

     ```
     perl Install
     ```

   Either of these commands should copy the compiled IBPerl files into your system directory. Now you can use IBPerl in any Perl script you write on this host.

BUILD ISSUES ON LINUX

Linux has a special issue related to the version of the C runtime library and compatibility with the InterBase client library for different releases of InterBase. You might receive an error message that includes the following phrase when you try to compile IBPerl with older versions of InterBase on newer versions of Linux:

```
...undefined symbol: _xstat
```

The GNU glibc 2.1 runtime library is not backward compatible with earlier releases of glibc. The designers of glibc changed *xstat* and some other functions between version 2.0 and 2.1. Dynamic shared libraries (such as InterBase's libgds.so) that were built using GNU glibc 2.0, find that some of the symbols have been taken out in glibc 2.1.

Any one of the three following solutions resolves this issue:

- Upgrade to InterBase V6.0 or V5.6 for Linux, which were built using glibc 2.1.
- Compile IBPerl with the additional library flag `-lNoVersion-2.1.2`, which resolves the missing symbols. IBPerl 0.8 and later attempts to link with this library if it is present in your operating system.
- Acquire, compile, and use glibc 2.0[1] (4MB) instead of the version of glibc that came with your Linux system.

Installing on Microsoft Windows

The instructions for installing IBPerl on Microsoft Windows are largely similar to those for on Linux or UNIX.

Differences from the Linux/UNIX instructions

Step 1. involves use of GNU tar, which your Windows installation does not necessarily have (you can get it with the Cygwin tools from Red Hat[2]). IBPerl alternately comes in a Zip archive distribution, so you can unpack it on Windows with Niko Mak Computing Inc.'s WinZip® or another Zip archive extraction tool.

Many Windows users do not have a C compiler installed, IBPerl includes a pre-compiled set of binaries ready to install. You can therefore skip step 3. and step 4.

If you have no make utility, you can use the Install script as described in step 6. to install the pre-compiled binaries for IBPerl.

Using IBPerl with ActiveState Perl

IMPORTANT *You cannot use ActiveState Perl with the pre-compiled binary provided in IBPerl.*

IBPerl, like any dynamic library, must match the binary object format of the calling executable. In this case, the calling executable is perl.exe. Windows binaries that were built using the Borland C++ compiler and binaries that were built using the Microsoft Visual C++ compiler are incompatible. The binary distribution of IBPerl is built with the free Borland compiler. Some binary distributions of Perl available on the Internet, such as the ActiveState port of Perl for Win32, are built with Microsoft Visual C++.

For this reason, you cannot use ActiveState Perl with the pre-compiled binary provided in IBPerl. You must compile IBPerl yourself using the Microsoft compiler. Makefile.PL attempts to detect ActiveState Perl and create a Makefile with Microsoft Visual C++ compiler commands. However, this is untested and likely to require further work. Good luck.

1. Download from ftp://prep.ai.mit.edu/gnu/glibc/glibc-2.0.6.tar.gz.
2. Downloadable from the URL http://www.redhat.com/support/manuals/gnupro99r1/6_embed/embCygwin.html.

A suitable binary distribution of Perl 5 for Win32 built with Borland C++ is available in Gurusamy Sarathy's area of the CPAN web site[3].

Using IBPerl in a Perl script

You can load IBPerl packages and methods to make them available in your script, by including the following command somewhere in your script:

```
use IBPerl;
```

This statement, and other `use` statements, conventionally belong near the top of a Perl script.

3. Downloadable from the URL http://www.cpan.org/authors/id/GSAR/perl5.00402-bindist04-bc.zip.

CHAPTER

2

Connecting to a database

This chapter describes methods to connect to an InterBase database from a Perl script using the `IBPerl::Connection` package. This assumes you have installed IBPerl and loaded it into your script with the `use IBPerl;` command.

Topics in this chapter are:

- **Starting a new connection session**
- **Specifying server and database path**
- **Specifying user and password**
- **Creating a new database**
- **Using database options**
- **Ending a connection**
- **Using concurrent connections**
- **Handling connection errors**
- **Connection class reference**

Starting a new connection session

You can connect to a database using the constructor `new` for the `IBPerl::Connection` package. This constructor returns a Perl scalar, which you use in your script as a handle to reference the database connection.

```
my $db = new IBPerl::Connection( ...parameters... );
```

The scalar `$db` is assigned with a handle to a Connection object. You can use this object in other IBPerl methods. The variable name `db` is arbitrary for the examples in this document. You can use any valid Perl scalar name.

Specifying server and database path

You use one or several parameters to specify the database to which the Connection object instance should connect.

Local databases

You can connect to a local database by using simply the `Path` parameter:

```
$db = new IBPerl::Connection(
   Path => '/usr/interbase/examples/employee.gdb' );
```

Tip You can supply Perl variables in place of literal scalars:

```
$database = '/usr/interbase/examples/employee.gdb';
$db = new IBPerl::Connection( Path => $database );
```

Remote databases

You can connect to a remote database by using the `Path` and the `Server` parameters together. The `Server` parameter is the hostname of the remote server host on which your InterBase database resides.

```
$db = new IBPerl::Connection(
   Server => 'linuxhost',
   Path => '/usr/interbase/examples/employee.gdb' );
```

The value of the `Server` parameter can also be an IP address if your client's system call `gethostbyname()` supports resolution of IP address strings. For example, Linux and Winsock 2 are known to support this behavior.

The pathname is relative to the server host, that is, the path to the database file if you were logged in on the server. You cannot use a pathname that is a mapped drive or NFS-mounted filesystem.

Protocol parameter

By default IBPerl uses TCP/IP network protocol from the client host to the server host. This works for any InterBase server. InterBase on Windows NT servers also supports NetBEUI protocol, and InterBase on Novell NetWare servers also supports IPX/SPX protocol.

Below is an example using the NetBEUI protocol:

```
$db = new IBPerl::Connection(
   Server => 'ntserver',
   Protocol => 'NetBEUI',
   Path => '/usr/interbase/examples/employee.gdb' );
```

Below is an example using the IPX/SPX protocol:

```
$db = new IBPerl::Connection(
    Server => 'netwareserver',
    Protocol => 'IPX/SPX',
    Path => '/usr/interbase/examples/employee.gdb' );
```

It is not required to use NetBEUI on IPX/SPX if you have an NT or NetWare server. TCP/IP works well for all server platforms. The `Protocol` parameter is therefore optional.

Specifying user and password

In addition to the server and path to the database, you must supply a valid user name and matching password, that exist in the InterBase password database on the database server.

```
$db = new IBPerl::Connection(
    Path => '/usr/interbase/examples/employee.gdb',
    User => 'algore',
    Password => 'green1');
```

IMPORTANT The password appears in *plain text* in your script; it is not encrypted. Anyone who can read the script can view the password. Take appropriate security precautions to prevent unauthorized persons from viewing your script if your database contains sensitive information.

You can also supply the user name or password using environment variables `ISC_USER` and `ISC_PASSWORD`. You can substitute this technique for either or both parameters.

IBPerl does not itself provide interactive password prompting. You can do this yourself by reading user input in your Perl script.

Creating a new database

You can also use the `IBPerl::Connection` package to create a new, empty database. Use the `create` method in a manner similar to the `new` method.

```
$db = create IBPerl::Connection( ...parameters... );
```

The `create` method also connects to the newly created database and returns a live connection to the calling script.

IMPORTANT If you specify a path (and optimally server hostname) to a database that already exists, the `create` method destroys the existing database and all data within it. The data is not recoverable. If you accidentally destroy a database that contained important data, your only recourse is to restore the database from a recent backup. Be careful using the `create` method.

Using database options

There are several more optional parameters that you can specify to the `new` and `create` methods of `IBPerl::Connection`.

SQL role

InterBase offers ANSI SQL roles, which are named groups of privileges. A given user might have been granted the privilege to adopt a role at the time of connection. You can specify a role to use when you connect to a database.

```
$db = new IBPerl::Connection( ...parameters... ,
    Role => 'DataEntry');
```

The named role must exist in the database, and the user must have the privilege to adopt the role.

Cache size

You can specify that the connection should use a greater amount of system memory on the server for caching data and index pages from the database for the duration of your connection. A larger cache often improves query performance. See “Setting Database Cache Buffers” is chapter 3 of the *Embedded SQL Guide* (titled the *Programmer’s Guide* in InterBase V4 and V5 manuals).

```
$db = new IBPerl::Connection( ...parameters... ,
    Cache => 3000);
```

The cache is measured in number of database pages.

Synonym for Cache: `Buffers`.

Page size

The `Page_Size` parameter is used only by the `create` method. Page size is a database property that you can set at the time you create the database. All pages in a database have the same page size. You cannot change the page size after you create the database, unless you back up and restore the database.

Values are 1024, 2048, 4096, or 8192 bytes. InterBase rounds down values other than these. For example, if you specify 5000 as the page size, InterBase creates a database with a page size of 4096.

```
$db = new IBPerl::Connection( ...parameters... ,
    Page_Size => 4096);
```

Synonyms for Page_Size: `PageSize`, `Pagesize`.

Character set

You can specify a character set with the `Charset` parameter when you connect to or create a database. InterBase can translate character-based data between a client’s preferred character set, and the character set the database uses to store the data.

```
$db = new IBPerl::Connection( ...parameters... ,
    Charset => 'ISO8859_1');
```

When you use the `Charset` parameter with the `new` connection constructor, you specify a client character set. When you SELECT text data, InterBase transliterates the data from the character set defined for the given field to the client character set. When you INSERT or UPDATE text data, InterBase transliterates values from the client character set to the storage character set.

When you use the `Charset` parameter with the `create` constructor, you specify the default character set for all subsequent definitions of CHAR, VARCHAR, and BLOB SUB_TYPE TEXT fields.

Synonym for Charset: `CharSet`.

SQL dialect of a Connection

InterBase V6 has two sets of SQL semantics. One is called "dialect 1" and it is similar to the semantics of previous releases of InterBase. The other is called "dialect 3" and it includes certain new SQL syntactic and semantic rules that are mutually exclusive with the older behavior.

Dialect 1	*Dialect 3*
DATE datatype is a 64-bit object encoding day and time	*DATE datatype is a 32-bit object encoding only the day*
TIME datatype is unknown	*TIME datatype is a 32-bit object encoding only the time since midnight*
TIMESTAMP datatype is unknown	*TIMESTAMP datatype is a 64-bit object encoding day and time*
NUMERIC(10) and greater, and DECIMAL(10) and greater, are stored using IEEE double precision floating point values; they suffer from inexact rounding errors; the maximum precision is 15	*NUMERIC(10) and greater, and DECIMAL(10) and greater, are stored using 64-bit (long long) integers with optional scale; they do not suffer from rounding errors; the maximum precision is 18*
Single quotes `'` *and double quotes* `"` *both indicate the delimiter of a string literal*	*Single quotes* `'` *indicate the delimiter of a string literal; double quotes* `"` *indicate the delimiter of a metadata object name, so that these names can contain SQL keywords, special characters, 8-bit international characters, or whitespace*

TABLE 2.1 *Differences between dialect and dialect 2*

Other SQL syntax and semantics, even features introduced in InterBase V6, work identically in both dialects.

Example:

```
$db = new IBPerl::Connection( ...parameters... ,
   Dialect => 3);
```

The default for the `Dialect` property is `undef`. In this case, the InterBase client determines the default.

See also **"SQL Dialect of a Statement" on page 31**.

Ending a connection

IBPerl and the InterBase client maintain an open socket to the InterBase server for the duration of the connection. When you are finished with database operations and wish to close this connection, you can use the virtual method[4], `disconnect`:

```
$db->disconnect();
```

IBPerl automatically invokes the `disconnect` method when your database connection object goes out of scope. For example:

```
{
    my $db = new IBPerl::Connection( . . . );
}
```

At the end of this code block, the $db object is garbage collected and its destructor calls disconnect.

Using concurrent connections

You can connect to two more databases at the same time in a single script using IBPerl. This is useful because your data might be logically and physically separated into more than one database. These databases can even be on separate server hosts. Your script can connect to databases that are on the same host where the script runs, or using the `Server` parameter, you can specify that the database is on a remote host elsewhere on your network.

Use the constructor to create multiple database connections:

```
$db1 = new IBPerl::Connection( Server => 'ntserver', PATH => 'C:/data/stock.gdb' );
$db2 = new IBPerl::Connection( Server => 'linuxserver', PATH => '/usr/data/invoice.gdb' );
. . .
$db1->disconnect();
$db2->disconnect();
```

You can create subsequent transactions and SQL queries in the context of only *one* database connection object. That is, you cannot create a query that joins data from tables in separate databases (though see the example of implementing client-side cross-database joins in **@@**).

Handling connection errors

There are many ways an attempt for a script to connect to a database can fail. For example, network outages can prevent the client host from communicating with the server host to initiate a socket. The user's password might have been changed. The pathname to the database file might contain a spelling error.

If something happens that prevents a connection request from succeeding, IBPerl returns to the calling script an invalid database connection handle. An instance of the `Connection` object has properties `Handle` and `Error`. In case of an unsuccessful connection, the `Handle` property is set to a negative number, and the `Error` property contains a description of the cause of the failure.

4. A ***virtual method*** **is a function that implicitly affects only the object for which it is called. You invoke a virtual method for a given object by using the syntax:** `$object->method()`.

For example, to check for and report a connection failure:

```
$db = new IBPerl::Connection( ...parameters... );
if ($db->{Handle} < 0) {
    carp "Failure to connect to database. Error as follows:\n";
    carp "$db->{Error}\n";
    exit 1; # No sense in continuing.
}
```

As with almost everything in the Perl language, there are other ways to structure code to accomplish the same task:

```
die "$db->{Error}\n" if ($db->{Handle} < 0);
```

Most IBPerl virtual methods simply return a negative number on failure. Only the constructor methods require you to examine the `Handle` property for an indicator of error.

IMPORTANT You should always include code to check for failure, report errors, and react accordingly. Code examples in this document might omit error-checking for brevity.

Connection class reference

Below is a table of all properties and methods of the `IBPerl::Connection` class.

Property/Method	*Description*
new()	*Static constructor method to return a connection to a database.*
create()	*Static constructor method to create a new database and return a connection.*
disconnect()	*Virtual method to terminate a connection to a database.*
Path	*Scalar argument to the constructor; the absolute pathname of a database on a server.*
Server	*Optional scalar argument to the constructor; the hostname of the server on which InterBase runs. Required if the database server is remote.*
Protocol	*Optional scalar argument to the constructor; the network protocol for a connection to a remote server. Defaults to 'TCP/IP' and can be 'NetBEUI' or 'IPX/SPX'.*
User	*Scalar argument to the constructor; the name of user that the script uses to identify itself to the InterBase server. Defaults to the environment variable* `ISC_USER`.
Password	*Scalar argument to the constructor; the password for the user named by the* `User` *parameter. Defaults to the environment variable* `ISC_PASSWORD`.
Cache	*Optional scalar argument to the constructor; number of data pages to allocate on the server for cache. Synonym: Buffers.*
Role	*Optional scalar argument to the constructor; the SQL Role to adopt when the user connects to a database.*
Page_Size	*Optional scalar argument to the* `create` *constructor; the database page size to use for a new database. Synonyms: PageSize, Pagesize.*

TABLE 2.2 `IBPerl::Connection` *properties and methods*

Property/Method	***Description***
Charset	*Optional scalar argument to the constructor; the default character set to use for a new database or for the context during a connection do an existing database. Synonym: CharSet.*
Handle	*Scalar used internally by the Connection class. You only use this to test if its value is less than zero. If so, the connection failed.*
Error	*A scalar error message in the case when* `Handle` *is less than zero.*
Dialect	*Optional scalar argument to the constructor; the SQL dialect of the database (InterBase V6 only). Values can be 1 or 3.*
Active	*IBPerl maintains this scalar property. It has a value of 'Y' while the connection is active, and 'N' if the connection fails or after it disconnects.*

TABLE 2.2 `IBPerl::Connection` *properties and methods*

CHAPTER

3

Using Transactions

This chapter describes the purpose of transactions, and the methods in IBPerl to control transactions with the `IBPerl::Transaction` package. This assumes you have installed IBPerl, loaded it into your script with the `use IBPerl;` command, and have a valid connection to a database.

Topics in this chapter are:

- **Understanding transactions**
- **Starting a new transaction**
- **Committing a transaction**
- **Rolling back a transaction**
- **Using concurrent transactions**
- **Handling transaction errors**
- **Understanding transaction options**
- **Transaction class reference**

Understanding transactions

A transaction is an entity that provides a context for any database operation. Every SQL query must execute within a transaction context.

Atomicity

A transaction provides a mechanism to make database changes atomic, so that you can perform one or many operations and apply them all at once to the database. All changes done in the context of one transaction therefore appear as one instantaneous change to other database clients. It is not possible for another client to view your database changes in a partially finished state.

Consistency

All work you perform within a transaction must transform the data in the database from one valid state to another valid state. For example, changes cannot be committed if they violate constraints or referential integrity. Also, if you abort the transaction, the database is returned to the previous valid state.

Isolation

Using a transactions also allows your application to view an isolated snapshot of the database, regardless of the changes applied by other clients in the meantime. The state of data in the database at the instant you begin a transaction is preserved for the duration of your transaction. Therefore if you query data in a given table once, and then use the same query again some minutes later, InterBase guarantees to return the same data, even though other clients might have committed changes to that table between the times of your two queries. This is important, for example, for reporting tools that generate multiple statistical analyses over the same data.

Durability

When you commit the work of a transaction, the changes are permanent and durable. That is, if you shut down the database or the machine it runs on, and then restart it, your changes are remembered by the database. Once a transaction completes successfully, its effects cannot be altered without running a compensating transaction.

Starting a new transaction

You can start a transaction with the constructor `new` for the package `IBPerl::Transaction`. You must provide a valid database object; you cannot start a transaction without being connected to a database.

```
my $tr = new IBPerl::Transaction( Database => $db );
```

The scalar `$tr` is assigned with a handle to a transaction object. You can use this object when invoking SQL queries. The variable name `tr` is arbitrary for the examples in this document. You can use any valid Perl scalar name.

Committing a transaction

When you are ready to apply your changes to the database and end the transaction, use the `commit` virtual method:

```
$tr->commit();
```

All changes that your SQL operations wrote to database in the context of the transaction are implicitly marked as committed, as an effect of the change in status of that transaction. A commit operation also commits any changes made by inference with triggers or cascading constraint effects. There is no way to commit some changes while discarding other changes that you made in the same transaction.

The transaction ends, and you cannot use this instance of the transaction object in further database queries. You must start a new transaction to do that.

IBPerl automatically invokes the `commit` method when your transaction object goes out of scope. For example:

```
{
   my $tr = new IBPerl::Transaction( . . . );
}
```

At the end of this code block, the `$tr` object is garbage collected and its destructor calls `commit`.

Rolling back a transaction

If you want to abort changes that you made to the database during a transaction, you can *roll back* the transaction. Use the `rollback` virtual method:

```
$tr->rollback();
```

When you roll back, all changes that your SQL operations wrote to database in the context of the transaction are implicitly discarded, as an effect of the change in status of that transaction. A rollback operation also discards any changes made by inference with triggers or cascading constraint effects. There is no way to roll back some changes while keeping other changes that you made in the same transaction.

The transaction ends, and you cannot use this instance of the transaction object in further database queries. You must start a new transaction to do that.

Using concurrent transactions

You can create multiple concurrent transaction objects within the context of a single database connection. This is useful, for instance, if you are running a complex transaction that involves many SQL queries and operations, but you need to perform a minor task on the database and commit it before you can finish your main transaction. A common reason for this is when allocating sequential ID values for use as a primary key.

```
$tr1 = new IBPerl::Transaction( Database => $db );
. . .
$tr2 = new IBPerl::Transaction( Database => $db );
. . .
$tr2->commit();
. . .
$tr1->commit();
```

There is no requirement to treat transactions as nested; given the example above, you could alternately commit `$tr1` before you commit `$tr2`.

You can also create and use transactions on different database connections concurrently and independently:

```
$db1 = new IBPerl::Connection( Path => 'stock.gdb' );
$tr1 = new IBPerl::Transaction( Database => $db1 );

$db2 = new IBPerl::Connection( Path => 'invoice.gdb' );
$tr2 = new IBPerl::Transaction( Database => $db2 );
```

Handling transaction errors

There are many ways an attempt for a script to start a transaction can fail. For example, the database handle might be invalid. The database might be a read-only database (InterBase V6 only). The network connection might fail after you make a successful database connection.

If something happens that prevents a transaction request from succeeding, IBPerl returns to the calling script an invalid database connection handle. An instance of the `Transaction` object has properties `Handle` and `Error`. In case of an unsuccessful operation, the `Handle` property is set to a negative number, and the `Error` property contains a description of the cause of the failure.

For example, to check for and report a transaction failure:

```
$tr = new IBPerl::Transaction( ...parameters... );
if ($tr->{Handle} < 0) {
    carp "Failure to start a transaction. Error as follows:\n";
    carp "$tr->{Error}\n";
    exit 1;
}
```

Most IBPerl virtual methods simply return a negative number on failure. Only the constructor methods require you to examine the `Handle` property for an indicator of error.

IMPORTANT You should always include code to check for failure, report errors, and react accordingly. Code examples in this document might omit error-checking for brevity.

Understanding transaction options

The current release of IBPerl does not provide properties for controlling some InterBase transaction options. All transactions in IBPerl use the default settings of READ WRITE, WAIT lock resolution mode, and SNAPSHOT isolation level.

IBPerl does not currently support two-phase commits for multi-database distributed transactions. Nor does it support other InterBase transaction options: READ ONLY, NO WAIT, READ COMMITTED, RECORD_VERSION, COMMIT RETAINING, ROLLBACK RETAINING, TABLE STABILTY, or pessimistic locking with the RESERVING clause. All these features are possibilities for future improvements to IBPerl.

For more information on the complex InterBase transaction model, the chapter 4 of the InterBase *Embedded SQL Guide* (titled the *Programmer's Guide* in InterBase V4 and V5 manuals).

Transaction class reference

Below is a table of all properties and methods of the `IBPerl::Transaction` class.

Property/Method	*Description*
new( )	*Static constructor method to return an active transaction in a database.*
commit( )	*Virtual method to end a transactions and mark as permanent changes made during that transaction. IBPerl invokes this method during the destructor for the*
rollback( )	*Virtual method to end a transaction and mark as discarded changes made during that transaction.*
Database	*Scalar argument to the constructor method, an IBPerl::Connection object.*
Active	*IBPerl maintains this property. It has a value of 'Y' while the transaction is active, and 'N' if the transaction fails or after it commits or rolls back.*
Handle	*Scalar used internally by the Transaction class. You only use this to test if its value is less than zero. If so, the transaction failed to start.*
Error	*A scalar error message in the case when* `Handle` *is less than zero.*
Mode	*Property reserved for future development.*
Resolution	*Property reserved for future development.*
Isolation	*Property reserved for future development.*
Reserving	*Property reserved for future development.*

TABLE 3.1 `IBPerl::Transaction` *properties and methods*

CHAPTER

4

Basic SQL

This chapter covers simple SQL statements, and the methods in IBPerl to execute queries and retrieve results with the `IBPerl::Statement` package. This assumes you have installed IBPerl, loaded it into your script with the `use IBPerl;` command, and have a valid connection and transaction.

Topics in this chapter are:

- **Preparing a new query**
- **Executing the query**
- **Fetching results**
- **Using query options**
- **Closing a query**
- **Using concurrent statements**
- **Executing statements other than SELECT**
- **Building SQL statements**
- **Handling statement errors**
- **Statement class reference**

Preparing a new query

You can start a query with the constructor `new` for the package `IBPerl::Statement`. You must provide a valid transaction handle; you cannot run a query without an active transaction.

Specify the SQL statement with a scalar parameter to the constructor. This parameter is called `SQL`.

```
my $st = new IBPerl::Statement(
   Transaction => $tr,
   SQL => 'SELECT * FROM EMPLOYEE');
```

The scalar `$st` is assigned with a handle to a statement object. You can use this object to invoke subsequent query methods. The variable name `st` is arbitrary for the examples in this document. You can use any valid Perl scalar name.

Synonyms for SQL: `Statement`, `Stmt`.

Each `Statement` object can have only one SQL statement. `IBPerl::Statement` supports the following statements types:

SELECT	*CREATE*	*GRANT*
INSERT	*ALTER*	*REVOKE*
UPDATE	*DROP*	*SET GENERATOR*
DELETE	*DECLARE EXTERNAL FUNCTION*	
EXECUTE PROCEDURE	*DECLARE FILTER*	

There are other statements described in the InterBase *Language Reference* manual. Some of these are available in IBPerl through other methods, for example, `$tr->commit()`. If you try to create a SQL Statement "COMMIT", you receive an error.

Executing the query

After creating a query, you can execute it. The constructor for the Statement object automatically sends the query to the InterBase server, which *prepares* the query. The statement does not execute automatically when you prepare it, the statement executes only when your script invokes the `execute` method.

You can do this explicitly with the `execute` virtual method of the Statement object:

```
$st->execute();
```

You must invoke execute for most types of SQL statements. You also have the option for SELECT and EXECUTE PROCEDURE statement to allow IBPerl to execute the statement automatically upon the first call to `fetch`.

Note Earlier beta releases of IBPerl also had a virtual method `open`. This method is obsolete, and you are encouraged to use `execute` in its place. Current versions of IBPerl maintain backward compatibility, by implementing an `open` method that simply calls the `execute` method.

Fetching results

After you execute a query consisting of a SELECT or EXECUTE PROCEDURE statement, you can fetch the result data set with the `fetch` virtual method. This section describes several ways of retrieving rows of the result data set.

Fetching into a scalar

You can fetch a row and have IBPerl save the result in a scalar Perl variable. Pass the scalar by reference[5] to the `fetch` method.

```
$st->fetch( \$result );
print "$result\n";
```

If the query had multiple columns, the values of these columns are separated by a semicolon (;) character in the scalar row result. You can change the separator symbol. See **“Setting the field Separator” on page 31**.

Fetching into a list

You can fetch a row and have IBPerl save the result in a Perl list variable. Pass a list by reference to the `fetch` method. IBPerl stores each column of the query in a list element, in the order specified in the query.

```
$st->fetch( \@result );
print "$result[3]\n";
```

Fetching into a hash list

You can fetch a row and have IBPerl save the result in a Perl hash list variable. Pass a hash object by reference to the `fetch` method. IBPerl stores each column of the query in an element of the hash, indexed by the name of the column.

```
$st->fetch( \%result );
print "$result{LAST_NAME}\n";
```

If you gave a column alias using the AS clause, IBPerl uses that alias as the index of the hash alias. If neither a column name nor an alias exists (for instance, for an expression that you give no alias), IBPerl uses COLUMN*nn* as the index of the hash element, where *nn* is the ordinal position of the column in the select list.

Fetching all rows

The `fetch` method retrieves only one record at a time. You should write code for a loop to call the `fetch` method until you have retrieved all rows in the data set. The return value of `fetch` is zero (0) if there are more records in the result set, and 100 if there are no more records in the result set.

If the return value of `fetch` is less than zero, this indicates that an error occurred while trying to fetch. The `Error` property of the Statement object contains the text describing the error.

5. A reference to a Perl variable `$foo` is written `\$foo`.

Below is an example of a loop to fetch rows in a dataset, and detect if the loop ended due to an error:

```
while ( ($status = $st->fetch( \$result )) == 0) {
    print "$result\n";
}
die "$st->{Error}\n" if ($status < 0);
```

Fetching Blob values

Perl scalars can be very long, but Blobs can be even longer. Therefore, IBPerl enforces an arbitrary limit of *one million bytes* for a Blob scalar when fetching or storing Blob data. If this limit is too small for your applications, and you have enough memory on your client host to handle larger streams of Blob data, you can raise the limit by changing the definition of the macro *MAX_SAFE_BLOB_LENGTH* in IBPerl.h and recompiling IBPerl.

Using query options

The Statement constructor permits you to customize the query interface further with the following optional parameters: `TimestampFormat`, `DateFormat`, `TimeFormat`, and `Separator`.

Setting the date and time formats

When you retrieve a TIMESTAMP, DATE, or TIME datatype, IBPerl formats the data into a string scalar with a default formatting. You can customize the presentation of this data with a formatting string.

IBPerl uses the standard C library function *strftime()* to format date and time values. The default formats for InterBase datatypes is as follows:

Datatype	*Format string property*	*Default format string*
TIMESTAMP	*TimestampFormat*	*%c*
DATE	*DateFormat*	*%x*
TIME	*TimeFormat*	*%X*

Note DATE has a different meaning in dialect 1 and in InterBase releases prior to V6. IBPerl uses the default formatting shown above for the DATE datatype regardless of its contents.

You can specify your own format string for any or all of these datatypes. See **Appendix B, Format Strings on page 61** for a list of elements you can use in format strings.

Below is an example of specifying a format string in the `Statement` constructor:

```
$st = new IBPerl::Statement( Transaction => $tr,
    SQL => 'SELECT LAST_NAME, HIRE_DATE FROM EMPLOYEE',
    TimestampFormat => '%h-%d-%y');
```

See also **"Fetching dates as a list" on page 42**.

Synonyms for TimestampFormat: `TimeStampFormat`, `TimeStampformat`, `Timestampformat`.

Synonym for DateFormat: `Dateformat`.

Synonym for TimeFormat: `Timeformat`.

Setting the field Separator

When you fetch a row into a scalar variable, IBPerl separates the columns using a scalar specified by the `Separator` property of a `Statement` object. This scalar is a semicolon (;) by default.

You can specify a different string by using the `Separator` property when calling the `Statement` constructor:

```
$st = new IBPerl::Statement( Transaction => $tr,
    SQL => 'SELECT LAST_NAME, HIRE_DATE FROM EMPLOYEE',
    Separator => "\t");
```

Synonym for Separator: `Sep`.

SQL Dialect of a Statement

The SQL dialect of an individual `Statement` object defaults to that of the `Dialect` value for the `Connection` in which you execute the `Statement`. That is, the `Dialect` property of a parent `Connection` object is the default `Dialect` of a `Statement`. However, you can choose a different `Dialect` than that of the `Connection` by specifying it with the `Dialect` parameter to the `Statement` constructor. This `Dialect` is in effect only for that specific `Statement` object; subsequent `Statement` objects continue to use the `Dialect` of the `Connection` object by default.

```
$st = new IBPerl::Statement( Transaction => $tr,
    SQL => 'SELECT LAST_NAME, HIRE_DATE FROM EMPLOYEE
            WHERE LAST_NAME STARTING WITH "S"',
    Dialect => 1);
```

In this example, the query uses double-quotes to delimit a string constant, which works only in SQL dialect 1.

See **"SQL dialect of a Connection" on page 17** for more information on the significance of SQL dialects.

Closing a query

When you are done with a query, you should close it. This operation frees some memory that IBPerl allocates. Use the `close` virtual method of the `Statement` object. For example:

```
$st->close();
```

The Statement terminates, and releases any unfetched rows of the result set stored in the InterBase server. You cannot use this instance of the `Statement` object in subsequent calls to `fetch`. You must execute the `Statement` again to do that.

IBPerl automatically invokes the `close` method when your `Statement` object goes out of scope. For example:

```
{
    my $st = new IBPerl::Statement( . . . );
}
```

At the end of this code block, the `$st` object is garbage collected and its destructor calls `close`.

Using concurrent statements

You can execute a second query before you have finished fetching all results of the first query. This is useful because sometimes you need to execute two queries and interleave your fetching of their results. For example, the parameters of the second query might depend on the results from each row of the first query. Sometimes you can accomplish this by using joins or a correlated subquery, but your code might be more clear or flexible if you use two separate queries.

```
$st1 = new IBPerl::Statement( Transaction => $tr,
    SQL => 'SELECT DEPT_NO, DEPARTMENT FROM DEPARTMENT');
$st1->execute();
while ($st1->fetch(\@dept) == 0)
{
    print "DEPARTMENT $dept[1]:\n";
    my $st2 = new IBPerl::Statement( Transaction => $tr,
        SQL => "SELECT FIRST_NAME, LAST_NAME FROM EMPLOYEE WHERE DEPT_NO = $dept[0]");
    $st2->execute();
    while ($st2->fetch(\%emp) == 0)
    {
        print "$emp{LAST_NAME, $emp{FIRST_NAME}\n";
    }
    $st2->close();
}
$st1->close();
```

You can see that the second `Statement` yields potentially different results each time it is executed in this loop, because the value of the department number is different. The first `Statement` remains open and active while the second `Statement` executes and the script fetches all of its results.

Executing statements other than SELECT

As mentioned earlier, IBPerl supports a variety of InterBase SQL statement types, not only SELECT queries. This section describes use of these other statement types.

INSERT, UPDATE, and DELETE

These three data manipulation language statements are used differently from SELECT queries in that these statements never generate result data sets. In other respects, prepare and execute them in a similar manner as you would other queries:

```
$st = new IBPerl::Statement( Transaction => $tr,
    SQL => "INSERT INTO DEPARTMENT \
        (DEPT_NO, HEAD_DEPT, DEPARTMENT, LOCATION) \
        VALUES (998, 900, 'Pewter Solutions', 'Capitola')"
    );
$st->execute();
```

You must use the `execute` method to cause these types of statements to execute. Using the constructor to prepare these statements does not execute them.

EXECUTE PROCEDURE

Submitting a query to execute a stored procedure is most similar to submitting a query for SELECT statement. The similarity is in the fact that EXECUTE PROCEDURE is the only statement other than SELECT that might have a row of return data. A procedure that is declared to return one or more values generates one row of data, just as a SELECT statement might.

To retrieve this row, use the fetch method in the same manner as you would for a SELECT query. For example:

```
$st = new IBPerl::Statement( Transaction => $tr,
    SQL => 'EXECUTE PROCEDURE ALL_LANGS');
$st->fetch( \$results );
```

There is no need to write a loop to retrieve multiple rows from an `EXECUTE PROCEDURE` statement. These statements by definition never return more than one row of data. A “Select procedure” that returns multiple rows of data using the SUSPEND mechanism must be invoked using a SELECT query, not EXECUTE PROCEDURE.

Data definition language

IBPerl supports SQL statements in the category of data definition language. These statements include CREATE, ALTER, DROP, DECLARE EXTERNAL FUNCTION, DECLARE FILTER, and SET GENERATOR.

The exception is that you cannot submit a CREATE DATABASE statement. You need an active database connection and transaction before you can execute a SQL statement. Typically, when you create a database, you aren’t connected to another database. The create database functionality is handled by the `create` method of the `IBPerl::Connection` package (see **“Creating a new database” on page 15**).

Below is an example of code to use a DDL statement to create a table:

```
$ddl = <<_SQL_;
CREATE TABLE cross_rate (
    from_currency   VARCHAR(10) NOT NULL,
    to_currency     VARCHAR(10) NOT NULL,
    conv_rate       FLOAT NOT NULL,
    update_date     DATE,
    PRIMARY KEY (from_currency, to_currency)
)
_SQL_

$st = new IBPerl::Statement( Transaction => $tr, SQL => $ddl );
$st->execute();
```

Tip Notice the use of the “here document” (the << operator) to assign the value of the SQL statement to a Perl variable. This is a useful technique for lengthy string assignments, because it helps to improve the readability of your scripts. InterBase ignores newlines in SQL statements, so it is legal to construct the statements in this way.

Note Data definition language statements are not automatically committed to the database. You might be accustomed to the behavior of the InterBase **isql** tool, which transparently commits DDL statements by default. But this is not the behavior of IBPerl. You need to commit your transaction before other users can view the metadata modifications you execute.

Other statements

IBPerl also supports GRANT and REVOKE statements with the `IBPerl::Statement` package. These are not strictly data definition language statements, nor are they data manipulation language statements.

Building SQL statements

Often you need to construct a SQL statement at runtime from the value of variables or user input. You can assign any scalar expression to the `SQL` property of the `Statement` constructor. Perl offers several ways of building strings in expressions.

- You can use Perl variables in a string constant if you use double-quotes to delimit the string:

```
... SQL => "SELECT COL1 FROM TABLE1 ORDER BY $sort_key"
```

```
... SQL => "INSERT INTO TABLE1 VALUES ('$name', $age, '$city')"
```

Notice the use of single-quotes around string values, but not around column names or integer values.

If you use InterBase V6 delimited identifiers within double-quoted Perl strings, you need to escape the SQL delimiters. For example:

```
... SQL => "SELECT \"ADMIN\" FROM TABLE1"
```

- You can embed newlines in Perl string literals. The InterBase SQL parser ignores newlines, as long as they aren't within quoted string constants in a SQL expression.

```
... SQL => "SELECT A_FIELD
          FROM TABLE1"
```

- You can use the string concatenation operator (`.`) to assemble separate strings. This is useful if you need to include the result of a Perl expression:

```
... SQL => "UPDATE TABLE1 SET MEASUREMENT = " . getAstrologicalData() . ", M_TIME = 'NOW'"
```

- You can get input from users:

```
$user_input = <>;
... SQL => $user_input
```

In summary, any expression that results in a valid SQL statement in a string is okay to use as the value of the `SQL` property when creating a new `IBPerl::Statement`.

Note These techniques produce a string to use as the SQL statement *before* IBPerl sends it to the InterBase engine. This allows greater flexibility than you can have when using parameterized SQL queries. You can substitute parameters in prepared SQL queries only for *constant values* in SQL expressions. See **"Parameterized queries" on page 40**.

Handling statement errors

There are many ways an attempt for a script to prepare or execute a query can fail. For example, the SQL may have a syntax or semantic error. The statement might be attempting an operation for which the user has no privilege.

If something happens that prevents a query from succeeding, IBPerl returns to the calling script an invalid statement connection handle. An instance of the `Statement` object has properties `Handle` and `Error`. In case of an unsuccessful operation, the `Handle` property is set to a negative number, and the `Error` property contains a description of the cause of the failure.

For example, to check for and report a query failure:

```
$st = new IBPerl::Statement( ...parameters... );
if ($st->{Handle} < 0)
{
    carp "Failure to prepare SQL query. Error as follows:\n";
    carp "$st->{Error}\n";
    exit 1;
}
```

Even after you have prepared a statement successfully, you might have an error with a subsequent `execute` method. The `execute` method returns a negative number if it fails for any reason, and the error message is stored in the `Error` property.

```
if ($st->execute < 0)
{
    carp "Failure to execute SQL query. Error as follows:\n";
    carp "$st->{Error}\n";
}
```

Most IBPerl virtual methods simply return a negative number on failure. Only the constructor methods require you to examine the `Handle` property for an indicator of error.

IMPORTANT You should always include code to check for failure, report errors, and react accordingly. Code examples in this document might omit error-checking for brevity.

Statement class reference

Below is a table of all properties and methods of the `IBPerl::Statement` class.

Property/Method	*Description*
new( )	*Static constructor method to parse and prepare a SQL statement.*
execute( )	*Virtual method to execute a prepared SQL statement.*
fetch( )	*Virtual method to fetch one row of the result set generated when executing a statement of type* `SELECT`, `SELECT_FOR_UPD`, *or* `EXEC_PROCEDURE`.
update( )	*Virtual method to perform a positioned update for a query of type* `SELECT_FOR_UPD`.
delete( )	*Virtual method to perform a positioned delete for a query of type* `SELECT_FOR_UPD`.
close( )	*Virtual method to close an open result set; IBPerl invokes this method during the destructor of a Statement object.*
Transaction	*Scalar argument to the constructor, an active Transaction object; the SQL statement executes within the context of this transaction.*
SQL	*Scalar argument to the constructor; the SQL statement to execute.* *Synonyms: Statement, Stmt.*

TABLE 4.1 `IBPerl::Statement` *properties and methods*

Property/Method	*Description*
Separator	*Optional scalar argument to the constructor; the character or characters to use to separate fields when fetching a row as a single scalar.* *Synonym: Sep*
TimestampFormat	*Optional scalar parameter to the constructor; the formatting string for TIMESTAMP data values, conforming to the formatting string of the POSIX strftime( ) function.* *A special value of 'tm' causes* `fetch( )` *to return the data as a reference to an array similar to the return value of the Perl function localtime( ).* *Synonyms: TimeStampFormat, Timestampformat, TimeStampformat*
DateFormat	*Similar to TimestampFormat, but specifically for the InterBase DATE datatype.* *Synonym: Dateformat*
TimeFormat	*Similar to TimestampFormat, but specifically for the InterBase V6 TIME datatype.* *Synonym: Timeformat*
Dialect	*Defaults*
Handle	*Scalar used internally by the Statement class. You only use this to test if its value is less than zero. If so, the transaction failed to start.*
Error	*A scalar error message in the case when* `Handle` *is less than zero.*
Prepared	*Scalar property; nonzero if the SQL statement is prepared, and zero if the prepare fails.*
Executed	*Scalar property maintained by IBPerl; zero initially, increments each time you execute the prepared statement.*
Fetched	*Scalar property maintained by IBPerl; zero initially, increments each time you fetch a row from the result set of a query.*
Type	*Scalar property maintained by IBPerl; indicates the type of SQL statement entered; the value is one of* `SELECT`, `INSERT`, `UPDATE`, `DELETE`, `DDL`, `EXEC_PROCEDURE`, *or* `SELECT_FOR_UPD`.
Values	*Array property of a Statement when the type is* `SELECT`, `SELECT_FOR_UPD`, *or* `EXEC_PROCEDURE`*; after you call fetch( ), the array contains scalars corresponding to the fields fetched.*
Columns	*Array property of a Statement; the array contains field names corresponding to the values fetched in the* `Values` *array.*
Nulls	*Array property of a Statement; the array contains field names corresponding to the values fetched in the* `Values` *array.*

TABLE 4.1 `IBPerl::Statement` *properties and methods*

Property/Method	*Description*
Lengths	*Array property of a Statement; the array contains field names corresponding to the values fetched in the* `Values` *array.*
Datatypes	*Array property of a Statement; the array contains field names corresponding to the values fetched in the* `Values` *array.*
Scales	*Array property of a Statement; the array contains zero for most fields, but a scale value for DECIMAL or NUMERIC fields corresponding to the values fetched in the* `Values` *array, if these fields have nonzero scale.*
Parms	*IBPerl creates this array from the arguments you give to execute( ) for a parameterized query.*
Changes	*IBPerl creates this associative array (hash list) from the arguments to give to update( ) for a positioned update.*

TABLE 4.1 `IBPerl::Statement` *properties and methods*

CHAPTER

5
Advanced Queries

This chapter covers more methods to use with SQL statements with IBPerl.

Topics in this chapter are:

- **Parameterized queries**
- **Blob parameters**
- **Positioned updates & deletes**
- **Fetching dates as a list**
- **Accessing Statement internals**
- **Translating quotes for SQL**

Parameterized queries

InterBase allows you to prepare a SQL statement once, and execute it multiple times with different constant values specified for each execution.

Specifying parameters

IBPerl supports queries with parameters by preparing a statement containing parameter placeholders noted with the question mark character (**?**) when you create it with the `new` method. You can supply parameters to the `execute` method to specify values for the placeholders.

IMPORTANT You can use parameters only to replace *constant values* in SQL expressions. You cannot parameterize other syntax elements, such as names of fields or tables, or predicate operators. If you need that level of flexibility, you must build a new SQL query (see **"Building SQL statements" on page 34**).

```
$st = new IBPerl::Statement( Transaction => $tr,
    SQL => "INSERT INTO TABLE1 (A, B, C) VALUES (?, ?, ?)");
$st->execute('String', 327, $scalar);
```

IBPerl uses the arguments you give to the `execute` method in the same order that the **?** placeholders appear in the prepared SQL statement. There is no method to name the parameters, you have to provide them in order. You have to provide an argument for every parameter in the SQL query.

Tip You don't need to supply quotes as part of the string when you pass a character value as a parameter. Neither do you need quotes around the **?** placeholder.

Specifying parameters with NULL state

You can pass Perl's built-in `undef` when you want to use the SQL NULL expression instead of a real value for a parameter. For example:

```
$st = new IBPerl::Statement( Transaction => $tr,
    SQL => "INSERT INTO TABLE1 (A, B, C) VALUES (?, ?, ?)");
$st->execute(undef, 327, $scalar);
```

The result in this example is a record where the field A has a NULL state.

Invoking a prepared query multiple times

The best value of a parameterized query is the opportunity to execute a prepared query multiple times, specifying a different value each time. You can design the query to return different results based on different values that you provide as parameters. Using code based on the example in **"Using concurrent statements" on page 32**, here is an example of executing a parameterized query repeatedly:

```
$st1 = new IBPerl::Statement( Transaction => $tr,
    SQL => 'SELECT DEPT_NO, DEPARTMENT FROM DEPARTMENT');
$st1->execute();
$st2 = new IBPerl::Statement( Transaction => $tr,
    SQL => "SELECT FIRST_NAME, LAST_NAME FROM EMPLOYEE WHERE DEPT_NO = ?");
while ($st1->fetch(\@dept) == 0)
{
    print "DEPARTMENT $dept[1]:\n";
    $st2->execute( ($dept[0]) );
```

```
    while ($st2->fetch(\%emp) == 0)
    {
        print "$emp{LAST_NAME, $emp{FIRST_NAME}\n";
    }
    $st1->close();
}
$st1->close();
```

This code prepares the second `Statement` before the fetch loop of the first `Statement` begins. This is good for performance, because it removes the invariant step of preparing the parameterized query from the loop. Any relocation of invariant code to outside a loop is good for performance.

Blob parameters

Blobs are sometimes difficult to manipulate, because Blobs can be much larger than the longest SQL statement string that InterBase accepts. Therefore, it is not straightforward how to perform an operation such as inserting a literal string of bytes into a Blob field using SQL.

IBPerl supports use of Blobs in INSERT and UPDATE statements through the parameter mechanism described in the previous section (**"Parameterized queries" on page 40**). You can create a SQL statement with a parameter in place of a Blob field. For example:

```
$st = new IBPerl::Statement( Transaction => $tr,
    SQL => "UPDATE TABLE1 SET MyBlobField = ?");
$st->execute("really really really long string...");
```

The value of the parameter you pass to execute for a Blob can be a Perl literal string, or it can be any Perl expression that results in a scalar. The fact that Perl scalars can contain non-ASCII data allows you to load binary data into Blobs. For example:

```
open(FILEHANDLE, "amoeba.gif");
sysread(FILEHANDLE, $image, 1000000);
$st = new IBPerl::Statement( Transaction => $tr,
    SQL => "INSERT INTO TABLE1 (ImageData) VALUES (?)");
$st->execute( ($image) );
```

The limit of one million bytes applied to Blob input parameters. See **"Fetching Blob values" on page 30**.

Positioned updates & deletes

IBPerl supports the InterBase feature of updating or deleting the most recent row that the application fetched. Use the `update` or `delete` virtual methods of a `Statement` object.

Using update( )

The `update` method is useful when you need to change the value of fields but you cannot easily express the new value as a SQL expression in an UPDATE statement.

The `update` method takes a hash list, the keys of which are names of fields to change, and the values of which are scalar expressions for each new value of a field.

You can use the `update` method only for SELECT statements that include the FOR UPDATE clause. IBPerl supports only positioned updates on queries that are based on a single table and include no aggregate functions.

Below is a simple example, to use positioned updates to make sure that names are capitalized properly.

```
$st = new IBPerl::Statement(Transaction => $tr,
    SQL => "SELECT * FROM EMPLOYEE FOR UPDATE");
while ($st->fetch(\%row) == 0)
{
    $st->update( LAST_NAME => correct_case($row{LAST_NAME}) );
}
```

You must commit the transaction in which you execute this statement for these changes to be permanent and visible to other transactions.

Using delete( )

The `delete` method is useful when you need to conditionally delete records but you cannot easily express the condition as a SQL expression in a DELETE statement.

You can use the `delete` method only for SELECT statements that include the FOR UPDATE clause. IBPerl supports only positioned deletes on queries that are based on a single table and include no aggregate functions.

Below is a simple example, to use a positioned delete to remove records based on user's choice:

```
$st = new IBPerl::Statement(Transaction => $tr,
    SQL => "SELECT * FROM EMPLOYEE FOR UPDATE");
while ($st->fetch(\$row) == 0)
{
    print "Delete row \n \"$row\"\n(yes/no)? ";
    $choice = <STDIN>;
    $st->delete if ($choice =~ /YES/io);
}
```

You must commit the transaction in which you execute this statement for these changes to be permanent and visible to other transactions.

Fetching dates as a list

IBPerl provides a flexible mechanism for specifying a string format for DATE, TIME, and TIMESTAMP datatypes (see **"Setting the date and time formats" on page 30**). However, sometimes you need to write a script that manipulates the elements of a TIMESTAMP individually.

IBPerl recognizes a special value for the format properties of "tm" (or "TM"). This is intended to be mnemonic for `struct tm`, which POSIX defines as a decomposed date/time representation. The values returned for `DATE`, `TIME`, or `TIMESTAMP` database fields in the result set for that Statement object are not scalar strings; the values are instead references to Perl lists, similar in content to the list returned by the Perl function *localtime( )*. That is, the list consists of the following sequence of elements:

0. Seconds 0...59
1. Minutes 0...59
2. Hour 0...23
3. Day of the month 1...31
4. Number of the month 0...11, with 0 indicating January

5. Number of the year since 1900; that is, the year 2001 returns a value 101; you should recreate the true year value by adding a numeric value 1900 to this element, *not* by appending the string '19' value to the element
6. Day of the week 0...6, with 0 indicating Sunday
7. Day of the year 1...366
8. A true value if the date value occurs during daylight savings time for the locale, false otherwise

Below is an example of using a `TimestampFormat` of 'tm' to get access to individual timestamp elements.

```
$st = new IBPerl::Statement( Transaction => $tr,
    SQL => 'SELECT LAST_NAME, HIRE_DATE FROM EMPLOYEE',
    TimestampFormat => 'tm');
while ($st->fetch(\%row) == 0)
{
    @weekdays = ('Sunday', 'Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday');
    @month = ('Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun',
        'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec');
    @times = @$row{DATEFIELD};
    if ($times[5] >= 100)
    {
        $times[5] += 1900;
        $timestamp_string = "$weekdays[$times[6]], $month[$times[4]] $times[3], $times[5]";
    } else {
        $timestamp_string = 'long long ago';
    }
    print "Employee $row{LAST_NAME} was hired $timestamp_string.\n";
}
```

Note the use of `@$`. The `$` operator treats the following identifier as the name of a scalar, which in this case is a reference object[6]. Then the `@` operator dereferences the reference object to reach the list that it points to. This might be a new kind of Perl expression for you, but it’s no more complicated than dereferencing a pointer to pointer in C or C++.

Accessing Statement internals

The `Statement` object contains several interesting members that you can use for specialized purposes. You can use IBPerl according to the methods described earlier in this document, but you can also access the “internal[7]” members for additional flexibility.

Statement properties

- The `$st->{Transaction}` member is a reference to the `Transaction` object in the context of which you started the `Statement`.
- The `$st->{Type}` scalar member indicates the type of SQL statement. When you invoke the Statement constructor, IBPerl parses and prepares the SQL statement. At this time, it determines the statement type and stores it in the `Type` member. The possible statement types are: SELECT, INSERT, UPDATE, DELETE, DDL, EXEC_PROCEDURE, or SELECT_FOR_UPD.

6. Reference objects are themselves scalars in Perl, regardless of whether the object to which they refer is a scalar or a list.
7. I use the term “internal” loosely. Perl packages have no private elements, as are supported in some object-oriented implementations.

- The `$st->{Plan}` scalar member is set after you prepare a SELECT or SELECT_FOR_UPD statement. This is the query plan that InterBase uses when optimizing the query.

  Note InterBase V4 on Linux has a bug that it crashes when it tries to record the `Plan` for a query of a SELECT procedure. You can avoid this bug by refraining from use of such queries with IBPerl, commenting out the code in **IBPerl.xs** that stores the `Plan`, or upgrading to the latest version of InterBase, which does not have this bug.

- The `$st->{Prepared}` scalar member is true if IBPerl successfully prepared the SQL statement in the constructor. `Prepared` is false otherwise, which is generally an error.

Dataset arrays

When you call the `fetch` virtual method, IBPerl stores many of the scalar attributes of the result set into a series of array members of the `Statement` object. These are relevant only for statement types SELECT, SELECT_FOR_UPD, and EXEC_PROCEDURE. All the arrays are in the order of columns in the SELECT list of the query. These arrays are as follows:

- `@$st->{Values}` is a list of scalar values.
- `@$st->{Columns}` is a list of column names. If your query declared column aliases with the AS clause, the alias is used in preference to the natural column name.
- `@$st->{Nulls}` is a list of NULL indicators. An element of the `Nulls` list is true if the corresponding field in the currently fetched row has a NULL state.
- `@$st->{Lengths}` is a list of the length in bytes of each field.
- `@$st->{Datatypes}` is a list of the SQL datatype names of each field. The `Datatype` values are: BLOB, VARCHAR, CHAR, DOUBLE PRECISION, FLOAT, INTEGER, SMALLINT, TIMESTAMP, D_FLOAT (VMS only), ARRAY, QUAD, TIME (InterBase V6 only), DATE, or INT64 (InterBase V6 only).
- `@$st->{Scales}` is a list of the SQL scales of each field. This is zero for all datatypes except DECIMAL(x, y) and NUMERIC(x, y) where y is nonzero.

Counting rows affected by an operation

The Statement constructor sets the scalar member `$st->{Count}` as it prepares an INSERT, UPDATE, or DELETE statement. This indicates how many rows are affected by the corresponding operation. For instance, a normal INSERT operation with literal values affects exactly one (1) row. A DELETE statement without a WHERE clause to restrict rows affects as many rows as are in the table.

Counting executions and fetches

You might find it useful to know how many rows you have fetched from a query. You could keep a counter variable in your Perl script and increment it each time you call `fetch`. IBPerl does this for you, and stores the result in the `Statement` scalar member `$st->{Fetched}`. Below is an example of how to use this member:

```
$st = new IBPerl::Statement( . . . );
while ($st->fetch() == 0) { . . . }
print "This query returned $st->{Fetched} rows.\n";
```

Likewise, IBPerl counts how many times you execute a prepared query, and stores the result in the `Statement` scalar member `$st->{Executed}`.

Translating quotes for SQL

How InterBase interprets quotes

Placing quote characters inside quoted strings in SQL requires special treatment of these symbols. If a string literal is delimited by single quotes and the string contains a single quote, the parser cannot distinguish the embedded literal quote symbol from the character that delimits the end of the string literal:

```
. . . WHERE LAST_NAME = 'O'Reilly' . . .
```

The SQL standard solution is to replace a single quote with two single quote characters.

```
. . . WHERE LAST_NAME = 'O''Reilly' . . .
```

This is interpreted in the SQL engine as a single literal quote, and that is what the RDBMS stores. When you fetch the value later, you get the string as it was intended, with one single quote symbol. You must remember to duplicate single-quote symbols when you use literal strings in SQL expressions. This is not too difficult.

It's slightly more difficult to remember to perform this quote alteration when supplying parameters for parameterized queries (see **"Parameterized queries" on page 40**), or when supplying values for positioned updates (see **"Positioned updates & deletes" on page 41**). Especially when your code generates these values procedurally and passes Perl scalars to the `execute` or `update` methods, it is inconvenient to filter strings for single-quote characters. For this reason, IBPerl does it for you.

How IBPerl translates quotes

In the `execute` and `update` virtual methods of the `Statement` package, IBPerl substitutes any single-quote character in a string with a pair of single-quote characters. So you can safely execute Perl code like the following, and know that IBPerl fixes the embedded quote character in your input parameter:

```
. . . WHERE LAST_NAME = ? . . .
$val = "O'Reilly";
$st->execute( ($val) );
```

The exception is when the single-quote character *already* has another single-quote character adjacent to it. In this case, IBPerl recognizes it and assumes that you have performed the filtering for quotes yourself, because a pair of single-quote characters rarely occurs naturally in text. So the following Perl code works too:

```
. . . WHERE LAST_NAME = ? . . .
$val = "O''Reilly";
$st->execute( ($val) );
```

Caveat: quotes and binary parameters

This works well for text values, but it's possible that a scalar input parameter containing binary data could contain a byte with the value 39, which is the ASCII value of the single quote symbol. IBPerl would duplicate this byte, resulting in probable alteration of your binary scalar. In a future version of IBPerl, a mechanism of the `Statement` object will allow you to inhibit the quote-editing feature of IBPerl temporarily while you input parameters with binary content.

Until then, if you have the need for binary input parameters, you should edit IBPerl.pm and comment out this line (on or near line 438) that performs this string editing in the `execute` method:

```
map { s/(^|[^'])'([^']|$)/$1''$2/g if $_ } @parms if (@parms)
```

CHAPTER

6 Examples

This chapter consists of complete examples of relatively complex IBPerl script that do useful tasks.

Examples in this chapter are:

- **Copying data from one database to another**
- **Joining queries from two databases**
- **Output an InterBase dataset as an HTML table**

Copying data from one database to another

It's often the case that you need to copy data from one database to another. InterBase doesn't provide a simple way to do this operation. There are some tools available such as the DataPump Expert in Delphi. But if you prefer a Perl solution, try the script below. This script is also in the examples/tablecopy.pl file bundled with the IBPerl download.

The script is intended as an example, not as a final solution to data copying. The script assumes that you are copying tables with exactly the same fields: the fields must have the same names, positions, and datatype definitions. It also works only if any referenced values already exist in master tables in the database.

```
#!/usr/bin/perl -w
#
# tablecopy.pl -- Copy contents of a table from one
#   database to another.
#
# Usage:
#   tablecopy.pl database1:table database2[:table]
#
# Bill Karwin, May 2000
# ------------------------------------------------------------------------

use strict;
use IBPerl;

my ($db1_name, $db2_name, $db1, $db2, $tr1, $tr2, $st1, $st2);
my ($table1_name, $table2_name, $sql1, $sql2, $err);

($db1_name, $table1_name) = split(':', $ARGV[1] || 'employee.gdb:COUNTRY');
($db2_name, $table2_name) = split(':', $ARGV[2] || 'empty.gdb:COUNTRY');
$table2_name ||= $table1_name;

# ------------------------------------------------------------------------
# Open a connection and transaction to the first
# database, and execute the query.
$db1 = new IBPerl::Connection( Path => $db1_name );
die "$0: Connection error: $db1->{Error}\n" if ($db1->{Handle} < 0);

$tr1 = new IBPerl::Transaction( Database => $db1 );
die "$0: Transaction error: $tr1->{Error}\n" if ($tr1->{Handle} < 0);

$sql1 = "SELECT * FROM $table1_name";
$st1 = new IBPerl::Statement( Transaction => $tr1, SQL => $sql1 );
die "$0: Prepare error: $st1->{Error}\n" if ($st1->{Handle} < 0);

if ($st1->execute() < 0)
{
   die "$0: Execute error: $st1->{Error}\n";
}

# Make sure we have some records to copy
$err = $st1->fetch();
die "$0: Fetch error: $st1->{Error}\n" if ($err < 0);
if ($err != 0)
{
   print "No records to copy.\n";
   exit 0;
}
```

```
# ------------------------------------------------------------------------
# Now that we have one row representing what we should
# import to the second database, we can build an
# appropriate INSERT statement.
$sql2 = "INSERT INTO $table2_name VALUES (" .
    join(', ',('?') x scalar(@$st1->{Values}))
    . ")";

# ------------------------------------------------------------------------
# Open a connection and transaction to the second
# database, and prepare the INSERT statement.

$db2 = new IBPerl::Connection( Path => $db2_name );
die "$0: Connection error: $db2->{Error}\n" if ($db2->{Handle} < 0);

$tr2 = new IBPerl::Transaction( Database => $db2 );
die "$0: Transaction error: $tr2->{Error}\n" if ($tr2->{Handle} < 0);

$st2 = new IBPerl::Statement( Transaction => $tr2, SQL => $sql2 );
die "$0: Prepare error: $st2->{Error}\n" if ($st2->{Handle} < 0);

# ------------------------------------------------------------------------
# Loop on fetched rows from the first database,
# executing the prepared INSERT to the second
# database with different values on each iteration.
do
{
    if ($st2->execute( @$st1->{Values} ) < 0)
    {
        die "$0: Execute error: $st2->{Error}\n";
    }
} until ($st1->fetch() != 0);

# ------------------------------------------------------------------------
# Clean up and exit.
print "$0: copied $st1->{Fetched} rows.\n";

$tr2->commit();
$db2->disconnect();

$tr1->commit();
$db1->disconnect();

exit 0;
```

Joining queries from two databases

One of the most frequently missed features of InterBase is the ability to join datasets from separate databases. If you need this feature, below is an example of fetching two datasets and performing an inner join in a client script. This script is also in the **examples/xjoin.pl** file that is bundled with the IBPerl download.

It is a nontrivial task for the InterBase software to perform an efficient join efficiently even in the context of a single database. InterBase does not support cross-database joins, for reasons of efficiency. The general case would require the client and server to transfer all the data from both tables across the network, and matching the values in the client. This is a very expensive and wasteful operation. However, sometimes one needs to do this operation despite the cost.

The script is intended as an example, not as a final solution to cross-database joins. It does not, for instance, perform outer joins, aggregates, or sorted result sets by columns other than the primary key.

```
#!/usr/bin/perl -w
#
# xjoin.pl -- Join datasets from two separate databases.
#
# Usage:
#   xjoin.pl database1:table.primary_col database2[:table[.referencing_col]]
#
# Bill Karwin, May 2000
# ----------------------------------------------------------------------

use strict;
use IBPerl;

my ($prim_db_name, $ref_db_name);
my ($prim_table_name, $ref_table_name, $temp);
my ($primary_key, $foreign_key);

($prim_db_name, $temp) =
    split(':', ($ARGV[0] || 'employee.gdb:DEPARTMENT.DEPT_NO'));
($prim_table_name, $primary_key) = split(/\./o, $temp);

($ref_db_name, $temp) =
    split(':', ($ARGV[1] || $ARGV[0] || 'employee.gdb:EMPLOYEE.DEPT_NO'));
($ref_table_name, $foreign_key) = split(/\./o, $temp);
$ref_table_name ||= $prim_table_name;
$foreign_key ||= $primary_key;

# Make sure key names are uppercase, since that's
# how IBPerl returns them in the hash list
$primary_key =~ s/[a-z]/\U$&/g;
$foreign_key =~ s/[a-z]/\U$&/g;

# ----------------------------------------------------------------------
# Open a connection and transaction to the first
# database, and execute the query.

my ($db1, $tr1, $sql1, $st1);

$db1 = new IBPerl::Connection( Path => $prim_db_name );
die "$0: Connection error: $db1->{Error}\n" if ($db1->{Handle} < 0);

$tr1 = new IBPerl::Transaction( Database => $db1 );
die "$0: Transaction error: $tr1->{Error}\n" if ($tr1->{Handle} < 0);

$sql1 = "SELECT * FROM $prim_table_name ORDER BY $primary_key";
```

```
$st1 = new IBPerl::Statement( Transaction => $tr1, SQL => $sql1 );
die "$0: Prepare error: $st1->{Error}\n" if ($st1->{Handle} < 0);

if ($st1->execute() < 0)
{
    die "$0: Execute error: $st1->{Error}\n";
}

# ------------------------------------------------------------------------
# Open a connection and transaction to the second
# database, and execute the query.

my ($db2, $tr2, $sql2, $st2);

$db2 = new IBPerl::Connection( Path => $ref_db_name );
die "$0: Connection error: $db2->{Error}\n" if ($db2->{Handle} < 0);

$tr2 = new IBPerl::Transaction( Database => $db2 );
die "$0: Transaction error: $tr2->{Error}\n" if ($tr2->{Handle} < 0);

$sql2 = "SELECT * FROM $ref_table_name ORDER BY $foreign_key";
$st2 = new IBPerl::Statement( Transaction => $tr2, SQL => $sql2 );
die "$0: Prepare error: $st2->{Error}\n" if ($st2->{Handle} < 0);

if ($st2->execute() < 0)
{
    die "$0: Execute error: $st2->{Error}\n";
}

# ------------------------------------------------------------------------
# Loop on fetched rows from the first database

my (%prim_row, %ref_row);

PRIMARY: while (1)
{
   my ($err, $col);

   $err = $st1->fetch(\%prim_row);
   die "$0: Fetch error: $st1->{Error}\n" if ($err < 0);
   last if ($err != 0);

   foreach $col (keys %prim_row)
   {
      printf("%-31s %s\n", $col, $prim_row{$col} || '[null]');
    }
   print "\n";

   # ------------------------------------------------------------------
   # Find a matching row in the referencing table

   while (($ref_row{$foreign_key} || '') lt ($prim_row{$primary_key} || ''))
   {
      $err = $st2->fetch(\%ref_row);
      die "$0: Fetch error: $st2->{Error}\n" if ($err < 0);
      last PRIMARY if ($err != 0);
   }
```

```
    # ------------------------------------------------------------------
    # Now that we've found it, continue output of detail
    # records until we stop matching

    while (($ref_row{$foreign_key} || '') eq ($prim_row{$primary_key} || ''))
    {
        foreach $col (keys %ref_row)
        {
            printf(" %-30s %s\n", $col, $ref_row{$col} || '[null]');
        }
        print "\n";

        $err = $st2->fetch(\%ref_row);
        die "$0: Fetch error: $st2->{Error}\n" if ($err < 0);
        last PRIMARY if ($err != 0);
    }
}

# ----------------------------------------------------------------------
# Clean up and exit

$tr2->commit();
$db2->disconnect();

$tr1->commit();
$db1->disconnect();

exit 0;
```

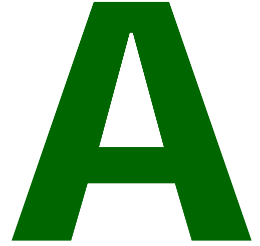
Output an InterBase dataset as an HTML table

Perl programmers frequently use this language for a handy web application scripting language. Therefore, it is a common need to be able to output data in an HTML table format. The example below shows a CGI script that queries any table in a database, and formats the output in HTML.

This script is found in the examples/table.cgi file in the IBPerl distribution.

```
#!/usr/bin/perl -w
#
# table.cgi -- Output an InterBase SQL query as an HTML table.
#
# See http://stein.cshl.org/WWW/software/CGI/cgi_docs.html
# for documentation on CGI.pm, which is used extensively in
# this script.
#
# Bill Karwin, May 2000
# ------------------------------------------------------------

use strict;
use CGI qw(:standard);
use IBPerl;

# Unbuffer the output, to help CGI work well
select STDERR; $|++;
select STDOUT; $|++;

my $request = new CGI;

my $database = $request->param('DATABASE') || 'employee.gdb';
my $table = $request->param('TABLE') || 'employee';
my $order = $request->param('ORDER');

# ------------------------------------------------------------
# Start HTML output

print header(), "\n";
print start_html(-title=>'InterBase Table output', -author=>'Bill_Karwin'), "\n";

# ------------------------------------------------------------
# Connect to database and open query

my $db = new IBPerl::Connection( Path => $database,
   User => 'nobody', Password => 'xxxxxxxx' );
die "$0: Connection error: $db->{Error}\n" if ($db->{Handle} < 0);

my $tr = new IBPerl::Transaction( Database => $db );
die "$0: Transaction error: $tr->{Error}\n" if ($tr->{Handle} < 0);

my $st = new IBPerl::Statement( Transaction => $tr,
   SQL => ("SELECT * FROM $table " . ($order? "ORDER BY $order": '')));

my @record;

# Check for statement error
if ($st->{Handle} < 0)
{
   print h1($st->{Error});
}
```

```
# Fetch the first row to see if the dataset is empty
elsif ($st->fetch(\@record) != 0)
{
    print h1("Table $table has no records.");
}

# -------------------------------------------------------------
# Here's where it gets interesting, formatting the data
# into an HTML table

else
{
    # The heading row contains hyperlinks back to this script,
    my $url = $request->url(-absolute=>1) . "?DATABASE=$database\&TABLE=$table";
    my @headings = map { a( {href => "$url\&ORDER=$_"}, $_) } @{$st->{Columns}};
    my @rows = th( {-bgcolor=>'#FFCC00'}, \@headings);

    # ---------------------------------------------------
    # Start loop over the rest of the dataset.

    do
    {
        # Alternating row colors
        my @rowcolor = ('Beige', 'White');

        # Table cells are blank unless there's something in them.
        @record = map { $_ || ' ' } @record;

        # Construct one row of the table,
        # using CGI.pm's distributive method

        push(@rows, td( { -bgcolor => $rowcolor[ $st->{Fetched} % 2 ] }, \@record));
    } until ($st->fetch(\@record) != 0);

    # -----------------------------------------------------
    # Here's where the complete HTML table is output,
    # using CGI.pm's distributive method

    print table(
        {-border=>1, cellpadding=>5 },
        caption( strong("Content of table $table") ),

        Tr( {-align=>'LEFT', -valign=>'TOP' }, \@rows));
}

# -------------------------------------------------------------
# Clean up and exit.

print end_html(), "\n";

close(STDOUT);

exit 0;
```



License terms

The Artistic License

Preamble

The intent of this document is to state the conditions under which a Package may be copied, such that the Copyright Holder maintains some semblance of artistic control over the development of the package, while giving the users of the package the right to use and distribute the Package in a more-or-less customary fashion, plus the right to make reasonable modifications.

Definitions

“Package” refers to the collection of files distributed by the Copyright Holder, and derivatives of that collection of files created through textual modification.

“Standard Version” refers to such a Package if it has not been modified, or has been modified in accordance with the wishes of the Copyright Holder as specified below.

“Copyright Holder” is whoever is named in the copyright or copyrights for the package.

“You” is you, if you’re thinking about copying or distributing this Package.

“Reasonable copying fee” is whatever you can justify on the basis of media cost, duplication charges, time of people involved, and so on. (You will not be required to justify it to the Copyright Holder, but only to the computing community at large as a market that must bear the fee.)

“Freely Available” means that no fee is charged for the item itself, though there may be fees involved in handling the item. It also means that recipients of the item may redistribute it under the same conditions they received it.

1. You may make and give away verbatim copies of the source form of the Standard Version of this Package without restriction, provided that you duplicate all of the original copyright notices and associated disclaimers.
2. You may apply bug fixes, portability fixes and other modifications derived from the Public Domain or from the Copyright Holder. A Package modified in such a way shall still be considered the Standard Version.
3. You may otherwise modify your copy of this Package in any way, provided that you insert a prominent notice in each changed file stating how and when you changed that file, and provided that you do at least ONE of the following:
   - a. place your modifications in the Public Domain or otherwise make them Freely Available, such as by posting said modifications to Usenet or an equivalent medium, or placing the modifications on a major archive site such as uunet.uu.net, or by allowing the Copyright Holder to include your modifications in the Standard Version of the Package.
   - b. use the modified Package only within your corporation or organization.
   - c. rename any non-standard executables so the names do not conflict with standard executables, which must also be provided, and provide a separate manual page for each non-standard executable that clearly documents how it differs from the Standard Version.
   - d. make other distribution arrangements with the Copyright Holder.
4. You may distribute the programs of this Package in object code or executable form, provided that you do at least ONE of the following:

- a. distribute a Standard Version of the executables and library files, together with instructions (in the manual page or equivalent) on where to get the Standard Version.
- b. accompany the distribution with the machine-readable source of the Package with your modifications.
- c. give non-standard executables non-standard names, and clearly document the differences in manual pages (or equivalent), together with instructions on where to get the Standard Version.
- d. make other distribution arrangements with the Copyright Holder.

5. You may charge a reasonable copying fee for any distribution of this Package. You may charge any fee you choose for support of this Package. You may not charge a fee for this Package itself. However, you may distribute this Package in aggregate with other (possibly commercial) programs as part of a larger (possibly commercial) software distribution provided that you do not advertise this Package as a product of your own. You may embed this Package's interpreter within an executable of yours (by linking); this shall be construed as a mere form of aggregation, provided that the complete Standard Version of the interpreter is so embedded.
6. The scripts and library files supplied as input to or produced as output from the programs of this Package do not automatically fall under the copyright of this Package, but belong to whomever generated them, and may be sold commercially, and may be aggregated with this Package. If such scripts or library files are aggregated with this Package via the so-called "undump" or "unexec" methods of producing a binary executable image, then distribution of such an image shall neither be construed as a distribution of this Package nor shall it fall under the restrictions of Paragraphs 3 and 4, provided that you do not represent such an executable image as a Standard Version of this Package.
7. C subroutines (or comparably compiled subroutines in other languages) supplied by you and linked into this Package in order to emulate subroutines and variables of the language defined by this Package shall not be considered part of this Package, but are the equivalent of input as in Paragraph 6, provided these subroutines do not change the language in any way that would cause it to fail the regression tests for the language.
8. Aggregation of this Package with a commercial distribution is always permitted provided that the use of this Package is embedded; that is, when no overt attempt is made to make this Package's interfaces visible to the end user of the commercial distribution. Such use shall not be construed as a distribution of this Package.
9. The name of the Copyright Holder may not be used to endorse or promote products derived from this software without specific prior written permission.
10. THIS PACKAGE IS PROVIDED "AS IS" AND WITHOUT ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTIBILITY AND FITNESS FOR A PARTICULAR PURPOSE.

The End

GNU General Public License

Version 2, June 1991

Copyright (C) 1989, 1991 Free Software Foundation, Inc.
59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software--to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

GNU GENERAL PUBLIC LICENSE
TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

   Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

   You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

   - a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.
   - b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.
   - c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

   These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

   Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

   In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

   - a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
   - b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
   - c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.
5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.
6. 6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.
7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

   If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

   It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

   This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.
8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.
9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

   Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.
10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.
12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

<one line to give the program's name and a brief idea of what it does.>
Copyright (C) 19yy <name of author>

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

Gnomovision version 69, Copyright (C) 19yy name of author
Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type `show w'.
This is free software, and you are welcome to redistribute it under certain conditions; type `show c' for details.

The hypothetical commands `show w' and `show c' should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than `show w' and `show c'; they could even be mouse-clicks or menu items--whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

Yoyodyne, Inc., hereby disclaims all copyright interest in the program `Gnomovision' (which makes passes at compilers) written by James Hacker.

<signature of Ty Coon>, 1 April 1989
Ty Coon, President of Vice

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

APPENDIX

B
Format Strings

Date/Time format elements

This appendix contains a summary of the format string elements that you can use. You can use these elements in the `TimestampFormat`, `DateFormat`, and `TimeFormat` properties of `IBPerl::Statement` objects. See **“Setting the date and time formats” on page 30**.

This information is excerpted from the man page of *strftime(3)*:

Specifier	*Description*
%a	*The abbreviated weekday name according to the current locale.*
%A	*The full weekday name according to the current locale.*
%b	*The abbreviated month name according to the current locale.*
%B	*The full month name according to the current locale.*
%c	*The preferred date and time representation for the current locale.*
%C	*The century number (year/100) as a 2-digit integer.*
%d	*The day of the month as a decimal number (range 01 to 31).*
%D	*American style date; equivalent to %m/%d/%y.*
%e	*Like %d, the day of the month as a decimal number, but a leading zero is replaced by a space.*
%E	*Modifier as in %Ec, %Ex, etc.; use alternative locale-dependent format.*
%G	*The ISO 8601 year with century as a decimal number. The 4-digit year corresponding to the ISO week number (see %V). This has the same format and value as %y, except that if the ISO week number belongs to the previous or next year, that year is used instead.*
%g	*Like %G, but without century, i.e., with a 2-digit year (00-99).*
%h	*Equivalent to %b.*
%H	*The hour as a decimal number using a 24-hour clock (range 00 to 23).*
%I	*The hour as a decimal number using a 12-hour clock (range 01 to 12).*
%j	*The day of the year as a decimal number (range 001 to 366).*
%k	*The hour (24-hour clock) as a decimal number (range 0 to 23); single digits are preceded by a blank. (See also %H.)*

Specifier	Description
%l	The hour (12-hour clock) as a decimal number (range 1 to 12); single digits are preceded by a blank. (See also %I.)
%m	The month as a decimal number (range 01 to 12).
%M	The minute as a decimal number (range 00 to 59).
%n	A newline character.
%O	Modifier as in %Od, %OM, etc.; use alternative numeric format.
%p	Either `AM' or `PM' according to the given time value, or the corresponding strings for the current locale. Noon is treated as `pm' and midnight as `am'.
%P	Like %p but in lowercase: `am' or `pm' or a corresponding string for the current locale.
%r	The time in a.m. or p.m. notation. In the POSIX locale this is equivalent to `%I:%M:%S %p'.
%R	The time in 24-hour notation (%H:%M). For a version including the seconds, see %T below.
%s	The number of seconds since the Epoch, i.e., since 1970-01-01 00:00:00 UTC.
%S	The second as a decimal number (range 00 to 61).
%t	A tab character.
%T	The time in 24-hour notation (%H:%M:%S).
%u	The day of the week as a decimal, range 1 to 7, Monday being 1. See also %w.
%U	The week number of the current year as a decimal number, range 00 to 53, starting with the first Sunday as the first day of week 01. See also %V and %W.
%V	The ISO 8601:1988 week number of the current year as a decimal number, range 01 to 53, where week 1 is the first week that has at least 4 days in the current year, and with Monday as the first day of the week. See also %U and %W.
%w	The day of the week as a decimal, range 0 to 6, Sunday being 0. See also %u.
%W	The week number of the current year as a decimal number, range 00 to 53, starting with the first Monday as the first day of week 01.
%x	The preferred date representation for the current locale without the time.
%X	The preferred time representation for the current locale without the date.
%y	The year as a decimal number without a century (range 00 to 99).
%Y	The year as a decimal number including the century.
%z	The time-zone as hour offset from GMT. Required to emit RFC822-conformant dates (using "%a, %d %b %Y %H:%M:%S %z").
%Z	The time zone or name or abbreviation.
%+	The date and time in date(1) format.
%%	A literal '%' character.

Refer to the online man page for *strftime(3)* for more details.